

**The Free SQL Book:**  
**A Tutorial Introduction to SQL**

**Tim Martyn**

**Copyright 2022**

## Dedications

My previous books were dedicated to my parents (Irene and Nathaniel Martyn), my wife (Janet Martyn), and my children (Julie Martyn and Jessica Brown). I dedicate this book to my grandparents, brother, and grandchildren.

Grandparents: Sydney Martyn (1870 - 1963)  
Mary Martyn (1876 - 1912)  
Philip Cahill (1877 - 1938)  
Nellie Cahill (1882 - 1936)

Brother: Stephen Martyn (1947)

Grandchildren: Hanna Brown (2014)  
Johnny Geryk (2015)  
Evan Brown (2017)  
Josephine Geryk (2017)  
Jacqueline Geryk (2019)

Sadly, three of my grandparents died before I was born. I only remember my Grandpa Sydney Martyn. He was a wonderful and especially kind man. I hope that my grandchildren will remember me with the same respect and affection that I hold for him.

## **This Book is *Almost* Free**

This book is free for almost all readers. Three restrictions on the "freeness" of this book are noted below.

1. This book is protected by a copyright. All copyright laws should be respected.
2. Any for-profit organization that uses this book within a professional training class should send a modest donation to one of the non-profit environmental organizations identified below. This donation should be \$5.00 per student per class day. For example, a 2-day class with 10 students implies a \$100.00 (2x10x5) donation. A brief statement acknowledging this donation should be emailed to: [freesqlbook@gmail.com](mailto:freesqlbook@gmail.com)
3. Currently, this book and related files must be downloaded from the [www.freesqlbook.com](http://www.freesqlbook.com) website. The author has not given permission to any "free-textbook" website to store this book for download purposes.

If, after reading this book, you conclude that it was worth at least \$5.00, you are encouraged to donate that amount (or more) to any of the following non-profit environmental organizations. Please include a note with your donation stating: "This donation was encouraged by the author of The Free SQL Book."

### Environmental Organizations

- Nature Conservancy ([www.nature.org](http://www.nature.org))
- World Wildlife Organization ([www.worldwildlife.org](http://www.worldwildlife.org))
- National Audubon Society ([www.audubon.org](http://www.audubon.org))

Massachusetts residents may prefer to donate to the:

- Kestrel Land Trust ([www.kestreltrust.org](http://www.kestreltrust.org))
- Trustees of Reservations ([www.thetrustees.org](http://www.thetrustees.org))
- Massachusetts Audubon Society ([www.massaudubon.org](http://www.massaudubon.org))

Finally, take a moment to explore some of the above websites. Because our planet needs more than a little help, please consider donating a few extra dollars to join one of these organizations.

## Preface

**Objective:** This book is a *tutorial* introduction to SQL. The teaching method is learn-by-example using many sample queries and exercises (with an answer book).

Like other SQL books, this book presents the basic syntax and logic of SQL statements. Unlike many other SQL books, this book highlights important “know-your-data” and “know-your-logic” considerations. This book also includes optional appendices that address other database topics related to SQL.

The SQL statements in this book have been tested on SQL Server, DB2, and ORACLE. Where relevant, commentary identifies differences in the syntax and behavior across these systems.

**Target Audience:** This book is written for the following categories of readers. (The term “user” collectively refers to all readers.)

SQL Rookie: Anyone wanting a “getting started” tutorial on SQL.

SQL Super-User: This person already knows some SQL and desires to become proficient in the language. Frequently, a super-user does not work in an IT Department. Instead, a super-user could be an accountant, engineer, actuary, bookie, or scientist who spends a considerable amount of time retrieving and manipulating data that is stored in a relational database.

Application Developer: This person has a working knowledge of some procedural programming language (e.g., Java, C++, COBOL, PHP, Python). Application developers usually write “embedded SQL” where SQL statements are embedded within an application program. An application developer must first learn SQL (as presented in this book) before she can embed SQL statements within a program. This book does not discuss the embedding process per se because the details, while not difficult, differ among various database systems and programming languages.

College Student: This person may be attending an academic course that covers a wide spectrum of database topics, including SQL. Typically, classroom time constraints imply that the instructor can offer no more than a few lectures on SQL. The tutorial orientation of this book allows such students to learn SQL via independent reading and homework exercises.

**Appendices:** This book includes two categories of appendices, Book Appendices and Chapter Appendices.

**Book Appendices:** These appendices are located at the end of the book. *The first book appendix* (Create Sample Tables for The Free SQL Book) *is relevant for all readers.* The second book appendix (Obtain Access to some Relational Database System) should help those readers who do not have access to any relational database system. The remaining book appendices are optional reading.

**Chapter Appendices:** All chapter appendices are *optional* reading. These appendices are located at the end of some (but not all) chapters. Each chapter appendix offers incidental by-the-commentary on topics related to the SQL statements introduced in the corresponding chapter. These appendices address Relational Database Theory, SQL Efficiency, and Database Analysis and Design.

**Suggestion for Reading this Book:** This book frequently presents a sample query on two pages as illustrated by Sample Query 1.1. The even-numbered page presents: (i) the query objective, (ii) the SQL statement that satisfies this objective, and (iii) the result table. The odd-numbered page presents commentary on syntax and logic.



If the reader has a large computer screen that can conveniently display two pages, this layout should facilitate reading the narrative about the sample query without flipping between pages. (This layout also facilitates reading a physical copy of this book. But hopefully very few readers will print this book. Save a tree!)

### **Suggestions for Academic Faculty & Professional Trainers**

Some thoughts about using this book can be found at the [www.freesqlbook.com](http://www.freesqlbook.com) website.

### **Feedback from Readers**

A method for offering feedback to the author, especially error identification, can be found at the [www.freesqlbook.com](http://www.freesqlbook.com) website.

## Where's Tim Hartley?

Tim Hartley and I have long shared a personal and SQL friendship. We coauthored multiple SQL articles and early SQL books on DB2 and ORACLE (published by McGraw Hill). So, why did I go it alone with this book? First, I would like to emphasize that, when writing this book, there were many occasions when I wished Tim were a coauthor. However, I didn't want to take advantage of his good nature. Imagine a friend approaches you with the following proposal.

Let's write another SQL book. It will be considerably longer than our previous books, and the selling price will be \$0.00! Furthermore, I expect to dillydally with my writing obligations, and I may arbitrarily "call it quits" anytime along the way.

I would be embarrassed to offer this proposal to anyone. Also, I have not asked Tim to read this book because I know how I would feel if I were asked to read a very long book (950+ pages) where I would learn absolutely nothing. Finally, Tim is still in working mode, whereas I am joyfully retired.

Note: I did indeed dillydally when writing this book. I took approximately 10 years to complete this task because, although I enjoyed playing with SQL code, I preferred to play with my young grandchildren, dig holes in my garden, and attempt to play some golf.

## Acknowledgements

In my previous SQL books, I lightheartedly acknowledged my favorite watering-hole and a few of its resident philosophers. The following acknowledgements are more sincere.

In my early working life, my mindset could be described by the old adage: Work is the curse of the drinking class. I worked to earn some honest money for the same reasons that most of us do. While I enjoyed most of my early work experiences, I never committed to any organization. I was too skeptical (and perhaps a little too cynical) to buy into any kind of organizational rah-rah. However, my attitude changed after I joined the Computer Science Department at The Hartford Graduate Center (HGC) in Hartford, Connecticut.

I attribute this change to my fellow faculty members in the Computer Science Department. No superficial rah-rah stuff here. These folks walked the walk. They always demonstrated a sincere commitment to our department's mission, which I summarize as: Provide a first-rate Master's Degree in Computer Science for the working professional. Beyond a strong commitment to their academic disciplines, these individuals were always friendly and helpful toward other faculty, staff, and students. In the academic world, where egomania and petty back-biting are not unknown, I never encountered any such chicken-shxt within our department. I acknowledge the following individuals by name because we were together during what I consider to be our department's golden age.

Roger Brown  
Michael Danchak  
Lynn DeNoia  
Heidi Ellis  
Tim Hartley  
Jim McKim  
Houman Younessi

These individuals earned my deepest respect. It is my very good fortune that we became friends.

I also acknowledge other members of the HGC family. These include two wonderful administrative women who made things happen, Judy Rohan and Flo Josephs. Both were kind enough to kick my butt when it (frequently) needed kicking. Also, I would like to acknowledge fellow faculty members in the Engineering Department and School of Management, and two former HGC Presidents, Homer Babbage and Worth Loomis.

Finally, I note that HGC students received their degrees from Rensselaer Polytechnic Institute (RPI) in Troy, NY. (In 1996, the Hartford Graduate Center was renamed to Rensselaer at Hartford.) I would like to acknowledge those members of the RRI Computer Science Faculty who graciously helped our small department fulfill its mission.



---

# Table of Contents

<u>Part I</u>	<u>The SELECT Statement</u>	<u>Page</u>
Chapter 0	Read this Chapter!	13
Chapter 1	Getting Started: The SELECT Statement	23
Chapter 2	Sorting the Result Table: ORDER BY	49
Chapter 3	Prohibiting Duplicate Rows: DISTINCT	73
Chapter 4	Boolean Connectors: AND-OR-NOT	83
Chapter 5	IN and BETWEEN	129
Chapter 6	Pattern Matching: LIKE	139
Chapter 7	Arithmetic Expressions	167
<u>Part II</u>	<u>Built-in Functions &amp; Null Values</u>	
Chapter 8	Aggregate Functions	187
Chapter 9	GROUP BY and HAVING Clauses	197
Chapter 9.5	Grouping by Multiple Columns	217
Chapter 10	Individual Functions	243
Chapter 10.5	Processing DATE Values	255
Chapter 11	Null Values	283
<u>Part III</u>	<u>Data Definition &amp; Data Manipulation</u>	
Chapter 12	Preview Sample Sessions	313
Chapter 13	CREATE TABLE Statement	331
Chapter 14	CREATE INDEX Statement	359
Chapter 15	INSERT, UPDATE, & DELETE Statements	371
<u>Part IV</u>	<u>Join Operations</u>	
Chapter 16	Inner-Join: Getting Started	383
Chapter 17	More about Inner-Join	417
Chapter 18	Multi-Table Inner-Joins	445
Chapter 19	Outer-Join: Getting Started	515
Chapter 20	Multi-Table Outer-Joins	547
Chapter 20.5	Mixing Inner-Join & Outer-Join Operations	581

<u>Part V</u>	<u>Set Operations &amp; CASE Expressions</u>	<u>Page</u>
Chapter 21	Set Operations: UNION, INTERSECT, and EXCEPT	607
Chapter 22	CASE Expressions	637
<u>Part VI</u>	<u>Sub-SELECTs</u>	
Chapter 23	"Regular" Sub-SELECTs	667
Chapter 24	Sub-SELECTs in DML	701
Chapter 25	Correlated Sub-SELECTs	715
Chapter 26	Inline Views	743
Chapter 27	WITH-Clause: Common Table Expressions	759
Chapter 28	CREATE VIEW Statement	775
<u>Part VII</u>	<u>Special Topics</u>	
Chapter 29	Transaction Processing: COMMIT & ROLLBACK	805
Chapter 30	Recursive Queries	837
<u>Book Appendices</u>		
I.	Create Sample Tables for The Free SQL Book	933
II.	Obtain Access to a Relational Database System	943
III.	Summary of Chapter Appendices	958
IV.	Post-Relational Database Systems	965
V.	Abbreviated Bibliography	969

This page is intentionally blank.

# PART I

## The SELECT Statement

This first part of this book introduces the most popular SQL statement, the SELECT statement, which is used to retrieve data from a relational database. If you are a SQL rookie, Chapter 0 is an especially important chapter.

Chapter 0 presents a very brief introduction to relational database concepts. It describes the PRESERVE table, one of the FREESQL sample tables, and previews six sample queries that retrieve data from this table. [Note: All sample queries in the following seven chapters reference the PRESERVE table.]

Chapter 1 introduces sample queries that retrieve data from the PRESERVE table. Attention is directed towards the column data-types specified in this table.

Chapter 2 introduces the ORDER BY clause that displays a result table in a specified row sequence.

Chapter 3 introduces the DISTINCT keyword that removes duplicate rows from a result table.

Chapter 4 introduces the Boolean Connectors (AND, OR, and NOT) that facilitate the specification of more complex row selection criteria.

Chapter 5 introduces the IN and BETWEEN keywords that, in some circumstances, support the abbreviation of row selection criteria.

Chapter 6 introduces the LIKE keyword that supports searching character-string columns for specified character-string patterns.

Chapter 7 introduces arithmetic expressions that are used to perform basic addition, subtraction, multiplication, and division.

This page is intentionally blank.

# Read this Chapter!

Preliminary Comment: This is a simple but very important chapter. Read this chapter as you would read a history book, for conceptual purposes only. The last few pages will direct you towards “hands-on” execution of SQL statements.

SQL is a popular computer language that is used to create, retrieve, and modify data in a relational database, such as ORACLE, SQL Server, DB2, and MYSQL. Relational base concepts will be introduced on the following page. For the moment we begin with a few preliminary observations about SQL.

- **SQL is easy.**
- **However, a careless user can easily produce an “almost correct” (i.e., incorrect) result.**

More explicitly:

- **Again, SQL is easy.**
- **However: Knowing your *data* can be a challenge.**
- **And: Knowing your *logic* can be a challenge.**

This book, like all other SQL books, presents the basic syntax and semantics of SQL statements. However, unlike many other SQL books, this book explicitly addresses issues pertaining to *knowing-your-data* and *knowing-your-logic*.

## Relational Databases

All relational database systems store data in tables. The following Figure 0.1 illustrates a table called PRESERVE. This table describes 14 nature preserves that have been acquired and maintained by the Nature Conservancy. (You are encouraged to visit [www.nature.org](http://www.nature.org))

PNO	PNAME	STATE	ACRES	FEE
5	HASSAYAMPA RIVER	AZ	660	3.00
3	DANCING PRAIRIE	MT	680	0.00
7	MULESHOE RANCH	AZ	49120	0.00
40	SOUTH FORK MADISON	MT	121	0.00
14	MCELWAIN-OLSEN	MA	66	0.00
13	TATKON	MA	40	0.00
9	DAVID H. SMITH	MA	830	0.00
11	MIACOMET MOORS	MA	4	0.00
12	MOUNT PLANTAIN	MA	730	0.00
1	COMERTOWN PRAIRIE	MT	1130	0.00
2	PINE BUTTE SWAMP	MT	15000	0.00
80	RAMSEY CANYON	AZ	380	3.00
10	HOFT FARM	MA	90	0.00
6	PAPAGONIA-SONOITA CREEK	AZ	1200	3.00

**Figure 0.1: PRESERVE Table**

The PRESERVE table, like every other table in this book, contains rows and columns. Each column has a name (PNO, PNAME, STATE, ACRES, and FEE). This table has 14 rows, where each row describes one nature preserve. (Most real-world database tables contain hundreds, thousands, millions, or billions of rows.)

## The SELECT Statement

This book will focus on the SELECT statement which is the most popular SQL statement. You will execute a SELECT statement whenever you want to retrieve and display data from a database table. As an example, we preview the SELECT statement for Sample Query 1.1 in Chapter 1.

```
SELECT *  
FROM PRESERVE
```

This statement will display all data stored in the PRESERVE table.

## Data-Types

Examination of the PRESERVE table shows that some columns contain numeric data and other columns contain character-string data.

The numeric columns are: PNO, ACRES, and FEE

The character-string columns are: PNAME and STATE

At a more detail level, you must learn about some different types of numbers and different types of character-strings. More precisely, you must learn the *data-type* of each column.

Regarding PRESERVE's numeric columns, note that:

- PNO is an INTEGER data-type.
- ACRES is an INTEGER data-type.
- FEE is a DECIMAL data-type.

You should already understand the difference between an integer value (e.g., 77, 0, -14) and a decimal value (e.g., 77.1, 0.0, -14.9). However, this difference will not become relevant until Chapter 7 where sample queries will illustrate numerical calculations using values from the ACRES and FEE columns.

Regarding PRESERVES's character-string columns, note that:

- STATE is a CHAR (fixed-length) character-string.
- PNAME is a VARCHAR (variable-length) character-string.

The difference between a fixed-length character-string (CHAR) and a variable-length character-string (VARCHAR) cannot be deduced by simply observing sample data. For example, eyeballing the data in the STATE column in Figure 0.1 does not allow you to definitively conclude that STATE is a fixed-length character-string. Likewise, eyeballing the data in the PNAME column does not allow you to definitively conclude that PNAME is a variable-length character-string. Chapter 6 will describe the differences between fixed-length and variable-length character-strings.

If you intend to query a table, you must learn the name and data-type of its columns. Frequently, you can read documentation about this information. You can also learn this information by examining the CREATE TABLE statement that created the table, as illustrated on the following page.



## Know-Your-Data (Knowing the PRESERVE Table)

Before you can retrieve data from a table, someone, usually a database administrator (DBA), must create the table and insert rows into it. The DBA executes a CREATE TABLE statement to create a table. Because the DBA is not a member of our target audience, the CREATE TABLE statement will not be presented in great detail. However, our *know-your-data philosophy* requires a brief introduction to the CREATE TABLE statement (Figure 0.2) that was used to create the PRESERVE table shown in Figure 0.1.

```
CREATE TABLE PRESERVE
(PNO          INTEGER          NOT NULL UNIQUE,
 PNAME       VARCHAR (25)     NOT NULL,
 STATE       CHAR (2)         NOT NULL,
 ACRES       INTEGER          NOT NULL,
 FEE         DECIMAL (5,2)    NOT NULL)
```

**Figure 0.2: CREATE TABLE Statement**

A CREATE TABLE statement assigns a name to the table; and it assigns a name to each column and specifies the column's data-type. The data-type specification also determines the maximum length of a column. Referencing the above figure, we note that:

- PNO and ACRES contain INTEGER values. The largest INTEGER value is slightly larger than 2,147,000,000.
- FEE contains DECIMAL values. Each FEE value has a maximum of five digits with two places after the decimal point. Hence the largest FEE value cannot exceed 999.99.
- STATE contains fixed-length (CHAR) character-strings. Each STATE value must have exactly two characters.
- PNAME contains variable-length (VARCHAR) character-strings. The length of each PNAME value can vary, but its length cannot exceed 25 characters.

Also observe that:

- The PNO column is designated as UNIQUE. Hence, the PNO column cannot contain any duplicate values.
- Each column is specified as NOT NULL. This means that no column value can be an "unknown" value.

[For the moment, there is no need to illustrate the INSERT statements that inserted the 14 rows into the PRESERVE table.]

## Preview: Sample Queries from Chapter 1

Without any explanation, we preview all sample queries that will be presented in Chapter 1. We expect that most readers can rely on their intuition to deduce "what's going on here."

**Sample Query 1.1:** Display all data in the PRESERVE table.

```
SELECT *  
FROM PRESERVE
```

PNO	PNAME	STATE	ACRES	FEE
5	HASSAYAMPA RIVER	AZ	660	3.00
3	DANCING PRAIRIE	MT	680	0.00
7	MULESHOE RANCH	AZ	49120	0.00
40	SOUTH FORK MADISON	MT	121	0.00
14	MCELWAIN-OLSEN	MA	66	0.00
13	TATKON	MA	40	0.00
9	DAVID H. SMITH	MA	830	0.00
11	MIACOMET MOORS	MA	4	0.00
12	MOUNT PLANTAIN	MA	730	0.00
1	COMERTOWN PRAIRIE	MT	1130	0.00
2	PINE BUTTE SWAMP	MT	15000	0.00
80	RAMSEY CANYON	AZ	380	3.00
10	HOFT FARM	MA	90	0.00
6	PAPAGONIA-SONOITA CREEK	AZ	1200	3.00

**Sample Query 1.2:** Display all information about any nature preserve that has an admission fee of \$3.00. (More precisely, display just those rows where the FEE value equals 3.00. Display all columns in these rows.)

```
SELECT *  
FROM PRESERVE  
WHERE FEE = 3.00
```

PNO	PNAME	STATE	ACRES	FEE
5	HASSAYAMPA RIVER	AZ	660	3.00
80	RAMSEY CANYON	AZ	380	3.00
6	PAPAGONIA-SONOITA CREEK	AZ	1200	3.00

**Sample Query 1.3a:** Display all information about any nature preserve that is located in Arizona. (I.e., Display just those rows where the STATE value is AZ.)

```
SELECT *  
  
FROM PRESERVE  
  
WHERE STATE = 'AZ'
```

<u>PNO</u>	<u>PNAME</u>	<u>STATE</u>	<u>ACRES</u>	<u>FEE</u>
5	HASSAYAMPA RIVER	AZ	660	3.00
7	MULESHOE RANCH	AZ	49120	0.00
80	RAMSEY CANYON	AZ	380	3.00
6	PAPAGONIA-SONOITA CREEK	AZ	1200	3.00

**Sample Query 1.3b:** Display all information about the Ramsey Canyon nature preserve.

```
SELECT *  
  
FROM PRESERVE  
  
WHERE PNAME = 'RAMSEY CANYON'
```

<u>PNO</u>	<u>PNAME</u>	<u>STATE</u>	<u>ACRES</u>	<u>FEE</u>
80	RAMSEY CANYON	AZ	380	3.00

**Observation:** Sample Query 1.3a references STATE, a CHAR column, whereas Sample Query 1.3b references PNAME, a VARCHAR column.

```
WHERE STATE = 'AZ'
```

```
WHERE PNAME = 'RAMSEY CANYON'
```

The similarity of the above WHERE-clauses illustrates that, in many (but not all) circumstances, fixed-length character-strings and variable-length character-strings are treated in the same manner.

**Sample Query 1.4:** For every row in PRESERVE, display its PNAME, ACRES, and STATE values (in that left-to-right column sequence).

```
SELECT PNAME, ACRES, STATE
FROM PRESERVE
```

<u>PNAME</u>	<u>ACRES</u>	<u>STATE</u>
HASSAYAMPA RIVER	660	AZ
DANCING PRAIRIE	680	MT
MULESHOE RANCH	49120	AZ
SOUTH FORK MADISON	121	MT
MCELWAIN-OLSEN	66	MA
TATKON	40	MA
DAVID H. SMITH	830	MA
MIACOMET MOORS	4	MA
MOUNT PLANTAIN	730	MA
COMERTOWN PRAIRIE	1130	MT
PINE BUTTE SWAMP	15000	MT
RAMSEY CANYON	380	AZ
HOFT FARM	90	MA
PAPAGONIA-SONOITA CREEK	1200	AZ

**Sample Query 1.5:** Select the PNAME and ACRES values for every nature preserve that is located in Arizona.

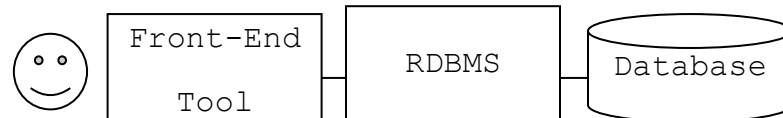
```
SELECT PNAME, ACRES
FROM PRESERVE
WHERE STATE = 'AZ'
```

<u>PNAME</u>	<u>ACRES</u>
HASSAYAMPA RIVER	660
MULESHOE RANCH	49120
RAMSEY CANYON	380
PAPAGONIA-SONOITA CREEK	1200

Chapter 1 will present details about the preceding sample queries. Many readers will find these sample queries to be self-evident. In fact, some SELECT statements are so obvious that you may be tempted to fly through Chapter 1 and ignore the related narrative. This might be OK for an experienced professional. However, most readers should not yield to this temptation. The narrative will highlight important know-your-data concepts.

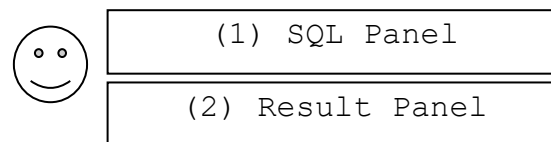
## System Architecture: Front-End Tool & Database Engine

A "Front-End Tool" is a software product that presents screen images (pages) similar to those shown in the preceding sample queries. Here, the front-end tool provides a (presumably) friendly interface to a Relational Database Management System (RDBMS) such as ORACLE, DB2, or SQL Server. The front-end tool and RDBMS constitute the high-level system architecture as illustrated in the following Figure 0.3.



**Figure 0.3: System Architecture**

**Front-End Tool:** After starting the front-end tool, it should present a SQL-Page similar to the following Figure 0.4.



**Figure 0.4: SQL-Page**

This generic SQL-Page displays two panels: (1) an SQL Panel where you enter your SELECT statement, and (2) a Result Panel which displays the result table. (This generic SQL-Page only illustrates part of what a user will see. Also, with some front-end tools, the user may have to interact with one or more preliminary pages before the SQL-Page is displayed.)

**RDBMS:** Most users never *directly* interact with the "back-end" RDBMS which is "under the hood." The RDBMS will: (i) accept a SELECT statement from a front-end tool, (ii) process this statement to retrieve the desired data from the database which resides on a physical disk, and (iii) return the result to the front-end tool for display.

The following Chapter 1 says more about the SQL-Page and expands upon Figure 0.4. Book Appendix-II presents more details about the front-end tool and its relationship to the database engine. (Both the front-end tool and RDBMS may be obtained from the same software vendor (e.g., Microsoft). Alternatively, a front-end tool and RDBMS may be obtained from different software vendors.)

## **Hands-On SQL: Learn by Doing**

An intellectually gifted reader might be able to learn SQL by reading this entire book without executing a SQL statement on a computer. However, learning SQL is really a learn-by-doing experience. Therefore, you are *strongly* encouraged to try some of the many exercises presented throughout this book. (An Answer Book is available at the [www.freesqlbook.com](http://www.freesqlbook.com) website.) Of course, this presumes that you can: (1) gain access to some RDBMS, and (2) create the sample tables referenced in this book's sample queries and exercises.

### **1. Access some RDBMS**

Most readers already have access to some RDBMS. However, some readers do not yet have access to any such system. Gaining access to an RDBMS may be easy, or it may require some effort. Book-Appendix-II describes three general methods to satisfy this objective. This appendix will also describe how to gain access to some front-end tool that can interact with the RDBMS.

### **2. Create the FREESQL sample tables in your system**

After gaining access to some front-end tool and RDBMS, you will want to create the FREESQL sample tables. This is not difficult. The process for creating these tables is described in Book-Appendix-I.

## ”What should I do Next?”

It depends upon your personal situation.

If you **already have access** to an RDBMS:

1. Read Book-Appendix-I and create the FREESQL tables.
2. Start reading this book.

If you **do not** have access to an RDBMS:

1. Read Book-Appendix-II to gain access to some relational database system (and front-end tool).
2. Read Book-Appendix-I, and create the FREESQL sample tables.
3. Start reading this book.

[If you think it could take you some time to gain access to an RDBMS, do not let this logistical problem cause a significant delay in your learning experience. Most readers will be able to understand the material presented in Chapters 1-5 without executing any SQL statements. After you gain access to an RDBMS and create the FREESQL sample tables, return to Chapters 1-5 and try its exercises.]

# Getting Started: The SELECT Statement

The SELECT statement is used to retrieve data from one or more tables. In this chapter, you will learn how to retrieve data from a single table, the PRESERVE table. You will do this by examining six sample queries and their SELECT statement solutions.

**Suggestion:** If this is your first exposure to SQL, it may be helpful to manually type and execute some of the SELECT statements presented in the sample queries. The results should match the results shown in this book. Do not cut-and-paste. For some reason, manually typing the SELECT statements reinforces the learning process.

**Again - Read the narrative for the sample queries:** As indicated in the previous chapter, some SELECT statements are so obvious that you may be tempted to ignore the related narrative. This might be appropriate for experienced users. However, most readers should not yield to this temptation. The narrative will highlight important *know-your-data* concepts.

**Exercises:** You are encouraged to try (some of) the exercises. This chapter's exercises are not difficult and should enhance your learning experience. (The last three pages in the previous chapter direct you towards gaining access to a relational database system and creating the sample tables used in this book.)



## Display an Entire Table

The first sample query asks you to display the entire PRESERVE table (i.e., display all rows and all columns). Executing the following simple SELECT statement will satisfy this objective.

**Sample Query 1.1:** Display all data in the PRESERVE table.

```
SELECT *  
FROM PRESERVE
```

PNO	PNAME	STATE	ACRES	FEE
5	HASSAYAMPA RIVER	AZ	660	3.00
3	DANCING PRAIRIE	MT	680	0.00
7	MULESHOE RANCH	AZ	49120	0.00
40	SOUTH FORK MADISON	MT	121	0.00
14	MCELWAIN-OLSEN	MA	66	0.00
13	TATKON	MA	40	0.00
9	DAVID H. SMITH	MA	830	0.00
11	MIACOMET MOORS	MA	4	0.00
12	MOUNT PLANTAIN	MA	730	0.00
1	COMERTOWN PRAIRIE	MT	1130	0.00
2	PINE BUTTE SWAMP	MT	15000	0.00
80	RAMSEY CANYON	AZ	380	3.00
10	HOFT FARM	MA	90	0.00
6	PAPAGONIA-SONOITA CREEK	AZ	1200	3.00

**Typing SQL Statements:** You can enter the above SELECT statement on a single line as shown below.

```
SELECT * FROM PRESERVE
```

**Syntax:** "SELECT" and "FROM" are reserved words, sometimes called keywords. Each reserved word has a specific meaning. Your SQL reference manual will contain a comprehensive description of all your system's reserved words.

**FROM:** The FROM-clause identifies the table that you want to query. Your system will have many tables. Each table will have a unique name. This name immediately follows the FROM keyword. The FROM-clause must follow the SELECT-clause.

**SELECT:** The SELECT-clause identifies the columns that you want to select. The asterisk (\*) following SELECT tells the system to "select all columns." The left-to-right column sequence in the result corresponds to the default column sequence determined by the order in which the columns were specified in the CREATE TABLE statement. (Review Figure 0.2). Sample Query 1.4 will illustrate how to select and display some subset of columns.

**Logic:** This SELECT statement retrieves all rows because it does not contain a WHERE-clause. The following Sample Query 1.2 introduces the WHERE-clause that is used to retrieve a subset rows from a table. We emphasize that *the absence of a WHERE-clause asks the system to retrieve all rows.*

**Row Sequence:** Observe that the rows in this result table are not displayed in any particular row sequence. This is because:

- In principle, database tables do not have any predefined row sequence, and
- This SELECT statement does not contain an ORDER BY clause. (The ORDER BY clause will be introduced in Chapter 2.)

In general, *you should never assume that a result table has any specific row sequence unless the SELECT statement has specified an ORDER BY clause.*

[\* If you execute this statement on your system, you might see a sorted result because your system happened to produce an "incidentally" sorted result. No harm here. We discuss incidentally sorted results in the following Chapter 2.]

**Termination Character:** Your system may require the specification of a termination character. Many systems use the semicolon (;) for this purpose. For example, you might be required to terminate this SELECT statement as shown below.

```
SELECT *  
FROM PRESERVE;
```

We emphasize that the semicolon is not part of the SELECT statement. The semicolon only indicates the end of the statement. Most of this book's SQL statements will not specify a semicolon.

**Formatting the Result Table:** This book does not focus on report formatting because SQL does not (directly) support this functionality. However, your front-end tool may offer some method to format a result table. Such formatting includes report headings, column spacing, the inclusion of dollar signs and commas in numeric values, etc.

**Articulating Query Objectives:** This sample query, like many future sample queries, asks you to: "Display ...". To be more precise, this query objective should state: "Retrieve and display ...". We make this distinction because we may want to execute a SELECT statement that retrieves data, but does not display the retrieved data. For example, in Chapter 8, some SELECT statements will retrieve and summarize data, but only display the summary total (without displaying the retrieved data that was summarized).

## WHERE-Clause: Numeric Comparison

With the exception of a very small table, you will rarely want to display all rows in a table. You will usually specify a WHERE-clause to identify some subset of rows that you want to display.

**Sample Query 1.2:** Display all information about any nature preserve that has an admission fee of \$3.00. (I.e., Display just those rows where the FEE value equals 3.00.) Display all columns in these rows.

```
SELECT *
FROM PRESERVE
WHERE FEE = 3.00
```

PNO	PNAME	STATE	ACRES	FEE
5	HASSAYAMPA RIVER	AZ	660	3.00
80	RAMSEY CANYON	AZ	380	3.00
6	PAPAGONIA-SONOITA CREEK	AZ	1200	3.00

**Syntax:** If specified, the WHERE-clause must follow the FROM-clause. The syntax of the WHERE-clause is:

WHERE condition

A condition identifies the rows to be retrieved. (Sometimes a condition is called a "predicate.") In this example, the condition is "FEE = 3.00". Only rows that match the condition will be retrieved. A condition can specify any of the following comparison operators:

= "Equals"  
<> "Not equals"  
< "Less than"  
> "Greater than"  
<= "Less than or equal to"  
>= "Greater than or equal to"

**Punctuation:** A numeric value may include a minus sign (-). If desired, you could code something like: WHERE FEE = -3.00. However, no other punctuation is permitted. *In particular, numeric values should not contain dollar signs and commas.*

**Logic - Know-Your-Data:** Recall that FEE is defined as a DECIMAL data type. Hence, "FEE = 3.00" is an example of a numeric comparison. This means the system compares on a mathematical basis rather than on a character-by-character basis. The following WHERE-clauses are logically equivalent and will retrieve the same rows.

```
WHERE FEE = 3
```

```
WHERE FEE = 3.0
```

```
WHERE FEE = 3.0000
```

Because FEE is a DECIMAL, it is strongly recommended (but not required) that the comparison value (3.00) be specified as a decimal value as illustrated in this SELECT statement.

**Incidental Sort:** Again, on some systems, you might observe that the result table is incidentally sorted. Likewise, for all of this chapter's sample queries (because their SELECT statements do not specify an ORDER BY clause).

**Simple Conditions:** The WHERE-clause (WHERE FEE = 3.00) specifies a simple condition because it does not specify any Boolean operators (e.g., AND, OR, NOT) that are used to formulate compound-conditions. Chapter 4 will discuss compound-conditions.

**Terminology - Restrict:** This SELECT statement retrieves all columns from a subset of rows. We will use the term "restrict" to refer to this kind of operation. (Appendix 1B will offer a more formal description of the restrict operation.)

### Exercises:

The following three exercises (1A, 1B, and 1C) reference the PRESERVE table.

- 1A. Display the row with a preserve number (PNO) of 5.
- 1B. Display all information about any nature preserve that does not charge an admission fee (i.e., the admission fee is zero).
- 1C. Display all information about any nature preserve that is larger than 1,000 acres.

The following Exercise 1D references another table.

- 1D. The sample database contains a table called EMPLOYEE. Assume you know nothing about this table except that it has a small number of rows. Display all data in this table.

## WHERE-Clause: Fixed-Length Character-String Comparison

The next sample query specifies a condition that references STATE, a fixed-length character-string (CHAR) column.

**Sample Query 1.3a:** Display all information about any nature preserve that is located in Arizona. (I.e., Display just those rows where the STATE value is AZ.)

```
SELECT *
FROM PRESERVE
WHERE STATE = 'AZ'
```

PNO	PNAME	STATE	ACRES	FEE
5	HASSAYAMPA RIVER	AZ	660	3.00
7	MULESHOE RANCH	AZ	49120	0.00
80	RAMSEY CANYON	AZ	380	3.00
6	PAPAGONIA-SONOITA CREEK	AZ	1200	3.00

**Syntax:** The character-string value must be enclosed within apostrophes ('AZ'). A rookie user might *incorrectly* specify a double quote (") instead of an apostrophe which will produce an error.

**Case Sensitivity:** All characters in the STATE column are stored in uppercase. Hence, this WHERE-condition must specify uppercase characters. The following WHERE-clauses would produce a "no hit" (no rows returned) result.

```
WHERE STATE = 'az'      → Error
```

```
WHERE STATE = 'Az'     → Error
```

```
WHERE STATE = 'aZ'     → Error
```

Case sensitivity can be a nuisance if a column contains both uppercase and lowercase characters. Chapter 10 will address this potential problem.

Case sensitivity (usually) does not apply to reserved words, table-names, and column-names. The following statement is ugly, but (*on most systems*) it is correct, and it will produce the same result as the above SELECT statement.

```
seLEct *
fRom PrEServe
WherE StAtE = 'AZ'
```

**Logic - Comparing CHAR Values:** The following comments apply when comparing a fixed-length (CHAR) column, such as the STATE column, to a character-string value.

Character-string comparison compares on a character-by-character basis. A special circumstance occurs when comparing two strings that do not have the same length. Before starting the compare operation, the system effectively pads the end of the shorter string with blanks such that both strings have the same length. This behavior is called "*blank-padded comparison semantics*." (Note: The system considers a blank to be a real character.)

For example, assume that Ireland joined the United States and was assigned a STATE code of I, a character-string of length 1. Because the data-type of the STATE column is CHAR (2), the I value would be stored as two characters, 'I '. *Observe the trailing blank.*

To display Irish nature preserves, your SELECT statement could correctly specify the following WHERE-clause.

```
WHERE STATE = 'I'
```

When comparing this single-character value ('I') to the two-character STATE column, the system pads the 'I' such that it becomes 'I '. Hence, the above WHERE-clause is evaluated as:

```
WHERE STATE = 'I '
```

*You could explicitly code the trailing blank in the above WHERE-clause. However, this is unconventional and we generally discourage coding trailing blanks in WHERE-clauses. (A special case exception will be described in Chapter 6.)*

Note that two strings are be equal only if all corresponding characters match. This means that you have to be careful about leading and embedded blanks. Note that the following WHERE-clause would not match the stored two-character STATE value of 'I '.

```
WHERE STATE = ' I' → No Hit
```

**Exercise:**

1E. Display all information about any nature preserve located in Montana.

## WHERE-Clause: Variable-Length Character-String Comparison

The following sample query specifies a condition that references PNAME, a variable-length character-string (VARCHAR) column.

**Sample Query 1.3b:** Display all information about the Ramsey Canyon nature preserve.

```
SELECT *
FROM PRESERVE
WHERE PNAME = 'RAMSEY CANYON'
```

PNO	PNAME	STATE	ACRES	FEE
80	RAMSEY CANYON	AZ	380	3.00

**Syntax & Logic:** Nothing New. This query objective requires retrieving information about the Ramsey Canyon preserve. Because the stored data contains all uppercase letters, this WHERE-condition requires specification of uppercase letters.

**Important Observation:** The STATE column contains CHAR values, and the PNAME column contains VARCHAR values. Note the similarity in the WHERE-clauses for this and the preceding sample query.

```
WHERE STATE = 'AZ'
```

```
WHERE PNAME = 'RAMSEY CANYON'
```

This similarity always applies with the exception of a rare special case circumstance that will be described in Chapter 6.

**Embedded Blanks:** We must know that there is exactly one blank character between individual words in a PNAME value. (We are told the system enforces this rule.) Hence, there is exactly one blank between RAMSEY and CANYON. This means that the following statements would produce a "no hit."

```
SELECT *
FROM PRESERVE
WHERE PNAME = 'RAMSEY CANYON'      → No Hit
```

```
SELECT *
FROM PRESERVE
WHERE PNAME = 'RAMSEY  CANYON'     → No Hit
```

### Exercise:

1F. Display all information about the Pine Butte Swamp preserve.

## CHAR versus VARCHAR: What's the Difference?

What is the difference between fixed-length (CHAR) versus variable-length (VARCHAR) character-strings? *We are not going to tell you here, but Chapter 6 will answer to this question.* Furthermore, with the exception of a few Chapter 6 sample queries, you will rarely need to consider these differences. Below we make some observations that apply throughout this book.

**CHAR Columns:** *Most CHAR columns in our sample tables (and real-world tables) contain trailing blanks.* However, when comparing a CHAR column to a constant character-string in a WHERE-clause, you never need to specify any trailing blanks.

Sample Query 1.3a discussed storing a one-character STATE code 'I' for Ireland in the two-character STATE column. We noted that the WHERE-clause to search for Irish nature preserves should look like:

```
WHERE STATE = 'I'
```

Again, we generally discourage coding trailing blanks like:

```
WHERE STATE = 'I '
```

**VARCHAR Columns:** Unlike CHAR columns, VARCHAR columns in our sample tables (and real-world tables) *almost never* contain trailing blanks. (We said "almost" because there is a special case circumstance that will be described in Chapter 6.) Therefore, when coding a constant character-string in a WHERE-clause, you *almost never need to specify trailing blanks* as illustrated by the following WHERE-clause specified in the preceding sample query.

```
WHERE PNAME = 'RAMSEY CANYON'
```

We generally discourage coding explicitly trailing blanks like:

```
WHERE PNAME = 'RAMSEY CANYON '
```

**Eyeballing Data Values:** In Chapter 0 we noted that the difference between a CHAR character-string and a VARCHAR character-string cannot be deduced by observing sample data. Eyeballing data in the STATE column does not allow you to definitively conclude that STATE is a fixed-length character-string. Likewise, eyeballing data in the PNAME column does not allow you to definitively conclude that PNAME is a variable-length character-string. Eyeballing data is not helpful because differences pertain to the internal (under-the-hood) representation of data. These differences will be described in Chapter 6.



## Displaying Specific Columns

Previous sample queries specified "SELECT \*" which asked the system to display all columns. In practice, we frequently want to display just some subset of columns. The following sample query illustrates how to achieve this objective by specifying the column-names of the desired columns in the SELECT-clause.

**Sample Query 1.4:** For every row in PRESERVE table, display its PNAME, ACRES, and STATE values (in that left-to-right column sequence).

```
SELECT PNAME, ACRES, STATE
FROM PRESERVE
```

PNAME	ACRES	STATE
HASSAYAMPA RIVER	660	AZ
DANCING PRAIRIE	680	MT
MULESHOE RANCH	49120	AZ
SOUTH FORK MADISON	121	MT
MCELWAIN-OLSEN	66	MA
TATKON	40	MA
DAVID H. SMITH	830	MA
MIACOMET MOORS	4	MA
MOUNT PLANTAIN	730	MA
COMERTOWN PRAIRIE	1130	MT
PINE BUTTE SWAMP	15000	MT
RAMSEY CANYON	380	AZ
HOFT FARM	90	MA
PAPAGONIA-SONOITA CREEK	1200	AZ

**Syntax:** Columns can be displayed in any left-to-right sequence. Commas must separate the column-names. You may optionally include one or more spaces before or after each comma.

**Logic:** No WHERE-clause is specified because all rows should be selected. Because the SELECT-clause specifies three column-names (PNAME, ACRES, STATE), only these column values are displayed.

**Terminology - Project:** This SELECT statement retrieves a subset of columns from all rows. Sometimes we will use the term "project" to refer to this kind of operation. (Appendix 1B will offer a more formal description of the project operation.)

**Duplicate Column-Names:** You can specify the same column-name multiple times in a SELECT-clause as illustrated below.

```
SELECT STATE, PNAME, FEE, FEE, FEE
FROM PRESERVE
```

Redundant specification of the same column-name may seem strange. However, it is valid and will produce three identical columns in the result table. Occasionally, this redundancy may be reasonable. For example, you may wish to return multiple copies of the same column to a front-end tool so that the front-end tool can perform a different calculation on each copy.

**Exercises:**

- 1G. Display the preserve number and name, in that left-to-right order, of all nature preserves.
- 1H. Display the state code and preserve name, in that left-to-right order, of all nature preserves.

## Display Some Subset of Rows and Columns

The following sample query does not introduce any new concepts or reserved words. This sample query uses previously described techniques to select some subset of rows (using a WHERE-clause) and some subset of columns (by specifying column-names in the SELECT-clause).

**Sample Query 1.5:** Display the PNAME and ACRES values of every nature preserve that is located in Arizona.

```
SELECT PNAME, ACRES
FROM PRESERVE
WHERE STATE = 'AZ'
```

<u>PNAME</u>	<u>ACRES</u>
HASSAYAMPA RIVER	660
MULESHOE RANCH	49120
RAMSEY CANYON	380
PAPAGONIA-SONOITA CREEK	1200

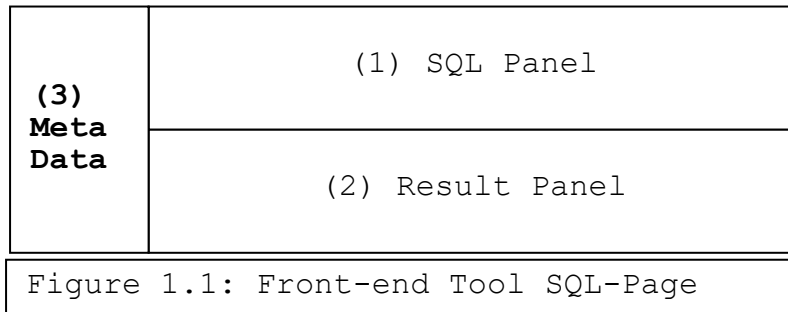
**Syntax & Logic:** Nothing new. Notice that this page's title uses the term "subset" to hint at the mathematical theory that serves as the theoretical foundation for all relational database systems. (You are invited to read the optional Appendix 1B to learn some basic concepts about this theory.)

### Exercises:

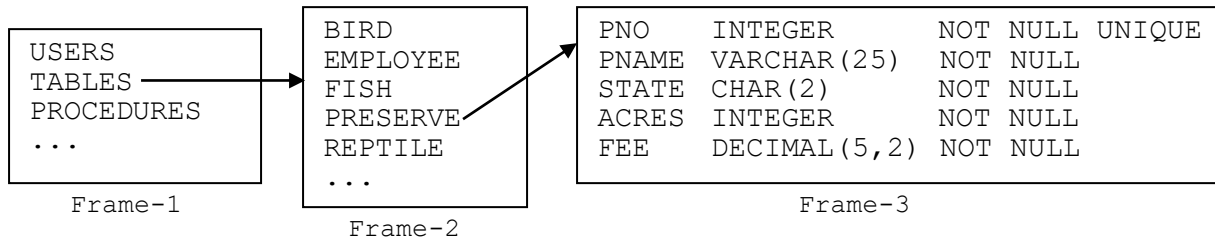
- 1I. Display the preserve number and name for all nature preserves where the number of acres exceeds 2,000.
- 1J. Display the preserve name of all nature preserves located in Massachusetts.

## Metadata: Learning about Tables, Columns, and Data-Types

Metadata is "data about data." Within the context of a relational database, metadata describes the names of all tables in the database, the column-names and data-types of all columns in these tables, and other relevant information. Most front-end tools provide a SQL-Page with a general structure that looks like the following Figure 1.1. (This figure enhances Figure 0.4).



Thus far, we have focused on (1) the SQL Panel where you enter your SELECT statements, and (2) the Result Panel which displays result tables. Most front-end tools also provide (3) a Metadata Panel that displays metadata. The Metadata Panel may look something like the following Frame-1.



In Frame-1, if you click on TABLES, Frame-2 appears with a list of all tables (or maybe just those tables that you are allowed to access). This list includes our EMPLOYEE and PRESERVE tables. When you click on PRESERVE, Frame-3 appears with a list of PRESERVE's column-names, corresponding data-types, NOT NULL indicators, and any UNIQUE designations. Notice that Frame-3 reflects the column descriptions specified in the CREATE TABLE statement that created PRESERVE. (See Figure 0.2.)

The Metadata Panel is very useful when you want to access unfamiliar tables. For example, assume you were just assigned to a new project for a zoology application. You could examine Frame-2, and then, using your intuition, start to explore the BIRD, FISH, and REPTILE tables.

Aside: Your front-end tool derives its metadata from the system's *Data Dictionary*. We will say more about data dictionaries later in this book.

## Qualified (Two-Part) Table-Names

All sample queries in this book reference a "one-part" table-name (e.g., PRESERVE). However, every table really has a "two-part" table-name (e.g., TM99999.PRESERVE) where the first part (TM9999) specifies the user-id of the owner of the table. (This user-id is usually the user-id of the person who executed the CREATE TABLE statement that created the table.)

Consider the following scenario.

Assume you are Jacqueline Juniper, and the DBA has assigned you a user-id of JJ011019. Then, if you were to create a table called PRESERVE, this table's qualified name would be: JJ011019.PRESERVE.

Now, assume you (Jacqueline) execute the following statement:

```
SELECT *
FROM PRESERVE
```

Because the system knows who you are, it automatically (under-the-hood) includes your user-id within the table-name and executes:

```
SELECT *
FROM JJ011019.PRESERVE
```

Next, assume two other users, Josephine Violet (user-id is JV061317) and Johnny Trouble (user-id is JT051015), also create tables called PRESERVE. The qualified names of these tables are: JV061317.PRESERVE and JT051015.PRESERVE. (These three PRESERVE tables may or may not have the same column-names and data-types.)

Question: Can you (Jacqueline) access Josephine's PRESERVE table by executing?

```
SELECT *
FROM JV061317.PRESERVE
```

Answer: Maybe. If Josephine has (somehow) granted you a SELECT-privilege on her JV061317.PRESERVE table, this SELECT statement will execute and display Josephine's PRESERVE table. However, if Josephine has not granted you this privilege, the system will deny you access to her table.

\* In this book, we assume that you have created all sample tables. Therefore, all sample queries will reference one-part table-names.

## Preview: Next Two Chapters

We review this chapter's result tables to make some observations about topics to be presented in the next two chapters.

**Chapter 2 (ORDER BY):** The result tables in this Chapter 1 are not displayed in any particular row sequence. Chapter 2 will show how an ORDER BY clause can be specified within a SELECT statement to produce a desired row sequence.

**Chapter 3 (DISTINCT):** The result tables in this Chapter 1 do not contain any duplicate rows. Chapter 3 will illustrate that some SELECT statements may produce duplicate rows. This chapter will introduce the DISTINCT keyword that removes duplicate rows (if present) from a result table.

You will see that specification of ORDER BY and DISTINCT is quite simple. However, know-your-data issues can present some complexity. For example, consider the following optional exercise pertaining to duplicate rows.

### Optional (Unfair) Preview Exercise:

This exercise is unfair because you have not read Chapter 3.

1K. Review the SELECT statement and result table for Sample Query 1.4. This SELECT statement is:

```
SELECT PNAME, ACRES, STATE
FROM PRESERVE
```

This result table does not show any duplicate rows. However, sometime in the future, in a very unusual circumstance, this result table could contain duplicate rows. Why might this happen?

## Summary

**Basic Syntax:** This chapter introduced the basic syntax of the SELECT statement as shown below.

```
SELECT column-name(s)
FROM table-name
[WHERE condition]
```

Brackets around the WHERE-clause indicate this clause is optional.

From a logical perspective, a SELECT statement is formulated by:

1. Identifying the table that contains the desired data and specifying its table-name in the FROM-clause.
2. Identifying the columns to be displayed and specifying their column-names in the SELECT-clause. ("SELECT \*" will display all columns.)
3. Coding a WHERE-clause to specify row selection criteria. The absence of a WHERE-clause implies that every row will be selected.

**Know-Your-Data:** You have to learn the name and data-type of each column in relevant tables. While this is not difficult, it may take some time. Also, some subtle points deserve your attention.

- Numbers versus character-strings: Sometimes a "number" is a string of digits that is defined as a character-string. For example, a social security may be defined as CHAR(9). We will see that mathematical calculations cannot be (directly) performed on a character-string of digits that happens to look like a number.

**Caveat:** Most systems provide some form of automatic data-type conversion. For example, some systems will execute the following statement and return the correct result.

```
SELECT *
FROM PRESERVE
WHERE PNO = '10'
```

The system will automatically convert the character-string '10' into an integer 10 before it does the comparison operation. You might think that automatic data-type conversion is a desirable feature. Maybe, in some circumstances. However, this author recommends the explicit specification of data-type conversion functions to be introduced in Chapter 10.

- **CHAR versus VARCHAR:** In this chapter, the under-the-hood differences between CHAR and VARCHAR values did not impact the coding of WHERE-clauses. However, Chapter 6 will present sample queries where these differences (to be described) become significant.
- **INTEGER versus DECIMAL:** Sample Query 1.2 explicitly coded a decimal point (WHERE FEE = 3.00) when comparing on a decimal column. We *discouraged* coding "WHERE FEE = 3" because FEE is a decimal column. However, most systems will automatically convert the integer 3 into a decimal value (3.0). Likewise, we would not code a decimal point when comparing on an integer value. For example, coding "WHERE ACRES > 100.00" is discouraged because ACRES is defined as INTEGER.

**Encoded Values:** Writing a correct WHERE-clause requires that you understand coding schemes used within your data. For example, you must know that the STATE column contains two-character versus four-character STATE codes. Our sample queries presume you know that MA (not MASS) represents the state of Massachusetts, MT (not MONT) represents Montana, and AZ (not ARIZ) represents Arizona. This coding scheme is simple. However, many real-world tables contain column values that utilize more complex coding schemes.

**Regarding Efficiency:** Don't worry about efficiency.

Focus on writing correct statements. At this early stage of learning SQL, efficiency is not important.

***It makes no sense to do the wrong thing fast!***

However, if you are interested in efficiency, you are invited to read Appendix 1A.

**Regarding Theory:** Don't worry about theory. However, if you are interested in the theoretical foundation of relational databases, you are invited to read Appendix 1B.



## Summary Exercises

Exercise 1D asked you to display the EMPLOYEE table. The result table looked like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
1000	MOE	2000.00	20
2000	LARRY	2000.00	10
3000	CURLY	3000.00	20
4000	SHEMP	500.00	40
5000	JOE	400.00	10
6000	GEORGE	9000.00	20

The following exercises reference this table. Details about its columns are described below.

ENO Employee Number: CHAR (4)  
This column contains unique values.  
Note: This "number" is represented by a character-string.

ENAME Employee Name: VARCHAR (25)

SALARY Employee Salary: DECIMAL (7, 2)

DNO Employee's Department Number: INTEGER

- 1L. Display all information about any employee whose SALARY value exceeds \$1,000.00.
- 1M. Display all information about Employee 2000 (i.e., ENO value is '2000').
- 1N. Display the ENAME and DNO values of every employee.
- 1O. Display the ENAME and SALARY values of every employee whose SALARY value is less than \$1,000.00.

## Appendix 1A: Efficiency

Again, we emphasize that *the primary objective of this book is to help you write logically correct SQL statements*. Therefore, you can skip this and other Efficiency Appendices. However, someday you may execute a correct SELECT statement and then have to wait a long time for the result to appear. These appendices offer some insight into why a SELECT statement may not execute with optimal efficiency. We begin by reconsidering the SELECT statement for Sample Query 1.5.

```
SELECT PNAME, ACRES
FROM PRESERVE
WHERE STATE = 'AZ'
```

To satisfy this query, the system has to perform multiple under-the-hood operations. Two of these operations are:

1. **Disk Input:** The system will access the disk to retrieve rows from the PRESERVE table and copy these rows into main memory.
2. **Process Rows:** The system will use its Central Processing Unit (CPU) to select those rows that match the WHERE-clause. Then it will extract the desired columns for display.

The first operation (reading the disk) is much slower than the second operation (CPU processing) because disk Input-Output (I/O) is always very slow when compared to CPU speed. Therefore, *database designers usually focus most of their attention on reducing disk I/O*.

When the system retrieves rows from disk, two factors impact efficiency. These are: (1) the size of the table, and (2) the access method the system uses to retrieve the desired rows.

**Table Size:** The SELECT statements for Sample Queries 1.1 and 1.4 did not specify WHERE-clauses. Hence, these queries asked the system to retrieve all rows from the PRESERVE table. Because PRESERVE only contains 14 rows, performance is very efficient. However, if PRESERVE contained 14 million rows, these queries would be much slower. The basic observation is that retrieving all rows (or many rows) from a large table would be slow because there would be a significant amount of disk input.

**Access Methods:** An access method is a low-level (under-the-hood) procedure used by the system to retrieve desired rows from disk. There are two general types of disk access methods: (1) *sequential scan* and (2) *direct access*.

1. A **sequential scan** of a table retrieves and copies all of its rows from the disk into main memory. Again, because the SELECT statements for Sample Queries 1.1 and 1.4 did not specify WHERE-clauses, these queries retrieved all rows. This type of query usually encourages the system to perform a sequential scan. Scanning a small table (analogous to reading a small book) is efficient. Scanning a large table (analogous to reading a large book) is less efficient.
2. A **direct access** method is basically a "shortcut" used by the system to retrieve only the desired rows. The system might use a direct access method if a table is large and the SELECT statement contains a WHERE-clause indicating that the desired result will only contain a few rows. The most popular direct access method is a database index.

**Database Indexes:** A database index is *conceptually* similar to an index found at the end of a very large book. If the book is about American history, you might ask: "What does this book say about Calvin Coolidge?" You could answer this question by reading the entire book (i.e., sequentially scan all pages in the book). Alternatively, it is probably more efficient to read the book's index and follow the page numbers that point to those pages that discuss Coolidge.

The following Figure A1.1 illustrates the basic features of a database index called INDSTATE which is based on the STATE column. This index could help Sample Query 1.5. Given the condition (STATE = 'AZ'), the system could access the index and follow the four AZ "pointers" to directly access the four Arizona rows. (A pointer is the physical disk address of a row.)

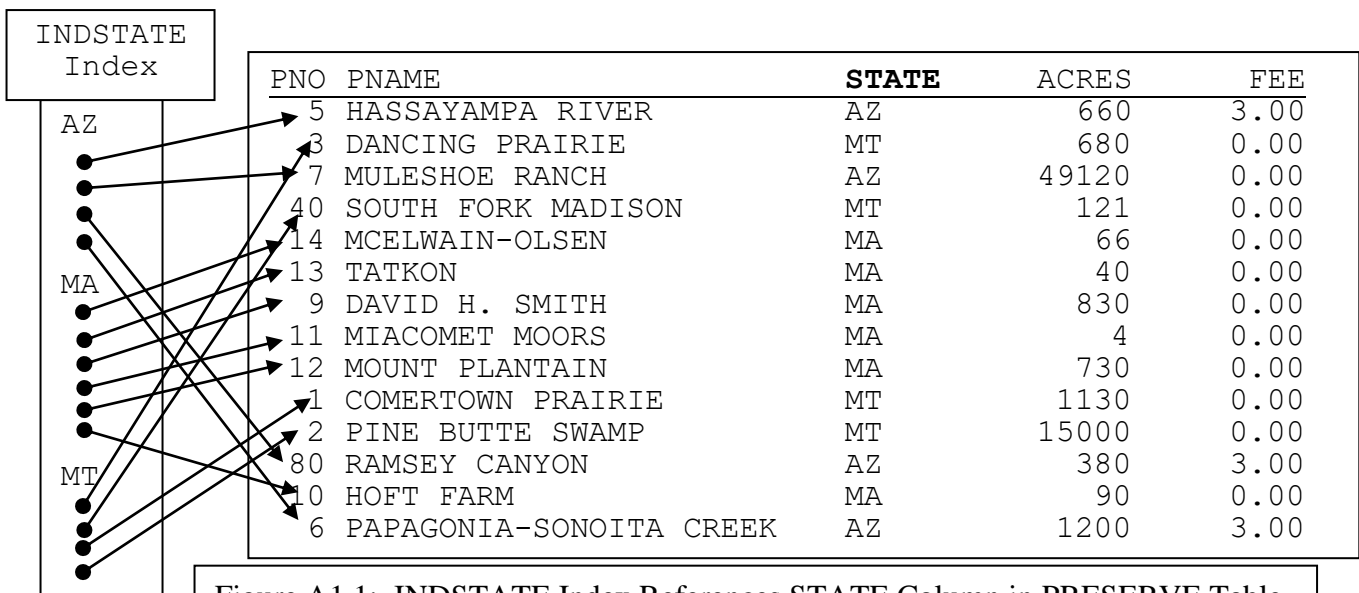


Figure A1.1: INDSTATE Index References STATE Column in PRESERVE Table

While there are conceptual similarities between a database index and a typical book index, two important differences must be noted.

First, imagine that, instead of a large book's index being located in the back of the book, the index is located in another much smaller book that references the page numbers in the larger book. In a similar manner, most (but not all) database indexes are stored separately from the table. Figure A1.1 shows that the INDSTATE index is a separate dataset from the PRESERVE table.

Second, most books have just one index that references all major topics. For example, a history book's index might reference the persons, locations, wars, treaties, etc. described in the book. However, a database table usually has multiple indexes where each index usually references just one column (analogous to one topic). Here the INDSTATE index is based on one column, the STATE column.

Because the PRESERVE table has six columns, the DBA could create six indexes, one for each column. (For the moment, we ignore the fact that it is possible to create a single index based on multiple columns.) Because an index incurs some cost (to be described in Appendix 2A), most designers only create a few indexes. Here we assume that PRESERVE has just one index (INDSTATE).

The system could use the INDSTATE index to satisfy Sample Query 1.5. This index could be very useful if the PRESERVE table contained 14 million rows. However, because PRESERVE only contains 14 rows, using this index would probably be slower than a sequential scan. (Appendix 4A will say more about this scenario.)

**No Relevant Index:** What if you looked at the end of the large history book and did not find an index? Then, to satisfy a query objective, you would be forced to read (sequentially scan) the entire book. Now, consider a similar scenario. Assume PRESERVE has 14 million rows and consider the following query that presumably retrieves just a few rows.

```
SELECT *
FROM PRESERVE
WHERE FEE = 97.22
```

Because there is no index of the FEE column, the system must scan the disk to copy every row into memory, and then examine the FEE value in each row to determine if it equals 97.22. The basic observation is that, in the absence of a useful index, a simple SELECT statement could incur an expensive sequential scan.

[Appendix A2 will continue this discussion of database indexes.]

## Appendix 1B: Theory

Again, we emphasize that the primary objective of this book is to help you write logically correct SQL statements. Therefore, as with the Efficiency Appendices, you can skip this and other Theory Appendices. However, you are encouraged to read on.

**Why Theory?** We offer a few good reasons for discussing database theory within a tutorial textbook directed towards practitioners.

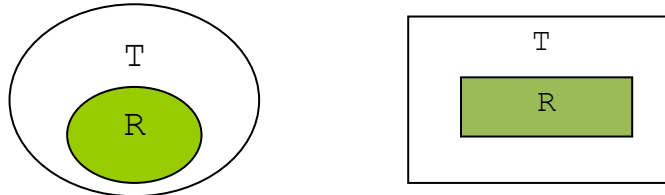
- Database theory is interesting.
- Database theory is not difficult.
- Database theory offers practical insights.
- These Theory Appendices will complement theory topics presented in an academic database course. They offer a potpourri-like informal introduction to a formal topic.

**History:** Once upon a time, way back in the late 1960s, Ted Codd, a researcher at IBM, concluded that his employer's database system (IMS/DLI), and every other commercial database system, contained many undesirable properties. These properties were associated with the architecture; they were so fundamental that they could not be remedied with just a few modifications to the database software. So Codd started from scratch and proposed a novel idea (a "model") of what a good database *should* look like. He called his idea a "Relational Model" for database systems.

Also, in the late 1960's, people began using the term "software engineer." When designing systems, engineers frequently use applied mathematics which is derived from some area of theoretical mathematics. (Do you want to fly in an airplane designed by an aeronautical engineer who did poorly in his math courses?) Codd's Relational Model offered a firm mathematical foundation to a new discipline called "data engineering." This foundation was based upon elementary set theory.

Codd effectively said to IBM and the rest of the world: Here is a good idea (a good model). I encourage you to build your database systems based on this idea. Sometime around 1974 IBM took his advice. Codd's Relational Model served as the basic architecture for IBM's System-R, a prototype system that evolved into DB2. Other database vendors (e.g., ORACLE) did likewise. The rest is history.

**Codd's Fundamental Insight:** Codd's Relational Model was based on set theory. A simple way to demonstrate the link between set theory and a relational database is to draw two logically equivalent Venn diagrams. (Hopefully you recall from junior high school that a Venn Diagram can be used to illustrate the concept of a subset.)



These diagrams represent the same concept: Set R is a subset of set T. Most math textbooks use circles/ovals to represent a Venn diagram as shown in left diagram. From a relational base perspective, we prefer the rectangular diagram where set T represents a database table, and the subset R represents a result table produced by a query against table T.

**Relational Model:** Codd's Relational Model is organized around three concepts. We present the first two concepts: (1) the structure of data, and (2) the query language. (The third concept, database integrity, will be presented later in this book.)

## 1. Structure of Data

Codd wanted data to be represented by sets. For example, you can consider the PRESERVE table to be a set where each row is an *element* of the set. Furthermore, you can also consider the entire database to be a set of tables. Therefore, a database is set of tables, where each table is itself a set of rows.

Codd's Terminology: A mathematical "**Relation**" formalizes a table. A relations contain "*n-tuples*" which formalize rows. Each n-tuple contains "*attributes*" which formalize column values.

## 2. Query Languages

Codd described a query result as a subset of a set. Consider the following example.

You are given set T consisting of all even integers between and including 2 and 10. (T is analogous to a table.)

$$T = \{2, 4, 6, 8, 10\}$$

Query Objective: Retrieve all T values that are greater than 5. The following set R is a subset of T that satisfies this query objective. (R is analogous to a result table.)

$$R = \{6, 8, 10\}$$

We have described set T and set R by enumerating their elements. This is convenient because these sets are very small. Mathematicians, in order to deal with very large (perhaps infinite) sets, use a mathematical language called the *predicate calculus*. Using the predicate calculus, the following notation describes set R as a subset of set T.

$$R = \{x \in T: x > 5\}$$

This notation says: "Set R contains those elements (x) *from* set T where each element (x) is greater than 5."

Codd's critical insight is that, if *relation* T (table T) is a set within a *relational* database system, the predicate calculus could be used to describe the *subset relation* (the result table) that you want the database system to retrieve. The user would code a statement similar to  $\{x \in T: x > 5\}$  and submit it to the system. The system would analyze the statement and return R, the desired subset.

Codd extended the predicate calculus into a query language for his Relational Model. He called this language the "relational calculus." However, some influential managers at IBM did not like Codd's relational calculus. Apparently, they did not think it was a very friendly language.

Codd responded by developing another (presumably friendlier) query language that he called the "relational algebra." Again, some people at IBM did not like this language. But they did like the idea of storing data in tables. Eventually other database researchers at IBM developed SQL. Conceptually, a SELECT statement has language features derived from both the relational calculus and the relational algebra. Below, we illustrate some similarities between SQL and Codd's database languages.

**SQL & the Relational Calculus:** Reconsider Sample Query 1.3a.

```
SELECT *
FROM PRESERVE
WHERE STATE = 'AZ'
```

The following calculus statement is equivalent to this SELECT statement.

$$\{r \in \text{PRESERVE}: r.\text{STATE} = \text{'AZ'}\}$$

Here r represents a row in the PRESERVE table, and r.STATE = 'AZ' corresponds to a WHERE-clause which identifies the desired subset of rows to be retrieved.

**SQL & the Relational Algebra:** Codd defined eight operations in his relational algebra. This chapter has already introduced two of these operations: restrict and project.

### Restrict

Informally, restrict is a database operation that returns a horizontal subset of rows. Sample Query 1.2 executes a restrict operation.

```
SELECT *
FROM PRESERVE
WHERE FEE = 3.00
```

This statement retrieves a *horizontal subset* of PRESERVE. We call this horizontal subset a “restriction” of PRESERVE.

PRESERVE



Sample Queries 1.3a and 1.3b also represent restrict operations on PRESERVE.

### Project

Informally, project is a database operation that returns a vertical subset of a table. Sample Query 1.4 executes a project operation.

```
SELECT PNAME, ACRES, STATE
FROM PRESERVE
```

This statement retrieves a *vertical subset* of PRESERVE. We call this vertical subset a “projection” of PRESERVE.

PRESERVE



Note: Codd’s definition of project (unlike SQL) automatically removes duplicate rows. (Duplicate rows will be discussed in Chapter 3.)

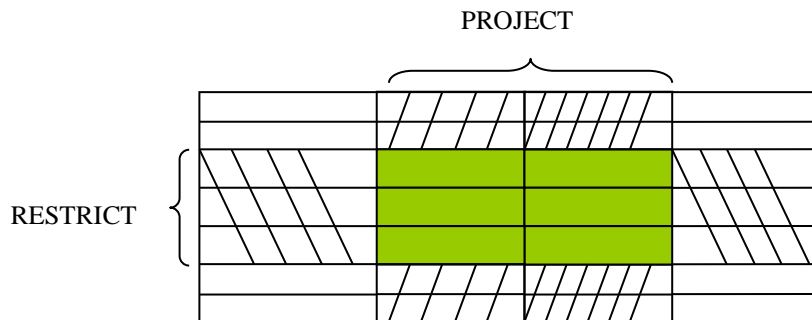


## Combining Restrict & Project

The SELECT statement for Sample Query 1.5 specifies both a restrict operation and a project operation.

```
SELECT PNAME, ACRES
FROM PRESERVE
WHERE STATE = 'AZ'
```

When a SELECT statement executes both restrict and project operations, the result table can be visualized as the intersection of a horizontal subset (defined by the WHERE-clause) and a vertical subset (defined by the SELECT-clause) as illustrated below.



*Conceptually*, this SELECT statement asks the system to (i) perform a restrict operation on PRESERVE to produce an intermediate result, and then (ii) perform a project operation of this intermediate result to produce the final result.

**Efficiency Comment:** Some Efficiency Appendices will describe the physical implementation of relational operations. For example, the system might use an index to implement the restrict operation. It may also incorporate project within restrict. For example, in the above SELECT statement, after the system retrieves an Arizona row, it may immediately extract its PNAME and ACRES values before retrieving the next Arizona row.

**History:** This appendix has introduced Codd's *classic* Relational Model as he defined it over 50 years ago. Subsequently, other database researchers have refined and extended this model.

**Conclusion:** Codd's grand idea, which laid the foundation for a trillion-dollar industry, is an elegant application of basic set theory.

# Sorting the Result Table:

## ORDER BY Clause

**Terminology - "Base Table" versus "Result Table":** Occasionally we will use the term "base table" to refer to a stored table such as the PRESERVE and EMPLOYEE tables. The CREATE TABLE statement creates a base table. Executing a SELECT statement produces a "result table" which is not a base table.

**ORDER BY Clause:** In general, we cannot make any assumptions about a predefined row sequence within a base table. However, it is usually desirable to display a result table in some specific row sequence. This objective is realized by coding an ORDER BY clause in the SELECT statement. (An ORDER BY clause cannot be specified in a CREATE TABLE statement.)

This chapter will demonstrate the simplicity and flexibility of the ORDER BY clause.

## ORDER BY Clause: Sorting by a Single Column

The following sample query specifies an ORDER BY clause to sort the result table by a single column.

**Sample Query 2.1:** Retrieve the STATE, PNO, and PNAME values from all rows in the PRESERVE table. Display the rows sorted by STATE values in ascending sequence.

```
SELECT STATE, PNO, PNAME
FROM PRESERVE
ORDER BY STATE
```

STATE	PNO	PNAME
AZ	5	HASSAYAMPA RIVER
AZ	7	MULESHOE RANCH
AZ	80	RAMSEY CANYON
AZ	6	PAPAGONIA-SONOITA CREEK
MA	14	MCELWAIN-OLSEN
MA	13	TATKON
MA	9	DAVID H. SMITH
MA	11	MIACOMET MOORS
MA	12	MOUNT PLANTAIN
MA	10	HOFT FARM
MT	3	DANCING PRAIRIE
MT	40	SOUTH FORK MADISON
MT	1	COMERTOWN PRAIRIE
MT	2	PINE BUTTE SWAMP

**Syntax:** The ORDER BY clause is usually the last clause in a SELECT statement. ORDER BY is followed by the name of the sort column. You can sort by any column. However, as illustrated here, you usually sort by the leftmost column in the result table.

**Logic:** This ORDER BY clause specifies STATE as the sort column. Because STATE contains character-string values, the sequence is an alphabetical sequence. Here the sequence defaults to an ascending sequence. Alternatively, you can use the ASC keyword to explicitly specify an ascending sequence.

```
ORDER BY STATE ASC
```

Sample Query 2.3 will illustrate the DESC keyword that produces a descending sort sequence.

**Important Observation:** We cannot make any assumptions about a second-level sort sequence within duplicate STATE values. The following sample query illustrates sorting by two columns to establish a two-level sort sequence.

## Sorting by Multiple Columns

The following sample query illustrates that an ORDER BY clause can reference multiple columns.

**Sample Query 2.2:** Display STATE, PNO, and PNAME values for every nature preserve. Sort the result by PNO within STATE. (I.e., STATE is the primary sort column, and PNO is the secondary sort column.)

```
SELECT STATE, PNO, PNAME
FROM PRESERVE
ORDER BY STATE, PNO
```

STATE	PNO	PNAME
AZ	5	HASSAYAMPA RIVER
AZ	6	PAPAGONIA-SONOITA CREEK
AZ	7	MULESHOE RANCH
AZ	80	RAMSEY CANYON
MA	9	DAVID H. SMITH
MA	10	HOFT FARM
MA	11	MIACOMET MOORS
MA	12	MOUNT PLANTAIN
MA	13	TATKON
MA	14	MCELWAIN-OLSEN
MT	1	COMERTOWN PRAIRIE
MT	2	PINE BUTTE SWAMP
MT	3	DANCING PRAIRIE
MT	40	SOUTH FORK MADISON

**Syntax:** ORDER BY is followed by the primary sort column (STATE) which is followed by the secondary sort column (PNO). A comma must separate the column names.

An ORDER BY clause can specify (practically) any number of columns. The following ORDER BY clause is valid and would establish a five-level sort sequence.

```
ORDER BY COL1, COL2, COL3, COL4, COL5
```

**Logic:** Observe the Arizona group (the first four rows) in this result table. The PNO values within this group are sorted (5, 6, 7, 80). Likewise for the MA and MT groups.

Because STATE contains character-string values, the STATE sequence is an alphabetical sequence; and, because PNO contains integer values, the PNO sequence is based on mathematical value.

## Descending Sort: DESC

The following sample query demonstrates the use of the DESC keyword to produce a result table that is displayed in a descending sequence.

**Sample Query 2.3:** Display the PNO, PNAME, and ACRES values acres for all nature preserves that are located in Arizona. Display the result by PNO values in descending sequence.

```
SELECT PNO, PNAME, ACRES
FROM PRESERVE
WHERE STATE = 'AZ'
ORDER BY PNO DESC
```

PNO	PNAME	ACRES
80	RAMSEY CANYON	380
7	MULESHOE RANCH	49120
6	PAPAGONIA-SONOITA CREEK	1200
5	HASSAYAMPA RIVER	660

**Syntax:** DESC follows the column-name in the ORDER BY clause. One or more spaces must separate the column name and the DESC keyword.

If an ORDER BY clause references multiple columns, ASC or DESC can be specified for each column as shown below.

```
ORDER BY COL1 ASC, COL2 DESC, COL3 ASC, COL4 DESC, COL5 ASC
```

**Logic:** Because the sort column (PNO) contains integer values, the descending sequence is based on mathematical value.

## “Sequence” versus “Sort”

Previous query objectives used the terms “sort” and “sequence” in a very casual manner. Sample Query 2.1 asked you to “Display the rows ... in ascending *sequence*,” whereas Sample Query 2.2 asked you to “*sort* the result by ...”.

**Sequence:** Stating a desired row sequence (versus asking the system to sort by some column) is a more accurate articulation of the query objective because it is a declarative statement. It states “what” to do, not “how to” do it.

**Sort:** Sorting is an internal (under-the-hood) process that may or may not be used to satisfy some sequence objective. (Appendix 2A will describe another internal method the system may use to produce a desired row sequence.)

Having made this distinction, many future query objectives will continue to state something like: “Sort the rows by column COLX.”

### **Exercise:**

- 2A. Display the entire PRESERVE table. Sort the result by the ACRES column in ascending sequence.
- 2B. Display the preserve name and admission fee of every nature preserve located in Montana. Sort the result by preserve name in descending sequence.
- 2C. Display the FEE and ACRES columns for every row in the PRESERVE table. Sort the displayed rows by ACRES within FEE. (FEE is the major sort field, and ACRES is the minor sort field.)

## ORDER BY Column-Number

An ORDER BY clause can reference a column by its relative position in the result table.

**Sample Query 2.4:** Display the preserve number, acres, and name of every nature preserve located in Arizona. Sort the result by the third column.

```
SELECT PNO, ACRES, PNAME
FROM PRESERVE
WHERE STATE = 'AZ'
ORDER BY 3
```

PNO	ACRES	PNAME
5	660	HASSAYAMPA RIVER
7	49120	MULESHOE RANCH
6	1200	PAPAGONIA-SONOITA CREEK
80	380	RAMSEY CANYON

**Syntax & Logic:** The relative column-number (3) refers to the column position in the result table, not the underlying base table. Here, PNAME is the third column of the result table (although PNAME is the second column in the underlying base table).

**Recommendation:** Specifying a column-number is acceptable for a one-time ad hoc query. However, if the SELECT statement will be saved for future execution, it is better to explicitly specify a column-name. This enhances readability and will not be affected by any future change to the SELECT-clause that reorders the left-to-right column sequence. This advice is especially relevant for application developers who code SELECT statements that will be embedded within stored procedures and application programs.

### Exercise:

2D. Display the entire PRESERVE table sorted by the fourth column in descending sequence.

## Sorting by a Non-displayed Column

Sometimes you want to sort by a column, but you do not want to display values from the sorted column. For example, the following result table is sorted by the ACRES column, but it does not display the ACRES values.

**Sample Query 2.5:** Display the PNO and PNAME values for all preserves. Display the result by ACRES in descending sequence.

```
SELECT PNO, PNAME
FROM PRESERVE
ORDER BY ACRES DESC
```

<u>PNO</u>	<u>PNAME</u>
7	MULESHOE RANCH
2	PINE BUTTE SWAMP
6	PAPAGONIA-SONOITA CREEK
1	COMERTOWN PRAIRIE
9	DAVID H. SMITH
12	MOUNT PLANTAIN
3	DANCING PRAIRIE
5	HASSAYAMPA RIVER
80	RAMSEY CANYON
40	SOUTH FORK MADISON
10	HOFT FARM
14	MCELWAIN-OLSEN
13	TATKON
11	MIACOMET MOORS

**Syntax & Logic:** Nothing new. You rarely want to sort a result table by a non-displayed column. However, on occasion, this can be useful. For example, you might want to display the ENAME column from the EMPLOYEE table and sort the result by the SALARY column. However, for confidentiality reasons, you do not want to display the SALARY values.

```
SELECT ENAME
FROM EMPLOYEE
ORDER BY SALARY
```

### Exercise:

2E. Assume (unrealistically) that PNO values are considered to be confidential. Display the PNAME value for each Arizona nature preserve. Display the result in ascending PNO sequence without displaying the PNO values.



The following sample query does not introduce any new concepts or keywords. It merely illustrates that all variations of the ORDER BY clause can be incorporated within a single clause. The following sample query, while not very realistic, demonstrates the flexibility of the ORDER BY clause.

**Sample Query 2.6:** Display the STATE, FEE, and PNAME values from all rows in the PRESERVE table where:

- STATE is the 1<sup>st</sup> level sort column (ascending)
- FEE is the 2<sup>nd</sup> level sort column (descending)
- PNAME is the 3<sup>rd</sup> level sort column (descending)

```
SELECT STATE, FEE, PNAME
FROM PRESERVE
ORDER BY STATE ASC, FEE DESC, 3 DESC
```

STATE	FEE	PNAME
AZ	3.00	RAMSEY CANYON
AZ	3.00	PAPAGONIA-SONOITA CREEK
AZ	3.00	HASSAYAMPA RIVER
AZ	0.00	MULESHOE RANCH
MA	0.00	TATKON
MA	0.00	MOUNT PLANTAIN
MA	0.00	MIACOMET MOORS
MA	0.00	MCELWAIN-OLSEN
MA	0.00	HOFT FARM
MA	0.00	DAVID H. SMITH
MT	0.00	SOUTH FORK MADISON
MT	0.00	PINE BUTTE SWAMP
MT	0.00	DANCING PRAIRIE
MT	0.00	COMERTOWN PRAIRIE

**Syntax and Logic:** Nothing new. The ORDER BY clause references the STATE and FEE columns by name. Quite arbitrarily, the PNAME column is referenced as the third column. The system displays STATE values in ascending sequence. FEE and PNAME values are displayed in descending sequence.

**Exercise:**

2F. Display the STATE, FEE, and PNO values for any preserve having more than 100 acres. Sort the result table. STATE is the first-level sort field in descending sequence. FEE is the second-level sort field in descending sequence. PNO is the third-level sort field in ascending sequence.

## Incidental Sort

The following query objective asks the system to display all values from a single column. No ORDER BY clause is specified because the query objective does designate any row sequence.

**Sample Query 2.7:** Display the PNO values for all preserves.

```
SELECT PNO
FROM PRESERVE
```

```
 PNO
  1
  2
  3
  5
  6
  7
  9
 10
 11
 12
 13
 14
 40
 80
```

**Syntax & Logic:** Nothing new. Note: No ORDER BY clause is specified, but the result is incidentally sorted. (Without explanation, this incidental sort would probably, but not necessarily, occur on most systems. Appendix 2A offers an explanation for this behavior.)

**Should You Care about Incidental Sorts:** You can correctly say: "So what - I don't care about any incidental sort." Good. But remember that you must specify an ORDER BY clause when you do want a sorted result.

**Important Observation:** This incidental sort may or may not happen in the future. Consider the following scenario. On Monday, you execute the above statement and observe an incidental sort. Then, you execute the very same statement on the following Wednesday, and you do *not* see the same incidental sort. Did some system change occur on Tuesday? Maybe. But you don't need to know. (Read Appendix 2A if you do want to know.)

**Conclusion:** From a logical perspective, if you do not specify an ORDER BY clause, you cannot make any unqualified prediction about the presence or absence of a row sequence.

## Character (Collating) Sequence

Determining row sequence becomes slightly more complex if character-strings contain both uppercase and lowercase letters, digits, and special symbols (e.g., !, ?, +). Your system resolves this complexity by sorting according to a system-defined "collating sequence."

**ASCII Collating Sequence:** Most database systems (excluding DB2 mainframe) use the ASCII collating sequence to determine the sequence of a column of character-strings. With the ASCII sequence, most (but not all) special characters sort before the digits, which sort before the uppercase letters, which sort before the lowercase letters. A blank character (space) is a special character that sorts before all characters. Your SQL reference manual will describe the details.

**ASCII Example:** The following figure shows an unsorted column of character-string values and the same values sorted according to the ASCII sequence.

<u>Unsorted</u>	<u>Sorted (ASCII)</u>
Zeek	!!!FIDO!!!
jessie	3M
JULIE	77aaaaaaaaAAAA
	JEssie
77aaaaaaaaAAAA	JULIE
JEssie	JULIe
julie	Jessie
Jessie	Zeek
3M	jessie
!!!FIDO!!!	julie

**DB2 Mainframe:** The DB2 mainframe database system uses a different collating sequence called the EBCDIC sequence. This chapter's Summary displays the above sample data in the EBCDIC sequence.

## Character-String Comparison

We usually compare character-string data with the equals (=) comparison operator. However, occasionally we want to compare character-string data using some other comparison operator (<, >, <=, >=, <>).

**Sample Query 2.8:** Display all information about any nature preserve with a PNAME value that follows the letter R in alphabetical sequence.

```
SELECT *
FROM PRESERVE
WHERE PNAME > 'R'
```

PNO	PNAME	STATE	ACRES	FEE
40	SOUTH FORK MADISON	MT	121	0.00
13	TATKON	MA	40	0.00
80	RAMSEY CANYON	AZ	380	3.00

**Syntax:** Nothing new.

**Logic:** Because the PNAME column contains uppercase letters with embedded spaces, character-string comparison produces a simple alphabetic sequence. Any row having a PNAME value that is alphabetically greater than R will be displayed.

Recall that a blank character (space) is a special character that sorts before all characters. When the system compares the string 'R' (length of 1) to 'RAMSEY CANYON' (length of 13), it effectively pads the 'R' with 12 trailing blanks such that its effective length is 13. Then it compares the following strings:

```
'R'
'RAMSEY CANYON'
```

RAMSEY CANYON appears in the result because its second character (A) is greater than the blank character.

### Exercises:

- 2G. Display all rows where the STATE value is greater than or equal to the letter M.
- 2H. Display every row where the preserve name value is less than TATKON.

## Deterministic versus Non-Deterministic Statements & Result Tables

Below we distinguish between a "deterministic" versus a "non-deterministic" statement and result table. Consider the following examples.

**Example-1:** Reconsider Sample Query 2.1 and the first four rows in its result table.

```
SELECT STATE, PNO, PNAME
FROM PRESERVE
ORDER BY STATE
```

STATE	PNO	PNAME
AZ	5	HASSAYAMPA RIVER
AZ	7	MULESHOE RANCH
AZ	80	RAMSEY CANYON
AZ	6	PAPAGONIA-SONOITA CREEK
. . . . .		

This ORDER BY clause specified a single *non-unique* column (STATE). Because a second-level sort is not specified, the four Arizona rows could have appeared in a different row sequence such as that shown below.

STATE	PNO	PNAME
AZ	80	RAMSEY CANYON
AZ	5	HASSAYAMPA RIVER
AZ	7	MULESHOE RANCH
AZ	6	PAPAGONIA-SONOITA CREEK
. . . . .		

Both of the above result tables are considered to be correct because both results satisfy the query objective which was silent about any second-level sequence. When a given SELECT statement can produce multiple correct results, we can say that the statement is a **non-deterministic statement**, and the result is a **non-deterministic result**.

**Comment:** We usually try to avoid non-deterministic statements with non-deterministic result tables. (See the following Example-2.)

**Comment:** Chapter 10.5 will introduce some useful non-deterministic functions such as the Date Functions which can return different results according to the date of statement execution.

**Example-2:** The following SELECT statement and result table for Sample Query 2.2 are deterministic.

```
SELECT STATE, PNO, PNAME
FROM PRESERVE
ORDER BY STATE, PNO
```

STATE	PNO	PNAME
AZ	5	HASSAYAMPA RIVER
AZ	6	PAPAGONIA-SONOITA CREEK
AZ	7	MULESHOE RANCH
AZ	80	RAMSEY CANYON
.	.	.

This ORDER BY clause references a *unique* column (PNO). Hence, this result is a **deterministic** result.

**Example-3:** Consider the following statement and result table.

```
SELECT STATE, PNAME
FROM PRESERVE
ORDER BY STATE, PNAME
```

STATE	PNAME
AZ	HASSAYAMPA RIVER
AZ	MULESHOE RANCH
AZ	PAPAGONIA-SONOITA CREEK
AZ	RAMSEY CANYON
.	.

This statement and its result table are non-deterministic because neither the STATE nor PNAME columns are declared to be UNIQUE. You might reasonably presume that no two nature preserves would have the same name. Furthermore, it would be very reasonable to presume that no two nature preserves within the same state would ever have the same name. However, you should not make these presumptions. Therefore, you cannot conclude that this statement and its result are deterministic.

If you want a deterministic result, you can include the unique PNO column in the ORDER BY clause as shown below.

```
SELECT STATE, PNAME
FROM PRESERVE
ORDER BY STATE, PNAME, PNO
```

Note: The PNO column is not displayed.

## “First” N Rows

Different systems offer different methods to realize the following query objective.

Preliminary Comment: Theory Appendix 2B will explain why the word “first” is enclosed within quotation marks.

**Sample Query 2.9:** Reference the PRESERVE table. Display the PNO and ACRES values of the “first” three rows. Sort the result by PNO values. The result table should look like:

<u>PNO</u>	<u>ACRES</u>
1	1130
3	680
9	830

DB2 and MYSQL can specify the LIMIT clause

```
SELECT PNO, ACRES                                DB2 & MYSQL
FROM PRESERVE
ORDER BY PNO
LIMIT 3
```

DB2 and ORACLE can specify the FETCH clause

```
SELECT PNO, ACRES                                DB2 & ORACLE
FROM PRESERVE
ORDER BY PNO
FETCH FIRST 3 ROWS ONLY
```

SQL Server can specify the TOP clause

```
SELECT TOP (3) PNO, ACRES                        SQL Server
FROM PRESERVE
ORDER BY PNO
```

Each of the above clauses support code variations that provide some flexibility. For example, you could specify an offset such that you can skip some designated number of rows before displaying the “first” N rows. Consult your SQL manual to learn these coding variations.

The above clauses specify an ORDER BY clause. This is not required, but it is a good idea. Without this ORDER BY clause, the result would not be deterministic.

## Summary

This chapter expanded our generic SELECT statement to include the optional ORDER BY clause. (Again, brackets around a clause imply that the clause is optional.)

```
SELECT column-name(s)
FROM table-name
[WHERE condition]
[ORDER BY sort-column(s)]
```

A result table can be sorted by one or more columns. The ORDER BY clause can reference a column-name or a relative column-number. The default sort sequence is ascending (ASC). The DESC keyword can be used to specify a descending sequence.

**DB2 Mainframe - EBCDIC Sequence:** Almost all systems utilize the ASCII collating sequence to determine the sequence of character-strings. Mainframe DB2 uses the EBCDIC collating sequence. An example of the EBCDIC is shown below. Most special symbols sort low, followed by lowercase letters, followed by uppercase letters, followed by digits.

<u>Unsorted</u>	<u>Sorted (EBCDIC)</u>
Zeek	!!!FIDO!!!
jessie	jessie
JULIE	julie
	Jess
77aaaaaaaaAAAA	Jessie
JEssie	JULIe
julie	JULIE
Jessie	Zeek
3M	3M
!!!FIDO!!!	77aaaaaaaaAAAA

The ASCII and EBCDIC sequences are very different. You might wish to compare the above EBCDIC example with the previously described example of an ASCII sequence.



## Summary Exercises

The following three exercises (2I, 2J, and 2K) pertain to the previously described EMPLOYEE table. Column-names are ENO, ENAME, SALARY, and DNO.

- 2I. Display the entire EMPLOYEE table sorted by employee name in ascending sequence.
- 2J. Display the name and salary of any employee whose salary is greater than \$2,000.00. Sort the result by salary in descending sequence.
- 2K. Display the department number, employee number, and employee name of all employees. Sort the result by employee number (in ascending sequence) within department number (in descending sequence).
- 2L. Do Sample Queries 2.3 - 2.5 return deterministic result tables?

```
SQ 2.3:  SELECT PNO, PNAME, ACRES
         FROM PRESERVE
         WHERE STATE = 'AZ'
         ORDER BY PNO DESC
```

```
SQ 2.4:  SELECT PNO, ACRES, PNAME
         FROM PRESERVE
         WHERE STATE = 'AZ'
         ORDER BY 3
```

```
SQ 2.5:  SELECT PNO, PNAME
         FROM PRESERVE
         ORDER BY ACRES DESC
```

## Appendix 2A: Efficiency

In Appendix A1, we noted that a relational database system can retrieve rows from a table by: (i) scanning all rows in the table, or (ii) using an index to directly access only the desired rows that were identified by a WHERE-clause. This appendix has more to say about indexes. However, before discussing indexes, we make some preliminary comments about sorting rows.

### Sorting

Although computer science textbooks describe a wide variety of sort methods, some general observations apply to all methods.

**Table Size:** The efficiency of a sort operation primarily depends upon the size of the result table (or intermediate result table) that needs to be sorted.

**Main Memory Sorting:** If the data is relatively small, then the system may be able to perform the entire sort operation within main memory. This internal sorting is very efficient. However, if the data are too large to fit into main memory, sorting becomes more complex and expensive.

**External (Non-Main-Memory) Sorting:** In general, the system will first sort one part of the data and write the sorted result to a temporary area on disk. Then it will sort a second part of the data and write the sorted result to another temporary area on disk. Etc. Finally, the system reads and merges the sorted intermediate results from the temporary disk areas, and passes a completely sorted result onto the next processing step. Frequently, this next step returns the sorted result to a front-end tool or program. Note that writing data to and then reading data from temporary disk areas involves relatively expensive disk I/O.

**Example:** Consider the result table size for Sample Query 2.1.

```
SELECT *
FROM PRESERVE
ORDER BY STATE
```

This statement retrieves all rows from the PRESERVE table. Because this table only has 14 rows, the sort would involve a very fast in-memory sort. However, if PRESERVE had 14 million rows, the sort operation would involve disk I/O and be much slower.

Distribution of Values: Assume PRESERVE has 14 million rows, but only 10 rows have a FEE of 9.23. Consider the following statement.

```
SELECT *
FROM PRESERVE
WHERE FEE = 9.23
ORDER BY PNAME
```

The system may or may not have to do a lot of work to retrieve the desired 10 rows. But, the subsequent cost of sorting the 10 rows would be trivial. However, if 9 million rows had a FEE value of 9.23, the system may have to sort the 9 million rows implying a significantly greater sort cost.

### More about Indexes

Assume the DBA has created an index called XPNO that is based on the PNO column. Recall that the PNO column contains unique values. Hence, XPNO is a *unique index*. The following Figure A2.1 illustrates this index and its connection to the PRESERVE table.

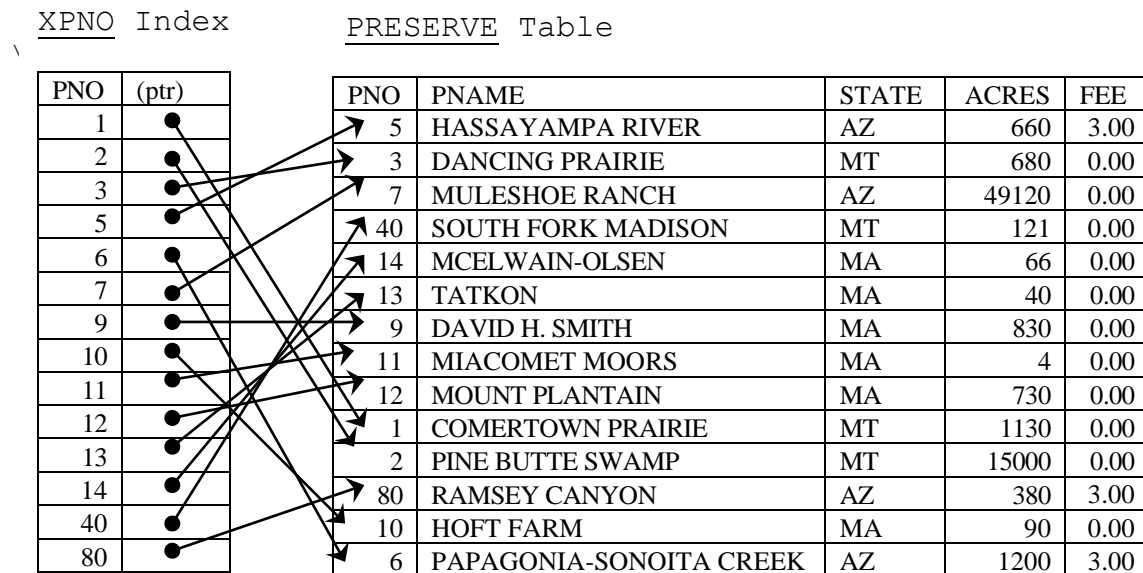


Figure A2.1: XPNO Index

The XPNO index has the same basic structure as the INDSTATE index described in Appendix A1. Hence, it can be used for a direct-access search. For example, consider the following statement:

```
SELECT *
FROM PRESERVE
WHERE PNO = 10
```

The system can search the XPNO index for 10 and then follow the "pointer" to the desired row. (The pointer is the disk address of the desired row.) If the PRESERVE table were very large, this direct access operation would be more efficient than scanning the table. If the PRESERVE table were very small, the system would ignore this index and scan the table.

**Other Index Advantages:** Beyond direct access advantages, indexes offer other advantages that will be described below. We begin by making a simple but very *important observation*.

<b>Index values are stored in sequence.</b>
---

Figure A2.1 shows that the PNO values in the XPNO index are organized in an ascending sequence. Also, Figure A1.1 shows the INDSTATE index values (AZ, MA, MT) are organized in ascending sequence. This sequential organization of index values offers four advantages.

**1. Avoid Sorting:** If a WHERE-clause references an indexed-column (PNO or STATE), the system might be able to avoid a potentially expensive sort by using the index. Consider the following statement:

```
SELECT *
FROM PRESERVE
WHERE PNO >= 14
ORDER BY PNO
```

The system could access the XPNO index for PNO 14 and follow the pointer to retrieve the row that becomes the first row in the result table. Then, it could get the next index entry for PNO 40 and follow its pointer to retrieve the next row. Then, it could get the next index entry for PNO 80 and follow its pointer to retrieve the last row. This process *retrieves these rows in PNO sequence without sorting any data*. This method could be more efficient than scanning the table for rows with PNO values that are greater 14 and then sorting these rows. However, this method would be less efficient if the WHERE-clause specified PNO >= 2.

**2. Help Search a Large Index:** Because indexes are sorted, it is easier to search a large index. Again, consider the history book analogy. If the book were very large (say 10,000 pages), then its index would also be large (maybe 200 pages). Assume you wanted to search the index to find the pages that discuss John Muir. Would you begin searching the 200-page index by sequentially reading from the first page in this 200-page index? No! Because, you know that *the book index is sorted*, you would estimate that the name "Muir" is about halfway into the alphabet, and so you would jump into the middle of the 200-page index, and work forwards or backwards to find "Muir". (This search logic is similar to manually searching on last name in an old paper telephone book.) A database system can utilize similar logic to search a large database index.

**3. Index-Only Searches:** Assume the large history book has an index that only references the names of people who are mentioned in the book. This person-index would not have any entries for locations, wars, etc.

Consider the following questions.

What are the names of all people referenced in the history book?

What are the names of all people referenced in the history book whose last name begins with the letter M?

You could answer these questions by only reading the person-index because all desired information is in this index. There is no need to follow page pointers to read the book's content pages. A similar search pattern applies to database indexes. For example, assume you executed the following statement:

```
SELECT PNO
FROM PRESERVE
ORDER BY PNO
```

The system knows there is an index (XPNO) based on PNO which is the *only column* that is referenced in this SELECT-clause. Therefore, the system can access all PNO values by scanning the PNO values in the XPNO index without accessing the PRESERVE table.

Also, note that the PNO values in the XPNO index are stored in ascending sequence (Figure A2.1). Hence, the system would not need to sort the PNO values to satisfy the ORDER BY clause.

**4. Enforce Column Uniqueness:** A unique index can be used to enforce uniqueness on a unique column. For example, assume PRESERVE has 14 million rows. The system can use the XPNO index to enforce uniqueness on the PNO column. Whenever a user attempts to insert a new row into the PRESERVE table, the system (before inserting the row) determines if the XPNO index already has an entry which equals this new PNO value. If it does, the system rejects the insert operation. Directly accessing the smaller XPNO index is much faster than scanning the larger PRESERVE table. (If the DBA drops the XPNO index, the system will use some other method to maintain PNO uniqueness.)

### Incidental Sorts

Consider the following scenario for Sample Query 2.7 which produced an incidental sort.

```
SELECT PNO
FROM PRESERVE      [← No ORDER BY clause]
```

On Monday, the system does an index-only search on the XPNO index, and the rows are returned in PNO sequence. This produces the incidental sort.

On Tuesday, for some unspecified reason, the DBA drops the XPNO index.

On Wednesday, because the XPNO index is gone, the system scans the table to satisfy the same SELECT statement. On most systems, it is unlikely that this scan would return the rows in some sequence. Hence, no incidental sort.

### Index Costs

Indexes have three costs that usually discourage the creation of many indexes on a single table. Assume the XPNO and INDSTATE indexes have been created on the PRESERVE table.

**1. Indexes Penalize Update Operations:** This is usually the major disadvantage. Assume a user executes an INSERT statement to insert a new row into the PRESERVE table. (INSERT, UPDATE, and DELETE statements will be presented in Chapter 15.) In addition to inserting the new row, the system will automatically insert new entries into the XPNO index and the INDSTATE index. From the user's perspective, the system only executes one INSERT operation. But internally the system executes three operations: (i) insert a row into the PRESERVE table, (ii) insert an entry into the XPNO index, and (iii) insert an entry into the INDSTATE index. Similar observations apply to UPDATE and DELETE statements.

2. **Indexes use Disk Storage:** The index in the back of the history book occupies pages. Likewise, database indexes occupy disk storage. The good news is that disk storage continues to become very cheap. However, disk storage is not infinite, and it's not free.

3. **Index Reorganization:** Without explanation, we note that, after many INSERT, UPDATE, and DELETE operations, an index may become physically fragmented. This can have a negative impact on index efficiency. To address this problem, the DBA must periodically execute a utility program to reorganize the index.

### **Tradeoffs**

Fundamental Design Objective: Data retrieval benefits should exceed index costs.

Understanding index costs should help you sympathize with a DBA who rejects your (presumably reasonable) request to create a new index. For example, you might desire a new index that could help your SELECT statement that references a very large CUSTOMER table. Assume your SELECT statement will be executed once a day. However, the DBA knows that every day, approximately 5,000 INSERT, UPDATE, or DELETE operations are applied to the same CUSTOMER table. Creating the index means that you gain a retrieval benefit once a day while the INSERT/UPDATE/DELETE operations suffer a greater cost 5,000 times a day. Hence, for good reason, the DBA denies your request to create another index.

Special Case Scenario: Assume your SELECT statements are executed within a data warehouse application where all INSERT, UPDATE, and DELETE operations have been moved to an off-line system that "runs at midnight." Then, your daytime on-line system can have many indexes because index-update costs (except for disk storage) are significantly reduced.

### **Concluding Comments**

Indexing is not the only method that can provide direct access to rows in a table. Some systems (e.g., ORACLE) can utilize another direct-access method called "hashing." A discussion of hashing is beyond the scope of this book.

Thus far, we have not been definitive about how and when the system decides to use an index. Appendix A4 will address this important issue.

## Appendix 2B: Theory

### Tables do not have any predefined sequence.

The theoretical justification for this assertion is that, *ideally*, a table is a set, and sets do not have any predefined sequence. Consider the following set T containing all even integers between and including 2 and 10. Most math books would enumerate the elements of T in ascending sequence as shown below.

$$T = \{2, 4, 6, 8, 10\}$$

However, the following enumeration is also valid.

$$T = \{4, 10, 6, 8, 2\}$$

Also, when you enumerate a set of character-string values, you may be less inclined to list these values in sequence. Consider the values in the following STOOGES set. These values are not listed in alphabetical sequence.

$$\text{STOOGES} = \{\text{'MOE'}, \text{'LARRY'}, \text{'CURLY'}, \text{'JOE'}, \text{'SHEMP'}\}$$

### "First" N Rows?

Sample Query 2.9 asked you to display the "first" three rows from a result table. We enclosed "first" within quotation marks because, mathematically, there is no *first* element of a set.

Consider the following four of 32 possible enumerations of the above set T.

$$T = \{2, 4, 6, 8, 10\}$$
$$T = \{4, 10, 6, 8, 2\}$$
$$T = \{10, 4, 6, 8, 2\}$$
$$T = \{6, 10, 4, 2, 8\}$$

No specific element in set T is the first element.

Because a result table is a table, and a table is a set, a result table is also a set. Hence, there is no first element (first row) in a result table.



## Appendix 2C: Theory & Efficiency

### **"Theory is Practical" (C.J. Date)**

We describe two advantages associated with the absence of a predefined row sequence within a table.

- 1. Data Independence:** This is the primary advantage. For example, assume an applications developer knows that the PRESERVE table has a predefined PNO sequence. Then she might code a program containing logic that relies on this sequence. What happens if the DBA changes this row sequence? If the DBA changed this sequence such that rows are stored in PNAME sequence, then the logic of the program would fail. Therefore, this design change would *require the developer to modify her program*.

If desired, the DBA might specify some internal row sequence that is unknown to the application developers. Then the DBA could occasionally change this sequence without notifying the developers and asking them to change their programs. This is just one example of *data independence*, a very important concept to be described in later appendices.

- 2. Simplify INSERT Operations:** If there is no predefined row sequence, then the system is not required to maintain any physical row sequence. Whenever a user executes an INSERT statement, the system can insert the new row(s) into any convenient location within the table. (But sequence would still be maintained within the indexes.)

# Prohibiting Duplicate Rows:

## DISTINCT

**Base Tables:** Examination of the PRESERVE table does not reveal any duplicate rows. (No two rows show all corresponding column values equal to each other.) Each row is distinct.

By adhering to an excellent design principle, all base tables in this book contain distinct rows. For reasons to be described in Chapter 15, the system will not allow any INSERT or UPDATE operation to introduce duplicate rows into a base table.

**Result Tables:** Does the distinctness property apply to result tables? If you examine the result tables shown in the preceding two chapters, you will observe that all result tables contain distinct rows. However, in this chapter, Sample Queries 3.1 and 3.3 will illustrate that, in some circumstances, duplicate rows can appear in a result table. Because duplicate rows can be confusing, this chapter's other sample queries will remove them by specifying the keyword DISTINCT in the SELECT-clause.

## Duplicate Rows in a Single-Column Result Table

The following sample query illustrates that duplicate rows can appear in a result table.

**Sample Query 3.1:** Display the STATE column in every row of the PRESERVE table.

```
SELECT STATE
FROM PRESERVE
```

```
STATE
AZ
MT
AZ
MT
MA
MA
MA
MA
MA
MT
MT
AZ
MA
AZ
```

**Syntax & Logic:** Nothing new.

**Observation:** The result table contains duplicate rows. There are fourteen rows, but these rows only have three distinct STATE values (AZ, MA, and MT). The next sample query will specify the reserved word DISTINCT to prohibit duplicate rows.

### Exercises:

3A1. Retrieve every row in PRESERVE. Only display the FEE value for each row. (Do not attempt to remove duplicate rows.) Before you execute the SELECT statement for this exercise, ask yourself the following question. "Can duplicate values possibly appear in this result?"

3A2. Retrieve every row in PRESERVE. Only display the ACRES value for each row. (Do not attempt to remove duplicate rows.) Before you execute the SELECT statement for this exercise, ask yourself the following question. "Can duplicate values possibly appear in this result?"

## DISTINCT

The next sample query is similar to the previous sample query, but duplicate rows are not displayed in the result table. The keyword `DISTINCT` is specified in the `SELECT`-clause to achieve this objective.

**Sample Query 3.2:** Modify the previous sample query. Display the state code of every state that contains a nature preserve described in the `PRESERVE` table. Do not display duplicate rows.

```
SELECT DISTINCT STATE
FROM PRESERVE
```

```
STATE
AZ
MA
MT
```

**Syntax:** `DISTINCT` must immediately follow the `SELECT` keyword separated by one or more spaces.

**Logic:** The result does not show any duplicate `STATE` values.

**Observation:** The above statement does not specify an `ORDER BY` clause, but the result table is incidentally sorted. On most (but not all) systems, execution of this statement will produce an incidental sort. (Appendix 3A will describe why specification of `DISTINCT` may produce an incidental sort.)

### Exercise:

3B. Display all admission fees in the `PRESERVE` table. Do not display duplicate values.

## Duplicate Rows in a Multi-Column Result Table

The following sample query displays multiple columns. Such queries are less likely to produce duplicate rows. However, as this example illustrates, duplicate rows can occur.

**Sample Query 3.3:** Display the STATE and FEE values for every row in the PRESERVE table. Do not remove duplicate rows from the result table. Sort the result by FEE within STATE to make it easier to detect duplicate rows.

```
SELECT STATE, FEE
FROM PRESERVE
ORDER BY STATE, FEE
```

<u>STATE</u>	<u>FEE</u>
AZ	0.00
AZ	3.00
AZ	3.00
AZ	3.00
MA	0.00
MA	0.00
MA	0.00
MA	0.00
MA	0.00
MA	0.00
MA	0.00
MT	0.00
MT	0.00
MT	0.00
MT	0.00

**Syntax & Logic:** Nothing new.

**Observation:** There are many duplicate rows in this result table. For example, the second, third, and fourth rows in this result table are duplicates. Also, the last four rows are duplicates. Recall that a row is considered to be a duplicate if every column value in the row matches every corresponding column value in some other row.

### Important Exercise:

3C. Display the FEE and ACRES values for every row in the PRESERVE table. Before you execute the SELECT statement for this exercise, ask yourself the following question. "Can duplicate rows possibly appear in this result?" What know-your-data insights help you answer this question?

## DISTINCT

**Sample Query 3.4:** Modify the preceding sample query. Display distinct pairs of STATE and FEE values from the PRESERVE table. Sort the result by FEE within STATE.

```
SELECT DISTINCT STATE, FEE
FROM PRESERVE
ORDER BY STATE, FEE
```

<u>STATE</u>	<u>FEE</u>
AZ	0.00
AZ	3.00
MA	0.00
MT	0.00

**Syntax:** Nothing new. DISTINCT can only appear once in a SELECT-clause. Sometimes a rookie user might incorrectly code multiple DISTINCT keywords as illustrated below.

```
SELECT DISTINCT COL1, COL2, DISTINCT COL3    → Error
FROM SOMETABLE
```

**Logic:** In the above statement, specifying DISTINCT twice might imply an invalid attempt to prohibit duplicate values in COL1 and COL3, but allow duplicate values in COL2. This is not consistent with the logic of DISTINCT which *prohibits duplicate rows*, not just duplicate values in some columns.

### Exercise:

- 3D. Display the FEE and ACRES values for every row in the PRESERVE table. Do not display duplicate rows in the result table.
- 3E. Optional Exercise: Remove the ORDER BY clause from the above Sample Query 3.4 such that it looks like:

```
SELECT DISTINCT STATE, FEE
FROM PRESERVE
```

Execute this statement. Most likely you will observe that the result table is incidentally sorted. If this occurs, the first-level sort column could be either the STATE column or the FEE column.

## Know-Your-Data

Because you know that, within the PRESERVE table, the PNO column is unique, you can be confident that duplicate rows cannot not appear in either of the following circumstances.

- The SELECT-clause specifies PNO.
- The result table contains just one row because the WHERE-clause compares on the PNO column using an equals (=) comparison operator. (E.g., WHERE PNO = 40)

**Sample Query 3.5:** Display the PNO and ACRES values of nature preserves located in Montana. Do not display duplicate rows.

```
SELECT PNO, ACRES
FROM PRESERVE
WHERE STATE = 'MT'
```

PNO	ACRES
3	680
40	121
1	1130
2	15000

**Logic:** You do not need to specify DISTINCT because the SELECT-clause references PNO, a unique column. (An argument can be made that every SELECT-clause should specify DISTINCT. See Appendix 3A.)

### Exercise:

3F. The following two statements return the same rows. Will these rows be in the same row sequence? Answer: Yes, No, or Maybe.

```
SELECT PNO
FROM PRESERVE;
```

```
SELECT DISTINCT PNO
FROM PRESERVE;
```

## Summary

This chapter introduced the optional `DISTINCT` keyword that is used to prohibit duplicate rows from appearing in a result table. Our generic `SELECT` statement is now extended to include `DISTINCT` in the `SELECT`-clause.

```
SELECT [DISTINCT] column-names
FROM   table-name
[WHERE condition]
[ORDER BY column-name(s)]
```

Specifying `DISTINCT` is simple. Knowing-your-data is the challenge. If you do not like duplicate rows, you could specify `DISTINCT` in all your `SELECT` statements. (The following Appendix 3A will comment on this issue.)

Also, your SQL reference manual will show

```
SELECT [ALL | DISTINCT] column-names
FROM . . .
```

`ALL` is the default, versus the optional `DISTINCT`. The `ALL` keyword allows duplicate rows to appear in a result table. Most practitioners never code `ALL` in a `SELECT` statement, as illustrated by every `SELECT` statement in this book.

## Summary Exercises

The following exercises reference the `EMPLOYEE` table.

3G. Display all `DNO` values in the `EMPLOYEE` table. Do not display duplicate values.

3H. Execute each of the following statements. Examine the result tables and make relevant observations.

```
SELECT DNO, SALARY
FROM EMPLOYEE;
```

```
SELECT DISTINCT DNO, SALARY
FROM EMPLOYEE;
```

```
SELECT DISTINCT DNO, SALARY
FROM EMPLOYEE
ORDER BY DNO, SALARY;
```



## Appendix 3A: Efficiency

Specifying `DISTINCT` may force the system to perform additional work to detect and remove any duplicate rows from a result table. Two methods for detecting and removing duplicate rows are described below.

1. **System does sort operation:** The system must detect duplicate rows before it can remove them. Therefore, `DISTINCT` may encourage the system to sort an intermediate result to facilitate detecting duplicate rows. In this circumstance, sort costs (as described in Appendix A2) would apply. Also, this sort process might indirectly produce a result table that is incidentally sorted.
2. **System references an index:** Consider the following statement for Sample Query 3.2.

```
SELECT DISTINCT STATE
FROM PRESERVE
```

Assuming the `INDSTATE` index (Figure A1.1) is available, the system could reference this index to extract just the distinct index values ('AZ', 'MA', 'MT'). This would be an index-only search. Also, because the index values are stored in ascending sequence, the result table would probably be displayed in an incidentally sorted sequence.

Because `DISTINCT` may require extra work, most practitioners do not specify `DISTINCT` in every `SELECT` statement. For example, consider the following statement.

```
SELECT DISTINCT PNO
FROM PRESERVE
```

Question: Why might we want to remove `DISTINCT` from the above statement?

Answer: We know that the `PNO` column is unique. Hence, we can deduce that `DISTINCT` is superfluous. Likewise, because the system knows that `PNO` is unique, the system should also deduce that `DISTINCT` is superfluous. Therefore, the system would (probably) not do any extra work to remove duplicate rows that cannot possibly occur.

**Always Specify `DISTINCT`?** A relational purist could argue that you should specify `DISTINCT` in every `SELECT` statement. From a logical perspective this is a good idea. However, perhaps unfortunately, this is not a standard practice among practitioners.

## Appendix 3B: Theory

### "Theory is Practical" (C.J. Date):

**Set Theory:** A set cannot contain duplicate values. Hence, the following lists of values are not valid sets.

$N = \{6, 3, 3, 3, 12\}$

$S = \{\text{'MOE'}, \text{'LARRY'}, \text{'LARRY'}, \text{'LARRY'}, \text{'MOE'}\}$

Because a table should be a set, all tables, *including result tables*, should not contain duplicate rows.

**Cognitive Psychology:** Duplicate rows can be confusing. Consider the result tables shown for the following two SELECT statements.

Statement-1: SELECT STATE, FEE FROM PRESERVE

Statement-2: SELECT DISTINCT STATE, FEE FROM PRESERVE

Result-1	
STATE	FEE
AZ	3.00
MT	0.00
AZ	0.00
MT	0.00
MA	0.00
MA	0.00
MA	0.00
MA	0.00
MA	0.00
MT	0.00
MT	0.00
AZ	3.00
MA	0.00
AZ	3.00

Result-2	
STATE	FEE
AZ	0.00
AZ	3.00
MA	0.00
MT	0.00

Many business users would find Result-1, with duplicate rows, to be confusing. They might find Result-2, without duplicate rows, to be more understandable. However, most users would probably prefer the following Result-3 that displays a count of duplicate rows.

Result-3		
STATE	FEE	COUNT
AZ	0.00	1
AZ	3.00	3
MA	0.00	6
MT	0.00	4

Chapter 9 will show how to produce Result-3.

**Controversy:** We begin with two observations.

Regarding Base Tables: Although it is very unusual, a base table could have duplicate rows. For example, the CREATE TABLE statement that created the PRESERVE table (Figure 0.2) could have omitted the *optional* keyword UNIQUE.

Regarding Result Tables: A result table may contain duplicate rows. (Review Sample Queries 3.1 and 3.3.)

These observations illustrate a SQL feature where *SQL has not been completely faithful to its mathematical heritage*. This allows us to *speculate* on two *hypothetical* versions of SQL.

1. **Ideal-SQL:** This version of SQL conforms to the Relational Model. All CREATE TABLE statements must specify some unique column (or unique combination of columns). Hence, duplicate rows would be prohibited from all base tables. Also, duplicate rows would be automatically removed from all intermediate and final result tables.
2. **Almost-Ideal-SQL:** This version of SQL would enforce default actions that conform to Ideal-SQL. However, Almost-Ideal-SQL would allow a user to explicitly override a specific default action. For example, SELECT DISTINCT would be the default. However, a user could allow duplicate rows in a result table by specifying SELECT ALL. Also, if a user's CREATE TABLE statement did not designate a unique column, then this statement must specify some other clause such as "ALLOW DUPLICATE ROWS". In general, Almost-Ideal-SQL would, by default conform to Ideal-SQL (which conforms to the Relational Model). However, unlike Ideal-SQL, it would allow user-specified deviations from Ideal-SQL.

**Real-World-SQL:** Some database experts believe that Real-World-SQL should conform to Ideal-SQL (which conforms to the Relational Model). However, other experts would accept some version of Almost-Ideal-SQL.

*Unfortunately*, regarding the matter of uniqueness, Real-World-SQL is not Almost-Ideal-SQL. In Real-World-SQL, designating a unique column in a CREATE TABLE statement is optional; and specifying DISTINCT in a SELECT statement is optional. These options are not consistent with Almost-Ideal-SQL.

Comment: Codd and Date have described many practical problems associated with duplicate rows. This is just one reason why they were very disappointed with Real-World SQL.

# Boolean Connectors:

## AND - OR - NOT

This chapter emphasizes the **know-your-logic** aspect of coding correct SQL statements. Rookie users must read this important chapter. Application developers who already understand Boolean logic can “fly thru” this chapter, briefly scanning the sample queries.

Previous sample queries illustrated WHERE-clauses that specified simple-conditions. This chapter introduces compound-conditions that contain Boolean connectors (AND, OR, and NOT). Here you will see WHERE-clauses that look like:

- WHERE condition-1 **AND** condition-2
- WHERE condition-1 **OR** condition-2
- WHERE **NOT** condition

This chapter is organized into three sections.

Section-A: Fundamental Boolean Connectors

Section-B: Mixing Different Boolean Connectors

Section-C: Logically Equivalent WHERE-Clauses

## A. Fundamental Boolean Connectors

### AND Connector

The following sample query asks the system to select a row if it matches both of the specified conditions.

**Sample Query 4.1:** Display all information about any nature preserve that is located in Arizona and has no admission fee (i.e., fee is zero dollars).

```
SELECT *
FROM PRESERVE
WHERE STATE = 'AZ' AND FEE = 0.00
```

PNO	PNAME	STATE	ACRES	FEE
7	MULESHOE RANCH	AZ	49120	0.00

**Syntax:** This WHERE-clause specifies two simple-conditions that are connected by the AND connector. These conditions are:

- STATE = 'AZ'
- FEE = 0.00

**Logic:** A selected row must match both conditions. This result table shows that only one row matches both conditions. The logic for AND is represented by the following truth table.

**Truth Table for AND:** The following table truth table illustrates the precise definition of AND. Consider the four possible True/False values for two arbitrary conditions, C1 and C2. Observe that C1 AND C2 is True only if both conditions are True (T). Otherwise, C1 AND C2 evaluate to False (F).

C1	C2	C1 AND C2
T	T	T
T	F	F
F	T	F
F	F	F

The following sample query illustrates two conditions that reference the same column (ACRES).

**Sample Query 4.2:** Display the PNO, PNAME, and ACRES values for any nature preserve with an ACRES value that is strictly between 90 and 1200.

```
SELECT *  
  
FROM PRESERVE  
  
WHERE ACRES > 90 AND ACRES < 1200
```

PNO	PNAME	ACRES
5	HASSAYAMPA RIVER	660
3	DANCING PRAIRIE	680
40	SOUTH FORK MADISON	121
9	DAVID H. SMITH	830
12	MOUNT PLANTAIN	730
1	COMERTOWN PRAIRIE	1130
80	RAMSEY CANYON	380

**Syntax:** The column-name (ACRES) must be specified in both conditions. The following WHERE-clause is invalid and will cause an error.

```
WHERE ACRES > 90 AND < 1200 → Error
```

**Logic:** This WHERE-clause selects rows with an ACRES value that is *strictly* greater than 90 and *strictly* less than 1200. Note that rows with ACRES values of 90 (HOFT FARM) and 1200 (PAPAGONIA-SONOITA CREEK) were not selected.

**Comment:** The following Chapter 5 will introduce the BETWEEN keyword which means “between and including.” Use of BETWEEN would be incorrect in the current example.

**Exercise:**

- 4A. Display all information about any nature preserve in Montana that is smaller than 1,000 acres.
- 4B. Display all information about any nature preserve that has an ACRES value between and including 1200 and 15000.

## Multiple ANDs

A compound-condition can contain more than two simple-conditions. The following sample query illustrates four simple-conditions that are connected with AND. In this example, a given row will be selected if it matches all four conditions.

**Sample Query 4.3:** Display the PNO, PNAME, FEE and ACRES values of all nature preserves that are located in Arizona, have a non-zero admission fee, and are greater than or equal to 660 acres, and less than or equal to 1200 acres.

```
SELECT PNO, PNAME, FEE, ACRES
FROM PRESERVE
WHERE STATE = 'AZ'
AND FEE <> 0.00
AND ACRES >= 660
AND ACRES <= 1200
```

PNO	PNAME	FEE	ACRES
5	HASSAYAMPA RIVER	3.00	660
6	PAPAGONIA-SONOITA CREEK	3.00	1200

**Syntax & Logic:** Nothing new. This WHERE-clause AND-connects four conditions. Any row that matches all four conditions will be displayed.

For all practical purposes there is no limit on the number of conditions that can be specified within a WHERE-clause.

### Exercise:

4C. Display all information about any nature preserve that is located in Montana, does not have an admission fee, and is greater than 10,000 acres.

## OR Connector

When two conditions are connected with OR, a row is selected if it matches either one or both of the specified conditions.

**Sample Query 4.4:** Display the PNO, PNAME, and STATE values of all nature preserves that are located in Arizona or Montana.

```
SELECT PNO, PNAME, STATE
FROM PRESERVE
WHERE STATE = 'AZ'
OR STATE = 'MT'
```

PNO	PNAME	STATE
5	HASSAYAMPA RIVER	AZ
3	DANCING PRAIRIE	MT
7	MULESHOE RANCH	AZ
40	SOUTH FORK MADISON	MT
1	COMERTOWN PRAIRIE	MT
2	PINE BUTTE SWAMP	MT
80	RAMSEY CANYON	AZ
6	PAPAGONIA-SONOITA CREEK	AZ

**Syntax:** The column-name must be explicitly specified in each simple-condition. Hence the following WHERE-clause is *invalid and will cause an error*.

```
WHERE STATE = 'AZ' OR 'MT' → Error
```

**Logic:** This WHERE-clause selects a row if it has a STATE value of AZ, or if it has a STATE value of MT. This logic is embodied in the following truth table.

**Truth Table for OR:** The following table truth table illustrates the precise definition of OR. Consider two arbitrary conditions, C1 and C2. Observe that C1 OR C2 is False if both conditions are False. Otherwise, C1 OR C2 evaluates to True.

C1	C2	C1 OR C2
T	T	T
T	F	T
F	T	T
F	F	F



## OR Means “Inclusive OR”

The next sample query illustrates that OR selects a row if it matches both conditions. This logic denotes an inclusive (versus exclusive) interpretation of OR.

**Sample Query 4.5:** Display the PNAME, ACRES, and STATE value of any preserve that is located in Arizona or has more than 1000 acres.

```
SELECT PNAME, ACRES, STATE
FROM PRESERVE
WHERE STATE = 'AZ'
OR ACRES > 1000
```

PNAME	ACRES	STATE
HASSAYAMPA RIVER	660	AZ
MULESHOE RANCH	49120	AZ
COMERTOWN PRAIRIE	1130	MT
PINE BUTTE SWAMP	15000	MT
RAMSEY CANYON	380	AZ
PAPAGONIA-SONOITA CREEK	1200	AZ

**Logic - OR means Inclusive-OR:** This SELECT statement will display any row which has a STATE value of AZ or an ACRES value greater than 1000. Observe that all Arizona nature preserves are selected, regardless of their acreage; and, all preserves over 1000 acres are selected, regardless of their location. We emphasize that any row matching both conditions (e.g., MULESHOE RANCH and PAPAGONIA-SONOITA CREEK) is *included* in the result table.

**Exclusive-OR (XOR):** Careful! Sometimes people casually use the word “or” to imply the Exclusive-OR (XOR). For example, a parent might tell a child that “you can have a slice of pie *OR* a slice of cake.” Presumably, the parent does not mean that the child can eat both pie and cake. Exclusive-OR implies that the pie-and-cake option is *excluded*. (Summary Exercise 4U will show how to code an Exclusive-OR condition.)

### Exercise:

- 4D. Display all information about nature preserves located in Montana or Massachusetts.
- 4E. Display all information about any nature preserve located in Montana or any preserve that is less than 1,000 acres.

## Multiple ORs

As with the AND connector, it is possible to connect multiple conditions by coding multiple OR connectors. The following SELECT statement illustrates five simple-conditions that are OR-connected.

**Sample Query 4.6:** Display the PNO and PNAME values for any nature preserve that has a PNO value equal to any of the values {3, 4, 7, 12, 40}

```
SELECT PNO, PNAME
FROM PRESERVE
WHERE PNO = 3
OR     PNO = 4
OR     PNO = 7
OR     PNO = 12
OR     PNO = 40
```

<u>PNO</u>	<u>PNAME</u>
3	DANCING PRAIRIE
7	MULESHOE RANCH
40	SOUTH FORK MADISON
12	MOUNT PLANTAIN

**Syntax:** For all practical purposes, there is no limit on the number of conditions that can be OR-connected. Note that the column-name (PNO) must be explicitly specified in each condition. Both of the following abbreviated WHERE-clauses are invalid.

```
WHERE PNO = 3 OR 4 OR 7 OR 12 OR 40      → Error
```

```
WHERE PNO = (3, 4, 7, 12, 40)          → Error
```

**Logic:** The nature preserves with PNO values of 3, 7, 12, 40 were selected. No nature preserve has a PNO value of 4.

**Comment:** Chapter 5 will introduce the IN operator that offers a more concise method for coding this statement.

## NOT

Previous sample queries specified conditions that identified, in a positive sense, the rows that you wanted to retrieve. Sometimes, it is more convenient to identify those rows that you do *not* want to retrieve.

**Sample Query 4.7:** Display the PNAME and STATE values of all nature preserves that are not located in Massachusetts.

```
SELECT PNAME, STATE
FROM PRESERVE
WHERE NOT STATE = 'MA'
```

<u>PNAME</u>	<u>STATE</u>
HASSAYAMPA RIVER	AZ
DANCING PRAIRIE	MT
MULESHOE RANCH	AZ
SOUTH FORK MADISON	MT
COMERTOWN PRAIRIE	MT
PINE BUTTE SWAMP	MT
RAMSEY CANYON	AZ
PAPAGONIA-SONOITA CREEK	AZ

**Syntax:** The NOT operator can be placed before any condition. This example uses NOT to negate a simple-condition. Sample Query 4.10 will use NOT to negate a compound-condition.

**Common Error:** Placing NOT immediately before a comparison operator will cause an error. The following WHERE-clause is *invalid*:

```
WHERE STATE NOT = 'MA'           → Error
```

**Logic:** Obvious. See the following truth table.

**Truth Table for NOT:** The following truth table illustrates the precise definition of NOT.

C1	NOT C1
T	F
F	T

### Exercises:

- 4F. Display the preserve number and name of any nature preserve having an admission fee that is not equal to zero. Use the keyword NOT in your solution.
- 4G. Same as the preceding exercise: Specify a not-equal symbol in your WHERE-clause.

## **B. Mixing Different Boolean Connectors**

Sample queries in this section will specify compound-conditions that include different Boolean connectors.

Sample Query 4.8 will specify the following compound-condition which includes both the OR and AND logical connectors:

```
ACRES > 1000 OR (STATE = 'AZ' AND FEE = 3.00)
```

When a compound-condition contains different logical connectors, the system must determine which logical operation should be evaluated first. This is important because the order of evaluation is significant. In the above compound-condition, the parentheses indicate that the user wants to evaluate the AND operation before the OR operation.

Sample Query 4.9 will specify the following compound-condition.

```
(ACRES > 1000 OR STATE = 'AZ') AND FEE = 3.00
```

Here, the parentheses indicate that the OR operation is evaluated before the AND operation.

Observe that both of the above compound-conditions specify the same three simple-conditions. The only difference is the placement of the parentheses. Sample Queries 4.8 and 4.9 will show *different* result tables because the order of operations, as specified by the parentheses, makes a significant difference.

What if a user does not specify any parentheses as shown below?

```
ACRES > 1000 OR STATE = 'AZ' AND FEE = 3.00
```

Then the system would evaluate the compound-condition according to a hierarchy of logical operations that will be described later in this section. (Preview: The system would evaluate the AND before the OR.)

**Sample Query 4.8:** Display the PNAME, STATE, FEE, and ACRES values of any nature preserve with more than 1000 acres, or any Arizona preserve with a \$3.00 admission fee.

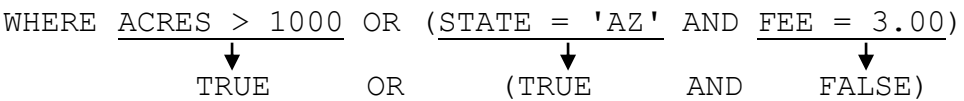
```
SELECT PNAME, STATE, FEE, ACRES
FROM PRESERVE
WHERE ACRES > 1000 OR (STATE = 'AZ' AND FEE = 3.00)
```

PNAME	STATE	FEE	ACRES
HASSAYAMPA RIVER	AZ	3.00	660
MULESHOE RANCH	AZ	0.00	49120
COMERTOWN PRAIRIE	MT	0.00	1130
PINE BUTTE SWAMP	MT	0.00	15000
RAMSEY CANYON	AZ	3.00	380
PAPAGONIA-SONOITA CREEK	AZ	3.00	1200

**Logic:** For example, consider the PRESERVE row shown below.

PNO	PNAME	STATE	ACRES	FEE
7	MULESHOE RANCH	AZ	49120	0.00

The evaluation for this row is:



Therefore, the evaluation for this row is:

$$\begin{aligned} & \text{TRUE OR (TRUE AND FALSE)} = \\ & \text{TRUE OR (FALSE)} = \\ & \text{TRUE} \end{aligned}$$

The final evaluation of TRUE implies this row is selected and appears in the above result table.

Similar "logical calculations" are applied to all rows in the PRESERVE table to produce to above result table.



## “Display Every Row Except...”

This following statement illustrates the formulation of a compound-condition to identify those rows that we do *not* want to display.

**Sample Query 4.10:** Display the PNO, STATE, and FEE values of every nature preserve *except* for those preserves that are located in Arizona and have a \$3.00 admission fee.

```
SELECT PNO, STATE, FEE
FROM PRESERVE
WHERE NOT (STATE = 'AZ' AND FEE = 3.00)
```

PNO	STATE	FEE
3	MT	0.00
7	AZ	0.00
40	MT	0.00
14	MA	0.00
13	MA	0.00
9	MA	0.00
11	MA	0.00
12	MA	0.00
1	MT	0.00
2	MT	0.00
10	MA	0.00

**Logic:** The logic is straightforward. We first code a condition to identify the rows we do *not* want. This is:

```
STATE = 'AZ' AND FEE = 3.00
```

Then we negate this condition by enclosing it within parentheses and placing a NOT in front of it.

```
NOT (STATE = 'AZ' AND FEE = 3.00)
```

**Exercises:**

- 4H. Display the preserve number and name of those nature preserves that do not have an admission fee of \$3.00 and do not have a fee of \$10.00.
- 4I. Display all information about any nature preserve located in Arizona that does not have an admission fee, or any preserve that is smaller than 100 acres (regardless of its STATE and FEE values).
- 4J. Display all information about any nature preserve that is smaller than 1,000 acres, and has an admission fee of zero dollars or is located in Arizona.
- 4K. Select all information about any nature preserve with an admission fee that is not greater than zero, or any other preserve, regardless of its fee, that is located in Montana and is larger than 1,000 acres.
- 4L. Display all information about every nature preserve except those Montana preserves without an admission fee.



## Hierarchy of Logical Operators

If a WHERE-clause contains three or more conditions, and *this WHERE-clause does not contain parentheses*, the system uses the following default hierarchy of operations:

- NOTs are evaluated first
- ANDs are evaluated second
- ORs are evaluated third

This is the same hierarchy that applies to most programming languages (e.g., JAVA, COBOL).

Example-1: Consider the following WHERE-clause that does not specify parentheses.

```
WHERE ACRES > 1000 OR STATE = 'AZ' AND FEE = 3.00
```

Because this compound-condition specifies different logical operators, and parentheses are not specified, the default hierarchy comes into play. The AND-condition is evaluated first as illustrated by the parentheses.

```
WHERE ACRES > 1000 OR (STATE = 'AZ' AND FEE = 3.00)
```

Note: This WHERE-condition was specified in Sample Query 4.8.

Example-2: Consider the following WHERE-clause that does not specify parentheses.

```
WHERE NOT FEE = 3.00 OR ACRES > 1000 AND STATE = 'AZ'
```

Because this compound-condition specifies different logical operators, and parentheses are not specified, the default hierarchy comes into play. The NOT-condition is evaluated first, the AND-condition is evaluated second, and the OR-condition is evaluated last, as illustrated below.

```
WHERE (NOT FEE = 3.0) OR (ACRES > 1000 AND STATE = 'AZ')
```

### Strong Recommendation:

Always use parentheses to specify the desired order of evaluation.

## An Ugly WHERE-Clause

The following WHERE-clause specifies three different Boolean operators (AND, OR, NOT), but it does not specify any parentheses. This example requires you to understand the Boolean hierarchy of "first NOT, then AND, then OR."

**Sample Query 4.11:** Display PNAME, STATE, FEE, and ACRES values about all preserves that are smaller than 50 acres, or any preserve that has a \$3.00 admission fee and is not located in Massachusetts.

```
SELECT PNAME, STATE, FEE, ACRES
FROM PRESERVE
WHERE ACRES < 50
OR FEE = 3.00
AND NOT STATE = 'MA'
```

<u>PNAME</u>	<u>STATE</u>	<u>FEE</u>	<u>ACRES</u>
HASSAYAMPA RIVER	AZ	3.00	660
TATKON	MA	0.00	40
MIACOMET MOORS	MA	0.00	4
RAMSEY CANYON	AZ	3.00	380
PAPAGONIA-SONOITA CREEK	AZ	3.00	1200

**Logic:** Most users would prefer the following equivalent WHERE-clause with parentheses.

```
WHERE ACRES < 50 OR (FEE = 3.00 AND (NOT STATE = 'MA'))
```

This WHERE-clause specifies a nested pair of parentheses. Specifying a not-equal symbol (<>) may enhance readability.

```
WHERE ACRES < 50 OR (FEE = 3.00 AND STATE <> 'MA')
```

### Exercise:

4M. Consider the following modified WHERE-clauses (*without parentheses*) for Sample Queries 4.8, 4.9 and 4.10. Which of the following modified WHERE-clauses will satisfy the specified query objectives?

4.8 WHERE ACRES > 1000 OR STATE = 'AZ' AND FEE = 3.00

4.9 WHERE ACRES > 1000 OR STATE = 'AZ' AND FEE = 3.00

4.10 WHERE NOT STATE = 'AZ' AND FEE = 3.00

## C. Logically Equivalent WHERE-Clauses

Exercise 4G required you to understand that the following WHERE-clauses are logically equivalent.

```
WHERE NOT STATE = 'MA'
```

```
WHERE STATE <> 'MA'
```

This section continues the theme of logical equivalency by revisiting three sample queries.

Sample Query 4.8. The WHERE-clause for this query is:

```
WHERE ACRES > 1000 OR (STATE = 'AZ' AND FEE = 3.00)
```

An alternative WHERE-clause is shown below. Convince yourself that this WHERE-clause is equivalent to the above WHERE-clause. (If this is problematic, reading the following pages should help.)

```
WHERE (ACRES > 1000 OR STATE = 'AZ')
      AND
      (ACRES > 1000 OR FEE = 3.00)
```

Analyzing the above WHERE-clauses raises four questions.

1. Are these two WHERE-clauses really equivalent? How do we know these WHERE-clauses will retrieve the same rows? Considering this question will take us into the game of “logical gymnastics” that will be discussed on the following pages.
2. Does this “equivalent WHERE-clause business” really matter? Assuming a given WHERE-clause is correct, why should you play a game of logical gymnastics to formulate an alternative WHERE-clause? Good question! In most circumstances, you do not need to contemplate other equivalent WHERE-clauses. *However*, someday you may have to analyze and possibly change a WHERE-clause written by another user whose logical mindset differs from yours.
3. Which WHERE-clause is friendlier? This question takes us into cognitive psychology. Other than making an occasional observation, we do not offer strong opinions on this matter.
4. Which WHERE-clause is more efficient? Both statements should execute with the same efficiency. However, sometimes, a WHERE-clause may be more efficient than another equivalent WHERE-clause. (Appendices 4A, 4B, and 4C will address this matter.)

Sample Query 4.9. The WHERE-clause for this query is:

```
WHERE (ACRES > 1000 OR STATE = 'AZ') AND FEE = 3.00
```

An alternative WHERE-clause is shown below. Convince yourself that it is equivalent to the above WHERE-clause.

```
WHERE (FEE = 3.00 AND ACRES > 1000)
      OR
      (FEE = 3.00 AND STATE = 'AZ')
```

Author Comment: Working backwards from this alternative WHERE-clause could encourage me to re-articulate the query objective as:

Display the PNAME, STATE, FEE, and ACRES values of any nature preserve that has a \$3.00 admission fee and is larger than 1000 acres, or any nature preserve that has a \$3.00 admission fee and is located in Arizona.

This revised articulation, while it may be less concise, may be a simpler description of the query objective.

Sample Query 4.10. The WHERE-clause for this query is:

```
WHERE NOT (STATE = 'AZ' AND FEE = 3.00)
```

Two alternative WHERE-clauses are shown below. Convince yourself that these clauses are equivalent to the above WHERE-clause.

```
WHERE (NOT STATE = 'AZ') OR (NOT FEE = 3.00)
```

```
WHERE (STATE <> 'AZ') OR (FEE <> 3.00)
```

**Logical Gymnastics:** In practice, most users rarely have to play this game. Their query objectives usually require them to code relatively simple WHERE-clauses, and then, maybe once a year, they encounter a logically complex condition. In this circumstance, they “muddle through” by testing many combinations of data values to produce a correct WHERE-clause. This approach may be OK. However!

Author Comment: When teaching SQL classes. I do not have enough time to present a comprehensive discussion of Boolean Logic. My sample queries are designed to present basic logic and “raise the anxiety level” regarding complex logical conditions. I advise the students, upon encountering a complex compound-condition, to “pay attention,” perform robust testing, and review this chapter.

## “Logical Gymnastics” via Laws of Logic

Laws of logic allow you to “mechanically” transform one logical expression into another equivalent expression. To illustrate some examples, we present two laws of logic, the Distributive Laws and De Morgan’s Laws. (Appendix 4B will elaborate on these and other laws of logic.)

### Distributive Laws

#### 1. Distribute OR over AND

$$C1 \text{ OR } (C2 \text{ AND } C3) = (C1 \text{ OR } C2) \text{ AND } (C1 \text{ OR } C3)$$

If a compound-condition fits the form of the left-side of the equation, you can rewrite the condition to fit the form of the right-side of the equation; and, vice versa.

Consider the compound-condition in Sample Query 4.8.

```
ACRES > 1000 OR (STATE = 'AZ' AND FEE = 3.00)
```

This compound-condition conforms to:  $C1 \text{ OR } (C2 \text{ AND } C3)$

Hence it can be rewritten as  $(C1 \text{ OR } C2) \text{ AND } (C1 \text{ OR } C3)$

```
(ACRES > 1000 OR STATE = 'AZ') AND (ACRES > 1000 OR FEE = 3.00)
```

#### 2. Distribute AND over OR

$$C1 \text{ AND } (C2 \text{ OR } C3) = (C1 \text{ AND } C2) \text{ OR } (C1 \text{ AND } C3)$$

Again, if a compound-condition fits the form of the left-side of the equation, you can rewrite the condition to fit the form of the right-side of the equation; and, vice versa.

Consider the compound-condition in Sample Query 4.9.

```
(ACRES > 1000 OR STATE = 'AZ') AND FEE = 3.00
```

Before applying the distributive law, we rewrite this compound-condition as:

```
FEE = 3.00 AND (ACRES > 1000 OR STATE = 'AZ')
```

This condition now conforms to:  $C1 \text{ AND } (C2 \text{ OR } C3)$

Hence it can be rewritten as  $(C1 \text{ AND } C2) \text{ OR } (C1 \text{ AND } C3)$

```
(FEE = 3.00 AND ACRES > 1000) OR (FEE = 3.00 AND STATE = 'AZ')
```

## De Morgan's Laws

1.  $\text{NOT (C1 AND C2)} = (\text{NOT C1}) \text{ OR } (\text{NOT C2})$

If you encounter a compound-condition that fits the form of  $\text{NOT (C1 AND C2)}$ , you can produce an equivalent condition by moving (distributing) the NOT inside the parentheses (place NOT before C1, and place NOT before C2), and then *replace AND with OR*. Likewise, if a compound-condition fits the form of the right side of the equation, you can modify it to fit the form of the left side of the equation.

2.  $\text{NOT (C1 OR C2)} = (\text{NOT C1}) \text{ AND } (\text{NOT C2})$

If you encounter a compound-condition that fits the form of  $\text{NOT (C1 OR C2)}$ , you can produce an equivalent condition by moving (distributing) the NOT inside the parentheses (place NOT before C1, and place NOT before C2), and then *replace OR with AND*. Likewise, if a compound-condition fits the form of the right side of the equation, you can modify it to fit the form of the left side of the equation.

**Example:** The compound-condition in Sample Query 4.10 fits De Morgan's first law:  $\text{NOT (C1 AND C2)}$

$\text{NOT (STATE = 'AZ' AND FEE = 3.00)}$

Distributing NOT to each individual condition and replacing AND with OR produces:

$(\text{NOT STATE = 'AZ'}) \text{ OR } (\text{NOT FEE = 3.00})$

Optionally, this condition could be rewritten as:

$(\text{STATE} <> \text{'AZ'}) \text{ OR } (\text{FEE} <> 3.00)$

And, optionally, removing the superfluous parentheses yields:

$\text{STATE} <> \text{'AZ'} \text{ OR FEE} <> 3.00$

Note: The above logical deductions yield four equivalent compound-conditions.

1.  $\text{NOT (STATE = 'AZ' AND FEE = 3.00)}$
2.  $(\text{NOT STATE = 'AZ'}) \text{ OR } (\text{NOT FEE = 3.00})$
3.  $(\text{STATE} <> \text{'AZ'}) \text{ OR } (\text{FEE} <> 3.00)$
4.  $\text{STATE} <> \text{'AZ'} \text{ OR FEE} <> 3.00$

[Appendix 4B where will present other logical laws. Appendices 4A and 4C describe how a database optimizer capitalizes on these laws.]

## Optional Exercises

The Distributed Laws apply to the following exercises.

4N1. Are the following WHERE-clauses logically equivalent?

```
WHERE STATE = 'MA' AND (ACRES > 1000 OR FEE = 0.0)
```

```
WHERE (STATE = 'MA' AND ACRES > 1000)
      OR
      (STATE = 'MA' AND FEE = 0.0)
```

4N2. Are the following WHERE-clauses logically equivalent?

```
WHERE STATE = 'MA' OR (ACRES > 1000 AND FEE = 0.0)
```

```
WHERE (STATE = 'MA' OR ACRES > 1000)
      AND
      (STATE = 'MA' OR FEE = 0.0)
```

De Morgan's Laws apply to the following exercises.

4O1. Are the following WHERE-clauses logically equivalent?

```
WHERE NOT (ACRES < 50 AND STATE = 'MA')
```

```
WHERE NOT ACRES < 50 AND NOT STATE = 'MA'
```

4O2. Are the following WHERE-clauses logically equivalent?

```
WHERE NOT (ACRES < 50 AND STATE = 'MA')
```

```
WHERE ACRES >= 50 OR STATE <> 'MA'
```

4P. Are the following WHERE-clauses logically equivalent?

```
WHERE NOT (ACRES < 50 OR STATE = 'MA')
```

```
WHERE NOT ACRES < 50 OR NOT STATE = 'MA'
```

4Q. Are the following WHERE-clauses logically equivalent?

```
WHERE NOT (ACRES < 50 OR STATE = 'MA')
```

```
WHERE NOT ACRES < 50 AND NOT STATE = 'MA'
```

## Articulating Query Objectives

Assume the “big boss” or a very important client sends you an email with the following query objective.

Display the PNO, PNAME, and STATE values of all preserves in Arizona and Montana.

If you take this query objective as stated, you might code the following SELECT statement.

```
SELECT PNO, PNAME, STATE
FROM PRESERVE
WHERE STATE = 'AZ' AND STATE = 'MT'
```

This would produce a “no hit,” and you suspect that something is wrong. So, you decide to become a mind reader.

You (*maybe* correctly) think that this person intended to say “or” (not “and”) in stating his query objective. So, you substitute OR for AND in the above SELECT statement, get the (*presumably*) correct result, and email it back. Hopefully, you are smart enough to send a CYA memo that does not insult anyone’s intelligence.

You see the problem. Articulating a query objective may not be easy, especially if a query objective is complex. Probably all human languages (unlike computer languages) allow some ambiguity. So, be careful because sloppy language can encourage sloppy logic.

**Conclusion:** Make sure that you really understand your query objectives.

Author Comment: In writing this book, I had to articulate many query objectives. Sometimes I cheated. For example, with some sample queries, I worked backwards. I began by writing the WHERE-clause’s compound-condition. Then, I used the WHERE-clause to (somehow) derive the narrative articulation of the query objective.

Bottom-Line: Writing a precise, concise, unambiguous query objective in a human language can be a challenge.



## Summary

Previous chapters emphasized knowing-your-data. This chapter emphasized knowing-your-logic. Sample queries showed WHERE-clauses that contained one or more Boolean connectors (AND, OR, and NOT). Their syntax and behavior are summarized below.

WHERE cond-1 AND cond-2: A row is selected only if both cond-1 and cond-2 are true.

WHERE cond-1 OR cond-2: A row is selected if either cond-1 or cond-2 or both conditions are true.

WHERE NOT condition: A row is selected if the condition is not true.

**Truth Tables** offer a precise representation of this logic.

C1	C2	C1 AND C2
T	T	T
T	F	F
F	T	F
F	F	F

C1	C2	C1 OR C2
T	T	T
T	F	T
F	T	T
F	F	F

C1	NOT C1
T	F
F	T

Understanding of Truth Tables is helpful for two reasons.

- Appendix 4B uses truth tables to verify some laws of logic.
- Chapter 11 (Null Values) will present more complex truth tables that expand upon the above truth tables.

**Again:** Make your logic explicit by specifying parentheses when you code compound-conditions.

## Summary Exercises

The following exercises 4R-4T reference the EMPLOYEE table.

- 4R. Display all information about any employee who works in Department 20 and earns less than \$5,000.00.
- 4S. Display the name and salary of any employee who earns less than \$1,000.00 or more than \$6,000.00.
- 4T. Display the name and department number of all employees who do not work for Department 20. Sort the result in ascending sequence by employee name.

The following exercise references the PRESERVE table. It is relatively complex. Yet it should be doable with a little thought.

- 4U. OR means Inclusive-OR. Code an "Exclusive-OR" for the following query which is a modification of Sample Query 4.5.

Display the PNAME, ACRES, and STATE value of any preserve that matches just one (but not both) of the following conditions. (1) The preserve is located in Arizona, or (2) the preserve has more than 1000 acres. The result should look like:

<u>PNAME</u>	<u>ACRES</u>	<u>STATE</u>
HASSAYAMPA RIVER	660	AZ
COMERTOWN PRAIRIE	1130	MT
PINE BUTTE SWAMP	15000	MT
RAMSEY CANYON	380	AZ

Hint-1: Assume you have two conditions, C1 and C2. The most direct way to think about the Exclusive-OR is:

The first condition (C1) is True OR the second condition (C2) is True  
AND  
It is not the case that both conditions are True.

Hint-2: Another way to think about the Exclusive-OR is:

The first condition is True AND the second condition is False.  
OR  
The first condition is False AND the second condition is True.

- 4V. Draw a truth table for the Exclusive-OR (XOR).

## Introduction to Appendices 4A - 4B - 4C

### **"Theory is Practical" - C. J. Date**

The following three appendices total 21 pages, approximately the same size as the preceding content of this chapter! (Many of the following chapters do not have any appendices.)

- Appendix 4A (Efficiency) introduces the fascinating topic of query optimization.
- Appendix 4B (Theory) introduces some laws of logic, a very old topic that predates database theory by centuries.
- Appendix 4C (Theory & Efficiency) describes how the laws of logic can help the optimizer produce efficient SELECT statements.

Author Comment: Appendix 4C integrates theory and practice. Unfortunately, in my academic and professional consulting-training experience, I have heard discouraging words like: "Academic courses are too theoretical. Faculty members live in an ivory tower, etc." And, conversely: "Many practitioners are hackers. They ignore formal logic, sound software engineering and data engineering principles, etc." I contend that individuals who make such comments suffer from tunnel vision, having an extreme bias towards practice or theory. Their comments reflect an Exclusive-OR-Attitude toward theory and practice. *Adopting an Inclusive-OR-Attitude is more interesting and more productive.*

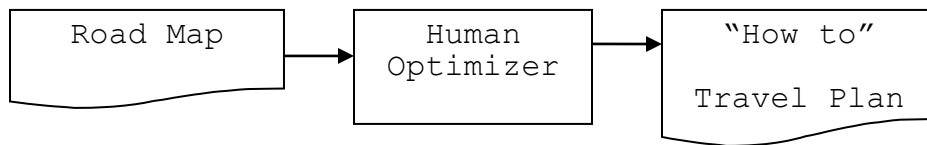
The following Appendix 4A is a conceptual introduction to query optimization. Real-world optimizers utilize many but not all of the techniques described on the following pages. Real-world optimizers are very sophisticated and utilize many other techniques not described herein. For example, the following three appendices only consider optimizing a SELECT statement that references a single table which is located on one computer; whereas real-world optimizers have to optimize multi-table queries where the tables may be distributed across multiple computers connected via a communications network.

## Appendix 4A: Efficiency

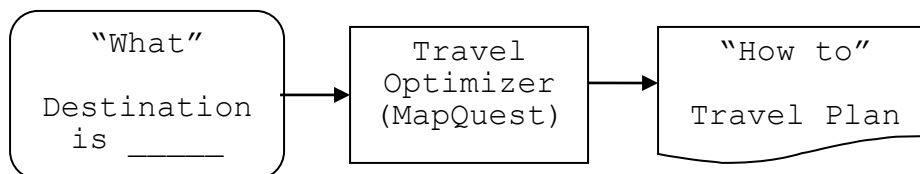
We have arrived at a point where we should discuss database query optimization. Before discussing this topic, it may be helpful to draw an analogy with travel by automobile.

### Analogy: Automobile Travel Plans

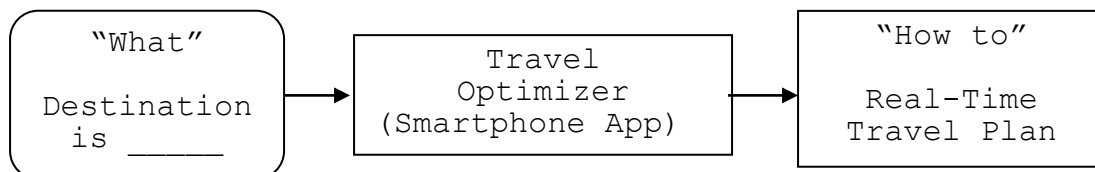
**Ancient History:** Once upon a time, when traveling by car, if you did not know the route to your destination, you obtained a paper map, read it, and formulated your own travel plan. This was a do-it-yourself approach to formulating a travel plan. One problem with this approach was that a paper map, after publication, became obsolete (e.g., a two-lane road became a four-lane road, or a bridge was closed).



**Modern-History-1:** Along came the personal computer and the Internet with programs like MapQuest. You specified your origin, usually your current location, and your destination. (This was the "What.") The program generated a travel plan. (This was the "How to.") Because the program knew travel distances and road speed limits, it would consider multiple possible travel plans and recommend the most efficient plan in terms of minimum travel time. Problems with obsolete information were reduced, but not eliminated. For example, programs were not aware of real-time traffic jams.

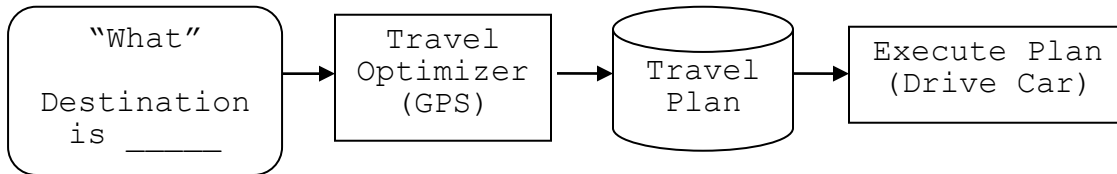


**Modern-History-2:** Along came smart phones with travel apps that used GPS facilities to access real-time travel information. After detecting a traffic problem, the app could dynamically generate an alternative travel plan.



Although travel plans have improved at each historical stage, a human being still has to drive the car to execute the travel plan.

**Near Future:** Self-driving cars/trucks should become common within the next decade. After you state your destination, the travel optimizer will generate a travel plan, and the self-driving car will execute the plan.



\*\*\* Relational systems are analogous to self-driving cars.

### Database Query Optimization & Application Plans

SQL is a **declarative language**. A SELECT statement declares "what" data should be retrieved. A SELECT statement does *not* tell the system "How to" (what internal processes to use to) retrieve the desired data. Instead, a relational database system has a component, called an **optimizer**, that figures out "how to" satisfy a query objective. This "how to" method is called an **application plan**. Also, if there is more than one application plan that can satisfy a query objective, the optimizer (ideally) generates the optimal plan.

Another database component, the **database engine**, reads and executes the application plan and returns the result table. The overall process of query optimization and execution is shown below in Figure 4A.1.

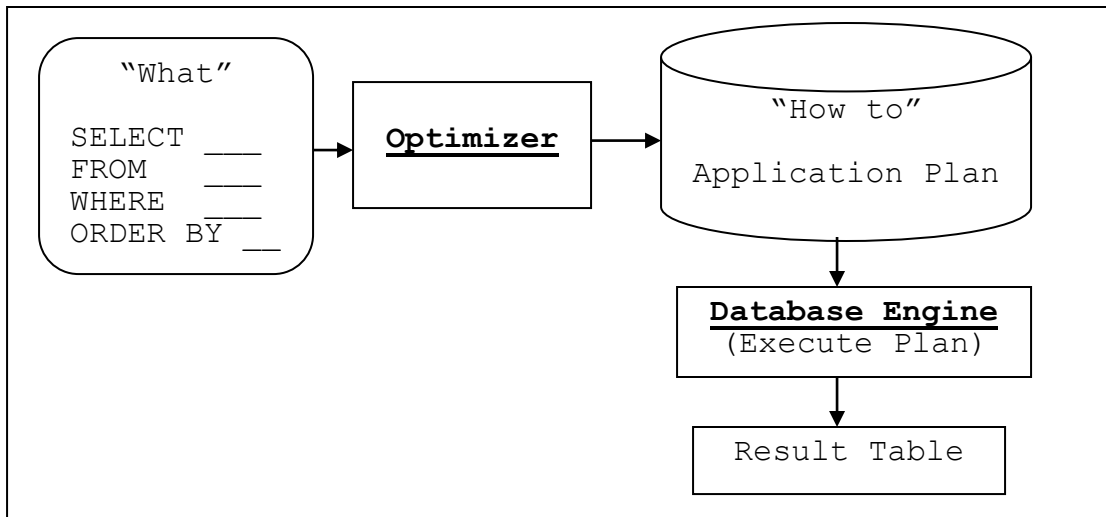
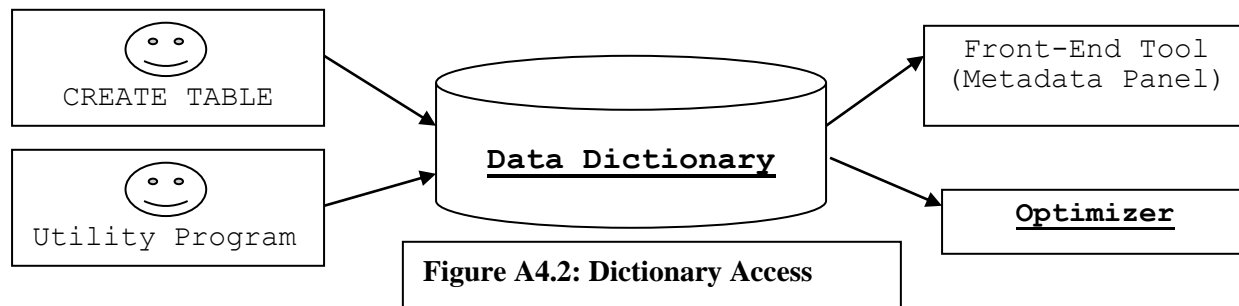


Figure A4.1: Query Optimization and Execution

## Data Dictionary

When a table is referenced by a SELECT statement, the optimizer would like to know the table's size, the availability of relevant indexes, and the distribution of column values. This metadata is stored in the system's Data Dictionary. [DB2's data dictionary is called a "Catalog."] The following Figure A4.2 illustrates Optimizer access to the Data Dictionary. It also illustrates other components that access the Data Dictionary.



When you execute a CREATE TABLE statement, in addition to creating the table, the system automatically updates the Data Dictionary with metadata that describes the table and its columns (data-type, length, etc.). The DBA may also execute a utility program to store and update other information in the data dictionary. Your front-end tool can read the data dictionary and display its metadata in the Metadata Panel. The following pages describe how the optimizer utilizes this metadata.

## Predicting the Optimizer

Previously, we made comments like: "The system *might* use an index," or "The system *could* scan the table." We said "might" and "could" because, in principle, we *cannot always predict what internal processes the system will use to satisfy a query objective*.

More precisely, we could have said:

"The *optimizer might* decide to use an index . . ."  
"The *optimizer could* decide to scan the table . . ."

Again, we used these same weasel words ("might" and "could"). You might ☺ wish that we could ☺ say:

"The optimizer *will* decide to use an index . . ."  
"The optimizer *will* decide to scan the table . . ."

However, because circumstances change (e.g., a small table become much larger), we cannot always predict the application plan the optimizer will generate. In fact, your SQL reference manuals will also use similar weasel words when discussing query optimization.

## Optimizer's "Thought Process"

The following examples introduce the optimizer's thought process. These examples assume the optimizer examines the data dictionary to learn if a table is small or large. (Periodically, the DBA executes a utility program that scans specified tables and indexes and stores relevant statistics in the data dictionary.) The optimizer also learns about indexes by examining the data dictionary. Here, we assume the XPNO index on the PNO column (illustrated in Figure 2.1 in Appendix 2A.1) is the only index on the PRESERVE table.

Example-1:           SELECT \* FROM PRESERVE

There is no WHERE-clause in this statement. Hence, because the query objective is to retrieve all rows, the optimizer decides to scan the entire table.

Example-2:           SELECT \* FROM PRESERVE WHERE FEE = 0.00

Because the objective is to retrieve a subset of rows, the optimizer might consider using an index. However, after consulting the data dictionary, the optimizer learns that there is no index based on the FEE column. Therefore, the optimizer decides to scan the entire table to return those rows that match the FEE = 0.00 condition.

Example-3:           SELECT \* FROM PRESERVE WHERE PNO = 40

This example forces the optimizer to do a little more thinking. By referencing the data dictionary, the optimizer learns that PNO values are unique, and it deduces that the result table cannot contain more than one row. Hence, the optimizer will *consider* using the XPNO index to perform a direct access search. To make its decision, the optimizer will consider the size of the table.

### Four Scenarios

1. Assume PRESERVE is very large (14 million rows). Hence, scanning the entire table would be very expensive. Therefore, the optimizer decides to use the XPNO index to directly access the desired row. We note that a direct access to one row usually requires two disk reads. The first read accesses the XPNO index; and, the second read follows the index pointer to access the desired row in the table.
2. Assume PRESERVE is very small (14 rows). Most likely all rows would be stored in one physical disk page (physical block). In this case, the optimizer would decide to scan the table because it only requires one disk read.

Continuing with Example-3 (WHERE PNO = 40), the next two scenarios describe some "best guess" dilemmas.

3. Assume the optimizer knows PRESERVE is very small, containing 90 rows spread across 3 disk pages. This is a borderline situation. Scanning the entire table involves reading 3 disk pages. A direct access would involve reading 2 pages, one to read the index and another to read the table. Although direct access appears to be slightly more efficient, other factors (not described here) could come into play.
4. Assume the optimizer does not know how many rows are in the PRESERVE table. (Somehow, the utility program that updates the data dictionary statistics was not executed.) Then, the optimizer must make its best guess. Most likely it would decide to use the index because directly accessing one row is not expensive. This would be much cheaper than scanning a potentially large table.

**Selectivity:** The optimizer estimates the *percentage of rows* to be retrieved. This estimate is called the *selectivity* of the WHERE-condition. For some SELECT statements, it is easy to determine their selectivity. This applies to Example-1 and Example-3.

Example-1 (no WHERE-clause): The optimizer knows that all rows will be selected. Hence, we have the largest possible selectivity value is 1.

Example-3 (WHERE PNO = 40): The optimizer knows only one row can be selected. If PRESERVE has 14 million rows, the optimizer will estimate an extremely small selectivity of 1/14,000,000.

A small selectivity measure encourages the optimizer to use a relevant index. However, determining selectivity can become a challenge. [Note: SQL manuals rarely identify a specific selectivity threshold value that determines the use of an index.]

Example-2 (WHERE FEE = 0.00): The FEE column is not unique. Hence it may contain duplicate values. To help estimate selectivity, the data dictionary may store the number of distinct FEE values. For example, if FEE contains five distinct values, the optimizer estimates a selectivity of 1/5. Of course, this presumes a uniform distribution of FEE values. This presumption leads to another best-guess scenario.

Optimizers can usually derive accurate selectivity measures *if* the DBA executes a utility program that stores data distribution statistics (histograms) about column values in the data dictionary. There is a cost to storing and maintaining such statistics. Therefore, the DBA rarely stores detail statistics about all columns in all tables.



**Selectivity for Compound-Conditions:** Estimating selectivity for compound-conditions becomes more of a challenge. We offer some insight into this topic by considering two statements that reference a very large table called CUSTOMER.

```
SELECT * FROM CUSTOMER
WHERE SEX = 'FEMALE' AND FOOTSIZE > 19
```

```
SELECT * FROM CUSTOMER
WHERE SEX = 'FEMALE' OR FOOTSIZE > 19
```

Use your intuition to estimate the selectivity of each WHERE-clause. The selectivity of the first WHERE-clause is extremely small because there are very few women with a foot size over 19. The selectivity of the second WHERE-clause is very large, about 51%. This estimate assumes that 51% of the customers are women, and the few men with very big feet do not significantly increase this estimate.

In some circumstances, the optimizer utilizes probability theory to derive selectivity estimates for compound-conditions.

### **Summary: Optimizer's (Simplified) Strategy**

Assume a SELECT statement references just one table.

If the table is small, then scan the table.

Otherwise

If the table is large, and the statement does not have a WHERE-clause, then scan the table.

Otherwise

If the table is large, and the statement has a WHERE-clause, then look for a relevant index.

If there is no relevant index, then scan the table.

Otherwise

If there is a relevant index and selectivity is good, use the index.

Otherwise

Assuming selectivity is bad, scan the table.

## Appendix 4B: Theory

Codd's Relational Model is based on set theory, and there is a very close relationship between set theory and logic. Therefore, we will take a closer look at some laws of logic that are listed in the following Figure 4B.1. This appendix will use truth tables to validate these laws. The following Appendix 4C will show the relevancy of each law to query optimization.

<p><u>Laws Involving AND of Two Conditions (C1, C2)</u></p> <p>1a. <math>C1 \text{ AND } C2 = C2 \text{ AND } C1</math></p> <p>1b. <math>C1 \text{ AND } \text{FALSE} = \text{FALSE}</math></p> <p>1c. <math>C1 \text{ AND } \text{TRUE} = C1</math></p> <p><u>Laws Involving OR of Two Conditions (C1, C2)</u></p> <p>2a. <math>C1 \text{ OR } C2 = C2 \text{ OR } C1</math></p> <p>2b. <math>C1 \text{ OR } \text{FALSE} = C1</math></p> <p>2c. <math>C1 \text{ OR } \text{TRUE} = \text{TRUE}</math></p> <p><u>De Morgan's Laws: Two Conditions (C1, C2)</u></p> <p>3a. <math>\text{NOT } (C1 \text{ AND } C2) = (\text{NOT } C1) \text{ OR } (\text{NOT } C2)</math></p> <p>3b. <math>\text{NOT } (C1 \text{ OR } C2) = (\text{NOT } C1) \text{ AND } (\text{NOT } C2)</math></p> <p><u>Laws Involving One Condition (C)</u></p> <p>4. <math>C \text{ OR } (\text{NOT } C) = \text{TRUE}</math></p> <p>5. <math>C \text{ AND } (\text{NOT } C) = \text{FALSE}</math></p> <p>6. <math>\text{NOT } (C \text{ AND } (\text{NOT } C)) = \text{TRUE}</math></p> <p><u>Distributive Laws: Three Conditions (C1, C2, C3)</u></p> <p>7a. <math>C1 \text{ AND } (C2 \text{ OR } C3) = (C1 \text{ AND } C2) \text{ OR } (C1 \text{ AND } C3)</math></p> <p>7b. <math>C1 \text{ OR } (C2 \text{ AND } C3) = (C1 \text{ OR } C2) \text{ AND } (C1 \text{ OR } C3)</math></p>
--

**Figure 4B.1: Some Laws of Logic**

Many of the following logical laws will conform to your intuition. Some laws are so obvious that you may wonder why we even bother to mention them. Appendix 4C will address this issue.

### **Validating Logical Laws**

Laws 1-3 involve two conditions (C1 and C2). Sometimes a condition is known to be TRUE; other times a condition is known to be FALSE.

1a. C1 AND C2 = C2 AND C1 (Commutative Law of AND)

This law seems trivial. But it is such an important law that it gets a name: Commutative Law of AND. Casually speaking, we say this law allows us to “flip” conditions C1 and C2 such that C2 becomes the first condition, and C1 becomes the second condition.

Law 2a will present a similar Commutative Law of OR. We note that a Commutative Law does not apply for all SQL operations. Chapter 7 introduces arithmetic expressions. There we will observe that addition and multiplication are commutative (e.g.,  $a+b=b+a$  and  $a*b=b*a$ ), but subtraction and division are not commutative.

1b. C1 AND FALSE = FALSE


This law states that: If you AND-connect any condition (C1) to another condition (C2) that is known to be FALSE, the result is FALSE.

More formally, there are two possible values for C1.

1. If C1 is TRUE, then TRUE AND FALSE = FALSE
2. If C1 is FALSE, then FALSE AND FALSE = FALSE

Hence, in both cases, the result is FALSE.

The following truth table also validates this law. The third column represents this law. Observe that all its column values are False (F).



C1	C2 (FALSE)	C1 AND FALSE
T	F	F
F	F	F

1c. C1 AND TRUE = C1

If you AND-connect any condition (C1) to another condition (C2) that is known to be TRUE, the result has the same truth value as the first condition (C1).

More formally, consider the two possible values for C1.

1. If C1 is TRUE,  
then TRUE AND TRUE = TRUE (the value of C1)
2. If C1 is FALSE,  
then FALSE AND TRUE = FALSE (the value of C1)

In both cases, the result has same value as C1.

The following truth table also validates this law because the first and third columns contain the *same corresponding T/F values*.

↓	C1	C2 (TRUE)	↓	C1 AND TRUE
	T	T		T
	F	T		F

2a. C1 OR C2 = C2 OR C1 (Commutative Law of OR)

This is another apparently trivial but important law that gets a name: Commutative Law of OR. Casually speaking, we say this law allows us to "flip" the conditions C1 and C2 such that C2 becomes the first condition, and C1 becomes the second condition.

2b. C1 OR FALSE = C1

If you OR-connect any condition (C1) to another condition (C2) that is known to be FALSE, the result has the same truth value as the first condition (C1).

More formally, consider the two possible values for C1:

1. If C1 is TRUE,  
then TRUE OR FALSE = TRUE (the value of C1)
2. If C1 is FALSE,  
then FALSE OR FALSE = FALSE (the value of C1)

In both cases, the result is same value as C1.

The following truth table also validates this law because the first and third columns contain the *same corresponding T/F values*.

↓	C1	C2 (FALSE)	↓	C1 OR FALSE
	T	F		T
	F	F		F

2c. C1 OR TRUE = TRUE

If you OR-connect any condition (C1) to another condition (C2) that is known to be TRUE, the result is TRUE.

More formally, consider the two possible values for C1:

1. If C1 is TRUE, then TRUE OR TRUE = TRUE
2. If C1 is FALSE, then FALSE OR TRUE = TRUE

Hence, in both cases, the result is TRUE.

The following truth table also validates this law. The third column represents this law. Observe that all its column values are True (T).

↓

C1	C2 (TRUE)	C1 OR TRUE
T	T	T
F	T	T

### De Morgan's Laws

3a. NOT (C1 AND C2) = (NOT C1) OR (NOT C2)

3b. NOT (C1 OR C1) = (NOT C1) AND (NOT C2)

We have already introduced De Morgan's Laws without formally validating these laws. We validate Law 3a below by developing two truth tables. Observe that the last column in each table shows the same corresponding T/F values. Hence, the two compound-conditions are equivalent.

C1	C2	C1 AND C2	NOT (C1 AND C2)
T	T	T	F
T	F	F	T
F	T	F	T
F	F	F	T



C1	C2	NOT C1	NOT C2	(NOT C1) OR (NOT C2)
T	T	F	F	F
T	F	F	T	T
F	T	T	F	T
F	F	T	T	T

Similar truth tables validate Law 3b.

Laws 4-6 only reference one condition (C). Each law represents a *tautology* or a *contradiction*.

A *tautology* is a statement that, because of its logical form, is always TRUE. (Only T values appear in its Truth Table column.)

A *contradiction* is a statement that, because of its logical form, is always FALSE. (Only F values appear in its Truth Table column.)

4. C OR (NOT C) = TRUE (Law of the Excluded Middle)

This is another apparently trivial law that has a rather sexy name, the Law of the Excluded Middle. This law states that every condition (C) must be TRUE or FALSE. No condition can be "somewhere in the middle" between TRUE and FALSE.

The third column in the following truth table represents this law. Note that all its column values are TRUE (T). Hence C OR (NOT C) is a tautology.



C	NOT C	C OR (NOT C)
T	F	T
F	T	T

5. C AND (NOT C) = FALSE

This law means that any condition (C) AND its negation (NOT C) must be FALSE. (Common Sense: "A statement cannot be both true and false at the same time.")

The third column in the following truth table represents this law. Note that all its column values are FALSE (F). Hence C AND (NOT C) is a contradiction.



C	NOT C	C AND (NOT C)
T	F	F
F	T	F

6. NOT (C AND (NOT C)) = TRUE (Non-Contradiction)

The negation of a contradiction is TRUE. (Common Sense: "Do not tell me that some condition is both TRUE and NOT TRUE.") To validate this law, we start with a contradiction (Law 5).

$$C \text{ AND } (\text{NOT } C) = \text{FALSE.}$$

Then, negate both sides of this equation to produce Law-6.

$$\begin{aligned} \text{NOT } (C \text{ AND } (\text{NOT } C)) &= \text{NOT } (\text{FALSE}) \\ &= \text{TRUE} \end{aligned}$$

The last column in the following truth table represents this law. Note that all its column values are TRUE (T). Therefore, NOT (C AND (NOT C)) is a tautology.

C	NOT C	C AND (NOT C)	NOT (C AND (NOT C))
T	F	F	T
F	T	F	T

Distributive Laws: [Three conditions (C1, C2, C3)]

7a.  $C1 \text{ AND } (C2 \text{ OR } C3) = (C1 \text{ AND } C2) \text{ OR } (C1 \text{ AND } C3)$

7b.  $C1 \text{ OR } (C2 \text{ AND } C3) = (C1 \text{ OR } C2) \text{ AND } (C1 \text{ OR } C3)$

Validate Law 7a by developing the following truth tables. The same corresponding T/F values in the last column of each table show that the two compound-conditions are equivalent. Similar truth tables validate Law 7b.

C1	C2	C3	(C2 OR C3)	C1 AND (C2 OR C3)
T	T	T	T	T
T	F	T	T	T
F	T	T	T	F
F	F	T	T	F
T	T	F	T	T
T	F	F	F	F
F	T	F	T	F
F	F	F	F	F

C1	C2	C3	(C1 AND C2)	(C1 AND C3)	(C1 AND C2) OR (C1 AND C3)
T	T	T	T	T	T
T	F	T	F	T	T
F	T	T	F	F	F
F	F	T	F	F	F
T	T	F	T	F	T
T	F	F	F	F	F
F	T	F	F	F	F
F	F	F	F	F	F

## Appendix 4C: Theory & Efficiency

This appendix illustrates how each logical law described in the previous Appendix 4B can be utilized by the optimizer. The following examples show how the optimizer can apply a specific logical law to rewrite a user-coded WHERE-clause into a presumably more efficient form.

**Optimizer Query Rewrite:** The laws of logic are symbolic patterns that can be recognized by the optimizer. When the optimizer detects such a pattern in a compound-condition within a user's WHERE-clause, it may "mechanically" rewrite this condition into an equivalent and presumably more efficient condition.

The following examples reference a CUSTOMER table that describes customer demographics. Assume the data dictionary shows that CUSTOMER has 14 million rows where 7 million customers are female. The data dictionary also has a histogram of FOOTSIZE values indicating that 25 customers have a foot size that exceeds 19.0.

*\* The following examples assume that dictionary statistics are accurate. Appendix 24B will describe similar examples where the dictionary statistics are inaccurate or approximately accurate.*

### **1a. C1 AND C2 = C2 AND C1 (Commutative Law of AND)**

Assume a user decides to inquire about female customers with big feet by coding the following SELECT statement.

```
SELECT * FROM CUSTOMER
WHERE SEX = 'F' AND FOOTSIZE > 19.0
```

We consider three scenarios. Scenario-1 does not rewrite the user's WHERE-clause.

Scenario-1: Search the CUSTOMER table using the user's compound-condition (SEX = 'F' AND FOOTSIZE > 19.0). Because the SEX = 'F' is specified first, we will assume that the system initially searches for female customers, and then, for each female customer, it examines her foot size information.

Assume there is no index on the SEX column, and no index on the FOOTSIZE column. This means the system must scan all 14 million CUSTOMER rows to satisfy the WHERE-condition. After the system brings each row into memory, it examines its SEX value. If this value is not F, the system immediately concludes the current row is a no-hit. (It does not examine the FOOTSIZE value.) If the SEX value is F, the system examines the FOOTSIZE value. This means that it will perform 7 million comparison operations on FOOTSIZE values.



In Scenario-2 and Scenario-3, the optimizer utilizes Logical Law 1a (Commutative Law of AND) to rewrite the user-coded WHERE-condition.

Scenario-2: Search the CUSTOMERE table after rewriting the compound-condition as:

```
FOOTSIZE > 19.0 AND SEX = 'F'
```

This rewrite implies that the system will initially search on the FOOTSIZE > 19.0 condition. Then, for each big-footed customer, the system will examine the corresponding SEX value.

Again, assume there are no indices on the SEX and FOOTSIZE columns. This means that the system must scan all 14 million rows to satisfy the WHERE-condition. For each row, the system first examines the FOOTSIZE value. If this value is not greater than 19, the system immediately concludes the current row is a no-hit. (It does not need to examine the corresponding SEX value.) If the FOOTSIZE value does exceed 19.0, the system examines the corresponding SEX value. This means that it will only have to perform 25 comparison operations on the SEX value. Obviously, this is more efficient than Scenario-1 which executed 7 million comparisons for the second condition.

The Commutative Law of AND allows the optimizer to perform this rewrite operation. But, how does the optimizer know that it should do this rewrite? Answer: Selectivity. The optimizer is motivated to rewrite the user's WHERE-condition because, after examining the Data Dictionary, the optimizer learns that the FOOTSIZE > 19.0 condition has very good selectivity (25/14,000,000), whereas the SEX = 'F' condition has very poor selectivity (7,000,000/14,000,000).

While Scenario-2 is better than Scenario-1, the improvement is not dramatic because it only reduces CPU time associated with examining SEX values in rows that are already in memory.

Scenario-3: Assume there is a selective index (XFOOTSIZE) on the FOOTSIZE column, and there is no index on the SEX column. Here, because the optimizer can use the XFOOTSIZE index, it decides to initially compare on the FOOTSIZE > 19 condition. Hence, the user-coded WHERE-clause is rewritten as:

```
FOOTSIZE > 19.0 AND SEX = 'F'
```

Using the XFOOTSIZE index to directly access the CUSTOMER table, the system retrieves 25 rows, and then it examines the SEX value in each of these 25 rows. This significantly improves efficiency because, compared to scanning the entire table, the 25 disk reads is significantly smaller; and the CPU time to examine the 25 SEX values is trivial.

**More than two Conditions:** The Commutative Law of AND can include any number of AND-connected columns. Assume a user-specified WHERE-clause contained five simple-conditions (C1, ..., C5) that are AND-connected as shown below.

WHERE C1 AND C2 AND C3 AND C4 AND C5

Scenario-1: Assume there are no relevant indexes that could help any of these conditions. Also assume the optimizer examines the data dictionary and determines the following selectivity measures for each condition.

Selectivity for C1 is 300/1000	(third best)
Selectivity for C2 is 500/1000	(fourth best)
Selectivity for C3 is 5/1000	(best)
Selectivity for C4 is 700/1000	(worst)
Selectivity for C5 is 10/1000	(second best)

Listing these selectivity values from best to worst we have:

Selectivity for C3 is 5/1000	(best)
Selectivity for C5 is 10/1000	(second best)
Selectivity for C1 is 300/1000	(third best)
Selectivity for C2 is 500/1000	(fourth best)
Selectivity for C4 is 700/1000	(worst)

This encourages optimizer rewrite this WHERE-clause as:

WHERE C3 AND C5 AND C1 AND C2 AND C4

**Historical Aside:** If your grandmother ever coded an ancient history COBOL program with an IF-statement that specified a compound-condition, she should *have* specified the most selective condition first. The efficiency gain could have been significant when using old 1960's mainframe technology.

Scenario-2: Assume there is a relevant index for the C5 condition. Because this index has good selectivity (10/1000), the optimizer might be influenced to move C5 to the first position in the above WHERE-clause as shown below.

WHERE C5 AND C3 AND C1 AND C2 AND C4

**Mathematical Aside:** Given five conditions, there are  $5! = (5*4*3*2*1) = 120$  possible permutations, or 120 ways to rewrite the user's WHERE-clause. In principle, the optimizer could evaluate all 120 possibilities and then chose the optimal compound-condition. In practice, the optimizer may start by evaluating a few possible compound-conditions and stop when some condition offers "good enough" efficiency, which might not be optimal.

**1b. C1 AND FALSE = FALSE**

Assume a user's WHERE-clause specified the following compound-condition.

```
SEX = 'F' AND FOOTSIZE > 19.0
```

Also assume that, after examining the data dictionary, the optimizer learns that the maximum FOOTSIZE value is 17.0. Therefore, without retrieving and examining any rows in the table, the optimizer deduces that the FOOTSIZE > 19.0 condition must evaluate to FALSE for every row. This allows the optimizer to rewrite the above compound-condition as:

```
SEX = 'F' AND FALSE
```

Next, by applying Logical Law 1b the optimizer deduces that this compound-condition is FALSE. Hence, without retrieving and examining any rows, the system returns a "no rows retrieved" message. Obviously, this is an extremely efficient application plan.

**1c. C1 AND TRUE = C1**

Assume a user's WHERE-clause specified the following compound-condition.

```
SEX = 'F' AND FOOTSIZE > 4.0
```

Also assume that, after examining the data dictionary, the optimizer learns that the minimum FOOTSIZE value is 5.5. Therefore, without retrieving and examining any rows in the table, the optimizer concludes that the FOOTSIZE > 4.0 condition must evaluate to TRUE for every row. This allows the optimizer to rewrite the above compound-condition as:

```
SEX = 'F' AND TRUE
```

Next, by applying Logical Law 1c the optimizer concludes that the compound-condition is equivalent to:

```
SEX = 'F'
```

The optimizer tells the system to scan the table. For each row, if the SEX = 'F' condition evaluates to TRUE, then the compound-condition must be TRUE. There is no need to examine FOOTSIZE values.

**2a. C1 OR C2 = C2 OR C1 (Commutative Law of OR)**

The following discussion parallels our presentation of the Commutative Law of AND. Assume a user's WHERE-clause specified the following compound-condition:

FOOTSIZE > 19.0 OR SEX = 'F'

Also, assume the data dictionary shows there are 7 million females and 25 customers with a foot size that exceeds 19.0. Now, with an OR comparison, it is more efficient to initially search on the poor selectivity (many matches) condition. Therefore, the optimizer could utilize the above Law 2a to justify rewriting the above compound-condition as:

SEX = 'F' OR FOOTSIZE > 19.0

For every row where the SEX = 'F' condition evaluates to TRUE (about 50% of the time), there is no need to evaluate the FOOTSIZE > 19.0 condition.

Similar to the Commutative Law of AND, the Commutative Law of OR can be generalized to any number of conditions.

**2b. C1 OR FALSE = C1**

Assume a user's WHERE-clause specified the following compound-condition.

SEX = 'F' OR FOOTSIZE > 19.0

Also, assume the dictionary shows the maximum FOOTSIZE value is 18.0. Then the optimizer concludes that the FOOTSIZE > 19.0 condition is FALSE for all rows. Then the above compound-condition is rewritten as:

SEX = 'F' OR FALSE

According to Logical Law 2b, this condition simplifies to:

SEX = 'F'

The optimizer tells the system to scan the table. For each row, if the SEX = 'F' condition evaluates to TRUE, then the compound-condition must be TRUE. There is no need to examine any FOOTSIZE values.

### 2c. C1 OR TRUE = TRUE

Assume a user's WHERE-clause specified the following compound-condition.

```
SEX = 'F' OR FOOTSIZE > 4.0
```

Also assume the optimizer examines the data dictionary and learns that the minimum FOOTSIZE value is 5.0. This implies that FOOTSIZE > 4.0 is always TRUE. Hence the above condition becomes:

```
SEX = 'F' OR TRUE
```

The above Law 2c tells us that the above compound-condition is always TRUE. Hence, the system returns all rows to the result table without examining any values in the retrieved rows.

### 3a. NOT (C1 AND C2) = (NOT C1) OR (NOT C2) [De Morgan's Laws]

Assume the PRESERVE table contains 14 million rows, and the user's WHERE-clause specifies the following compound-condition (coded in Sample Query 4.10).

```
NOT (STATE = 'AZ' AND FEE = 3.00)
```

Then, the optimizer may apply De Morgan's Law to rewrite this compound-condition as:

```
NOT (STATE = 'AZ') OR (NOT FEE = 3.00)
```

Then, removing the NOT keywords this condition becomes:

```
STATE <> 'AZ' OR FEE <> 3.00
```

Finally, the optimizer will examine dictionary statistics for the STATE and FEE columns to determine if it should apply the Commutative Law of OR (Law 2a) to rewrite the condition as:

```
FEE <> 3.00 OR STATE <> 'AZ'
```

Similar observations can be made about user-written compound-conditions that fit De Morgan's Second Law 3b.

#### **4. C OR (NOT C) = TRUE [Excluded Middle]**

Sometimes, a compound-condition similar to the following is submitted to the system.

```
SEX = 'F' OR NOT (SEX = 'F')
```

The optimizer, after observing that this compound-condition fits the C OR (NOT C) pattern, evaluates to TRUE. Hence, the system returns all rows without examining any SEX values.

You might think that any user who coded the above compound-condition is not very intelligent. However, there is a circumstance where an intelligent user could *indirectly* submit such a condition to the system. This pertains to View Processing, a topic that will be described in Chapter 28.

#### **5. C AND (NOT C) = FALSE [Contradiction]**

Sometimes, a compound-condition similar to the following is submitted to the system.

```
SEX = 'F' AND NOT (SEX = 'F')
```

The optimizer observes that this compound-condition conforms to the above C AND (NOT C) pattern and must evaluate to FALSE. Therefore, without retrieving and examining any rows, the system responds with a "no rows returned" message.

Again, an intelligent user might *indirectly* submit such a compound-condition. This pertains to View Processing, a topic that will be described in Chapter 28.

## 6. NOT (C AND (NOT C)) = TRUE [Non-Contradiction]

Sometimes, a compound-condition similar to the following is submitted to the system.

```
NOT (SEX = 'F' AND NOT (SEX = 'F'))
```

The optimizer observes this compound-condition fits the above NOT (C AND (NOT C)) pattern and evaluates to TRUE. Hence, the system returns all rows without examining any SEX values.

Alternatively, the optimizer might apply De Morgan's Law to produce the following equivalent condition.

```
(NOT SEX = 'F') OR NOT (NOT SEX = 'F')
```

Then apply the Commutative Law of OR to produce:

```
NOT (NOT SEX = 'F') OR (NOT SEX = 'F')
```

Then, remove the double NOTs in the first condition to produce:

```
SEX = 'F' OR (NOT SEX = 'F')
```

This compound-condition fits the Law of the Excluded Middle (Logical Law 4). Hence, the system returns all rows to the result table without examining any SEX values.

Again, an intelligent user might *indirectly* submit such a compound-condition. This pertains to View Processing, a topic that will be described in Chapter 28.

## Distributive Laws

7a.  $C1 \text{ AND } (C2 \text{ OR } C3) = (C1 \text{ AND } C2) \text{ OR } (C1 \text{ AND } C3)$

7b.  $C1 \text{ OR } (C2 \text{ AND } C3) = (C1 \text{ OR } C2) \text{ AND } (C1 \text{ OR } C3)$

Example-1: Assume a user's WHERE-clause specified the following compound-condition (where WT is the customer's weight).

```
(FOOTSIZE > 4.0 AND SEX = 'F') OR (FOOTSIZE > 4.0 AND WT > 100)
```

The optimizer could apply the above Law 7a to factor out the common condition (FOOTSIZE > 4.0) to get:

```
FOOTSIZE > 4.0 OR (SEX = 'F' AND WT > 100)
```

This compound-condition would be more efficient if FOOTSIZE > 4.0 is always TRUE, or there is an index on the FOOTSIZE column that has good selectivity.

Example-2: Assume a user's WHERE-clause specified the following compound-condition.

```
SEX = 'F' AND (SEX = 'M' OR WT > 300)
```

The optimizer might apply Law 7a to get:

```
(SEX= 'F' AND SEX= 'M') OR (SEX= 'F' AND WT > 300)
```

Since no column in a row can simultaneously contain two different values, (SEX= 'F' AND SEX= 'M') must evaluate to FALSE. Hence the expression reduces to:

```
FALSE OR (SEX= 'F' AND WT > 300)
```

Then apply Law 2b to produce:

```
(SEX = 'F' AND WT > 300)
```

If there is an index on the WT column, and WT > 300 has good selectivity, the optimizer might apply Law 1a to generate the final compound-condition.

```
WT > 300 AND SEX= 'F'
```



## Conclusions

**Optimizer Query-Rewrite:** The Laws of Logic allow an optimizer to *mechanically* transform a user-written compound-condition into another equivalent condition. If a user-coded compound-condition is complex, the optimizer may have to evaluate a large number of equivalent conditions. The optimizer can (in principle) evaluate the efficiency of all equivalent conditions. However, in practice, the optimizer may decide to evaluate just some conditions and choose to implement a condition that offers "good enough" efficiency.

**What about User Query-Rewrite?** A user can also utilize the same laws of logic to code a (presumably) efficient WHERE-clause. This is a good idea, especially if an application developer is coding an embedded SQL statement that will be executed many times in the future. However, there are two caveats.

First. An *ideal* optimizer, with its knowledge of logical laws, and its access to real-time metadata, will produce an optimal application plan. Again, users should focus on logical correctness.

Second. Things change. A small table can become much larger, or vice versa. A new more helpful index could be created, or a relevant index could be dropped. Etc. Hence the user's SQL statement may have to be re-optimized to account for these changes. This could undo the user's efforts at efficiency. So, again, users should focus on logical correctness.

*However, users should still understand the laws of logic for two practical reasons.*

First, the user may want to change a WHERE-clause originally coded by another user who had a different logical mindset.

Second, on a rare occasion, a user may need to override a decision made by an *imperfect* optimizer. This could involve a do-it-yourself rewrite of a WHERE-clause (More details on this issue will be presented towards the end of this book.)

## IN and BETWEEN

The previous chapter introduced compound-conditions that specified the Boolean connectors: AND, OR, and NOT. This chapter introduces two new keywords, IN and BETWEEN, that, in some circumstances, can be used as alternatives to coding Boolean connectors.

The keywords IN and BETWEEN do not offer any new functionality. Therefore, in principle, you can skip this entire chapter. However, you may find IN and BETWEEN to be useful because these keywords provide abbreviations that, in some circumstances, can facilitate coding smaller and perhaps more understandable WHERE-clauses.

This is a short chapter. The first five sample queries introduce the IN and BETWEEN keywords. This is followed by commentary about their equivalence to compound-conditions specified with AND, OR, and NOT.

## IN Keyword

The following sample query illustrates a convenient way of asking the system to select a row if a specified column contains a value in a set of values.

**Sample Query 5.1:** Display the PNO, PNAME, and ACRES values for any nature preserve with an ACRES value equal to one of the following values: {40, 90, 630, 660}.

```
SELECT PNO, PNAME, ACRES
FROM PRESERVE
WHERE ACRES IN (40, 90, 630, 660)
```

PNO	PNAME	ACRES
5	HASSAYAMPA RIVER	660
13	TATKON	40
10	HOFT FARM	90

**Syntax:** WHERE column IN (value1, value2, ...)

The set of values must be enclosed within parentheses with commas separating each value. These values can be numeric or character-strings. (Sample Query 5.2 will show that character-string values must be enclosed by apostrophes.)

The values in this IN-clause happen to be written in ascending sequence. This may help readability, but it is not required. For all practical purposes there is no upper limit on the number of specified values.

**Equivalent WHERE-Clause:** This WHERE-clause could be coded as:

```
WHERE ACRES = 40
      OR ACRES = 90
      OR ACRES = 630
      OR ACRES = 660
```

### Exercise:

5A. Display all information about any nature preserve that has a preserve number in the set {2, 4, 6, 8, 10}

## NOT IN

The following sample query specifies a NOT IN comparison operation.

**Sample Query 5.2:** Display the PNO, PNAME, and STATE of any nature preserve that is not located in Arizona and is not located in Montana.

```
SELECT PNO, PNAME, STATE
FROM PRESERVE
WHERE STATE NOT IN ('AZ', 'MT')
```

PNO	PNAME	STATE
14	MCELWAIN-OLSEN	MA
13	TATKON	MA
9	DAVID H. SMITH	MA
11	MIACOMET MOORS	MA
12	MOUNT PLANTAIN	MA
10	HOFT FARM	MA

**Syntax:** STATE contains character-string values. Therefore, each specified value must be enclosed within apostrophes.

**Logic:** NOT IN complements IN. This WHERE-clause selects data from just those PRESERVE rows that were not selected in the previous sample query.

Two alternative WHERE-clauses are:

```
WHERE NOT STATE = 'AZ' AND NOT STATE = 'MT'
```

```
WHERE STATE <> 'AZ' AND STATE <> 'MT'
```

### Exercises:

- 5B. Display all information about the following nature preserves: DANCING PRAIRIE, MULESHOE RANCH, MCELWAIN-OLSEN, and TATKON.
- 5C. Display all information about all nature preserves except: DANCING PRAIRIE, MULESHOE RANCH, MCELWAIN-OLSEN, and TATKON.

## BETWEEN

The following sample query specifies BETWEEN to select rows where a specified column has a value within a given range of values.

**Sample Query 5.3:** Display the name and size of any nature preserve where its size is between, or includes, 830 and 15,000 acres.

```
SELECT PNAME, ACRES
FROM PRESERVE
WHERE ACRES BETWEEN 830 AND 15000
```

<u>PNAME</u>	<u>ACRES</u>
DAVID H. SMITH	830
COMERTOWN PRAIRIE	1130
PINE BUTTE SWAMP	15000
PAPAGONIA-SONOITA CREEK	1200

**Syntax:** WHERE column BETWEEN smaller-value AND larger-value

Note that the smaller value (830) is specified first, to the left of AND. We will say more about this observation later in this chapter.

**Logic:** BETWEEN means "between and including." Any row matching a boundary ACRES value of 830 or 15000 is included in the result table.

An alternative WHERE-clause is:

```
WHERE ACRES >= 830 AND ACRES <= 15000
```

### Exercise:

5D. Display all information about any nature preserve with a PNO value between and including 3 and 10.

## NOT BETWEEN

NOT BETWEEN is used to select rows where a specified column value falls outside a given range of values. The following sample query selects just those rows that were not selected in the previous sample query.

**Sample Query 5.4:** Display the name and the size of any nature preserve where its size is not between 830 and 15,000 acres.

```
SELECT PNAME, ACRES
FROM PRESERVE
WHERE ACRES NOT BETWEEN 830 AND 15000
```

<u>PNAME</u>	<u>ACRES</u>
HASSAYAMPA RIVER	660
DANCING PRAIRIE	680
MULESHOE RANCH	49120
SOUTH FORK MADISON	121
MCELWAIN-OLSEN	66
TATKON	40
MIACOMET MOORS	4
MOUNT PLANTAIN	730
RAMSEY CANYON	380
HOFT FARM	90

**Syntax:** WHERE column NOT BETWEEN smaller-value AND larger-value.

Note that the smaller value (830) is specified first, to the left of AND. We will say more about this observation later in this chapter.

**Logic:** NOT BETWEEN complements BETWEEN. Any row with an ACRES value of 830 or 15000 is *not* included in the above result.

An alternative WHERE-clause is:

```
WHERE ACRES < 830 OR ACRES > 15000
```

### Exercise:

5E. Display all information about any nature preserve having a PNO value that is less than 3 or greater than 10.

## BETWEEN with Character-String Values

BETWEEN and NOT BETWEEN conditions can reference character-string values.

**Sample Query 5.5:** Display the name of all nature preserves having a name that, based upon an alphabetical (collating) sequence, lies between and including the strings 'M' and 'MZZZ'.

```
SELECT PNAME
FROM PRESERVE
WHERE PNAME BETWEEN 'M' AND 'MZZZ'
```

```
PNAME
MULESHOE RANCH
MCELWAIN-OLSEN
MIACOMET MOORS
MOUNT PLANTAIN
```

**Syntax:** Nothing new.

**Logic:** Note that, within an alphabetical sequence, 'M' is the smaller value and 'MZZZ' is the larger value. Because we can reasonably assume that no PNAME value that begins with the letter 'M' will ever be larger than 'MZZZ', we can restate the query objective as:

Display the name of all nature preserves with a name that begins with the letter M.

The following Chapter 6 introduces the keyword LIKE which will offer a more direct way to satisfy this query objective.

## Logic: IN and NOT IN

**Implied-OR:** The logic of IN is straightforward. Note that when you specify IN before a set of values, each comma effectively represents an *"implied-OR"*. Therefore, the following WHERE-clauses are equivalent.

```
WHERE COLA IN (v1, v2, v3, v4)
```

```
WHERE COLA = v1
      OR COLA = v2
      OR COLA = v3
      OR COLA = v4
```

**Implied-AND:** The logic of NOT IN is slightly more complex. When you specify NOT IN before a set of values, each comma represents an *"implied-AND"*. Therefore, the following WHERE-clauses are equivalent.

```
WHERE COLA NOT IN (v1, v2, v3, v4)
```

```
WHERE NOT COLA = v1
      AND NOT COLA = v2
      AND NOT COLA = v3
      AND NOT COLA = v4
```

To summarize:

When coding IN, a comma is an implied-OR.

When coding NOT IN, a comma is an implied-AND.

**Observation:** The specification of redundant values is meaningless.

The system interprets: WHERE COLA IN (2, 2, 3, 2, 3, 3, 2)

As: WHERE COLA IN (2, 3)

Why bother to make this observation? Without explanation, this observation will become relevant in Chapter 23 which introduces Sub-SELECTs.



## Logic: BETWEEN and NOT BETWEEN

Although the logic of BETWEEN and NOT BETWEEN is straightforward, we emphasize that you should specify the smaller-value first.

BETWEEN: WHERE COLA BETWEEN V1 AND V2

is equivalent to:

WHERE COLA >= V1 AND COLA <= V2

Intentional Error - Specify the larger-value first.

WHERE ACRES BETWEEN 15000 AND 830

The system would interpret this WHERE-clause as:

WHERE ACRES >= 15000 AND ACRES <= 830

This condition will always produce a "no rows returned" result because no value can be both greater than or equal to 15000 and less than or equal to 830.

NOT BETWEEN: WHERE COLA NOT BETWEEN V1 AND V2

is equivalent to:

WHERE COLA < V1 OR COLA > V2

Intentional Error - Specify the larger-value first.

WHERE ACRES NOT BETWEEN 15000 AND 830

The system would interpret this WHERE-clause as:

WHERE ACRES < 15000 OR ACRES > 830

Every ACRES value must match this condition because any arbitrary value must be less than 15000 or greater than 830. Hence all rows would be returned.

For tutorial purposes, the following sample query presents a more complex query objective.

**Sample Query 5.6:** Display the PNAME, STATE, ACRES and FEE values for all nature preserves that are located in Montana or Arizona and have a size greater than or equal to 660 acres and less than or equal to 10000 acres, or any other preserve having an admission fee of \$0.00. Sort the result by the PNAME column in ascending sequence.

```
SELECT PNAME, STATE, ACRES, FEE
FROM PRESERVE
WHERE (STATE IN ('MT', 'AZ') AND ACRES BETWEEN 660 AND 1000)
OR FEE = 0.00
ORDER BY PNAME
```

PNAME	STATE	ACRES	FEE
COMERTOWN PRAIRIE	MT	1130	0.00
DANCING PRAIRIE	MT	680	0.00
DAVID H. SMITH	MA	830	0.00
HASSAYAMPA RIVER	AZ	660	3.00
HOFT FARM	MA	90	0.00
MCELWAIN-OLSEN	MA	66	0.00
MIACOMET MOORS	MA	4	0.00
MOUNT PLANTAIN	MA	730	0.00
MULESHOE RANCH	AZ	49120	0.00
PINE BUTTE SWAMP	MT	15000	0.00
SOUTH FORK MADISON	MT	121	0.00
TATKON	MA	40	0.00

**Syntax & Logic:** Nothing new.

**Exercise:**

5F. Display the state, preserve number, and size of any nature preserve that is not in Montana and not in Arizona and is less than 50 acres or greater than 800 acres. Sort the result by preserve number in descending sequence.

## Summary

This chapter introduced the IN and BETWEEN keywords. Both keywords can be preceded by NOT. In some circumstances, these keywords can help you code more compact and readable conditions.

WHERE COLX **IN** (value1, value2, value3,...):

A row is selected if its COLX value equals any value in the specified list of values.

WHERE COLX **NOT IN** (value1, value2, value3,...):

A row is selected if its COLX value is not equal to any value in the specified list of values.

WHERE COLX **BETWEEN** value1 AND value2:

A row is selected if its COLX value is greater than or equal to value1 and less than or equal to value2.

WHERE COLX **NOT BETWEEN** value1 AND value2:

A row is selected if its COLX value falls outside the range specified by value1 and value2.

## Summary Exercises

The following exercises pertain to the EMPLOYEE table. Specify the IN and BETWEEN keywords in the SELECT statements for the following exercises.

- 5G. Display all information about any employee who works in a department with a DNO value in the following list: {10, 40}.
- 5H. Display all information about any employee who works in a department with a DNO value that is not in the following list: {10, 40}.
- 5I. Display all information about any employee whose salary is greater than or equal to \$500.00 and less than or equal to \$2,000.00.
- 5J. Display all information about any employee whose salary is less than \$500.00 or greater than \$2,000.00.

There are no Appendices for this chapter.

## Pattern Matching: LIKE

This chapter is organized into three sections.

Section-A - CHAR versus VARCHAR: We have *not* yet presented the major difference between the CHAR and VARCHAR data-types. This short (1-page), but very important section, will describe this difference to set the stage for our introduction to the LIKE keyword.

Section-B - LIKE Keyword: This section illustrates the LIKE keyword that provides a method to select rows where a character-string value matches some string-pattern. Below we preview a simple example.

Objective: Reference the PRESERVE table. Display all PNAME values beginning with the letter M.

```
SELECT PNAME
FROM PRESERVE
WHERE PNAME LIKE 'M%'
```

```
PNAME
MULESHOE RANCH
MCELWAIN-OLSEN
MIACOMET MOORS
MOUNT PLANTAIN
```

Here, the LIKE-pattern ('M%') is a character-string enclosed within apostrophes. This pattern contains a wildcard symbol, the percent sign (%), which represents any string of any length.

Section-C - DB2, SQL Server, and ORACLE: This section describes the string comparison logic used by DB2, SQL Server, and ORACLE. It also presents potential problems associated with the differences between the CHAR and VARCHAR data-types.

## A. CHAR versus VARCHAR

To distinguish a CHAR value from a VARCHAR value, we must go under the hood to look at the internal representation of character-string values. We do this by examining the DEMO1 table which is illustrated in the following two figures.

Figure 6.1 shows the user's outside view of the DEMO1 table. This table has two columns, CHARNAME and VCHARNAME. The CHARNAME column has a CHAR(10) data-type; and the VCHARNAME column has a VARCHAR(10) data-type. Figure 6.1 shows these columns contain the same values, whereas Figure 6.2 shows that these same values have different internal representations.

<u>CHARNAME</u>	<u>VCHARNAME</u>
DAVID	DAVID
SOLOMON	SOLOMON
MATTHEW	MATTHEW
MARK	MARK
LUKE	LUKE
JOHN	JOHN
EUCLID	EUCLID
WASHINGTON	WASHINGTON
ADAMS	ADAMS
JEFFERSON	JEFFERSON
MADISON	MADISON

Figure 6.1: Outside View of DEMO1

<u>CHARNAME</u>	<u>LEN</u>	<u>VCHARNAME</u>
DAVIDbbbbbb	5	DAVID
SOLOMONbbb	7	SOLOMON
MATTHEWbbb	7	MATTHEW
MARKbbbbbb	4	MARK
LUKEbbbbbb	4	LUKE
JOHNbbbbbb	4	JOHN
EUCLIDbbbb	6	EUCLID
WASHINGTON	10	WASHINGTON
ADAMSbbbbbb	5	ADAMS
JEFFERSONb	9	JEFFERSON
MADISONbbb	7	MADISON

Figure 6.2: Inside View of DEMO1

**CHARNAME Column:** CHARNAME contains CHAR(10) values where all values have exactly 10 characters. This means that some values may have trailing blanks. In Figure 6.2, the lower case "b" represents a blank (space). Notice that every CHARNAME value, except WASHINGTON, has one or more trailing blanks.

**VCHARNAME Column:** VCHARNAME contains VARCHAR(10) values. Here, the 10 means that no VCHARNAME value can exceed 10 characters. Examination of Figure 6.2 shows that the length (LEN) of each VCHARNAME value is stored along with its data value. Important: Observe that no VCHARNAME value has any trailing blanks. (In an atypical *circumstance*, trailing blanks may appear at the end of a VARCHAR value. Such a circumstance will be described in Sample Query 6.11.)

**Design Comment:** Unlike the short VARCHAR columns in our FREESQL sample tables, your DBA might specify a VARCHAR column to save disk storage used by trailing blanks. For example, a CONTRACT\_NARRATIVE column may contain many short character-string values with lengths that are less than 150 characters. However, this column might be defined as VARCHAR (20000) to accommodate a few very long character-strings.

## B. LIKE Keyword: Search for Pattern at Beginning of Character-String

The following sample queries search for two characters located at the beginning of a character-string.

**Sample Query 6.1a:** Reference the CHARNAME column in DEMO1. Display all values that begin with MA.

```
SELECT CHARNAME                                DB2, ORACLE, & SQL Server
FROM DEMO1
WHERE CHARNAME LIKE 'MA%'
```

```
CHARNAME
MATTHEW
MARK
MADISON
```

**Sample Query 6.1b:** Reference the VCHARNAME column in DEMO1. Display all values that begin with MA.

```
SELECT VCHARNAME                                DB2, ORACLE, & SQL Server
FROM DEMO1
WHERE VCHARNAME LIKE 'MA%'
```

```
VCHARNAME
MATTHEW
MARK
MADISON
```

**Syntax:** WHERE character-column LIKE 'pattern'

Both SELECT statements specify 'MA%' as the LIKE-pattern. The percent sign (%) is a "wildcard" symbol that represents any string of any length. Note: LIKE-patterns are case sensitive.

**Logic:** Because the first two characters in the pattern are MA, every selected value must begin with MA. The percent sign (%) indicates that the following characters can be any string of any length, including an "empty string" with a length of zero.

**Important Observation:** The above sample queries illustrate that, when searching for a LIKE-pattern at the beginning of a character-string, CHAR and VARCHAR columns behave in the same manner. Sample Queries 6.3a and 6.3b will show this observation does not apply when searching for a LIKE-pattern at the end of a character-string.

## Search for a Pattern “Anywhere” in a Character-String

The following sample queries illustrate the use of multiple percent sign symbols to match on a LIKE-pattern located anywhere at the beginning, “middle,” or end of a character-string.

**Sample Query 6.2a:** Reference the CHARNAME column in DEMO1. Display every CHARNAME value that has an E anywhere.

```
SELECT CHARNAME                                DB2, ORACLE, & SQL Server
FROM DEMO1
WHERE CHARNAME LIKE '%E%'
```

```
CHARNAME
MATTHEW
LUKE
EUCLID
JEFFERSON
```

**Sample Query 6.2b:** Reference the VCHARNAME column in DEMO1. Display every VCHARNAME value that has an E anywhere.

```
SELECT VCHARNAME                                DB2, ORACLE, & SQL Server
FROM DEMO1
WHERE VCHARNAME LIKE '%E%'
```

```
VCHARNAME
MATTHEW
LUKE
EUCLID
JEFFERSON
```

**Syntax:** A LIKE-pattern may specify multiple wildcard symbols.

**Logic:** Recall that the percent sign can represent an empty string. Here, '%E%' matched on EUCLID with an E at the beginning of the string; E matched on LUKE with an E at the end of the string; and E appeared “somewhere in the middle” of the MATTHEW and JEFFERSON strings.

**Important Observation:** When searching for a pattern that can appear anywhere in the string, CHAR and VARCHAR columns behave the same manner.

### Exercises:

- 6A. Reference the PRESERVE table. Display the PNAME value of all nature preserves with a name that begins with the letter D.
- 6B. Reference the PRESERVE table. Display the name of any nature preserve with TOWN anywhere in its name.

## Search for a Pattern at End of a VARCHAR Character-String

Unlike the previous two sample queries, when searching for a string-pattern at the end of a character-string you must be aware of the differences between the CHAR and VARCHAR data-types.

The following sample query searches for a LIKE-pattern located at the end of a VARCHAR character-string. Recall that, within the VCHARNAME column, no value has trailing blanks.

**Sample Query 6.3a:** Reference the VCHARNAME column in DEMO1. Display every VCHARNAME value that ends with the letters ON.

```
SELECT VCHARNAME          DB2, ORACLE, & SQL Server
FROM DEMO1
WHERE VCHARNAME LIKE '%ON'
```

```
VCHARNAME
SOLOMON
WASHINGTON
JEFFERSON
MADISON
```

**Syntax:** Nothing new.

**Logic:** This LIKE-clause is straightforward because we know that VCHARNAME values do not have trailing blanks. (Again, this is typical.)

### Exercise:

6C. Reference the PRESERVE table. Display the PNAME value of all nature preserves with a name that ends with PRAIRIE.



## Search for a Pattern at End of a CHAR Character-String (Careful!)

Sometimes, *different systems produce different results for the same SELECT statement! Ouch!* This is the first chapter where we must (unfortunately) consider some differences between DB2, ORACLE, and SQL Server.

The next sample query searches for a pattern located at the end of a fixed-length (CHAR) character-string. Recall that CHAR values frequently have trailing blanks.

**\*\*\* Important:** *When performing a LIKE-comparison, some systems (e.g., DB2 and ORACLE) are sensitive to trailing blanks, but other systems (e.g., SQL Server) ignore trailing blanks.*

**Sample Query 6.3b:** Reference the CHARNAME column in DEMO1. Display all CHARNAME values that end with the letters ON.

SELECT CHARNAME	<u>SQL Server</u>
FROM DEMO1	
WHERE CHARNAME LIKE '%ON'	

**SQL Server:** Correct Result

<u>CHARNAME</u>
SOLOMON
WASHINGTON
JEFFERSON
MADISON

SQL Server satisfies the query objective because it *ignores trailing blanks*. Three CHARNAME values (SOLOMON, JEFFERSON, and MADISON) end with ON followed by trailing blanks. However, because these trailing blanks are ignored, SOLOMON, JEFFERSON, and MADISON appear along with WASHINGTON in the result table.

**DB2 & ORACLE:** *Incorrect Result*

<u>CHARNAME</u>
WASHINGTON

DB2 and ORACLE do *not satisfy this query objective because these systems do not ignore trailing blanks*, and SOLOMON, JEFFERSON, and MADISON are stored with trailing blanks. Hence these values do not appear in the result table.

Sample Query 6.13 will utilize the RTRIM function to remove trailing blanks. For tutorial purposes, the following sample queries do not utilize this function.

## Common Error

For tutorial purposes, the following examples specify some "almost correct" (i.e., incorrect) LIKE-patterns.

Example-1: Same as Sample Query 6.3a.

Display all VCHARNAME values that end with ON.

SELECT VCHARNAME FROM DEMO1 WHERE VCHARNAME LIKE '%ON%'	<u>DB2, ORACLE, &amp; SQL Server</u> → <b>Error</b> but "got lucky"
---	---

```
VCHARNAME  
SOLOMON  
WASHINGTON  
JEFFERSON  
MADISON
```

Example-2: Same as Sample Query 6.3b.

Display all CHARNAME values that end with ON.

SELECT CHARNAME FROM DEMO1 WHERE CHARNAME LIKE '%ON%'	<u>DB2, ORACLE, &amp; SQL Server</u> → <b>Error</b> but "got lucky"
---	---

```
CHARNAME  
SOLOMON  
WASHINGTON  
JEFFERSON  
MADISON
```

**Logic: What's wrong with '%ON%'?** Both results happen to be correct (by good luck) because, in both the CHARNAME and VCHARNAME columns, ON *only* appears at the end of a character-string.

Assume you saved the above SELECT statements, and then, sometime in the future, someone inserted a new row with CHARNAME and VCHARNAME values of MADISONXX. Subsequent execution of these saved SELECT statements would produce incorrect results because MADISONXX would appear in the results. *Your good luck is really bad luck.*

## Exercises

Reference the PRESERVE table for Exercises 6D-6H.

The PNAME column is a VARCHAR data-type where trailing blanks cannot occur.

- 6D. Display the name of any nature preserve where the name begins with MULE.
- 6E. Display the name of any nature preserve having the string ING anywhere in its name.
- 6F. Display the name of any nature preserve where the name ends with the letter E.
- 6G. Display the name of any nature preserve that has the letter E immediately after the letter M anywhere in its name.
- 6H. Display the name of any nature preserve that has the letter E anywhere after the letter M in its name.

Reference the DEMO1 table for the following exercise.

- 6I. Display all CHARNAME values in the DEMO1 table where the CHARNAME value ends with D.
  - (a) SQL Server users can solve this exercise.
  - (b) Optionally, DB2 and ORACLE users can solve this exercise if they to jump ahead to Sample Query 6.13 to learn about the RTRIM function.

## Another Wildcard: Underscore (\_) Symbol

The following two sample queries introduce another wildcard symbol, the underscore (\_) that represents exactly one character in a character-string.

**Sample Query 6.4a:** Reference the CHARNAME column in DEMO1. Display every CHARNAME value that has the letter S in the fifth character position.

```
SELECT CHARNAME          DB2, ORACLE, & SQL Server
FROM DEMO1
WHERE CHARNAME LIKE '_ _ _ _ S%'
```

**Sample Query 6.4b:** Reference the VCHARNAME column in DEMO1. Display every VCHARNAME value that has the letter S in the fifth character position.

```
SELECT VCHARNAME        DB2, ORACLE, & SQL Server
FROM DEMO1
WHERE VCHARNAME LIKE '_ _ _ _ S%'
```

Both statements produce the same correct result.

```
CHARNAME
ADAMS
MADISON
```

**Logic:** The underscore symbol represents exactly one character position, and *that position must be present in the string*. In these examples, the CHARNAME and VCHARNAME values match if:

- There is a first character, but we don't care what it is.
- There is a second character, but we don't care what it is.
- There is a third character, but we don't care what it is.
- There is a fourth character, but we don't care what it is.
- There is a fifth character, and it must be S.

There may or may not be more characters after the fifth character. If there are other characters, % implies that we don't care what these characters are.

## Underscore Symbol with CHAR Values

**Sample Query 6.5:** Reference CHARNAME in DEMO1. Display each CHARNAME value with the letters ON in the sixth and seventh positions.

```
SELECT CHARNAME                                DB2, ORACLE, & SQL Server
FROM DEMO1
WHERE CHARNAME LIKE ' _ _ _ _ _ ON%'

CHARNAME
SOLOMON
MADISON
```

**Syntax & Logic:** Nothing New.

The following LIKE-pattern specifies a string length of 10 which is the length of the CHARNAME column. While correct, this LIKE-pattern is discouraged because it is awkward, requiring the user to account for exactly 10 characters.

```
SELECT CHARNAME
FROM DEMO1
WHERE CHARNAME LIKE ' _ _ _ _ _ ON _ _ _ _ _ '
```

## Underscore Symbol with VARCHAR Values

**Sample Query 6.6:** Reference VCHARNAME in DEMO1. Display each VCHARNAME value with the letters ON in the sixth and seventh positions.

```
SELECT VCHARNAME      DB2, ORACLE & SQL Server
FROM DEMO1
WHERE VCHARNAME LIKE ' _ _ _ _ _ ON% '
```

```
VCHARNAME
SOLOMON
MADISON
```

**Logic:** Nothing new.

Note that the following LIKE-pattern would fail.

```
VCHARNAME LIKE ' _ _ _ _ _ ON _ _ _ '
```

This pattern fails to select any rows because it specifies a string length of 10, and all VCHARNAME values with ON in sixth and seventh position have shorter lengths.

## WHERE-Clause Specifies Multiple LIKE-Patterns

Sometimes you want to search for a string pattern that cannot be expressed by a single LIKE-pattern. Consider the following query objective.

**Sample Query 6.7:** Reference CHARNAME in DEMO1. Display any CHARNAME value that has ID in the fourth and fifth positions or has ID in the fifth and sixth positions.

```
SELECT CHARNAME                                DB2, ORACLE & SQL Server
FROM DEMO1
WHERE CHARNAME LIKE ' _ _ _ ID%'
OR CHARNAME LIKE ' _ _ _ _ ID%'
```

```
CHARNAME
DAVID
EUCLID
```

Syntax & Logic: Nothing new. Note that DAVID has D in the fifth position, and EUCLID has I in this same position. Hence, a single LIKE-pattern cannot match both values. (An exception for SQL Server is presented on the following page.)

The result would be the same if this statement referenced the VCHARNAME column.

### Exercises:

- 6J. Reference the PRESERVE table. Display the name of any nature preserve that has the letter A in the second character position.
- 6K. Reference the PRESERVE table. Display the name of any nature preserve that has a blank anywhere in its name.
- 6L. Reference the PRESERVE table. Display the name of any nature preserve having FARM or SWAMP or PRAIRIE anywhere in its name.
- 6M. Reference the PRESERVE table. Display the name of any nature preserve with a period or a hyphen anywhere in its name.
- 6N. Reference the PRESERVE table. Display the name of any nature preserve that has the letter R in the fifth position and ends with PRAIRIE.

## Grammar Query

**Sample Query 6.8:** Display every CHARNAME value that has a vowel in the second position.

<pre>SELECT CHARNAME FROM DEMO1 WHERE CHARNAME LIKE ' _A%' OR CHARNAME LIKE ' _E%' OR CHARNAME LIKE ' _I%' OR CHARNAME LIKE ' _O%' OR CHARNAME LIKE ' _U%'</pre>	<b><u>DB2, ORACLE &amp; SQL Server</u></b>
--	--

```
CHARNAME
DAVID
SOLOMON
MATTHEW
MARK
LUKE
JOHN
EUCLID
WASHINGTON
JEFFERSON
MADISON
```

**Syntax & Logic:** The underscore implies that any character can appear in the first position. Each LIKE-pattern specifies a different vowel in the second position. The remaining positions of the character-string may be any value of any length as indicated by the percent sign. With the exception of ADAMS, all CHARNAME values appear in this result.

The result would be the same if this statement referenced the VCHARNAME column.

**SQL Server:** Without explanation, we note that SQL Server, but not DB2 and ORACLE, could satisfy this query objective by coding the following LIKE-clause.

```
LIKE ' _[AEIOU]%'
```

**Regular Expressions:** If you frequently have to code multiple LIKE-patterns, you may prefer to specify a "regular expression." Regular expressions (not covered in this book) offer string-processing facilities that transcend the relatively simple LIKE-patterns described in this chapter. Most relational database systems support regular expressions.



## NOT LIKE

Recall that IN can be complemented by NOT IN, and BETWEEN can be complemented by NOT BETWEEN. In a similar manner LIKE can be complemented by NOT LIKE.

The next sample query illustrates NOT LIKE. As you would expect, NOT LIKE is used to select rows with character-string values that do not match a specified LIKE-pattern.

**Sample Query 6.9:** Display all CHARNAME values that do not begin with MA. (This example retrieves rows not selected in Sample Query 6.1.)

```
SELECT CHARNAME          DB2, ORACLE & SQL Server
FROM DEMO1
WHERE CHARNAME NOT LIKE 'MA%'
```

```
CHARNAME
DAVID
SOLOMON
LUKE
JOHN
EUCLID
WASHINGTON
ADAMS
JEFFERSON
```

**Syntax & Logic:** Nothing new.

**Logical Equivalency:** An equivalent WHERE-clause is:

```
WHERE NOT CHARNAME LIKE 'MA%'
```

### Exercise:

60. Reference the PRESERVE table. Display the preserve name of any nature preserve that does not end with an E.

**Sample Query 6.10:** Display every CHARNAME value that does not have a vowel in its second position. Code three logically equivalent statements that satisfy this query objective. The result should look like:

CHARNAME  
ADAMS

```
SELECT CHARNAME          DB2, ORACLE & SQL Server
FROM DEMO1
WHERE CHARNAME NOT LIKE ' _A%'
AND CHARNAME NOT LIKE ' _E%'
AND CHARNAME NOT LIKE ' _I%'
AND CHARNAME NOT LIKE ' _O%'
AND CHARNAME NOT LIKE ' _U%'
```

**Syntax & Logic:** Nothing new. The above WHERE-clause simply AND-connects multiple NOT LIKE patterns.

---

```
SELECT CHARNAME          DB2, ORACLE & SQL Server
FROM DEMO1
WHERE NOT CHARNAME LIKE ' _A%'
AND NOT CHARNAME LIKE ' _E%'
AND NOT CHARNAME LIKE ' _I%'
AND NOT CHARNAME LIKE ' _O%'
AND NOT CHARNAME LIKE ' _U%'
```

**Syntax & Logic:** Nothing new. This WHERE-clause moves the NOT before the column-name.

---

```
SELECT CHARNAME          DB2, ORACLE & SQL Server
FROM DEMO1
WHERE NOT (CHARNAME LIKE ' _A%'
OR CHARNAME LIKE ' _E%'
OR CHARNAME LIKE ' _I%'
OR CHARNAME LIKE ' _O%'
OR CHARNAME LIKE ' _U%')
```

**Syntax & Logic:** Nothing new. This WHERE-clause applies De Morgan's Law to the first WHERE-clause.

---

**Exercise:**

6P. Reference the PRESERVE table. Display the preserve name of any nature preserve that does not end with an E and does not end with an N.

## Trailing Blanks in VARCHAR Values

In the DEMO1 table, trailing blanks appeared in the CHAR column, but trailing blanks did not appear in the VARCHAR column. (Again, this is typical of real-world data.)

Here, for the first time, we allow a VARCHAR column to contain values with trailing blanks. The first two columns in the following DEMO1A table are similar to the corresponding columns in the DEMO1 table. *The third column, UGLY, is a VARCHAR column where the SOLOMON and MADISON values have trailing blanks.*

<u>CHARNAME</u>	<u>LEN</u>	<u>VCHARNAME</u>	<u>LEN</u>	<u>UGLY</u>
SOLOMONbbb	7	SOLOMON	10	SOLOMONbbb
WASHINGTON	10	WASHINGTON	10	WASHINGTON
MADISONXXb	9	MADISONXX	9	MADISONXX
MADISONbbb	7	MADISON	8	MADISONb

Figure 6.3: Inside View of DEMO1A Table

**Sample Query 6.11:** Reference the UGLY column in the DEMO1A table. After ignoring trailing blanks, display every UGLY value that ends with ON.

```
SELECT UGLY
FROM DEMO1A
WHERE UGLY LIKE '%ON'
```

**SQL Server**

```
UGLY
SOLOMON
WASHINGTON
MADISON
```

**Logic:** Because SQL Server ignores trailing blanks, this statement return returns the correct result.

**DB2 & ORACLE:** Because DB2 and ORACLE do not ignore trailing blanks, this statement produces to following incorrect result.

```
UGLY
WASHINGTON
```

The following page presents a DB2 and ORACLE solution to this sample query.

## Preview: RTRIM Function

Chapter 12 will introduce the individual functions. Without detail explanation, we specify the RTRIM function that can be useful when dealing with trailing blanks.

The RTRIM function is used to ignore trailing blanks in CHAR and VARCHAR values. We revisit a previous Sample Query 6.11 that was problematic for DB2 and ORACLE users.

**Sample Query 6.11 (Again):** Reference the UGLY column in the DEMO1A table. After ignoring trailing blanks, display every UGLY value that ends with ON.

```
SELECT UGLY                                DB2 & ORACLE
FROM DEMO1A
WHERE RTRIM (UGLY) LIKE '%ON'
```

```
UGLY
SOLOMON
WASHINGTON
MADISON
```

**Logic:** Because we know that UGLY may contain trailing blanks, we use the RTRIM function to effectively remove these trailing blanks before the LIKE operation is applied.

**SQL Server:** Because SQL Server supports the RTRIM function, this SELECT will execute and return the correct result.

## Preview: “Length” Functions

Chapter 12 will also introduce two other individual functions, the `LENGTH` function supported by DB2 and ORACLE, and the `LEN` function supported by SQL Server. These functions can be used to determine the length of a character-string.

Without detail explanation, the following sample query illustrates these functions.

**Sample Query 6.12:** Reference the DEMO1A table. How long are the VCHARNAME and UGLY values in the DEMO1A table? Also, which of the UGLY values have trailing blanks?

**DB2 & ORACLE:** Specify the `LENGTH` function.

```
SELECT VCHARNAME, LENGTH (VCHARNAME),
       UGLY, LENGTH (UGLY)
FROM DEMO1A
```

**SQL Server:** Specify the `LEN` function.

```
SELECT VCHARNAME, LEN (CHARNAME),
       UGLY, LEN (UGLY)
FROM DEMO1A
```

Both of these SELECT statements produce the same result.

VCHARNAME	LENGTH (VCHARNAME)	UGLY	LENGTH (UGLY)
SOLOMON	7	SOLOMON	10 ←
WASHINGTON	10	WASHINGTON	10
MADISONXX	9	MADISONXX	9
MADISON	7	MADISON	8 ←

By comparing the length of each VCHARNAME value to its corresponding UGLY value, you can deduce that:

- the SOLOMON value in UGLY has 3 (10-7) trailing blanks, and
- the MADISON value in UGLY has 1 (8-7) trailing blank.

## C. DB2, SQL Server, and ORACLE

This section examines 12 SELECT statements to illustrate the differences in character-string comparison as implemented by DB2, SQL Server, and ORACLE. (This section is especially important for ORACLE users.) These statements reference the DEMO1A table redisplayed below.

<u>CHARNAME</u>	<u>LEN</u>	<u>VCHARNAME</u>	<u>LEN</u>	<u>UGLY</u>
SOLOMONbbb	7	SOLOMON	10	SOLOMONbbb
WASHINGTON	10	WASHINGTON	10	WASHINGTON
MADISONXXb	9	MADISONXX	9	MADISONXX
MADISONbbb	7	MADISON	8	MADISONb

Figure 6.3: Inside View of DEMO1A Table

Optional Exercise: Before reading the following pages, you are invited to predict the result for each statement. Based upon the material presented in the previous Section B, DB2 and SQL Server users should be able to make correct predictions. However, this should be a challenge for ORACLE users.

Statements 1-3 reference the CHARNAME column which has a CHAR(10) data-type. Notice that the CHARNAME value for SOLOMON has three trailing blanks. Each statement's objective is to display the CHARNAME value for SOLOMON. Note that Statement-2 has a literal with one trailing blank, and Statement-3 has a literal with six trailing blanks. Which statements satisfy the query objective?

1. SELECT CHARNAME FROM DEMO1A WHERE CHARNAME = 'SOLOMON';
2. SELECT CHARNAME FROM DEMO1A WHERE CHARNAME = 'SOLOMON ';
3. SELECT CHARNAME FROM DEMO1A WHERE CHARNAME = 'SOLOMON ';

Statements 4-6 reference the VCHARNAME column which has a VARCHAR(10) data-type. Notice that no VCHARNAME has any trailing blanks. Each statement's objective is to display the VCHARNAME value for SOLOMON. Note that Statement-5 has a literal with one trailing blank, and Statement-6 has a literal with six trailing blanks. Which statements satisfy the query objective?

4. SELECT VCHARNAME FROM DEMO1A WHERE VCHARNAME = 'SOLOMON';
5. SELECT VCHARNAME FROM DEMO1A WHERE VCHARNAME = 'SOLOMON ';
6. SELECT VCHARNAME FROM DEMO1A WHERE VCHARNAME = 'SOLOMON ';

Statements 7-9 reference the UGLY column which has a VARCHAR(10) data-type. Notice that the UGLY value for SOLOMON has three trailing blanks. Each statement's objective is to display the UGLY value for SOLOMON. Note that Statement-8 has a literal with one trailing blank, and Statement-9 has a literal with six trailing blanks. Which statements satisfy the query objective?

7. SELECT UGLY FROM DEMO1A WHERE UGLY = 'SOLOMON';
8. SELECT UGLY FROM DEMO1A WHERE UGLY = 'SOLOMON ';
9. SELECT UGLY FROM DEMO1A WHERE UGLY = 'SOLOMON ';

Statements 10-12 specify WHERE-conditions that compare two columns values in each row. Each statement's objective is to display a row if both columns have the same value after ignoring trailing blanks.

10. SELECT \* FROM DEMO1A WHERE CHARNAME = VCHARNAME;
11. SELECT \* FROM DEMO1A WHERE CHARNAME = UGLY;
12. SELECT \* FROM DEMO1A WHERE VCHARNAME = UGLY;

Comment: Statements 10-12 are especially relevant for application developers who code WHERE-clauses that reference a host variable (e.g., :UGLY) passed from a host program that may contain trailing blanks.

## DB2 and SQL Server

Review important observations made in the previous Section B.

- DB2: When comparing two character-strings, DB2 pad-fills the shorter string with trailing blanks so that it has the same length of the longer string.
- SQL Server: When comparing two character-strings, SQL Server ignores trailing blanks.

Although DB2 and SQL Server utilize different string comparison methods, both systems satisfy the desired query objectives.

Statements 1-9 satisfy the query objectives by returning the same 1-row result (with different column headings).

1. SELECT CHARNAME FROM DEMO1A WHERE CHARNAME = 'SOLOMON';
2. SELECT CHARNAME FROM DEMO1A WHERE CHARNAME = 'SOLOMON ';
3. SELECT CHARNAME FROM DEMO1A WHERE CHARNAME = 'SOLOMON ';

CHARNAME  
SOLOMON

4. SELECT VCHARNAME FROM DEMO1A WHERE VCHARNAME = 'SOLOMON';
5. SELECT VCHARNAME FROM DEMO1A WHERE VCHARNAME = 'SOLOMON ';
6. SELECT VCHARNAME FROM DEMO1A WHERE VCHARNAME = 'SOLOMON ';

VCHARNAME  
SOLOMON

7. SELECT UGLY FROM DEMO1A WHERE UGLY = 'SOLOMON';
8. SELECT UGLY FROM DEMO1A WHERE UGLY = 'SOLOMON ';
9. SELECT UGLY FROM DEMO1A WHERE UGLY = 'SOLOMON ';

UGLY  
SOLOMON

Statements 10-12 produce the same 4-row result.

10. SELECT \* FROM DEMO1A WHERE CHARNAME = VCHARNAME;
11. SELECT \* FROM DEMO1A WHERE CHARNAME = UGLY;
12. SELECT \* FROM DEMO1A WHERE VCHARNAME = UGLY;

<u>CHARNAME</u>	<u>VCHARNAME</u>	<u>UGLY</u>
SOLOMON	SOLOMON	SOLOMON
WASHINGTON	WASHINGTON	WASHINGTON
MADISONXX	MADISONXX	MADISONXX
MADISON	MADISON	MADISON



# ORACLE

## Preliminary Observation: VARCHAR versus VARCHAR2

Many CREATE TABLE statements in the CREATE-ALL-TABLES-ORACLE script specify a VARCHAR (instead of a VARCHAR2) data-type to define a variable-length character-string. However, after you create these sample tables, examination of your Metadata Panel will show that ORACLE automatically converts VARCHAR to VARCHAR2. ORACLE documentation states:

Do not use the VARCHAR data type. Use the VARCHAR2 data type instead. Although the VARCHAR data-type is currently synonymous with VARCHAR2, the VARCHAR data-type is scheduled to be redefined as a separate data type used for variable-length character strings...

Despite this advice, to facilitate code portability, this author specifies VARCHAR instead of VARCHAR2.

## ORACLE Character-String Comparison

Depending on the data-types of the character-strings to be compared, ORACLE uses either: (1) *blank-padded comparison semantics*, or (2) *nonpadded comparison semantics*.

1. ORACLE uses *blank-padded comparison semantics* when it compares two CHAR columns, or when it compares a CHAR column to a text literal.

This rule applies to Statements 1-3.

1. SELECT CHARNAME FROM DEMO1A WHERE CHARNAME = 'SOLOMON';
2. SELECT CHARNAME FROM DEMO1A WHERE CHARNAME = 'SOLOMON ';
3. SELECT CHARNAME FROM DEMO1A WHERE CHARNAME = 'SOLOMON ';

Hence these statements return the following desired result.

```
CHARNAME  
SOLOMON
```

2. ORACLE uses *nonpadded comparison semantics* when one or both of the compared columns is a VARCHAR data-type. This applies to Statements 4-12.

This nonpadded comparison semantics behaves quite differently than DB2 and SQL Server. Examination of results for Statements 4-12 will show that only one statement (Statement 4) satisfies the query objective and produces the same result as DB2 and SQL Server. We offer an observation about ORACLE's character-string comparison logic. (The precise description of this logic is offered in Appendix 6B.)

Observation: When comparing two character-strings where one or both character-strings have a VARCHAR data-type, ORACLE judges the strings to be equal if: (i) *both strings have the same length*, and (ii) the corresponding characters match. For example, 'JESSIE' is not equal to 'JESSIE ' because these strings have different lengths.

The following Statement-4 satisfies the query objective and produces the desired result. Here, the compared character-strings have the same length and the corresponding characters match.

```
4. SELECT VCHARNAME FROM DEMO1A WHERE VCHARNAME = 'SOLOMON';
```

```
    VCHARNAME  
    SOLOMON
```

Statement-5 and Statement-6 do not satisfy the query objective because the stored value for SOLOMON has a length of 7, and the text literals have different lengths.

```
5. SELECT VCHARNAME FROM DEMO1A WHERE VCHARNAME = 'SOLOMON ';
```

```
6. SELECT VCHARNAME FROM DEMO1A WHERE VCHARNAME = 'SOLOMON    ';
```

Result is: "No rows returned"

Likewise, Statements 7-9 do not satisfy the query objective because the stored value for SOLOMON has a length of 10, and the text literals have different lengths.

```
7. SELECT UGLY FROM DEMO1A WHERE UGLY = 'SOLOMON';
```

```
8. SELECT UGLY FROM DEMO1A WHERE UGLY = 'SOLOMON ';
```

```
9. SELECT UGLY FROM DEMO1A WHERE UGLY = 'SOLOMON    ';
```

Result is: "No rows returned"

Statements 10-12 compare two column values where at least one column (VCHARNAME or UGLY) has a VARCHAR data-type. Hence ORACLE applies the nonpadded comparison semantics.

```
10.  SELECT * FROM DEMO1A WHERE CHARNAME = VCHARNAME;
```

<u>CHARNAME</u>	<u>VCHARNAME</u>	<u>UGLY</u>
WASHINGTON	WASHINGTON	WASHINGTON

```
11.  SELECT * FROM DEMO1A WHERE CHARNAME = UGLY;
```

<u>CHARNAME</u>	<u>VCHARNAME</u>	<u>UGLY</u>
SOLOMON	SOLOMON	SOLOMON
WASHINGTON	WASHINGTON	WASHINGTON

```
12.  SELECT * FROM DEMO1A WHERE VCHARNAME = UGLY;
```

<u>CHARNAME</u>	<u>VCHARNAME</u>	<u>UGLY</u>
WASHINGTON	WASHINGTON	WASHINGTON
MADISONXX	MADISONXX	MADISONXX

The above result tables do not satisfy the query objective. The non-matching character-strings fail to match because the character-strings have different lengths.

These exercises should motivate ORACLE users to utilize the RTRIM function when referencing a VARCHAR column that may contain values with trailing blanks. For example, Statement-12 should be rewritten as:

```
SELECT * FROM DEMO1A WHERE VCHARNAME = RTRIM (UGLY);
```

## Summary

The keyword `LIKE` is used to test for a pattern in a character-string column. The general format is:

```
WHERE character-column LIKE 'pattern'
```

The pattern must be enclosed in apostrophes and may contain special wildcard characters. The percent sign (%) represents any string of any length. The underscore (\_) represents exactly one character. Unfortunately, in some circumstances, the logic of a `LIKE`-comparison may vary across different systems. Read your reference manual for details.

**Trailing Blanks:** As we have seen, trailing blanks occur frequently with `CHAR` values and rarely with `VARCHAR` values.

Sometimes, all values in a `CHAR` column will not have any trailing blanks. This applies to the `STATE` column in the `PRESERVE` table.

Some `VARCHAR` values may have trailing blanks. This applies to the `UGLY` column in `DEMO1A`. But this is a special case, and most DBAs take some action to prohibit this special case.

## Summary Exercises

The following exercises pertain to the `EMPLOYEE` table. The `ENAME` column has a `VARCHAR (25)` data-type. `ENAME` values cannot have any trailing blanks.

- 6Q. Display the name of any employee whose name of begins with the letter S.
- 6R. Display the name of any employee whose has the consecutive letters RR anywhere in his name.
- 6S. Display the name of any employee whose name of ends with the letter Y.
- 6T. Display the name of any employee whose name has the letter O in the second position.

## Appendix 6A: Efficiency

**Gaming the Optimizer:** We describe an ancient history war story that applied to a very popular database system. Assume that you wanted to display every PNAME value that begins with the letter M. Two valid WHERE-clauses are shown below.

```
Statement-1:  SELECT PNAME FROM PRESERVE
              WHERE PNAME BETWEEN 'M' AND 'MZZZ'
```

```
Statement-2:  SELECT PNAME FROM PRESERVE
              WHERE PNAME LIKE 'M%'
```

Someday you may be asked to change a very old (early 1980's) SELECT statement. You are surprised to see that the user coded something like Statement-1. Why?

One possible reason is ignorance. Maybe the user did not know about the LIKE keyword. However, there is another more interesting reason indicating that the user was quite knowledgeable. The choice of Statement-1 might have occurred within the following scenario.

1. The user knew that PRESERVE is a very big table.
2. The user knew there was an index on PNAME.
3. The user estimated that 1/26 of the PNAME values begin with the letter M.
4. Assuming this is good selectivity, the user concluded that the optimizer *should* choose to use the index on PNAME.
5. *However! The user also knew something about her system's imperfect optimizer. For some reason, this optimizer would never use an index when searching on a LIKE-condition. Yet, this optimizer would consider using an index when searching on a BETWEEN-condition.*
6. Hence, for efficiency reasons, in order to work around the weakness of the old optimizer, the user specified a BETWEEN clause.

Today, your optimizer will consider using an index for LIKE-patterns. It should generate the optimal application plan for both of the above SELECT statements.

## Appendix 6B: ORACLE Documentation

The following narrative is extracted from ORACLE's SQL reference manual.

### Nonpadded Character-String Comparison Semantics

With nonpadded semantics, Oracle compares two values character by character up to the first character that differs. The value with the greater character in that position is considered greater. If two values of different length are identical up to the end of the shorter one, then the longer value is considered greater. If two values of equal length have no differing characters, then the values are considered equal. Oracle uses nonpadded comparison semantics whenever one or both values in the comparison have the datatype `VARCHAR2` or `NVARCHAR2`.

The results of comparing two character values using different comparison semantics may vary. The table that follows shows the results of comparing five pairs of character values using each comparison semantic. Usually, the results of blank-padded and nonpadded comparisons are the same. The last comparison in the table illustrates the differences between the blank-padded and nonpadded comparison semantics.

<b>Blank-Padded</b>	<b>Nonpadded</b>
'ac' > 'ab'	'ac' > 'ab'
'ab' > 'a '	'ab' > 'a '
'ab' > 'a'	'ab' > 'a'
'ab' = 'ab'	'ab' = 'ab'
'a ' = 'a'	'a ' > 'a'

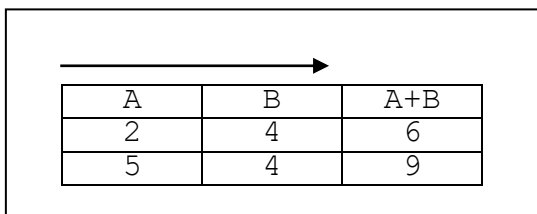
This page is intentionally blank.

# Arithmetic Expressions

Previous SELECT statements retrieved data from tables without performing any calculations on the retrieved data. This chapter introduces SELECT statements with arithmetic expressions that perform basic calculations on the retrieved data. These calculated results can be displayed in the result table.

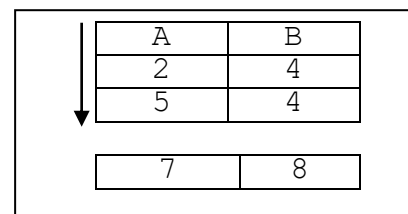
We casually describe an arithmetic expression as a meaningful combination of column names, constants, and arithmetic operators. The arithmetic operators are addition (+), subtraction (-), multiplication (\*), and division (/). The formation of a "meaningful" arithmetic expression conforms to the same rules you learned in junior high school.

In addition to arithmetic expressions, SQL Aggregate Functions (to be introduced in the Chapter 8) offer additional computational capabilities. The following figures highlight the major difference between an arithmetic expression and an aggregate function. Figure 7.1a illustrates an arithmetic expression (A+B) by drawing a horizontal arrow indicating that an expression calculates "across each row." Figure 7.1b illustrates an aggregate function (SUM) by drawing a downward vertical arrow indicating that the aggregate function calculates "down a column."



A	B	A+B
2	4	6
5	4	9

Figure 7.1a: Arith. Expression



A	B
2	4
5	4

7	8
---	---

Figure 7.1b: SUM Function

This chapter restricts its attention to arithmetic expressions. Chapters 8, 9, and 9.5 will cover aggregate functions.



## Addition – Subtraction – Multiplication - Division

The following sample query illustrates arithmetic expressions that perform addition, subtraction, multiplication, and division.

**Sample Query 7.1:** Suppose we are interested in the impact of changing admission fees for nature preserves located in Arizona. We want to perform four what-if calculations.

- What is the adjusted fee for each preserve if the current fee were increased by \$1.50?
- What is the adjusted fee for each preserve if the current fee were decreased by \$1.00?
- What is the adjusted fee for each preserve if the current fee is multiplied by 2.05?
- What is the adjusted fee for each preserve if the current fee were reduced by 50%?

For each row in the PRESERVE table, display the preserve number, the current fee, and the four adjusted fees.

```
SELECT PNO, FEE,  
        FEE + 1.50,  
        FEE - 1.00,  
        FEE * 2.05,  
        FEE / 2.0  
  
FROM PRESERVE  
  
WHERE STATE = 'AZ'
```

PNO	FEE	FEE+1.50	FEE-1.00	FEE*2.05	FEE/2.0
5	3.00	4.50	2.00	6.15	1.50
7	0.00	1.50	-1.00	0.00	0.00
80	3.00	4.50	2.00	6.15	1.50
6	3.00	4.50	2.00	6.15	1.50

**Syntax & Logic:** "FEE+1.50" is an arithmetic expression that produces the third column in the above result table. Coding this expression in the SELECT-clause asks the system to access the FEE value, perform the addition, and display the calculated result. The same process applies to the other three expressions.

Observe the row describing Preserve 80 with a FEE value of zero. Subtracting \$1.00 from this fee produces negative value.

These expressions show spaces between the arithmetic operator and the operands. This spacing may improve readability, but it is not required.

## Data-Type Considerations

**Decimal Values:** FEE is declared as a DECIMAL data-type. Notice that we specified decimal constants (1.50, 1.00, 2.05, and 2.00) in these expressions. As indicated in Chapter 1, this is a good idea.

**Integer Values:** The ACRES column, to be referenced in Sample Query 7.2, is an INTEGER data-type. Integer values can be referenced in arithmetic expressions without problems with one important exception. Sample Query 7.2 will demonstrate an important special case involving division (/) where we must be very sensitive to the data-types of the operands.

**Display Format for Calculated Values:** Most front-end tools allow you to format a calculated column (e.g., display a dollar sign). These tools will automatically round or truncate decimal values to some default decimal accuracy. Chapter 10 will present the rounding and truncation functions that provide specific rounding/truncation of calculated values.

**Column Headings for Calculated Columns:** A column containing values produced by an expression does not have any predefined column-name. Therefore, the front-end tool will automatically generate default column headings. *Different front-end tools will generate different column headings.* You can change a heading by using the reporting facilities of your front-end tool. You can also specify a "column alias" to change a column heading. This will be illustrated in Sample Query 7.3.

### Exercise:

7A. What would be the new adjusted size (acreage) of each nature preserve if its current size were doubled? Display the preserve name, current acreage, and adjusted acreage.

## Careful with Integer Division!

If an arithmetic computation involves two decimals, the result is a decimal; and, if the computation involves two integers, the result is an integer; and, if the computation involves a decimal and an integer, the result is a decimal. With the exception of one special case, this behavior does not cause any problems.

*Special Case Division: On some systems (e.g., DB2 and SQL Server) if you divide an integer by an integer, the result is an integer where the decimal component is lost. The following sample query illustrates this behavior. (Again, know-your-data.)*

**Sample Query 7.2:** For each nature preserve that is located in Montana, display the preserve's PNAME and ACRES values followed by the results of two calculations. (1) Divide the ACRES value (an integer) by 2.0 (a decimal), and (2) divide the ACRES value by 2 (an integer).

```
SELECT PNAME, ACRES,
        ACRES/2.0,
        ACRES/2
FROM PRESERVE
WHERE STATE = 'MT'
```

PNAME	ACRES	ACRES/2.0	ACRES/2
DANCING PRAIRIE	680	340.00	340
SOUTH FORK MADISON	121	60.50	60 ←
COMERTOWN PRAIRIE	1130	565.00	565
PINE BUTTE SWAMP	15000	7500.00	7500

Note the loss of decimal accuracy for the ACRES/2 calculation in the second row of the result table (121/2 = 60). Using DB2 or SQL Server, when an integer is divided by an integer, the result is an integer. This loss of accuracy does not occur with some other systems (e.g., ORACLE).

**Workaround:** Assume X and Y are INTEGER data-types, and you want to calculate X/Y. Consider the following expressions.

$X / (Y + 0.0)$       or       $X / (Y * 1.0)$

Adding 0.0 to Y, or multiplying Y by 1.0, indirectly converts Y to a decimal value. Hence these expressions produce a decimal result.

### Exercise:

7B. What would be the size of each nature preserve if its current size were reduced to one third its current size? Display the preserve name, current acreage, and adjusted acreage.

## Column Alias

This book does not focus on report formatting. However, we present the following formatting technique because it is simple, and it has another application to be described later in this book.

A "column alias" can be used to temporarily rename a column.

**Sample Query 7.3:** Display the PNAME and FEE values of preserves located in Arizona. Also display the result of adding \$1.50 to each FEE value, and display the result of subtracting \$1.00 from each FEE value. Specify a column alias to change the default heading for the addition operation to INCREASED; and specify another alias to change the default heading for the subtraction operation to DECREASED.

```
SELECT PNAME, FEE,  
       FEE+1.50 INCREASED,  
       FEE-1.00 DECREASED  
FROM PRESERVE  
WHERE STATE = 'AZ'
```

<u>PNAME</u>	<u>FEE</u>	<u>INCREASED</u>	<u>DECREASED</u>
HASSAYAMPA RIVER	3.00	4.50	2.00
MULESHOE RANCH	0.00	1.50	-1.00
RAMSEY CANYON	3.00	4.50	2.00
PAPAGONIA-SONOITA CREEK	3.00	4.50	2.00

**Syntax:** Within the SELECT-clause, the "FEE+1.00" expression is followed by a column alias (INCREASED) with a space (not a comma) between the expression and the alias. Likewise for the DECREASED alias that follows the "FEE-1.00" expression.

**Logic:** Nothing new. The alias effectively becomes the name of the column in the result table. In this example we used aliases to produce different column headings for the expressions. We could have used the same technique to change the headings for the PNAME and FEE columns.

**Column Alias is allowed in an ORDER BY Clause:** Later in this chapter, we will see that a column alias can be specified in an ORDER BY clause (e.g., ORDER BY DECREASED).

**Column Alias is not allowed in a WHERE-Clause:** Referencing a column alias in a WHERE-clause will cause an error.

```
SELECT ENO, ENAME, SALARY + 10.00 NEWSALARY  
FROM EMPLOYEE  
WHERE NEWSALARY > 2000.00 ← Error
```

Previous arithmetic expressions specified constant values. This is not required. Consider the following sample query.

**Sample Query 7.4:** For each nature preserve located in Arizona, display its name followed by the result of dividing its ACRES value by its FEE value. Exclude any row that has a FEE value of zero.

```
SELECT PNAME, ACRES/FEE
FROM PRESERVE
WHERE STATE = 'AZ'
AND FEE <> 0
```

<u>PNAME</u>	<u>ACRES/FEE</u>
HASSAYAMPA RIVER	220.00000
RAMSEY CANYON	126.66666
PAPAGONIA-SONOITA CREEK	400.00000

**Logic:** For each Arizona row with a non-zero FEE value, ACRES is divided by FEE.

**Divide-by-Zero:** The statement specifies a WHERE-clause to exclude those Arizona rows with a FEE is zero. This avoids the divide-by-zero problem (another know-your-data consideration).

What happens if you do not exclude rows where FEE equals zero? Some systems will attempt to divide by zero and return an error message. Other systems will return a null value for each zero-divide operation. Null values will be discussed in Chapter 11.

**Accuracy of Displayed Decimal Results:** The ACRES column contains integer values, and the FEE column contains decimal values. Therefore, ACRES/FEE will produce decimal results.

Some calculations produce repeating digits in the decimal part of the result (e.g., 126.6666...). The number of displayed digits will vary for each system. Your reporting tool can adjust this number. Also, SQL rounding and truncation functions will be presented in Chapter 10.

## Order of Execution for Arithmetic Operations

The SQL hierarchy of arithmetic operators is the same hierarchy you learned in junior high school. For tutorial purposes we review this hierarchy.

### Hierarchy of Arithmetic Operators:

1. Multiplication and division operations are evaluated first in a left-to-right scan of the expression.
2. Then addition and subtraction operations are evaluated in a left-to-right scan of the expression.
3. Using parentheses can change the order of evaluation. Expressions within parentheses are evaluated first.

Examples:

$10 / 5 * 2$	evaluates to 4
$10 + 5 * 2$	evaluates to 20
$10 / (5 * 2)$	evaluates to 1
$(10 + 5) * 2$	evaluates to 30
$10 + 5 * 10 + 2$	evaluates to 62
$(10 + 5) * (10 + 2)$	evaluates to 180
$(10 + 5 * 10) + 2$	evaluates to 62
$(10 + 5) * 10 + 2$	evaluates to 152

**Strong Recommendation:** When coding arithmetic expressions, specify parentheses to avoid relying on the default hierarchy of operations.

## Parentheses in Arithmetic Expressions

**Sample Query 7.5:** For each nature preserve located in Arizona, display its name followed by the result of adding 1000 to its ACRES value and then dividing the result by its FEE value. Exclude any row that has a FEE value of zero.

```
SELECT PNAME, (ACRES+1000)/FEE
FROM PRESERVE
WHERE FEE <> 0 AND STATE = 'AZ'
```

<u>PNAME</u>	<u>(ACRES+1000)/FEE</u>
HASSAYAMPA RIVER	553.33333
RAMSEY CANYON	460.00000
PAPAGONIA-SONOITA CREEK	733.33333

**Logic:** This query objective requires the addition of 1000 to ACRES before the division by FEE. For this reason, the addition operation was enclosed within parentheses. Observe what happens if you fail to specify the parentheses as shown below.

```
SELECT PNAME, ACRES+1000/FEE
FROM PRESERVE
WHERE FEE <> 0 AND STATE = 'AZ'
```

The *incorrect* result would be:

<u>PNAME</u>	<u>ACRES+1000/FEE</u>
HASSAYAMPA RIVER	993.33333
RAMSEY CANYON	713.33333
PAPAGONIA-SONOITA CREEK	1,533.33333

Removing parentheses from the expression means that division is performed first. This calculated result does not conform to the query objective.

### Exercise:

7C. For all nature preserves, display the preserve name and its current admission fee. Also display an adjusted fee that is calculated by adding \$50.00 to the current fee and then dividing by 2.

## ORDER BY an Arithmetic-Expression

An ORDER BY clause can reference an arithmetic expression or the column alias for an expression.

For example, consider the following SELECT-clause that specifies an arithmetic expression as the fourth column in a result table. MYCALC is specified as a column alias for this column.

```
SELECT COLA, COLB, COLC, (COLA+COLB)*COLC MYCALC
```

The following three ORDER BY clauses are valid.

```
ORDER BY 4                (Sort by column number)
```

```
ORDER BY (COLA+COLB)*COLC (Sort by expression)
```

```
ORDER BY MYCALC          (Sort by column alias)
```

### Examples:

The following three statements extend Sample Query 7.3. Each statement includes an ORDER BY clause that sorts the result table by the FEE+1.50 expression.

```
SELECT PNAME, FEE,
           FEE+1.50 INCREASED,
           FEE-1.00 DECREASED
FROM PRESERVE
WHERE STATE = 'AZ'
ORDER BY 2 ←
```

```
SELECT PNAME, FEE,
           FEE+1.50 INCREASED,
           FEE-1.00 DECREASED
FROM PRESERVE
WHERE STATE = 'AZ'
ORDER BY FEE+1.50 ←
```

```
SELECT PNAME, FEE,
           FEE+1.50 INCREASED,
           FEE-1.00 DECREASED
FROM PRESERVE
WHERE STATE = 'AZ'
ORDER BY INCREASED ←
```



## Calculated Conditions: Expressions in WHERE-Clauses

Previous examples specified arithmetic expressions in SELECT-clauses. An arithmetic expression can also be specified in a WHERE-clause. *A row is selected if the calculated result produces a match on the WHERE-condition.*

**Sample Query 7.6:** Assume the acreage of each nature preserve is tripled. Display a preserve's name and acreage if the size of its adjusted acreage exceeds 100,000 acres.

```
SELECT PNAME, ACRES
FROM PRESERVE
WHERE (ACRES * 3) > 100000
```

<u>PNAME</u>	<u>ACRES</u>
MULESHOE RANCH	49120

**Logic:** The condition  $(ACRES * 3) > 100000$  contains an arithmetic expression  $(ACRES * 3)$  which is evaluated for each row. If the result is greater than 100,000, the row is selected.

**Logical Equivalency:** This WHERE-clause could be rewritten as:

```
WHERE ACRES > 100000/3.0
```

```
WHERE ACRES > 33333.33
```

Appendix 7A offers some insight into why calculated conditions may enhance efficiency.

## Potential Problem: Divide-by-Zero in Calculated Conditions

There is no problem with the following WHERE-clause because the expression divides by a non-zero value (5.00):

```
SELECT PNAME, ACRES/5.00
FROM PRESERVE
WHERE ACRES/5.00 > 200.00
```

However, a divide-by-zero problem will occur with the following WHERE-clause because the expression divides by FEE where some FEE values are zero.

```
SELECT PNAME, ACRES/FEE
FROM PRESERVE
WHERE ACRES/FEE > 200.00
```

**Problematic Workarounds:** Consider and then *reject* both of the following two statements.

```
Statement-1:  SELECT PNAME, ACRES/FEE
              FROM PRESERVE
              WHERE FEE <> 0 AND ACRES/FEE > 200.00
```

This (apparently reasonable) compound-condition *may or may not* identify and avoid a divide-by-zero event. Here, by initially specifying the FEE <> 0 condition, the user intends to *initially* filter out FEE values of zero before executing the ACRES/FEE expression. Conceptually, this is a very good idea. And, it might work on your system. But, don't do it.

Author Comment: In writing this book, I executed the above Statement-1 on a very popular database system where it failed and returned a divide-by-zero error. I was initially surprised. Then, in an exploratory mindset, I executed the following Statement-2, and (surprise) it worked!

```
Statement-2:  SELECT PNAME, ACRES/FEE
              FROM PRESERVE
              WHERE ACRES/FEE > 200.00 AND FEE <> 0
```

Statement-1 failed, and Statement-2 worked as desired! But the opposite can happen on your system.

**Workarounds:** An explanation of this potential problem and effective workarounds will be presented in Sample Query 22.11 and Exercises 26P and 27P.

**Optional Theory Question:** Within one highly regarded database system, the previous Statement-1 failed while Statement-2 worked. What's going on here? Appendix 7B will address this question.

**System Response to Divide-by-Zero Event:** Your system will respond to a divide-by-zero event in one of two ways.

1. Some systems (e.g., DB2 and SQL Server) terminate execution and return a divide-by-zero error message.
2. Other systems (e.g., ORACLE) take a different approach which is described in the following scenario. Assume PRESERVE has 14 million rows, and the system has already processed 13 million rows without a divide-by-zero event. Then such an error occurs. If the system terminates with a divide-by-zero error message, then, after you (somehow) fix this problem, you have to start all over and process all 14 million rows. To avoid this situation, ORACLE returns a "null value" whenever it encounters a divide-by-zero event. Then it continues to process the SELECT statement. (Null values will be described in Chapter 11. For the moment we note that a WHERE-clause will not select a row if its condition evaluates to null.)

**Optional Theory Question:** We have described two ways a system can respond to a divide-by-zero event. Which is the better of the two ways? Appendix 7B will address this question.

## Summary

This chapter introduced arithmetic expressions to perform basic calculations. Sample queries illustrated that arithmetic expressions can be specified in either a SELECT-clause or a WHERE-clause. The next three chapters will expand upon the theme of computational SQL by introducing mathematical built-in functions.

## Summary Exercises

The following exercises reference the EMPLOYEE table. Use aliases for calculated columns.

- 7D. Assume each employee's salary is increased by 10%. Display the employee's number, name, old salary, and new salary. The result table should have four columns named, ENO, ENAME, OLDSALARY, and NEWSALARY. Sort the result by the new salary.
- 7E. Modify the previous Exercise 7D. Only display rows for those employees whose new salary exceeds \$2,000.00.
- 7F: Optional Exercise: Commentary for Sample Query 7.3 noted that you cannot reference a column alias in a WHERE-clause. The following WHERE-clause causes an error.

```
SELECT ENO, ENAME, SALARY + 10.00 NEWSALARY
FROM EMPLOYEE
WHERE NEWSALARY > 2000.00 ← Error
```

Can you speculate *why* the system does not allow a WHERE-clause to reference a column alias?

## Summary Advice for Mathematical Computations

### “Do it in SQL”

A desired calculation must be implemented within some software component. Within a database environment, we consider two major components: (1) The front-end tool (or application program), and (2) the database engine. An arithmetic expression can be executed within either component, as illustrated by the following Figure 7.2.

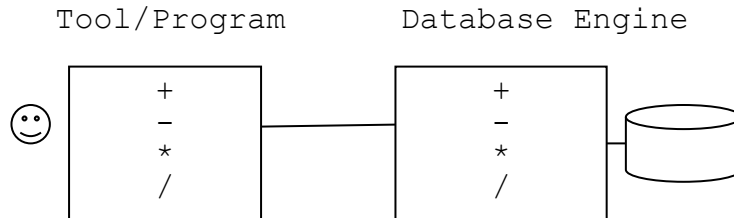


Figure 7.2: Computations within Database Environment

Excluding a few special case scenarios, we recommend executing computations in the database engine by coding SQL arithmetic expressions (or built-in functions to be described in Part II of the book).

### Advantages of Computational SQL

**Efficiency:** See Appendix 7A.

**Standardization:** There is great variety among front-end tools and programming languages, and an organization may use multiple such tools and languages. This inhibits standardization. However, because SQL is (pretty much) standardized, coding arithmetic expressions in SQL allows for some degree of standardization.

**Productivity (Do it Once - Correctly):** Don't "reinvent the wheel." Consider a large complex arithmetic expression, maybe for a sophisticated financial or scientific calculation, that will be specified in multiple SQL statements. Productivity is increased if such an expression is *correctly* coded once and reused by multiple users. Business users might save such an expression in a simple text file. An applications developer can save an expression for reuse by embedding the expression within a stored procedure/function.

**Perform Calculations in a Front-End Tool:** Execute a SELECT statement without an arithmetic expression and return the raw data to your front-end tool. Then use the tool to perform a desired calculation. There is a conceptual tidiness to this approach. The database system only retrieves data, and the tool does all the calculations. We describe two circumstances where this is approach is reasonable.

- You are in query analysis mode where your computational objectives are fuzzy. You can execute a SELECT-statement that returns data to your front-end tool so that you can “play around” with different computations. After your computational objectives have been finalized, consider implementing these computations within SQL.
- Your front-end tool may be able to perform computations that transcend SQL’s computational capabilities. For example, spreadsheets and data mining tools may perform computations that cannot be done in SQL.

## Appendix 7A: Efficiency

A SQL arithmetic expression is executed within the database engine. This may improve efficiency. We consider two examples.

### 1. Expression in the SELECT-clause

Reconsider Sample Query 7.3 which specified two expressions in the SELECT-clause.

```
SELECT PNAME, FEE,  
       FEE+1.50 INCREASED,  
       FEE-1.00 DECREASE  
FROM . . .
```

In general, efficiency may not be significantly better or worse if the tool/program executed these expressions.

### 2. Expression in the WHERE-clause (Calculated Condition)

Assume your front-end tool is running on a local computer that is connected to a remote database server via a communications network.

Reconsider Sample Query 7.6 which specified an expression in the WHERE-clause.

```
SELECT . . .  
FROM . . .  
WHERE (ACRES * 3) > 100000
```

This statement returned just one of the 14 rows (7%) in the PRESERVE table. If the remote database server could not execute the calculation condition, all rows would have to be passed from the database server to the tool/program. This would be very expensive if the PRESERVE table contained 14 million rows, especially within a slow communications environment. *Specifying an expression in a WHERE-clause reduces the number of returned rows, thereby reducing communication costs.*

**Query Rewrite:** Reconsider the  $(ACRES * 3) > 100000$  calculated condition. Assume there is a useful index on the ACRES column. On some systems, specifying a column within an expression  $(ACRES * 3)$  would disable use of the index. Therefore, you might want to rewrite the condition such that ACRES is isolated on one side of the comparison operation  $(ACRES > 100000/3)$ . A smart optimizer should automatically perform this rewrite operation.

## Appendix 7B: Theory

**Data versus Process:** Codd's original Relational Model focused on data. His database languages, the relational calculus and the relational algebra, could retrieve data, but these languages did not process the data after it was retrieved.

There was a fundamental split between data and processing. The database system retrieves data and returns the data to a program or front-end tool. The program/tool processes the data. This system architecture is reflected in the ancient history term "data processing." Non-relational first-generation database systems (e.g., DL/I and IDMS) could not do arithmetic. They could only retrieve data and return it an application program (usually written on COBOL) for subsequent processing.

This chapter illustrates that designers of SQL made a pragmatic decision to have their systems to perform some computational processing. This enhances the functionality of the language and enhances efficiency (as described in preceding Appendix 7A).

Recognizing the significant advantages of supporting computational functionality within a database language, some members of the database research community have proposed computational extensions to Codd's relational calculus and algebra. Others have proposed object-oriented database systems. A discussion of these topics is beyond the scope of this book.

**Theory Questions:** This chapter asked two optional theory questions.

1. Consider the following two compound-conditions.

FEE <> 0 AND ACRES/FEE > 200.00

ACRES/FEE > 200.00 AND FEE <> 0

Within one highly regarded database system, the first compound-condition failed by terminating the statement and returning a divide-by-zero error message. However, the second condition worked as intended by excluding FEE values of zero from the division operation. What's going on here?

The user wants to specify the following IF-THEN logic.

IF FEE <> 0 is true, THEN evaluate ACRES/FEE > 200.00

*But IF-THEN logic is not the meaning of AND logic.*

[CASE Expressions, to be introduced in Chapter 22, will show how to specify IF-THEN logic within a SQL statement.]



2. We have described two ways a system can respond to a divide-by-zero event. Which is the better of the two ways?

Some systems (e.g., DB2 and SQL Server) terminate the statement and return a divide-by-zero error message.

Other systems (e.g., ORACLE) return a "null value" whenever it encounters a divide-by-zero problem. It does not terminate the SELECT statement. Instead, it continues to process the statement. (Null values will be described in Chapter 11. Here, we note that a WHERE-clause will not select a row if its condition evaluates to null.)

This author contends that terminating the statement and returning an error-message is a better approach. Why? The mathematical definition of division explicitly excludes division by zero. Just don't do it; if you do, it's an error. Hence, to be consistent with the mathematics, the system should return an error message when it occurs.

Furthermore, regarding null values, Chapter 11 will show that null values introduce many potential problems that you would like to avoid. Finally, although the null value approach may be more efficient, we remind you that correctness trumps efficiency.

---

# PART II

## Built-in Functions & NULL Values

A function is a program that can be referenced within a SELECT statement. A “built-in” function is a function that is provided as part of the SQL language. Many built-in functions are simple and have names that indicate their purpose. For example, all database systems support the SUM function. An example of specifying this function is shown below.

```
SELECT SUM (ACRES)
FROM PRESERVE
```

It is not difficult to guess what the SUM function does. (You are invited to preview Sample Query 8.1.) All database systems provide a large number of built-in functions, but most users only need to learn details about a few functions. Your SQL reference manual contains a description of all your system’s built-in functions.

Many built-in functions are standardized in the sense that they have the same names and functionality across almost all database systems. All database systems also support a variety of system-specific functions that may be very useful, but inhibit portability of SQL code across different systems.

### Categories of Built-in Functions

This book presents two general categories of built-in functions:

- (1) Aggregate Functions, and
- (2) Individual Functions.

A possible future version of this book will present a third category of built-in functions, the OLAP Functions.

**Aggregate Functions:** Most aggregate functions “summarize” multiple values from a designated column. For example, SUM (ACRES) will return the sum of selected values from the ACRES column, and AVG (FEE) will return the average of selected values from the FEE column. Most aggregate functions are simple and easy to use. The most popular aggregate functions are presented in Chapter 8. In Chapters 9 and 9.5, these same aggregate functions will be used in conjunction with the GROUP BY and HAVING clauses.

**Terminology:** The documentation for your database system might not use the term “Aggregate” to identify this category of functions. Aggregate functions are sometimes called “Grouping” functions because they can work in conjunction with the GROUP BY clause. Alternatively, they may be called the “Column” functions because they operate on values from a column.

**Individual Functions:** An individual function works on a single value and returns a single value. (It does not “summarize” multiple values.) For example, ABS (FEE) returns the absolute value of a FEE value, and LOWER (PNAME) returns a PNAME value in lowercase letters.

**Terminology:** This book uses the term “Individual” function as a generic term. DB2 and SQL Server use the term *Scalar* function, and ORACLE uses the term *Single-Row* function.

All database systems support many individual functions. Your SQL reference manual will organize its individual functions into (sub-)categories. In this book, the individual functions are categorized as:

- Arithmetic Functions
- Character-String Functions
- Data-Type Conversion Functions
- Date-Time Functions

Again, we note there is considerable variation among the built-in functions across different database systems. This is especially true for the individual functions. Sample queries will comment on some of these differences.

## NULL Values

Chapter 11 introduces null values and describes some of the potentially confusing issues associated with processing null values.

# Aggregate Functions

Aggregate Functions are used to “summarize” selected values from a designated column. This chapter presents the following popular aggregate functions. (All systems support other aggregate functions that are described in your SQL reference manual.)

- SUM
- AVG
- MAX
- MIN
- COUNT

An aggregate function operates on a group of column values and returns a single value for each group. In this chapter, we treat the entire PRESERVE table as a single group. Sample queries and exercises will return just one row containing summary values for this group.

The following chapter will introduce the GROUP BY clause which allows you to form multiple groups of column values and calculate a result for each group.

## SUM and AVG

**Sample Query 8.1:** Display the sum of all ACRES values in the PRESERVE table.

```
SELECT SUM (ACRES)
FROM PRESERVE
```

```
  SUM (ACRES)
  -----
    70051
```

**Syntax:** SUM (numeric-column)

**Logic:** This statement does not contain a WHERE-clause. Therefore, all rows are selected, and all ACRES values are used to calculate the summary total. Although the SUM function summarizes ACRES values over 14 rows, only one row with the summary total (70051) appears in the result.

**Column Headings and Formatting:** An aggregate function has no predefined column-name. As with arithmetic expressions, different front-end tools will display different column headings and have different defaults for decimal accuracy.

**Sample Query 8.2:** Display the average acreage of the Massachusetts preserves.

```
SELECT AVG (ACRES)
FROM PRESERVE
WHERE STATE = 'MA'
```

```
  AVG (ACRES)
  -----
    293.33
```

**Syntax:** AVG (numeric-column)

**ORACLE:** The six Massachusetts rows are retrieved, and the average of their ACRES values is calculated *with decimal accuracy*.

**DB2 and SQL Server:** Because ACRES is an INTEGER data-type, DB2 and SQL Server will produce an integer result. Hence, the above statement will return 293. To obtain decimal accuracy you can execute:

```
SELECT AVG (ACRES*1.0)
FROM PRESERVE
WHERE STATE = 'MA'
```

## MIN and MAX

The following sample query illustrates that the MIN and MAX functions can accept either a numeric or a character-string argument.

**Sample Query 8.3:** Display the smallest and largest admission fees in the PRESERVE table. Also, display the smallest and largest PNAME values according to alphabetical (collating) sequence.

```
SELECT MIN(FEE), MAX(FEE), MIN(PNAME), MAX(PNAME)
FROM PRESERVE
```

MIN(FEE)	MAX(FEE)	MIN(PNAME)	MAX(PNAME)
0.00	3.00	COMERTOWN PRAIRIE	TATKON

**Syntax:** MIN (column) and MAX (column)

**Logic:** MIN (FEE) returns the smallest FEE value.

MAX (FEE) returns the largest FEE value.

MIN (PNAME) returns the smallest PNAME value within alphabetical (collating) sequence.

MAX (PNAME) returns the largest PNAME value within alphabetical (collating) sequence.

**Common Error:** You cannot specify an aggregate function in a WHERE-clause. For example, assume you want to display just those rows with an admission fee that exceeds the average fee. You might be tempted to execute the following *incorrect* statement:

```
SELECT *
FROM PRESERVE
WHERE FEE > AVG(FEE) → Error
```

Although this WHERE-clause appears to be reasonable, it generates an error because an *aggregate function, unlike an arithmetic expression, cannot be referenced in a WHERE-clause.* (Chapter 23 will resolve this limitation on the WHERE-clause.)

## COUNT Functions

We present two of the three variations of the COUNT function. (The third variation is only relevant when a column contains null values. Sample Query 11.6 will present this variation of COUNT.)

1. COUNT (\*) Counts selected rows
2. COUNT (DISTINCT column) Counts the number of distinct values in the specified column

**Sample Query 8.4.1:** Display the number of rows in the PRESERVE table. Specify CTPRESERVE as a column heading for this result.

```
SELECT COUNT (*) CTPRESERVE
FROM PRESERVE
```

```
CTPRESERVE
14
```

**Logic:** The system counts the number of retrieved rows.

Sometimes you want to count the number of rows in a subset of rows. For example, you can count the number of nature preserves in Montana by executing:

```
SELECT COUNT (*)
FROM PRESERVE
WHERE STATE = 'MT'
```

**Sample Query 8.4.2:** How many different states have nature preserves described in the PRESERVE table? Specify CTSTATE as a column heading for this result.

```
SELECT COUNT (DISTINCT STATE) CTSTATE
FROM PRESERVE
```

```
CTSTATE
3
```

**Logic:** COUNT (DISTINCT STATE) examines the STATE column to return the number of unique values in that column.

## DISTINCT with SUM and AVG

The following example illustrates that DISTINCT can be specified with the SUM and AVG functions.

**Sample Query 8.5:** Calculate the sum and the average of all distinct values in the FEE column.

```
SELECT SUM (DISTINCT FEE), AVG (DISTINCT FEE)
FROM PRESERVE
```

SUM (DISTINCT FEE)	AVG (DISTINCT FEE)
3.00	1.50

**Logic:** The FEE column contains 14 values, but only two distinct values, 0.00 and 3.00. The system used these two values to calculate the above sum and average values.

### Exercises:

- 8A. Display the average, maximum, and minimum ACRES value of all nature preserves located in Arizona.
- 8B. Display the first preserve name that appears in alphabetic sequence.
- 8C. Do not consider zero admission fees. How many distinct fees are present in the PRESERVE?
- 8D. Write a SELECT statement to demonstrate that PNAME does not currently contain any duplicate values.

**Suggestion:** Take a five-minute detour. We have just introduced five popular built-in functions: SUM, MAX, MIN, AVG, and COUNT. *Your system supports many other built-in functions.* You can get some idea of just how many functions by searching the web. For example, if you are using ORACLE, search for:

ORACLE built-in functions

Likewise for other database systems. Don't bother with details. Simply scan the documentation describing your system's built-in functions.



## Statistical Functions

Most systems support more sophisticated statistical functions such as those presented in a college course on probability and statistics.

The next sample query illustrates the VARIANCE (variance) and STDDEV (standard deviation) functions that are supported by DB2 and ORACLE. While statisticians will appreciate these functions, many other users will never use them. We illustrate these functions without explaining their statistical significance or underlying computations. Your reference manual will present these details.

**Sample Query 8.6:** Display the variance and standard deviation of all ACRES values in the PRESERVE table.

```
SELECT VARIANCE (ACRES), STDDEV (ACRES)      DB2 & ORACLE
FROM PRESERVE
```

VARIANCE (ACRES)	STDDEV (ACRES)
163733859	12795

**DB2:** Decimal accuracy may be gained by coding:

```
SELECT VARIANCE (ACRES*1.00), STDDEV (ACRES*1.00)
```

Also, DB2 will return values in scientific notation.

**SQL Server:** Substitute VARP for VARIANCE, and STDEVP for STDDEV.

```
SELECT VARP (ACRES), STDEVP (ACRES)
FROM PRESERVE
```

Many systems support other statistical functions such as COVARIANCE, CORRELATION, and functions related to regression analysis. Consult your SQL reference manual for details.

## Combining Aggregate Functions & Arithmetic Expressions

Aggregate functions can be combined with arithmetic expressions to perform more complex calculations. A function may be applied to an intermediate-result produced by executing an expression; and, an expression can specify an aggregate function as an operand.

**Sample Query 8.7:** Calculate and display two summary totals.

The first total is the sum of all admission fees assuming that each fee has been increased by \$5.00.

The second total is the result of adding \$5.00 to the sum of all admission fees.

```
SELECT SUM (FEE + 5.00), SUM (FEE) + 5.00
FROM PRESERVE
```

<u>SUM(FEE + 5.00)</u>	<u>SUM(FEE) + 5.00</u>
79.00	14.00

**Syntax:** Nothing new.

SUM (FEE + 5.00) specifies an aggregate function (SUM) that has an expression (FEE + 5.00) as an argument.

SUM (FEE) + 5.00 specifies an arithmetic expression that has the SUM function as an operand.

**Logic:** The placement of parentheses impacts the sequence of operations that produces different results.

SUM (FEE + 5.00): The system will first evaluate the expression (FEE + 5.00) for each selected row. Then apply the SUM function to these values.

SUM (FEE) + 5.00: SUM (FEE) is evaluated first to produce 9.00 which is incremented by 5.00 to produce the final result of 14.00.

**Exercise:**

8E. Assume that you intend to establish a new policy for calculating admission fees. Each nature preserve will charge a fee equal to \$0.02 per acre. What will be the average admission fee for the Arizona preserves?

## Displaying Detail-Lines with a Summary Total

An aggregate function only returns a summary total. It does not return the raw data used to calculate this total. What if you want to display both the raw data along with the summary total? Good idea, but not yet, because you *do not know enough SQL to satisfy this query objective*. Consider the following example.

**Example:** Produce a report where each "detail row" displays the PNO and ACRES values for a Montana preserve, followed by a "summary row" that displays the total acreage of all the Montana preserves. The result should look like:

<u>MTPNO</u>	<u>ACRES</u>	
1	1130	(detail row)
2	15000	(detail row)
3	680	(detail row)
40	121	(detail row)
	16931	(summary row)

We consider four possible solutions.

Sol-1

Display the detail rows by executing.

```
SELECT PNO MTPNO, ACRES
FROM PRESERVE
WHERE STATE = 'MT'
```

Use your front-end tool's computational capabilities to produce the summary row.

Sol-2

Execute the following two statements.

```
SELECT PNO MTPNO, ACRES
FROM PRESERVE
WHERE STATE = 'MT';
```

```
SELECT SUM (ACRES)
FROM PRESERVE
WHERE STATE = 'MT';
```

Do some manual labor. Cut-and-paste the two result tables to produce the desired report.

Sol-3

Use the ALL option for the UNION operation to be presented in Chapter 21.

Sol-4

Use the ROLLUP option for the GROUP BY clause to be previewed at the end of Chapter 9.5.

Sol-1 is a popular (but not the best) solution. Here, the database gets the data, and the tool does the arithmetic. But, (i) you must learn your tool's computational facilities, and (ii) you will sacrifice the previously described advantages associated with "doing everything in one SQL statement."

Sol-2 is bad. Cut-and-past is ugly. Also, you will sacrifice the previously described advantages associated with "doing everything in one SQL statement." This solution only has one advantage; you already know enough SQL to do it now.

SOL-3 is an acceptable (but not the best) solution. However, it will generate the desired result by executing one SELECT statement. Chapter 21 will present this solution.

SQL-4 is the best solution. Sample Query 9.21 in Chapter 9.5 will present this solution.

## Summary

This chapter introduced the most popular aggregate functions: SUM, AVG, MAX, MIN, and COUNT. These functions operate over a group of values from a column. In this chapter we restricted our attention to a single group corresponding to the entire table. The next chapter will introduce the GROUP BY clause that can be used to form multiple groups such that an aggregate function can be applied to each group.

## Summary Exercises

The following exercises pertain to the EMPLOYEE table.

- 8F. Display the sum, average, maximum, and minimum of all SALARY values.
- 8G. How many employees work in Department 20?
- 8H. How many departments have employees?

## Appendix 8A: Efficiency

**Computational Benefits:** Appendix 7A identified the advantages of arithmetic expressions. The basic idea is that it is usually more efficient to perform calculations within the database engine versus the front-end tool/program. This same idea is especially relevant for aggregate functions.

Every sample query in this chapter returned just one row. (The query objectives presumed the users only wanted the summarized results; they did not need to see the detail data from the individual rows.) There is *significant performance advantage to having the database execute aggregate calculations* and send just one row back to the tool/program. If this were not possible, the system would have to send all the required data to the tool/program. Consider the following statement.

```
SELECT SUM (ACRES)
FROM PRESERVE
```

Assume your front-end tool is remotely located from the database engine. What if PRESERVE contained 14 million rows? If the database engine could not perform the SUM calculation, it would have to send 14 million ACRES values down the communications network to your front-end tool.

**Multiple SELECT Statements:** The section "Displaying Detail-Lines with a Summary Total" commented on the situation where a user wants to display both the raw data and the summary totals. Solution Sol-2 executed two SELECT statements. From an efficiency perspective, this approach is *not* desirable. Two independent statements require two trips to the database engine, involving two scans of PRESERVE. This is obviously redundant, and it would be very inefficient if PRESERVE were a very large table.

**Query Optimization - Index-Only Search:** Consider the following two statements.

- SELECT COUNT (\*) FROM PRESERVE  
WHERE STATE = 'MT'

The optimizer can satisfy this statement by *only* accessing the INDSTATE index. (See Figure A1.1.)

- SELECT COUNT (\*), COUNT (DISTINCT PNO),  
MIN (PNO), MAX (PNO)  
FROM PRESERVE

The optimizer can satisfy this statement by *only* accessing the XPNO index. (See Figure A2.1).

# GROUP BY Clause:

## Grouping by a Single Column

This chapter introduces the GROUP BY clause that is used to form groups of rows such that an aggregate function can be applied to each group. For example, you might wish to select all PRESERVE rows, group these rows by STATE values, and then apply SUM (ACRES) to each group to produce a result table that looks like:

<u>STATE</u>	<u>SUM(ACRES)</u>
AZ	51360
MA	1760
MT	16931

This chapter's sample queries will specify a GROUP BY clause in conjunction with an aggregate function to:

1. Organize selected rows into groups, and
2. Apply the aggregate function to each group

**Author Advice:** When using this book to teach SQL, students rarely have trouble with the first eight chapters. They do most of the exercises in a very short time, and most errors are caused by simple typos. Things usually change when we get to this chapter. Students take more time with the exercises, and more errors are conceptual. Therefore, I suggest that you proceed at a slower pace.

## GROUP BY Clause

The following sample query illustrates the formation of groups based upon STATE values.

**Sample Query 9.1:** For each state referenced in the PRESERVE table, display the total acres of all preserves within the state.

```
SELECT STATE, SUM (ACRES)
FROM PRESERVE
GROUP BY STATE
```

STATE	SUM(ACRES)
AZ	51360
MA	1760
MT	16931

**Syntax:** SELECT     **grouping-column**, aggregate-function  
FROM            table-name  
WHERE           condition  
**GROUP BY**     **grouping-column**

**Logic:** The GROUP BY clause places selected rows into groups such that all rows with the same grouping value are placed into the same group. Here, because there are three distinct STATE values (AZ, MA, and MT), three groups are formed to produce an intermediate-result table that conceptually looks like:

STATE	ACRES
AZ	660
AZ	49120
AZ	380
AZ	1200
- - - - -	
MA	66
MA	40
MA	830
MA	4
MA	730
MA	90
- - - - -	
MT	680
MT	121
MT	1130
MT	15000

Then the SUM function is applied to each group, and summary totals are displayed.

### Exercise:

9A. For each state referenced in the PRESERVE table, display the ACRES value of the smallest nature preserve within the state.

## WHERE with GROUP BY

The WHERE-clause can be used to include or exclude specified rows *prior to* the formation of groups. The following sample query illustrates this behavior.

**Sample Query 9.2:** Modify the preceding sample query such that only rows with PNO values less than 12 are selected for consideration.

```
SELECT STATE, SUM (ACRES)
FROM PRESERVE
WHERE PNO < 12
GROUP BY STATE
```

STATE	SUM(ACRES)
AZ	50980
MA	924
MT	16810

**Logic:** The WHERE-clause initially selects rows with PNO values less than 12. This filtering of rows occurs *before* the formation of the groups. Then, using the STATE and ACRES values from the selected rows, groups are formed for each STATE value as illustrated below.

STATE	ACRES
AZ	660
AZ	49120
AZ	1200
- - - - -	-
MA	830
MA	4
MA	90
- - - - -	-
MT	680
MT	1130
MT	15000

Then the SUM function is applied to each group, and summary totals are displayed.

### Exercise:

9B. Do not consider any nature preserve that has more than 10,000 acres. For each state referenced in the PRESERVE table, display the ACRES value of the largest nature preserve within the state.



## Sorting by Group Totals

A result table can be displayed in sequence by group totals.

**Sample Query 9.3:** Display the result table from the preceding sample query. Sort this result in ascending sequence by the summary totals.

```
SELECT STATE, SUM (ACRES)
FROM   PRESERVE
WHERE  PNO < 12
GROUP BY STATE
ORDER BY 2
```

<u>STATE</u>	<u>SUM(ACRES)</u>
MA	924
MT	16810
AZ	50980

**Syntax:** On most systems, the above ORDER BY clause can be changed to explicitly reference an aggregate function as illustrated below.

```
SELECT STATE, SUM (ACRES)
FROM   PRESERVE
WHERE  PNO < 12
GROUP BY STATE
ORDER BY SUM (ACRES)
```

Also, on most systems, the ORDER BY clause can reference a column-alias as illustrated below.

```
SELECT STATE, SUM (ACRES) TOTAL_ACRES
FROM   PRESERVE
WHERE  PNO < 12
GROUP BY STATE
ORDER BY TOTAL_ACRES
```

**Incidental Sort:** If you omit the ORDER BY clause, your result table may show an incidental sort by the grouping column, as illustrated by the previous two sample queries. (Appendix 9A comments on this behavior.)

### Exercise:

9C. Same as preceding Exercise 9B. Sort the result by the maximum values in descending sequence.

## Groups with One Row

Sometimes a particular group happens to contain just one row. You may want to be aware of this circumstance.

**Sample Query 9.4:** Modify Sample Query 9.2. Here, we are only interested in rows where PNO values are less than 10. Also, display a count of the number of rows in each group.

```
SELECT STATE, SUM (ACRES), COUNT (*)
FROM PRESERVE
WHERE PNO < 10
GROUP BY STATE
```

STATE	SUM (ACRES)	COUNT (*)	
AZ	50980	3	
MA	830	1	←
MT	16810	3	

**Logic:** The MA group has just one row. Note that its ACRES value is 830. This observation could help an unethical user violate the following confidentiality rule.

**Violate Confidentiality:** Assume that, for reasons of confidentiality, you have been told not to display any PNO value along with its corresponding ACRES value. (Within the EMPLOYEE table, this is similar to displaying an ENO value along with the corresponding SALARY value which is confidential data).

After examining the above result table, an unethical user could execute the following statement that does not violate this confidentiality rule.

```
SELECT PNO FROM PRESERVE
WHERE STATE = 'MA' AND ACRES = 830
```

Result shows:  $\frac{PNO}{9}$

This user can now *deduce* that Preserve 9 has 830 acres. This is a simple example of a more general database problem whereby a user may be able to deduce confidential data by displaying statistical summaries.

## Potential Groups with No Rows

We have shown that a group can contain just one row. What about a group with no rows? We refer to this circumstance as a "potential group" (another unofficial term). The following sample query specifies a WHERE-clause that eliminates all rows from a potential group.

**Sample Query 9.5:** Display the state code and average admission fee for any state with a nature preserve that is larger than 1,000 acres.

```
SELECT STATE, AVG (FEE)
FROM PRESERVE
WHERE ACRES > 1000
GROUP BY STATE
```

<u>STATE</u>	<u>AVG(FEE)</u>
AZ	1.50
MT	0.00

**Logic:** This result table does not contain a row for the MA group. It just so happens that the WHERE-clause eliminates (perhaps unintentionally) all the Massachusetts rows. Hence, this potential group is not formed and does not appear in the result table.

You might prefer to display a result that looks something like:

<u>STATE</u>	<u>AVG(FEE)</u>
AZ	1.50
MT	0.00
MA	no-rows-selected

You must wait until Chapter 21 to learn how to do this.

## HAVING-Clause

In the following sample query, we want to select all rows, form groups, perform summary calculations for each group, and *then display a group's summary total only if this total matches some condition*. The HAVING-clause is used for this purpose. This clause examines each group's summary total and displays this total if it matches a specified condition.

**Sample Query 9.6:** For each state referenced in the PRESERVE table, calculate the total acreage for all its preserves. Then display each STATE and its total acreage only if that total exceeds 15,000 acres.

```
SELECT STATE, SUM (ACRES)
FROM PRESERVE
GROUP BY STATE
HAVING SUM (ACRES) > 15000
```

<u>STATE</u>	<u>SUM(ACRES)</u>
AZ	51360
MT	16931

**Syntax:** HAVING condition

When specified, *the HAVING-clause must immediately follow a GROUP BY clause*. The condition must reference some column in the intermediate-result table produced by the GROUP BY clause.

**Logic:** The first three clauses in this SELECT statement are identical to the SELECT statement shown in Sample Query 9.1.

```
SELECT STATE, SUM (ACRES)
FROM PRESERVE
GROUP BY STATE
```

This statement produces the following result.

<u>STATE</u>	<u>SUM(ACRES)</u>
AZ	51360
MA	1760
MT	16931

Here, this result constitutes an intermediate-result table. The HAVING-condition, `SUM (ACRES) > 15000`, selects just the AZ and MT groups for display in the final result. Observe that the MA group was formed and the sum of its acres was calculated. However, this group was not displayed because its total acreage (1760) did not exceed 15000.

## WHERE & HAVING

The following sample query highlights the difference between the WHERE and HAVING clauses. Again, the WHERE-clause initially selects rows for inclusion into groups, and the HAVING-clause subsequently selects just certain groups for display.

**Sample Query 9.7:** Do not consider Montana preserves. Display each state and its total acreage if that total acreage exceeds 15,000 acres.

```
SELECT STATE, SUM (ACRES)
FROM PRESERVE
WHERE STATE <> 'MT'
GROUP BY STATE
HAVING SUM (ACRES) > 15000
```

STATE	SUM(ACRES)
AZ	51360

**Logic:** The system executes the following sequence of operations.

The WHERE-clause excludes Montana preserves (even though Montana's total acreage happens to exceed 15,000 acres).

Two groups are formed for the Arizona and Massachusetts preserves, and the SUM (ACRES) is calculated for each group.

Finally, the HAVING-clause only returns the Arizona group because, unlike the Massachusetts group, its total exceeds 15,000.

**Observation:** Although the above SELECT statement is only five lines of code, there is "a whole lot of action going on" in this code. This motivates the author's advice on the first page of this chapter. For some SQL rookies, it takes a little more effort to internalize within the mind's eye the sequence of processing operations associated with the above SELECT statement.

**Summary Outline:** Figure 9.1 (following page) outlines the sequence of operations that are specified by this SELECT statement.

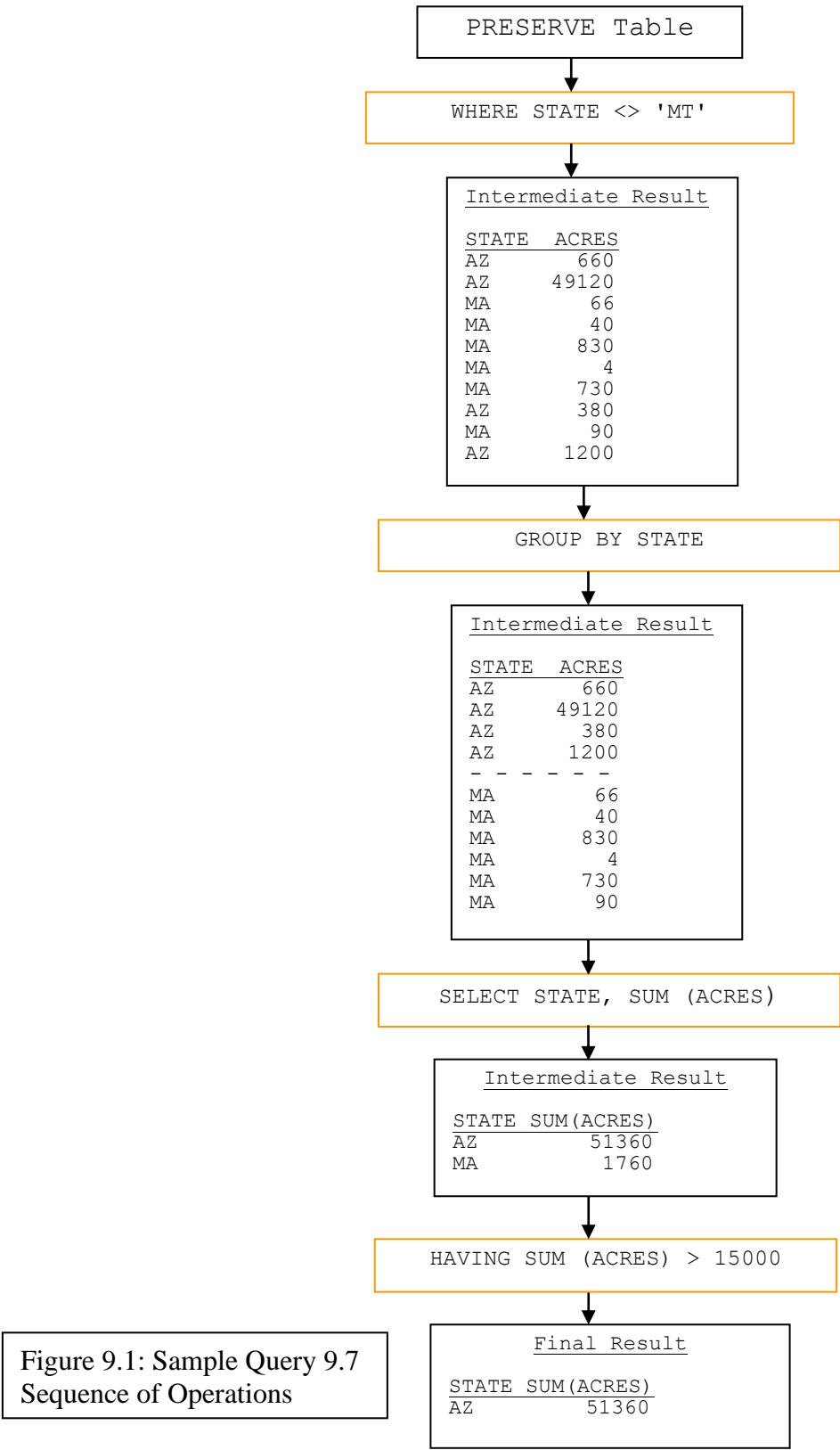


Figure 9.1: Sample Query 9.7  
Sequence of Operations

## Potential Confusion: HAVING versus WHERE

Sometimes, SQL rookies become confused over the difference between the HAVING-clause versus the WHERE-clause. Both of these clauses specify conditions, but they serve very different purposes.

Figure 9.1 shows that the WHERE-clause specifies a condition for row selection that is executed *before* the GROUP BY and HAVING clauses. Any row that is not selected by the WHERE-clause will not be placed into a group. The HAVING-clause comes into play *after* the groups have been formed and after the aggregate function has been executed. The HAVING-clause determines which groups are to be selected for display.

**Common Error:** In the previous Sample Query 9.7, the HAVING-clause correctly references the SUM function.

```
HAVING SUM (ACRES) > 15000
```

However, sometimes a SQL rookie fails to specify the function-name (SUM) and codes something like:

```
HAVING ACRES > 15000 → Error
```

An error occurs because group totals do not contain individual ACRES values.

### Exercises:

- 9D. Display the size of the largest ACRES value protected by a nature preserve within each state if that value is less than 25,000 acres.
- 9E. For each state referenced in the PRESERVE table, display the number of acres in the state's smallest preserve if that number is less than 100.
- 9F. Only consider nature preserves that have more than 1,000 acres. For each state, display its name and the size of the largest ACRES value for a nature preserve within each state if that value is less than 25,000 acres.

## Common Error: Grouping by Unique Column

Sometimes a careless user groups by a unique column (or a unique combination of columns). Consider the following example.

```
SELECT PNO, SUM (ACRES)
FROM PRESERVE
GROUP BY PNO
ORDER BY PNO
```

PNO	SUM(ACRES)
1	1130
2	15000
3	680
5	660
6	1200
7	49120
9	830
10	90
11	4
12	730
13	40
14	66
40	121
80	380

Because the syntax is valid, the statement executes and produces a result table. However, the result is not very useful because it does not display any new information.

Recall the PNO column is unique within the PRESERVE table, and recall that PRESERVE has 14 rows. Therefore, the above statement forms 14 groups where each group has one row, and the SUM function only summarizes over one ACRES value in each group. Such grouping is not reasonable.

The following statement produces the same result without grouping and summarizing.

```
SELECT PNO, ACRES
FROM PRESERVE
ORDER BY PNO
```



## HAVING-Clause with Boolean Connectors

Chapter 4 introduced more complex WHERE-clauses that specified Boolean connectors (AND, OR, NOT). In a similar manner, a HAVING-clause can also specify Boolean connectors.

**Sample Query 9.8:** Display the minimum and maximum acreage values for nature preserves located in each state if the minimum ACRES value is greater than 10 acres and the maximum ACRES value is less than 20,000 acres. Display the state code along with its minimum and maximum acreage values.

```
SELECT STATE, MIN (ACRES), MAX (ACRES)
FROM PRESERVE
GROUP BY STATE
HAVING MIN (ACRES) > 10 AND MAX (ACRES) < 20000
```

<u>STATE</u>	<u>MIN (ACRES)</u>	<u>MAX (ACRES)</u>
MT	121	15000

**Syntax & Logic:** The same rules of logic apply to both the HAVING and WHERE-clauses.

In the above statement, if we ignore the HAVING-clause, the first three clauses produce the following intermediate-result table.

<u>STATE</u>	<u>MIN(ACRES)</u>	<u>MAX(ACRES)</u>
AZ	380	49120
MA	4	830
MT	121	15000

When the system applies the HAVING-clause, only the Montana (MT) row matches both of its conditions.

### Exercise:

9G. Display the state and total size of the preserves located in the state if the total size is greater than or equal to 10,000 acres and less than or equal to 50,000 acres.

## Nesting Aggregate Functions

The following sample query nests an aggregate function (SUM) within another aggregate function (MIN). Some systems (e.g., ORACLE) allow this kind of nesting. Other systems do **not** allow this kind of nesting.

**Sample Query 9.9:** Determine the total acreage for each state, and then display the smallest of these totals.

```
SELECT MIN (SUM (ACRES))      ORACLE
FROM PRESERVE
GROUP BY STATE
```

```
MIN (SUM (ACRES))
      1760
```

**Logic:** Consider executing the above statement without the MIN function as shown below.

```
SELECT SUM (ACRES)
FROM PRESERVE
GROUP BY STATE
```

The result would look like:

```
SUM (ACRES)
  51360
   1760
  16931
```

This result becomes an intermediate-result table that is processed by the MIN function.

**Other Systems:** If your system does not allow the nesting of aggregate functions, Chapters 26 and 27 will present alternative methods to satisfy this query objective. (Exercises 26G and 27G will solve a similar query objective.)

### Exercise:

9H. *If your system allows the nesting of aggregate functions, then determine the average number of preserve acres for each state and display the largest of these averages.*

## Subtotals & Final Total

Revisit Sample Query 9.1 and its result shown below.

SELECT STATE, SUM (ACRES) FROM PRESERVE GROUP BY STATE	<table><thead><tr><th>STATE</th><th>SUM(ACRES)</th></tr></thead><tbody><tr><td>AZ</td><td>51360</td></tr><tr><td>MA</td><td>1760</td></tr><tr><td>MT</td><td>16931</td></tr></tbody></table>	STATE	SUM(ACRES)	AZ	51360	MA	1760	MT	16931
STATE	SUM(ACRES)								
AZ	51360								
MA	1760								
MT	16931								

The totals produced by the SUM function can be interpreted as subtotals of a final grand total. Users frequently ask, "How can I display the final total along with the subtotals?" These users want a result table that looks like:

STATE	SUM(ACRES)
AZ	51360
MA	1760
MT	16931
	<b>70051</b>

We cannot (yet) present a single SELECT statement that satisfies this query objective. This objective is similar the "Displaying Detail-Lines with a Summary Total" topic that was described in Chapter 8. There we presented four general solutions that are also applicable here.

The two good solutions, SOL-3 and SOL-4, must be postponed until later in this book.

Solution SOL-3 will specify the ALL option with the UNION keyword. See Chapter 21.

Solution SOL-4 (the best solution) will specify the ROLLUP clause within the GROUP BY clause. See Chapter 9.5.

Given your current lack of knowledge about UNION and ROLLUP, the following imperfect solutions, SOL-1 and SOL-2, can serve as workarounds.

Solution SOL-1: Execute the above statement to get the subtotal rows. Then use your front-end tool's computational facilities to produce the final summary row.

Solution SOL-2: Execute two independent SELECT statements. Then cut-and-paste to merge the results.

The same advantages/disadvantages for each solution that were described in Chapter 8 apply here.

## Summary

Thus far, we have presented the following clauses within our generic SELECT statement. All clauses are optional except the SELECT-clause and FROM-clause. *These clauses must be specified in the top-down order shown below.*

```
SELECT [DISTINCT] column, aggregate-function
FROM table
[WHERE condition (s)]
[GROUP BY grouping-column]
[HAVING condition (s)]
[ORDER BY column(s)]
```

**GROUP BY Clause:** The GROUP BY clause forms groups of rows based upon values from a specified column. (The following Chapter 9.5 will show that a GROUP BY clause can reference multiple columns.) This clause is specified when you want to apply an aggregate function to each group.

**HAVING-Clause:** When specified, a HAVING-clause must immediately follow a GROUP BY clause. The HAVING-clause selects specific groups for display. The HAVING-clause may also include Boolean connectors. If the HAVING-clause is not present, all groups are selected for display.

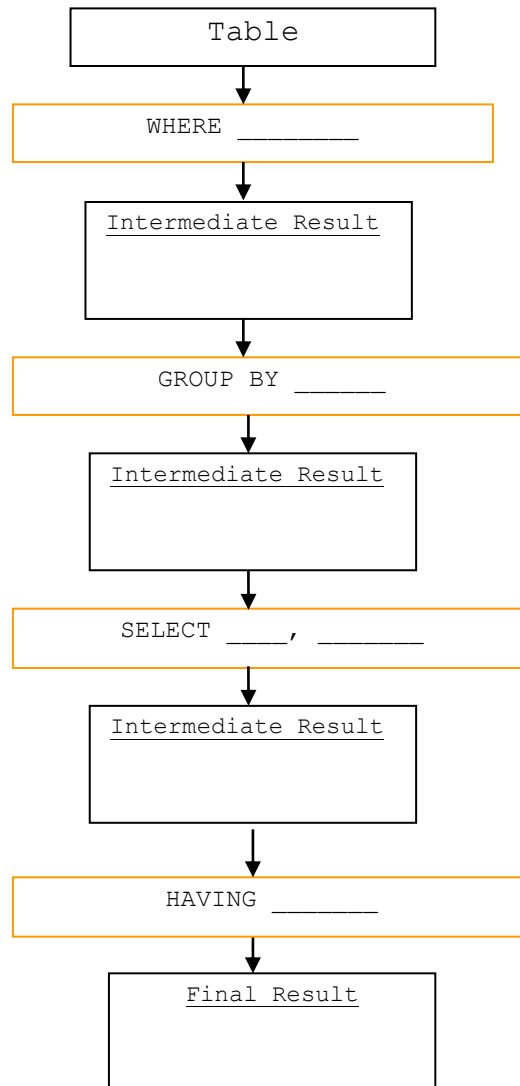
### Recommendation:

1. Review the above generic SELECT statement.
2. Review Figure 9.1 that illustrated the sequence of operations for Sample Query 9.7.
3. Examine Figure 9.2 (on the following page) that generalizes Figure 9.1

## Summary: Sequence of Operations

The following Figure 9.2 is important. It describes the sequence of operations associated with each clause in a SELECT statement (as presented thus far in this book).

Figure 9.2  
Sequence of Operations



## Author Suggestion: Careful! Read at your own Risk

I thought twice about including the following observations in this book.

When working with SQL rookies who were experiencing some difficulty with this chapter, I resorted to making the following observations. Sometimes, the "light went on," and there was an "ah ha" moment. Sometimes, I only added to the confusion.

The preceding Figure 9.2 outlines the way I *think* when *designing* (not coding) a SELECT statement that involves grouping and summarizing data.

Similar to Figure 9.2, I *think* in the following sequence.

FROM table	→ specify desired table
WHERE condition(s)	→ select desired rows
GROUP BY group-col	→ form groups with selected rows
SELECT group-col, agg-fn	→ calculate group summary totals
HAVING condition	→ select desired groups

Finally, I code the above clauses in the required sequence.

```
SELECT group-col, agg-function
FROM table
WHERE condition
GROUP BY group-col
HAVING condition
```

## Summary Exercises

The following exercises pertain to the EMPLOYEE table.

- 9I. For all department DNO values found in the EMPLOYEE table, display the DNO value followed by the average SALARY for that department.
- 9J. For all department DNO values found in the EMPLOYEE table, display the DNO value followed by the sum, maximum, and minimum of SALARY values for that department.
- 9K. Consider all departments except for Department 40. For these departments, display their DNO value followed by the number of employees who work in that department.
- 9L. For each department, determine the total number of employees who work in that department. Assume the SALARY column contains confidential data, and that someone could deduce this confidential data by examining the total of each department's salaries. Therefore, you decide to display only those departments where the employee count exceeds two.
- 9M. Outline a poor man's cut-and-paste solution that can be used to produce the following result. (A much better solution will be described later in this book.)

STATE	SUM(ACRES)
AZ	51360
MA	1760
MT	16931
TOTAL	<b>70051</b>

## Appendix 9A: Efficiency

**Reduced Communication Costs:** In Appendix 8A we emphasized the performance advantages of having the database engine perform calculations and send the result to the front-end tool or program. Similar efficiency advantages apply the GROUP BY and HAVING clauses used in conjunction with the aggregate functions.

For example, if an average group contains 20 rows, then, after grouping and summarizing, the result table is 5% (1/20) of the size of the table. Returning a result that is only 5% the size of the table is far more efficient than returning all rows so that the grouping and summarizing are done by a front-end tool/program.

**Potential Sort Costs:** Appendix 2A described sorting costs associated the ORDER BY clause; and, Appendix 3A noted that DISTINCT might produce an internal sort to detect duplicate rows. Similar observations also apply to the GROUP BY clause which might utilize an internal sort to form groups.

**Incidental Sort:** With the exception of Sample Query 9.3, all examples in this chapter did not include an ORDER BY clause. However, you *may* have observed that result tables were incidentally sorted by the grouping-column. This *may* occur if the optimizer decides to sort selected rows to facilitate the formation of groups. Hence, GROUP BY may incur incidental sort costs. (Internal sorting for the purpose of grouping does not always happen. Therefore, you should always specify an ORDER BY clause if you want rows displayed in some desired row sequence.)

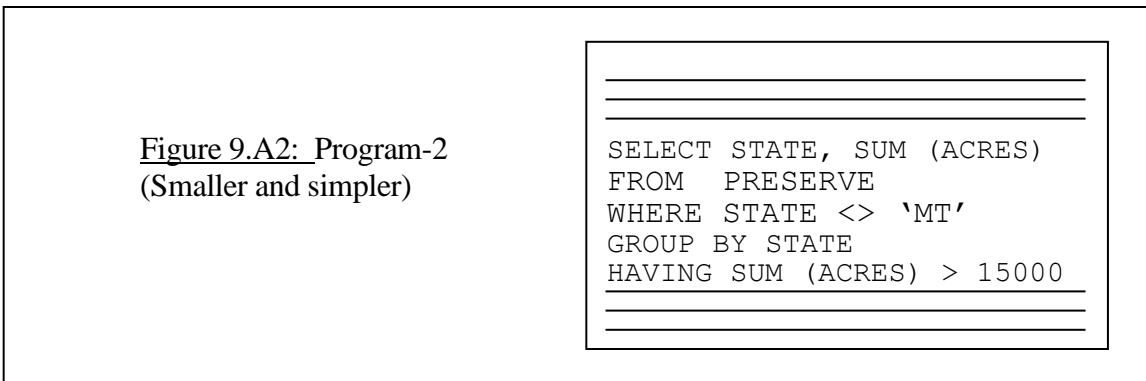
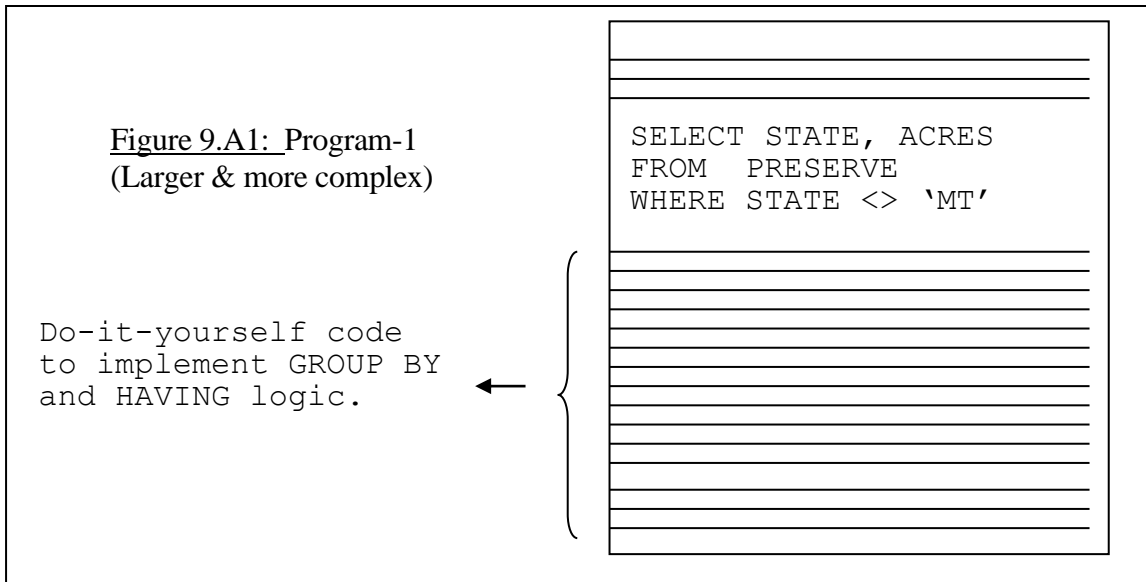
**Application Program Efficiency & Simplicity:** Unfortunately, some application programmers who write embedded SQL fail to capitalize on the advantages provided by the GROUP BY and HAVING clauses.

Assume that the PRESERVE table has 14 million rows. Consider two programs written in some programming language (e.g., JAVA, COBOL) with embedded SELECT statements. Both programs want to satisfy the query objective for Sample Query 9.7.

Program-1 (Figure 9.A1 on next page) is coded by a programmer who never learned, or forgot about, the GROUP BY and HAVING clauses. Therefore, Program-1 satisfies the query objective by coding an embedded SELECT statement that does not capitalize on GROUP BY and HAVING.

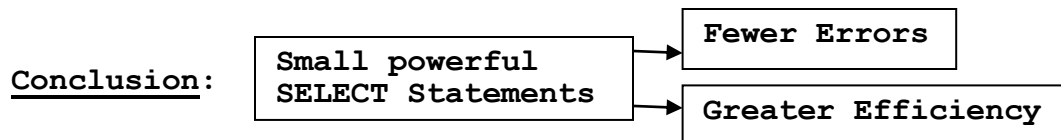
Program-2 (Figure 9.A2 on next page) has the same SELECT statement shown in Sample Query 9.7. This statement specifies a GROUP BY and HAVING clause to perform the grouping operations, summary calculations, and group selection.





**Program-2 is smaller and simpler.** Hence, it is less prone to errors. Program-1 has a greater chance of error because the programmer's logic for grouping, summarizing, and group selection could be faulty.

**Program-2 is faster.** When Program-2 finishes, no more than 49 rows (for all states except Montana) have been returned to the program. Program-1 is very inefficient. The SELECT statement would return approximately 98% (49/50) of the 14 million rows! (The cursor in Program-1 iterates millions of times, while the cursor in Program-2 only iterates 49 times.)



## Grouping by Multiple Columns

### GROUP BY Clause:

This *optional* chapter, which goes deeper into the GROUP BY clause, is a continuation of the preceding Chapter 9. (This chapter starts with Sample Query 9.10.) Specifically, this chapter presents grouping and summarizing by multiple columns where the GROUP BY clauses look like:

```
GROUP BY COLX, COLY
```

```
GROUP BY COLZ, COLY, COLX
```

Many users simply retrieve data and only “look at it.” Occasionally, these users do just a “little bit of arithmetic” using the SQL facilities already presented in Chapters 7-9. These users can safely skip this chapter without any loss of continuity. You should read this chapter if your grouping and summarizing objectives are more complex than the sample queries presented in the previous chapter.

**“Simple” Tables:** Thus far, most sample queries and exercises have referenced the PRESERVE and EMPLOYEE tables. Because these tables are easy to understand, we have been able to focus our attention on SELECT statements without having to spend much effort trying to understand the data in these tables. Also, one simplifying feature is that each table has a unique column. For example, PNO is unique within the PRESERVE table, and ENO is unique within the EMPLOYEE table.

**A More “Complex” Table - The PURCHASE Table:** In this chapter, all sample queries will reference the PURCHASE table. This table is more complex because no single column will necessarily contain unique values. The following pages describe this table.

## PURCHASE: A More Complex Table

Simply “eyeball” the following PURCHASE table. (Its rows are sorted for easier reading.) Observe that no single column contains unique values. The semantics of this table are described on the following page.

PNO	SNO	ENO	PJNO	PURDAY	COST	DISCOUNT
P1	S1	E1	PJ1	1	200	0
P1	S1	E1	PJ1	17	250	50
P1	S1	E1	PJ1	80	300	0
P1	S1	E1	PJ3	5	200	0
P1	S1	E1	PJ3	6	200	0
P1	S2	E2	PJ3	85	300	100
P2	S1	E1	PJ2	1	1000	0
P2	S1	E2	PJ2	2	1000	0
P2	S1	E2	PJ2	85	1200	0
P3	S1	E1	PJ1	10	500	100
P3	S1	E1	PJ1	11	500	100
P3	S1	E1	PJ1	70	550	200
P3	S3	E3	PJ1	7	1200	1100
P3	S3	E3	PJ2	14	1200	1000
P3	S4	E4	PJ1	10	1100	0
P3	S4	E4	PJ2	21	1100	50
P4	S1	E1	PJ3	11	300	75
P4	S1	E1	PJ3	22	300	75
P4	S1	E2	PJ2	11	300	0
P4	S2	E2	PJ2	10	300	0
P4	S2	E2	PJ2	70	400	50
P4	S2	E2	PJ3	33	300	0
P4	S3	E3	PJ1	8	1200	9000
P4	S3	E3	PJ2	15	1200	1000
P4	S4	E4	PJ1	15	1200	0
P4	S4	E4	PJ1	70	1200	0
P5	S1	E1	PJ1	11	1200	0
P5	S1	E1	PJ1	22	1200	0
P5	S1	E1	PJ1	33	1500	25
P5	S3	E3	PJ1	9	1500	1400
P5	S3	E3	PJ2	16	1500	1300
P5	S4	E4	PJ2	20	1500	50
P6	S2	E2	PJ2	11	200	0
P6	S2	E2	PJ2	33	100	0
P6	S4	E4	PJ2	71	1500	0
P7	S3	E1	PJ1	11	1000	0
P7	S3	E1	PJ1	33	1000	0
P7	S4	E1	PJ1	22	1000	0
P7	S4	E2	PJ2	61	1200	0
P7	S4	E2	PJ2	72	1200	200
P8	S1	E1	PJ1	61	100	0
P8	S1	E1	PJ1	72	100	0
P8	S1	E1	PJ1	73	100	0
P8	S1	E1	PJ2	43	100	100
P8	S1	E2	PJ2	41	100	50
P8	S2	E1	PJ2	42	100	75

Figure 9.2: PURCHASE Table

## Semantics of the PURCHASE Table

Each row in the PURCHASE table contains information about the purchase of a part from a supplier by an employee for use in a project. We note that:

- Each part is identified by a part number (PNO)
- Each supplier is identified by a supplier number (SNO)
- Each employee is identified by an employee number (ENO)
- Each project is identified by a project number (PJNO)

These four columns contain character-string data.

The other three columns contain integer values. These are:

- PURDAY: Date of purchase. Here, 1 represents the first day of the current year, 2 represents the second day of the current year, etc.
- COST: Dollar cost of the part.
- DISCOUNT: Discount amount which can range from zero to the full COST of the part.

**Important:** Any part may be purchased from any supplier by any employee for use in any project. Also, on different days, the same part can be purchased from the same supplier by the same employee for use in the same project. (For example, examine the first three rows in Figure 9.2.) *However, this kind of purchase cannot happen multiple times during the same day.* Therefore, any combination of (PNO, SNO, ENO, PJNO, PURDAY) values must be unique.

Verify the above observations by examining the PNO, SNO, ENO, PJNO, and PURDAY columns in the PURCHASE table. Note that:

- Every column contains some duplicate values.
- Every *pair* of columns contains some duplicate values. For example, the first five rows which shows the same (PNO, SNO) values.
- Every *triplet* of columns contains some duplicate values. For example, the first five rows show the same (PNO, SNO, ENO) values.
- Every *quadruple* of columns contains some duplicate values. For example, the first three rows show the same (PNO, SNO, ENO, PJNO) values.
- *However, every quintuple of (PNO, SNO, ENO, PJNO, PURDAY) values is unique.*

## Review: Grouping by One Column

The following sample query references the PURCHASE table and groups rows by a single column.

**Sample Query 9.10:** Reference the PURCHASE table. For each project, display its project number (PJNO) followed by the total cost of all parts purchased for the project. Sort the result by PJNO values.

```
SELECT PJNO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY PJNO
ORDER BY PJNO
```

PJNO	TOTCOST
PJ1	16900
PJ2	15200
PJ3	1600

**Syntax & Logic:** Nothing new.

**ORDER BY Clause:** Sorting a result table by the grouping-column (PJNO), as illustrated in this example, makes it easier to understand. Therefore, all of this chapter's examples will specify an ORDER BY clause. (Here, if you did not specify an ORDER BY clause, there is a good chance that you might observe an incidentally sorted result.)

**Sample Query 9.11:** Reference the PURCHASE table. For each supplier, display its supplier number (SNO) followed by the total cost of all parts purchased from the supplier.

```
SELECT SNO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY SNO
ORDER BY SNO
```

PJNO	TOTCOST
S1	11200
S2	1700
S3	9800
S4	11000

**Syntax & Logic:** Nothing new.

## Exercises

The following exercises group by one column of the PURCHASE table.

- 9N. Reference the PURCHASE table. For each part, display its part number (PNO) followed by the total cost of all its parts. The result should look like:

<u>PNO</u>	<u>TOTCOST</u>
P1	1450
P2	3200
P3	6150
P4	6700
P5	8400
P6	1800
P7	5400
P8	600

- 9O. Reference the PURCHASE table. For each employee who purchased a part, display his employee number (ENO) followed by the total cost of all parts purchased by the employee. The result should look like:

<u>ENO</u>	<u>TOTCOST</u>
E1	11700
E2	6600
E3	7800
E4	7600

## Grouping by Two Columns

The following sample query specifies a GROUP BY clause that references two columns.

**Sample Query 9.12:** Reference the PURCHASE table. For each project that used parts supplied by some supplier, display the project's PJNO value, followed by the supplier's SNO value, followed by the total cost of the project's parts purchased from the supplier. Sort the result by SNO within PJNO.

```
SELECT PJNO, SNO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY PJNO, SNO
ORDER BY PJNO, SNO
```

PJNO	SNO	TOTCOST
PJ1	S1	6500
PJ1	S3	5900
PJ1	S4	4500
PJ2	S1	3700
PJ2	S2	1100
PJ2	S3	3900
PJ2	S4	6500
PJ3	S1	1000
PJ3	S2	600

**Syntax:** GROUP BY (Column1, Column2)

**Logic:** Group by pairs of (PJNO, SNO) values, and then summarize the COST values for each group.

**Important Observation-1:** You must know that duplicate pairs of (PJNO, SNO) values can possibly appear in some rows. Otherwise, there is no need to summarize over unique pairs of values.

**Important Observation-2:** Make a simple modification to the above SELECT statement. "Swap" the two columns specified in the GROUP BY clause such that the revised statement now looks like:

```
SELECT PJNO, SNO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY SNO, PJNO ←
ORDER BY PJNO, SNO
```

Execute this revised statement. Observe the *same result*. We will elaborate on this observation after Sample Query 9.17.

## Different Query Objective: Same Result Table (Almost)

The query objective of the previous Sample Query 9.12 focused on projects. Note that PJNO was specified as the first (leftmost) column in the result table. Consider the following query objective that focuses on suppliers where SNO is specified as the first column in the result table.

**Sample Query 9.13:** For each supplier who sold at least one part for use in some project, display the supplier's SNO value, followed by the project's PJNO value, followed by the total cost of the parts that the supplied sold for the project. Sort the result by PJNO within SNO.

```
SELECT SNO, PJNO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY SNO, PJNO
ORDER BY SNO, PJNO
```

SNO	PJNO	TOTCOST
S1	PJ1	6500
S1	PJ2	3700
S1	PJ3	1000
S2	PJ2	1100
S2	PJ3	600
S3	PJ1	5900
S3	PJ2	3900
S4	PJ1	4500
S4	PJ2	6500

**Syntax and Logic:** Nothing new.

**Important Observation:** *This result table displays the same data as the previous result table. The only differences pertain to the left-to-right column sequence and the top-to-bottom row sequence.* From a practical viewpoint, most business users will consider the revised left-to-right/top-to-bottom sequences to be significant and contend that this result table differs from the previous result table. (A few theoretically minded users might disagree. From a theory viewpoint, Sample Queries 9.11 and 9.12 return the same result set. Review Appendix 2B.)

**Again, an Important Observation:** Reorder the GROUP BY columns such that it looks like:

```
GROUP BY PJNO, SNO
```

Execute the revised statement. Observe the *same result*. We will elaborate on this observation after Sample Query 9.17.



**Sample Query 9.14:** Reference the PURCHASE table. For each employee who purchased parts from a supplier, display the employee's ENO value, followed by the supplier's SNO value, followed by the total cost of parts the employee purchased from the supplier.

```
SELECT ENO, SNO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY ENO, SNO
ORDER BY ENO, SNO
```

ENO	SNO	TOTCOST
E1	S1	8600
E1	S2	100
E1	S3	2000
E1	S4	1000
E2	S1	2600
E2	S2	1600
E2	S4	2400
E3	S3	7800
E4	S4	7600

**Syntax & Logic:** Nothing new.

**Observation:** This result shows that Employee E4 only purchased parts from one supplier (S4). There may be a good reason for this. (Or, maybe Employee E4 plays a lot of golf with Supplier S4.) A similar observation applies to Employee E3.

**Again, an Important Observation:** Reorder the GROUP BY columns such that it looks like:

```
GROUP BY SNO, ENO
```

Execute the revised statement. Observe the *same result*. We will elaborate on this observation after Sample Query 9.17.

## Different Query Objective: Same Result Table (Almost)

The previous Sample Query 9.14 focused on employees where ENO was specified first in the SELECT-clause. Consider the following query objective that focuses on suppliers where SNO is specified first in the SELECT-clause.

**Sample Query 9.15:** Reference the PURCHASE table. For each supplier who sold parts to an employee, display the supplier's SNO value, followed by the employee's ENO value, followed by the total cost of parts the supplier sold to the employee.

```
SELECT SNO, ENO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY SNO, ENO
ORDER BY SNO, ENO
```

SNO	ENO	TOTCOST
S1	E1	8600
S1	E2	2600
S2	E1	100
S2	E2	1600
S3	E1	2000
S3	E3	7800
S4	E1	1000
S4	E2	2400
S4	E4	7600

**Syntax & Logic:** Nothing new.

**Observation:** *This result table displays the same data as the previous result table for Sample Query 9.14. The only differences pertain to the left-to-right column sequence and the top-to-bottom row sequence.*

**Again - Reorder GROUP BY Columns:** Reorder the GROUP BY columns such that it looks like:

```
GROUP BY ENO, SNO
```

Execute the revised statement. Observe the *same result*. We will elaborate on this observation after Sample Query 9.17.

**Front-End Tool Reorders Columns/Rows:** Many front-end tools allow you to "reorganize" the preceding result table (for Sample Query 9.14) into the above result table. Such tools provide a simple way to (i) alter the left-to-right column sequence, and (ii) sort a result table into a different row sequence. This can be very useful for ad hoc modification of a report's format.

## Grouping by Three Columns

The following sample query specifies a GROUP BY clause that references three columns.

**Sample Query 9.16:** Access the PURCHASE table. For each project where an employee purchased a part from a supplier for use in the project, display the corresponding PJNO, ENO, and SNO values followed by the total cost of parts purchased by the employee from the supplier for the project. Sort the result by SNO within ENO within PJNO.

```
SELECT PJNO, ENO, SNO, SUM (COST) TOTCOST
FROM   PURCHASE
GROUP BY PJNO, ENO, SNO
ORDER BY PJNO, ENO, SNO
```

PJNO	ENO	SNO	TOTCOST
PJ1	E1	S1	6500
PJ1	E1	S3	2000
PJ1	E1	S4	1000
PJ1	E3	S3	3900
PJ1	E4	S4	3500
PJ2	E1	S1	1100
PJ2	E1	S2	100
PJ2	E2	S1	2600
PJ2	E2	S2	1000
PJ2	E2	S4	2400
PJ2	E3	S3	3900
PJ2	E4	S4	4100
PJ3	E1	S1	1000
PJ3	E2	S2	600

**Again - Reorder GROUP BY Columns:** The following GROUP BY clauses are equivalent to each other.

```
GROUP BY PJNO, ENO, SNO
GROUP BY PJNO, SNO, ENO
GROUP BY ENO, PJNO, SNO
GROUP BY ENO, SNO, PJNO
GROUP BY SNO, ENO, PJNO
GROUP BY SNO, PJNO, SNO
```

Again, we explain after Sample Query 9.17.

## Different Query Objective: Same Result Table (Almost)

The query objective of the previous Sample Query 9.16 focused on projects. Notice that PJNO was specified as the first column in the result table. Consider the following query objective that focuses on suppliers where SNO is specified as the first column in the result table.

**Sample Query 9.17:** For each supplier who sold a part to an employee for use in a project, display the corresponding SNO, PJNO, and ENO values followed by the total cost of parts purchased from the supplier by the employee for use in the project. Sort the result by ENO within PJNO within SNO.

```
SELECT SNO, PJNO, ENO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY SNO, PJNO, ENO
ORDER BY SNO, PJNO, ENO
```

SNO	PJNO	ENO	TOTCOST
S1	PJ1	E1	6500
S1	PJ2	E1	1100
S1	PJ2	E2	2600
S1	PJ3	E1	1000
S2	PJ2	E1	100
S2	PJ2	E2	1000
S2	PJ3	E2	600
S3	PJ1	E1	2000
S3	PJ1	E3	3900
S3	PJ2	E3	3900
S4	PJ1	E1	1000
S4	PJ1	E4	3500
S4	PJ2	E2	2400
S4	PJ2	E4	4100

**Syntax & Logic:** Nothing new.

**Observation:** *This result table displays the same data as the previous result for Sample Query 9.16. The only differences pertain to the left-to-right column sequence and the top-to-bottom row sequence.*

**Again, Reorder GROUP BY Columns:** The following GROUP BY clauses are equivalent to each other.

```
GROUP BY SNO, PJNO, ENO
GROUP BY SNO, ENO, PJNO
GROUP BY PJNO, ENO, SNO
GROUP BY PJNO, SNO, ENO
GROUP BY ENO, PJNO, SNO
GROUP BY ENO, SNO, PJNO
```

## GROUP BY a “Combination” of Columns

The “Reorder GROUP BY Columns” commentary in the previous sample queries should have (hopefully) encouraged you to conclude that you can specify the GROUP BY columns in any sequence. This observation leads into a little mathematics.

Given a set of values: {10, 30, 20}

Each of the following triplets represent the same “*combination*” of values.

```
(10, 20, 30)
(10, 30, 20)
(20, 10, 30)
(20, 30, 10)
(30, 10, 20)
(30, 20, 10)
```

**Key point:** The GROUP BY clause specifies a *combination* of columns.

Example: Given three columns, COL1, COL2, and COL3, the following GROUP BY clauses are equivalent.

```
GROUP BY COL1, COL2, COL3
GROUP BY COL1, COL3, COL2
GROUP BY COL2, COL1, COL3
GROUP BY COL2, COL3, COL1
GROUP BY COL3, COL1, COL2
GROUP BY COL3, COL2, COL1
```

**Articulation of Query Objectives:** Assuming the user understands the meaning of “combination,” you could re-articulate the query objective for Sample Query 9.16 as:

```
For each combination of (PJNO, ENO, SNO) values, display the
PJNO, ENO, SNO values in left-to-right column sequence,
followed by the total cost for each combination of these
values. Sort the result . . .
```

Again, writing a precise, concise, and unambiguous query objective in some human language may not be easy.

## Exercises

The following exercises group by two columns.

- 9P. Access the PURCHASE table. Calculate the total of COST for each combination of (PJNO, ENO) of values. Display these columns in the (PJNO, ENO) left-to-right column sequence followed by the total cost. Sort the result in ascending sequence by (PJNO, ENO). The result should look like:

<u>PJNO</u>	<u>ENO</u>	<u>TOTCOST</u>
PJ1	E1	9500
PJ1	E3	3900
PJ1	E4	3500
PJ2	E1	1200
PJ2	E2	6000
PJ2	E3	3900
PJ2	E4	4100
PJ3	E1	1000
PJ3	E2	600

- 9Q. Access the PURCHASE table. Calculate the total of COST for each combination of (PNO, SNO) of values. Display these columns in the (PNO, SNO) left-to-right column sequence followed by the total cost. Sort the result in ascending sequence by (PNO, SNO). The result should look like:

<u>PNO</u>	<u>SNO</u>	<u>TOTCOST</u>
P1	S1	1150
P1	S2	300
P2	S1	3200
P3	S1	1550
P3	S3	2400
P3	S4	2200
P4	S1	900
P4	S2	1000
P4	S3	2400
P4	S4	2400
P5	S1	3900
P5	S3	3000
P5	S4	1500
P6	S2	300
P6	S4	1500
P7	S3	2000
P7	S4	3400
P8	S1	500
P8	S2	100

The following exercises group by three columns.

- 9R. Access the PURCHASE table. Calculate the total of COST for each combination of (PJNO, ENO, SNO) values. Display these columns in the (PJNO, ENO, SNO) left-to-right column sequence followed by the total cost. Sort the result in ascending sequence by (PJNO, ENO, SNO). The result should look like:

<u>PJNO</u>	<u>ENO</u>	<u>SNO</u>	<u>TOTCOST</u>
PJ1	E1	S1	6500
PJ1	E1	S3	2000
PJ1	E1	S4	1000
PJ1	E3	S3	3900
PJ1	E4	S4	3500
PJ2	E1	S1	1100
PJ2	E1	S2	100
PJ2	E2	S1	2600
PJ2	E2	S2	1000
PJ2	E2	S4	2400
PJ2	E3	S3	3900
PJ2	E4	S4	4100
PJ3	E1	S1	1000
PJ3	E2	S2	600

- 9S. Access the PURCHASE table. Calculate the total of COST for each combination of (PNO, SNO, ENO) values. Display these columns in the (PNO, SNO, ENO) left-to-right column sequence followed by the total cost. Sort the result in ascending sequence by (PNO, SNO, ENO). The result should look like:

<u>PNO</u>	<u>SNO</u>	<u>ENO</u>	<u>TOTCOST</u>
P1	S1	E1	1150
P1	S2	E2	300
P2	S1	E1	1000
P2	S1	E2	2200
P3	S1	E1	1550
P3	S3	E3	2400
P3	S4	E4	2200
P4	S1	E1	600
P4	S1	E2	300
P4	S2	E2	1000
P4	S3	E3	2400
P4	S4	E4	2400
P5	S1	E1	3900
P5	S3	E3	3000
P5	S4	E4	1500
P6	S2	E2	300
P6	S4	E4	1500
P7	S3	E1	2000
P7	S4	E1	1000
P7	S4	E2	2400
P8	S1	E1	400
P8	S1	E2	100
P8	S2	E1	100

## Grouping by Four Columns

The following sample query specifies a GROUP BY clause that references four columns.

**Sample Query 9.18:** Access the PURCHASE table. Extend Sample Query 9.16 with another level of detail by including PNO values. For each project where an employee purchased a part from a supplier for use in the project, display the corresponding PJNO, ENO, SNO and PNO values followed by the total cost of *each part* purchased by the employee from the supplier for the project. Sort the result by PNO within SNO within ENO within PJNO.

*Alternatively, for each combination of (PJNO, ENO, SNO, PNO) values, display the PJNO, ENO, SNO, PNO columns in left-to-right column sequence, followed by the total cost for each combination of values. Sort by . . .*

```
SELECT PJNO, ENO, SNO, PNO, SUM (COST) TOTCOST
FROM   PURCHASE
GROUP BY PJNO, ENO, SNO, PNO
ORDER BY PJNO, ENO, SNO, PNO
```

PJNO	ENO	SNO	PNO	TOTCOST
PJ1	E1	S1	P1	750
PJ1	E1	S1	P3	1550
PJ1	E1	S1	P5	3900
PJ1	E1	S1	P8	300
PJ1	E1	S3	P7	2000
PJ1	E1	S4	P7	1000
PJ1	E3	S3	P3	1200
PJ1	E3	S3	P4	1200
PJ1	E3	S3	P5	1500
PJ1	E4	S4	P3	1100
PJ1	E4	S4	P4	2400

.. total of 30 rows,

.. includes rows for Projects PJ2, PJ3, and PJ4

**Syntax & Logic:** Nothing new. The GROUP BY clause specified four columns.

**Reorder GROUP BY Columns:** Given 4 columns, there are a total of  $4! = 24$  equivalent GROUP BY clauses.



## Important Syntax Rule

We could have introduced the following syntax rule earlier in this chapter. However, our sample queries were simple enough to allow us to delay presenting this slightly complex rule.

**Author Comment:** When I help a student debug an erroneous SELECT statement that includes a GROUP BY clause, the first thing I do is determine if the statement obeys the following syntax rule. Frequently, failure to obey this rule is the source of the problem. This is especially true if the system returns some kind of "GROUP BY" error message.

**GROUP BY Syntax Rule:** Whenever a SELECT-clause specifies one or more aggregate functions, if this SELECT-clause also specifies a column without an aggregate function, this column must be specified within a GROUP BY clause.

**The following examples obey this syntax rule:**

**Example-1:** Sample Query 9.1 specified the following SELECT and GROUP BY clauses which obey this rule.

```
SELECT STATE, SUM (ACRES)
FROM PRESERVE
GROUP BY STATE
ORDER BY STATE
```

The second column in this SELECT-clause specifies an aggregate function (SUM). However, the first column, STATE, does not specify an aggregate function. Therefore, STATE must be specified in the GROUP BY clause (as shown).

---

**Example-2:** The following SELECT and GROUP BY clauses obey this rule.

```
SELECT SNO, PNO, AVG (COST)
FROM PURCHASE
GROUP BY SNO, PNO
ORDER BY SNO, PNO
```

The third column in this SELECT-clause specifies an aggregate function (AVG). The first (SNO) and second (PNO) columns do not specify an aggregate function. Therefore, both the SNO and PNO must be specified in the GROUP BY clause (as shown).

**Examples that violate the GROUP BY syntax rule:**

**Example-3:**       SELECT STATE, SUM (ACRES)  
                  FROM PRESERVE         → Error

The second column in the SELECT-clause specifies an aggregate function, and the STATE column does not specify aggregate function. Hence, STATE must be specified in a GROUP BY clause. But there is no GROUP BY clause.

---

**Example-4:**       SELECT STATE, FEE, SUM (ACRES)  
                  FROM PRESERVE  
                  GROUP BY STATE         → Error

The third column in this SELECT-clause specifies an aggregate function, and the STATE and FEE columns do not specify aggregate functions. Hence, both the STATE and FEE columns must be specified in the GROUP BY clause. However, this GROUP BY clause fails to specify the FEE column.

---

**Example-5:**       SELECT PNO, STATE, AVG (FEE), SUM (ACRES), FEE  
                  FROM PRESERVE  
                  GROUP BY STATE, FEE   → Error

The third and fourth columns in this SELECT-clause specify aggregate functions, and none of the other three columns specify an aggregate function. Hence, these other columns (PNO, STATE and FEE) must be specified in the GROUP BY clause. However, this GROUP BY clause fails to specify the PNO column.

---

**Example-6:**       SELECT PJNO, SNO, PNO, MAX (COST)  
                  FROM PURCHASE  
                  GROUP BY PONO, PNO    → Error

The fourth column in this SELECT-clause specifies an aggregate function, and the other columns (PONO, SNO, and PNO) do not specify an aggregate function. Hence, PONO, SNO, and PNO must be specified in the GROUP BY clause. However, this GROUP BY clause fails to specify the SNO column.

## Another Common Error: Grouping by Unique Combination of Columns

**Review:** The previous chapter noted that the following statement produces a useless result table because the GROUP BY clause referenced a unique column.

```
SELECT PNO, SUM (ACRES)
FROM PRESERVE
GROUP BY PNO ←
ORDER BY PNO
```

This kind of error extends to any unique combination of columns. Consider the following example which groups by five columns. Grouping by five, or any number of columns, is fine, if we know that the combination of columns is always non-unique. However, this does not apply to the following grouping of the (SNO, PNO, ENO, PJNO, PURDAY) columns.

```
SELECT SNO, PNO, ENO, PJNO, PURDAY, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY SNO, PNO, ENO, PJNO, PURDAY
ORDER BY SNO, PNO, ENO, PJNO, PURDAY
```

SNO	PNO	ENO	PJNO	PURDAY	TOTCOST
S1	P1	E1	PJ1	1	200
S1	P1	E1	PJ1	17	250
S1	P1	E1	PJ1	80	300
S1	P1	E1	PJ3	5	200
S1	P1	E1	PJ3	6	200
S1	P2	E1	PJ2	1	1000
S1	P2	E2	PJ2	2	1000
S1	P2	E2	PJ2	85	1200
S1	P3	E1	PJ1	10	500
S1	P3	E1	PJ1	11	500

. . . . . total of 46 rows . . .

**Logic:** Recall the business rule that on different days, the same part can be purchased from the same supplier by the same employee for use in the same project. However, this kind of purchase cannot happen multiple times during the same day. Hence the combination of (SNO, PNO, ENO, PJNO, PURDAY) values is unique.

Because the syntax is valid, the statement executes and produces the above result table. However, this result is not useful because it does not present any new information. The above result has 46 rows, the same number of rows in the PURCHASE table. The above GROUP BY clause forms 46 groups where each group contains just one row, and the SUM function only summarizes over the one COST value in each group. Such grouping is not reasonable.

## Basic Grouping Pattern

Previous query objectives and corresponding SELECT statements fit a basic pattern. Consider the SELECT statement for Sample Query 9.17.

```
SELECT SNO, PJNO, ENO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY SNO, PJNO, ENO
ORDER BY SNO, PJNO, ENO
```

SELECT-Clause: The combination of grouping-columns (SNO, PJNO, ENO) is not unique. Also, the grouping-columns appear to the left of the aggregate functions. This is common, but not required.

GROUP BY Clause: This clause specifies the grouping-columns and must be in sync with the SELECT-clause (as described in previous Important Syntax Rule). Also, grouping-columns are usually specified in the same left-to-right column sequence as grouping columns in SELECT-clause. This is common, but not required.

ORDER BY Clause: This clause *frequently* specifies the grouping-columns specified in the same left-to-right column sequence as the grouping-columns in the SELECT and GROUP BY clauses. This is common, but not required.

**Generic Skeleton-Code:** The above observations imply that the following skeleton-code represents many SELECT statements that group by multiple columns.

```
SELECT COLX, COLA, COLB, agg-fn1, agg-fn2, . . .
FROM _____
WHERE _____
GROUP BY COLX, COLA, COLB
HAVING _____
ORDER BY COLX, COLA, COLB
```

The following sample query requires additional code that does not change the basic query pattern described on the previous page. This SELECT statement includes a WHERE-clause.

**Sample Query 9.19:** Access the PURCHASE table. Only consider purchases by Employee E1. For each supplier that Employee E1 purchased a part from, display the corresponding ENO and SNO values, followed by the total cost, maximum cost, and minimum cost of Employee E1's purchases from each supplier.

```
SELECT ENO, SNO, SUM (COST) TOTCOST,  
          MIN (COST) MINCOST,  
          MAX (COST) MAXCOST  
FROM PURCHASE  
WHERE ENO = 'E1'  
GROUP BY ENO, SNO  
ORDER BY ENO, SNO
```

ENO	SNO	TOTCOST	MINCOST	MAXCOST
E1	S1	8600	100	1500
E1	S2	100	100	100
E1	S3	2000	1000	1000
E1	S4	1000	1000	1000

**Syntax & Logic:** Nothing new.

The following query objective is a little more complex. It requires specifying a WHERE-clause and a HAVING-clause. It also sorts the result table by a calculated total.

**Sample Query 9.20:** Does the PURCHASE table show that some supplier has sold the same part at different prices? I.e., does every purchase which references the same combination of (SNO, PNO) values have the same COST value? For any supplier who has charged different prices for the same part, display the following information.

- The corresponding SNO and PNO values
- The minimum COST (MINCOST) for each pair of (SNO, PNO) values
- The maximum COST (MAXCOST) for each pair of (SNO, PNO) values
- The difference (COSTDIFF) between these maximum and minimum costs
- Only display rows where the maximum and minimum costs are different.
- Display this information in the following column sequence: SNO, PNO, MAXCOST, MINCOST, COSTDIFF
- Sort the result in descending sequence by COSTDIFF. Within duplicate COSTDIFF values, sort by PNO (ascending) within SNO (ascending).

```
SELECT SNO, PNO,
       MIN (COST) MINCOST,
       MAX (COST) MAXCOST,
       MAX (COST) - MIN (COST) COSTDIFF
FROM   PURCHASE
GROUP BY SNO, PNO
HAVING MAX (COST) > MIN (COST)
ORDER BY COSTDIFF DESC, SNO, PNO
```

SNO	PNO	MINCOST	MAXCOST	COSTDIFF
S1	P5	1200	1500	300
S1	P2	1000	1200	200
S4	P7	1000	1200	200
S1	P1	200	300	100
S2	P4	300	400	100
S2	P6	100	200	100
S1	P3	500	550	50

**Syntax & Logic:** Nothing new. But, "putting the pieces together" requires a little more thought.

## Exercises

9T. Access the PURCHASE table. Exclude from consideration all rows associated with Project PJ2. Display the total of COST for each combination of (PJNO, ENO, SNO) of values (excluding Project PJ2). The result should look like:

PJNO	ENO	SNO	TOTCOST
PJ1	E1	S1	6500
PJ1	E1	S3	2000
PJ1	E1	S4	1000
PJ1	E3	S3	3900
PJ1	E4	S4	3500
PJ3	E1	S1	1000
PJ3	E2	S2	600

9U. Access the PURCHASE table. Display the total COST for each combination of (PNO, SNO, ENO) values if that total is greater than or equal to 2000. The result should look like:

PNO	SNO	ENO	TOTCOST
P2	S1	E2	2200
P3	S3	E3	2400
P3	S4	E4	2200
P4	S3	E3	2400
P4	S4	E4	2400
P5	S1	E1	3900
P5	S3	E3	3000
P7	S3	E1	2000
P7	S4	E2	2400

## Preview: ROLLUP (Column-1)

This chapter concludes with a brief preview of the ROLLUP and CUBE options for the GROUP BY clause.

[There is much more that can be said about ROLLUP and CUBE. If all goes well, these topics will be included in a future edition of this book.]

**Sample Query 9.21:** For each project, display its PJNO value followed by the total cost of all parts purchased for the project. Also, display a grand total cost for all projects.

```
SELECT PJNO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY ROLLUP (PJNO)
ORDER BY PJNO
```

PJNO	TOTCOST	
PJ1	16900	
PJ2	15200	
PJ3	1600	
-	33700	←

**Syntax:** GROUP BY ROLLUP (grouping-expression-list)

In this example, the grouping-expression-list contains a single grouping-expression consisting of a single column (PJNO). The following sample query will specify multiple columns within the grouping-expression-list.

**Logic:** ROLLUP tells the system to generate a COST (sub)total for each group, plus the overall grand total of all COST values.



## Preview: ROLLUP (Column-1, Column-2)

**Sample Query 9.22:** Access the PURCHASE table to display three summary totals:

- Subtotal cost for each combination of (PJNO, SNO) values
- Subtotal cost for each PJNO value
- Grand total of all costs

```
SELECT PJNO, SNO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY ROLLUP (PJNO, SNO)
ORDER BY PJNO, SNO
```

PJNO	SNO	TOTCOST
PJ1	S1	6500
PJ1	S3	5900
PJ1	S4	4500
PJ1	-	16900
PJ2	S1	3700
PJ2	S2	1100
PJ2	S3	3900
PJ2	S4	6500
PJ2	-	15200
PJ3	S1	1000
PJ3	S2	600
PJ3	-	1600
-	-	33700

**Syntax:** GROUP BY ROLLUP (column-1, column-2,...)

**Logic:** ROLLUP (PJNO, SNO) asks the system to generate a subtotal for each combination of (PJNO, SNO) values, a subtotal for each PJNO value, and a grand total.

**Important:** While the *left-to-right column sequence* is not significant within the basic GROUP BY clause, it becomes significant when you specify ROLLUP.

ROLLUP (C1, C2) and ROLLUP (C2, C1) produce different results.

Optional Exercise: Change the above ROLLUP clause to:

```
ROLLUP (PJNO, SNO)
```

Observe a different result.

## Preview: CUBE (Column-1, Column-2)

The next sample query introduces the CUBE keyword which generates summary totals for all possible subsets of the specified columns.

**Sample Query 9.23:** Access the PURCHASE table to display four summary totals:

- Subtotal cost for each combination of (PJNO, SNO) values
- Subtotal cost for each PJNO value
- Subtotal cost for each SNO value
- Grand total of all costs

```
SELECT PJNO, SNO, SUM (COST) TOTCOST
FROM PURCHASE
GROUP BY CUBE (PJNO, SNO)
ORDER BY PJNO, SNO
```

PJNO	SNO	TOTCOST
PJ1	S1	6500
PJ1	S3	5900
PJ1	S4	4500
PJ1	-	16900
PJ2	S1	3700
PJ2	S2	1100
PJ2	S3	3900
PJ2	S4	6500
PJ2	-	15200
PJ3	S1	1000
PJ3	S2	600
PJ3	-	1600
-	S1	11200
-	S2	1700
-	S3	9800
-	S4	11000
-	-	33700

**Syntax:** CUBE (grouping-expression-list)

**Logic-Important:** *While the left-to-right column sequence is significant with ROLLUP, it is not significant in CUBE.*

CUBE (PJNO, SNO) and CUBE (SNO, PJNO) produce same result. You can observe this fact by changing the above CUBE-clause to CUBE (SNO, PJNO) and executing the statement.

## Summary

Our generic SELECT statement is expanded to include extensions to the GROUP BY clause.

```
SELECT [DISTINCT] column(s)
FROM   table
[WHERE condition (s)]
[GROUP BY [ROLLUP | CUBE ] group-col1, group-col2, . . .]
[HAVING condition (s)]
[ORDER BY column(s)]
```

## Summary Exercise

- 9V. Reconsider Exercise 9S [Access the PURCHASE table. Calculate the total of COST for each combination of (PNO, SNO, ENO) values. Display these columns in the (PNO, SNO, ENO) left-to-right column sequence followed by the total cost. Sort the result in ascending sequence by (PNO, SNO, ENO)].

This final result table contained 23 rows. Some of these rows corresponded to groups that summarized over just one or two individual PURCHASE rows. To reduce the number rows in the final result, exclude any summary row from the final result if that summary represents a total of just one or two rows. The result should look like:

PNO	SNO	ENO	TOTCOST	GPCT
P1	S1	E1	1150	5
P3	S1	E1	1550	3
P4	S2	E2	1000	3
P5	S1	E1	3900	3
P8	S1	E1	400	4

The GPCT column contains the number of rows in the group.

## Individual Functions

This chapter presents a brief introduction to the individual functions. (Instead of the term "individual," DB2 and SQL Server use the term *Scalar Function*, and ORACLE uses the term *Single-Row Function*.) We begin by describing the fundamental distinction between an aggregate function and an individual function.

**Aggregate Functions:** Recall that an aggregate function operates on a group of values *from multiple rows* and returns a single value. For example, the following statement selects the four FEE values from the four Arizona rows. These values are 3.0, 0.0, 3.0, and 3.0. The aggregate function SUM operates on these values and returns a single value of 9.00.

```
SELECT SUM (FEE) FROM PRESERVE WHERE STATE = 'AZ'
```

<u>SUM (FEE)</u>
9.00

**Individual Functions:** An individual function operates on a value *from a single row* and returns a single value. We illustrate this behavior by previewing the SQRT function to display the square root of each ACRES values in the Arizona preserves.

```
SELECT SQRT (ACRES) FROM PRESERVE WHERE STATE = 'AZ'
```

<u>SQRT (ACRES)</u>
25.69
221.63
19.49
34.64

The SQRT function examines the ACRES value in each selected row and returns the square root of each value. Unlike an aggregate function, the SQRT function does not "summarize" values from multiple rows into a single value.

The primary difference between an aggregate function and an individual function is straightforward. This chapter will also illustrate two other differences between individual functions and aggregate functions. These are:

1. Unlike an aggregate function, an individual function can be referenced in a WHERE-clause. (See Sample Query 10.6.)
2. Unlike an aggregate function, an individual function can always be nested within another individual function. (See Sample Query 10.2.)

**Categories of Individual (Scalar, Single-Row) Functions:** Your SQL reference manual will present the individual functions organized within categories. This chapter and the following chapter present examples of individual functions organized within four categories.

Arithmetic Functions: Sample Queries 10.1-10.2

Character-String Functions: Sample Queries 10.3-10.8

Data-Type Conversion Functions: Sample Query 10.9

Date-Time Functions: Chapter 10.5

Most systems also support other more specialized categories of functions (e.g., XML functions, regular-expression functions), plus some miscellaneous functions.

**System Incompatibility:** Unfortunately, there is *considerable variation among the individual functions supported by different database systems*. Therefore, this chapter only presents a few examples that were executed on DB2, ORACLE, and SQL Server. *These examples provide a conceptual framework to help you explore your SQL reference manual for a description of all individual functions provided by your database system.*

**No Exercises:** Given the great variation among individual functions supported by different database systems, this chapter only offers one summary exercise which is optional.

## DEMO2 Table

This chapter's sample queries will reference the DEMO2 table shown below.

I1	D1	V1	F1
-10	-8.82	Julie Martyn	Hello
-5	-5.28	JESSIE MARTYN	GOOD
0	0.00	Janet Martyn	By
2	6.42	Frank	BYE
9	9.98	Wally	HYY

**Figure 10.1a: DEMO2 Table**

The data-type of each column is:

- I1 - integer values [INTEGER]
- D1 - decimal values: [DECIMAL (5,2)]
- V1 - variable-length character-strings [VARCHAR (15)]
- F1 - fixed-length character-strings [CHAR (5)]

In case you are interested, the following Figure 10.1b presents the CREATE TABLE statement used to create DEMO2.

```
CREATE TABLE DEMO2
(I1 INTEGER,
 D1 DECIMAL (5, 2),
 V1 VARCHAR (15),
 F1 CHAR (5));
```

**Figure 10.1b: CREATE TABLE for DEMO2 Table**

This CREATE TABLE statement is presented to focus your attention to the data-type of each column.

## A. Arithmetic Functions

Sample Queries 10.1 and 10.2 demonstrate the following arithmetic functions. N represents the name of a numeric column. M is an integer value.

- ROUND(N,M) - Round N to M decimal places
- TRUNC(N,M) - Truncate N to M decimal places
- ABS(N) - Absolute value of N
- SIGN(N) - +1 if N > 0, 0 if N = 0, -1 if N < 0
- MOD(N,M) - Remainder of division: N/M
- POWER(N,M) - N raised to Mth power
- SQRT(N) - Square root of N

**Sample Query 10.1:** Reference Column D1 (a decimal column) in the DEMO2 table to display:

- Column D1
- D1 rounded to the nearest integer
- D1 rounded to the tenths position
- D1 truncated to the integer position
- D1 truncated to the tenths position

SELECT D1,					<b><u>DB2 &amp; ORACLE</u></b>
ROUND (D1,0)	ROUND0,				
ROUND (D1,1)	ROUND1,				
TRUNC (D1,0)	TRUNC0,				
TRUNC (D1,1)	TRUNC1				
FROM DEMO2					

D1	ROUND0	ROUND1	TRUNC0	TRUNC1
-8.82	-9.00	-8.80	-8.00	-8.80
-5.28	-5.00	-5.30	-5.00	-5.20
0.00	0.00	0.00	0.00	0.00
6.42	6.00	6.40	6.00	6.40
9.98	10.00	10.00	9.00	9.90

**SQL Server:** SQL Server does not understand TRUNC. You must use a variation of the ROUND function to implement truncation. Recode the above statement as:

```
SELECT D1,
       ROUND (D1, 0)           ROUND0,
       ROUND (D1, 1)           ROUND1,
       ROUND (D1, 0, -1)      TRUNC0,    ←
       ROUND (D1, 1, -1)      TRUNC1     ←
FROM DEMO2;
```

**Sample Query 10.2:** Reference Column I1 (an INTEGER column) in the DEMO2 table to display:

- Column I1
- The absolute value of I1
- The sign of I1
- The remainder of dividing I1 by 4
- I1 raised to the third power
- The square root of the absolute value of I1

<pre>SELECT I1,       ABS  (I1)       SIGN (I1)       MOD  (I1, 4)       POWER (I1, 3)       SQRT (ABS (I1)) FROM DEMO2</pre>	<b><u>DB2 &amp; ORACLE</u></b>
---	--------------------------------

I1	ABS I1	SIGN I1	MOD4 I1	CUBE I1	SQRT I1
-10	10	-1	-2	-1000	3.162
-5	5	-1	-1	-125	2.236
0	0	0	0	0	0.000
2	2	1	2	8	1.414
9	9	1	1	729	3.000

These functions reference I1, an integer column. The same functions work if they reference D1, a decimal column.

**SQL Server:** SQL Server does not understand MOD. Replace MOD (I1, 4) with I1 % 4

**Nesting Individual Functions:** Consider the above specification of SQRT (ABS (I1)). Executing SQRT (I1) would fail because some of the I1 values are negative. Therefore, the ABS function is nested inside the SQRT function. Note that both the SQRT and ABS functions are individual functions.

**Nesting An Aggregate and Individual Function:** Recall that some systems do not allow the nesting of two aggregate functions. However, all systems allow you to nest an aggregate function within an individual function, and vice versa.

Nest ABS (individual function) within SUM (aggregate function).

Example: SELECT SUM (ABS (D1)) FROM DEMO2

Nest SUM (aggregate function) within ABS (individual function).

Example: SELECT ABS (SUM (D1)) FROM DEMO2



## **B. Character-String Functions**

Sample Queries 10.3-10.8 demonstrate the following character-string functions that are supported by most systems.

C represents a column containing a fixed-length or variable-length character-string.

LENGTH(C) - return the length of C

UPPER(C) - return C in uppercase characters

LOWER(C) - return C in lowercase characters

LTRIM(C) - return C after trimming any leading blanks

RTRIM(C) - return C after trimming any trailing blanks

SUBSTR(C,I,L) - return a substring of C: start at the I<sup>th</sup> character for length of L positions

CONCAT (C1, C2) - return a concatenation of columns C1 and C2

**Sample Query 10.3:** Reference Column V1, a variable-length character-string, in the DEMO2 table. Return:

- V1
- The length of V1
- V1 in lower-case characters
- V1 in upper-case characters

<pre>SELECT V1,         LENGTH (V1) LENV1,         LOWER (V1) LOWERV1,         UPPER (V1) UPPERV1 FROM DEMO2</pre>	<b><u>DB2 &amp; ORACLE</u></b>
--	--------------------------------

V1	LENV1	LOWERV1	UPPERV1
Julie Martyn	12	julie martyn	JULIE MARTYN
JESSIE MARTYN	13	jessie martyn	JESSIE MARTYN
Janet Martyn	12	janet martyn	JANET MARTYN
Frank	5	frank	FRANK
Wally	5	wally	WALLY

These functions will also work if they reference F1, a fixed-length character-string.

**SQL Server:** Specify LEN instead of LENGTH in this statement. LEN will not include trailing blanks in its count.

**Sample Query 10.4:** Reference Column F1, a fixed-length character-string, in the DEMO2 table. Return:

- F1
- The length of F1
- F1 after trimming trailing blanks
- The length of F1 after trimming trailing blanks

<pre>SELECT F1,        LENGTH (F1)          LEN,        RTRIM (F1)          RTRIM,        LENGTH (RTRIM (F1)) LENRTRIM FROM DEMO2</pre>	<b><u>DB2 &amp; ORACLE</u></b>
---	--------------------------------

F1	LEN	RTRIM	LENRTRIM
Hello	5	Hello	5
GOOD	5	GOOD	4
By	5	By	2
BYE	5	BYE	3
HYY	5	HYY	3

**LENGTH (RTRIM (F1)):** In this result table, columns F1 and RTRIM appear to be identical because the *trailing blanks in F1 are not visible*. To illustrate that trailing blanks are present, we specified LENGTH (RTRIM (F1)) to display the length of F1 values after trailing blanks have been removed. Again, note that we have nested an individual function (RTRIM) within another individual function (LENGTH).

**SQL Server:** Remember to replace LENGTH with LEN.

<pre>SELECT F1,        LEN (F1)          SSLEN,        RTRIM (F1)       RTRIM,        LEN (RTRIM (F1)) LENRTRIM FROM DEMO2</pre>	<b><u>SQL Server</u></b>
--	--------------------------

F1	SSLEN	RTRIM	LENRTRIM
Hello	<b>5</b>	Hello	5
GOOD	<b>4</b>	GOOD	4
By	<b>2</b>	By	2
BYE	<b>3</b>	BYE	3
HYY	<b>3</b>	HYY	3

**Important:** Observe the SSLEN column in the above result table. You do not see all 5s as in the DB2 and ORACLE result table. Recall that SQL Server ignores trailing blanks as previously described back in Chapter 6 (Sample Query 6.3b).

**Sample Query 10.5:** Reference Column V1, a VARCHAR character-string, in the DEMO2 table. Return:

- V1
- The 3<sup>rd</sup> and 4<sup>th</sup> characters in V1
- The 2<sup>nd</sup>, 3<sup>rd</sup>, 4<sup>th</sup>, and 5<sup>th</sup> characters in V1
- The 4<sup>th</sup> character and all following characters in V1

```
SELECT V1,                                DB2 & ORACLE
      SUBSTR (V1, 3, 2)    SUB34,
      SUBSTR (V1, 2, 4)    SUB25,
      SUBSTR (V1, 4)       SUB4END
FROM DEMO2
```

V1	SUB34	SUB25	SUB4END
Julie Martyn	li	ulie	ie Martyn
JESSIE MARTYN	SS	ESSI	SIE MARTYN
Janet Martyn	ne	anet	et Martyn
Frank	an	rank	nk
Wally	ll	ally	ly

The SUBSTR function references V1, a variable-length character-string. SUBSTR also works if it references a fixed-length character-string.

**SQL Server:** Specify SUBSTRING instead of SUBSTR. Also, SUBSTRING must have three arguments. Recode the above statement as:

```
SELECT V1,
      SUBSTRING (V1, 3, 2)          SUB34,
      SUBSTRING (V1, 2, 4)          SUB25,
      SUBSTRING (V1, 4, (LEN (V1)-3)) SUB4END
FROM DEMO2
```

**Suggestion:** Most database systems support a wide variety of individual built-in functions. Take a five-minute detour to observe an overview of these functions. Depending upon your particular database system, search the web for:

ORACLE *single-row* built-in functions, or

DB2 *scalar* built-in functions, or

SQL Server *scalar* built-in functions

Don't bother with details. Simply scan the documentation.

## Individual Function in a WHERE-Clause

Unlike aggregate functions, individual functions can be specified in a WHERE-clause. This feature can be very useful.

**Sample Query 10.6:** Reference the DEMO2 table. Note that the V1 column values are stored in uppercase, lowercase, or mixed case. Display all V1 values that end with "Martyn" coded in any mixture of uppercase and lowercase characters.

```
SELECT V1 DB2, ORACLE & SQL Server  
FROM DEMO2  
WHERE UPPER (V1) LIKE '%MARTYN'
```

```
V1  
Julie Martyn  
JESSIE MARTYN  
Janet Martyn
```

**Sample Query 10.7:** Reference the DEMO2 table. Display F1 values that end with a "Y" (uppercase) or "y" (lowercase). Recall that F1 is a fixed-length character-string and may contain trailing blanks.

```
SELECT F1 DB2, ORACLE & SQL Server  
FROM DEMO2  
WHERE UPPER (RTRIM (F1)) LIKE '%Y'
```

```
F1  
By  
HYY
```

## Concatenation: CONCAT Function and Operator

Character-string concatenation, informally called "string addition," is implemented using the CONCAT function. Most systems also support a concatenation operator (|| or +).

**Sample Query 10.8:** Reference the DEMO2 table. Display two columns in the result table. The first column should display a concatenation of V1 characters followed by the F1 characters. The second column should display a concatenation of F1 characters followed by the V1 characters.

```
SELECT CONCAT (V1,F1), CONCAT (F1,V1) DB2, ORACLE & SQL Server
FROM DEMO2
```

<u>CONCAT (V1,F1)</u>	<u>CONCAT (F1,V1)</u>
Julie MartynHello	HelloJulie Martyn
JESSIE MARTYNGOOD	GOOD JESSIE MARTYN
Janet MartynBy	By Janet Martyn
FrankBYE	BYE Frank
WallyHYY	HYY Wally

**Observation:** Four of the five character-strings in the second column of the result table show embedded blanks corresponding to the training blanks in F1. For example, in the third row, observe the three spaces after "By".

**Alternative Solutions:** Specify a concatenate operator.

**DB2 & ORACLE:** Specify the || symbol.

```
SELECT V1||F1, F1||V1 DB2 & ORACLE
FROM DEMO2
```

**SQL Server:** Specify the + symbol.

```
SELECT V1+F1, F1+V1 SQL Server
FROM DEMO2
```

## C. Data-Type Conversion Function: CAST

All database systems provide functions that convert a data-type value into another data-type value. The CAST function is supported by most systems.

**Sample Query 10.9a:** Reference DEMO2. Display column I1 as an integer and as a decimal. Also, display column D1 as a decimal and as an integer.

```
SELECT I1, CAST (I1 AS DECIMAL) I1ASDEC,      DB2, ORACLE & SQL Server
       D1, CAST (D1 AS INTEGER) D1ASINT
FROM DEMO2
```

I1	I1ASDEC	D1	D1ASINT
-10	-10.0	-8.82	-8
-5	-5.0	-5.28	-5
0	0.0	0.00	0
2	2.0	6.42	6
9	9.0	9.98	9

This statement illustrates data-type conversions using numeric data-types. CAST also supports data-type conversions using non-numeric data-types.

The following sample query applies a LIKE-comparison to the numeric values in column D1 after CAST has converted these values into 5-character character-strings.

**Sample Query 10.9b:** Reference DEMO2. Display column D1 values that begin with a minus sign, have a decimal point in the third position, and the digit 2 in the fourth position.

```
SELECT D1                                     DB2, ORACLE & SQL Server
FROM DEMO2
WHERE CAST (D1 AS CHAR (5)) LIKE '-_ .2%'
```

D1  
-5.28

**DB2:** DB2 supports additional data-type conversion functions such as DECIMAL and INTEGER. Execution of the following statement will produce the same result shown above for Sample Query 10.9a.

```
SELECT I1, DECIMAL (I1, 5, 2) I1ASDEC,
       D1, INTEGER (D1) D1ASINT
FROM DEMO2
```

## Summary

All systems support many arithmetic, character-string, and data-type conversion functions. This chapter only illustrated a few popular DB2, ORACLE, and SQL Server functions in order to present basic concepts. Again, you are encouraged to consult your SQL reference manual for a description of all individual functions provided by your system.

## Optional Summary Exercise

VARCHAR columns rarely contain character-strings with leading or trailing blanks. Assume you think that some character-strings with leading or trailing blanks somehow found their way into the V1 column in the DEMO2 table. You would like to discover these potentially problematic rows.

Code a SELECT statement to display the V1 value and its length if that value that has a blank in its first-character position or a blank in its last-character position.

The current version of DEMO2 does not have any V1 values with leading/trailing blanks. Therefore, to test your SELECT statement, you can execute the following two INSERT statements to insert two problematic rows. (Do not worry about details of these INSERT statements. INSERT will be covered in Chapter 15.)

```
INSERT INTO DEMO2 VALUES (999, 999, ' JOSEPHINE', 'XXX');
```

```
INSERT INTO DEMO2 VALUES (888, 888, 'JACQUELINE ', 'XXX');
```

After you have tested your SELECT statement, delete the two problematic rows by executing the following DELETE statement. (Again, do not worry about details of this DELETE statement. DELETE will be covered in Chapter 15.)

```
DELETE FROM DEMO2 WHERE I1 IN (999, 888);
```

## Processing DATE Values

This chapter continues our introduction to the individual functions. This chapter's functions fall into the category of "date-time" functions.

**Date-Time Data:** Before discussing SQL's date-time functions we must say a few words about date-time data. All real-world database systems contain information that represents dates, times, or timestamps (combination of date and time). More precisely, these systems provide special temporal data-types (e.g., DATE, TIME, and TIMESTAMP) for date-time data.

This chapter will focus on *temporal concepts*. It will illustrate these concepts via sample queries that access and manipulate a column defined as a DATE. Similar concepts apply to columns defined as TIME or TIMESTAMP. The last two sample queries in this chapter will illustrate access and manipulation of TIMESTAMP data.

Many SQL users merely "look at" DATE values and occasionally sort a result table by these values. These users only have to read the next four pages.

Alternatively, other users, especially applications developers, work on applications that involve considerable comparing and manipulation of DATE values. These users should read the entire chapter.

**Again, System Incompatibility:** Unfortunately, date-time functions show great *variation across different database systems*.



## Introduction: DEMO3 Table

We begin with an apparently simple question.

### What do DATE values look like?

Unfortunately, there is no single answer to this question. It depends upon your particular database system because each system has its own *default date-format*. For this reason, the following Figure 10.2a displays two images of the DEMO3 table that display different date-formats in the BDDATE columns.

MNAME	BDDATE	BDCHAR1	BDCHAR2	BDCHAR3
EVAN	<b>2017-06-05</b>	2017-06-05	06/05/2017	June 5, 2017
HANNAH	<b>2014-11-25</b>	2014-11-25	11/25/2014	November 25, 2014
JACQUELINE	<b>2019-01-10</b>	2019-01-10	01/10/2019	January 10, 2019
JESSIE	<b>1982-03-07</b>	1982-03-07	03/07/1982	March 7, 1982
JONHHY	<b>2015-05-10</b>	2015-05-10	05/10/2015	May 10, 2015
JOSEPHINE	<b>2017-06-13</b>	2017-06-13	06/13/2017	June 13, 2017
JULIE	<b>1978-05-17</b>	1978-05-17	05/17/1978	May 17, 1978

DB2 & SQL Server

MNAME	BDDATE	BDCHAR1	BDCHAR2	BDCHAR3
EVAN	<b>05-JUN-17</b>	2017-06-05	06/05/2017	June 5, 2017
HANNAH	<b>25-NOV-14</b>	2014-11-25	11/25/2014	November 25, 2014
JACQUELINE	<b>10-JAN-19</b>	2019-01-10	01/10/2019	January 10, 2019
JESSIE	<b>07-MAR-82</b>	1982-03-07	03/07/1982	March 7, 1982
JONHHY	<b>10-MAY-15</b>	2015-05-10	05/10/2015	May 10, 2015
JOSEPHINE	<b>13-JUN-17</b>	2017-06-13	06/13/2017	June 13, 2017
JULIE	<b>17-MAY-78</b>	1978-05-17	05/17/1978	May 17, 1978

ORACLE

**Figure 10.2a: DEMO3 Table**

The DEMO3 table was created by executing the same CREATE TABLE statement (Figure 10.2b) on three different systems: DB2, ORACLE, and SQL Server. On each system, the *BDDATE* column contains the same DATE values. However, when displaying these values, some systems display the BDDATE values using different date-formats.

The above BDDATE columns illustrate that DB2 and SQL Server display DATE values in the YYYY-MM-DD default format, whereas ORACLE displays DATE values in the DD-MON-YY default format. (On some systems, the DBA can optionally designate an alternative default date-format.)

**Optional Exercise:** On your system, execute: `SELECT * FROM DEMO3`  
The BDDATE column will illustrate your default date-format.

**Other "Date-Columns" - BDCHAR1, BDCHAR2, and BDCHAR3:** These columns are not "real" DATES. They are character-string columns with values that happen to look like dates. Examining the following CREATE TABLE statement that created the DEMO3 table will clarify this distinction.

```
CREATE TABLE DEMO3
(MNAME      CHAR (10) NOT NULL UNIQUE,
 BDDATE     DATE      NOT NULL, ←
 BDCHAR1    CHAR (10) NOT NULL,
 BDCHAR2    CHAR (10) NOT NULL,
 BDCHAR3    CHAR (20) NOT NULL)
```

**Figure 10.2b: Create DEMO3 Table**

**DATE Data-Type:** The BDDATE column is interesting because it has a DATE date-type; it contains "real" dates. The last three columns (BDCHAR1, BDCHAR2, and BDCHAR3) contain character-strings that look like dates.

When a column is defined as a DATE data-type, the database system will use some internal (hidden) coding scheme to represent DATE values. *We do not care* about this internal coding scheme. When a SELECT statement displays a DATE-column, the system automatically converts the internal date to the default date-format.

**Important Question:** Other than default date-formats, what is the difference between a DATE versus a character-string that looks like date?

**Answer:** *When processing DATE values,* the system utilizes the notion of **chronological time**. The following sample queries will illustrate this feature.

## ORDER BY Column with DATE values

When an ORDER BY clause references a DATE column, the sort sequence is a *chronological sequence*. Consider the following two sample queries.

**Sample Query 10.10a:** Reference DEMO3. Display MNAME and BDCHAR2 values in all rows. Sort the result by the BDCHAR2 column.

```
SELECT MNAME, BDCHAR2
FROM DEMO3
ORDER BY BDCHAR2
```

MNAME	BDCHAR2
JACQUELINE	01/10/2019
JESSIE	03/07/1982
JOHNNY	05/10/2015
JULIE	05/17/1978
EVAN	06/05/2017
JOSEFINE	06/13/2017
HANNAH	11/25/2014

Note: BDCHAR2 is *not* sorted within a chronological sequence. It is sorted within an ascending alphanumeric (collating) sequence.

**Sample Query 10.10b:** Reference DEMO3. Display MNAME and BDDATE values in all rows. Sort the result by the BDDATE column.

```
SELECT MNAME, BDDATE
FROM DEMO3
ORDER BY BDDATE
```

### DB2 & SQL Server Result

MNAME	BDDATE
JULIE	1978-05-17
JESSIE	1982-03-07
HANNAH	2014-11-25
JOHNNY	2015-05-10
EVAN	2017-06-05
JOSEPHINE	2017-06-13
JACQUELINE	2019-01-10

### ORACLE Result

MNAME	BDDATE
JULIE	17-MAY-78
JESSIE	03-MAR-82
HANNAH	25-NOV-14
JOHNNY	10-MAY-15
EVAN	05-JUN-17
JOSEPHINE	13-JUN-17
JACQUELINE	01-JAN-19

The BDDATE columns in these result tables, despite different date-formats, show an ascending *chronological sequence*. The older BDDATE values appear before the more recent BDDATE values. The preceding Sample Query 10.10a illustrates that chronological sorting does not apply to character-string columns.

Also, as you would expect, specifying DESC would produce a descending chronological sequence where the more recent dates appear before the older dates.

## Jump to the Next Chapter?

After reading the previous pages, many users have learned everything they need to know about DATE values. Such users can bypass the remainder of this chapter. Other users will be interested in the following temporal operations.

- Convert a DATE value to a character-string value. This is useful if you do not like your default date-format. (Sample Query 10.11)
- Convert a character-string value to a DATE value. (Sample Query 10.12)
- Display Today's Date. (Sample Query 10.13)
- Extract components (day, month, year) from a DATE value. (Sample Query 10.14)
- Display the weekday name (e.g., Tuesday) of a DATE value. (Sample Query 10.15)
- Reference a DATE column in a WHERE-clause. (Sample Query 10.16)

After presenting the above topics, Sample Queries 10.17 and 10.18 will introduce DATE calculations involving the temporal notion of an "interval" (e.g., 3 days).

### Important Exercise:

10A. Consider the following two statements where the ORDER BY clauses reference character-string columns. One of these statements (somehow) produces a desired result where the rows are sorted in chronological sequence. Which statement? Execute the statements to verify your answer.

```
SELECT MNAME, BDCHAR3
FROM DEMO3
ORDER BY BDCHAR3
```

```
SELECT MNAME, BDCHAR1
FROM DEMO3
ORDER BY BDCHAR1
```

[This is the only exercise in this chapter due to the great variation among DATE-functions supported by different database systems.]

## Data-Type Conversion: DATE → Character-String

As already stated, there is great variation among date-time functions across different database systems. The following sample queries will illustrate this fact.

Figure 10.3 (on following page) illustrates some date-formats that are supported within different database systems. Users frequently want to display a DATE value as a character-string using one of these date-formats. This is especially true if a user does not like her default date-format. To realize this objective:

- DB2 will use its CHAR function
- SQL Server will use its CONVERT function
- ORACLE will use its TO\_CHAR function.

**Sample Query 10.11:** Display all BDDATE values using the MM/DD/YYYY format.

**DB2:** The CHAR function specifies a code (e.g., USA) to identify the desired date-format. The DB2 reference manual shows multiple codes and associated date-formats.

```
SELECT BDDATE,                                     DB2
       CHAR (BDDATE, USA) MYFMT
FROM DEMO3
```

**SQL Server:** The CONVERT function specifies a code (e.g., 101) to identify the desired date-format. The SQL Server reference manual shows multiple codes and associated date-formats.

```
SELECT BDDATE,                                     SQL Server
       CONVERT (CHAR, BDDATE, 101) MYFMT
FROM DEMO3
```

Both the DB2 and SQL Server result tables look like:

<u>BDDATE</u>	<u>MYFMT</u>
2017-06-05	06/05/2017
2014-11-25	11/25/2014
2019-01-10	01/10/2019
1982-03-07	03/07/1982
2015-05-10	05/10/2015
2017-06-13	06/13/2017
1978-05-17	05/17/1978

**ORACLE:** The TO\_CHAR function can specify a date-format (e.g., MM/DD/YYYY). The ORACLE reference manual shows multiple date-formats.

```
SELECT BDDATE, ORACLE
       TO CHAR (BDDATE, 'MM/DD/YYYY') MYFMT
FROM DEMO3
```

ORACLE Result Table

<u>BDDATE</u>	<u>MYFMT</u>
05-JUN-17	06/05/2017
25-NOV-14	11/25/2014
10-JAN-19	01/10/2019
07-MAR-82	03/07/1982
10-MAY-15	05/10/2015
13-JUN-17	06/13/2017
17-MAY-78	05/17/1978

**Conclusion:** Learn you default date-format. If desired, you can use some SQL function to display your DATE values in an alternative format. See your SQL reference manual to learn details about the date-time formats supported by your system.

- YYYY-MM-DD (e.g., 2019-01-10) [DB2, SQL Server]
- YYYYMMDD (e.g., 20190110)
- MM/DD/YYYY (e.g., 01/10/2019)
- Month Day, Year (e.g., January 10, 2019)
- DD-MON-YY (e.g., 10-JAN-19) [ORACLE]
- DD-Mon-YYYY (e.g., 10-Jan-2019) [ORACLE]
- DD/MM/YYYY (e.g., 10/01/2019) [British Format]
- DD.MM.YYYY (e.g., 10.01.2019) [European Format]

**Figure 10.3: Common Date-Formats**

## Data-Type Conversion: Character-String → DATE

Users frequently want to convert a character-string to a DATE. Some systems, but not all systems, use the CAST function to realize this objective. (Recall that Sample Query 10.9a specified the CAST function to convert a DECIMAL value to an INTEGER value, and vice versa.) In the following sample query, DB2 and SQL Server use CAST to convert character-strings to DATES, and ORACLE uses its TO\_DATE function to perform this data-type conversion.

**Sample Query 10.12:** Display all MNAME, BDCHAR1, and BDCHAR2 values. Use your system's data-type conversion function to:

- Convert BDCHAR1 values to DATES
- Convert BDCHAR2 values to DATES

```
SELECT  MNAME,                                DB2 & SQL Server
        BDCHAR1, CAST (BDCHAR1 as DATE) MYDATE1,
        BDCHAR2, CAST (BDCHAR2 as DATE) MYDATE2
FROM DEMO3
ORDER BY MNAME
```

MNAME	BDCHAR1	MYDATE1	BDCHAR2	MYDATE2
EVAN	2017-06-05	2017-06-05	06/05/2017	2017-06-05
HANNAH	2014-11-25	2014-11-25	11/25/2014	2014-11-25
JACQUELINE	2019-01-10	2019-01-10	01/10/2019	2019-01-10
JESSIE	1982-03-07	1982-03-07	03/07/1982	1982-03-07
JONHHY	2015-05-10	2015-05-10	05/10/2015	2015-05-10
JOSEPHINE	2017-06-13	2017-06-13	06/13/2017	2017-06-13
JULIE	1978-05-17	1978-05-17	05/17/1978	1978-05-17

**DB2:** The DB2 version of CAST understands the BDCHAR1 and BDCHAR2 formats. However, DB2 does not understand the BDCHAR3 date-format.

**SQL Server:** The SQL Server version of CAST understands the BDCHAR1, BDCHAR2, and BDCHAR3 formats.

**Important Observation:** This result table shows that the BDCHAR1 and MYDATE1 columns appear to be identical. However, because MYDATE1 has a DATE data-type, it operates according to chronological time. This does not apply to the BDCHAR1 character-string.

**ORACLE:** The TO\_DATE function converts the BDCHAR1, BDCHAR2, and BDCHAR3 character-strings to DATE values.

Syntax: TO\_DATE (column, 'date-format')

This 'date-format' is the format of the stored column. (It is not the format of the desired result.) The desired result is ORACLE's default date-format. The ORACLE reference manual describes date-formats that are recognized by the TO\_DATE function.

**ORACLE**

```
SELECT MNAME, BDCHAR1, TO_DATE (BDCHAR1, 'YYYY-MM-DD') MYDATE1,  
       BDCHAR2, TO_DATE (BDCHAR2, 'MM/DD/YYYY') MYDATE2,  
       BDCHAR3, TO_DATE (BDCHAR3, 'MONTH DD, YYYY') MYDATE3  
FROM DEMO3  
ORDER BY MNAME
```

MNAME	BDCHAR1	MYDATE1	BDCHAR2	MYDATE2	BDCHAR3	MYDATE3
EVAN	2017-06-05	05-JUN-17	06/05/2017	05-JUN-17	June 5, 2017	05-JUN-17
HANNAH	2014-11-25	25-NOV-14	11/25/2014	25-NOV-14	November 25, 2014	25-NOV-14
JACQUELINE	2019-01-10	10-JAN-19	01/10/2019	10-JAN-19	January 10, 2019	10-JAN-19
JESSIE	1982-03-07	07-MAR-82	03/07/1982	07-MAR-82	March 7, 1982	07-MAR-82
JONHHY	2015-05-10	10-MAY-15	05/10/2015	10-MAY-15	May 10, 2015	10-MAY-15
JOSEPHINE	2017-06-13	13-JUN-17	06/13/2017	13-JUN-17	June 13, 2017	13-JUN-17
JULIE	1978-05-17	17-MAY-78	05/17/1978	17-MAY-78	May 17, 1978	17-MAY-78



## Today's DATE

All systems provide some method to display today's date. This is the date when the SQL statement is executed.

**Sample Query 10.13:** Display Jacqueline's BDDATE value, followed by today's date. Assume today is Feb 27, 2019.

**DB2:** DB2 provides a CURRENT\_DATE Register. CURRENT\_DATE is referenced just like any other column in a table, even though it is not a column in a table.

**SQL Server:** The GETDATE() function returns today's date-time value. Because this function returns time information along with date information, the CAST is used to convert the GETDATE() result to a DATE value without the time information.

```
SELECT MNAME, BDDATE BIRTHDAY, DB2
      CURRENT_DATE TODAY
FROM DEMO3
WHERE MNAME = 'JACQUELINE'
```

```
SELECT MNAME, BDDATE BIRTHDAY, SQL Server
      CAST (GETDATE () AS DATE) TODAY
FROM DEMO3
WHERE MNAME = 'JACQUELINE'
```

DB2 and SQL Server result tables look like:

<u>MNAME</u>	<u>BIRTHDAY</u>	<u>TODAY</u>
JACQUELINE	2019-01-10	2019-02-27

**ORACLE:** ORACLE provides a SYSDATE pseudo-column. SYSDATE is referenced like any other column in a table, even though it is not a column in a table.

```
SELECT MNAME, BDDATE BIRTHDAY, ORACLE
      SYSDATE TODAY
FROM DEMO3
WHERE MNAME = 'JACQUELINE'
```

<u>MNAME</u>	<u>BIRTHDAY</u>	<u>TODAY</u>
JACQUELINE	10-JAN-19	27-FEB-19

The SYSDATE pseudo-column also contains a time component. This result table shows that the time component is automatically removed from the result. Sample Query 10.19 will show how to use ORACLE's TO\_CHAR function with SYSDATE to display both date and time values.

## Extracting DATE Components

All systems provide functions that extract the year, month, and day components from DATE values.

**Sample Query 10.14:** Reference the DEMO3 table. Display the:

- MNAME and BDDATE columns
- Month component of BDDATE
- Day component of BDDATE
- Year component of BDDATE

**DB2 and SQL Server:** Both systems provide the MONTH, DAY, and YEAR functions which accept a DATE as an argument and return an integer result.

```
SELECT MNAME, BDDATE, DB2 & SQL Server
      MONTH (BDDATE) MYMONTH,
      DAY   (BDDATE) MYDAY,
      YEAR  (BDDATE) MYYEAR
FROM DEMO3
ORDER BY MNAME
```

MNAME	BDDATE	MYMONTH	MYDAY	MYYEAR
EVAN	2017-06-05	6	5	2017
HANNAH	2014-11-25	11	25	2014
JACQUELINE	2019-01-10	1	10	2019
JESSIE	1982-03-07	3	7	1982
JONHHY	2015-05-10	5	10	2015
JOSEPHINE	2017-06-13	6	13	2017
JULIE	1978-05-17	5	17	1978

**ORACLE:** The EXTRACT function specifies a "day", "month", or "year" argument to extract the desired component of a DATE.

```
SELECT MNAME, BDDATE, ORACLE
      EXTRACT (month FROM BDDATE) MYMONYH,
      EXTRACT (day   FROM BDDATE) MYDAY,
      EXTRACT (year  FROM BDDATE) MYYEAR
FROM DEMO3
ORDER BY MNAME
```

MNAME	BDDATE	MYMONTH	MYDAY	MYYEAR
EVAN	05-JUN-17	6	5	2017
HANNAH	25-NOV-14	11	25	2014
JACQUELINE	10-JAN-19	1	10	2019
JESSIE	07-MAR-82	3	7	1982
JONHHY	10-MAY-15	5	10	2015
JOSEPHINE	13-JUN-17	6	13	2017
JULIE	17-MAY-78	5	17	1978

## Weekday Names

You can display the weekday names (e.g., Monday, Tuesday) of DATE values.

**Sample Query 10.15:** For all rows, display the MNAME and BDDATE values, followed by the weekday name of each BDDATE value.

**DB2:** The DAYNAME function returns the weekday name.

```
SELECT MNAME, BDDATE, DAYNAME (BDDATE) BDDAY
FROM DEMO3
ORDER BY MNAME
```

**SQL Server:** The DATENAME function returns the weekday name. The first argument (dw) is one of many possible arguments that can be specified by this function.

```
SELECT MNAME, BDDATE, DATENAME (dw, BDDATE) BDDAY
FROM DEMO3
ORDER BY MNAME
```

<u>MNAME</u>	<u>BDDATE</u>	<u>DBDAY</u>
EVAN	2017-06-05	Monday
HANNAH	2014-11-25	Tuesday
JACQUELINE	2019-01-10	Thursday
JESSIE	1982-03-07	Sunday
JONHHY	2015-05-10	Sunday
JOSEPHINE	2017-06-13	Tuesday
JULIE	1978-05-17	Wednesday

**ORACLE:** The TO\_CHAR function can specify 'DAY' as an argument. This function supports other arguments.

```
SELECT MNAME, BDDATE, TO_CHAR (BDDATE, 'DAY') BDDAY
FROM DEMO3
ORDER BY MNAME
```

<u>MNAME</u>	<u>BDDATE</u>	<u>BDDAY</u>
EVAN	05-JUN-17	MONDAY
HANNAH	25-NOV-14	TUESDAY
JACQUELINE	10-JAN-19	THURSDAY
JESSIE	07-MAR-82	SUNDAY
JONHHY	10-MAY-15	SUNDAY
JOSEPHINE	13-JUN-17	TUESDAY
JULIE	17-MAY-78	WEDNESDAY

## What weekday were you born on?

The following examples assume that your birthdate is 1943-08-30. You can substitute your own birthdate for this date.

Each of the following SELECT statements applies the same general method. An individual function is nested within another individual function. First, a function (CAST or TO\_DATE) converts the '1943-08-30' character-string to a DATE value. Then a second function (DAYNAME, TO\_CHAR, or DATENAME) nests the first function as an argument to return the weekday name.

### DB2

```
SELECT DAYNAME (CAST ('1943-08-30' AS DATE)) MYBDAY
FROM DEMO3
```

### SQL Server

```
SELECT DATENAME (dw, CAST ('1943-08-30' AS DATE)) MYBDAY
FROM DEMO3
```

### ORACLE

```
SELECT TO_CHAR
      (TO_DATE ('1943-08-30', 'YYYY-MM-DD'), 'DAY') MYBDAY
FROM DEMO3
```

These statements produce the same result table. Because DEMO3 has eight rows, Monday appears eight times.

```
MYBDAY
Monday
Monday
Monday
Monday
Monday
Monday
Monday
Monday
```

You can specify DISTINCT to remove the duplicate rows. An alternative method is described in Appendix 10.5B.

## Comparing DATE Values

If a WHERE-clause references a DATE column, the system will perform the comparison operation based upon the notion of chronological time.

**Sample Query 10.16a:** Reference the DEMO3 table. Display all BDDATE values that predate the year 2000.

**DB2 & SQL Server:** Use CAST to convert '2000-01-01' to a DATE.

```
SELECT BDDATE DB2 & SQL Server
FROM DEMO3
WHERE BDDATE < CAST ('2000-01-01' AS DATE)
```

```
  BDDATE
 1982-03-07
 1978-05-17
```

**Syntax & Logic:** The CAST function was used to convert '2000-01-01' character-string to a DATE. Then the system compared the two DATE values according to chronological time.

---

**ORACLE:** Use TO\_DATE to convert '2000-01-01' to a DATE.

```
SELECT BDDATE ORACLE
FROM DEMO3
WHERE BDDATE < TO_DATE ('2000-01-01', 'YYYY-MM-DD')
```

```
  BDDATE
 07-MAR-82
 17-MAY-78
```

**Syntax & Logic:** The TO\_DATE function was used to convert '2000-01-01' ('YYYY-MM-DD' format) to a DATE. Then the system compared the two DATE values according to chronological time.

---

\* The above examples used built-in functions to *explicitly* convert character-strings to DATES. This allowed the WHERE-clauses to compare "a DATE to a DATE." We emphasize this point because, sometimes (perhaps unfortunately), you can implicitly convert a character-string to a DATE. The following page illustrates an example.

## Implicit versus Explicit Conversions to DATE

In Chapter 1, we noted that you could code WHERE FEE = 0 even though FEE is a DECIMAL. The system performed an implicit data-type conversion of 0 to 0.0 and everything worked. However, we discouraged this sloppy code. Likewise, sometimes, you can “get away with” an *implicit* conversion of a character-string to a DATE. The following statement, which satisfies the previous Sample Query 10.16, works in DB2 and SQL Server.

```
SELECT BDDATE                                DB2 & SQL Server
FROM DEMO3
WHERE BDDATE < '2000-01-01'                 ←Not-so-good
```

**Recommendation:** Avoid *implicit data-type conversion*. Systems can be fickle, and character-strings must be in some acceptable format. We recommend that, when you want to perform a chronological comparison, *explicitly* convert character-strings to DATES.

In the preceding Sample Query 10.16a, the BDDATE column already contained DATE values. Hence, we only had to convert the '2000-01-01' character-string to a DATE. The following sample query requires a chronological comparison of two character-strings.

**Sample Query 10.16b.** Reference the DEMO3 table. Display all BDCHAR2 values that predate the year 2000.

```
SELECT BDCHAR2                                DB2 & SQL Server
FROM DEMO3
WHERE CAST (BDCHAR2 AS DATE) <
      CAST ('2000-01-01' AS DATE)
```

```
SELECT BDCHAR2                                ORACLE
FROM DEMO3
WHERE TO_DATE (BDCHAR2, 'MM/DD/YYYY') <
      TO_DATE ('2000-01-01', 'YYYY-MM-DD')
```

```
BDCHAR2
03/07/1982
05/17/1978
```

**Logic:** Conversion functions explicitly converted both character-strings to DATE values before the less-than (<) comparison operation.

## Calculations Involving DATE Values

Consider some common-sense questions about DATE-Calculations.

- Does it make sense to "add" two DATE values?
- Does it make sense to "subtract" two DATE values?
- Does it make sense to "multiple" two DATE values?
- Does it make sense to "divide" two DATE values?

Your intuitive answers to these questions should offer some insight into SQL's acceptance or rejection of the following expressions.

DATE1 + DATE2	not reasonable → reject
DATE1 - DATE2	reasonable → accept
DATE1 * DATE2	not reasonable → reject
DATE1 / DATE2	not reasonable → reject

Most readers correctly conclude that adding, multiplying, and dividing two DATE values does not make sense. But, subtracting one DATE value from another is reasonable. For example, a common temporal calculation involves subtracting a birth-date from today's date.

TODAY - BIRTHDATE

Most systems support DATE-Subtraction and will accept the above expression. (Other systems provide a built-in function that performs DATE-subtraction.)

**Important Question:** When you subtract a DATE from a DATE (e.g., TODAY - BIRTHDATE), is the result a DATE? Before answering this question, it may be helpful to revisit SQL calculations that we have already encountered.

**Numeric Calculations:** When you add, subtract, multiply, or divide any two numbers, the result is (surprise) a number (unless you try to divide by zero).

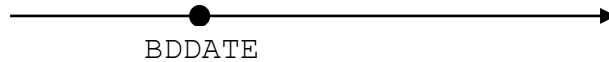
**Character-String Calculations:** Consider the CONCAT function to be a form of "string-addition," and the SUBSTR function to be a form of "string-subtraction." These operations produce a result which is (surprise) a character-string.

However, unlike numeric and character-string calculations, when you subtract a DATE from a DATE (e.g., DATE1 - DATE2) the result is not another DATE. Instead, DATE-subtraction returns an *interval*, which is described on the following page.

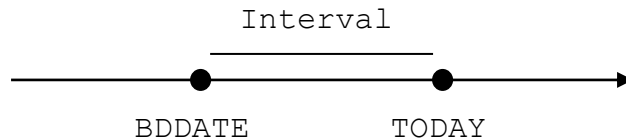
## Intervals

The notion of an interval is another important temporal concept, and all systems offer some support for intervals. (DB2 uses the term "duration" instead of "interval.")

**Points on a Timeline:** Within the context of DATES (without a time component), a date is the smallest atomic unit of time. Therefore, a DATE can be illustrated as a point on a *timeline* as shown below.



**Intervals:** An interval is a temporal distance between two DATES as illustrated below.



An interval can be measured in some number of days, or months, or years. (Time intervals are measured in terms of hours, minutes, seconds, milliseconds, etc.)

**Temporal Calculations:** Intervals allows you to perform date-time calculations. For example, the above interval represents a person's age. You can measure this interval in terms of years, months, or days. The following Sample Query 10.17 will display this interval in terms of days.

**INTERVAL Data-Type:** Some database systems support the built-in INTERVAL data-type (not described here) that directly represents an interval.



## DATE Subtraction

The following sample query illustrates DATE-subtraction that returns an interval result. Assume today is March 1, 2019.

**Sample Query 10.17:** Display Hannah and Jacqueline's MNAME and BDDATE values, followed by today's date, followed by their age in days.

**DB2:** The DAYS function returns the number of days after Day 0001-01-01. Apply DAYS to both CURRENT\_DATE and BDDATE. The difference between the results is the age in terms of days.

```
SELECT MNAME, BDDATE BIRTHDAY, CURRENT_DATE TODAY, DB2
       DAYS (CURRENT_DATE) - DAYS (BDDATE) AGEDAYS
FROM DEMO3
WHERE MNAME IN ('HANNAH', 'JACQUELINE')
```

MNAME	BIRTHDAY	TODAY	AGEDAYS
HANNAH	2014-11-25	2019-03-01	1557
JACQUELINE	2019-01-10	2019-03-01	50

*Careful!* DB2 allows you to simply subtract two DATE values without specifying the DAYS function, as shown below. However, this result may not conform to your intuition. Hannah's age of 40306 represents 4 years, 03 months, and 06 days.

```
SELECT MNAME, BDDATE BIRTHDAY, CURRENT_DATE TODAY, DB2
       CURRENT_DATE - BDDATE YMMDD
FROM DEMO3
WHERE MNAME IN ('HANNAH', 'JACQUELINE')
```

MNAME	BIRTHDAY	TODAY	YMMDD
HANNAH	2014-11-25	2019-03-01	40306
JACQUELINE	2019-01-10	2019-03-01	122

**ORACLE:** Recall that SYSDATE stores today's date and time values. Use the TRUNC function to truncate the time component before subtracting the BDDATE value. The result is person's age measured in days.

```
SELECT MNAME, BDDATE BIRTHDAY, SYSDATE TODAY, ORACLE
       TRUNC (SYSDATE) - BDDATE AGEDAYS
FROM DEMO3
WHERE MNAME IN ('HANNAH', 'JACQUELINE')
```

MANME	BIRTHDAY	TODAY	AGEDAYS
HANNAH	25-NOV-14	01-MAR-19	1557
JACQUELINE	10-JAN-19	01-MAR-19	50

**SQL Server:** SQL Server provides a DATEDIFF function to calculate the interval difference between two DATE values. This function allows you to specify the interval unit ("day", "month", or "year") for the result. Here, the result is measured in days.

```
SELECT MNAME, BDDATE,
       CAST (GETDATE() AS DATE) TODAY,
       DATEDIFF (day, BDDATE, GETDATE() ) AGEDAYS
FROM DEMO3
WHERE MNAME IN ('HANNAH', 'JACQUELINE')
```

**SQL Server**

MNAME	BIRTHDAY	TODAY	AGEDAYS
HANNAH	2014-11-25	2019-03-01	1557
JACQUELINE	2019-01-10	2019-03-01	50

**Other Intervals:** We usually want to display a person's age as some number of years. This interval value can be approximated by dividing the calculated AGEDAYS value by 365. We say "approximated" because some years have 366 days. To address this imprecision, most systems also support MONTHS and YEARS intervals.

## DATE + Interval; DATE – Interval

The following query objectives require adding and subtracting intervals to DATE values to produce DATE values.

**Sample Query 10.18:** Reference the DEMO3 table. Display each BDDATE value followed by three other DATE values which are: 10 days, 3 months, and 5 years *after* the BDDATE value.

**DB2:** DB2 supports the DAYS, MONTHS, and YEARS keywords to represent intervals. These keywords are preceded by a number to specify the size of the interval.

```
SELECT BDDATE,
       BDDATE + 10 DAYS PLUS10D,
       BDDATE + 3 MONTHS PLUS3M,
       BDDATE + 5 YEARS PLUS5Y
FROM DEMO3
```

**DB2**

BDDATE	PLUS10D	PLUS3M	PLUS5Y
2017-06-05	2017-06-15	2017-09-05	2022-06-05
2014-11-25	2014-12-05	2015-02-25	2019-11-25
2019-01-10	2019-01-20	2019-04-10	2024-01-10
1982-03-07	1982-03-17	1982-06-07	1987-03-07
2015-05-10	2015-05-20	2015-08-10	2020-05-10
2017-06-13	2017-06-23	2017-09-13	2022-06-13
1978-05-17	1978-05-27	1978-08-17	1983-05-17

A similar query objective requires subtracting an interval from a DATE value to produce a DATE result. Display the BDDATE column followed by more three dates which are: 10 days, 3 months, and 5 years *before* the BDDATE values.

```
SELECT BDDATE,
       BDDATE - 10 DAYS MINUS10D,
       BDDATE - 3 MONTHS MINUS3M,
       BDDATE - 5 YEARS MINUS5Y
FROM DEMO3
```

**DB2**

BDDATE	MINUS10D	MINUS3M	MINUS5Y
2017-06-05	2017-05-26	2017-03-05	2012-06-05
2014-11-25	2014-11-15	2014-08-25	2009-11-25
2019-01-10	2018-12-31	2018-10-10	2014-01-10
1982-03-07	1982-02-25	1981-12-07	1977-03-07
2015-05-10	2015-04-30	2015-02-10	2010-05-10
2017-06-13	2017-06-03	2017-03-13	2012-06-13
1978-05-17	1978-05-07	1978-02-17	1973-05-17

**SQL Server:** SQL Server supports the DATEADD function which allows you to add some number of days, weeks, or years to a DATE value to return another DATE value.

```
SELECT BDDATE,
       DATEADD (day, 10, BDDATE) PLUS10D,
       DATEADD (month, 3, BDDATE) PLUS3M,
       DATEADD (year, 5, BDDATE) PLUS5Y
FROM DEMO3
```

You can specify a negative value to subtract units of days, months, or years.

```
SELECT BDDATE,
       DATEADD (day, -10, BDDATE) MINUS10D,
       DATEADD (month, -3, BDDATE) MINUS3M,
       DATEADD (year, -5, BDDATE) MINUS5Y
FROM DEMO3
```

[DATEADD also allows you to specify units of time such as hours, minutes, or seconds.]

**ORACLE:** If you add a number to a DATE, the system assumes the number represents days. ORACLE also supports an ADD\_MONTHS function to add some number of months to a DATE value. This function can be used to add years by converting years to months.

The following statement adds: 10 days (PLUS10D), 3 months (PLUS3M), and 5 years (PLUS5Y) to the BDDATE value; and it subtracts: 10 days (MINUS10D), 3 months (MINUS3M), and 5 years (MINUS5Y) from the BDDATE value.

```
SELECT BDDATE,
       BDDATE + 10 PLUS10D,
       ADD_MONTHS (BDDATE , 3) PLUS3M,
       ADD_MONTHS (BDDATE , 60) PLUS5Y,
       BDDATE -10 MINUS10D,
       ADD_MONTHS (BDDATE , -3) MINUS3M,
       ADD_MONTHS (BDDATE , -60) MINUS5Y
FROM DEMO3
```

BDDATE	PLUS10D	PLUS3M	PLUS5Y	MINUS10D	MINUS3M	MINUS5Y
05-JUN-17	15-JUN-17	05-SEP-17	05-JUN-22	26-MAY-17	05-MAR-17	05-JUN-12
25-NOV-14	05-DEC-14	25-FEB-15	25-NOV-19	15-NOV-14	25-AUG-14	25-NOV-09
10-JAN-19	20-JAN-19	10-APR-19	10-JAN-24	31-DEC-18	10-OCT-18	10-JAN-14
07-MAR-82	17-MAR-82	07-JUN-82	07-MAR-87	25-FEB-82	07-DEC-81	07-MAR-77
10-MAY-15	20-MAY-15	10-AUG-15	10-MAY-20	30-APR-15	10-FEB-15	10-MAY-10
13-JUN-17	23-JUN-17	13-SEP-17	13-JUN-22	03-JUN-17	13-MAR-17	13-JUN-12
17-MAY-78	27-MAY-78	17-AUG-78	17-MAY-83	07-MAY-78	17-FEB-78	17-MAY-73

## Time Data

Some applications need to store information about time. Again, there is considerable variation among different systems. We only say a few words on this topic to help you get started reading your SQL reference manual.

DB2 supports a `TIME` data-type which represents hour, minute, and second values. It also supports a `TIMESTAMP` data-type which represents year, month, day, hour, minute, second, and microsecond values.

SQL Server supports a `TIME` data-type which represents hour, minute, second, and fractional seconds. It also supports a `DATETIME` data-type which represents year, month, day, hour, minute, second, and fractional seconds. (A `DATETIME2` data-type is same as the `DATETIME` data-type, but is has greater accuracy in its fractional seconds.)

ORACLE supports a `TIMESTAMP` data-type which represents year, month, day, hour, minute, second, and fractional seconds.

Because time values are not stored in the `DEMO3` table, we use today's time to demonstrate system-specific methods to access today's time of statement execution. Assume today's date is March 1, 2019. The time values vary in the following results because these statements were executed at different times.

**Sample Query 10.19:** Display all `MNAME` values followed by today's date and time of execution.

**DB2:** In addition to the `CURRENT_DATE` register, DB2 also supports a `CURRENT_TIME` and `CURRENT_TIMESTAMP` registers. `CURRENT_TIME` returns hour, minute, and second values, whereas `CURRENT_TIMESTAMP` returns year, month, day, hour, minute, second, and millisecond values. The following statement specifies `CURRENT_TIMESTAMP`.

```
SELECT MNAME, DB2
       CURRENT_TIMESTAMP MYDATETIME
FROM DEMO3
ORDER BY MNAME
```

<u>MNAME</u>	<u>MYDATETIME</u>
EVAN	2019-03-01 15:23:58.441
HANNAH	2019-03-01 15:23:58.441
JACQUELINE	2019-03-01 15:23:58.441
JESSIE	2019-03-01 15:23:58.441
JONHHY	2019-03-01 15:23:58.441
JOSEPHINE	2019-03-01 15:23:58.441
JULIE	2019-03-01 15:23:58.441

**SQL Server:** The GETDATE() function returns year, month, day, hour, minute, second, and millisecond values.

```
SELECT MNAME,
       GETDATE () MYDATETIME
FROM DEMO3
```

**SQL Server**

MNAME	MYDATETIME
EVAN	2019-03-01 16:49:14.613
HANNAH	2019-03-01 16:49:14.613
JACQUELINE	2019-03-01 16:49:14.613
JESSIE	2019-03-01 16:49:14.613
JONHHY	2019-03-01 16:49:14.613
JOSEPHINE	2019-03-01 16:49:14.613
JULIE	2019-03-01 16:49:14.613

If you need greater time accuracy, you can reference SQL Server's SYSDATETIME function.

**ORACLE:** ORACLE's SYSDATE pseudo-column contains both today's date and time values. (Recall that Sample Query 10.14 referenced SYSDATE which, by default, only returned the date value.) To access the time component, you must use the TO\_CHAR function to convert the SYSDATE value to a character-string pattern that includes both date and time. The ORACLE reference manual describes many such patterns.

```
SELECT MNAME,
       TO_CHAR (SYSDATE, 'DD-MM-YYYY HH24:MI:SS') MYDATETIME
FROM DEMO3
ORDER BY MNAME
```

**ORACLE**

MNAME	MYDATETIME
EVAN	01-03-2019 15:27:35
HANNAH	01-03-2019 15:27:35
JACQUELINE	01-03-2019 15:27:35
JESSIE	01-03-2019 15:27:35
JONHHY	01-03-2019 15:27:35
JOSEPHINE	01-03-2019 15:27:35
JULIE	01-03-2019 15:27:35

If you need greater time accuracy, you can reference ORACLE's SYSTIMESTAMP pseudo-column.

**Other Time Processing:** As with DATE values, your system provides methods to reformat, compare, and calculate time values. This book only illustrates the extraction of time components.

## Extracting Time Components

Sample Query 10.14 illustrated functions that extract the year, month, and day components of a DATE value. Similar functions can extract the hour, minute, and second components of TIME and TIMESTAMP values. Because time values are not stored in the BDDATE column, we use today's timestamp to demonstrate system-specific methods to access today's time of statement execution.

**Sample Query 10.20:** Reference the DEMO3 table. Display the:

- MNAME column
- Hour component of the current timestamp
- Minute component of the current timestamp
- Second component of the current timestamp

The following DB2, SQL Server, and ORACLE functions produce the same result table (with the exception that ORACLE will also display the fractional part of a second).

MNAME	MYHOUR	MYMINUTE	MYSECOND
EVAN	10	14	2
HANNAH	10	14	2
JACQUELINE	10	14	2
JESSIE	10	14	2
JONHHY	10	14	2
JOSEPHINE	10	14	2
JULIE	10	14	2

**DB2:** Utilize the HOUR, MINUTE, and SECOND functions.

```
SELECT MNAME,
       HOUR   (CURRENT_TIMESTAMP) MYHOUR,
       MINUTE (CURRENT_TIMESTAMP) MYMINUTE,
       SECOND (CURRENT_TIMESTAMP) MYSECOND
FROM DEMO3
ORDER BY MNAME
```

**SQL Server:** Specify the DATEPART function with HOUR, MINUTE, or SECOND as arguments. (The DATEPART function can also be used to extract year, month, and date components.)

```
SELECT MNAME,
       DATEPART (HOUR, GETDATE()) MYHOUR,
       DATEPART (MINUTE, GETDATE()) MYMINUTE,
       DATEPART (SECOND, GETDATE()) MYSECOND
FROM DEMO3
ORDER BY MNAME
```

**ORACLE:** Specify the EXTRACT function with HOUR, MINIUTE, or SECOND as arguments.

```
SELECT MNAME, ORACLE
      EXTRACT (HOUR FROM SYSTIMESTAMP) MYHOUR,
      EXTRACT (MINUTE FROM SYSTIMESTAMP) MYMINUTE,
      EXTRACT (SECOND FROM SYSTIMESTAMP) MYSECOND
FROM DEMO3
ORDER BY MNAME
```

## Summary

All systems support date-time data-types and functions. This chapter only illustrated a few popular DB2, ORACLE, and SQL Server functions in order to present basic concepts.

Undoubtedly, you have observed that date-time functions are more complex than the basic arithmetic and character-string functions introduced in Chapter 10. This occurs because **date-time processing is inherently complex**. Consider the following questions that this chapter did *not* address.

- What is the oldest historical date that your system can represent? Also, if your system supports very old historical dates, how does it handle the transition from the Julian Calendar to Gregorian Calendar which was implemented in different years in different countries?
- How does your system incorporate time zones?
- How does your system handle daylight savings time which applies within some geographic locations but not others?

Many systems address some of these issues. The concepts are similar, but coding details differ. Again, you are encouraged to consult your SQL reference manual for a description of all date-time data-types and functions provided by your system.



## Appendix 10.5A: Theory

We mention two topics that may be of interest to application developers who have some experience with object-oriented programming languages (e.g., C++ and JAVA) or work on projects that have significant date-time processing requirements.

**Object Orientation:** The DATE data-type uses some internal (hidden) coding scheme to represent DATE values, and this chapter's sample queries illustrate that users do not need to consider this coding scheme. This implies that DATE is a built-in abstract data-type. In addition to other built-in abstract data-types (e.g., TIME, TIMESTAMP), many systems support user-defined abstract data types via a CREATE TYPE statement.

Without explanation, we note that supporting abstract data-types is an important step towards to providing *object-oriented* functionality within a relational database system. This is an advanced topic that is not discussed in this book.

**Temporal Databases:** This is an advanced topic that transcends the date-time concepts presented in this book. Briefly, a *temporal database* supports the notions of: (1) "*valid time*" which designates when a data-item becomes true in the real world, and (2) "*transaction time*" which designates when a data-item was recorded within the database.

\*\*\*\*\* Within the context of SQL, the *best resource* to begin learning about temporal databases is a wonderful (but slightly dated) book by Richard Snodgrass, [Developing Time-Oriented Database Applications in SQL](#). You can purchase this book from Amazon and other book sellers, or you can obtain a **free** PDF version of this book by visiting Richard Snodgrass's web site at the University of Arizona. [Thank you, Richard!!!]

## Appendix 10.5B: One-Row “Dummy” Tables

Reconsider the three SELECT statements that satisfied the “What weekday were you born on?” question. The DB2 solution looked like:

```
SELECT DAYNAME (CAST ('1943-08-30' AS DATE)) MYBDAY
FROM DEMO3
```

Because DEMO3 has 8 rows, the result shows 8 rows.

```
MYBDAY
Monday
Monday
Monday
Monday
Monday
Monday
Monday
Monday
```

We could specify DISTINCT to remove duplicate rows, but there is an alternative way. First, note that this query objective *does not require access to the DEMO3 table or any other table*. We only used DEMO3 because it is a small table, and every SELECT statement (excluding SQL Server) must specify a FROM-clause.

Some systems (e.g., DB2, ORACLE) provide a “dummy” table with just one row. Referencing this table allows you to produce the following a one-row result table.

```
MYBDAY
Monday
```

**DB2:** DB2 supports a dummy table called SYSIBM.SYSDUMMY1.

```
SELECT DAYNAME (CAST ('1943-08-30' AS DATE)) MYBDAY
FROM SYSIBM.SYSDUMMY1
```

**ORACLE:** ORACLE supports a dummy table called DUAL.

```
SELECT TO_CHAR
      (TO_DATE ('1943-08-30', 'YYYY-MM-DD'), 'DAY') MYBDAY
FROM DUAL
```

**SQL Server** users use another method to produce a one-row result table. Instead of a dummy table, SQL Server supports a “no table” option, where you can code a SELECT statement without a FROM-clause.

```
SELECT DATENAME (dw, CAST ('1943-08-30' AS DATE)) MYBDAY
```

This page is intentionally blank.

## Null Values

A null value indicates that a value is unknown. In this book, a hyphen (-) is used to represent a null value. For example, the hyphen in the COLY column of the following table indicates that the COLY value is unknown.

COLX	COLY
100	200
400	-
700	100

[Different front-end tools display different symbols to represent null values. One tool might display the question mark, another might display "null," and yet another might display blanks.]

Below, we revisit the CREATE TABLE statement for the PRESERVE table to note that the optional NOT NULL clause was specified for every column.

```
CREATE TABLE PRESERVE
(PNO          INTEGER          NOT NULL UNIQUE,
 PNAME       VARCHAR (25)     NOT NULL,
 STATE       CHAR (2)         NOT NULL,
 ACRES       INTEGER          NOT NULL,
 FEE         DECIMAL (5,2)    NOT NULL)
```

Specification of a NOT NULL clause tells the system to prohibit any operation that attempts to store a null value in a column. Hence, all previous sample queries and exercises did not have to consider the problematic (but interesting) issues associated with null values. Sample queries that address these issues will be presented in this chapter.

## NTAB and NTAB2 Tables

This chapter's sample queries will reference the following NTAB table (Figure 11.1), and the exercises will reference the NTAB2 table (Figure 11.2). The following CREATE TABLE statements, which created these tables, used different methods to designate that a column is allowed to contain null values.

The column definitions in NTAB explicitly specify NULL implying that null values are allowed. Because the column definitions in NTAB2 do not specify NULL or NOT NULL, the system assumes the default of NULL has been specified. Hence all columns in these tables will accept null values.

```
CREATE TABLE NTAB
(A INTEGER NULL,
 B INTEGER NULL)
```

```
CREATE TABLE NTAB2
(A INTEGER,
 B INTEGER)
```

Assume that INSERT operations have inserted some rows with null values. (The INSERT statement will be introduced in Chapter 15.) In particular, observe that the last row in each table contains all null values, a valid but most unusual situation.

A	B
5	5
5	10
5	-
-	10
-	-

Figure 11.1: NTAB Table

A	B
10	-
15	10
-	30
-	10
40	40
-	-

Figure 11.2: NTAB2 Table

**Complexity:** Sample Queries 11.1-11.11 will introduce some problematic issues associated with null values. After examining these sample queries, you might conclude that null values are unnecessarily complex and hope that every column is declared as NOT NULL. However, null values appear in many real-world tables. You cannot avoid them.

There are two general coding techniques that address problematic issues associated with null values. Your SQL code may specify a:

- (1) WHERE-clause that rejects rows with null values.  
or
- (2) Built-in function that substitutes a real value for a null value.

Sample Queries 11.12-11.14 will illustrate these techniques.

## Arithmetic Expressions Involving Null Values

Assume you know how many dollars you have in your pocket. Call this amount A. Also, assume you do not know how many dollars I have in my pocket. Call this amount B. How many total dollars will we have if we pool our money? (What is A+B?) You don't know. The following sample query illustrates this logic.

**Sample Query 11.1:** Reference the NTAB table.  
Display columns A, B, and the value of A+B.

A	B
5	5
5	10
5	-
-	10
-	-

NTAB

```
SELECT A, B, A+B
FROM NTAB
```

A	B	A+B
5	5	10
5	10	15
5	-	-
-	10	-
-	-	-

**Logic:** An arithmetic expression returns a null value if any operand is null. This behavior is reasonable and conforms to the intuition of most users.

### Exercise:

11A. Display the result table produced by executing:

```
SELECT A, B, A-B
FROM   NTAB2
```

## Aggregate Functions Involving Null Values

*Careful! Unlike arithmetic expressions, aggregate functions simply ignore null values.*

**Sample Query 11.2:** Reference the NTAB table. What is the sum of column A?

A	B
5	5
5	10
5	-
-	10
-	-

NTAB

```
SELECT SUM (A)
FROM NTAB
```

```
SUM (A)
-----
      15
```

**Important Observations:** After reading Sample Queries 11.1 and 11.2, you might think: That's strange. In the presence of null values, an aggregate function behaves differently than an arithmetic expression. The essence of this difference is:

As Sample Query 11.1 illustrates, within an arithmetic expression, if any operand is null value, the result is a null value.

As the above Sample Query 11.2 illustrates, an aggregate function ignores null values. The function performs the calculation using only the non-null values.

Some users are shocked by this apparently inconsistent behavior. They say something like: "This is crazy!!!!" This apparently inconsistent behavior can become confusing. See the following Sample Query 11.3.

**Special Case Circumstance:** An aggregate function (e.g., SUM, MAX, MIN, AVG, COUNT) will return a null value if the function references a column (or an intermediate-result column) that contains all null values.

### Exercise:

11B. Display the result table produced by executing:

```
SELECT SUM(A), SUM(B)
FROM   NTAB2
```

## Careful!!!

Before the computer era, paper-and-pencil bookkeepers would cross-tabulate a table to detect a calculation error. Consider the following table without null values.

A	B	A+B
5	5	10
5	10	15
100	5	105
200	10	210
100	100	200
-----		
410	130	<b>540</b>

The bookkeeper hoped that  $SUM(A+B)$  would equal  $SUM(A)+SUM(B)$ .

$$SUM(A+B) = 10 + 15 + 105 + 210 + 200 = 540$$

$$SUM(A)+SUM(B) = 410 + 130 = 540$$

Obtaining the same result (540) offers pretty good evidence that there were no manual calculation errors.

**Sample Query 11.3:** Perform a cross-tabulation on the NTAB table. Show (A+B) does not equal  $SUM(A)+SUM(B)$ .

A	B
5	5
5	10
5	-
-	10
-	-

**NTAB**

```
SELECT SUM(A+B), SUM(A)+SUM(B)
FROM NTAB
```

SUM(A+B)	SUM(A)+SUM(B)
25	40

**Logic:**

A	B	A+B
5	5	10
5	10	15
5	-	-
-	10	-
-	-	-
-----		
15	25	<b>(40 or 25)</b>

Conclusion: Be careful when calculating with null values.

**Exercise:**

11C. Display the result table produced by executing:

```
SELECT SUM(A+B), SUM(A) + SUM(B)
FROM NTAB2
```



## Comparing with Null Values

Assume you know how many dollars you have in your pocket. Call this amount A. You do not know how many dollars I have in my pocket. Call this amount B. Do you know if we both have the same number of dollars? (Does A=B?) You don't know. The A=B result is unknown. SQL applies the same logic.

The system only selects a row if a WHERE-condition evaluates to True. If the condition evaluates to False or Unknown, the row is not selected. Consider the following example. The interesting row is the last row where both A and B contain a null value.

**Sample Query 11.4:** Display all rows from NTAB where A = B.

A	B
5	5
5	10
5	-
-	10
-	-

NTAB

```
SELECT *
FROM   NTAB
WHERE  A = B
```

A	B
5	5

**Logic:** Consider the A=B condition for each row in NTAB.

1. The first row is selected. ("5=5" is True).
2. The second row is not selected. ("5=10" is False).
3. The third row is not selected  
("5=null" is Unknown: only True implies selection.)
4. The fourth row is not selected  
("null=10" is Unknown: only True implies selection.)
5. The fifth row is not selected. You may find this to be confusing. After all, both A and B contain null values. We will explain this logic on the following page. For the moment, we simply state that:

***A null value is not equal to another null value!***

This sample query specifies an equals (=) comparison operator. The same logic applies to the other comparison operators. For example, a greater than (>) comparison is "unknown" if either or both of the operands are null.

**"NULL=NULL" is Unknown:** This logic can be explained by asking you to consider two people, Person-A and Person-B. Assume you do not know how many dollars each person has in his pocket. (A is null, and B is null.) Obviously, you cannot conclude that both people have the same number of dollars. Hence "NULL=NULL" is Unknown:

Occasionally someone will disagree and say something like:

"Given any value X, it must be true that X=X. Therefore, it should be the case that NULL = NULL also evaluates to True."

This comment is not valid because X=X assumes that X is a *value*.

However! **NULL is not a value.**

**NULL indicates the absence of a value.**

For any value X, X=X does indeed evaluate to True. However, because *NULL is not a value*, we cannot conclude that two NULL "values" are equal to each other.

**Then, why do we call them Null "values?"** We should not call them "NULL values." However, we do so because almost everyone else does, including your SQL reference manual. (Ted Codd, who started all this null-value business, later said that he wished he had called them null "marks" where a mark indicates the absence of a value.)

**Exercise:**

11D. Display the result table produced by executing:

```
SELECT *
FROM   NTAB2
WHERE  A = B
```

**"NULL<>NULL" is Unknown:** Referring to the previous scenario, both persons may or may not have the same number dollars in their pockets. You don't know. Therefore, if you cannot conclude that Person-A and Person-B have the same number of dollars in their pockets, then, likewise, you cannot conclude that these persons do not have the same number of dollars in their pockets.

**Sample Query 11.5:** Display all rows from NTAB where A <> B.

A	B
5	5
5	10
5	-
-	10
-	-

**NTAB**

```
SELECT *
FROM   NTAB
WHERE  A <> B
```

A	B
5	10

**Logic:** Consider the A<>B condition for each row in NTAB.

1. The first row is not selected.  
("5<>5" is False, implying the row is not selected.)
2. The second row is selected.  
("5<>10" is True, implying the row is selected.)
3. The third row is not selected.  
("5<>null" is Unknown: only True implies selection.)
4. The fourth row is not selected  
("null<>10" is Unknown: only true implies selection.)
5. The fifth row is not selected.  
("null<>null" is Unknown: only True implies selection.)

**Exercise:**

11E. Display the result table produced by executing:

```
SELECT *
FROM   NTAB2
WHERE  A <> B
```

## COUNT (\*) – Careful with Nulls!

Sample Query 8.4.1 introduced the COUNT(\*) function that returned the number of selected rows *without considering the values that are stored in the rows*. Therefore, the presence or absence of null values does not impact the behavior of COUNT(\*). For example,

```
SELECT COUNT(*) ROWCT
FROM NTAB

      ROWCT
      ----
         5
```

**Incorrect Do-it-yourself Average:** Using COUNT(\*) would lead to an error if you try to calculate a do-it-yourself average. Consider the following statement that *incorrectly* attempts to calculate the average of the known values in column A.

A	B
5	5
5	10
5	-
-	10
-	-

```
SELECT SUM (A * 1.00)/COUNT(*)  BADAVG
FROM NTAB

      BADAVG
      ----
         3.00  → Error
```

NTAB

Here, SUM(A) returned 15.00, and COUNT(\*) returned 5  
Then, (15.00/5) = 3.00

**Correct Average:** The correct average is 5.00 as shown below.

```
SELECT AVG (A*1.00) GOODAVG
FROM NTAB

      GOODAVG
      ----
         5.00
```

Here, the AVG function simply ignored the NULL values and calculated 15.00/3 = 5.

The following page presents another variation of the COUNT function that will be used to correctly code a do-it-yourself average.

## COUNT (column)

The COUNT (column) function returns the number of *non-null* values in a specified column. The following sample query illustrates the behavior of COUNT (column).

**Sample Query 11.6:** Reference the NTAB table. How many non-null values are in column A?

A	B
5	5
5	10
5	-
-	10
-	-

NTAB

```
SELECT COUNT (A)
FROM  NTAB

COUNT (A)
-----
          3
```

**Observation:** The following statement produces the correct average of Column A.

```
SELECT SUM(A * 1.00)/COUNT(A) GOODAVG
FROM  NTAB

GOODAVG
-----
      5.00
```

Here, SUM(A \* 1.00) returned 15.00, and COUNT(A) returned 3  
Then, (15.00/3) = 5.00

### Exercise:

11F. Display the result table produced by executing:

```
SELECT COUNT (*), COUNT (A), COUNT (B)
FROM  NTAB2
```

## Three-Value Logic (3VL) System

**Two-Value Logic (2VL):** In previous chapters, when considering a WHERE-clause, the truth value for a condition was either (1) True or (2) False. There were only two truth-values. There was no third truth-value. Hence, we had a **2-Value Logic (2VL)** system.

**Three-Value Logic (3VL):** Sample Queries 11.4 and 11.5 showed that, in the presence of Null values, a WHERE-clause could have three possible truth values: True (T), False (F), and Unknown (U). Hence, we have **3-Value Logic (3VL)** system. Understanding the 3VL is important. Consider the following scenario.

A user knows there are 5 rows in NTAB. Then he executes the following statement and observes the result.

```
SELECT COUNT(*) CT
FROM   NTAB
WHERE  A=B
```

```
CT
 1
```

Next, he considers, but does not execute:

```
SELECT COUNT(*) CT
FROM   NTAB
WHERE  A<>B
```

Instead, he *incorrectly deduces* that the result would be:

```
CT
 4 ← Error
```

However, if he had executed this statement, he *would see the correct* result is:

```
CT
 1
```

This user made an erroneous deduction because he incorrectly applied the 2VL in the context of null values. (Review Sample Query 11.5.) Within the 2VL system without null values, given 5 rows in any table, and knowing that "A=B" returns one row, you can correctly deduce that "A<>B" will return 4 rows. However, this conclusion is not valid in a 3VL system.

Conclusion: Practically all computer languages utilize the traditional 2VL. SQL is the only major computer language that (in the presence of null values) utilizes a 3VL.

## Truth Tables for 3VL

The 3VL becomes more complex when we consider the Boolean connectors (AND, OR, NOT). Within a 3VL system, the conventional (2VL) truth-table is enhanced to include the U (Unknown) truth value as shown below in Figure 11.4. [Suggestion: Review the truth tables for the 2VL in the Summary for Chapter 4.]

	C1	C2	C1 AND C2	C1 OR C2	NOT C1	NOT C2
	T	T	T	T	F	F
	T	F	F	T	F	T
→	T	<b>U</b>	<b>U</b>	<b>T</b>	<b>F</b>	<b>U</b>
	F	T	F	T	T	F
	F	F	F	F	T	T
→	F	<b>U</b>	<b>F</b>	<b>U</b>	<b>T</b>	<b>U</b>
→	<b>U</b>	<b>U</b>	<b>U</b>	<b>U</b>	<b>U</b>	<b>U</b>

Figure 11.4: Boolean Operators within 3VL

Figure 11.4 adds three new rows (designated by →) to the conventional truth-table for a 2VL. This figure summarizes the behavior of the Boolean operators within a 3VL system. The evaluations of True (T), False (F), and Unknown (U) are *consistent with the semantics of AND, OR, and NOT in the 2VL*. Below, we justify the truth-values for the new rows in the above figure.

**AND-Conditions:** As with the 2VL, the AND of two conditions evaluates to True if both conditions are True.

### T AND U evaluates to U

Assume the U-condition is actually True. Then, under this assumption, we have T AND T which evaluates to True.

Assume the U-condition is actually False. Then, under this assumption, we have T AND F which evaluates to False.

Hence, because these two assumptions produce different truth-values, we conclude that T AND U evaluates to U.

### F AND U evaluates to F

Assume the U-condition is actually True. Then, under this assumption, we have F AND T which evaluates to False.

Assume the U-condition is actually False. Then, under this assumption, we have F AND F which evaluates to False.

Hence, because both assumptions produce the same truth-value of False, we conclude that F AND U evaluates to False.

### U AND U evaluates to U

Assume both U-conditions are actually True. Then, under this assumption, we have T AND T which evaluates to True.

Assume both U-conditions are actually False. Then, under this assumption, we have F AND F which evaluates to False.

Hence, U OR U evaluates to U because these two assumptions do not produce the same truth-value.

**OR-Conditions:** As with the 2VL, the OR of two conditions evaluates to True if one or both conditions are True.

### T OR U evaluates to T

Assume the U-condition is actually True. Then, under this assumption, we have T OR T which evaluates to True.

Assume the U-condition is actually False. Then, under this assumption, we have T OR F which evaluates to True.

Hence, T OR U evaluates to T because both assumptions produce the same truth-value of True.

### F OR U evaluates to U

Assume the U-condition is actually True. Then, under this assumption, we have F OR T which evaluates to True.

Assume the U-condition is actually False. Then, under this assumption, we have F OR F which evaluates to False.

Hence, F OR U evaluates to U because these two assumptions produce different truth-values.

### U OR U evaluates to U

Assume both U-conditions are actually True. Then, under this assumption, we have T OR T which evaluates to True.

Assume both U-conditions are actually False. Then, under this assumption, we have F OR F which evaluates to False.

Hence, U OR U evaluates to U because these two assumptions do not produce the same truth-value.

**NOT U evaluates to U:** Assume a U-condition is actually True. Under this assumption, we have NOT T which evaluates to F. Next, assume a U-condition is actually False. Under this assumption, we have NOT F which evaluates to T. Hence, NOT U evaluates to U because both assumptions produce different truth-values.



## Compound-Conditions with 3VL

Obviously, the 3VL system requires greater attention when you verify a result table. The following sample queries invite you to verify your understanding of the Boolean operators within a 3VL.

**Sample Query 11.7a:** Display all rows from NTAB where  
A = 5 and B > 5.

A	B
5	5
5	10
5	-
-	10
-	-

NTAB

```
SELECT *
FROM NTAB
WHERE A = 5 AND B > 5
```

A	B
5	10

**Logic:**

A	B	A=5	B>5	A=5 <b>AND</b> B>5
5	5	T	F	F
<b>5</b>	<b>10</b>	<b>T</b>		<b>T ←</b>
5	-	T	U	U
-	10	U	T	U
-	-	U		U

**Sample Query 11.7b:** Display all rows from NTAB where  
A = 5 or B > 5.

```
SELECT *
FROM NTAB
WHERE A = 5 OR B > 5
```

A	B
5	5
5	10
5	-
-	10

**Logic:**

A	B	A=5	B>5	A=5 <b>OR</b> B>5
<b>5</b>	<b>5</b>	<b>T</b>	<b>F</b>	<b>T ←</b>
<b>5</b>	<b>10</b>	<b>T</b>	<b>T</b>	<b>T ←</b>
<b>5</b>	<b>-</b>	<b>T</b>	<b>U</b>	<b>T ←</b>
<b>-</b>	<b>10</b>	<b>U</b>	<b>T</b>	<b>T ←</b>
-	-	U	U	U

**Sample Query 11.8:** Display all rows from NTAB where  
A = B or A<>B.

A	B
5	5
5	10
5	-
-	10
-	-

**NTAB**

```
SELECT *
FROM NTAB
WHERE A = B
OR A <> B
```

A	B
5	5
5	10

**Logic:**

A	B	A=B	A<>5	A=B OR A<>B
5	5	T	F	T ←
5	10	F	T	T ←
5	-	U	U	U
-	10	U	U	U
-	-	U	U	U

**Theory Comment:** If you have read Appendix 4C, you may observe that the above statement conforms to Aristotle's Law of the Excluded Middle. However, this law only applies within a 2VL system. It does not apply within a 3VL system. Notice that above truth table does not contain all True values in the last column.

**Exercises:**

11G. Display the result table produced by executing:

```
SELECT *
FROM NTAB2
WHERE A <> B OR B < 20
```

11H. Display the result table produced by executing:

```
SELECT *
FROM NTAB2
WHERE A=B OR A<>B
```

## ORDER BY with Null Values

The next three sample queries present the ORDER BY, DISTINCT, and GROUP BY clauses within the context of null values. These sample queries may *appear* to be inconsistent with some concepts presented earlier in this chapter.

**Sample Query 11.9:** Reference the NTAB table.  
Display Column B in ascending sequence

NTAB	
A	B
5	5
5	10
5	-
-	10
-	-

```
SELECT B
FROM   NTAB
ORDER BY B
```

<u>DB2 &amp; ORACLE</u>	<u>SQL Server</u>
B	B
5	-
10	-
10	5
-	10
-	10

**DB2 and ORACLE:** Null values appear last, at the bottom of the output display, because they sort *higher* than any known value.

**SQL Server:** Null values will appear first, at the top of the output display, because they sort *lower* than any known value.

**Apparent Semantic Problem:** Having null values sort higher or lower than known values raises a subtle semantic issue. Earlier we emphasized that a comparison involving a null value evaluates to Unknown. However, sorting a collection of values involves comparing them, and, if a value is null, then how does the system know where to place the null value within the sequence? This apparent inconsistency is resolved by noting that a null value must go somewhere. DB2 and ORACLE place null values last, at the high end of the sequence. SQL Server places null values first, at the low end of the sequence.

11I. On your system, what result table is produced by executing the following statement?

```
SELECT B
FROM   NTAB2
ORDER BY B
```

## DISTINCT with Null Values

If we specify DISTINCT in the presence of null values, we encounter another apparent inconsistency. DISTINCT treats null values similar to known values because it does not display "duplicate" null values.

**Sample Query 11.10:** Reference the NTAB table. Display all values (including null values) in column A. Do not display duplicate values.

NTAB	
A	B
5	5
5	10
5	-
-	10
-	-

```
SELECT DISTINCT A
FROM   NTAB
```

A
5
-

**Logic:** Column A has two null values. But, only one null value appears in the result. Again, this may appear to be inconsistent with our previous observation that a null value is not equal to another null value.

We resolve this issue by noting that each null value is represented by some symbol. And, multiple null values are represented by the same symbol. When DISTINCT encounters multiple occurrences of this symbol, it treats them as duplicates. (This argument may not be satisfying, but that's the way it works.)

11J. Display the result table produced by executing:

```
SELECT DISTINCT A
FROM   NTAB2
```

## GROUP BY with Null Values

If we specify GROUP BY COLX where column COLX contains null values, we encounter another apparent inconsistency. GROUP BY treats null values similar to known values when it forms groups.

**Sample Query 11.11:** Reference the NTAB table. Group the rows by column A and display the sum of the column B values for each group.

NTAB	
A	B
5	5
5	10
5	-
-	10
-	-

```
SELECT A, SUM (B)
FROM NTAB
GROUP BY A
```

A	SUM (COLA)
5	15
-	10

**Logic:** For the purpose of grouping, SQL treats null values as equal to each other. Hence, in the current example, the null group contains two rows as illustrated below

A	B
5	5
5	10
5	-
-----	
-	10
-	-

Again, this behavior may appear to be inconsistent with previous statements that a null value is not equal to another null value.

We resolve this issue by contending that rows with null values should be stored in some group. Because all null values are represented by the same symbol, these rows should be placed in their own group. (Again, this argument may not be satisfying, but that's the way it works.)

11K. Display the result table produced by executing:

```
SELECT A, SUM(B)
FROM NTAB2
GROUP BY A
```

## Addressing the Complexity of Null Values

The preceding sample queries illustrated subtle logical issues pertaining to null values and the 3VL. These potential problems may be eliminated if every column in every table is declared to be NOT NULL as shown below in Figure 11.5.

```
CREATE TABLE NTABX
(A    INTEGER    NOT NULL,
 B    INTEGER    NOT NULL)
```

Figure 11.5: Columns specified as NOT NULL

Most front-end tools include a Metadata Panel as illustrated in Figure 1.1. This metadata includes the specification NOT NULL for columns.

There is a pretty good chance that some of your tables have one or more columns that allow null values. (A theoretical justification for representing unknown values in a database system is presented in Appendix 11A.) This author speculates that 99% of all SQL users will encounter null values in at least one of their tables. Hence, you cannot completely avoid null values.

The remaining sample queries present some SQL techniques for addressing null values. These techniques can help in many but not all circumstances.

## IS [NOT] NULL Condition

You can explicitly test a column for the presence or absence of null values. Specifying "column IS NULL" will select rows having a null value in the specified column. Specifying "column IS NOT NULL" will select rows without nulls in the specified column.

**Sample Query 11.12a:** Display all NTAB rows with a null value in column A.

A	B
5	5
5	10
5	-
-	10
-	-

NTAB

```
SELECT *
FROM NTAB
WHERE A IS NULL
```

A	B
-	10
-	-

**Syntax:** Do not specify an equal sign. The following WHERE-clause will not satisfy the query objective because a null value does not equal any value.

```
SELECT *
FROM NTAB
WHERE A = NULL      → Error
```

**Sample Query 11.12b:** Display all NTAB rows with a non-null value in column A.

```
SELECT *
FROM NTAB
WHERE A IS NOT NULL
```

A	B
5	5
5	10
5	-

**Syntax:** Do not specify a not-equal sign. You should specify "WHERE A IS NOT NULL".

The next sample query modifies Sample Query 11.3. This example shows that the accountant's cross-tabulation method returns consistent results after you eliminate null values.

**Sample Query 11.13:** Calculate the overall total of the *non-null* values found in columns A and B. Use two approaches: First, find the sum of column A, the sum of column B, and then add the results. Second, for each row, add its A and B values, and then summarize these totals.

```
SELECT SUM(A) + SUM(B), SUM(A+B)
FROM NTAB
WHERE A IS NOT NULL
AND B IS NOT NULL
```

<u>SUM(A) + SUM(B)</u>	<u>SUM(A+B)</u>
25	25

**Logic:** Because the WHERE-clause eliminated every row with a null value, only two rows were selected. Hence, the following cross-tabulation was applied.

	<u>A</u>	<u>B</u>	<u>A+B</u>
	5	5	<b>10</b>
	5	10	<b>15</b>
<b>SUM</b>	<b>10</b>	<b>15</b>	<b>25</b>

Sometimes, it is reasonable to exclude null values. However, note that the above "IS NOT NULL" conditions rejected 60% (3/5) of all NTAB rows. Therefore, the usefulness of the calculated results may be questionable. It may be better to substitute some real value for each null value, as will be illustrated in Sample Query 11.14b.



## Default Values

The DBA may be able to help users avoid some of the problems associated with null values by designating some real (non-null) "default" value to represent an unknown value. (Chapter 13 will show how this default can be specified within a CREATE TABLE statement.) Specifying a default value is a very old technique that was and still is applied within conventional 2VL systems.

For example, assume TABLEX contains a decimal column, COLX, that has some unknown values. The DBA could designate COLX as NOT NULL, and then specify some value (e.g., -1.0) as the default value to represent an unknown value. Then, to summarize all known values in the COLX, a user would execute:

```
SELECT SUM (COLX)
FROM TABLEX
WHERE COLX <> -1.0
```

Of course, this presumes that -1.0 can never appear as a real value in COLX. This method may be effective in many circumstances. However, there are circumstances where designating a default value may not be a good idea.

**Know-Your-Data Problem with Default Values:** You must remember the default value for each column. What if another decimal column, COLY, could also contain unknown values, but COLY may contain negative values, including -1.0? Hence, you cannot specify -1.0 as the default value for COLY. Some other default value must be specified, and you must remember it. You might hope that the DBA could specify some generic default value (e.g., -9999.999) that applies to all columns. However, what if COLY is an integer or a character-string column? You can see where all this is going. You must be prepared to deal with different default values for different columns.

**Good News:** NULL represents an unknown value for any data-type.

**Conclusion:** Default values might help, but they cannot help in all circumstances. This is one reason why a DBA may permit some columns to contain null values.

## COALESCE Function: Substitute a “Real” Value for a Null Value

The default-value method is “permanent” in the sense that a default value stays in effect until the DBA makes a design change. Also, this method is “global” in the sense that all SQL statements written by different users encounter the same default value.

The following method allows a column to contain null values. Then, when desired, the user can code a SQL statement with the COALESCE function to substitute a “temporary and local” value for a null value. This substituted value only applies for this statement.

**Sample Query 11.14a:** Display the NTAB table with the following substitutions. Substitute 6 for any null value in column A, and substitute 9 for any null value in column B.

A	B
5	5
5	10
5	-
-	10
-	-

**NTAB**

```
SELECT COALESCE (A, 6), COALESCE (B, 9)
FROM NTAB
```

COALESCE (A, 6)	COALESCE (B, 9)
5	5
5	10
5	9
6	10
6	9

**Syntax & Logic:** The first argument of COALESCE is a column or expression that could possibly evaluate to null. The second argument is a value to be substituted for the null value.

COALESCE (A, 6) substitutes 6 for every null value in column A.

COALESCE (B, 9) substitutes 9 for every null value in column B.

**Alternative System-Specific Functions:** Most database systems support the COALESCE function. Also, most systems also provide other system-specific functions that can be used instead of COALESCE. Without explanation, we present some examples.

With DB2, you could specify the VALUE function.

```
SELECT VALUE (A, 6), VALUE (B, 9) FROM NTAB
```

With DB2 and ORACLE, you could specify the NVL function.

```
SELECT NVL (A, 6), NVL (B, 9) FROM NTAB
```

With SQL Server, you could specify the ISNULL function.

```
SELECT ISNULL (A, 6), ISNULL (B, 9) FROM NTAB
```

**Sample Query 11.14b:** Calculate two grand totals of all values in the NTAB table using the cross-tabulation method. This time, substitute 0 for any null value in column A, and substitute 1 for any null value in column B.

```
SELECT
  SUM (COALESCE (A,0) + COALESCE (B,1))          GRANDTOTAL1,
  SUM (COALESCE (A,0))+ SUM (COALESCE (B,1))    GRANDTOTAL2
FROM NTAB
```

<u>GRANDTOTAL1</u>	<u>GRANDTOTAL2</u>
42	42

**Syntax & Logic:** Nothing new. This query assumes that the user, in consultation with the business expert, decides that 0 is a good default for an unknown value in column A; and 1 is a good default for an unknown value in column B.

## NULLS FIRST and NULLS LAST

Assume you want to override your system's placement of null values within a sorted column. For example, consider the following statement from Sample Query 11.9.

```
SELECT B
FROM NTAB
ORDER BY B
```

If your system sorts null values last, the result table will look like:

```
  B
  --
  5
 10
 10
  -
  -
```

Assume you want null values to sort first such that the result looks like:

```
  B
  --
  -
  -
  5
 10
 10
```

The following ORDER BY clause specifies the NULLS FIRST option to satisfy this objective.

```
SELECT B
FROM NTAB
ORDER BY B NULLS FIRST
```

Alternatively, if your system already sorts null values first, specifying ORDER BY B **NULLS LAST** will sort null values last.

**Important:** Not all systems support the NULLS FIRST and NULLS LAST options. Therefore, you may have to code some do-it-yourself approach to place null values in a desired first/last position within a sequence. In Chapter 22, Sample Queries 22.12a and 22.12b will demonstrate such a do-it-yourself approach.

## Summary

A null value is a symbol (indicator, mark) denoting that a value is unknown. If your SQL statement references a column with null values, you must be aware of the following:

- An arithmetic expression will produce a null value if any operand is a null value.
- Aggregate Functions ignore null values. They calculate a result using only the non-null values.
- COUNT(\*) counts selected rows. This function is not influenced by null values in the selected rows.
- COUNT (column) counts the number of non-null values in a column.
- A null value is neither equal nor unequal to another null value.
- Null values imply a three-value logic (3VL) system.
- Null values may sort higher or lower than non-null values. For example, in DB2 and ORACLE, null values sort higher; in SQL Server, null values sort lower.
- GROUP BY will place all null values in the same group.
- DISTINCT will treat all null values as duplicates.
- To test for the presence/absence of null values in column COLX, specify "COLX IS NULL" or "COLX IS NOT NULL".
- All systems support some function (e.g., COALESCE) that allows you to substitute a real value for null value within an SQL statement.

## Summary Exercises

You are given the following NTAB3 table.

A	B
20	20
50	-
-	-
-	30
10	50
10	10
40	50

**Figure 11.6: NTAB3 table.**

What is the result of executing the following statements?

1. `SELECT A, B, A*B FROM NTAB3;`
2. `SELECT MAX(A), MIN (B) FROM NTAB3;`
3. `SELECT SUM(A)+SUM(B), SUM(A+B) FROM NTAB3;`
4. `SELECT COUNT (*), COUNT(A) FROM NTAB3;`
5. `SELECT * FROM NTAB3 WHERE A = B;`
6. `SELECT * FROM NTAB3 WHERE A <> B;`
7. `SELECT COUNT (*) FROM NTAB3 WHERE A = B OR A <> B;`
8. `SELECT * FROM NTAB3 WHERE A <> 10 AND B < 10`
9. `SELECT * FROM NTAB3 WHERE A = 10 OR B < 10`
10. `SELECT * FROM NTAB3 ORDER BY A;`
11. `SELECT DISTINCT A FROM NTAB3;`
12. `SELECT A, SUM(B) FROM NTAB3 GROUP BY A;`
13. `SELECT * FROM NTAB3 WHERE A IS NULL;`
14. `SELECT * FROM NTAB3 WHERE A IS NOT NULL;`
15. `SELECT SUM(A)+SUM(B), SUM(A+B) FROM NTAB3  
WHERE A IS NOT NULL AND B IS NOT NULL;`
16. `SELECT COALESCE (A,25), COALESCE (B,15) FROM NTAB3;`

## Appendix 11A: Theory

The database literature contains many publications that debate the pros and cons of null values. This literature also proposes alternative methods for representing and processing unknown values. Some of these methods may be considered to be slightly esoteric. For example, Codd also proposed a 4VL system.

**Philosophical justification for representing unknown values in a database:** *Ideally, (1) a database design should accurately represent its application domain. (2) Many application domains have unknown data. Therefore, (3) a database design should utilize some method to represent unknown data.*

For example, within a medical or health insurance application, a person who is adopted might not be able to provide a simple yes/no answer to a question like: Did your biological father ever have cancer? If this person answers "I don't know," any default yes/no value would not be accurate. In this circumstance, data accuracy requires the database design to "somehow" represent unknown data.

**Laws of Logic:** Appendix 4B described two tautologies, the Law of the Excluded Middle and the Law of Non-Contradiction. *These laws do not apply within a 3VL system.*

The Law of the Excluded Middle,  $C \text{ OR } (\text{NOT } C)$ , is not a tautology. The following truth table shows this expression does not contain all T-values.

C	NOT C	C OR (NOT C)
T	F	T
F	T	T
U	U	U

The Law of Non-Contradiction,  $\text{NOT } (C \text{ AND } (\text{NOT } C))$  is not a tautology. The following truth table shows this expression does not contain all T-values.

C	NOT C	C AND (NOT C)	NOT (C AND (NOT C))
T	F	F	T
F	T	F	T
U	U	U	U

**Conclusion:** What is the best way to deal with unknown values within a database system? This is an interesting and debatable question.

# PART III

## Data Definition & Data Manipulation

This part of the book consists of four chapters that introduce some of SQL's data definition and data manipulation statements. Some of these chapters are optional. Below we outline the topics presented in each chapter and offer some advice to help you determine which of these chapters you should read.

**Terminology:** The **Data Definition Language (DDL)** refers to SQL statements that create and drop database objects. DDL statements presented in this part of the book include the CREATE TABLE, DROP TABLE, CREATE INDEX, and DROP INDEX statements.

**Terminology:** The **Data Manipulation Language (DML)** refers to SQL statements that modify data in a database table. DML statements presented in this part of the book include the INSERT, UPDATE, and DELETE statements.

**\*\*\* Chapter 12:** *This is an important chapter, and all readers should read it.* This chapter presents five sample sessions that introduce SQL's basic data definition and data manipulation statements. Do not focus on details (which will be presented in the following three chapters). Reading this chapter will help you understand some very important *know-your-data concepts* associated with more complex sample queries to be presented later in this book.

*If you are only interested in executing SELECT statements, after reading Chapter 12, you can bypass the remainder of this Part III and proceed to Part IV.*



**Chapter 13:** This chapter presents a discussion of the CREATE TABLE statement that was previewed in Chapter 0 (Figure 0.1). Application developers who want to create tables within a test database should read this chapter. All other users can skip this chapter.

**Chapter 14:** This chapter introduces the CREATE INDEX and DROP INDEX statements. This chapter is optional for all readers. However, if you have read the preceding efficiency appendices, you have already learned some basic concepts about database indexes. This chapter illustrates that coding CREATE INDEX and DROP INDEX statements is relatively straightforward.

**Chapter 15:** This chapter introduces SQL's three major data manipulation statements.

- The INSERT statement is used to store new row(s) into a table.
- The UPDATE statement is used to modify existing row(s) in a table.
- The DELETE statement is used to remove existing row(s) from a table.

Application developers should read this chapter because they will probably embed INSERT, UPDATE, and DELETE statements within stored procedures and application programs. All other users can skip this chapter.

# 12

## **Read This Chapter!**

### **Preview Sample Sessions:**

*This is an important chapter.* It presents five sample sessions that preview the CREATE TABLE, DROP TABLE, INSERT, UPDATE, and DELETE statements.

Focus on concepts. Do not worry about syntactical details that will be presented in the following chapters. Commentary will focus on important know-your-data concepts that will become very relevant when we introduce join operations in the following Part IV of this book.

## Sample-Session-1: Getting Started

**Objective:** Introduce the basic syntax and behavior of the CREATE TABLE, DROP TABLE, INSERT, UPDATE, and DELETE statements.

You should: (i) scan the following sample session, (ii) read the commentary about each statement on the following pages, and then (iii) execute each statement. This sample session shows "SELECT \* FROM DOG" after each INSERT, UPDATE, and DELETE statement so that you can observe the changes produced by each statement.

```
DROP TABLE DOG;

CREATE TABLE DOG
  (DNO      INTEGER      NOT NULL,
   DNAME    CHAR (10)   NOT NULL);

SELECT * FROM DOG;

INSERT INTO DOG VALUES (1000, 'SPOT');

SELECT * FROM DOG;

INSERT INTO DOG VALUES (3000, 'ROVER');

SELECT * FROM DOG;

INSERT INTO DOG VALUES (2000, 'WALLY');

SELECT * FROM DOG;

UPDATE DOG
SET DNAME = 'SPIKE'
WHERE DNO = 3000;

SELECT * FROM DOG;

UPDATE DOG
SET DNAME = 'REX'
WHERE DNAME LIKE 'S%';

SELECT * FROM DOG;

DELETE FROM DOG
WHERE DNAME = 'REX';

SELECT * FROM DOG;

DROP TABLE DOG;
```

## Commentary on Sample-Session-1 Statements

DROP TABLE DOG;

You should see an error message indicating that you attempted to drop a table that does not exist.

```
CREATE TABLE DOG
  (DNO INTEGER,
   DNAME CHAR (10));
```

You should see a message indicating creation of the DOG table. (Observe that no column is designated as UNIQUE.)

```
SELECT * FROM DOG;
```

Observe that the new DOG table is empty.

<u>DOG</u>	
<u>DNO</u>	<u>DNAME</u>

```
INSERT INTO DOG VALUES (1000, 'SPOT');
```

Observe a message indicating a successful insert.

```
SELECT * FROM DOG;
```

Verify the preceding insert operation.

<u>DOG</u>		
<u>DNO</u>	<u>DNAME</u>	
1000	SPOT	←

```
INSERT INTO DOG VALUES (3000, 'ROVER');
```

Observe a message indicating a successful insert.

```
SELECT * FROM DOG;
```

Verify the preceding insert operation.

<u>DOG</u>		
<u>DNO</u>	<u>DNAME</u>	
1000	SPOT	
3000	ROVER	←

```
INSERT INTO DOG VALUES (2000, 'WALLY');
```

Observe a message indicating a successful insert.

```
SELECT * FROM DOG;
```

Verify the preceding insert operation.

DOG	
<u>DNO</u>	<u>DNAME</u>
1000	SPOT
3000	ROVER
2000	WALLY

←

```
UPDATE DOG  
SET DNAME = 'SPIKE'  
WHERE DNO = 3000;
```

Observe a message indicating a successful update operation. This statement indicates that you want to change the DNAME value of Dog 3000 to SPIKE.

```
SELECT * FROM DOG;
```

Verify the preceding update of one row.

DOG	
<u>DNO</u>	<u>DNAME</u>
1000	SPOT
3000	<b>SPIKE</b>
2000	WALLY

←

```
UPDATE DOG
SET DNAME = 'REX'
WHERE DNAME LIKE 'S%';
```

Observe a message indicating a successful UPDATE operation. The two rows describing SPIKE and SPOT have their DNAME values changed to REX.

```
SELECT * FROM DOG;
```

Verify the preceding update of two rows.

<u>DOG</u>	
<u>DNO</u>	<u>DNAME</u>
1000	<b>REX</b> ←
3000	<b>REX</b> ←
2000	WALLY

```
DELETE FROM DOG
WHERE DNAME = 'REX';
```

Observe a message indicating a successful delete operation. This DELETE operation applies to the two REX rows.

```
SELECT * FROM DOG;
```

Verify the preceding deletion of two rows.

<u>DOG</u>	
<u>DNO</u>	<u>DNAME</u>
2000	WALLY

```
DROP TABLE DOG;
```

All rows are automatically deleted before the DOG table is dropped from the database.

## Sample-Session-2: Table with Duplicate Rows

**Objective:** Illustrate the insertion of duplicate rows into a table, and describe why duplicate rows are problematic.

```
DROP TABLE MAN;

CREATE TABLE MAN
(MNO    INTEGER    NOT NULL,
 MNAME CHAR (10)  NOT NULL);

INSERT INTO MAN VALUES (77, 'MOE');
INSERT INTO MAN VALUES (99, 'CURLY');
INSERT INTO MAN VALUES (88, 'LARRY');
INSERT INTO MAN VALUES (99, 'CURLY'); ← Inserts problematic row

SELECT * FROM MAN;
```

### **Commentary:**

The DROP TABLE statement drops any pre-existing MAN table.

The CREATE TABLE statement creates the MAN table. The basic syntax for the CREATE TABLE statement was introduced in Figure 0.2 which created the PRESERVE table. Unlike the CREATE TABLE for PRESERVE, *the above CREATE TABLE statement does not designate any column as UNIQUE.*

The first three INSERT statements store three rows into the MAN table. There is nothing problematic about these rows. Observe that these three rows are distinct.

***The fourth INSERT statement is valid, but problematic.*** This statement executes without an error. However, we consider this row to be “garbage” because it is an exact duplicate of another row. The following Sample Session-3 demonstrates how to the system can automatically prevent this kind of “garbage insert” operation.

At the end of Sample Session-2, the MAN table looks like:

<u>MAN</u>	
<u>MNO</u>	<u>MNAME</u>
77	MOE
99	CURLY
88	LARRY
99	CURLY

**Know-Your-Data Observation:** Upon scanning this table you might say: "Hey - this table has some duplicate rows." This observation is very important. It should prompt you to contact the business expert to discuss your observation.

You: The MAN table has duplicate rows. Should this duplication be expected?

Business Expert: This is bad! I will ask the DBA to prevent this kind of problem.

**What's wrong with duplicate rows?** We offer a practical answer and a theoretical answer.

Practical Answer: A result table with duplicate rows can be confusing. Also, query objectives can become fuzzy. For example, assume your query objective is: How many men are represented in the MAN table? If duplicate rows are allowed, this query objective is not equivalent to: How many rows are in the MAN table?

Theoretical Answer: C.J. Date quotes Ted Codd: "If you say something is true, saying it twice doesn't make it any truer!" Other theoretical issues associated with duplicate rows were in presented Appendix 3B.



## Sample-Session-3: PRIMARY KEY Clause

**Objective:** Illustrate how the PRIMARY KEY clause prohibits the insertion of duplicate values into a column. Hence, duplicate rows cannot appear in a table.

**PRIMARY KEY versus UNIQUE:** Figure 0.2, which illustrated the CREATE TABLE statement for the PRESERVE table, specified the PNO column as UNIQUE. This differs from the following CREATE TABLE statement for the MAN table where the MNO column is specified as the PRIMARY KEY.

*\*\*\* Both the PRIMARY KEY clause and the UNIQUE clause prohibit duplicate values. However, declaring a column as the PRIMARY KEY (versus UNIQUE) provides additional advantages that will be described on the following page.*

```
DROP TABLE MAN;

CREATE TABLE MAN
(MNO  INTEGER  NOT NULL PRIMARY KEY,
 MNAME CHAR (10) NOT NULL);

INSERT INTO MAN VALUES (77, 'MOE');
INSERT INTO MAN VALUES (99, 'CURLY');
INSERT INTO MAN VALUES (88, 'LARRY');
INSERT INTO MAN VALUES (55, 'CURLY');

SELECT * FROM MAN;

INSERT INTO MAN VALUES (99, 'SHEMP');
```

← ERROR - duplicate MNO

The DROP TABLE, CREATE TABLE, and first four INSERT statements execute successfully.

The last INSERT fails because the MNO is the primary-key and this column already contains an MNO value of 99. *The system will automatically trap this error, prevent this insert operation, and generate an error message.* At the end of Sample Session-3, the MAN table looks like:

<u>MAN</u>	
<u>MNO</u>	<u>MNAME</u>
77	MOE
99	CURLY
88	LARRY
55	CURLY

## PRIMARY KEY Advantages

Similar to a UNIQUE column:

- A primary key column is used to uniquely identify a row in a table.
- Specifying a column as the primary-key column helps the system automatically prevent duplicate rows.
- Future sample queries will show that knowing a column is a primary-key will facilitate query analysis.

Unlike UNIQUE:

- Sample Session-5 will introduce another very important database concept, the "Foreign-Key." We will see that a foreign-key references a primary-key. Hence, a primary-key must be defined before a foreign-key can reference it.

## Column-Constraint versus Table-Constraint

Column-Constraint: The following CREATE TABLE statement specifies the PRIMARY KEY clause as a column-constraint. Here, this clause is specified in the definition of the MNO column.

```
CREATE TABLE MAN
(MNO INTEGER NOT NULL PRIMARY KEY,
 MNAME CHAR (10) NOT NULL)
```

Table-Constraint: Alternatively, the above PRIMARY KEY constraint could have been specified as a table-constraint that is specified after all columns have been defined.

```
CREATE TABLE MAN
(MNO INTEGER NOT NULL,
 MNAME CHAR (10) NOT NULL,
 PRIMARY KEY (MNO))
```

Sometimes, as illustrated by this example, you can optionally specify either a column-constraint or a table-constraint. Chapter 13 will describe circumstances where you will not have this option.

## Sample-Session-4A: "Related" Tables

**Objective:** Introduce a database design with two tables that are related to each other.

This session, using the same techniques shown in the preceding sessions, creates and populates two tables, MAN and DOG. Notice that the DOG table has a column (MNO) that contains the man number of the man who owns the dog. This column represents a **relationship** between the MAN and DOG tables. A casual description this relationship is: "Man owns dog." More precisely, this relationship means that:

- Each dog must be owned by exactly one man.
- A man can own any number of dogs (including no dogs).

```
DROP TABLE MAN;
DROP TABLE DOG;

CREATE TABLE MAN
(MNO    INTEGER    NOT NULL PRIMARY KEY,
 MNAME  CHAR (10)  NOT NULL);

INSERT INTO MAN VALUES (77, 'MOE');
INSERT INTO MAN VALUES (99, 'CURLY');
INSERT INTO MAN VALUES (88, 'LARRY');
INSERT INTO MAN VALUES (55, 'CURLY');

CREATE TABLE DOG
(DNO    INTEGER    NOT NULL PRIMARY KEY,
 DNAME  CHAR (10)  NOT NULL,
 MNO    INTEGER    NOT NULL);

INSERT INTO DOG VALUES (1000, 'SPOT', 99);
INSERT INTO DOG VALUES (3000, 'ROVER', 77);
INSERT INTO DOG VALUES (2000, 'WALLY', 99);

SELECT * FROM MAN;
SELECT * FROM DOG;
```

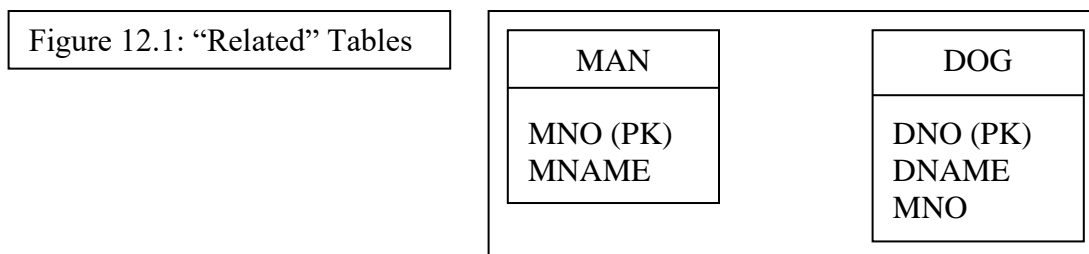
After this sample session, the MAN and DOG tables contain the following values.

<u>MAN</u>		<u>DOG</u>		
<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<b><u>MNO</u></b>
77	MOE	1000	SPOT	<b>99</b>
99	CURLY	3000	ROVER	<b>77</b>
88	LARRY	2000	WALLY	<b>99</b>
55	CURLY			

Observations: By scanning the MAN and DOG tables we learn details about this "man owns dog" relationship.

- Dog 1000 is owned by Man 99.
- Dog 2000 is owned by Man 99.
- Dog 3000 is owned by Man 77.
- Man 77 owns one dog, Dog 3000.
- Man 99 owns two dogs, Dog 1000 and Dog 2000.
- Two men, Man 88 and Man 55, do not own any dogs.

**Data Model:** When an application design includes multiple tables (the usual situation), the database designer usually produces a high-level graphical model called a *data model*. An example is shown below in Figure 12.1.



This data model identifies table-names, column-names, and primary key (PK) columns. It excludes some detail information such as column data-types and specification of NOT NULL clauses.

**Important:** Figure 12.1 has a label showing "Related" in quotes. These quotes are meant to imply that *these tables are not related in the sense that the database system knows about this "man-owns-dog" relationship.* You know about this relationship, but the *system does not*. Therefore, the system would allow someone to code an INSERT statement that violates this relationship. The following Sample Session-4B illustrates an example.

## Sample-Session-4B: Violation of Referential Integrity

**Objective:** Introduce the concept of "Referential Integrity" and illustrate a violation of this concept.

This session creates and populates the same MAN and DOG tables shown in the previous Sample Session-4A. It inserts the same four rows into the MAN table and the same three rows into the DOG table. Then, it inserts one more row, a problematic row, into the DOG table.

We have already noted that each dog must be owned by exactly one man. To be more precise, each dog must be owned by a man who is described in the MAN table. Now, to be very precise, every MNO value in the DOG table must match some MNO value in the MAN table. The following sample session illustrates a violation this rule.

```
DROP TABLE MAN;
DROP TABLE DOG;

CREATE TABLE MAN
(MNO    INTEGER    NOT NULL PRIMARY KEY,
 MNAME  CHAR (10)  NOT NULL);

INSERT INTO MAN VALUES (77, 'MOE');
INSERT INTO MAN VALUES (99, 'CURLY');
INSERT INTO MAN VALUES (88, 'LARRY');
INSERT INTO MAN VALUES (55, 'CURLY');

CREATE TABLE DOG
(DNO    INTEGER    NOT NULL PRIMARY KEY,
 DNAME  CHAR (10)  NOT NULL,
 MNO    INTEGER    NOT NULL);

INSERT INTO DOG VALUES (1000, 'SPOT', 99);
INSERT INTO DOG VALUES (3000, 'ROVER', 77);
INSERT INTO DOG VALUES (2000, 'WALLY', 99);

INSERT INTO DOG VALUES (4000, 'SPIKE', 11);

SELECT * FROM MAN;
SELECT * FROM DOG;
```

← **Problematic INSERT**

Consider the last INSERT statement that inserts an MNO value of 11 into the DOG table. This is problematic because there is no man with an MNO value of 11 in the MAN table. Hence, we consider this MNO value (11) in the DOG table to be garbage.

After executing these statements, the MAN and DOG tables contain the following values.

<u>MAN</u>		<u>DOG</u>		
<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO</u>
77	MOE	1000	SPOT	99
88	LARRY	3000	ROVER	77
99	CURLY	2000	WALLY	99
		4000	SPIKE	<b>11 ← problematic</b>

The above problematic row violates the notion of "Referential Integrity."

**Referential Integrity** is a database design concept. In this example, it means that each dog should refer to a "real" man; a dog should not refer to a "fictitious" man (i.e., any man whose MNO value is not 77, 88, or 99).

This description of referential integrity is at a high-level (real-world level) in terms of men and dogs. It does not refer to lower-level SQL terms such as tables, columns, and keys. Sample Session-5 will introduce a SQL feature called a "**Foreign Key**" that tells the system to automatically enforce this higher-level notion of referential integrity.

The following Sample Session-5 will specify a FOREIGN KEY clause in the DOG table. This clause effectively tells the system about the man-owns-dog relationship. When the system knows about this relationship (as you do), it will automatically prohibit execution of the problematic INSERT statement.

## Sample-Session-5: Foreign Key Enforces Referential Integrity

**Objective:** Illustrate how the FOREIGN KEY clause enforces referential integrity.

This session re-creates the same MAN and DOG tables shown Sample Sessions 4A and 4B with one very significant enhancement. The CREATE TABLE statement for the DOG table includes the following clause.

### **FOREIGN KEY (MNO) REFERENCES MAN**

This clause declares the MNO column in the DOG table is a FOREIGN KEY that references the primary key column (MNO) in the MAN table. This means that every MNO value in the DOG table must appear somewhere in the primary-key column of the MAN table. Given this constraint, the system will automatically reject the last INSERT statement.

```
DROP TABLE MAN;
DROP TABLE DOG;

CREATE TABLE MAN
(MNO INTEGER NOT NULL PRIMARY KEY,
 MNAME CHAR (10) NOT NULL);

INSERT INTO MAN VALUES (77, 'MOE');
INSERT INTO MAN VALUES (88, 'LARRY');
INSERT INTO MAN VALUES (99, 'CURLY');

CREATE TABLE DOG
(DNO INTEGER NOT NULL PRIMARY KEY,
 DNAME CHAR (10) NOT NULL,
 MNO INTEGER NOT NULL,
 FOREIGN KEY (MNO) REFERENCES MAN);

INSERT INTO DOG VALUES (1000, 'SPOT', 99);
INSERT INTO DOG VALUES (3000, 'ROVER', 77);
INSERT INTO DOG VALUES (2000, 'WALLY', 99);
INSERT INTO DOG VALUES (4000, 'SPIKE', 11);

SELECT * FROM MAN;
SELECT * FROM DOG;
```

← **Error:** Non-matching  
Foreign key

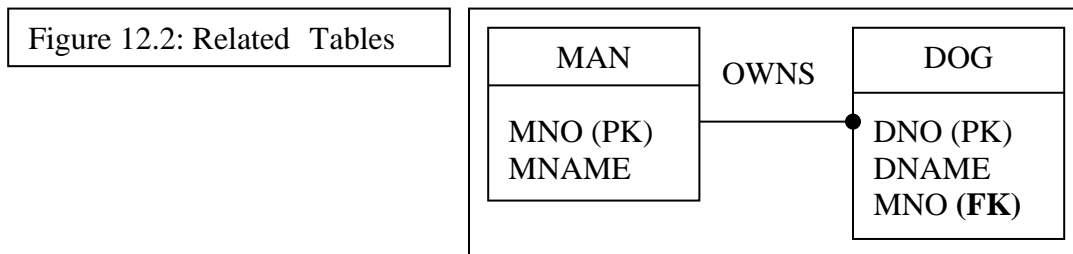
The last INSERT statement is *automatically rejected by system* because it *attempts* to insert "garbage" (11) into the MNO column of DOG.

The MAN and DOG tables now contain the following values.

MAN		DOG		
MNO	MNAME	DNO	DNAME	MNO
77	MOE	1000	SPOT	99
88	LARRY	3000	ROVER	77
99	CURLY	2000	WALLY	99

Observe there is no violation of referential integrity.

**Database Relationship:** A FOREIGN KEY clause effectively implements a database relationship. In this example, the FOREIGN KEY clause defines a one-to-many relationship between the MAN and DOG tables. The following Figure 12.2 illustrates this relationship by enhancing the data model shown in Figure 12.1.



**Data Model:** To represent the man-owns-dog relationship, the above Figure 12.2 includes a **line (labeled OWNS)** between the MAN and DOG rectangles. This figure also designates the MNO column in the DOG table as a foreign key (**FK**).

A FOREIGN KEY (FK) clause defines a **one-to-many** relationship. In this example, the OWNS relationship means that: (i) each man can own many dogs; and, (ii) each dog must have exactly one owner. The solid circle in the —● symbol is used to designate that DOG is the “many-side” of the one-to-many relationship.

**Terminology - “Parent” and “Child”:** When two tables are related via a one-to-many relationship, the “one-table” is called the **parent**, and the “many-table” is called the **child**. In Figure 12.2, MAN is the parent of DOG. DOG is the child of MAN. A parent can have any number of children (including no children), but each child must have exactly one parent.



## Preview: Join Operations & Foreign Keys

Examine the following query objective and its result table.

Objective: Only consider men who own dogs. Display all information about these men and their dogs.

MNO	MNAME	DNO	DNAME	MNO
99	CURLY	1000	SPOT	99
77	MOE	3000	ROVER	77
99	CURLY	2000	WALLY	99

*We do not show the SQL code that produces this result table. We only make some important observations.*

- This result table should be intuitively obvious. Each row shows information about a man and his dog. For example, the first row shows that CURLY owns SPOT. Any man (e.g., Man 88) who does not own a dog does not appear in the result.
- This result displays columns from both the MAN and DOG tables. To produce this result, the SELECT statement must reference both tables. Specifically, the SELECT statement will contain code that **joins** these tables. Part IV of this book will present a detail description of join-operations.
- It should be clear that joining two tables involves “matching” some column value in the first table against some column value in the second table. The above result shows rows where MNO values in the MAN table matched MNO values in the DOG table. For example, CURLY’s MNO value of 99 matched SPOT’s MNO value of 99. *Only rows with matching MNO values appear in the result table.*
- **This matching operation was based upon a primary-key/foreign-key relationship.** Here, the MNO column (primary-key column) of MAN was compared to the MNO column (foreign-key column) of DOG. In real-world queries, most join-operations involve comparing primary-key values with foreign-key values.

## Dot-Notation: Qualifying Column-Names

The MAN and DOG tables have a column with the same name (MNO). If a SELECT statement references two tables, and both tables have a column with the same name, the *dot-notation* is used to distinguish each column. For example:

MAN.MNO references the MNO column in the MAN table.

DOG.MNO references the MNO column in the DOG table.

The dot-notation can be utilized in a SELECT statement that references a single table. For example, the following two statements are equivalent.

<u>Statement-1</u>	<u>Statement-2</u>
<pre>SELECT PNO,        PNAME,        ACRES FROM PRESERVE</pre>	<pre>SELECT PRESERVE.PNO,        PRESERVE.PNAME,        PRESERVE.ACRES FROM PRESERVE</pre>

Statement-2 uses the dot-notation where each column-name is preceded by the name of the table that contains the column. In this statement, we have "qualified" each column-name with its table-name.

Clearly, the unqualified column-names in Statement-1 are much simpler. Hence, the dot-notation is rarely used for single-table queries.

## Philosophical Question: What's in a Database?

We offer two answers to this question. The first answer is correct but incomplete. This second answer is profound; it distinguishes a database system from a conventional file system.

Answer-1: **A database is a collection of data.** This answer might be satisfactory if we restrict our attention to Chapters 1-11 where we executed SELECT statements against a single table.

Answer-2: *A database is a collection of data **plus relationships between the data**.* Within a relational database, foreign-keys define relationships.

## Summary

This chapter previewed the CREATE TABLE and DROP TABLE statements and the three most popular DML statements (INSERT, UPDATE, and DELETE). Users were encouraged to preview these statements even though they may never execute such statements. Most application developers will execute DML statements. Therefore, Chapter 15 will offer more details about the DML statements previewed in this chapter.

We have seen that the PRIMARY KEY and FOREIGN KEY clauses prevent some kinds of garbage from entering a table. However, most importantly, from a query perspective, knowledge about primary-keys and foreign-keys will be very helpful when you begin to code SELECT statements that specify join-operations.

## CREATE TABLE Statement

**Introduction:** Again, most users do not need to read this chapter. However, this chapter should be very useful for application developers who wish to create tables in a testing environment.

CREATE TABLE statements were previewed in Chapter 0 (Figure 0.2), and described in sample sessions in the preceding Chapter 12. The primary objective of this chapter is to offer more details about the CREATE TABLE statement.

**Database Analysis/Design:** Database administrators do not arbitrarily create tables. They are given design specifications produced by database analysts/designers. These specifications designate the:

- tables to be created
- columns in each table
- data-type for each column
- database integrity constraints (e.g., primary keys)

Coding CREATE TABLE statements becomes a relatively straightforward task *if* you are given a correct design specification. The real challenge pertains to following some methodology of database analysis and design to produce this design specification. Appendixes 13A and 13B present some basic concepts pertaining to database analysis and design.

## CREATE TABLE Statement

The following Figure 13.1 outlines the general syntax for the CREATE TABLE statement.

```
CREATE TABLE table-name

(Column1-name data-type [column-constraint(s)],
 Column2-name data-type [column-constraint(s)],
 . . . ,
[table-constraint(s)] );
```

Figure 13.1: General Syntax for CREATE TABLE Statement

The following Figure 13.2 illustrates two CREATE TABLE statements that create the TESTDEPT and TESTEMP tables. (Two DROP TABLE statements are initially executed in case these tables already exist.) Take a close look at the column definitions in each statement. Your intuition should give you some idea about the purpose of each clause which is described on the following pages.

```
DROP TABLE TESTEMP;
DROP TABLE TESTDEPT;

CREATE TABLE TESTDEPT
(DNO      INTEGER      NOT NULL PRIMARY KEY,
 DNAME    VARCHAR(20)  NOT NULL UNIQUE,
 BLD      CHAR(3)      NOT NULL DEFAULT 'AB1',
 BUDGET   DECIMAL(9,2) NOT NULL DEFAULT 0.0 );

CREATE TABLE TESTEMP
(ENO      CHAR(3)      NOT NULL,
 ENAME    VARCHAR(25)  NOT NULL,
 JCODE    INTEGER      NOT NULL CHECK (JCODE IN (1,3,7,11)),
 SALARY   DECIMAL(7,2) NOT NULL DEFAULT 0.0,
 OTMAX    DECIMAL(5,2) NOT NULL DEFAULT 0.0,
 DNO      INTEGER      NOT NULL,
 PRIMARY KEY (ENO),
 FOREIGN KEY (DNO) REFERENCES TESTDEPT,
 CHECK (OTMAX <= .15 * SALARY));
```

Figure 13.2: CREATE TABLE Statements

**Data-Types:** The data-types of the columns in these tables are the familiar INTEGER, DECIMAL, CHAR and VARCHAR types. The following Section-A will introduce other data-types.

**Constraints:** We have already introduced four major database constraints (PRIMARY KEY, UNIQUE, FOREIGN KEY, and NOT NULL). Examination of Figure 13.2 shows the specification of two other database constraints, CHECK and DEFAULT. The following Section-B will discuss these constraints, and Section-C will say more about the FOREIGN KEY constraint.

**Column-Constraints versus Table-Constraints:** Figure 13.1 shows that a *column-constraint* is specified for a specific column, and a *table-constraint* is specified after all columns have been defined. The CREATE TABLE statement for TESTDEPT (Figure 13.2) shows all column-constraints, whereas the CREATE TABLE statement for TESTEMP shows both column-constraints and table-constraints. We also note that:

- Some constraints can be specified as either a column-constraint or a table-constraint. For example, within the CREATE TABLE statement for TESTDEPT, the PRIMARY KEY clause is specified as a column-constraint; whereas, within the CREATE TABLE statement for TESTEMP, the PRIMARY KEY clause is specified as a table-constraint. This is possible because both primary keys are atomic (single-column) keys.
- Some constraints, such as the NOT NULL constraint, must be specified as a column-constraint.
- Some constraints must be defined as a table-constraint. For example, the CHECK constraint for the TESTEMP table cannot be a column-constraint because it references multiple columns (OTMAX and SALARY).

**Sequence for Executing CREATE TABLE Statements:** In Figure 13.2, the TESTDEPT table was created before the TESTEMP table. This was necessary because the foreign-key in TESTEMP references the primary-key in TESTDEPT, *which presumably has already been created*. If we had created the TESTEMP first, we would get an error because its FOREIGN KEY clause would reference a non-existing table. (However, Figure 13.6 will illustrate a method that allows us to initially create the TESTEMP table.)

## A. Data-Types

All systems support a small number of standard data-types, including those we have already encountered: `INTEGER`, `DECIMAL`, `CHAR`, and `VARCHAR`. All systems also support the `SMALLINT` data-type to be described below. These five data-types are sufficient for most business applications.

All systems also support many vendor-specific data-types, such as date-time data-types. Furthermore, modern systems have evolved to include more specialized object-oriented and XML data-types. (Discussion of these data-types is beyond the scope of this book.) For the sake of illustration, we outline many of the data-types supported by DB2.

### DB2 Numeric Data-Types

**SMALLINT** two-byte integer with a precision of 5 digits  
Range: -32,768 to 32,767

**INTEGER** four-byte integer with a precision of 10 digits  
Range: -2,147,483,648 to +2,147,483,647

**BIGINT** eight-byte integer with a precision of 19 digits  
Range: -9,223,372,036,854,775,808 to  
+9,223,372,036,854,775,807

**DECIMAL (p, s)**

Maximum precision is 31 digits  
Scale (number of digits in the fractional part) cannot be negative or greater than the precision.  
Range:  $-10^{31}+1$  to  $10^{31}-1$

**REAL** *single-precision floating-point* number  
32-bit approximation of a real number  
Range: zero, or -3.402E+38 to -1.175E-37,  
or 1.175E-37 to 3.402E+38

**DOUBLE [or FLOAT]** *double-precision floating-point* number  
64-bit approximation of a real number  
Range: zero, or -1.79769E+308 to -2.225E-307,  
or 2.225E-307 to 1.79769E+308

## DB2 Character-String Data-Types

**CHAR (n)** Fixed-length (n: 1-254 bytes)

**VARCHAR (n)** Variable-length (n: 1-32,672 bytes)

**CLOB (n)** Character Large Object (Max = two gigabytes - 1)  
Special restrictions apply to CLOB. For example, CLOB columns cannot be referenced in:

- SELECT-clauses that specify DISTINCT
- GROUP BY, ORDER BY, BETWEEN, and IN clauses
- Character-string patterns in LIKE-clauses

## DB2 Temporal Data-Types

**DATE** Three-parts (year, month, and day).

Range of year is 0001-9999.

Range of month is 1-12.

Range of day is 1-*m*, where *m* depends on the month.

**TIME** Three-parts (hour, minute, and second) 24-hour clock

Range of the hour is 0-24. If hour=24, minute and second specifications are zero

Range of the minute is 0-59.

Range of the second is 0-59.

**TIMESTAMP** Seven-parts (year, month, day, hour, minute, second, and microsecond)

Date and time as defined above, except time includes a fractional specification of microseconds.

## Some other DB2 Data-Types include

- BLOB and DBCLOB
- GRAPHIC (n), VARGRAPHIC (n) and LONG VARGRAPHIC (n)
- DATALINK
- XML



## B. Database Constraints

The basic idea behind database integrity is to have the system *automatically* reject any statement that attempts to store garbage in a table. To realize this objective, the DBA must define what constitutes valid data. Then the system can reject any operation that violates this definition.

Characteristics of valid data are defined by *integrity constraints*. You have already encountered four kinds of integrity constraints. These are the PRIMARY KEY, UNIQUE, FOREIGN KEY, and NOT NULL constraints. This chapter offers more details about these constraints and introduces two other kinds of constraints (DEFAULT and CHECK). The following discussion refers to the integrity constraints illustrated in Figure 13.2.

**PRIMARY KEY:** A table can only have one primary-key. For example, DNO is declared to be the primary-key of the TESTDEPT table. This means that the system will not allow duplicate values in the DNO column. Also, PRIMARY KEY column(s) must be declared as NOT NULL.

*Composite Primary-Keys:* Sometimes, no individual column will contain unique values. For example, every individual column in the PURCHASE table (Figure 9.2) contains duplicate values. However, within this table, every combination of (PNO, SNO, ENO, PJNO, PURDAY) values is unique. Hence, your CREATE-ALL-TABLES script shows a *composite* primary-key specified as:

```
PRIMARY KEY (PNO, SNO, ENO, PJNO, PURDAY)
```

Because this PRIMARY KEY clause references multiple columns, it must be specified as a table-constraint.

**FOREIGN KEY:** Future examples will show that, if a foreign-key references a composite primary-key, then the foreign-key will also be composite.

Also, note that the foreign-key column (DNO) in TESTEMP has the same name as the referenced primary-key column in TESTDEPT. This naming pattern is common, but it is not mandatory.

**NOT NULL:** Chapter 11 introduced the NOT NULL clause and described potential problems associated with null values. Note that all columns in TESTDEPT and TESTEMP tables are declared to be NOT NULL.

**UNIQUE:** DNAME is declared to be a UNIQUE column. As with the DNO column, the system will prohibit the storage of duplicate DNAME values. However, there are two important differences between the PRIMARY KEY versus UNIQUE clauses.

1. A table can only specify one PRIMARY KEY clause. However, multiple UNIQUE clauses may be specified.
2. A FOREIGN KEY clause almost always references a PRIMARY KEY. (Many systems allow a FOREIGN KEY clause to reference a UNIQUE column. However, most practitioners will only do so in special case design circumstances.)

**DEFAULT:** Assume some column is defined as NOT NULL. What happens when you want to execute an INSERT statement and you do not know this column's value? In this circumstance, it may make sense to specify some non-null *default* value that is stored whenever an actual value is unknown. This is the purpose of the DEFAULT clause specified for the BLD and BUDGET columns in the TESTDEPT table and the SALARY and OTMAX columns in the TESTEMP table. If an INSERT statement does not specify a value for any of these columns, the system automatically stores the specified default value in the column.

**CHECK:** The CHECK clause allows you to identify all valid values for a column. Consider the TESTEMP table. The JOBCODE column can only contain 1, 3, 7, or 11. Users should know about this constraint. Otherwise, they might code the following unreasonable WHERE-clause which will return an error message or "no hit" message.

```
SELECT * FROM TESTEMP WHERE JOBCODE = 5
```

A CHECK-clause can specify other SQL conditions, such as:

```
CHECK column-name BETWEEN ____ AND ____
```

Finally, a CHECK clause can reference multiple columns from the same table. For example, CHECK (OTMAX <= .15 \* SALARY) ensures that an employee's maximum overtime amount (OTMAX) will not exceed 15% of his salary.

## C. More about Foreign Keys

We describe potential errors associated with foreign-keys. Again, we refer to the TESTDEPT and TESTEMP tables defined in Figure 13.2. Note that the DNO column in TESTEMP is a foreign-key that must match some DNO value in TESTDEPT. Below we consider the behavior of the INSERT, UPDATE, and DELETE statements within the context of foreign-keys. The system will automatically reject any operation if the operation fails a foreign-key data verification check.

Review Notation:

- TESTDEPT.DNO refers to the DNO column in TESTDEPT
- TESTEMP.ENO refers to the ENO column in TESTEMP
- TESTEMP.DNO refers to the DNO column in TESTEMP

### TESTEMP Table (Child Table)

INSERT row into TESTEMP: The system verifies that the new TESTEMP.DNO value is equal to some TESTDEPT.DNO value. Chapter 12 (Sample-Session-5) presented an example of this type of verification.

UPDATE DNO column in TESTEMP: The system verifies that the new TESTEMP.DNO value is equal to some TESTDEPT.DNO value.

DELETE row from TESTEMP: No validation is necessary. A department is not required to have any employees.

### TESTDEPT Table (Parent Table)

INSERT row into TESTDEPT: No validation is necessary. A department is not required to have any employees.

UPDATE DNO column in TESTDEPT: The system verifies that the current TESTDEPT.DNO value (which will be changed) is *not* equal to some existing TESTEMP.DNO value.

DELETE row from TESTDEPT: The system verifies that the TESTDEPT.DNO value is *not* equal to some existing TESTEMP.DNO value.

To summarize, the system automatically performs a validity check on TESTDEPT for each INSERT and UPDATE of TESTEMP, and it performs a validity check on TESTEMP for each DELETE and UPDATE of TESTDEPT. The general objective is to verify that each child-row is always related to some parent-row. This validation maintains *referential integrity*.

**Variations in FOREIGN KEY Clause:** All systems support variations of the FOREIGN KEY clause. For example, most systems allow specification of the "ON DELETE CASCADE" clause as shown below.

```
CREATE TABLE TESTEMP
. . . . .
FOREIGN KEY (DNO) REFERENCES TESTDEPT ON DELETE CASCADE
```

ON DELETE CASCADE changes the behavior of a DELETE operation on a parent-table. Here, deleting a parent-row will also cause the automatic deletion of all its children. For example, if you delete a TESTDEPT row, all TESTEMP rows with matching DNO values are automatically deleted. Other variations of the FOREIGN KEY clause are beyond the scope of this book.

**Database Design**

**A Foreign-Key Defines a One-to-Many (1:M) Relationship:** The FOREIGN KEY clause in Figure 13.2 implies that:

- Each employee works for exactly one department.
- Each department can hire many employees. ("Many" includes one and zero. A department may have just one employee or no employees.)

The following data model illustrates a one-to-many relationship (HIRES) between TESTDEPT (parent-table) and TESTEMP (child-table).

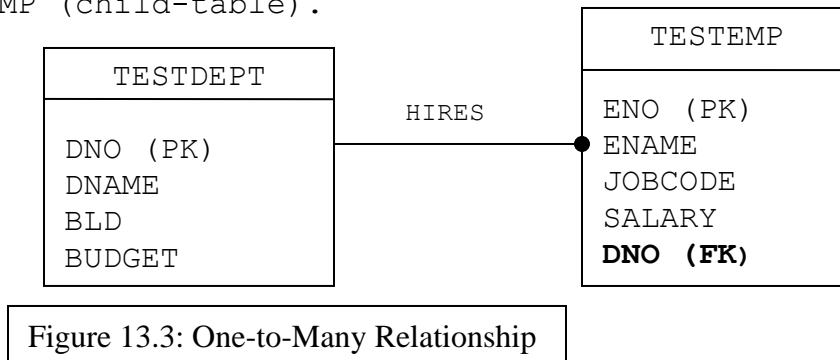


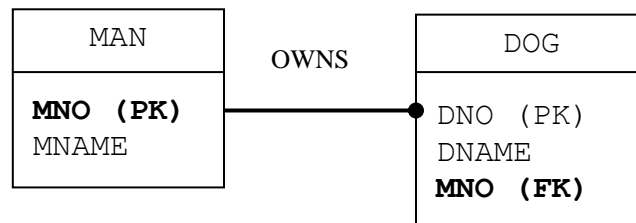
Figure 13.3: One-to-Many Relationship

## Examples: Two Data Models

Below we present two data models, each with a one-to-many relationship. These examples illustrate that a one-to-many relationship between a parent-table and a child-table is represented by specifying a FOREIGN KEY clause in the child-table.

**1. MAN-OWNS-DOG Relationship:** Sample-Session-5 in Chapter 12 presented a one-to-many relationship (OWNS). Each dog is owned by one man; and, a man can own many dogs.

The following data model illustrates the primary-key column (MNO) in the MAN table (the parent-table) and a related foreign-key column (MNO) in the DOG table (the child-table).



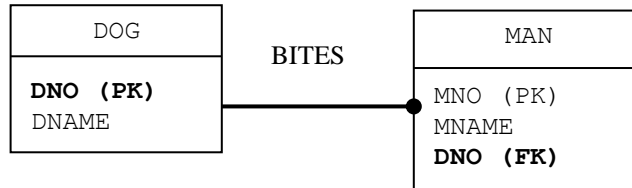
```
CREATE TABLE MAN
(MNO    INTEGER        NOT NULL,
 MNAME CHAR (10)      NOT NULL,
  PRIMARY KEY (MNO) );

CREATE TABLE DOG
(DNO    INTEGER        NOT NULL,
 DNAME CHAR (10)      NOT NULL,
 MNO    INTEGER        NOT NULL,
  PRIMARY KEY (DNO),
  FOREIGN KEY (MNO) REFERENCES MAN);
```

The FOREIGN KEY clause is specified in DOG, the child-table. It designates the MNO column as the foreign-key column. Note that this column is NOT NULL. This implies that every dog *must* be owned by some man.

**Comment:** A data model may also represent a *many-to-many* relationship. For example, a man may own many dogs, and an individual dog may be owned by multiple men. Appendix 13A will address this topic.

**2. DOG-BITES-MAN Relationship:** Assume that a dog may bite many men, and (unrealistically) each man must be bitten by exactly one dog. In this silly scenario, DOG becomes the parent-table and MAN becomes the child-table. Hence, a foreign-key in MAN references the primary-key of the DOG table.



```

CREATE TABLE DOG
(DNO      INTEGER      NOT NULL,
 DNAME    CHAR (10)    NOT NULL,
          PRIMARY KEY (DNO));

CREATE TABLE MAN
(MNO      INTEGER      NOT NULL,
 MNAME    CHAR (10)    NOT NULL,
 DNO      INTEGER      NOT NULL,
          PRIMARY KEY (MNO),
          FOREIGN KEY (DNO) REFERENCES DOG);
  
```

Examine the DNO column in the above CREATE TABLE statement for the MAN table. This column, like all other columns, is specified as NOT NULL. This is consistent with our design objective that "(unrealistically) each man *must* be bitten by exactly one dog."

Alternatively, assume we want to represent a more realistic design scenario where a man *may or may not be bitten* by a dog. (But he cannot be bitten by more than one dog). In this circumstance, the designer decides to allow the foreign-key column (MAN.DNO) to contain a null value for any man who has not been bitten by a dog. Hence, the DNO column in the following CREATE TABLE statement does not specify a NOT NULL clause.

```

CREATE TABLE MAN
(MNO      INTEGER      NOT NULL,
 MNAME    CHAR (10)    NOT NULL,
 DNO      INTEGER,
          PRIMARY KEY (MNO),
          FOREIGN KEY (DNO) REFERENCES DOG)
  
```

← No NOT NULL clause

## D. ALTER TABLE Statement

The ALTER TABLE statement allows you to make some changes to the definition of an existing table that may contain rows. We illustrate this statement by presenting two examples.

**Example-1:** Some DBAs initially code CREATE TABLE statements without any PRIMARY KEY or FOREIGN KEY clauses. These CREATE TABLE statements are followed by ALTER TABLE statements that specify the PRIMARY Key and FOREIGN KEY clauses. This approach is shown in the following Figure 13.4. It produces the same design as the code shown in Figure 13.2. This method has the advantage of allowing you to execute CREATE TABLE statements in any sequence without worrying about foreign-key dependencies.

```
DROP TABLE TESTEMP;
DROP TABLE TESTDEPT;

CREATE TABLE TESTEMP
(ENO      CHAR(3)      NOT NULL,
 ENAME    VARCHAR(25)  NOT NULL,
 JCODE    INTEGER      NOT NULL CHECK (JCODE IN (1,3,7,11)),
 SALARY   DECIMAL(7,2) NOT NULL DEFAULT 0,
 OTMAX    DECIMAL(5,2) NOT NULL DEFAULT 0,
 DNO      INTEGER      NOT NULL,
          CHECK (OTMAX <= .15 * SALARY));

CREATE TABLE TESTDEPT
(DNO      INTEGER      NOT NULL,
 DNAME    VARCHAR(20)  NOT NULL UNIQUE,
 BLD      CHAR(3)      NOT NULL DEFAULT 'AB1',
 BUDGET   DECIMAL(9,2) NOT NULL DEFAULT 0);

ALTER TABLE TESTDEPT
  ADD PRIMARY KEY (ENO);

ALTER TABLE TESTEMP
  ADD PRIMARY KEY (ENO)
  ADD FOREIGN KEY (DNO) REFERENCES TESTDEPT;
```

**Figure 13.4: ALTER TABLE Statements**

**Example-2:** The following Figure 13.5 contains an ALTER TABLE statement that adds two new columns, MADDR and MIQ, to the MAN table. Existing rows will contain null MADDR values, and their MIQ values will default to 100.

```
DROP TABLE DOG;
DROP TABLE MAN;

CREATE TABLE MAN
(MNO          INTEGER    NOT NULL PRIMARY KEY,
 MNAME       CHAR (10)  NOT NULL);

INSERT INTO MAN VALUES (77, 'MOE');
INSERT INTO MAN VALUES (88, 'LARRY');
INSERT INTO MAN VALUES (99, 'CURLY');

CREATE TABLE DOG
(DNO          INTEGER    NOT NULL PRIMARY KEY,
 DNAME       CHAR (10)  NOT NULL,
 MNO         INTEGER    NOT NULL,
 FOREIGN KEY (MNO) REFERENCES MAN);

INSERT INTO DOG VALUES (1000, 'SPOT', 99);
INSERT INTO DOG VALUES (3000, 'ROVER', 77);
INSERT INTO DOG VALUES (2000, 'WALLY', 99);

ALTER TABLE MAN
  ADD COLUMN MADDR VARCHAR (30)
  ADD COLUMN MIQ   INTEGER NOT NULL DEFAULT 100;
```

**Figure 13.5: ALTER TABLE Statement**

After executing the above statements, you can display the MAN table to observe the null values for the new MADDR column, and the 100 values for the new MIQ column.

Comment: There is considerable variation among the ALTER TABLE options available on different database systems. Again, consult your SQL manual for details.



## E. Constraint Names

You can specify the *optional* keyword `CONSTRAINT` to explicitly assign a name to a constraint. This name is stored in the system's data dictionary. (If `CONSTRAINT` is not specified, the system will automatically assign some system-generated name.) For example, you could assign the name `PK_MAN` to the `PRIMARY KEY` constraint for the `MAN` table by coding:

```
CONSTRAINT PK_MAN PRIMARY KEY (MNO)
```

Many DBAs assign constraint names according to some pattern (e.g., constraint names for primary keys begin with `PK`). This simplifies searching the data dictionary.

The `CREATE TABLE` statements in the following Figure 13.6 include constraint names for the `PRIMARY KEY` and `FOREIGN KEY` clauses. This figure does not assign constraint names to the `NOT NULL` constraints.

```
CREATE TABLE MAN
(MNO      INTEGER    NOT NULL,
 MNAME    CHAR(10)   NOT NULL,
  CONSTRAINT PK_MAN PRIMARY KEY (MNO));

CREATE TABLE DOG
(DNO      INTEGER    NOT NULL,
 DNAME    CHAR(10)   NOT NULL,
 MNO      INTEGER    NOT NULL,
  CONSTRAINT PK_DOG PRIMARY KEY (DNO),
  CONSTRAINT FK_MAN FOREIGN KEY (MNO) REFERENCES MAN);
```

**Figure 13.6: Constraint Names**

## F. DROP TABLE Statement

Sample sessions presented in Chapter 12 illustrated dropping tables in preparation for creating other tables with the same names.

The DROP TABLE statement is simple. Its general syntax is:

```
DROP TABLE table-name
```

This statement deletes all rows in the table as part of the drop table process.

**DROP TABLE Sequence:** Our discussion of the CREATE TABLE statements in Figure 13.2 noted that we created TESTDEPT (the parent table) before creating TESTEMP (the child table) because TESTEMP has a foreign key that references TESTDEPT. Likewise, if you intend to drop both the TESTDEPT and TESTEMP tables, you should drop TESTEMP first. If you were allowed to drop TESTDEPT first, then the foreign key values in TESTEMP would not match.

So, what happens if you attempt to drop TESTDEPT first without dropping TESTEMP? Different systems will take different actions. For example, if you executed:

```
DROP TABLE TESTDEPT
```

DB2 would first remove the foreign-key constraint from TESTEMP and then drop TESTDEPT.

ORACLE would return an error message. However, if you wanted ORACLE to behave like DB2, you would execute:

```
DROP TABLE TESTDEPT CASCADE CONSTRAINTS
```

**Recommendation for Application Developers:** As stated above, before dropping a table, the system will automatically delete all its rows. The deletion of many rows could incur a large efficiency cost associated with transaction processing, a topic which is discussed in Chapter 29. Application developers who execute DML statements are strongly encouraged to read this chapter.

## Summary

**Tip of the Iceberg:** This chapter covered a lot of ground. Yet, our discussion omitted many details about the CREATE TABLE and ALTER TABLE statements.

As you read the following chapters, you will realize that the information presented in this chapter will be very useful when you tackle multi-table queries presented later in this book.

**Data Model Notation:** The literature on data models shows a variety of different graphical notations. Below, we illustrate three other notations for representing a one-to-many relationship between TESTDEPT and TESTEMP.

Crow's Feet Notation:



Chen's Entity-Relationship Notation:



Arrow Notation:



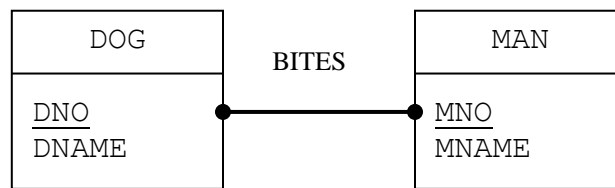
With the Arrow Notation, the arrow illustrates the direction of a foreign-key referencing a primary-skey.

## Appendix 13A: Representing Many-to-Many Relationships

The real world frequently presents the database designer with a many-to-many relationship. However, a foreign-key can only represent a one-to-many relationship. Therefore, *whenever a database designer encounters a many-to-many relationship, she must transform it into a two one-to-many relationships.* Below we present an example that illustrates this transformation.

Assume your logical data model already contains the MAN and DOG tables. Also assume that a dog may bite many men; and, a man may be bitten by many dogs. We will call this many-to-many relationship BITES.

Step-1: Represent BITES within the data model by drawing a line between MAN and DOG. This line has nodes at both ends as illustrated below in Figure 13.8.1.



**Figure 13.8.1: Step-1**

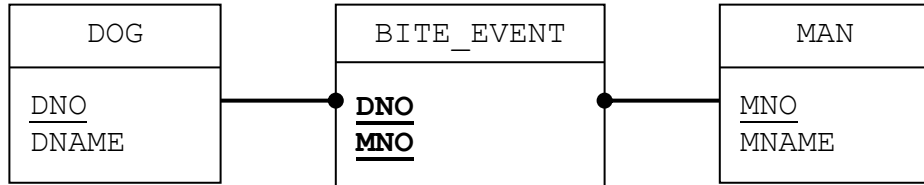
[Notation: This figure underlines the primary-key columns.]

Step-2: Transform the data model such that the many-to-many BITES relationship is replaced by a new "relationship table," the BITE\_EVENT table. Also, two new one-to-many relationships are specified. These are the one-to-many relationship between DOG and BITE-EVENT, and one-to-many relationship between MAN and BITE-EVENT. The modified data model now looks like:



**Figure 13.8.2: Step-2**

Step-3: Define a composite primary-key for the new BITE\_EVENT table composed of the primary-keys from the related tables. In this example (DNO, MNO) is designated as the primary-key of the BITE\_EVENT table. The data model now looks like the following Figure 13.8.3.



**Figure 13.8.3: Step-3**

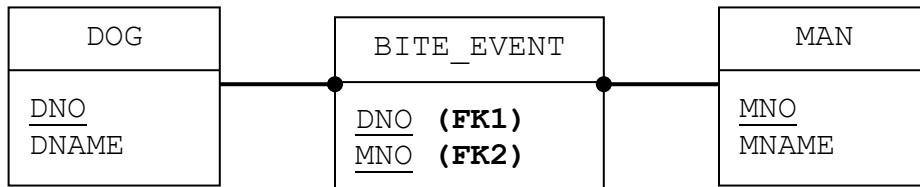
[Aside: Sometimes, a designer may also include one or more non-key columns (e.g., BITEDATE) in the BITE\_EVENT table.]

Step-4: The above data model now shows two one-to-many relationships. Hence, we must specify two foreign-keys.

(i) Considering the one-to-many relationship between DOG and BITE-EVENT, we designate BITE\_EVENT.DNO as a foreign-key referencing DOG. FK1 designates this foreign key.

(ii) Considering the one-to-many relationship between MAN and BITE\_EVENT, we designate BITE\_EVENT.MNO as a foreign-key referencing MAN. FK2 designates this foreign key.

Our data model now looks like the following Figure 13.8.4.



**Figure 13.8.4: Step-4**

The preceding four steps have transformed the original data model with a many-to-many relationship (Figure 13.8.1) into an equivalent data model (Figure 13.8.4) with a new BITE\_EVENT table and two one-to-many relationships. This revised model can now be codified as a collection of CREATE TABLE statements.

Step5: The Step-4 data model (Figure 13.8.4) contains most of the information required to formulate the CREATE TABLE statements. We only need to determine the data-type of each column and decide if any columns can contain null values. For tutorial purposes, we will make simple assumptions about data-types, and we will assume that all columns are non-null. The following Figure 13.8.5 shows the CREATE TABLE statements derived from the Step-4 data model.

```

DROP TABLE BITE_EVENT;
DROP TABLE DOG;
DROP TABLE MAN;

CREATE TABLE MAN
(MNO INTEGER NOT NULL PRIMARY KEY,
 MNAME CHAR(10) NOT NULL);

CREATE TABLE DOG
(DNO INTEGER NOT NULL PRIMARY KEY,
 DNAME CHAR(10) NOT NULL);

CREATE TABLE BITE_EVENT
(DNO INTEGER NOT NULL,
 MNO INTEGER NOT NULL,
 PRIMARY KEY (DNO, MNO),
 FOREIGN KEY (DNO) REFERENCES DOG,
 FOREIGN KEY (MNO) REFERENCES MAN);

```

**Figure 13.8.5: Step-5**

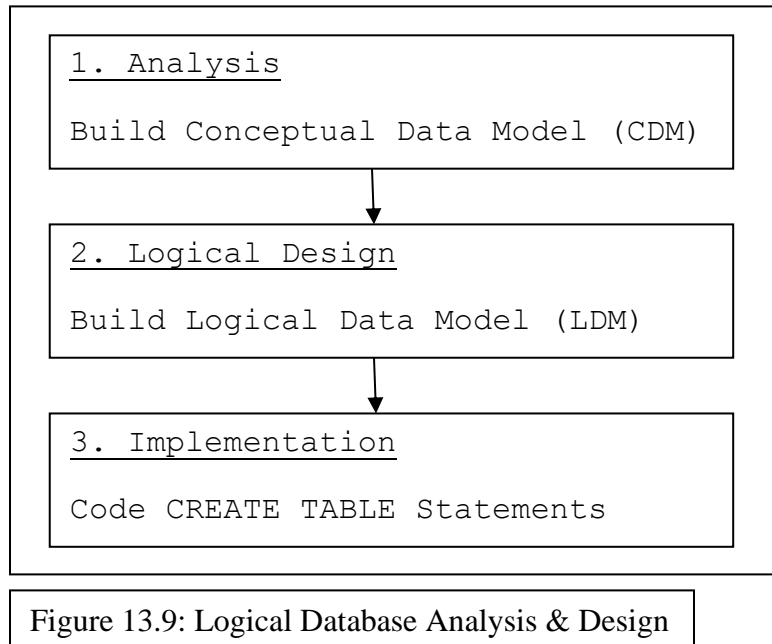
Optional Step-6: After reading Chapter 15, you will know how to code INSERT statements to insert data that looks like:

<u>MAN</u>		<u>DOG</u>		<u>BITE EVENT</u>	
<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>DNO</u>	<u>MNO</u>
77	MOE	1000	SPOT	1000	88
88	LARRY	3000	ROVER	2000	99
99	CURLY	2000	WALLY	2000	88

This sample data shows that every DNO value in BITE\_EVENT matches some DNO value in DOG; and, every MNO value in BITE\_EVENT matches some MNO value in MAN. It also shows that Dog 2000 has bitten multiple men (99 and 88); Man 88 was bitten by multiple dogs (1000 and 2000); Man 77 was not bitten by any dog; and Dog 3000 did not bite any man.

## Appendix 13B: Database Analysis & Design

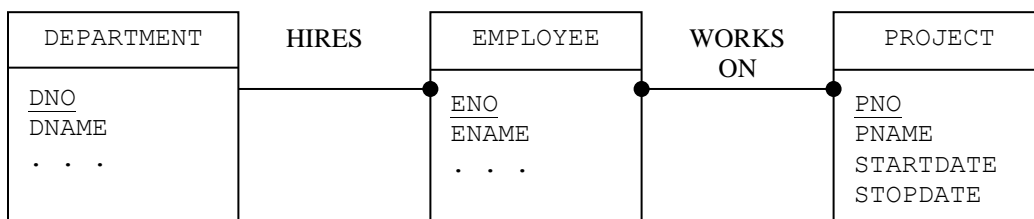
Previous examples of data models and tabular designs have previewed some design-steps for a methodology of database analysis and design. This appendix presents a more comprehensive (but still incomplete) high-level overview of an analysis/design methodology. The following Figure 13.9 illustrates a methodology that is organized into three major phases.



1. Analysis is the process of determining "what" we want to do. Specifically, database analysis determines what data we want to store in our database. The primary task to realize this objective is to build a Conceptual Data Model (CDM).
2. Design is the process of determining "how to" do what we want to do. Logical database design determines what tables, columns, and keys will be used represent the data to be stored in the database. The primary task to realize this objective is to transform the Conceptual Data Model (CDM) into a Logical Data Model (LDM). This transformation from a CDM into a LDM is generally (but not entirely) a "cookbook" process.
3. Implementation transforms the LDM into a collection of CREATE TABLE statements. This is another "cookbook" process.

## 1. Analysis

The database analyst communicates with business experts to discover relevant business object-types (e.g., DEPARTMENT and EMPLOYEE) and relationships (e.g., HIRES) between the object-types. These object-types and relationships are represented within a data model similar to the data models shown in Figures 13.2, 13.4, and 13.15.4. These models are called **Conceptual Data Models (CDM)**. A CDM is “business” model. A CDM model is presumably easy to understand because it is a graphical model that ignores many technical details. The following Figure 13.9 illustrates an example.



**Figure 13.9: Conceptual Data Model (CDM)**

A CDM excludes many details. For example, the above CDM:

- does not specify all columns for DEPARTMENT and EMPLOYEE object-types
- does not specify column data-types
- does not specify foreign-keys
- does not transform a many-to-many relationship (WORKS\_ON) into two one-to-many relationships

For these reasons, a CDM cannot be directly transformed into a collection of CREATE TABLE statements.

Database analysis involves creating a complete and correct CDM that represents your real-world application domain. This is a very challenging task which is not described in this book. Our examples will assume that someone has already done this analysis and produced a valid CDM.

[The database literature contains a large body of work devoted to formulating conceptual data models. Two popular models for representing conceptual data models are the ER (Entity-Relationship) Model and the UML (Unified Modeling Language) Model. These models include many features that are not illustrated in this book.]

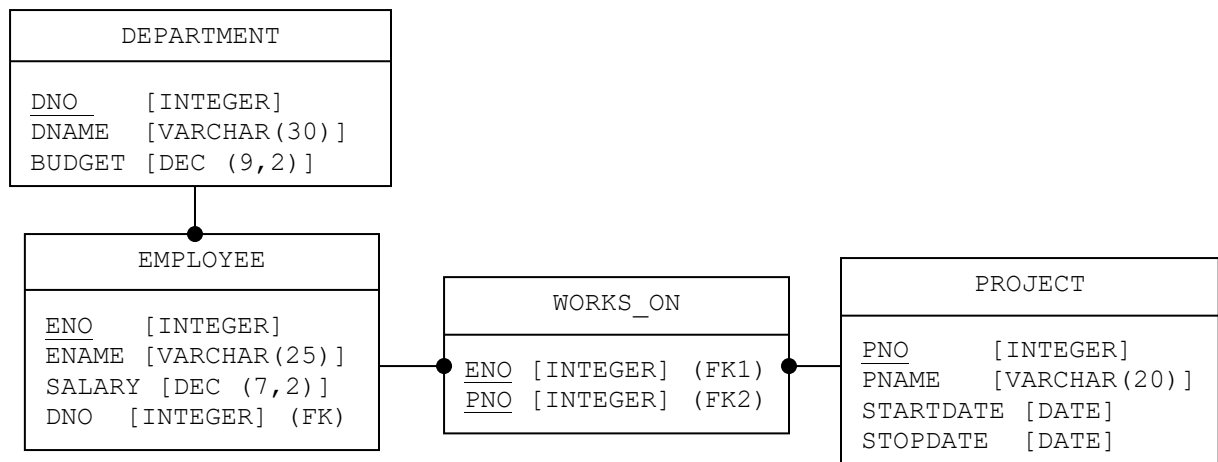


## 2. Logical Design

Assuming that database analysis has produced a complete and correct Conceptual Data Model, the logical design process becomes a step-by-step “cookbook” process. This process transforms the CDM into a more detailed Logical Data Model (LDM) that can be implemented via CREATE TABLE Statements. The basic steps for converting a CDM into a LDM are outlined below.

- Represent each object-type (e.g., DEPARTMENT) as a table and designate the primary-key for each table.
- Represent each one-to-many relationship as a foreign-key that is stored in the child-table referencing the primary-key of the parent-table.
- Represent each many-to-many relationship as a relationship-table. The primary key of this table is a composite key composed of the keys from the related tables. Each component of the composite primary key becomes a foreign-key. (Details were described in Appendix 13A.)
- Specify all columns for each table, and specify a data-type for each column.
- Specify other constraints (e.g., NOT NULL)

Applying these steps, the CDM shown in Figure 13.9 is transformed into the following LDM (Figure 13.10).



**Figure 13.10: Logical Data Model**

### 3. Implementation

Implementation transforms the LDM into a collection of CREATE TABLE statements. Here, the LDM shown in Figure 13.10 is transformed into the following collection of CREATE TABLE statements (Figure 13.11). This transformation is "cookbook." Within the LDM, each rectangle becomes a table; each attribute becomes a column; each LDM primary-key becomes a PRIMARY KEY clause; and each LDM foreign-key (FK) becomes a FOREIGN KEY clause.

```
CREATE TABLE DEPARTMENT
(DNO          INTEGER      NOT NULL PRIMARY KEY,
 DNAME        VARCHAR(20)  NOT NULL,
 BUDGET       DECIMAL(9,2) );

CREATE TABLE EMPLOYEE
(ENO          INTEGER      NOT NULL,
 ENAME        VARCHAR(25)  NOT NULL,
 SALARY       DECIMAL(7,2) NOT NULL,
 DNO          INTEGER      NOT NULL,
 PRIMARY KEY (ENO),
 FOREIGN KEY (DNO) REFERENCES DEPARTMENT);

CREATE TABLE PROJECT
(PNO          INTEGER      NOT NULL PRIMARY KEY,
 PNAME        VARCHAR(20)  NOT NULL,
 STARTDATE    DATE         NOT NULL,
 STOPDATE     DATE );

CREATE TABLE WORKS_ON
(ENO          INTEGER      NOT NULL,
 PNO          INTEGER      NOT NULL,
 PRIMARY KEY (ENO, PNO),
 FOREIGN KEY (ENO) REFERENCES EMPLOYEE,
 FOREIGN KEY (PNO) REFERENCES PROJECT);
```

**Figure 13.12: Implementation (CREATE TABLE Statements)**

For testing purposes, we recommend building a prototype database by executing these CREATE TABLE statements, populating the tables with sample data, and executing some representative sample queries.

Also, notice that the above Implementation Phase does not consider machine efficiency. Efficiency issues are considered in a separate design task called Physical Database Design.

## Physical Database Design

The preceding methodology of Logical Database Analysis and Design is "Logical" because it can be used with any relational database. It is not tightly coupled to any specific database system (e.g., DB2, ORACLE). However, Physical Database Design becomes more system-specific because each system supports different (but similar) internal physical structures that can be tuned for efficiency purposes.

The DBA controls many performance related factors that are *not addressed in this book*. These include creating special table structures (e.g., clustered tables), special types of indexes (e.g., bitmap index), increasing the size of memory buffers, assigning multiple tables to the same tablespace, and partitioning a table to facilitate parallel processing.

A database index is the only physical design structure that has been addressed in this book's Efficiency Appendices. The following optional chapter on the CREATE INDEX statement will provide more insight into index design. Below we make a few preliminary observations about this topic.

Most database systems automatically create an index for each primary-key. Given the CREATE TABLE statements shown in Figure 13.11, we conclude that most systems will automatically create four indexes.

- Index on DNO in TESTDEPT
- Index on ENO in TESTEMP
- Index on PNO in PROJECT
- Index on (ENO, PNO) in WORKS\_ON

Also, there are good reasons for creating an index on most (perhaps all) foreign-keys. However, your system will not automatically create these indexes.

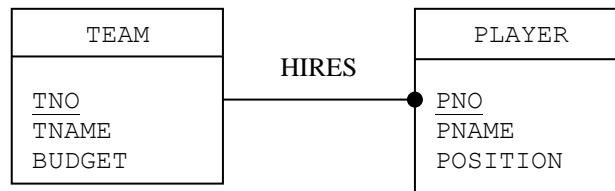
Physical design usually requires a CREATE TABLE statement to include special clauses (e.g., a TABLESPACE clause) before this statement is executed in a production environment. Such clauses are not necessary within a testing environment, and this book does not discuss these clauses.

Finally, physical database design is a fun topic. If you have enjoyed reading the preceding Efficiency Appendices, you are encouraged to read the following chapter.

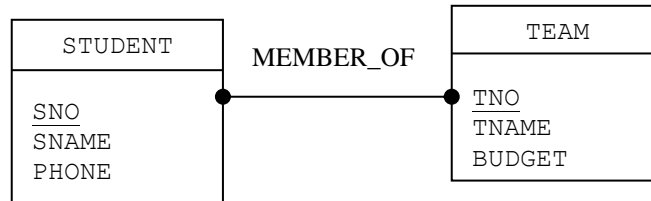
## Appendix Exercises

These exercises are optional. (They pertain to database design, a topic that is *not* the primary focus of this book.) For these exercises, you are given a Conceptual Data Model (CDM) that has been produced by database analysis. Transform this model into a Logical Data Model (LDM) and then into a collection of CREATE TABLE statements. Specify foreign-keys. Make reasonable assumptions about data-types. All columns are non-null.

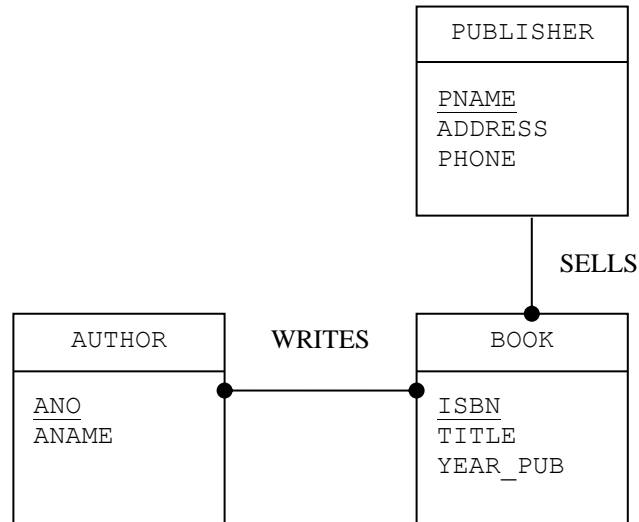
13.1 Professional Sports Team: Each player plays on just one team. Each team has many players.



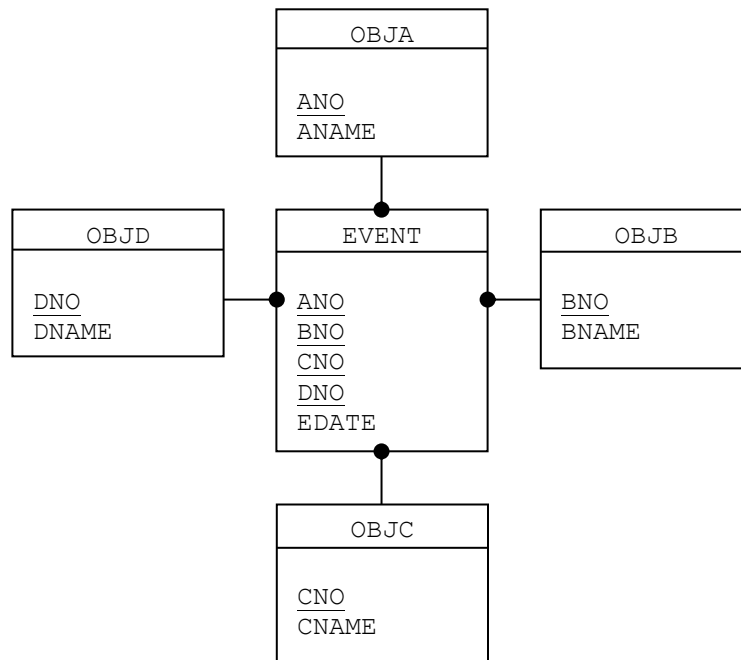
13.2 College Sports Team: A student may play on many teams. Each team has many players.



13.3 Book Publishing: A publisher sells many books. Each book has one publisher. A book may have multiple coauthors. An author may write many books.



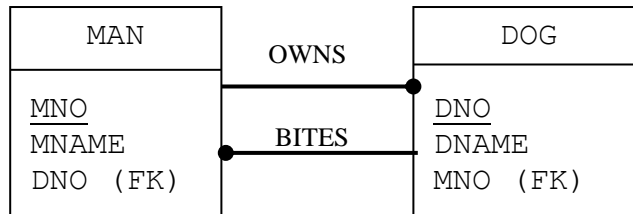
13.4 Star Design: Sometimes, within a data warehouse application, a designer creates a CDM that looks like a "star." The following star model shows an EVENT object-type as the center of the star where all other object-types (OBJA, OBJB, OBJC, and OBJD) surround EVENT and have a one-to-many-relationship with EVENT.



13.5 Cyclic Design: Sometimes multiple relationships between object-types can form a cycle. Assume we have the MAN and DOG object-types with the following two relationships.

OWNS Relationship: A man can own many dogs; and, each dog must be owned by one man.

BITES Relationship: A dog may bite many men; and each man must be bitten by one dog.



The following two CREATE TABLE statements are “almost correct.” The problem involves designating which table to create first. In the following example, which initially attempts to create the MAN table, an error occurs because its foreign-key references DOG, a table that has not yet been created. A similar problem occurs if we attempt to create the DOG table first.

```

“Almost” Correct (Chicken-Egg Problem)

CREATE TABLE MAN
(MNO    INTEGER           NOT NULL,
 MNAME  CHAR (10)        NOT NULL,
 DNO    INTEGER           NOT NULL,
 PRIMARY KEY (MNO),
 FOREIGN KEY (DNO) REFERENCES DOG);

CREATE TABLE DOG
(DNO    INTEGER           NOT NULL,
 DNAME  CHAR (10)        NOT NULL,
 MNO    INTEGER           NOT NULL,
 PRIMARY KEY (DNO),
 FOREIGN KEY (MNO) REFERENCES MAN);
    
```

Utilize ALTER TABLE statements, as illustrated in Figure 13.4 to resolve this problem.

This page is intentionally blank.

## CREATE INDEX Statement

This chapter is optional reading for all readers. Appendices 1A, 2A, and 4A have already explained “why” you might want to create an index. This chapter’s sample statements explain “how to” create an index.

The CREATE INDEX statement is (obviously) used to create a database index. Its syntax is straightforward. However, you must understand some basic concepts before you create an index. Therefore, you are encouraged to read/review Appendices 1A, 2A, and 4A before reading this chapter. Furthermore, if you decide to read this chapter, you should also read its related Appendix 14A.

This chapter’s sample statements will reference the TESTDEPT and TESTEMP tables shown in Figure 13.2. Most systems automatically create an index on any column that is declared to be a PRIMARY KEY or UNIQUE. Therefore, we will assume that your system has automatically created indexes on the TESTDEPT.DNO, TESTDEPT.DNAME, and TESTEMP.ENO columns. Because each of these columns contains unique values, each corresponding index is a “unique-index.” Appendix A2 (Figure A2.1) illustrated an example of a unique-index, the XPNO index, based upon the PNO column in the PRESERVE table.

Your system will *not* automatically create an index on a foreign-key column. However, many designers create an index on every foreign-key because foreign-key columns are frequently referenced in SELECT statements the specify join-operations (to be described Chapter 16).



## “Simple” Index

A “simple” (another unofficial term) index is an index that is based on just one column.

**Sample Statement 14.1:** Create an index, called XENAME, based on the ENAME column in the TESTEMP table.

```
CREATE INDEX XENAME
ON TESTEMP (ENAME)
```

**System Response:** The system should return a message indicating successful creation of the index.

**Syntax:** A simplified version of the general syntax is:

```
CREATE INDEX index-name
ON table-name (column-name)
```

- The name of the index follows CREATE INDEX. (We specified an “X” before the column-name to form “XENAME”. This naming pattern is optional.)
- The ON-clause must reference a valid table-name.
- The column-name must be enclosed within parentheses and must reference a column within the specified table.

**Behavior:** If TESTEMP is a recently created table that does not contain any rows, the system will construct an “empty” index. Subsequently, it will insert a new ENAME entry into the index after each successful INSERT operation. If the TESTEMP table already contains some rows, the system will construct the index by scanning the table to obtain ENAME values and pointers to row locations. This process could take some time if the table has many rows.

After an index is created, the system will automatically maintain it. For example, if you insert a new row into the TESTEMP table, the system will automatically use its ENAME value and its row location to store a new entry in the XENAME index; and, if you delete a row, the system will automatically remove the index entry that references the row. Simply put, once you create the index, the system does the rest.

## Unique Index

A unique index must be based on a unique column.

**Sample Statement 14.2:** For this example, *temporarily* assume the CREATE TABLE statement for TESTDEPT did *not* designate the DNAME column as UNIQUE. Under this circumstance, the system would not have automatically created an index on the DNAME column. Therefore, you could explicitly create a *unique* index on DNAME by executing the following statement.

```
CREATE UNIQUE INDEX XDNAME
ON TESTDEPT (DNAME)
```

**System Response:** If you attempt to execute this statement, the system should return an error message indicating that there already exists a unique index on DNAME column.

**Syntax:** The optional keyword UNIQUE is placed after CREATE. The general syntax expands to:

```
CREATE [UNIQUE] INDEX index-name
ON table-name (column-name)
```

**Behavior:** Assume an INSERT operation attempted to insert ACCOUNTING into the DNAME column in the TESTDEPT table. Before allowing this operation to successfully complete, the system must verify that ACCOUNTING does not already appear in the DNAME column of any existing row. Without the XDNAME index, the system would have to scan all TESTDEPT rows to determine the presence/absence of an existing ACCOUNTING value. With the XDNAME index, the system could directly access this index to determine the presence/absence of an existing ACCOUNTING value.

**UNIQUE Constraint versus CREATE UNIQUE INDEX:** We previously stated that most systems automatically create a *unique-index on each UNIQUE column*. Therefore, you would not need to create a unique-index on any column that was already specified as UNIQUE. *Declaring a column as UNIQUE within the CREATE TABLE statement is usually the preferred method.* (The answer to Exercise 14.3 in Appendix 14A will explain this preference.)

## Composite Index

Assume you frequently execute SELECT statements with WHERE-clauses that match the following code pattern.

```
SELECT * FROM TESTEMP
WHERE JOBCODE = _____ AND SALARY = _____
```

You could create a simple index on either or both of the JOBCODE or SALARY columns. However, you should also consider creating one index that is defined on both columns. This kind of index is called a composite (or compound) index.

**Sample Statement 14.3:** Create a composite index on two columns in the TESTEMP table. The first column is JOBCODE. The second column is SALARY. (We will see that the order of column specification is important.)

```
CREATE INDEX XJOBSAL
ON TESTEMP (JOBCODE, SALARY)
```

**System Response:** The system should return a message indicating the successful creation of the index.

**Syntax:** The general syntax expands to:

```
CREATE [UNIQUE] INDEX index-name
ON table-name (column-name1, column-name2,...)
```

**Logic:** This index could help any SELECT statement that specifies one of following WHERE-clause patterns.

```
WHERE JOBCODE = _____ AND SALARY = _____
```

```
WHERE SALARY = _____ AND JOBCODE = _____
```

```
WHERE JOBCODE = _____
```

However, most systems (usually) would not use the XJOBSAL index for a SELECT statement with the following WHERE-clause pattern.

```
WHERE SALARY = _____
```

An explanation is provided in Appendix 14A.

## Unique-Composite Index

A composite index can be defined as UNIQUE. The previous CREATE INDEX statement did not create a unique-composite index on the JOBCODE and SALARY columns because we implicitly assumed that multiple EMPLOYEE rows could possibly have the same JOBCODE and SALARY values.

**Sample Statement 14.4:** Note that the TESTEMP.ENAME column is allowed to contain duplicate values. Assume that we have an unusual business rule: If two employees have the same name, they cannot work in the same department. This means that each pair of (ENAME, DNO) values should always be unique. Create a unique composite index on these two columns where ENAME is the first part of the index.

```
CREATE UNIQUE INDEX XENAMEDNO
ON TESTEMP (ENAME, DNO)
```

**System Response:** Again, the system should return a message indicating the successful creation of the index.

**Logic:** This index will enforce the stated business rule, and it could also help any SELECT statement that specifies one of following WHERE-clause patterns.

WHERE ENAME = \_\_\_\_\_ AND DNO = \_\_\_\_\_

WHERE DNO = \_\_\_\_\_ AND ENAME = \_\_\_\_\_

WHERE ENAME = \_\_\_\_\_

Most systems (usually) will not use the XENAMEDNO index for a query with the following WHERE-clause pattern.

WHERE DNO = \_\_\_\_\_

An explanation is provided in Appendix 14A.

## Index Sequence

In Appendix 2A, we noted that the system could utilize an index to return rows in some sequence without sorting the rows. (See the section on "Indexes Facilitate Sorting.") Figure 2.1 illustrated an index with values stored in ascending sequence. This is the default sequence. Therefore, previous CREATE INDEX statements defaulted to an ascending (ASC) sequence.

After creating the XENAME index, the system could use it when a SELECT statement contains ORDER BY ENAME. However, what if many of your SELECT statements contained the following ORDER BY clause?

```
ORDER BY ENAME DESC
```

In this circumstance, you should consider creating an index where its ENAME values are stored in descending sequence.

**Sample Statement 14.5:** Similar to Sample Statement 14.1. Create an index, called XENAME2, based on the ENAME column in TESTEMP. The values in this index should be stored in descending sequence.

```
CREATE INDEX XENAME2  
ON TESTEMP (ENAME DESC)
```

**System Response:** Again, the system should return a message indicating the successful creation of the index.

**Syntax:** The general syntax expands to:

```
CREATE [UNIQUE] INDEX index-name  
ON table-name (col-name1 [ASC|DESC],  
               col-name2 [ASC|DESC],...);
```

Finally, we note that all database systems offer other extensions to the CREATE INDEX statement that are not covered in this book.

## Summary

This chapter introduced the basic syntax of the CREATE INDEX statement. We conclude some observations.

**Indexes do not require any changes to SELECT statements:** The presence or absence of an index should not impact the coding of SELECT statements. The optimizer (introduced in Appendix 4A) should decide to use an index if it is beneficial.

**Index Design:** Each table can have many indexes. Recall that Appendix 2A described circumstances where the cost of an index might exceed its benefits. The following Appendix 14A will offer some general guidelines.

**Naming Indexes:** Index-names can be explicitly assigned by two methods.

1. As illustrated in this chapter, the CREATE INDEX statement assigns a name to the index.
2. Assume the system automatically creates an index in response to a column being designated as a PRIMARY KEY or UNIQUE. *If* a CONSTRAINT clause has been specified, it can be assigned a name, and this name becomes the name of the index. For example, assume the CREATE TABLE statement that created the TESTDEPT table (Figure 13.2) was changed to look like:

```
CREATE TABLE TESTDEPT
(DNO      INTEGER      NOT NULL
                        CONSTRAINT PKDNO PRIMARY KEY,
 DNAME    VARCHAR(20)  NOT NULL
                        CONSTRAINT UDNAME UNIQUE,
 BLD      CHAR(3)      NOT NULL DEFAULT 'AB1',
 BUDGET   DECIMAL(9,2) NOT NULL DEFAULT 0);
```

In this circumstance, PKDNO is the name of the index based on the DNO column, and UDNAME is the name of the index based on the DNAME column.

Otherwise, if a column is designated as a PRIMARY KEY or UNIQUE column *without* specifying a CONSTRAINT name (as in Figure 13.2), the system assigns some system-generated index-name (which may be ugly).

## Appendix 14A: Index Design for Efficiency

**Guidelines for Creating Indexes:** In most real-world applications, choosing the optimal set of indexes is practically impossible. Without detail information about query patterns and frequency of query execution, the best you can do is to follow *some general* guidelines.

- Create a unique index on PRIMARY KEY and UNIQUE column(s). Most systems will automatically do this for you.
- Create an index on each FOREIGN KEY in a large table.
- Create an index on any column in a large table that will be frequently referenced in a WHERE-clause.
- Create an index on any column in a large table that will be frequently referenced in an ORDER BY clause.
- Avoid "index redundancy." For example, if you have a composite index on ENAME-JOBCODE, do not create another simple index on ENAME. (See following topic.)

**Designing Composite Indexes:** *The column sequence within a composite index is significant.* The basic idea becomes evident when we consider a traditional telephone book where each person's last name appears before their first name, and the book is sorted by first name within last name.

Assume the telephone book is represented by a database table with three columns (LASTNAME, FIRSTNAME, PHONENUMBER), and you created a composite index on the (LASTNAME, FIRSTNAME) columns. Each entry in this index would look like:

LASTNAME	FIRSTNAME	[pointer to row in table]
----------	-----------	---------------------------

This index, like a physical telephone book, provides direct access for the following three query objectives.

- Q1. Display the PHONENO for each person whose LASTNAME is Martyn and FIRSTNAME is Jessica.
- Q2. Display the PHONENO for each person whose FIRSTNAME is Jessica and LASTNAME is Martyn.
- Q3. Display the PHONENO values for all persons whose LASTNAME is Martyn.

For these three query objectives, you would you start your search by *directly* accessing the first person whose last name is Martyn in the M-section of the phonebook/index.

However, this phonebook/index does *not provide direct access* for the following query objective.

Q4. Display the PHONENO values for all persons whose FIRSTNAME is Jessica. (Note: FIRSTNAME is the second part of index.)

To summarize, the composite index (LASTNAME, FIRSTNAME) provides direct access for the following WHERE-clause patterns because each WHERE-clause references LASTNAME which is the first component of the composite index.

```
WHERE FIRSTNAME = _____ AND LASTNAME = _____  
WHERE LASTNAME = _____ AND FIRSTNAME = _____  
WHERE LASTNAME = _____
```

This index does not offer direct access for the following WHERE-clause pattern because it does not reference LASTNAME.

```
WHERE FIRSTNAME = _____
```

Returning to Sample Statement 14.3, you should now understand why the XJOBSAL index helps the following search conditions.

```
WHERE JOBCODE = _____ AND SALARY = _____  
WHERE SALARY = _____ AND JOBCODE = _____  
WHERE JOBCODE = _____
```

The XJOBSAL composite index (JOBCODE, SALARY) specifies JOBCODE as the first component of this index, and each of the above WHERE-clause patterns reference JOBCODE. You should also understand why the XJOBSAL index does not help the following WHERE-clause pattern.

```
WHERE SALARY = _____
```

Caveat: In some circumstances (not described in this book), a system might *scan* (not directly access) the XJOBSAL index to satisfy the above search objective (WHERE SALARY = \_\_\_\_\_). This is called "skip sequential" access. See your reference manual for details.



**Composite Index on Three Columns:** A composite index can reference more than two columns. If you consider the number of columns in a typical table, the number of permutations of all columns becomes very large. To simplify our query analysis, we assume that our WHERE-clauses frequently reference three popular columns (COLA, COLB, and COLC). Then we can consider creating any of the following 15 possible indexes.

(COLA)	(COLA, COLB)	(COLA, COLB, COLC)
(COLB)	(COLA, COLC)	(COLA, COLC, COLB)
(COLC)	(COLB, COLA)	(COLB, COLA, COLC)
	(COLB, COLC)	(COLB, COLC, COLA)
	(COLC, COLA)	(COLC, COLA, COLB)
	(COLC, COLB)	(COLC, COLB, COLA)

However, observe that there is considerable redundancy within these indexes. For example, if you have already created the (COLA, COLB) index, then it is highly unlikely that you will need to create the (COLA) index.

Observe that the following three indexes can satisfy the same direct access objectives as the above 15 indexes.

Index-1: (COLA, COLB, COLC)

- helps WHERE-clauses that reference all three columns
- helps WHERE-clauses that reference COLA and COLB
- helps a WHERE-clause that references COLA

Index-2: (COLB, COLC)

- helps WHERE-clauses that reference COLB and COLC
- helps a WHERE-clause that references COLB

Index-3: (COLC, COLA)

- helps WHERE-clauses that reference COLC and COLA
- helps a WHERE-clause that references COLC

These three indexes provide direct access for any of the 15 possible (simple or compound) conditions that reference columns COLA, COLB, or COLC.

Real-world: Because you will search on more than three columns, choosing an optimal set of indexes presents a very complex design challenge. Therefore, in practice, designers are forced to create some "near-optimal" collection of indexes.

## Appendix Exercises

14.1 Assume that the solution to Exercise 13.1 is coded in the following script.

```
DROP TABLE WRITES;
DROP TABLE AUTHOR;
DROP TABLE BOOK;
DROP TABLE PUBLISHER;

CREATE TABLE PUBLISHER
(PNAME      VARCHAR (30)      NOT NULL,
 ADDRESS    VARCHAR (40)      NOT NULL,
 PHONE      CHAR (10),
 PRIMARY KEY (PNAME));

CREATE TABLE BOOK
(ISBN       VARCHAR (20)      NOT NULL,
 TITLE      VARCHAR (40)      NOT NULL,
 YEAR_PUB   CHAR (4)          NOT NULL,
 PNAME      VARCHAR (30)      NOT NULL,
 PRIMARY KEY (ISBN),
 FOREIGN KEY (PNAME) REFERENCES PUBLISHER);

CREATE TABLE AUTHOR
(ANO        INTEGER           NOT NULL,
 ANAME      VARCHAR (30)      NOT NULL,
 PRIMARY KEY (ANO));

CREATE TABLE WRITES
(ANO        INTEGER           NOT NULL,
 ISBN       VARCHAR (20)      NOT NULL,
 PRIMARY KEY (ANO, ISBN),
 FOREIGN KEY (ANO) REFERENCES AUTHOR,
 FOREIGN KEY (ISBN) REFERENCES BOOK);
```

- a. What indexes would (usually) be automatically created by the system?
- b. Create an index on all foreign-keys.
- b. Create a composite index on the TITLE and YEAR\_PUB columns (in that order) found in the BOOK table.
- c. After creating the above indexes, and taking into consideration the automatically created indexes, the total number of indexes is \_\_\_\_.

14.2 Assume that: (i) both the TESTDEPT and TESTEMP tables are very large, (ii) the ENAME column is not unique because two employees may have the same name, and (iii) your organization has an unusual policy of forbidding the assignment of two employees having the same name to the same job. Consider the following query patterns:

You will frequently search on JOBCODE only.

WHERE JOBCODE = \_\_\_\_\_

You almost never execute a query that searches on ENAME only.

WHERE ENAME = \_\_\_\_\_

Occasionally you execute a query with a WHERE-clause that looks like:

WHERE ENAME = \_\_\_\_\_ AND JOBCODE = \_\_\_\_\_

or

WHERE JOBCODE = \_\_\_\_\_ AND ENAME = \_\_\_\_\_

Create one composite index on both the ENAME and JOBCODE columns that could be helpful.

14.3 This is an unfair exercise. But we invite you to speculate on an answer.

Discussion of Sample Statement 14.2 raised a design decision. If a column will contain unique values, should you declare a UNIQUE constraint or create a UNIQUE index? We stated that declaring a UNIQUE column within the CREATE TABLE statement is usually the preferred approach. Justify this preference.

## Data Manipulation: INSERT – UPDATE - DELETE

This chapter presents details about the three major DML statements (INSERT, UPDATE, and DELETE) that were previewed in Chapter 12. These DML statements are very simple. However, because they are very powerful, careless execution of these statements could destroy valuable data. Therefore, every database system provides a security system to verify that each user has the necessary privileges to execute specific DML statements on a given table. Also, if you intend to execute DML statements within a production environment, you are strongly encouraged read Chapter 29 on Transaction Processing.

**MYDEPT Table:** If you want to execute the sample statements presented in this chapter, you must create a table called MYDEPT by executing the following CREATE TABLE statement. By creating this table, you automatically have all privileges required to execute any DML statement against this table.

```
CREATE TABLE MYDEPT
(DNO      INTEGER          NOT NULL PRIMARY KEY,
 DNAME    VARCHAR (20)    NOT NULL UNIQUE,
 BLD      CHAR (3)        NOT NULL DEFAULT 'AB1',
 BUDGET   DECIMAL (9,2))
```

The following sample statements will insert rows into MYDEPT, update some of these rows, delete some of these rows, and finally drop this table.

## INSERT Statement

Assume MYDEPT has just been created. Hence it is empty.

**Sample Statement 15.1.1:** Insert a row into MYDEPT with the following values.

- DNO: 10
- DNAME: ACCOUNTING
- BLD: XX5
- BUDGET: 75000.00

```
INSERT INTO MYDEPT
VALUES (10, 'ACCOUNTING', 'XX5', 75000.00)
```

**System Response:** The system should respond with a message implying successful insertion of the new row. Verify the presence of the new row by executing:

```
SELECT * FROM MYDEPT
```

The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BLD</u>	<u>BUDGET</u>
10	ACCOUNTING	XX5	75000.00

**Syntax:** The table-name (MYDEPT) follows INSERT INTO. This is followed by VALUES, followed by the column values enclosed within parentheses. Each inserted value must have the correct data-type. (Apostrophes must enclose character-strings.) Each value must be separated by a comma, which may or may not be followed by one or more spaces.

This INSERT statement does not explicitly specify column-names. Therefore, the VALUES-clause must specify its values in the MYDEPT table's left-to-right column sequence (as specified in its CREATE TABLE statement).

**Logic:** The system automatically validates the data-type of each value. Because DNO is the primary key of MYDEPT, the system will verify that the new DNO value (10) does not already exist in the table. Also, because DNAME is declared to be unique, the system will verify that the new DNAME value (ACCOUNTING) does not already exist in the table.

**Sample Statement 15.1.2:** We satisfy the previous statement objective by presenting an alternative INSERT statement that explicitly specifies column-names.

[Do not execute this statement if you have already executed the previous INSERT statement. It will fail because of duplicate DNO values.]

```
INSERT INTO MYDEPT (DNO, DNAME, BUDGET, BLD)
VALUES (10, 'ACCOUNTING', 75000.00, 'XX5')
```

**Syntax:** For tutorial purposes, the above column-names are *not* specified in MYDEPT's left-to-right column sequence. Note that BUDGET is the fourth column in the MYDEPT table, but it is specified as the third column in the above list of column-names. This statement illustrates that column-names can be specified in any left-to-right sequence, but corresponding values in the VALUES clause must be specified in the same sequence.

**Exercise:**

15A. Execute the following statement to create the JUNK1 table.

```
CREATE TABLE JUNK1
(C1 INTEGER NOT NULL PRIMARY KEY,
C2 CHAR (5),
C3 VARCHAR (10))
```

Insert the following row into JUNK1.

250	HELLO	DOOPY
-----	-------	-------

Verify the CREATE TABLE and INSERT operations by executing:  
SELECT \* FROM JUNK1

## INSERT with NULL Values

Note that the BUDGET column is the only MYDEPT column that allows null values. The following INSERT statement specifies NULL to represent an unknown BUDGET value. Again, we show two equivalent INSERT statements.

**Sample Statement 15.2.1:** Insert a row into MYDEPT with the following known values.

- DNO: 40
- DNAME: ENGINEERING
- BLD: XX7

```
INSERT INTO MYDEPT
VALUES (40, 'ENGINEERING', 'XX7', NULL)
```

**Syntax & Logic:** The keyword NULL is specified for the unknown BUDGET value. NULL can be specified for any data-type.

Verify the new row by executing: `SELECT * FROM MYDEPT`  
The result should look like:

DNO	DNAME	BLD	BUDGET
10	ACCOUNTING	XX5	75000.00
40	ENGINEERING	XX7	-

The hyphen represents a null value. (Your system may display a blank or some different symbol to represent a null value.)

**Sample Statement 15.2.2:** The following equivalent INSERT statement specifies column-names. [Do not execute this statement if you have already executed the previous INSERT statement.]

```
INSERT INTO MYDEPT (DNO, DNAME, BLD)
VALUES (40, 'ENGINEERING', 'XX7')
```

**Syntax:** The names and values of the three *known* columns (DNO, DNAME, BLD) are specified. The unspecified column (BUDGET) value will automatically be set to a null.

## INSERT with Default Values

Note that the BLD column is the only MYDEPT column that specifies a default value.

**Sample Statement 15.3:** Insert another row into MYDEPT. Assume BLD and BUDGET values are unknown. The BLD column will be assigned its default value ('AB1'), and BUDGET will be assigned a null value. The known values are:

- DNO: 20
- DNAME: INFO. SYS.

```
INSERT INTO MYDEPT (DNO, DNAME)
VALUES (20, 'INFO. SYS.')
```

**Syntax & Logic:** The column-names and values of the two *known* values are specified. Verify the insert operation by executing:

```
SELECT * FROM MYDEPT
```

The result should look like:

DNO	DNAME	BLD	BUDGET
10	ACCOUNTING	XX5	75000.00
40	ENGINEERING	XX7	-
20	INFO. SYS.	<b>AB1</b>	-

The following Sample Statement 15.4 will execute an UPDATE statement against MYDEPT which contains the above three rows.

Cautionary Comment: Unlike some programming languages, SQL does not allow you to code adjacent commas to imply that a value is not specified. The system will reject the following statement.

```
INSERT INTO MYDEPT
VALUES (40, 'ENGINEERING', , )      → Error
```



## UPDATE Statement

The UPDATE statement is used to change row values in one or more columns of a table. The following sample statement only changes one column value in one row.

**Sample Statement 15.4:** Change the BLD value of the department with a DNO value of 40. Set the new BLD value to XX5.

```
UPDATE MYDEPT
SET BLD = 'XX5',
WHERE DNO = 40
```

**System Response:** The system should respond with a message implying successful update of the row. Verify the update by executing: `SELECT * FROM MYDEPT`

The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BLD</u>	<u>BUDGET</u>
10	ACCOUNTING	XX5	75000.00
40	ENGINEERING	<b>XX5</b>	-
20	INFO. SYS.	AB1	-

**Syntax & Logic:** This statement has three clauses.

- UPDATE is followed by the name of the table to be changed.
- The SET-clause follows the UPDATE-clause. It specifies one or more columns to be changed. The above SET-clause only modifies one column (BLD). The SET-clause can specify a value (e.g., 'XX5'), an expression, or NULL.
- The WHERE-clause follows the SET-clause. This WHERE-clause is coded like a WHERE-clause in a SELECT statement. It has the same syntax and may contain Boolean operators. Here, instead of identifying rows to be retrieved, the WHERE-clause identifies the row(s) to be changed.

**Important Know-Your-Data Observation:** *Knowing that DNO is unique guarantees that no more than one row can be changed.*

**More about the SET-Clause:** Sample Statement 15.5 will illustrate that more than one column can be changed by placing a comma between each "column-name = \_\_\_\_\_" expression as illustrated below.

```
UPDATE table-name
SET  column-name = _____,
     column-name = _____,
     column-name = _____,
     column-name = _____
WHERE condition
```

**Important:** If the WHERE-clause identifies multiple rows, then all such rows are updated.

**Be Very Careful!!!** The WHERE-clause is optional. However, we emphasize that *the absence of a WHERE-clause will change every row in the table*. For example, to change all BUDGET values to 45000.00, you would execute

```
UPDATE MYDEPT
SET    BUDGET = 45000.00
```

[Do not execute this statement.]

This behavior is analogous to the SELECT statement. The absence of a WHERE-clause means that all rows are selected. With an UPDATE statement, the absence of a WHERE-clause means that all rows are updated.

**Exercises:**

15B. Insert the following three rows into JUNK1 (which currently has one row). The hyphen represents a null value.

150	-	HAPPY
350	HI	SAD
850	BYE	-

Verify these insert operations by executing:  
SELECT \* FROM JUNK1

15C. Update the JUNK1 table. Change the row where column C1 equals 150. Its new C3 value should be GRUMPY. Verify this update operation by executing: SELECT \* FROM JUNK1

## Update Multiple Columns in Multiple Rows

**Sample Statement 15.5:** For every MYDEPT row that has a DNO value that is less than 35, make the following changes.

- Set the BDL value to WW9
- Increase the BUDGET value by 20%

```
UPDATE MYDEPT
SET   BLD = 'WW9',
      BUDGET = BUDGET * 1.20
WHERE DNO < 35
```

**System Response:** The system should respond with a message implying a successful UPDATE operation.

Verify by executing: SELECT \* FROM MYDEPT

The result should look like:

DNO	DNAME	BLD	BUDGET
10	ACCOUNTING	<b>WW9</b>	<b>90000.00</b>
40	ENGINEERING	XX5	-
20	INFO. SYS.	<b>WW9</b>	-

**Syntax:** This example changes two columns. These changes are specified by the two expressions in the SET-clause. A comma separates these expressions. Some systems will allow you to code a more compact version of this SET-clause as shown below:

```
UPDATE MYDEPT
SET (BLD, BUDGET) = ('WW9', BUDGET*1.20)
WHERE DNO < 35
```

**Logic:** This statement changes the two rows that match the WHERE-clause. Both new BLD values are WW9. New BUDGET values are based on the current value. For the row with DNO = 10, the current value of 75,000 is increased by 20% yielding a new value of 90,000. For the row with DNO = 20, the current value is null; hence the expression produces a null value.

### Exercise:

15D. Update the JUNK1 table. Change all rows having a C2 value beginning with the letter H. The new C3 value for each row should be set to CRANKY. Verify this update operation by executing: SELECT \* FROM JUNK1

## DELETE Statement

The DELETE statement can delete any number of rows from a table. (You cannot delete columns from a table.)

**Sample Statement 15.6:** Delete every row from the MYDEPT table that has a DNAME value ending with "ING".

```
DELETE
FROM MYDEPT
WHERE DNAME LIKE '%ING'
```

**System Response:** The system should return a message confirming the successful deletion of rows. Verify the deletes by executing: SELECT \* FROM MYDEPT

The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BLD</u>	<u>BUDGET</u>
20	INFO. SYS.	WW9	-

**Syntax:** The syntax is simple. DELETE FROM is followed by the table-name. The optional WHERE-condition identifies the row(s) to be deleted.

**Logic:** The two rows matching the WHERE-clause were deleted.

**Careful!** We emphasize that **failure to include a WHERE-condition will delete all rows**. For example, the following statement will delete all rows in MYDEPT.

```
DELETE FROM MYDEPT → Careful!!!
```

If you executed this statement because you intended to delete all rows, the MYDEPT table will still exist, but it will be empty. If desired, you could then insert rows. This is different than executing DROP TABLE MYDEPT which automatically deletes all rows before it drops the table.

### Exercise:

15E. Delete any row from the JUNK1 table with a C1 value that exceeds 300. Verify this delete operation by executing: SELECT \* FROM JUNK1

## Summary

This chapter introduced SQL's three basic DML statements: INSERT, UPDATE, and DELETE. Only the basic versions of these statements were described. In particular, the specification of "Sub-SELECT" clauses were not described. Chapter 24 will address this topic.

**Other DML Statements:** Most systems support other DML statements in addition to INSERT, UPDATE, and DELETE. Some systems support the MERGE statement that blends the functionality of INSERT and UPDATE within a single statement. Also, some systems support the TRUNCATE statement that offers an efficient method to delete all rows from a table.

**Transaction Processing:** This chapter executed INSERT, UPDATE, and DELETE statements within an interactive environment. However, DML statements are usually embedded within application programs and stored procedures that execute in a production environment. Such programs/procedures frequently include transaction-processing statements such as COMMIT and ROLLBACK. *Transaction processing is a very important topic for application developers. This topic will be presented in Chapter 29.*

## Summary Exercises

15F. DELETE all rows from JUNK1.

15G. INSERT the following rows into JUNK1.

98	YES1	YES2
95	NO1	NO2

15H. Update JUNK1. Set all C2 values to MAYBE.

15I. DELETE any row where the C1 value is greater than 95.

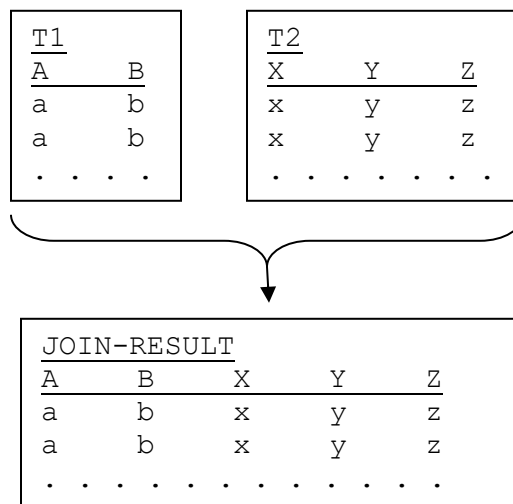
15J. Drop the JUNK1 table.

/

# PART IV

## Join-Operations

Casually speaking, a join-operation matches and merges data from multiple tables into a single result (or intermediate result) table. For example, given the following tables, T1 and T2, you can produce a result table that looks something like the following JOIN-RESULT table.



There are two fundamental variations of the join-operation. These are: (1) the Inner-Join, and (2) the Outer-Join. The Inner-Join is described in Chapters 16-18. The Outer-Join is described in Chapters 19-20. Chapter 20.5 will present sample queries that specify a mixture of inner-join and outer-join operations.

## Topics

### Chapter 16: Inner-Join: Getting Started

This chapter introduces the basic inner-join of two tables.

### Chapter 17: More about Inner-Joins

This chapter does not introduce any new concepts. It presents sample queries that specify the inner-join of two tables in conjunction with previously described SQL operations.

### Chapter 18: Multi-Table Inner-Joins

This chapter presents inner-join operations applied to three or more tables.

### Chapter 19: Outer-Join: Getting Started

This chapter introduces the three basic types of outer-join operations. Sample queries are restricted to outer-join operations involving two tables.

### Chapter 20: Multi-Table Left Outer-Joins

This chapter presents left outer-join operations applied to three or more tables.

### Chapter 20.5: Mixing Inner-Joins and Outer-Joins

This chapter presents sample queries with FROM-clauses that specify both inner-join and outer-join operations.

A Reminder about Column Names: Future sample queries will join tables where multiple tables usually have a column with the same column-name. For example, Figure 13.2 showed the creation of the TESTDEPT and TESTEMP tables where both tables contain a column called DNO. Recall that:

TESTDEPT.DNO references the DNO column in the TESTDEPT table.

TESTEMP.DNO references the DNO column in the TESTEMP table.

## Inner-Join: Getting Started

This is a very important chapter. It lays the foundation for the next five chapters and many other sample queries throughout the remainder of this book.

**Terminology:** Whenever we say “join,” we mean *inner-join* (versus *outer-join*).

This chapter introduces the inner-join operation by presenting sample queries that reference three very similar but different database designs.

**Design-1:** Figure 16.1 will illustrate the most common design scenario. It consists of two tables, DEPARTMENT and EMPLOYEE, where there is a Primary-Key - Foreign-Key (PK-FK) relationship between the tables, and the foreign-key is declared to be NOT NULL.

**Design-2:** Figure 16.2 illustrates another common design scenario that is almost identical to Design-1. It consists of two tables, DEPARTMENT and EMPLOYEE2, where there is a PK-FK relationship between the tables. The only difference is that the foreign-key column is allowed to contain null values.

**Design-3:** Figure 16.3 will present a realistic but less common design scenario. It consists of two tables, DEPARTMENT and EMPLOYEE3 where there is no PK-FK relationship between the tables.



## Design-1: PK-FK Relationship (FK is NOT NULL)

Design-1 is implemented by executing the following CREATE TABLE statements. The first seven sample queries will reference these tables.

```
DROP TABLE EMPLOYEE;
DROP TABLE DEPARTMENT;

CREATE TABLE DEPARTMENT
(DNO      CHAR (2)      NOT NULL PRIMARY KEY,
 DNAME   VARCHAR(20)   NOT NULL UNIQUE,
 BUDGET  DECIMAL(9,2)  NOT NULL);

CREATE TABLE EMPLOYEE
(ENO      CHAR (4)      NOT NULL PRIMARY KEY,
 ENAME   VARCHAR(25)   NOT NULL,
 SALARY  DECIMAL(7,2)  NOT NULL,
 DNO     INTEGER       NOT NULL,
 FOREIGN KEY (DNO) REFERENCES DEPARTMENT);
```

Figure 16.1: Design-1

We highlight the PK-FK relationship between DEPARTMENT and EMPLOYEE. It is important to observe that the foreign-key (DNO) in the DEPARTMENT table is declared as NOT NULL. Sample data for these tables are illustrated below in Figure 16.1a. Observe that the EMPLOYEE.DNO values satisfy the foreign-key constraints.

<u>DEPARTMENT</u>		
DNO	DNAME	BUDGET
10	ACCOUNTING	75000.00
20	INFO. SYS.	20000.00
30	PRODUCTION	7000.00
40	ENGINEERING	25000.00

<u>EMPLOYEE</u>			
ENO	ENAME	SALARY	DNO
1000	MOE	2000.00	20
2000	LARRY	2000.00	10
3000	CURLY	3000.00	20
4000	SHEMP	500.00	40
5000	JOE	400.00	10
6000	GEORGE	9000.00	20

Figure 16.1a: Sample Data for Design-1

**Join DEPARTMENT and EMPLOYEE:** The inner-join operation matches and merges rows from each table by comparing values from designated columns. The following result was generated by comparing the DNO values in the DEPARTMENT and EMPLOYEE tables. It is important to note that *only rows with matching DNO values appear in this result*. In particular, observe that the DEPARTMENT row with the DNO value of 30 does not appear in this result.

ENO	ENAME	SALARY	DNO	DNO1	DNAME	BUDGET
1000	MOE	2000.00	20	20	INFO. SYS.	20000.00
2000	LARRY	2000.00	10	10	ACCOUNTING	75000.00
3000	CURLY	3000.00	20	20	INFO. SYS.	20000.00
4000	SHEMP	500.00	40	40	ENGINEERING	25000.00
5000	JOE	400.00	10	10	ACCOUNTING	75000.00
6000	GEORGE	9000.00	20	20	INFO. SYS.	20000.00

A very popular way (*not the only way*) to code an inner-join is described below.

**FROM-Clause:** The FROM-clause must reference both tables. Either table can be specified first. A comma must separate the table-names.

**FROM EMPLOYEE, DEPARTMENT**

**WHERE-Clause:** When joining the DEPARTMENT and EMPLOYEE tables, we compare the primary-key column (DNO) in the parent-table (DEPARTMENT) with a foreign-key column (DNO) in the child-table (EMPLOYEE) by coding a WHERE-clause. This WHERE-clause specifies the "*join-condition*." The following join-condition states that result table should only contain rows with matching DEPARTMENT.DNO and EMPLOYEE.DNO values.

**WHERE EMPLOYEE.DNO = DEPARTMENT.DNO**

In this chapter, the first seven sample queries contain the same FROM and WHERE clauses that look like:

```
SELECT . . .
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DNO = DEPARTMENT.DNO
```

## Inner-Join

The following sample query illustrates an inner-join operation where the join-condition is based on a PK-FK relationship.

**Sample Query 16.1:** Display all information about every employee, followed by all information about the employee's department. (I.e., Join the EMPLOYEE and DEPARTMENT tables by matching on their DNO values.)

```
SELECT *
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DNO = DEPARTMENT.DNO
```

ENO	ENAME	SALARY	DNO	DNO1	DNAME	BUDGET
1000	MOE	2000.00	20	20	INFO. SYS.	20000.00
2000	LARRY	2000.00	10	10	ACCOUNTING	75000.00
3000	CURLY	3000.00	20	20	INFO. SYS.	20000.00
4000	SHEMP	500.00	40	40	ENGINEERING	25000.00
5000	JOE	400.00	10	10	ACCOUNTING	75000.00
6000	GEORGE	9000.00	20	20	INFO. SYS.	20000.00

**Syntax:** The basic syntax of the inner-join operation is:

```
SELECT column(s)
FROM TABLE1, TABLE2
WHERE TABLE1.COLX = TABLE2.COLY
```

**FROM-clause:** Both table-names are specified in the FROM-clause, separated by a comma.

**WHERE-clause:** This WHERE-clause specifies the join-condition. Here, the comparison operator is the equals (=) symbol. This precise name for this kind of join is "*inner equijoin*."

**SELECT-clause:** This SELECT-clause does not specify column-names. Therefore, all EMPLOYEE columns are displayed to the left of all DEPARTMENT columns because EMPLOYEE is designated first in the FROM-clause. As mentioned on the previous page, table-names can be specified in any sequence.

**Logic:** Again, we emphasize that, when you execute an inner-join, *only matching rows appear in the result*. Observe that:

The row for DEPARTMENT 20 matches three EMPLOYEE rows. (Department 20 has three employees.) In general, a row in a parent-table (DEPARTMENT) may match with none, one, or many child-table (EMPLOYEE) rows.

Because Department 30 does not have any employees, the result table does not contain any data for this department.

**Redundant Columns in Result Table:** This result table contains two DNO columns which have identical values. To avoid potential ambiguity associated with displaying two columns with the same name (DNO), your front-end tool may assign some other name to one of the DNO columns. For example, DB2 might show DNO1 as the column heading for the second DNO column. Your front-end tool may assign a different heading.

Most users consider this column redundancy to be undesirable. The following Sample Query 16.2 will display just one DNO column.

**Data-Type Compatibility:** The columns being compared must be "data-type compatible." This means that you must compare numbers to numbers, character-strings to character-strings, and date-time values to date-time values. For example, you could compare INTEGER to DECIMAL, DECIMAL (5,2) to DECIMAL (7,3), CHAR to VARCHAR, etc. With these comparisons, the system must perform some internal data-type transformations. It is better to compare columns that have the exact same data-type. (For this reason, database designers almost always designate each foreign-key column to have the exact same data-type and length as its corresponding primary-key column.)

**Size of Result Table:** Observe that the EMPLOYEE table has six rows, and the result table also has six rows. In Design-1, when you join the two tables based upon their PK-FK relationship, the number of rows in the join-result equals the number of rows in the child-table (EMPLOYEE).

## Natural-Join

The result table for Sample Query 16.1 displayed all columns from both the DEPARTMENT and EMPLOYEE tables. Notice that both of the join-columns have the same name (DNO). Also, notice that the result table shows that the DNO values (in columns DNO and DNO1) contain identical values. Because redundant data can be confusing, we *naturally* prefer to avoid it. The following sample query eliminates this kind of column redundancy.

**Sample Query 16.2:** Display the ENAME, ENO, and SALARY values for all employees followed by the DNO, DNAME, and BUDGET values of the departments they work in.

```
SELECT ENAME, ENO, SALARY,
       DEPARTMENT.DNO, DNAME, BUDGET

FROM EMPLOYEE, DEPARTMENT

WHERE EMPLOYEE.DNO = DEPARTMENT.DNO
```

ENAME	ENO	SALARY	DNO	DNAME	BUDGET
MOE	1000	2000.00	20	INFO. SYS.	20000.00
LARRY	2000	2000.00	10	ACCOUNTING	75000.00
CURLY	3000	3000.00	20	INFO. SYS.	20000.00
SHEMP	4000	500.00	40	ENGINEERING	25000.00
JOE	5000	400.00	10	ACCOUNTING	75000.00
GEORGE	6000	9000.00	20	INFO. SYS.	20000.00

**Syntax & Logic:** Nothing new. This statement contains the same FROM-clause and WHERE-clause shown in the previous Sample Query 16.1.

```
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DNO = DEPARTMENT.DNO
```

Only the SELECT-clause is different. It specifies just one DNO column, the DEPARTMENT.DNO column. Alternatively, the EMPLOYEE.DNO column could have been specified.

**Terminology:** “*Natural-join*” is the precise name for this kind of join. However, this term is rarely used within the community of practitioners. (Also, some systems support a NATURAL JOIN clause that is not described in this book.)

## Join with ORDER BY

Previous result tables were not explicitly sorted. (The result may have been incidentally sorted because of some internal processing by the system.) Appending an ORDER BY clause allows you to display the result table in some desired row sequence.

**Sample Query 16.3:** Enhance the previous Sample Query 16.2. Display the same result sorted by employee name.

```
SELECT ENAME, ENO, SALARY,
       DEPARTMENT.DNO, DNAME, BUDGET
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DNO = DEPARTMENT.DNO
ORDER BY ENAME
```

<u>ENAME</u>	<u>ENO</u>	<u>SALARY</u>	<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>
CURLY	3000	3000.00	20	INFO. SYS.	20000.00
GEORGE	6000	9000.00	20	INFO. SYS.	20000.00
JOE	5000	400.00	10	ACCOUNTING	75000.00
LARRY	2000	2000.00	10	ACCOUNTING	75000.00
MOE	1000	2000.00	20	INFO. SYS.	20000.00
SHEMP	4000	500.00	40	ENGINEERING	25000.00

**Syntax and Logic:** Nothing new.

## Common Error!

Assume the query objective is the same as Sample Query 16.1 which required an inner-join of the EMPLOYEE and DEPARTMENT tables. However, also assume that you forget to code a WHERE-clause to specify the join-condition. You *incorrectly* execute:

```
SELECT *  
  
FROM DEPARTMENT, EMPLOYEE
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
30	PRODUCTION	80000.00	1000	MOE	2000.00	20
40	ENGINEERING	25000.00	1000	MOE	2000.00	20
10	ACCOUNTING	75000.00	2000	LARRY	2000.00	10
20	INFO. SYS.	20000.00	2000	LARRY	2000.00	10
30	PRODUCTION	80000.00	2000	LARRY	2000.00	10
40	ENGINEERING	25000.00	2000	LARRY	2000.00	10
10	ACCOUNTING	75000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
30	PRODUCTION	80000.00	3000	CURLY	3000.00	20
40	ENGINEERING	25000.00	3000	CURLY	3000.00	20
10	ACCOUNTING	75000.00	4000	SHEMP	500.00	40
20	INFO. SYS.	20000.00	4000	SHEMP	500.00	40
30	PRODUCTION	80000.00	4000	SHEMP	500.00	40
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	5000	JOE	400.00	10
30	PRODUCTION	80000.00	5000	JOE	400.00	10
40	ENGINEERING	25000.00	5000	JOE	400.00	10
10	ACCOUNTING	75000.00	6000	GEORGE	9000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	80000.00	6000	GEORGE	9000.00	20
40	ENGINEERING	25000.00	6000	GEORGE	9000.00	20

**Logic:** *If you fail to specify a join-condition, the system will match every row in the first table with every row in the second table.* The above result table shows that the system matched the four DEPARTMENT rows with the six EMPLOYEE rows to produce a result table of 24 (4x6) rows. This result does not conform to our query objective. A join-condition is necessary to specify the matching criteria. (We will revisit this SELECT statement in Sample Query 17.1.)

## Subset of Join-Result

The following sample query displays a subset of rows and columns from an intermediate result table produced by an inner-join operation.

**Sample Query 16.4:** For each employee earning less than \$3,000.00, display the name of the employee's department followed by the employee's name and salary. Sort the result by employee names within department names.

```
SELECT DNAME, ENAME, SALARY
FROM   EMPLOYEE, DEPARTMENT
WHERE  EMPLOYEE.DNO = DEPARTMENT.DNO
AND    SALARY < 3000.00
ORDER BY DNAME, ENAME
```

<u>DNAME</u>	<u>ENAME</u>	<u>SALARY</u>
ACCOUNTING	JOE	400.00
ACCOUNTING	LARRY	2000.00
ENGINEERING	SHEMP	500.00
INFO. SYS.	MOE	2000.00

**Syntax:** Nothing new. The SALARY < 3000.00 condition is AND-connected to the join-condition.

**Logic:** *Conceptually:*

1. Code an inner-join to produce an intermediate join-result.
2. Append an AND-condition to select some subset of rows from the intermediate join-result.
3. Select the desired columns by specifying the column-names in the SELECT-clause.
4. Specify an ORDER BY clause to display the final result in the desired row sequence.



**Sample Query 16.5:** Only consider employees having a salary that is less than \$999.00. If any such employee works for a department having a budget that is less than or equal to \$75,000.00, display the department's DNO and BUDGET values along with the employee's ENAME and SALARY values. Sort the result by the DNO column.

```
SELECT DEPARTMENT.DNO, BUDGET, ENAME, SALARY
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DNO = DEPARTMENT.DNO
AND SALARY < 999.00
AND BUDGET <= 75000.00
ORDER BY DEPARTMENT.DNO
```

DNO	BUDGET	ENAME	SALARY
10	75000.00	JOE	400.00
40	25000.00	SHEMP	500.00

**Logic:** *Conceptually:*

1. Code an inner-join to produce an intermediate join-result.
2. Append two AND-conditions to select some subset of rows from the intermediate join-result.
3. Select the desired columns by specifying the column-names in the SELECT-clause.
4. Specify an ORDER BY clause to display the final result in the desired row sequence.

The following page presents a procedure (Procedure-1) which adds more detail to this logic. We also present another procedure (Procedure-2) which follows a different sequence of steps to satisfy the same query objective.

**Procedure-1:** This following sequence of operations corresponds to the previously described logical procedure.

1. Produce an intermediate result (IR1) by joining the EMPLOYEE and DEPARTMENT.

```
IR1 ← SELECT * FROM EMPLOYEE, DEPARTMENT
        WHERE EMPLOYEE.DNO = DEPARTMENT.DNO
```

2. Produce a second intermediate result (IR2) by extracting the desired rows and columns from IR1.

```
IR2 ← SELECT DNO, BUDGET, ENAME, SALARY FROM IR1
        WHERE SALARY < 999.00 AND BUDGET <=75000.00
```

3. Produce the final result by sorting IR2.

**Procedure-2:** Alternatively, a different *logical* sequence of operations yields the same result.

1. Produce intermediate result (IRA) by extracting the necessary rows and columns from EMPLOYEE.

```
IRA ← SELECT DNO, ENAME, SALARY FROM EMPLOYEE
        WHERE SALARY < 999.00
```

2. Produce a second intermediate-result (IRB) by extracting the necessary rows and columns from DEPARTMENT.

```
IRB ← SELECT DNO, BUDGET FROM DEPARTMENT
        WHERE BUDGET <=75000.00
```

3. Join IRA and IRB to form IRC.

```
IRC ← SELECT IRA.DNO, BUDGET, ENAME, SALARY
        FROM IRA, IRB
        WHERE IRA.DNO = IRB.DNO
```

4. Produce the final result by sorting IRC.

Most users will conceptualize this SELECT statement in terms of Procedure-1. However, a few users will conceptualize it in terms of Procedure-2. Both are correct. Our commentary on future sample queries involving join-operations will follow Procedure-1.

## Join Two Tables – Only Display Columns from the Child-Table

Sample Queries 16.1-16.5 joined the DEPARTMENT and EMPLOYEE tables and displayed columns from both tables. Occasionally, your query objective requires that you specify a join-operation, but you only want to display data from one table. The next example joins the EMPLOYEE and DEPARTMENT tables. But the SELECT-clause only displays columns from one table, the child-table (EMPLOYEE).

**Sample Query 16.6:** Display the employee number and name of any employee who works for a department having a budget that is greater than or equal to \$25,000.00

```
SELECT ENO, ENAME
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DNO = DEPARTMENT.DNO
AND BUDGET >= 25000.00
```

```
ENO ENAME
2000 LARRY
4000 SHEMA
5000 JOE
```

**Logic:** Although only EMPLOYEE columns are displayed, a join is required because the AND-condition (BUDGET >= 25000.00) references the BUDGET column in the DEPARTMENT table.

**Know-Your-Data Observations:** This SELECT-clause only specifies columns from the *child-table*. (This observation becomes very relevant in the following sample query where the SELECT-clause only specifies columns from the parent-table.) We make two observations.

1. All EMPLOYEE rows must be present in the intermediate join-result. Hence, no EMPLOYEE row can disappear because it fails to match on the join-operation.
2. Because each EMPLOYEE row must match exactly one DEPARTMENT row, EMPLOYEE information cannot be duplicated in the join-result.

*Similar observations will not apply to the following sample query.*

## Join Two Tables – Only Display Columns from the Parent-Table

The next query is similar to the preceding example. But an important know-your-data observation requires your attention. Again, we join two tables and then display columns from just one table. However, here we only display columns from DEPARTMENT, the parent-table.

**Sample Query 16.7:** *We are only interested in departments that have at least one employee. Display the DNAME and BUDGET values for any such department with a budget that is less than or equal to \$50,000.00.*

```
SELECT DNAME, BUDGET
FROM   EMPLOYEE, DEPARTMENT
WHERE  EMPLOYEE.DNO = DEPARTMENT.DNO
AND    BUDGET <= 50000.00
```

<u>DNAME</u>	<u>BUDGET</u>
INFO. SYS.	20000.00
INFO. SYS.	20000.00
INFO. SYS.	20000.00
ENGINEERING	25000.00

**Know-Your-Data Observations:** The SELECT-clause only specifies columns from the *parent-table*. We make two observations.

1. One or more parent (DEPARTMENT) rows may not be present in the intermediate join-result. Hence, they cannot appear in the final result. The PRODUCTION Department (Department 30) is missing because it does not have any employees. (Note: This query objective stated “we are only interested in departments that have at least one employee.”)

2. Duplicate rows may appear in the result table for any department with more than one employee. Three duplicate rows for the INFO. SYS. Department (Department 20) appear because this department has three employees. You can remove this duplication by specifying DISTINCT.

```
SELECT DISTINCT DNAME, BUDGET FROM . . .
```

We emphasize that similar observations did not apply to the previous Sample Query 16.6.

## Design-2: Foreign Key Contains Null Values

Design-2 is almost identical to Design-1. Both designs have the same DEPARTMENT table and very similar employee tables (EMPLOYEE and EMPLOYEE2) which have the same columns. Also, EMPLOYEE2 specifies the same foreign-key column (DNO) that references the DEPARTMENT table. The only difference is that the foreign-key column (*EMPLOYEE2.DNO*) is allowed to contain null values. (The specification of the DNO column in the following CREATE TABLE statement for EMPLOYEE2 does not specify a NOT NULL clause.) This design allows you to hire a new employee (insert a new row into EMPLOYEE2) without assigning the new employee to a department.

**Figure 16.2:**  
Design-2

```
{Same DEPARTMENT table}

CREATE TABLE EMPLOYEE2
(ENO      CHAR (4)      NOT NULL,
 ENAME    VARCHAR(25)  NOT NULL,
 SALARY   DECIMAL(7,2) NOT NULL,
 DNO      INTEGER,
 PRIMARY KEY (ENO),
 FOREIGN KEY (DNO) REFERENCES DEPARTMENT);
```

Sample data for DEPARTMENT and EMPLOYEE2 tables are shown below. The DEPARTMENT table is unchanged. The only difference in the data in the EMPLOYEE2 table (versus the EMPLOYEE table) is that the last row describing Employee 6000 has a null DNO value.

**Figure 16.2a:**  
Design-2 Sample Data

<u>DEPARTMENT</u>		
DNO	DNAME	BUDGET
10	ACCOUNTING	75000.00
20	INFO. SYS.	20000.00
30	PRODUCTION	7000.00
40	ENGINEERING	25000.00

<u>EMPLOYEE2</u>			
ENO	ENAME	SALARY	DNO
1000	MOE	2000.00	20
2000	LARRY	2000.00	10
3000	CURLY	3000.00	20
4000	SHEMP	500.00	40
5000	JOE	400.00	10
6000	GEORGE	9000.00	-

## Join DEPARTMENT and EMPLOYEE2

**Sample Query 16.8:** We are only interested in employees who are known to be assigned to some department. Display all information about those employees, along with all information about the departments they work in. (I.e., Join the DEPARTMENT table and the EMPLOYEE2 table based on the PK-FK relationship.)

```
SELECT *  
  
FROM EMPLOYEE2, DEPARTMENT  
  
WHERE EMPLOYEE2.DNO = DEPARTMENT.DNO
```

ENO	ENAME	SALARY	DNO	DNO1	DNAME	BUDGET
1000	MOE	2000.00	20	20	INFO. SYS.	20000.00
2000	LARRY	2000.00	10	10	ACCOUNTING	75000.00
3000	CURLY	3000.00	20	20	INFO. SYS.	20000.00
4000	SHEMP	500.00	40	40	ENGINEERING	25000.00
5000	JOE	400.00	10	10	ACCOUNTING	75000.00

**Syntax:** Nothing new.

**Logic:** Notice that the query objective states "For those employees who are known to be assigned to some department." The *critical observation* is that the result table does not contain any information about Employee 6000 because his DNO value is null. Hence, it cannot match any DNO value in DEPARTMENT.

Again, observe that the result table has no information about the PRODUCTION department (DNO value of 30) which does not have any employees.

**Know-your-data:** The syntax and logic for this SELECT statement is straightforward. (It has the same structure as Sample Query 16.1.) The critical observation pertains to knowing your data. To correctly understand the result table, you must be aware that some EMPLOYEE2 rows could have null DNO values.

### Design-3: No PK-FK Relationship

Design-3 is similar to Design-2 except no foreign-key column is specified. This design scenario is not very common, but it is valid.

Assume the database designer is told that it is possible to hire an employee and assign the employee a DNO value that is not present in the DEPARTMENT table. This assumption leads to Design-3 shown below in Figure 16.3. Note that the DNO column in EMPLOYEE3 is *not* declared as a foreign key. Also note that it may contain null values.

**Figure 16.3:**  
Design-3

```
{Same DEPARTMENT table}

CREATE TABLE EMPLOYEE3
(ENO          CHAR (4)          NOT NULL,
 ENAME        VARCHAR(25)      NOT NULL,
 SALARY       DECIMAL(7,2)     NOT NULL,
 DNO         INTEGER,
 PRIMARY KEY (ENO));
```

Sample data for DEPARTMENT and EMPLOYEE3 tables are shown below. Notice that the first EMPLOYEE3 row for Employee 1000 has a non-null DNO value (99) that is *not* found in the DEPARTMENT table. (Presumably a row describing Department 99 would be inserted into the DEPARTMENT table at some point in the future.)

**Figure 16.3a:**  
Design-3 Sample Data

<u>DEPARTMENT</u>		
DNO	DNAME	BUDGET
10	ACCOUNTING	75000.00
20	INFO. SYS.	20000.00
30	PRODUCTION	7000.00
40	ENGINEERING	25000.00

<u>EMPLOYEE3</u>			
ENO	ENAME	SALARY	DNO
1000	MOE	2000.00	<b>99</b> ←
2000	LARRY	2000.00	10
3000	CURLY	3000.00	20
4000	SHEMP	500.00	40
5000	JOE	400.00	10
6000	GEORGE	9000.00	-

## Join DEPARTMENT and EMPLOYEE3

This example joins the DEPARTMENT and EMPLOYEE3 tables by specifying a join-condition that matches on the DNO columns in both tables. However, this join-condition is *not* based upon a PK-FK relationship. In such circumstances, it is possible that both tables may contain rows that do not match the join-condition.

**Sample Query 16.9:** For each employee who is assigned to some "real" department, display all information about the employee and all information about the department he works in. (I.e., Join the DEPARTMENT and EMPLOYEE3 tables matching on their DNO columns.)

```
SELECT *
FROM   EMPLOYEE3, DEPARTMENT
WHERE  EMPLOYEE3.DNO = DEPARTMENT.DNO
```

ENO	ENAME	SALARY	DNO	DNO1	DNAME	BUDGET
2000	LARRY	2000.00	10	10	ACCOUNTING	75000.00
3000	CURLY	3000.00	20	20	INFO. SYS.	20000.00
4000	SHEMP	500.00	40	40	ENGINEERING	25000.00
5000	JOE	400.00	10	10	ACCOUNTING	75000.00

**Syntax:** Nothing new.

**Logic:** Notice that the result table does not contain rows for Employee 1000 (because its DNO value of 99 does not match) and Employee 6000 (because its DNO value is null).

Again, we observe there is no row corresponding to the PRODUCTION department (DNO value of 30).

**Know-your-data:** You must be aware that: (1) Some EMPLOYEE3.DNO values might not match any DEPARTMENT.DNO value, and (2) some EMPLOYEE3.DNO values may be null.



## Exercises

16A. What result tables are produced by executing the following statements?

- a. 

```
SELECT ENAME, DNAME
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DNO = DEPARTMENT.DNO
AND BUDGET <= 50000.00
ORDER BY ENAME
```
- b. 

```
SELECT ENAME, DNAME
FROM EMPLOYEE2, DEPARTMENT
WHERE EMPLOYEE2.DNO = DEPARTMENT.DNO
AND BUDGET <= 50000.00
```
- c. 

```
SELECT ENAME, DNAME
FROM EMPLOYEE3, DEPARTMENT
WHERE EMPLOYEE3.DNO = DEPARTMENT.DNO
AND BUDGET <= 50000.00
```

16B. What result tables are produced by executing the following statements?

- a. 

```
SELECT DEPARTMENT.DNO, DNAME, ENO, ENAME
FROM DEPARTMENT, EMPLOYEE
WHERE EMPLOYEE.DNO = DEPARTMENT.DNO
AND ENAME NOT LIKE '%O%E%'
ORDER BY DEPARTMENT.DNO, ENAME
```
- b. 

```
SELECT DEPARTMENT.DNO, DNAME, ENO, ENAME
FROM DEPARTMENT, EMPLOYEE2
WHERE EMPLOYEE2.DNO = DEPARTMENT.DNO
AND ENAME NOT LIKE '%O%E%'
ORDER BY DEPARTMENT.DNO, ENAME
```
- c. 

```
SELECT DEPARTMENT.DNO, DNAME, ENO, ENAME
FROM DEPARTMENT, EMPLOYEE3
WHERE EMPLOYEE3.DNO = DEPARTMENT.DNO
AND ENAME NOT LIKE '%O%E%'
ORDER BY DEPARTMENT.DNO, ENAME
```

The next three exercises reference Design-1 (DEPARTMENT and EMPLOYEE tables).

16C. Display every employee's name, salary, and the name of the department he works in. Sort the result table by employee name.

16D. Display the employee number and name of any employee who works for a department having a budget that is greater than \$24,000.00. Sort the result table by employee number.

16E. Display the department numbers and names of all departments that have at least one employee earning a salary that is greater than \$1,000.00. Sort the result table by department numbers.

## Variations in Join Syntax

There are multiple ways to code the same inner-join operation. The following Figure 16.4 shows four equivalent methods for coding a SELECT statement for Sample Query 16.1.

### 1. Old-Syntax (Sample Query 16.1)

```
SELECT *  
  
FROM EMPLOYEE, DEPARTMENT  
  
WHERE EMPLOYEE.DNO = DEPARTMENT.DNO
```

### 2. Old-Syntax with Table Alias

```
SELECT *  
  
FROM EMPLOYEE E, DEPARTMENT D  
  
WHERE E.DNO = D.DNO
```

### 3. JOIN-ON-Syntax

```
SELECT *  
  
FROM EMPLOYEE INNER JOIN DEPARTMENT  
  
ON EMPLOYEE.DNO = DEPARTMENT.DNO
```

### 4. JOIN-ON-Syntax with Table Alias

```
SELECT *  
  
FROM EMPLOYEE E INNER JOIN DEPARTMENT D  
  
ON E.DNO = D.DNO
```

Figure 16.4: Variations in Inner-Join Syntax

1. **Old-Syntax:** Sample Queries 16.1-16.9 demonstrated this syntax. This syntax was part of the first (1970's) version of SQL.
2. **Old-Syntax with Table Alias:** This syntax was also part of the first version of SQL and is very popular. A table is assigned a *table alias* within the FROM-clause to temporarily rename the table. The following FROM clause specifies E as an alias for EMPLOYEE and D as an alias for DEPARTMENT.

```
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.DNO = D.DNO
```

3. **JOIN-ON-Syntax:** In the 1990s, the SQL Standards Committee introduced the JOIN-ON syntax. Today, all major vendors support this syntax. Within the FROM-clause, INNER JOIN is placed between the table-names, and the keyword ON is used to specify the join-condition.

```
FROM EMPLOYEE INNER JOIN DEPARTMENT
ON EMPLOYEE.DNO = DEPARTMENT.DNO
```

4. **JOIN-ON-Syntax with Table Alias:** The JOIN-ON syntax can also specify a table alias.

```
FROM EMPLOYEE E INNER JOIN DEPARTMENT D
ON E.DNO = D.DNO
```

Note: The keyword **INNER** is optional when using the JOIN-ON syntax. For example:

```
FROM EMPLOYEE INNER JOIN DEPARTMENT
```

Can be rewritten as:

```
FROM EMPLOYEE JOIN DEPARTMENT
```

We recommend coding INNER to explicitly distinguish an INNER JOIN from an OUTER JOIN (to be introduced in Chapter 19).

## Examples

The SELECT statement for Sample Query 16.5 is shown below.

```
SELECT DEPARTMENT.DNO, BUDGET, ENAME, SALARY
FROM   EMPLOYEE, DEPARTMENT
WHERE  EMPLOYEE.DNO = DEPARTMENT.DNO
AND    SALARY < 999.00
AND    BUDGET <=75000.00
ORDER BY DEPARTMENT.DNO
```

This statement can be rewritten as:

### Old-Syntax with Table Aliases

```
SELECT D.DNO, D.BUDGET, E.ENAME, E.SALARY
FROM   DEPARTMENT D, EMPLOYEE E
WHERE  D.DNO = E.DNO
AND    D.BUDGET <=75000.00
AND    E.SALARY < 999.00
ORDER BY D.DNO
```

### JOIN-ON Syntax

```
SELECT DEPARTMENT.DNO, DEPARTMENT.BUDGET,
       EMPLOYEE.ENAME, EMPLOYEE.SALARY
FROM   DEPARTMENT INNER JOIN EMPLOYEE
ON    DEPARTMENT.DNO = EMPLOYEE.DNO
WHERE  DEPARTMENT.BUDGET <=75000.00
AND    EMPLOYEE.SALARY < 999.00
ORDER BY DEPARTMENT.DNO
```

### JOIN-ON Syntax with Table Aliases

```
SELECT D.DNO, D.BUDGET, E.ENAME, E.SALARY
FROM   DEPARTMENT D INNER JOIN EMPLOYEE E
ON    D.DNO = E.DNO
WHERE  D.BUDGET <=75000.00
AND    E.SALARY < 999.00
ORDER BY D.DNO
```

## Advantages of JOIN-ON Syntax

The JOIN-ON syntax has two advantages.

First, Chapter 19 will show that the JOIN-ON syntax is used to code outer-join operations.

The second advantage pertains to conceptual tidiness. It is easier to learn a computer language if a given buzzword is associated with just one concept, and vice versa. It can be confusing if the same concept is implemented by multiple buzzwords. Conversely, it can be confusing if the same buzzword is applied to multiple concepts where you must rely on context to deduce the proper meaning. This criticism applies to the keyword WHERE.

- WHERE is used to specify restriction (select some subset of rows).
- WHERE is also used to specify a join-condition.

The JOIN-ON syntax is conceptually cleaner.

- INNER JOIN explicitly indicates that you want to execute an inner-join operation.
- ON is used to specify the join-condition.
- WHERE is used to specify a restriction.

Consider the statement:

```
SELECT DEPARTMENT.DEPT, DEPARTMENT.BUDGET,  
       EMPLOYEE.ENAME, EMPLOYEE.SALARY  
FROM   DEPARTMENT INNER JOIN EMPLOYEE  
ON    DEPARTMENT.DEPT = EMPLOYEE.DEPT } ← join-operation  
WHERE  DEPARTMENT.BUDGET <=75000.00 } ← restriction  
AND    EMPLOYEE.SALARY < 999.00  
ORDER BY DEPARTMENT.DEPT
```

In this example, the WHERE-clause specifies a restriction that is applied to the intermediate result produced by the inner-join operation.

## Exercises

- 16F. Rewrite the following SELECT statement using:
- (a) Alias D for DEPARTMENT and alias E for EMPLOYEE
  - (b) The JOIN-ON syntax without table aliases
  - (c) The JOIN-ON syntax with table aliases

```
SELECT ENO, ENAME, SALARY,  
       DEPARTMENT.DNO, DNAME, BUDGET  
FROM   EMPLOYEE, DEPARTMENT  
WHERE  EMPLOYEE.DNO = DEPARTMENT.DNO
```

- 16G. Rewrite the following SELECT statement using:
- (a) Alias DP for DEPARTMENT and alias EMP for EMPLOYEE
  - (b) The JOIN-ON syntax without table aliases
  - (c) The JOIN-ON syntax with table aliases

```
SELECT DEPARTMENT.DNO, ENAME  
FROM   DEPARTMENT, EMPLOYEE  
WHERE  DEPARTMENT.DNO = EMPLOYEE.DNO  
AND    BUDGET > 21000  
ORDER BY DEPARTMENT.DNO, ENAME
```

## Estimating Size of Join-Result

Before executing a SELECT statement, it can be helpful to have some ballpark estimate of the size of your result table. If, after executing a SELECT statement, the result appears to have too few or too many rows, you should suspect that something *may* be wrong with your logic.

**Example:** Assume       Table T1 has 1,000 rows and  
                          Table T2 has 9,998 rows.

Estimate the number of rows in the result table produced by joining tables T1 and T2.

```
SELECT *
FROM   T1, T2
WHERE  T1.A = T2.B
```

**Scenario-1:** Assume there is *no* PK-FK relationship between T1 and T2. In this circumstance, deriving an accurate estimate may not be possible. Without additional information, the best we can say is:

Smallest size = 0                   (no rows match)  
Largest size  = 9,998,000 (every row matches every row)

**Scenario-2 (PK-FK):** Assume there is a PK-FK relationship where Column T1.A is a primary-key column that is referenced by the non-null foreign-key column, T2.B. In this circumstance, we can conclude that:

*The result table has exactly 9,998 rows.*

This 9,998 rows corresponds to the number of rows in T2, the child-table. Recall that each row in the child-table (T2) must match exactly one row in the parent-table (T1).

**Conclusion:** Knowledge of a PK-FK relationships can help you estimate the number of rows in an inner-join result.



## Inner-Join and Restriction

Sample Query 16.5 noted that the same result is returned if  
 (1) the inner-join is executed before restriction, or if  
 (2) restriction is executed before the inner-join. Consider the following tables and SELECT statement.

<u>MAN</u>		<u>DOG</u>		
<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO</u>
77	MOE	1000	SPOT	99
88	LARRY	3000	ROVER	77
99	CURLY	2000	WALLY	99
		4000	SPIKE	-

```
SELECT *
FROM MAN INNER JOIN DOG ON MAN.MNO = DOG.MNO
WHERE DOG.DNAME LIKE 'S%'
```

The following examples illustrate that an inner-join operation can be executed either before or after the restriction. Both examples produce the same one-row result.

**First Join,  
Then Restrict**

1. MAN INNER JOIN DOG ON MAN.MNO = DOG.MNO

<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO1</u>
77	MOE	3000	ROVER	77
99	CURLY	1000	SPOT	99
99	CURLY	2000	WALLY	99

2. WHERE DOG.DNAME LIKE 'S%'

<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO1</u>
99	CURLY	1000	SPOT	99

**First Restrict,  
Then Join**

1. WHERE DOG.DNAME LIKE 'S%'

<u>MAN</u>		<u>DOG</u>		
<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO</u>
77	MOE	1000	SPOT	99
88	LARRY	4000	SPIKE	-
99	CURLY			

2. MAN INNER JOIN DOG ON MAN.MNO = DOG.MNO

<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO1</u>
99	CURLY	1000	SPOT	99

## INNER JOIN-ON Syntax: WHERE versus AND

This page makes a simple observation. If you are coding an inner-join operation using the JOIN-ON syntax, you can substitute the keyword AND for WHERE as illustrated by the following two statements. *Both of these statements produce the same result table.*

### Statement-1

```
SELECT *
FROM DEPARTMENT D INNER JOIN EMPLOYEE E
ON D.DNO = E.DNO
→ WHERE E.SALARY < 1000.00
```

### Statement-2

```
SELECT *
FROM DEPARTMENT D INNER JOIN EMPLOYEE E
ON D.DNO = E.DNO
→ AND E.SALARY < 1000.00
```

Ignoring the SELECT-clause, note that Statement-1 specifies two operations: (i) an inner-join and (ii) a restriction.

```
FROM DEPARTMENT D INNER JOIN EMPLOYEE E } ← Inner-join
ON D.DNO = E.DNO
WHERE E.SALARY < 1000.00 } ← Restriction
```

However, notice that Statement-2 only specifies one operation, an inner-join with a compound join-condition. (To emphasize this point we move "**AND** E.SALARY < 1000.00" up to the end of the ON-clause.)

```
FROM DEPARTMENT D INNER JOIN EMPLOYEE E } ← Inner-join
ON D.DNO = E.DNO AND E.SALARY < 1000.00 }
```

Again, Statement-1 and Statement-2 produce the same result. (The author believes that Statement-1 may be friendlier, but this is a personal bias.)

\*\*\* This substitution of AND for WHERE has been presented as a mere "mechanical rewrite" of a SELECT statement. However, this issue requires more explanation that is delayed until Chapter 19. There we will see that, *when specifying an outer-join, you cannot arbitrarily replace WHERE with AND.*

## Naming Foreign-Key Columns

The EMPLOYEE table has a foreign-key column (DNO) with the same name as the corresponding primary-key column (DNO) in the DEPARTMENT table. Many database designers follow this naming pattern. **However, corresponding primary-key and foreign-key columns are not required to have the same name.** Consider the following CREATE table statement for the EMPLOYEE4 table.

```
CREATE TABLE EMPLOYEE4
(ENO      CHAR (4)      NOT NULL PRIMARY KEY,
 ENAME    VARCHAR(25)   NOT NULL,
 SALARY   DECIMAL(7,2) NOT NULL,
 DEPTNUM INTEGER       NOT NULL,
 FOREIGN KEY (DEPTNUM) REFERENCES DEPARTMENT);
```

This table has the same structure as the EMPLOYEE table with one difference. The fourth column is named DEPTNUM. This means that coding a join of DEPARTMENT and EMPLOYEE requires a change in the join-condition as illustrated by the following equivalent statements.

```
SELECT *
FROM DEPARTMENT, EMPLOYEE
WHERE DEPARTMENT.DNO = EMPLOYEE.DEPTNUM;
```

```
SELECT *
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.DNO = E.DEPTNUM;
```

```
SELECT *
FROM DEPARTMENT, EMPLOYEE
WHERE DNO = DEPTNUM;
```

Notice, the last statement does not need to specify table aliases because the PK-FK column-names are different. Someone might consider this to be an advantage, but most users prefer PK-FK columns to have the same name.

## Common Conceptual Error

Consider the following result table.

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>	<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>
10	ACCOUNTING	75000.00	2000	LARRY	2000.00
10	ACCOUNTING	75000.00	5000	JOE	400.00
20	INFO. SYS.	20000.00	1000	MOE	2000.00
20	INFO. SYS.	20000.00	3000	CURLY	3000.00
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00
40	ENGINEERING	25000.00	4000	SHEMP	500.00

Do *not* attempt to modify a SELECT statement to display the above result table so that it looks like the following report.

<u>10</u>	<u>ACCOUNTING</u>	75000.00			
	2000 LARRY	2000.00			
	5000 JOE	400.00			
<u>20</u>	<u>INFO. SYS.</u>	20000.00			
	1000 MOE	2000.00			
	3000 CURLY	3000.00			
	6000 GEORGE	9000.00			
<u>40</u>	<u>ENGINEERING</u>	25000.00			
	4000 SHEM	500.00			

This is a report formatting issue. As previously stated, formatting issues should not influence the coding of a SELECT statement. Your front-end query/reporting tool should be able to transform the above tabular result into some desired report-format.

## Summary

This chapter introduced the inner-join operation. The basic logic for this operation is straightforward. However, perhaps unfortunately, there are many different ways to code an inner-join operation.

This chapter presented three similar, but different versions of a table about employees (EMPLOYEE, EMPLOYEE2, and EMPLOYEE3). After joining each of these tables with the DEPARTMENT table, we observed small differences in the result table. Again, know-your-data!

## Summary Exercises

These exercises reference Design-1 (DEPARTMENT and EMPLOYEE tables). Do not display duplicate rows in any result table. Produce two solutions using the FROM-WHERE syntax and the JOIN-ON syntax. Both solutions should specify table aliases.

16H. Display every employee's number, department number, and the name of the department he works in. Sort the result table by employee number. The result should look like:

<u>ENO</u>	<u>DNO</u>	<u>DNAME</u>
1000	20	INFO. SYS.
2000	10	ACCOUNTING
3000	20	INFO. SYS.
4000	40	ENGINEERING
5000	10	ACCOUNTING
6000	20	INFO. SYS.

16I. Display the employee name and salary of any employee who works for a department having a budget that is less than \$25,000.00. Sort the result table by employee name. The result should look like:

<u>ENAME</u>	<u>SALARY</u>
CURLY	3000.00
GEORGE	9000.00
MOE	2000.00

16J. Display the department numbers and budgets of all departments that have at least one employee earning a salary that is greater than \$1,000.00. Sort the result table by department number. The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>
10	ACCOUNTING	75000.00
40	ENGINEERING	25000.00

## **Appendix 16A: De-Normalized Tables**

Thus far, every table presented in this book has been "normalized." This means the table's design is "good" according to criteria to be (informally) described below. A "de-normalized" table is a table that is not normalized, and therefore may be considered to be "not-so-good."

This appendix presents a short, informal, and very practical look at one de-normalized table, DNEMPLOYEE, illustrated below.

```
CREATE TABLE DNEMPLOYEE
(ENO      CHAR (4)      NOT NULL PRIMARY KEY,
 ENAME    VARCHAR (25)  NOT NULL,
 SALARY   DECIMAL (7,2) NOT NULL,
 DNO      CHAR (2)      NOT NULL,
 DNAME    VARCHAR (20)  NOT NULL,
 BUDGET   DECIMAL (9,2) NOT NULL);
```

Figure A16.1: DNEMPLOYEE – De-Normalized Table

Sample data for DNEMPLOYEE are shown below.

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>
1000	MOE	2000.00	20	INFO. SYS.	20000.00
2000	LARRY	2000.00	10	ACCOUNTING	75000.00
3000	CURLY	3000.00	20	INFO. SYS.	20000.00
4000	SHEMP	500.00	40	ENGINEERING	25000.00
5000	JOE	400.00	10	ACCOUNTING	75000.00
6000	GEORGE	9000.00	20	INFO. SYS.	20000.00

Figure A16.2: Sample Data for DNEMPLOYEE

## Normalized Tables

Appendix 13B introduced some basic concepts of database analysis and design. This process almost always produces multiple tables because database analysis almost always discovers multiple types of objects (entities). For example, the Appendix 13B case study produced four tables (Figure 13.12), including the DEPARTMENT and EMPLOYEE tables. We will note that both the DEPARTMENT and EMPLOYEE tables are normalized (i.e., good). We also note that DNEMPLOYEE is de-normalized (i.e., not-so-good).

Casually speaking, *all columns in a normalized table describe just one type of object (entity)*. Notice that all DEPARTMENT columns only describe departments; and, all EMPLOYEE columns only describe employees.

## De-Normalized Tables

Joining two tables requires some query analysis, and the system might have to do a lot of work to join large tables. For these reasons, a reasonable person might ask: *Why not store data about departments and employees in a single table (DNEMPLOYEE)?* Maybe the DBA should “pre-join” the DEPARTMENT and EMPLOYEE tables, store the result in the DNEMPLOYEE table, and allow users access to this table.

**Advantages of DNEMPLOYEE:** Many (but not all) SELECT statements that reference the DNEMPLOYEE table will be “*fast and friendly*.” For example, Sample Query 16.2 could be coded as:

```
SELECT * FROM DNEMPLOYEE
```

Coding this query is simpler because you do not have to code a join-operation; and, this query executes faster because the system does not have to execute a potentially costly join-operation. Based upon these observations, creating DNEMPLOYEE appears to be a good idea. However, it might be a very bad idea. The following pages address this issue by presenting an *informal and incomplete* description of normalized versus de-normalized tables.

We begin by making three observations (not definitions) about de-normalized tables.

- Within a de-normalized table, columns may describe different types of objects. For example, within DNEmployee, the SALARY column describes employees, and the BUDGET column describes departments.
- As previous noted, a de-normalized table may be produced by "pre-joining" two or more tables.
- As with any join-result, a de-normalized table may have considerable redundancy. For example, within DNEmployee, the same DNAME and BUDGET values for Department 20 appear in three rows.

**DML Disadvantages of DNEmployee:** Although queries against DNEmployee may be "fast and friendly," there are two significant disadvantages associated with the DNEmployee table.

1. **"Lost Information:"** Examination of the DNEmployee table (Figure 16A.2) shows that we have lost information about Department 30. The row with a DNO value of 30 did not match any EMPLOYEE row during the join-operation that produced the DNEmployee table. The DBA might not tell users about the underlying DEPARTMENT table. However, assume the DBA allowed users access to the DEPARTMENT, EMPLOYEE, and DNEmployee tables. This could work. However, users would then encounter a potential know-your-data problem associated with finding the same employee data in two tables (EMPLOYEE and DNEmployee).
2. **Problems with DML Operations:** The following page describes potential problems associated with the INSERT, UPDATE, and DELETE statements.



## Problems with DML Operations on De-Normalized Tables

### INSERT Rows in DNEmployee

Note that ENO would be designated as the primary key of DNEmployee implying that every INSERT operation must specify some non-null ENO value. Therefore, a row cannot be inserted into DNEmployee describing a new department that does not have at least one employee. (This is why information about Department 30 does not appear in the DNEmployee table.)

### DELETE Rows from DNEmployee

Assume all employees in Department 20 quit, and you execute the following statement:

```
DELETE
FROM DNEmployee
WHERE DNO = 20
```

This statement will successfully delete all employees in Department 20. However, you will also lose information about Department 20! The DNAME (INFO. SYS.) and BUDGET (20000.00) values are lost.

### UPDATE Rows in DNEmployee

Assume you want to change some DNO, DNAME, or BUDGET values in DNEmployee. These columns contain redundant values for any department that has multiple employees. For example, if you want to change the BUDGET value for Department 20, you will have to update three rows. This may not be a problem if you execute an UPDATE statement with "WHERE DNO = 20". However, this does not help an applications developer who executes an UPDATE-Cursor operation within embedded-SQL (a topic not covered in this book).

*We emphasize that these DML problems do not apply to the normalized DEPARTMENT and EMPLOYEE tables.*

**Conclusion:** You must know-your-data if you are asked to modify data in a de-normalized table.

**Comment:** Despite these problems, many real-world systems contain a few de-normalized tables. Also, some specialized applications, especially data warehouse applications, will contain many de-normalized tables.

# 17

## More about Inner-Join

This chapter's sample queries do not introduce any new syntax or logic. Instead, these sample queries integrate two-table inner-join operations with SQL concepts that were presented earlier in this book. Sample queries will address the following topics:

- Cross Product Operations
- Arithmetic Expressions with Join
- Aggregate Functions with Join
- Grouping with Join
- Error: Accidental Cross Product
- Joining on Non-Key Columns
- Theta-Joins
- Arithmetic Expressions in Join-Conditions
- Compound Join-Conditions
- Joining a Table with Itself

SELECT statements presented in this chapter will specify inner-join operations using the FROM-WHERE syntax with table aliases.

## Cross Product

Before continuing our discussion of the inner-join, we discuss an operation called the "Cross Product" (or "Cartesian Product"). This operation can be considered to be a special variation of an inner-join. (Appendix 17B presents a more formal observation about the relationship between inner-join and cross product.)

The cross product matches every row in the first table with every row in the second table. The following statement produces the cross product of the DEPARTMENT and EMPLOYEE tables.

```
SELECT *  
FROM DEPARTMENT, EMPLOYEE
```

This statement was introduced as a "common error" in the preceding chapter. In that chapter, our intention was to emphasize the importance of the join-condition. The above SELECT statement was considered to be an error because, by omitting the join-condition, it failed to satisfy the query objective. However, sometimes (but not very often) you might wish to produce a cross product as an intermediate or final result.

**Sample Query 17.1:** Display the cross product of the DEPARTMENT and EMPLOYEE tables.

```
SELECT *  
FROM DEPARTMENT, EMPLOYEE
```

{Result table on following page.}

**Syntax:** Observe the absence of a WHERE-clause.

**Logic:** Every row in the DEPARTMENT table matches with every row in the EMPLOYEE table.

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
30	PRODUCTION	80000.00	1000	MOE	2000.00	20
40	ENGINEERING	25000.00	1000	MOE	2000.00	20
10	ACCOUNTING	75000.00	2000	LARRY	2000.00	10
20	INFO. SYS.	20000.00	2000	LARRY	2000.00	10
30	PRODUCTION	80000.00	2000	LARRY	2000.00	10
40	ENGINEERING	25000.00	2000	LARRY	2000.00	10
10	ACCOUNTING	75000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
30	PRODUCTION	80000.00	3000	CURLY	3000.00	20
40	ENGINEERING	25000.00	3000	CURLY	3000.00	20
10	ACCOUNTING	75000.00	4000	SHEMP	500.00	40
20	INFO. SYS.	20000.00	4000	SHEMP	500.00	40
30	PRODUCTION	80000.00	4000	SHEMP	500.00	40
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	5000	JOE	400.00	10
30	PRODUCTION	80000.00	5000	JOE	400.00	10
40	ENGINEERING	25000.00	5000	JOE	400.00	10
10	ACCOUNTING	75000.00	6000	GEORGE	9000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	80000.00	6000	GEORGE	9000.00	20
40	ENGINEERING	25000.00	6000	GEORGE	9000.00	20

**Size of Result Table:** The DEPARTMENT table has 4 rows, and the EMPLOYEE table has 6 rows. Hence, this cross product will have 24 (4x6) rows. In some circumstances, a cross product might produce a very large result. For example, if both tables contain a million rows, the result table will contain a trillion rows!

**Careful!** If you intend to code a join, but accidentally code a cross product, an unexpectedly large final result may indicate that something is wrong. However, if the cross product is an intermediate result, it may be difficult to make this observation. Sample Query 17.5 will illustrate this kind of error.

## Arithmetic Expression with Join

Arithmetic expressions were introduced in Chapter 7. This sample query shows that an arithmetic expression can reference columns from different tables after the tables have been joined to form an intermediate join-result.

**Sample Query 17.2:** For all employees, display their ENO and SALARY values along with the DNO and BUDGET values of the department they work for. Also, display the ratio of each employee's salary to his department's budget.

```
SELECT E.ENO, E.SALARY, D.DNO, D.BUDGET,  
       E.SALARY/D.BUDGET  
FROM   DEPARTMENT D, EMPLOYEE E  
WHERE  D.DNO = E.DNO
```

ENO	SALARY	DNO	BUDGET	SALARY/BUDGET
1000	2000.00	20	20000.00	0.10000
2000	2000.00	10	75000.00	0.02666
3000	3000.00	20	20000.00	0.15000
4000	500.00	40	25000.00	0.02000
5000	400.00	10	75000.00	0.00533
6000	9000.00	20	20000.00	0.45000

**Syntax:** Nothing new.

**Logic:** The join-operation produces an intermediate join-result where each row contains a SALARY value and BUDGET value. The arithmetic expression (E.SALARY/D.BUDGET) is evaluated, and the desired columns are displayed.

**Exercise:**

17A. Modify this Sample Query 17.2 to express each ratio as a percentage.

## Aggregate Functions with Join

Functions can be specified in statements that involve join-operations. The tables are joined to form an intermediate join-result before the functions operate on the data.

**Sample Query 17.3.1:** Display the total salary of those employees who work for a department with a budget that is less than or equal to \$50,000.00.

```
SELECT SUM (E.SALARY)

FROM   DEPARTMENT D, EMPLOYEE E

WHERE  D.DNO = E.DNO

AND    D.BUDGET <= 50000.00
```

```
SUM (E.SALARY)
14500.00
```

**Logic:** We join the DEPARTMENT and EMPLOYEE tables because we want to summarize SALARY values (from the EMPLOYEE table) and specify a restriction on the BUDGET column (from the DEPARTMENT table). The join and restriction operations produce an intermediate result. The SUM function is applied to the SALARY column in this intermediate result.

**Cautionary Question and a Suggestion:** Does your intuition tell you that the final result is a reasonable value? If you cannot provide an affirmative answer to this question, you might consider executing a preliminary statement similar to the following statement that displays relevant intermediate result data.

```
SELECT E.ENO, E.SALARY
FROM   DEPARTMENT D, EMPLOYEE E
WHERE  D.DNO = E.DNO
AND    D.BUDGET <= 50000.00
```

### Exercise:

17B. Display the average salary of employees who work for a department with a budget that is greater than \$20,000.00.

## Careful! Summarizing over Columns from a Parent-Table

The following sample query has a query objective that is very similar to the previous sample query. Here, the result *"happens to be" correct, by "good luck."* To understand this situation, note that the previous Sample Query 17.3.1 applied the SUM function to the SALARY column, a column in the child-table. The following sample query applies the SUM function to the BUDGET column, a column in the parent-table.

**Sample Query 17.3.2:** Only consider departments that have at least one employee and have a budget that is less than or equal to \$50,000.00. Display the summary total budget for these departments. [The following SELECT statement is *"almost correct" (i.e., wrong)*. Try to detect the error before reading the commentary.]

```
SELECT SUM (DISTINCT D.BUDGET)
FROM   DEPARTMENT D, EMPLOYEE E
WHERE  D.DNO = E.DNO
AND    D.BUDGET <= 50000.00
```

← **Error**

```
SUM (DISTINCT D.BUDGET)
45000.00
```

**Logic:** We want to summarize over the BUDGET column from the DEPARTMENT table. However, we must join DEPARTMENT with EMPLOYEE because we want to eliminate from consideration any department (e.g., Department 30) without employees.

Applying the suggestion from the previous page, we display the intermediate result before applying the SUM function.

```
SELECT D.DNO, D.BUDGET
FROM   DEPARTMENT D, EMPLOYEE E
WHERE  D.DNO = E.DNO
AND    D.BUDGET <= 50000.00
```

```
DNO      BUDGET
20      20000.00
20      20000.00
20      20000.00
40      25000.00
```

Examination of this intermediate result leads to an important observation. Data for Department 20 appears three times because this department has three employees. This observation motivates us to specify DISTINCT within the SUM function. Then just one of the 20000.00 values is added to the 25000.00 value to produce the final total of 45000.00.

To generalize from this example, you must be aware that, whenever you join a parent-table (DEPARTMENT) and a child-table (EMPLOYEE), some data from the parent-table may appear redundantly in the join-result. Data from the child-table do not appear redundantly. Hence, with the previous sample query, we specified SUM (E.SALARY) because the SALARY column resides in the child-table; but we specified SUM (DISTINCT D.BUDGET) because BUDGET resides in the parent table. However!

**Almost Correct!** This SELECT statement got lucky (not really)! Today, it happens to produce the correct answer, but it might produce a wrong answer tomorrow. From a business perspective you should ask yourself: Can two or more departments have the exact same budget? If yes, the current SELECT statement would produce an incorrect result. For example, assume Department 40 has its budget changed to 20,000. Then the intermediate result would be:

<u>DNO</u>	<u>BUDGET</u>	
20	20000.00	
20	20000.00	
20	20000.00	
40	<b>20000.00</b>	<b>←</b>

Then the SUM (DISTINCT D.BUDGET) produces a result of 20000.00 which is *wrong*.

**Correct Solution:** We need to introduce Sub-SELECTs before we can present a correct solution. (See Exercise 23I and its solution.)



## Grouping with Join

The next three sample queries join the DEPARTMENT and EMPLOYEE tables, group by column(s) from the parent-table (DEPARTMENT), and summarize a column from the child-table (EMPLOYEE).

**Sample Query 17.4.1:** Reference the DEPARTMENT and EMPLOYEE tables. For each department with at least one employee, display the department number and total salary of all employees who work in the department.

```
SELECT D.DNO, SUM (E.SALARY)
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.DNO = E.DNO
GROUP BY D.DNO
```

<u>DNO</u>	<u>SUM (SALARY)</u>
10	2400.00
20	500.00
40	14000.00

**Logic:** After joining the DEPARTMENT and EMPLOYEE tables, groups are formed, and group summaries are calculated. Note that the join-operation eliminated Department 30, the only department without employees

**Better Solution:** The following statement (without a join-operation) is a better solution because the join-operation is unnecessary.

```
SELECT DNO, SUM (SALARY)
FROM EMPLOYEE
GROUP BY DNO
```

*This alternative solution requires that EMPLOYEE.DNO is a non-null foreign key that references DEPARTMENT. If the SELECT statement referenced the EMPLOYEE2 or EMPLOYEE3 tables, the result would be wrong. Verify by executing the following statements.*

```
SELECT DNO, SUM (SALARY)
FROM EMPLOYEE2
GROUP BY DNO
```

```
SELECT DNO, SUM (SALARY)
FROM EMPLOYEE3
GROUP BY DNO
```

The next sample query is similar to the previous example. The only difference is that we group by DNAME instead of DNO.

**Sample Query 17.4.2:** Reference the DEPARTMENT and EMPLOYEE tables. For each department with at least one employee, display the department name and total salary of all employees who work in the department.

```
SELECT D.DNAME, SUM (E.SALARY)
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.DNO = E.DNO
GROUP BY D.DNAME
```

<u>DNAME</u>	<u>SUM (SALARY)</u>
ACCOUNTING	2400.00
ENGINEERING	500.00
INFO. SYS.	14000.00

**Syntax & Logic:** Nothing new.

**Know-your-data:** *It is important to know that DNAME is unique.* Otherwise, if two departments could have the same name, then employee salaries from both departments would be summarized within the same group. (If DEPARTMENT could contain duplicate DNAME values, it would be a good idea to revise the query objective to group by and display both the DNO and DNAME columns. See the following sample query.)

The next sample query requires that you group by multiple columns. (You may wish to review Sample Query 9.12 in Chapter 9.5.)

**Sample Query 17.4.3:** Reference the DEPARTMENT and EMPLOYEE tables. For each department with at least one employee, display the department number, department name, and total salary of all employees who work in the department.

```
SELECT D.DNO, D.DNAME, SUM (E.SALARY)
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.DNO = E.DNO
GROUP BY D.DNO, D.DNAME
```

<u>DNO</u>	<u>DNAME</u>	<u>SUM (SALARY)</u>
10	ACCOUNTING	2400.00
20	ENGINEERING	500.00
40	INFO. SYS.	14000.00

**Logic:** Nothing new. Note that this SELECT statement obeys the following syntax rule (introduced in Chapter 9.5).

**GROUP BY Syntax Rule:** Whenever a SELECT-clause specifies one or more aggregate functions, if this SELECT-clause also specifies a column without an aggregate function, this column must be specified within a GROUP BY clause.

**Exercises:**

17Ca. Reference the DEPARTMENT and EMPLOYEE tables. For each department that has employees, display the department name along with the maximum departmental salary for employees who work in the department.

17Cb. Reference the DEPARTMENT and EMPLOYEE tables. For each department with at least one employee, display the department number, department name, department budget, maximum salary, and minimum salary of all employees who work in the department.

## Error: Accidental Cross Product

The following SELECT statement incorrectly generates a cross product as an intermediate result. This is not observable in the final result which is a statistical summary.

**Sample Query 17.5:** We are only interested in those departments that have employees and have a department name that begins with the letter A. Display the department names and average salaries of all employees who work in such departments. *The following statement produces a **wrong** answer.*

```
SELECT D.DNAME, AVG (E.SALARY)
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.DNAME LIKE 'A%'
GROUP BY D.DNAME
```

← Error

<u>DNAME</u>	<u>AVG (SALARY)</u>
ACCOUNTING	2816.66

**Logic:** The absence of a join-condition produced a cross product as the *intermediate* result. This intermediate result contained duplicate SALARY values that were used to calculate an incorrect average. The *correct* AVG (SALARY) is: 1200.00.

The **correct** SELECT statement is:

```
SELECT D.DNAME, AVG (E.SALARY)
FROM DEPARTMENT D, EMPLOYEE E
→ WHERE D.DNO = E.DNO
AND D.DNAME LIKE 'A%'
GROUP BY D.DNAME
```

**Observation:** Because the cross-product intermediate result will contain duplicate SALARY values (see Sample Query 17.1), the above result is wrong. If you do not have good intuition about SALARY values, you may not detect that the incorrect result is unreasonable.

## Join on Non-key Columns

All previous joins of the DEPARTMENT and EMPLOYEE tables were based upon their PK-FK relationship. The next sample query specifies a query objective that requires the join-condition to compare on non-key columns. The following query objective, which is not very realistic, is presented for tutorial purposes only.

**Sample Query 17.6:** Does any department have a name that is the same as some employee name? If yes, then display the department name.

```
SELECT D.DNAME
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.DNAME = E.ENAME
```

DNAME

(No rows returned)

**Syntax:** The syntax is valid. The join-condition satisfies the requirement that the columns being compared have compatible data-types.

**Logic:** Execution of this statement produces an empty result table because no DNAME value matches any ENAME value.

The following sample query presents a more realistic query objective that requires joining on non-key columns.

## Theta-Join

Previous join-operations implicitly assumed that "matching" means "equals." The next sample query illustrates that a join-condition can specify any of the conventional comparison operations (=, <>, >, >=, <, <=).

"*Theta-join*" is a generic term used to designate a join-operation based upon any comparison operation. The next sample query performs a "greater-than-join." Notice that the join-condition compares two non-key columns, EMPLOYEE.SALARY and DEPARTMENT.BUDGET.

**Sample Query 17.7:** Does any employee have such a large salary that it exceeds the budget of some department? If yes, display the employee's name, salary, and department number, followed by the corresponding department number and budget.

```
SELECT E.ENAME, E.SALARY, E.DNO EMPDNO,
       D.DNO DEPTDNO, D.BUDGET

FROM   EMPLOYEE E, DEPARTMENT D

WHERE  E.SALARY > D.BUDGET
```

<u>ENAME</u>	<u>SALARY</u>	<u>EMPDNO</u>	<u>DEPTDNO</u>	<u>BUDGET</u>
GEORGE	9000.00	20	30	7000.00

**Syntax:** This join-condition (E.SALARY > D.BUDGET) specifies a greater-than (>) operator. Again, note that neither SALARY nor BUDGET is a key-column.

**Logic:** Every SALARY value is compared to every BUDGET value. Only one comparison found a SALARY value that was greater than a BUDGET value.

## Join-Condition Contains an Arithmetic Expression

The following sample query specifies a greater-than-or-equal-to ( $\geq$ ) join-condition that compares a non-key column to an arithmetic expression.

**Sample Query 17.8:** Does any employee have a salary that is greater than or equal to one third of any departmental budget? If yes, display the employee's name, salary and department number, followed by the corresponding department number and budget.

```
SELECT E.ENAME, E.SALARY, E.DNO EMPDNO,  
       D.DNO DEPTDNO, D.BUDGET  
  
FROM   EMPLOYEE E, DEPARTMENT D  
  
WHERE  E.SALARY  $\geq$  (D.BUDGET * 0.333)
```

ENAME	SALARY	EMPDNO	DEPTDNO	BUDGET
CURLY	3000.00	20	30	7000.00
GEORGE	9000.00	20	20	20000.00
GEORGE	9000.00	20	30	7000.00
GEORGE	9000.00	20	40	25000.00

**Syntax:** This WHERE-clause illustrates that an arithmetic expression can be specified within a join-condition.

**Logic:** Each SALARY value is compared to  $1/3$  of every BUDGET value. Observe that only one row (the second row) in the result table has the same EMPDNO and DEPTDNO values (20) indicating that the employee's salary was greater than  $1/3$  of his own departmental budget. This observation motivates the next sample query.

## Compound Join-Conditions

A join-operation can specify a compound join-condition.

**Sample Query 17.9:** Does any employee have a salary that exceeds one third of *his own* departmental budget? If yes, display the employee's name, salary, and department number, followed by the budget amount for his department.

```
SELECT E.ENAME, E.SALARY, E.DNO, D.BUDGET
FROM   EMPLOYEE E, DEPARTMENT D
WHERE  E.DNO = D.DNO
AND    E.SALARY > (D.BUDGET*0.333)
```

<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>	<u>BUDGET</u>
GEORGE	9000.00	20	20000.00

**Syntax & Logic:** There are two valid interpretations of this SELECT statement.

First, the join-operation specifies a join-condition, `E.DNO = D.DNO`, that produces an intermediate join-result. The AND-condition, `E.SALARY > (D.BUDGET*0.333)` specifies a restriction on this intermediate join-result.

Second, the join-operation specifies a *compound join-condition*. The first component of the join-condition is `E.DNO = D.DNO`, and the second component of the join-condition is `E.SALARY > (D.BUDGET*0.333)`. Two rows match if (1) the corresponding DNO values are equal, and (2) the SALARY value is greater than 1/3 of the BUDGET value.

### Exercise:

17D. Assume every employee is given a \$20,000.00 raise. Under this circumstance, does any employee have a salary that exceeds the budget of *his own* department? If yes, display the employee's name, old salary, and new salary.



## Join Table with Itself

Sometimes we want to join a table with itself. This may seem unusual. However, there are circumstances where such a join can be very useful. (Chapter 30 on recursive queries will illustrate such a circumstance.) Although the following sample query is rather contrived, it illustrates the basic idea.

**Sample Query 17.10.1:** Display the ENO, ENAME, and SALARY values for each pair of employees that have the same SALARY value. The following statement produces a correct, but imperfect result.

```
SELECT E1.ENO ENO1, E1.ENAME ENAME1, E1.SALARY SALARY1,
       E2.ENO ENO2, E2.ENAME ENAME2, E2.SALARY SALARY2

FROM   EMPLOYEE E1, EMPLOYEE E2

WHERE  E1.SALARY = E2.SALARY
```

ENO1	ENAME1	SALARY1	ENO2	ENAME2	SALARY2
1000	MOE	2000.00	1000	MOE	2000.00
1000	MOE	2000.00	2000	LARRY	2000.00
2000	LARRY	2000.00	1000	MOE	2000.00
2000	LARRY	2000.00	2000	LARRY	2000.00
3000	CURLY	3000.00	3000	CURLY	3000.00
4000	SHEMP	500.00	4000	SHEMP	500.00
5000	JOE	400.00	5000	JOE	400.00
6000	GEORGE	9000.00	6000	GEORGE	9000.00

**Syntax:** The EMPLOYEE table is specified twice in the FROM-clause. If a FROM-clause references the same table twice, a table alias must be associated with at least one of the table-names. This example specifies two aliases, E1 and E2.

**Logic:** Conceptually, we have two tables, E1 and E2, which happen to contain the same data. This result is correct. However, observe that every employee matches himself! (For example, the first row shows MOE matching with MOE.) The following sample query eliminates self-matching rows from the result.

**Sample Query 17.10.2:** Modify the preceding statement to eliminate self-matching rows. This statement produces a better, but still imperfect result.

```
SELECT E1.ENO E1ENO, E1.ENAME E1ENAME, E1.SALARY E1SALARY,
       E2.ENO E2ENO, E2.ENAME E2ENAME, E2.SALARY E2SALARY

FROM   EMPLOYEE E1, EMPLOYEE E2

WHERE  E1.SALARY = E2.SALARY

AND    E1.ENO <> E2.ENO
```

<u>ENO1</u>	<u>ENAME1</u>	<u>SALARY1</u>	<u>ENO2</u>	<u>ENAME2</u>	<u>SALARY2</u>
1000	MOE	2000.00	2000	LARRY	2000.00
2000	LARRY	2000.00	1000	MOE	2000.00

**Logic:** Referencing the primary-key column (ENO) uniquely identifies each row. Therefore, the second join-condition (E1.ENO <> E2.ENO) eliminates self-matching rows. This result is better than the previous result, but it is not perfect. Examination of the result table shows MOE matches LARRY, and LARRY matches MOE! We would like to display just one of these rows.

**Sample Query 17.10.3:** One more modification.

```
SELECT E1.ENO E1ENO, E1.ENAME E1ENAME, E1.SALARY E1SALARY,
       E2.ENO E2ENO, E2.ENAME E2ENAME, E2.SALARY E2SALARY

FROM   EMPLOYEE E1, EMPLOYEE E2

WHERE  E1.SALARY = E2.SALARY

AND    E1.ENO < E2.ENO
```

<u>ENO1</u>	<u>ENAME1</u>	<u>SALARY1</u>	<u>ENO2</u>	<u>ENAME2</u>	<u>SALARY2</u>
1000	MOE	2000.00	2000	LARRY	2000.00

**Logic:** Specify < instead of <>. The result shows the E1.ENO value (1000) is less than the E2.ENO value (2000).

## Summary

This chapter did not introduce any new SQL syntax or logic. Sample queries implemented two-table inner-join operations to produce an intermediate join-result. Then additional processing (e.g., restriction, grouping, or summarization) produced a final result.

## Summary Exercises

- 17E. Assume that every employee is given a raise equal to 5% of the employee's departmental budget. Display every employee's name, old salary, and new salary.
- 17F. Only consider departments that have employees. How many of these departments have a budget that exceeds \$20,000.00? And, what is the total number of employees hired by these departments?
- 17G. Extend the previous exercise. Calculate a third column by dividing the second column (number of employees) by the first value (number of departments) to determine the overall average of employees per department.
- 17H. Only consider departments that have employees. Display the department name and the average departmental salary for each department.
- 17I. Modify the previous exercise. Display the department name and the average departmental salary if that average is less than \$1,000.00
- 17J. Only consider departments that have employees. For each such department, display the department name and the minimum salary paid to an employee who works in the department.
- 17K. Modify the previous exercise. We want to display the department name and the smallest salary paid to some employee who works in the department only if that minimum salary value is less than \$1,000.00.

## Appendix 17A: Efficiency

Whenever a SELECT statement asks the system to join two tables, the system uses some internal method to implement the join-operation. All systems support multiple methods for implementing this operation. The optimizer determines which internal method is the most efficient.

We briefly outline two basic join methods, the Nested-Loop method and Sort-Merge method. Assume that we are joining two tables, T1 and T2, where the join-condition is  $T1.X = T2.Y$ . Columns X and Y may or may not be key-columns. With all methods, whenever a match on the join-condition occurs, the matching rows are merged and saved for further processing.

**Nested-Loop:** The basic *row comparison* logic is:

- Read the first row in T1. Then read all rows in T2, comparing the first T1.X value to each T2.Y value.
- Read the second row in T1. Then (again) read all rows in T2, comparing the second T1.X value to each T2.Y value.
- Read the third row in T1. Then (again) read all rows in T2, comparing the third T1.X value to each T2.Y value.
- Etc.

**Sort-Merge:** The basic *row comparison* logic is:

- Sort table T1 by column X.
- Sort table T2 by column Y.
- The system can now compare T1.X and T2.Y by making a single synchronized pass over each table.

Neither of these methods appears to be very efficient. In the Nested-Loop method, the repeated reading of T2 could be costly. And, in the Sort-Merge method, the sorting of both T1 and T2 could be costly. However, there are more efficient variations of each method. These variations are only relevant in special circumstances, such as when a table is already sorted or a potentially useful index has been created.

**More Efficient Nested-Loop:** Assume the optimizer knows there is an index on column Y in T2. Then the basic logic is modified to:

- Read the first row in T1. Use the index to search T2 for rows where the T2.Y value equals the first T1.X value.
- Read the second row in T1. Use the index to search T2 for rows where the T2.Y value equals the second T1.X value.
- Read the third row in T1. Use the index to search T2 for rows where the T2.Y value equals the third T1.X value.
- Etc.

This variation of the Nested-Loop method depends on the presence of the T2.Y index. Assume the join-operation is based on a PK-FK relationship (as it usually is). If T2.Y is a foreign key, it may or may not have an index. However, as mentioned earlier, many designers create indexes on foreign keys. Under this circumstance, the optimizer might be inclined to use this variation of the Nested-Loop method.

Also, note that T1 was the "first" table in our description of the Nested-Loop join. In this context, T1 is called the "outer-table" (or the "driving table"), and T2 is called the "inner-table." Logically, either table can be designated as the outer-table because  $(T1 \text{ inner-join } T2) = (T2 \text{ inner-join } T1)$ . The optimizer may find some advantage by designating a specific table, T1 or T2, as the outer-table.

**More Efficient Sort-Merge:** Assume T1 is already sorted by column X. Assume T2 is small and it is not sorted. The basic logic is modified as shown below.

- Sort T2 by column Y (cheap sort for small table)
- The system can now make a single synchronized pass over each table as it compares T1.X and T2.Y values.

Observation: This join-method could produce an *incidentally sorted* join-result.

**Optimization Challenges:** Your system's optimizer attempts to determine the most efficient way to perform the join-operation. The optimizer considers the size of the tables, the presence or absence of indexes, possible sort sequences, and tradeoffs between different join-methods. Appendix 17.C will say more about optimizing join-operations.

## Appendix 17B: Theory

**Relational Database Languages:** In Appendix 1B we noted that Codd's first query language for relational databases was the **Relational Calculus**. (If you have taken a course in discrete mathematics, you may have encountered the predicate calculus. The predicate calculus provides the mathematical foundation for Codd's relational calculus.) Shortly after defining his relational calculus, Codd defined a second query language called the **Relational Algebra**. This language was equivalent to the calculus in the sense that any statement written in the calculus could be rewritten in the algebra, and vice versa. Most students feel that Codd's algebra is simpler than his calculus. Within SQL, the SELECT statement has features that are derived from both the relational calculus and relational algebra. This appendix will take a closer look at Codd's relational algebra.

**Relational Algebra:** Codd's original algebra has eight operations. You have already encountered four of these operations. These are:

1. RESTRICT (Sample Queries 1.2, 1.3 & Appendix 1B)
2. PROJECT (Sample Query 1.4 & Appendix 1B)
3. INNER JOIN (Sample Query 16.1)
4. CROSS PRODUCT (Sample Query 17.1)

We review each operation and present a pseudo-code algebraic notation for each operation. Our discussion will reference relations (tables) T1 and T2 with the following attributes (columns).

T1 (A, B, C, D)

T2 (W, X, Y, Z)

### 1. RESTRICT

SQL: SELECT \* FROM T1 WHERE A = 44

Algebra: **RESTRICT T1 where A = 44**

## 2. PROJECT

SQL:           SELECT A, C, D FROM T1

Algebra:   **PROJECT T1 [A, C, D]**

## 3. CROSS PRODUCT

SQL:           SELECT \* FROM T1, T2

Algebra:   **CROSS (T1, T2)**

## 4. JOIN

SQL:           SELECT \* FROM T1, T2 WHERE T1.A = T2.Z

Algebra:   **T1 JOIN T2 where T1.A = T2.Z**

Appendix 21B will present three more algebraic operations (UNION, INTERSECT, and EXCEPT).

The following Appendix 17C will describe a useful application of the relational algebra. For the moment, we note that you (and your system's optimizer) can rewrite a SELECT statement as an equivalent sequence of algebraic operations. For example, consider the following statement.

```
SELECT       A, C, D
FROM         T1
WHERE         T1.A = 44
```

The following sequence of algebraic operations produces the same result.

```
TEMP    ← RESTRICT T1 where T1.A = 44
RESULT ← PROJECT TEMP [A, C, D]
```

The arrow (←) represents the assignment of a result to a temporary result (TEMP) or a final result (RESULT).

These RESTRICT and PROJECT operations can be represented in a single line of code as shown below.

```
RESULT ← PROJECT (RESTRICT T1 where T1.A = 44) [A, C, D]
```

**Primitive Algebraic Operations:** As stated above, Codd's algebra defined eight operations. Five of these operations are "primitive" in the sense that they are necessary. These primitive operations produce results that cannot be produced by using some combination of the other primitive operations.

Non-primitive operations are logically superfluous. They produce results that can be produced by using the primitive operations. Although these non-primitive operations are unnecessary, they are useful. Otherwise, Codd would not have included them in his Relational Algebra.

Of the four algebraic operations presented above, RESTRICT, PROJECT, and CROSS are primitive. *JOIN is not primitive.* Join is useful, but we really don't need it. To illustrate this point, we will present two algebraic expressions that are equivalent to the following SELECT statement.

```
SELECT *
FROM   DEPARTMENT, EMPLOYEE
WHERE  DEPARTMENT.DNO = EMPLOYEE.DNO
```

AlgebraProc-1 specifies a JOIN to produce the desired result. AlgebraProc-2 specifies the CROSS and RESTRICT operations to produce the same result.

AlgebraProc-1

```
RESULT ← DEPARTMENT JOIN EMPLOYEE
        where DEPARTMENT.DNO = EMPLOYEE.DNO
```

AlgebraProc-2

```
TEMP ← CROSS (DEPARTMENT, EMPLOYEE)
RESULT ← RESTRICT TEMP where TEMP.DNO1 = TEMP.DNO2
```

You can make the same observation by considering the SQL syntax for join.

```
SELECT *
FROM   T1, T2 } ← cross product
WHERE  T1.A = T2.Z ← restriction
```

The first two lines (SELECT \* FROM T1, T2) correspond to CROSS. The last line (WHERE T1.A = T2.Z) corresponds to RESTRICT.



## Appendix 17C: Theory & Efficiency

This appendix adds more substance to our previous introduction to query optimization.

We have shown that a SELECT statement can be expressed using Codd's relational algebra. In the previous Appendix 17B, we considered four of Codd's eight algebraic operations: RESTRICT, PROJECT, CROSS, and JOIN. In this appendix we utilize these operations to demonstrate *the role the relational algebra plays within query optimization*.

**Caveat:** Not all SELECT statements can be expressed in Codd's algebra. Many SELECT statements include arithmetic expressions, built-in functions, grouping, and sorting. These operations involve computing and manipulating data after it has been retrieved. Codd's algebra did not include these operations. His algebra focused on retrieving data from relations (tables). Therefore, all commercial optimizers utilize other operations in addition to the algebraic operations.

The optimizer begins by validating the syntax of the SELECT statement. If valid, the optimizer translates the SELECT statement into a sequence of lower-level procedures. Some of these procedures correspond to the algebraic operations. From an overly simplified perspective, you can think of the system having an internal procedure called RESTRICT, another procedure called JOIN, etc. These procedures do the real work. We present an example using the following statement.

```
SELECT DEPARTMENT.DNO, BUDGET, ENAME, SALARY
FROM   EMPLOYEE, DEPARTMENT
WHERE  EMPLOYEE.DNO = DEPARTMENT.DNO
AND    SALARY < 999.00
AND    BUDGET <=75000.00
```

This optimizer translates this statement into sequence of algebraic operations such as: **JOIN** EMPLOYEE and DEPARTMENT to form an intermediate result. Then, **RESTRICT** this result to rows where SALARY < 1000.00 and BUDGET <= 75000.00. Then **PROJECT** the desired columns (DNO, BUDGET, ENAME, SALARY). Using this algebraic pseudo-code, we have a sequence of operations that we call the JOIN-First Procedure.

### JOIN-First Procedure

```
TEMP1 ← (EMPLOYEE JOIN DEPARTMENT)
        where EMPLOYEE.DNO = DEPARTMENT.DNO

TEMP2 ← RESTRICT TEMP1
        where SALARY < 1000.00 AND BUDGET <=75000.00

RESULT ← PROJECT TEMP2 [DNAME, ENAME, SALARY]
```

The optimizer could direct the system to execute this procedure. However, it might decide to generate another logically equivalent sequence of algebraic operations that may be more efficient.

Alternative sequences of operations are generated by applying heuristic rules like: "Executing RESTRICT before JOIN is usually more efficient than executing JOIN before RESTRICT. Either way, you get the same result." (A visual illustration of this rule is shown in Figure 17C.1 at the end of this appendix.) By applying this rule, the optimizer can derive another sequence of operations that we call RESTRICT-PROJECT-First Procedure.

### RESTRICT-PROJECT-First Procedure

Generate an intermediate result by **restricting** EMPLOYEE (SALARY < 1000.00) and **projecting** the relevant columns (ENAME, SALARY, DNO). Next, generate a second intermediate result by **restricting** DEPARTMENT (BUDGET <=75000.00) and **projecting** the relevant columns (DNO, BUDGET). Then **join** the two intermediate results. The algebraic pseudo-code is:

```
TEMP1 ← PROJECT (RESTRICT EMPLOYEE where SALARY < 1000)
        [ENAME, SALARY, DNO]

TEMP2 ← PROJECT (RESTRICT DEPARTMENT where BUDGET <= 75000)
        [DNO, BUDGET]

RESULT ← (TEMP1 JOIN TEMP2) where TEMP1.DNO = TEMP2.DNO
```

For the sake of simplicity, assume your system only supports two internal procedures that implement the RESTRICT operation on a table (T).

RS(T) : RESTRICT-BY-SCAN procedure

RI(T) : RESTRICT-USING-INDEX procedure

Also, assume that both RS and RI perform both RESTRICT and PROJECT. When the procedure retrieves a row, it only saves values for those columns that are relevant to the query.

Likewise, assume your system only has two procedures, JNL and JSM, that implement the JOIN operation for tables (T1, T2) where T1 is the outer table.

JNL(T1,T2) : JOIN-BY-NESTED-LOOP procedure

JSM(T1,T2) : JOIN-BY-SORT-MERGE procedure

Considering the above procedures, the optimizer generates more detail plans. For example, if the SALARY column has an index, and there is no index on the BUDGET column, the optimizer considers the following plan which adds details to RESTRICT-PROJECT-First sequence and the join-operation.

```
TEMP1 ← RI (EMPLOYEE where SALARY < 1000) [ENAME, SALARY, DNO]
```

```
TEMP2 ← RS (DEPARTMENT where BUDGET <= 75000) [DNO, BUDGET]
```

```
RESULT ← (TEMP2 JNL TEMP1) where TEMP1.DNO = TEMP2.DNO
```

### Many Possible Application Plans

It becomes evident that an optimizer must "do a lot of thinking" when it examines a SELECT statement and attempts to formulate an efficient application plan. For example, reconsider the preceding SELECT statement.

```
SELECT DEPARTMENT.DNO, BUDGET, ENAME, SALARY
FROM   EMPLOYEE, DEPARTMENT
WHERE  EMPLOYEE.DNO = DEPARTMENT.DNO
AND    SALARY < 999.00
AND    BUDGET <= 75000.00
```

Below, we focus on the JOIN and RESTRICT operations. (We assume that PROJECT is usually implemented within the RESTRICT operation).

For the JOIN operation, the optimizer must:

- Determine the overall logic.  
(JOIN-FIRST versus RESTRICT-PROJECT-First)
- Determine the driving (outer) table.  
(T1 JOIN T2 versus T2 JOIN T1).
- Determine the join-method.  
(E.g., Nested-loop versus sort-merge).

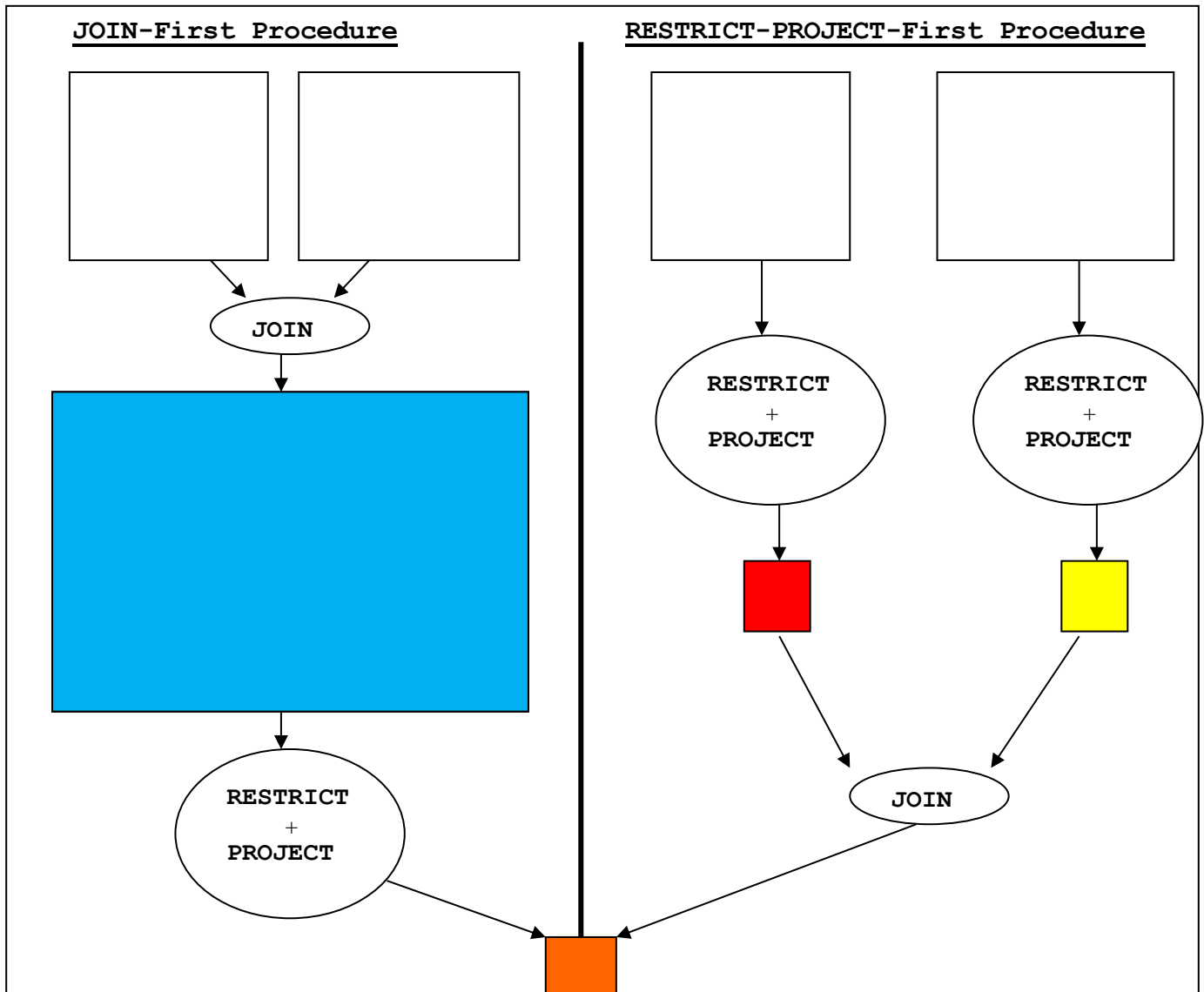
For the RESTRICT operations, the optimizer must:

- Determine the fastest way (scan or use an index) to retrieve EMPLOYEE rows where SALARY < 999.00
- Determine the fastest way (scan or use an index) to retrieve DEPARTMENT rows where BUDGET <= 75000.00.

You can see that an optimizer has many options when formulating an application plan. In fact, there may be too many options. (This SELECT statement is relatively simple. The following chapter will present SELECT statements that join seven tables.) We also note that some optimizer decisions are not independent. For example, the presence of a relevant index could influence decisions about both the JOIN and RESTRICTION operations.

The optimizer considers most (maybe all) possible applications plans, evaluates the cost of each plan, and chooses the plan with the lowest cost. When you consider all possible plans involving the number of tables, the size of each table, the number and type of indexes, the type of join-methods, and the type of restriction-methods, you should realize that the optimizer frequently has to evaluate many different query plans.

**Summary - Appreciate Your Optimizer:** The optimizer does the heavy lifting. Your job is to write correct SQL. Note the role played by the relational algebra. Again, we quote C. J. Date: "Theory is practical."



We do not prove that both join-procedures always produce the same result. (Trust me - they do.) We note that the RESTRICT-PROJECT-First procedure usually performs better because JOIN-First could generate a very large intermediate result. If this result does not fit into main memory, it has to be written to disk and then reread from disk for subsequent the RESTRICT+PROJECT operation. Alternatively, with the JOIN-First procedure, by initially executing the two RESTRICT+PROJECT operations, smaller intermediate results are produced and the subsequent JOIN operation is applied to these smaller intermediate results.

However, with the RESTRICT-PROJECT-FIRST Procedure, if one or both of the RESTRICT or PROJECT operations produce a very large intermediate result, the optimizer may choose the JOIN-FIRST Procedure.

**Figure 17C.1: Equivalent Sequences of Algebraic Operations**

## Multi-Table Inner-Joins

This chapter introduces three-table, four-table, five-table etc. inner-join operations. It also introduces the MTPC Data Model that will be referenced throughout the remainder of this book. This is a large chapter that is organized into the following sections.

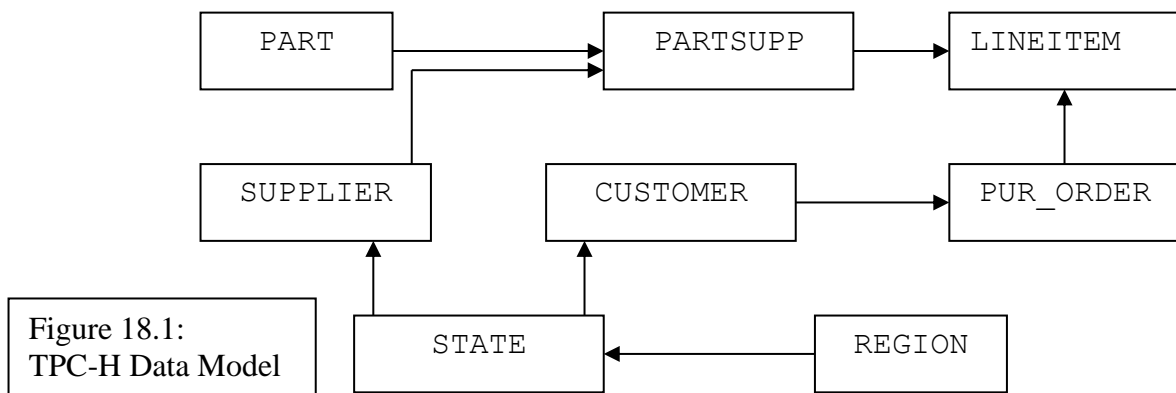
- A. **Sample Tables:** This chapter's sample queries reference a database design consisting of eight tables that are described in this section.
- B. **Getting Started: Joining Three Tables.** The basic syntax and logic of a three-table join are introduced.
- C. **Joining Four or More Tables:** A three-table join is extended to a four-table join, followed by a five-table join, etc., to an eight-table join.
- D. **Query Analysis and Coding Guidelines:** The previous Sections B and C introduced a rather "mechanical" approach to coding multi-table inner-join operations. This section presents a conceptual framework where a multi-table SELECT statement is derived by mapping a query objective to a logical data model.
- E. **Join with Other Operations:** Sample queries incorporate restriction and grouping within multi-table joins.
- F. **JOIN-ON Syntax:** Examples illustrate multi-table joins that are coded using the JOIN-ON syntax.

This chapter concludes with six optional appendices that address data modeling and efficiency considerations.

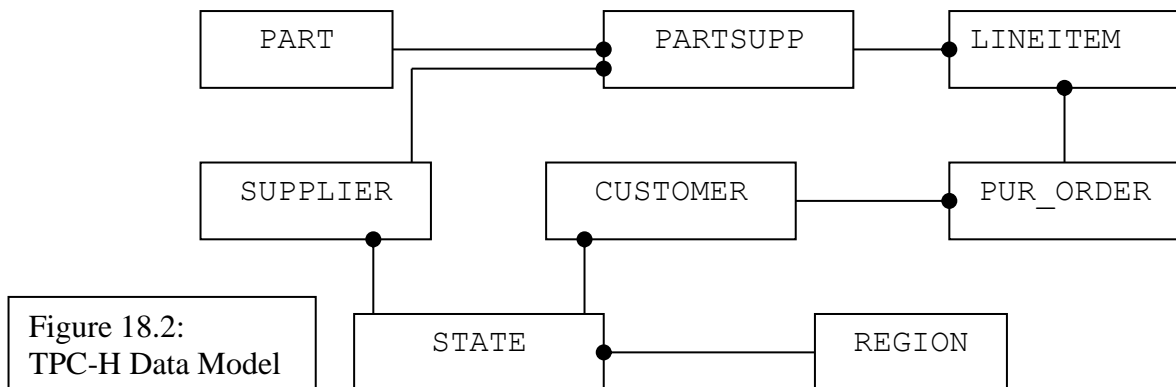
## A. Sample Tables – The MTPCH Database

**TPC Council:** The Transaction Processing Council (TPC) is an organization that defines benchmarks used within the database community to evaluate the performance of commercial database systems. [Visit: [www.tpc.org](http://www.tpc.org).] This organization has published documentation on the *TPC-H* database that is used to evaluate database performance within the context of decision support systems.

**TPC-H Database:** The following data model (Figure 18.1), copied from the TPC website, represents the TPC-H database.



This model uses an arrowhead to designate the many-side of a one-to-many relationship. The following data model (Figure 18.2) is equivalent to the above data model. Here, a circle is used to designate the many-side of a one-to-many relationship. This notation will be used throughout the remainder of this book.



**Semantics of TPC-H Data Model:** This model assumes that you (the user) work for a company that purchases parts from suppliers for resale to customers. Data about parts, suppliers, and customers are stored in the PART, SUPPLIER, and CUSTOMER tables. Each customer is located in one state, and each supplier is also located in one state. Each state is located in one geographic region. The regions and states are described by the REGION and STATE tables.

Each supplier may sell many parts, and each part may be purchased from many suppliers. However, a given part cannot always be purchased from any supplier who happens to sell that part. The PARTSUPP table specifies all acceptable part-supplier combinations, indicating which parts can be purchased from which suppliers. The PARTSUPP table also indicates the price that your company pays the supplier for the part.

Your company's customers complete purchase-orders with one or more line-items. This information is stored in the PUR\_ORDER and LINEITEM tables. Each row in the LINEITEM table is associated with one part and indicates the quantity and price the customer paid to your company for the part. We note that the same part may be sold for different prices in different line-items.

Finally, we note that each LINEITEM must be associated with some valid part-supplier (PNO, SNO) combination found in the PARTSUPP table. This assures that every purchase of a part is associated with some acceptable supplier of the part.

**Modified TPC-H (MTPCH) Database:** This chapter's sample queries will reference a database called the *MTPCH* Database which is a modification of the TPC-H database. (The author decided to utilize the TPC-H model because its semantics reflect the design complexity of a real-world database.) We emphasize that the MTPCH Database has the *same data model* as the *TPC-H data model* shown in Figure 18.2. However, for tutorial purposes, each table has fewer columns and rows, and the column-names have been simplified.



## The MTPCH Data Model

The following Figure 18.3 adds detail to Figure 18.2. This figure includes column-names and designates each primary-key (underlined) and foreign-key (FK). Also, this figure is hierarchically orientated (parent-over-child). (Note: Our designation "hierarchical orientation" does not strictly conform to a "hierarchical graph" as defined in graph theory.)

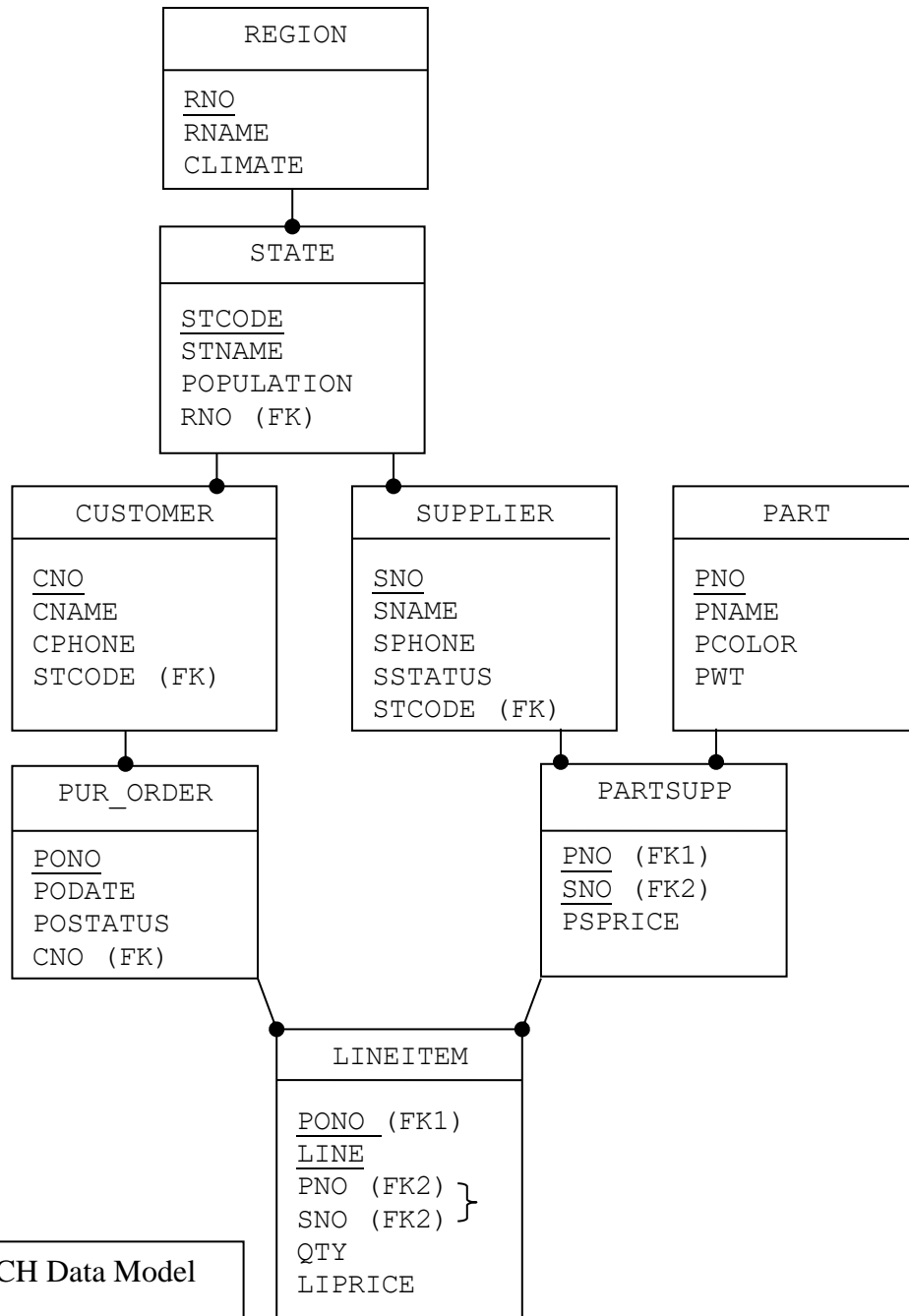


Figure 18.3: MTPCH Data Model

## Detailed MTPCH Model

The following figure enhances Figure 18.3 by including column data-types, table aliases, and join-conditions for PK-FK relationships. This model will facilitate the coding of SELECT statements. (You might want to print a copy of this page and the next two pages to reference when you work on this chapter's exercises.)

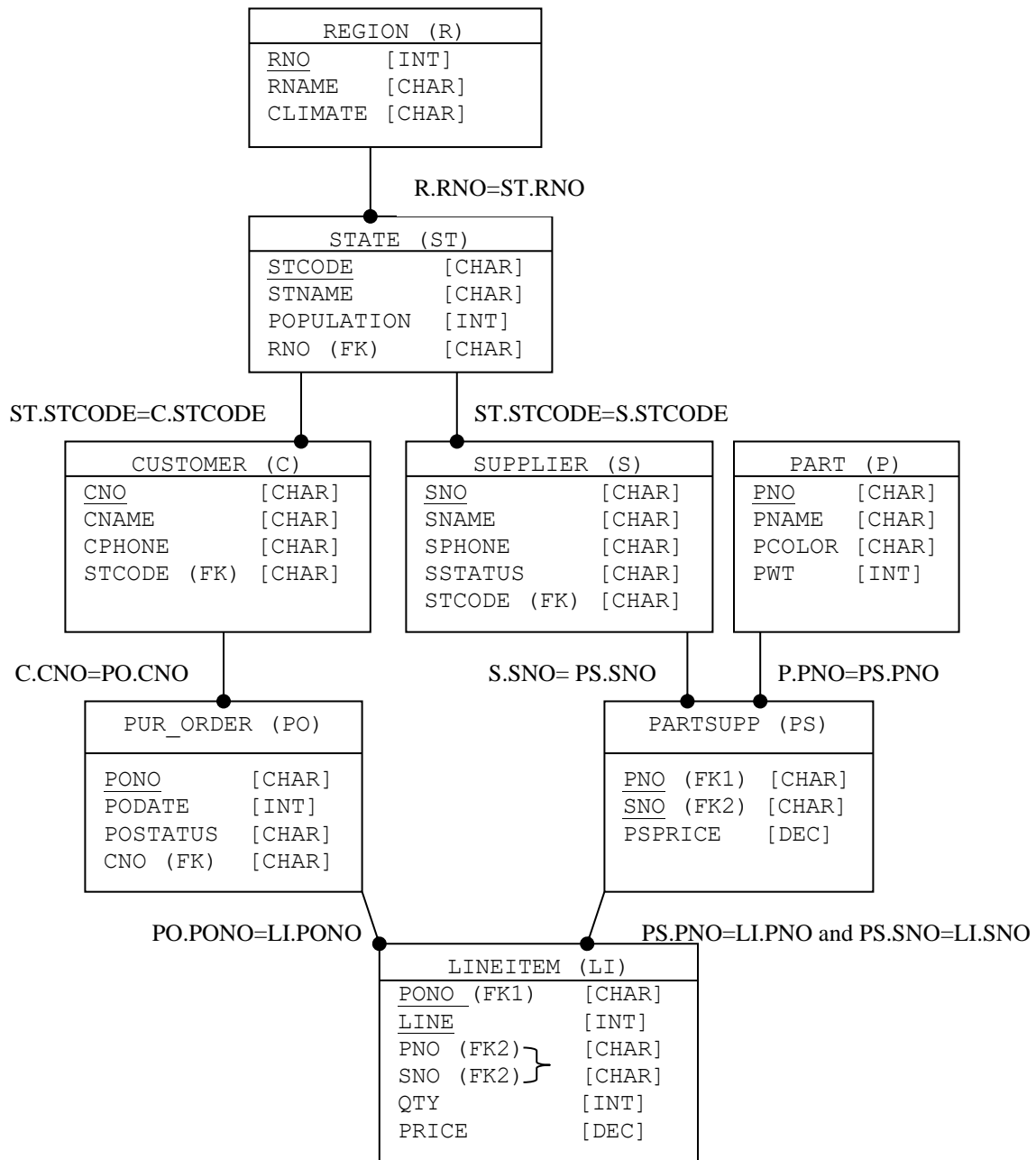


Figure 18.4: Detailed MTPCH Data Model

## Sample Data for MTPCH Model

For the sake of readability, rows in the following tables are displayed in primary-key sequence. This may or may not conform to any physical sequence (or absence of sequence) in the stored tables.

REGION		
RNO	RNAME	CLIMATE
1	NORTHEAST	Cold
2	NORTHWEST	Cold
3	SOUTHEAST	Hot
4	SOUTHWEST	Hot
5	MIDWEST	Empty

STATE			
STCODE	STNAME	POPULATION	RNO
AZ	ARIZONA	6339000	4
CT	CONNECTICUT	3502000	1
FL	FLORIDA	18251000	3
GE	GEORGIA	9545000	3
MA	MASSACHUSETTS	6450000	1
NM	NEW MEXICO	1970000	4
OR	OREGON	3747000	2
WA	WASHINGTON	6468000	2

CUSTOMER			
CNO	CNAME	CPHONE	STCODE
100	PYTHAGORAS	800-999-9999	MA
110	EUCLID	800-999-8888	MA
200	HYPATIA	800-888-9999	MA
220	ZENO	800-888-8888	MA
230	BOLYAI	800-888-7777	MA
300	NEWTON	800-777-9999	OR
330	LEIBNIZ	800-777-8888	OR
400	DECARTES	800-666-9999	WA
440	PASCAL	800-666-8888	WA
500	HILBERT	877-999-1234	MA
600	BOOLE	877-888-4321	FL
660	CANTOR	877-888-8765	FL
700	RUSSELL	877-777-1235	GE
770	GODEL	877-777-5321	GE
780	CHURCH	877-777-6321	NM
800	VON NEUMANN	877-666-9123	NM
880	TURING	877-666-3219	AZ
890	MANDELBROT	877-666-5219	AZ

PART			
PNO	PNAME	PCOLOR	PWT
P1	PART1	RED	20
P2	PART2	BLUE	10
P3	PART3	PINK	20
P4	PART4	YELLOW	10
P5	PART5	RED	20
P6	PART6	BLUE	12
P7	PART7	PINK	20
P8	PART8	PINK	15

SUPPLIER				
SNO	SNAME	SPHONE	SSTATUS	STCODE
S1	SUPPLIER1	888-999-9999	S	MA
S2	SUPPLIER2	888-999-1111	P	MA
S3	SUPPLIER3	888-999-3333	S	CT
S4	SUPPLIER4	888-999-4444	S	FL
S5	SUPPLIER5	888-999-6666	S	GE
S6	SUPPLIER6	889-888-9999	S	WA
S7	SUPPLIER7	889-888-3333	P	OR
S8	SUPPLIER8	889-888-2222	S	OR

PARTSUPP		
PNO	SNO	PSPRICE
P1	S2	10.50
P1	S4	11.00
P3	S3	12.00
P3	S4	12.50
P4	S4	12.00
P5	S1	10.00
P5	S2	10.00
P5	S4	11.00
P6	S4	4.00
P6	S6	4.00
P6	S8	4.00
P7	S2	2.00
P7	S4	3.00
P7	S5	3.50
P7	S6	3.50
P8	S4	5.00
P8	S6	4.00
P8	S8	3.00

**PUR\_ORDER**

PONO	PODATE	POSTATUS	CNO
11101	1	C	100
11102	3	P	100
11108	47	C	110
11109	49	P	110
11110	20	C	200
11111	21	P	200
11120	22	C	220
11121	23	C	220
11122	5	P	220
11124	6	P	230
11130	7	C	300
11133	8	P	300
11139	9	C	330
11141	61	P	330
11142	62	C	400
11144	63	P	400
11146	64	C	440
11148	65	C	440
11149	71	P	440
11150	72	P	500
11152	73	C	600
11153	74	P	600
11154	75	C	660
11155	1	P	660
11156	1	C	700
11157	3	P	770
11158	3	C	800
11159	3	C	880
11160	4	P	880
11170	10	P	880
11198	10	P	880

**LINEITEM**

PONO	LINE	PNO	SNO	QTY	LIPRICE
11101	1	P1	S2	10	11.50
11101	2	P3	S3	10	12.00
11102	1	P3	S3	20	13.00
11102	2	P4	S4	20	13.00
11108	1	P5	S1	10	11.00
11108	2	P6	S4	10	5.00
11109	1	P1	S2	10	11.50
11109	2	P7	S2	20	3.00
11109	3	P8	S4	20	6.00
11110	1	P8	S4	30	6.00
11111	1	P1	S4	10	12.00
11111	2	P3	S4	10	13.50
11120	1	P4	S4	20	13.00
11120	2	P5	S2	20	11.00
11121	1	P6	S6	20	5.00
11121	2	P7	S4	20	4.00
11122	1	P1	S2	10	11.50
11122	2	P3	S3	10	13.00
11124	3	P4	S4	10	13.00
11124	4	P5	S1	10	11.00
11130	1	P6	S4	5	5.00
11130	2	P7	S2	5	3.00
11130	3	P5	S4	10	12.00
11133	1	P1	S4	10	12.00
11139	1	P3	S4	20	13.50
11139	2	P5	S2	20	11.00
11141	1	P5	S4	10	12.00
11141	2	P6	S4	10	5.00
11142	1	P6	S8	10	5.00
11142	2	P7	S2	20	3.00
11144	1	P7	S4	20	4.00
11144	2	P8	S6	10	5.00
11146	1	P7	S5	10	4.50
11146	2	P8	S6	10	5.00
11148	1	P1	S2	10	11.50
11148	2	P8	S4	10	6.00
11149	1	P7	S5	40	4.50
11149	2	P8	S8	40	4.00
11150	1	P3	S4	20	13.50
11150	2	P6	S4	10	5.00
11152	1	P5	S4	10	12.00
11152	2	P7	S2	5	3.00
11152	3	P8	S8	40	4.00
11153	1	P8	S8	40	4.00
11154	1	P1	S2	10	11.50
11154	2	P3	S4	20	14.50
11154	3	P4	S4	10	13.00
11154	4	P5	S1	10	11.00
11155	1	P8	S8	40	4.50
11156	1	P1	S2	10	11.50
11156	2	P3	S4	20	13.50
11156	3	P5	S1	10	11.00
11157	1	P3	S4	20	13.50
11157	2	P5	S1	10	11.00
11158	1	P1	S2	10	11.50
11158	2	P3	S4	20	13.50
11159	1	P6	S4	10	5.00
11159	2	P7	S2	5	3.00
11160	1	P1	S2	10	12.50
11160	2	P7	S2	5	3.00
11170	1	P3	S4	20	12.50
11170	2	P4	S4	10	13.00

## B. Getting Started: Joining Three Tables

**Review:** This section begins with two sample queries that only require two-table join-operations. Thereafter, the query objectives for these sample queries are extended such that they require three-table join-operations.

**Sample Query 18.1:** Access the REGION and STATE tables. For every state, display its STCODE and STNAME values along with its region's RNO and RNAME values. Display these columns in the following left-to-right sequence: RNO, RNAME, STCODE, and STNAME. Sort the result by STCODE within RNO.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME
FROM   REGION R,
       STATE ST
WHERE  R.RNO = ST.RNO
ORDER BY R.RNO, ST.STCODE
```

<u>RNO</u>	<u>RNAME</u>	<u>STCODE</u>	<u>STNAME</u>
1	NORTHEAST	CT	CONNECTICUT
1	NORTHEAST	MA	MASSACHUSETTS
2	NORTHWEST	OR	OREGON
2	NORTHWEST	WA	WASHINGTON
3	SOUTHEAST	FL	FLORIDA
3	SOUTHEAST	GE	GEORGIA
4	SOUTHWEST	AZ	ARIZONA
4	SOUTHWEST	NM	NEW MEXICO

**Syntax & Logic:** Nothing new. Examine Figure 18.4. Observe that REGION is the parent-table, and STATE is the child-table. Therefore, all STATE rows appear in this result, and any non-matching REGION row (e.g., the MIDWEST region with a RNO value of 5) does not appear in this result.

Also, note that the join-condition (R.RNO = ST.RNO) is specified in Figure 18.4.

**Sample Query 18.2:** Access the PART and PARTSUPP tables. For every part that you can purchase from some supplier, display the part's PNO and PNAME values, the supplier's SNO value, and the price (PSPRICE) paid to the supplier for the part. Display these columns in the following left-to-right sequence: PNO, PNAME, SNO, and PSPRICE. Sort the result by SNO within PNO.

```

SELECT P.PNO, P.PNAME, PS.SNO, PS.PSPRICE
FROM   PART P,
       PARTSUPP PS
WHERE  P.PNO = PS.PNO
ORDER BY P.PNO, PS.SNO

```

PNO	PNAME	SNO	SPRICE
P1	PART1	S2	10.50
P1	PART1	S4	11.00
P3	PART3	S3	12.00
P3	PART3	S4	12.50
P4	PART4	S4	12.00
P5	PART5	S1	10.00
P5	PART5	S2	10.00
P5	PART5	S4	11.00
P6	PART6	S4	4.00
P6	PART6	S6	4.00
P6	PART6	S8	4.00
P7	PART7	S2	2.00
P7	PART7	S4	3.00
P7	PART7	S5	3.50
P7	PART7	S6	3.50
P8	PART8	S4	5.00
P8	PART8	S6	4.00
P8	PART8	S8	3.00

**Syntax & Logic:** Nothing new. Examine Figure 18.4. Observe that PART is the parent-table, and PARTSUPP is the child-table. Therefore, all PARTSUPP rows appear in this result, and any non-matching PART rows (e.g., Part P2) do not appear in this result.

Also, note that the join-condition (P.PNO = PS.PNO) is specified in Figure 18.4.

## Three-Table Join

The following sample query extends Sample Query 18.1 from a two-table join into a three-table join in order to display a customer's number (CNO) and name (CNAME) values.

**Sample Query 18.3:** Access the REGION, STATE, and CUSTOMER tables. For all customers, display their CNO and CNAME values, along with their state's STCODE and STNAME values, along with their region's RNO and RNAME values. Display these columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, CNO and CNAME. Sort the result by CNO within STCODE within RNO.

```
SELECT R.RNO, R.RNAME,
       ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME

FROM   REGION R,
       STATE ST,
       CUSTOMER C    ← new table

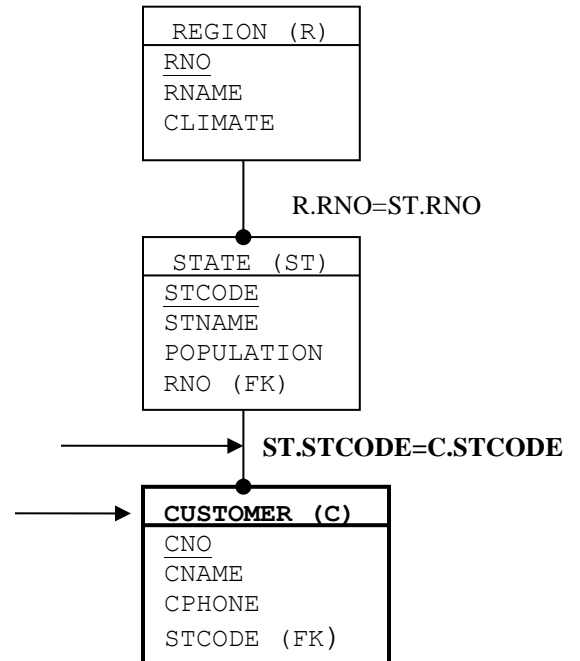
WHERE  R.RNO = ST.RNO
AND    ST.STCODE = C.STCODE    ← new join-condition

ORDER BY R.RNO, ST.STCODE, C.CNO
```

RNO	RNAME	STCODE	STNAME	CNO	CNAME
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO
1	NORTHEAST	MA	MASSACHUSETTS	230	BOLYAI
1	NORTHEAST	MA	MASSACHUSETTS	500	HILBERT
2	NORTHWEST	OR	OREGON	300	NEWTON
2	NORTHWEST	OR	OREGON	330	LEIBNIZ
2	NORTHWEST	WA	WASHINGTON	400	DECARTES
2	NORTHWEST	WA	WASHINGTON	440	PASCAL
3	SOUTHEAST	FL	FLORIDA	600	BOOLE
3	SOUTHEAST	FL	FLORIDA	660	CANTOR
3	SOUTHEAST	GE	GEORGIA	700	RUSSELL
3	SOUTHEAST	GE	GEORGIA	770	GODEL
4	SOUTHWEST	AZ	ARIZONA	880	TURING
4	SOUTHWEST	AZ	ARIZONA	890	MANDELBROT
4	SOUTHWEST	NM	NEW MEXICO	780	CHURCH
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN

**Syntax & Logic:** Nothing new. Considering this sample query as an extension of Sample Query 18.1 (which accessed the REGION and STATE tables), we see that we have to include another table, the CUSTOMER table.

Looking at the MTPCH model (Figure 18.4) makes this easy. We simply append the new table (CUSTOMER) to the FROM-clause, and AND-connect the new join-condition (ST.STCODE=C.STCODE) to the WHERE-clause.



**Again, Syntax & Logic:** Alternatively, if we do not consider this sample query to be an extension of Sample Query 18.1, query analysis begins with: "What tables do I need?" Here, we want to display data from columns found in the REGION, STATE, and CUSTOMER tables. Hence, you must reference these three tables in the FROM-clause and specify two join-conditions. (It is not always this simple. Sample Query 18.5 will access three tables, but only display columns from two tables.)

Also, as with any inner-join, you must ask: "What rows might not appear in the join result?" In this example, we get all rows from the CUSTOMER table because all its foreign-keys must match. However, there may be rows in the STATE table (e.g., CONNECTICUT) that do not match the CUSTOMER table (i.e., states without customers); and, there may be rows in a REGION table (e.g., MIDWEST region) that do not match the STATE table (i.e., regions without states).



The following sample query extends Sample Query 18.2 from a two-table join to a three-table join in order to access supplier names (SNAME). This query objective requires access to the SUPPLIER table which contains the required SNAME values.

**Sample Query 18.4:** Access the PART, SUPPLIER, and PARTSUPP tables. For every part that you can purchase from some supplier, display the part's PNO and PNAME values, the supplier's SNO and SNAME values, and the price (PSPRICE) paid to the supplier for the part. Display these columns in the following left-to-right sequence: PNO, PNAME, SNO, SNAME, and PSPRICE. Sort the result by SNO within PNO.

```

SELECT P.PNO, P.PNAME,
       S.SNO, S.SNAME,
       PS.PSPRICE

FROM   PART P,
       PARTSUPP PS,
       SUPPLIER S           ← new table

WHERE  P.PNO = PS.PNO
AND    S.SNO = PS.SNO     ← new join-condition

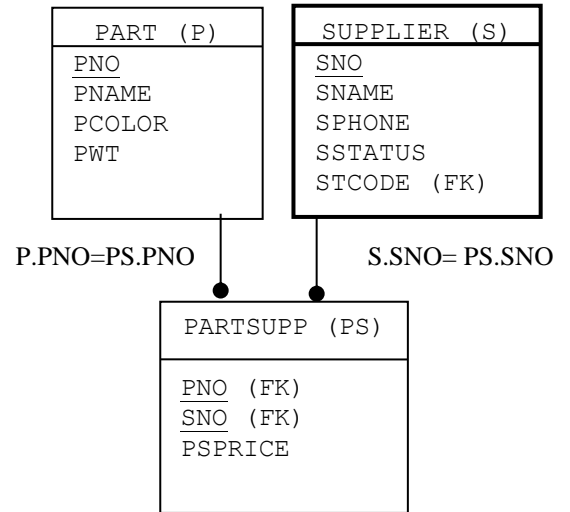
ORDER BY P.PNO, S.SNO

```

PNO	PNAME	SNO	SNAME	PSPRICE
P1	PART1	S2	SUPPLIER2	10.50
P1	PART1	S4	SUPPLIER4	11.00
P3	PART3	S3	SUPPLIER3	12.00
P3	PART3	S4	SUPPLIER4	12.50
P4	PART4	S4	SUPPLIER4	12.00
P5	PART5	S1	SUPPLIER1	10.00
P5	PART5	S2	SUPPLIER2	10.00
P5	PART5	S4	SUPPLIER4	11.00
P6	PART6	S4	SUPPLIER4	4.00
P6	PART6	S6	SUPPLIER6	4.00
P6	PART6	S8	SUPPLIER8	4.00
P7	PART7	S2	SUPPLIER2	2.00
P7	PART7	S4	SUPPLIER4	3.00
P7	PART7	S5	SUPPLIER5	3.50
P7	PART7	S6	SUPPLIER6	3.50
P8	PART8	S4	SUPPLIER4	5.00
P8	PART8	S6	SUPPLIER6	4.00
P8	PART8	S8	SUPPLIER8	3.00

**Syntax & Logic:** Nothing new. Considering this sample query as an extension of Sample Query 18.2 (which accessed the PART and PARTSUPP tables), we see that we have to include another table, the SUPPLIER table.

Looking at Figure 18.4 makes this easy. We append the new table (SUPPLIER) to the FROM-clause, and AND-connect another join-condition (S.SNO = PS.SNO) to the WHERE-clause.



**Syntax & Logic:** Alternatively, if we do not consider this sample query to be an extension of Sample Query 18.2, query analysis begins with: "What tables do I need?" Here, we want to display data found in PART, SUPPLIER, and PARTSUPP tables. Hence, you must reference these tables in the FROM-clause and specify the join-conditions.

Also, as with any inner-join, you must ask: "What rows might not appear in the join result?" In this example, we get all rows from the PARTSUPP table. However, there may be rows in a PART table (e.g., Part P2) that do not match PARTSUPP table; and there may be rows in the SUPPLIER table (e.g., Supplier S7) that do not match PARTSUPP.

**\* Terminology - "Three-Table Join":** We have used (and will continue to use) the term "three-table join," even though it is not very accurate. More precisely, a three-table join involves two two-table join-operations. From a logical perspective, this sample query initially joins the PART and PARTSUPP tables to produce an intermediate join-result. Then the second two-table join-operation joins this intermediate result with the SUPPLIER table to produce the final join-result. (Likewise, for a "four-table join", "five-table join," etc.)

## “Link” Table

The following sample query makes a minor adjustment to Sample Query 18.3. It does not display any columns from the STATE table. However, the SELECT statement must still access the STATE table to provide a path from REGION to CUSTOMER. In this circumstance, the STATE table is sometimes called a “link” table.

**Sample Query 18.5:** For all customers, display their CNO and CNAME values, along with their region’s RNO and RNAME values. Display these columns in the following left-to-right sequence: RNO, RNAME, CNO and CNAME. Sort the result by CNO within RNO.

```

SELECT R.RNO, R.RNAME,
       C.CNO, C.CNAME

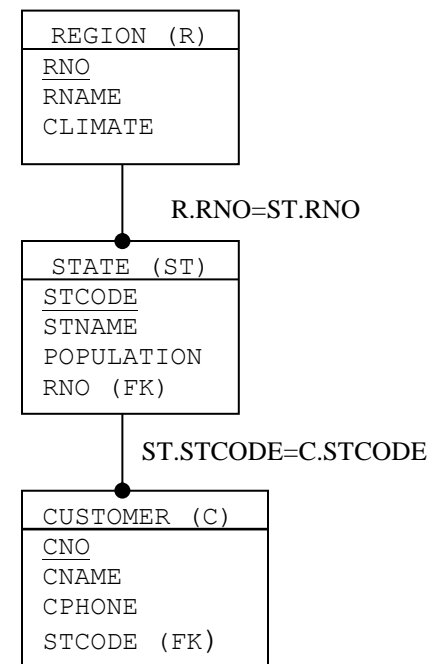
FROM   REGION R,
       STATE ST,      ← link table
       CUSTOMER C

WHERE  R.RNO = ST.RNO
AND    ST.STCODE = C.STCODE

ORDER BY R.RNO, C.CNO

```

RNO	RNAME	CNO	CNAME
1	NORTHEAST	100	PYTHAGORAS
1	NORTHEAST	110	EUCLID
1	NORTHEAST	200	HYPATIA
1	NORTHEAST	220	ZENO
1	NORTHEAST	230	BOLYAI
1	NORTHEAST	500	HILBERT
2	NORTHWEST	300	NEWTON
2	NORTHWEST	330	LEIBNIZ
2	NORTHWEST	400	DECARTES
2	NORTHWEST	440	PASCAL
3	SOUTHEAST	600	BOOLE
3	SOUTHEAST	660	CANTOR
3	SOUTHEAST	700	RUSSELL
3	SOUTHEAST	770	GODEL
4	SOUTHWEST	780	CHURCH
4	SOUTHWEST	800	VON NEUMANN
4	SOUTHWEST	880	TURING
4	SOUTHWEST	890	MANDELBROT



The following sample query makes a minor adjustment to the previous Sample Query 18.5. It only displays columns from the REGION table. It does not display any columns from the STATE table; and it does not display any columns from the CUSTOMER table. However, the query objective requires the SELECT statement to access the STATE and CUSTOMER tables.

**Sample Query 18.6:** For any region which has at least one customer, display the region's RNO and RNAME values. Sort the result by RNO.

```
SELECT DISTINCT R.RNO, R.RNAME
FROM   REGION R,
       STATE ST,
       CUSTOMER C
WHERE  R.RNO = ST.RNO
AND    ST.STCODE = C.STCODE
ORDER BY R.RNO
```

<u>RNO</u>	<u>RNAME</u>
1	NORTHEAST
2	NORTHWEST
3	SOUTHEAST
4	SOUTHWEST

**Syntax** Nothing new.

**Logic:** The keyword DISTINCT was specified because REGION is a parent-table and some parent-rows (e.g., NORTHEAST) could match on multiple states.

**Suggestion:** Execute this statement after removing the DISTINCT keyword. Observe the duplicate rows. Understand why these duplicates appeared in the result.

## C. Joining Four or More Tables

**Sample Query 18.7:** Extend Sample Query 18.4 to display one more column that resides in the STATE table: For every part that you can purchase from some supplier, display the part's PNO and PNAME values, the supplier's SNO and SNAME values, and the price (PSPRICE) you pay the supplier for the part. Also display the name of the state where the supplier is located. Display these columns in the following left-to-right sequence: STNAME, SNO, SNAME, PNO, PNAME, and PSPRICE. Sort the result by PNO within SNO within STNAME.

```
SELECT ST.STNAME, S.SNO, S.SNAME,
       P.PNO, P.PNAME, PS.PSPRICE

FROM   PART P,
       PARTSUPP PS,
       SUPPLIER S,
       STATE ST      ← new table

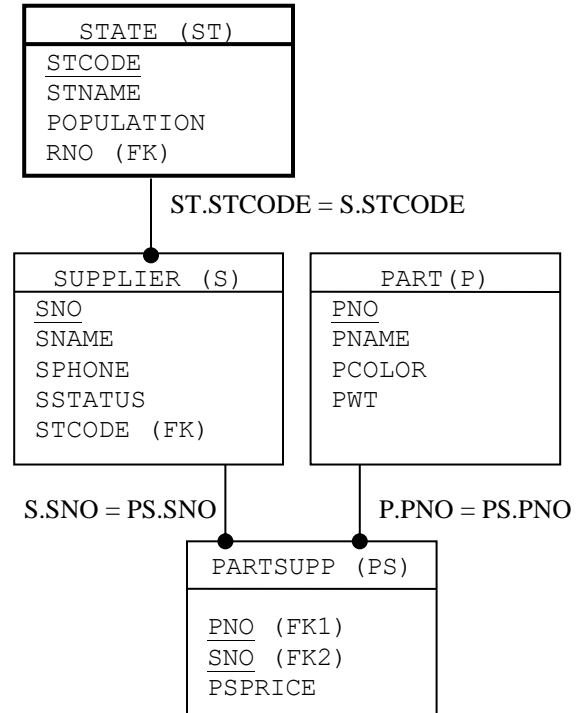
WHERE  P.PNO = PS.PNO
AND    PS.SNO = S.SNO
AND    ST.STCODE = S.STCODE    ← new join-condition

ORDER BY ST.STNAME, S.SNO, P.PNO
```

STNAME	SNO	SNAME	PNO	PNAME	PSPRICE
CONNECTICUT	S3	SUPPLIER3	P3	PART3	12.00
FLORIDA	S4	SUPPLIER4	P1	PART1	11.00
FLORIDA	S4	SUPPLIER4	P3	PART3	12.50
FLORIDA	S4	SUPPLIER4	P4	PART4	12.00
FLORIDA	S4	SUPPLIER4	P5	PART5	11.00
FLORIDA	S4	SUPPLIER4	P6	PART6	4.00
FLORIDA	S4	SUPPLIER4	P7	PART7	3.00
FLORIDA	S4	SUPPLIER4	P8	PART8	5.00
GEORIGIA	S5	SUPPLIER5	P7	PART7	3.50
MASSACHUSETTS	S1	SUPPLIER1	P5	PART5	10.00
MASSACHUSETTS	S2	SUPPLIER2	P1	PART1	10.50
MASSACHUSETTS	S2	SUPPLIER2	P5	PART5	10.00
MASSACHUSETTS	S2	SUPPLIER2	P7	PART7	2.00
OREGON	S8	SUPPLIER8	P6	PART6	4.00
OREGON	S8	SUPPLIER8	P8	PART8	3.00
WASHINGTON	S6	SUPPLIER6	P6	PART6	4.00
WASHINGTON	S6	SUPPLIER6	P7	PART7	3.50
WASHINGTON	S6	SUPPLIER6	P8	PART8	4.00

**Syntax & Logic:** Nothing new. Considering this sample query as an extension of Sample Query 18.4 (which accessed the SUPPLIER, PART, and PARTSUPP tables), we see that we have to include another table, the STATE table.

Looking at Figure 18.4 makes this easy. We append the new table (STATE) to the FROM-clause, and AND-connect join-condition (ST.STCODE = S.STCODE) to the WHERE-clause.



**Again, Syntax & Logic:** Alternatively, if we do not consider this sample query to be an extension of Sample Query 18.4, query analysis begins with: "What tables do I need?" Here, we want to display data found in the STATE, PART, SUPPLIER, and PARTSUPP tables. Hence, you must reference these tables in the FROM-clause and specify the corresponding join-conditions.

Again, consider non-matching rows. We only want information about those parts and suppliers referenced by PARTSUPP. Joining PART and SUPPLIER with PARTSUPP will exclude Part P2 and Supplier S7. Also, the final result excludes data about those states (New Mexico and Arizona) that do not have suppliers.

## Five-Table Join

**Sample Query 18.8:** Extend the previous Sample Query 18.7 to display one more column that resides in the REGION table. Include the name of the region (RNAME) where each supplier is located. Display the columns in the following left-to-right sequence: RNAME, STNAME, SNO, SNAME, PNO, PNAME, and PSPRICE. Sort the result by PNO within SNO within STNAME, within RNAME.

```
SELECT R.RNAME, ST.STNAME,
       S.SNO, S.SNAME, P.PNO, P.PNAME, PS.PSPRICE

FROM   PART P,
       PARTSUPP PS,
       SUPPLIER S,
       STATE ST,
       REGION R      ← new table

WHERE  P.PNO = PS.PNO
AND    PS.SNO = S.SNO
AND    ST.STCODE = S.STCODE
AND    R.RNO = ST.RNO      ← new join-condition

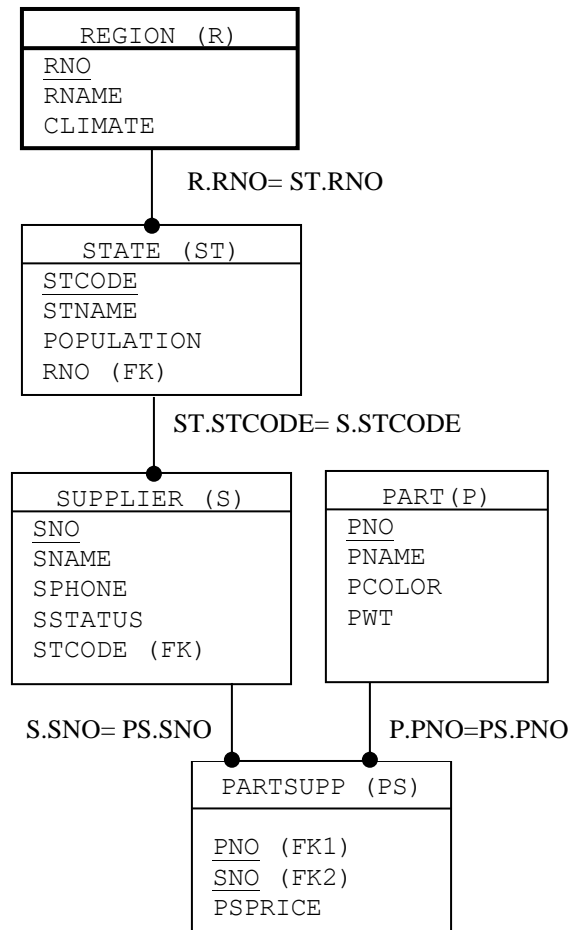
ORDER BY R.RNAME, ST.STNAME, S.SNO, P.PNO
```

RNAME	STNAME	SNO	SNAME	PNO	PNAME	PSPRICE
NORTHEAST	CONNECTICUT	S3	SUPPLIER3	P3	PART3	12.00
NORTHEAST	MASSACHUSETTS	S1	SUPPLIER1	P5	PART5	10.00
NORTHEAST	MASSACHUSETTS	S2	SUPPLIER2	P1	PART1	10.50
NORTHEAST	MASSACHUSETTS	S2	SUPPLIER2	P5	PART5	10.00
NORTHEAST	MASSACHUSETTS	S2	SUPPLIER2	P7	PART7	2.00
NORTHWEST	OREGON	S8	SUPPLIER8	P6	PART6	4.00
NORTHWEST	OREGON	S8	SUPPLIER8	P8	PART8	3.00
NORTHWEST	WASHINGTON	S6	SUPPLIER6	P6	PART6	4.00
NORTHWEST	WASHINGTON	S6	SUPPLIER6	P7	PART7	3.50
NORTHWEST	WASHINGTON	S6	SUPPLIER6	P8	PART8	4.00
SOUTHEAST	FLORIDA	S4	SUPPLIER4	P1	PART1	11.00
SOUTHEAST	FLORIDA	S4	SUPPLIER4	P3	PART3	12.50
SOUTHEAST	FLORIDA	S4	SUPPLIER4	P4	PART4	12.00
SOUTHEAST	FLORIDA	S4	SUPPLIER4	P5	PART5	11.00
SOUTHEAST	FLORIDA	S4	SUPPLIER4	P6	PART6	4.00
SOUTHEAST	FLORIDA	S4	SUPPLIER4	P7	PART7	3.00
SOUTHEAST	FLORIDA	S4	SUPPLIER4	P8	PART8	5.00
SOUTHEAST	GEORGIA	S5	SUPPLIER5	P7	PART7	3.50

**Syntax & Logic:** Nothing new. Considering this sample query as an extension of Sample Query 18.7 (which accessed the STATE, SUPPLIER, PART, and PARTSUPP tables), we have to include another table, the REGION table.

Looking at Figure 18.4 makes this easy. We append the new table (REGION) to the FROM-clause, and AND-connect the new join-condition (R.RNO = ST.RNO) to the WHERE-clause.

**Again, Syntax & Logic:** Alternatively, if we do not consider this sample query to be an extension of Sample Query 18.7, query analysis begins with: "What tables do I need?" Here, the goal is to display data found in the REGION, STATE, PART, SUPPLIER, and PARTSUPP tables. Hence, you must reference these tables in the FROM-clause and specify the corresponding join-conditions.



Again, consider non-matching rows. We only want information about those parts and suppliers referenced by PARTSUPP. Joining PART and SUPPLIER with PARTSUPP will exclude Part P2 and Supplier S7. Also, the final result excludes data about those states (New Mexico and Arizona) that do not have suppliers; and it excludes data about those regions (MIDWEST) that do not have any states.

**Atomic & Composite Primary Keys:** Consider the above data model that represents the tables accessed in this query. Observe that the primary-keys for the REGION, STATE, SUPPLIER, and PART tables are "atomic" keys. This means that the primary-key consists of just one column. Only the PARTSUPP table has a multi-column (composite) primary-key (PNO, SNO). This observation becomes relevant in the next sample query.



## Six-Table Join: Composite Join-Condition

**Sample Query 18.9:** Extend the previous Sample Query 18.8. Only consider parts that have been purchased. These are parts that are referenced in some line-item of some purchase order. Display these part names, the names of their suppliers, and related purchase-order numbers that are stored in the LINEITEM table. Also include the region name and state name of each supplier's location. Display these columns in the following left-to-right sequence: RNAME, STNAME, PNAME, SNAME, and PONO. Sort the result by LIPONO within PNAME within SNAME within STNAME within RNAME.

```
SELECT R.RNAME, ST.STNAME, S.SNAME, P.PNAME, LI.PONO
FROM   PART P,
       PARTSUPP PS,
       SUPPLIER S,
       STATE ST,
       REGION R,
       LINEITEM LI    ← new table
WHERE  PS.PNO = P.PNO
AND    PS.SNO = S.SNO
AND    S.STCODE = ST.STCODE
AND    ST.RNO = R.RNO
AND    PS.PNO = LI.PNO AND PS.SNO = LI.SNO ← new join-condition
ORDER BY R.RNAME, ST.STNAME, S.SNAME, P.PNAME, LI.PONO
```

**Syntax & Logic:** Nothing new. It is important to note that the new table (LINEITEM) is related to the PARTSUPP table via a *composite (compound) join-condition*.

PS.PNO = LI.PNO AND PS.SNO = LI.SNO

Examine Figure 18.4 to verify this fact. Failure to specify both components of the composite key in the join-condition will generate incorrect results.

Also note that no PARTSUPP columns were displayed. PARTSUPP was included as a link table between the PART and SUPPLIER tables.

RNAME	SNAME	SNAME	PNAME	PONO
NORTHEAST	CONNECTICUT	SUPPLIER3	PART3	11101
NORTHEAST	CONNECTICUT	SUPPLIER3	PART3	11102
NORTHEAST	CONNECTICUT	SUPPLIER3	PART3	11122
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	11108
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	11124
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	11154
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	11156
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	11157
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11101
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11109
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11122
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11148
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11154
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11156
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11158
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11160
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART5	11120
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART5	11139
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11109
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11130
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11142
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11152
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11159
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11160
NORTHWEST	OREGON	SUPPLIER8	PART6	11142
NORTHWEST	OREGON	SUPPLIER8	PART8	11149
NORTHWEST	OREGON	SUPPLIER8	PART8	11152
NORTHWEST	OREGON	SUPPLIER8	PART8	11153
NORTHWEST	OREGON	SUPPLIER8	PART8	11155
NORTHWEST	WASHINGTON	SUPPLIER6	PART6	11121
NORTHWEST	WASHINGTON	SUPPLIER6	PART8	11144
NORTHWEST	WASHINGTON	SUPPLIER6	PART8	11146
SOUTHEAST	FLORIDA	SUPPLIER4	PART1	11111
SOUTHEAST	FLORIDA	SUPPLIER4	PART1	11133
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11111
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11139
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11150
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11154
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11156
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11157
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11158
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11170
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	11102
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	11120
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	11124
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	11154
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	11170
SOUTHEAST	FLORIDA	SUPPLIER4	PART5	11130
SOUTHEAST	FLORIDA	SUPPLIER4	PART5	11141
SOUTHEAST	FLORIDA	SUPPLIER4	PART5	11152
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	11108
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	11130
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	11141
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	11150
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	11159
SOUTHEAST	FLORIDA	SUPPLIER4	PART7	11121
SOUTHEAST	FLORIDA	SUPPLIER4	PART7	11144
SOUTHEAST	FLORIDA	SUPPLIER4	PART8	11109
SOUTHEAST	FLORIDA	SUPPLIER4	PART8	11110
SOUTHEAST	FLORIDA	SUPPLIER4	PART8	11148
SOUTHEAST	GEORGIA	SUPPLIER5	PART7	11146
SOUTHEAST	GEORGIA	SUPPLIER5	PART7	11149

## Seven-Table Join

**Sample Query 18.10:** Extend the preceding Sample Query 18.9 to display the POSTATUS column in the PUR\_ORDER table. Display the columns in the following left-to-right sequence: RNAME, STNAME, SNAME, PNAME, PONO, and POSTATUS. Sort the result by LI.PONO within PNAME within SNAME within STNAME within RNAME.

```
SELECT R.RNAME, ST.STNAME, S.SNAME, P.PNAME,
       LI.PONO, PO.POSTATUS

FROM   PART P,
       PARTSUPP PS,
       SUPPLIER S,
       STATE ST,
       REGION R,
       LINEITEM LI,
       PUR_ORDER PO ← new table

WHERE  PS.PNO = P.PNO
AND    PS.SNO = S.SNO
AND    S.STCODE = ST.STCODE
AND    ST.RNO = R.RNO
AND    PS.PNO = LI.PONO AND PS.SNO = LI.SNO
AND    LI.PONO = PO.PONO ← new join-condition

ORDER BY R.RNAME, ST.STNAME, S.SNAME, P.PNAME, LI.PONO
```

**Syntax:** Nothing new. The FROM-clause references seven tables, and the WHERE-clause specifies six join-conditions.

**Logic:** Nothing new. To access the POSTATUS column, we include the PUR\_ORDER table and the corresponding join-condition (LI.PONO = PO.PONO).

Again, we did not display any columns from the PARTSUPP table which serves as a link table between the PART and SUPPLIER tables.

RNAME	SNAME	SNAME	PNAME	PONO	POSTATUS
NORTHEAST	CONNECTICUT	SUPPLIER3	PART3	11101	C
NORTHEAST	CONNECTICUT	SUPPLIER3	PART3	11102	P
NORTHEAST	CONNECTICUT	SUPPLIER3	PART3	11122	P
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	11108	C
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	11124	P
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	11154	C
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	11156	C
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	11157	P
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11101	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11109	P
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11122	P
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11148	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11154	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11156	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11158	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	11160	P
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART5	11120	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART5	11139	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11109	P
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11130	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11142	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11152	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11159	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	11160	P
NORTHWEST	OREGON	SUPPLIER8	PART6	11142	C
NORTHWEST	OREGON	SUPPLIER8	PART8	11149	P
NORTHWEST	OREGON	SUPPLIER8	PART8	11152	C
NORTHWEST	OREGON	SUPPLIER8	PART8	11153	P
NORTHWEST	OREGON	SUPPLIER8	PART8	11155	P
NORTHWEST	WASHINGTON	SUPPLIER6	PART6	11121	C
NORTHWEST	WASHINGTON	SUPPLIER6	PART8	11144	P
NORTHWEST	WASHINGTON	SUPPLIER6	PART8	11146	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART1	11111	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART1	11133	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11111	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11139	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11150	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11154	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11156	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11157	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11158	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	11170	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	11102	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	11120	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	11124	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	11154	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	11170	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART5	11130	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART5	11141	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART5	11152	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	11108	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	11130	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	11141	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	11150	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	11159	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART7	11121	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART7	11144	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART8	11109	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART8	11110	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART8	11148	C
SOUTHEAST	GEORGIA	SUPPLIER5	PART7	11146	C
SOUTHEAST	GEORGIA	SUPPLIER5	PART7	11149	P

## Eight-Table Join

One last time!

**Sample Query 18.11:** Extend the preceding Sample Query 18.10. Also display the name of the customer (CNAME) who completed each purchase order. Display the columns in the following left-to-right sequence: RNAME, STNAME, SNAME, PNAME, CNAME, PONO, and POSTATUS. Sort the result by LIPONO within CNAME within PNAME within SNAME within STNAME within RNAME.

```
SELECT R.RNAME, ST.STNAME, S.SNAME, P.PNAME, C.CNAME,
       LI.PONO, PO.POSTATUS

FROM   PART P,
       PARTSUPP PS,
       SUPPLIER S,
       STATE ST,
       REGION R,
       LINEITEM LI,
       PUR_ORDER PO,
       CUSTOMER C           ← new table

WHERE  PS.PNO = P.PNO
AND    PS.SNO = S.SNO
AND    S.STCODE = ST.STCODE
AND    ST.RNO = R.RNO
AND    PS.PNO = LI.PNO AND PS.SNO = LI.SNO
AND    LI.PONO = PO.PONO
AND    PO.CNO = C.CNO      ← new join-condition

ORDER BY R.RNAME, ST.STNAME, S.SNAME, P.PNAME,
         C.CNAME, LI.PONO
```

**Syntax & Logic:** Nothing new. The FROM-clause references eight tables; the WHERE-clause specifies seven join-conditions. CUSTOMER is the new table, and the corresponding join-condition is PO.CNO = C.CNO.

**\*\*\* "Mechanical" SQL:** The previous sample queries are intended to describe the "mechanics" of using PK-FK relationships to "follow the yellow brick road" when joining multiple tables. The following Section D presents a more conceptual framework for query analysis and related coding guidelines.

RNAME	STNAME	SNAME	PNAME	CNAME	PONO	POSTATUS
NORTHEAST	CONNECTICUT	SUPPLIER3	PART3	PYTHAGORAS	11101	C
NORTHEAST	CONNECTICUT	SUPPLIER3	PART3	PYTHAGORAS	11102	P
NORTHEAST	CONNECTICUT	SUPPLIER3	PART3	ZENO	11122	P
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	BOLYAI	11124	P
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	CANTOR	11154	C
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	EUCLID	11108	C
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	GODEL	11157	P
NORTHEAST	MASSACHUSETTS	SUPPLIER1	PART5	RUSSELL	11156	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	CANTOR	11154	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	EUCLID	11109	P
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	PASCAL	11148	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	PYTHAGORAS	11101	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	RUSSELL	11156	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	TURING	11160	P
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	VON NEUMANN	11158	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART1	ZENO	11122	P
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART5	LEIBNIZ	11139	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART5	ZENO	11120	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	BOOLE	11152	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	DECARTES	11142	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	EUCLID	11109	P
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	NEWTON	11130	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	TURING	11159	C
NORTHEAST	MASSACHUSETTS	SUPPLIER2	PART7	TURING	11160	P
NORTHWEST	OREGON	SUPPLIER8	PART6	DECARTES	11142	C
NORTHWEST	OREGON	SUPPLIER8	PART8	BOOLE	11152	C
NORTHWEST	OREGON	SUPPLIER8	PART8	BOOLE	11153	P
NORTHWEST	OREGON	SUPPLIER8	PART8	CANTOR	11155	P
NORTHWEST	OREGON	SUPPLIER8	PART8	PASCAL	11149	P
NORTHWEST	WASHINGTON	SUPPLIER6	PART6	ZENO	11121	C
NORTHWEST	WASHINGTON	SUPPLIER6	PART8	DECARTES	11144	P
NORTHWEST	WASHINGTON	SUPPLIER6	PART8	PASCAL	11146	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART1	HYPATIA	11111	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART1	NEWTON	11133	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	CANTOR	11154	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	GODEL	11157	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	HILBERT	11150	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	HYPATIA	11111	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	LEIBNIZ	11139	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	RUSSELL	11156	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	TURING	11170	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART3	VON NEUMANN	11158	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	BOLYAI	11124	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	CANTOR	11154	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	PYTHAGORAS	11102	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	TURING	11170	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART4	ZENO	11120	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART5	BOOLE	11152	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART5	LEIBNIZ	11141	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART5	NEWTON	11130	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	EUCLID	11108	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	HILBERT	11150	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	LEIBNIZ	11141	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	NEWTON	11130	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART6	TURING	11159	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART7	DECARTES	11144	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART7	ZENO	11121	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART8	EUCLID	11109	P
SOUTHEAST	FLORIDA	SUPPLIER4	PART8	HYPATIA	11110	C
SOUTHEAST	FLORIDA	SUPPLIER4	PART8	PASCAL	11148	C
SOUTHEAST	GEORGIA	SUPPLIER5	PART7	PASCAL	11146	C
SOUTHEAST	GEORGIA	SUPPLIER5	PART7	PASCAL	11149	P

## D. Query Analysis & Coding Guidelines

Consistent with the tutorial theme of this book, we have presented sample queries before describing general concepts. Having presenting examples of multi-table inner-join operations, we can now offer some general advice about analyzing and coding such queries. This section attempts to generalize the query analysis and coding techniques illustrated by the previous sample queries.

### **Getting Started: Access & Understand your Data Model**

**Find your data model.** If it exists, Appendix 18D can help you find it.

**No data model?** Can't find your data model? Maybe it's a big secret, or maybe it doesn't exist. Then Appendix 18E (Reverse Engineering) will help you build it.

**Your data model is not hierarchical orientated:** If, after finding/building your data model, it looks like a spaghetti model (i.e., similar to Figure 18.2), Appendix 18F will show you how to redraw this model in a hierarchal form (similar to Figure 18.3).

**Understand your data model:** Understand the semantics of your data model, especially as it relates to your query objective. Hopefully, you have access to documentation that offers a "business perspective" of your model.

**Understand your query objective:** Sometimes, articulating a query objective is not easy. Technical people frequently criticize business users for failing to articulate precise, concise, and complete descriptions of their query objectives. However, this can be difficult. Sometimes, it can be helpful to present and explain your data model to the business user. It is very important that you describe this model as a "*business*" model which is the foundation of your database design. Hopefully, this will help you and the business user to collectively formulate a valid query objective.

## "Cookbook" Method for Coding SELECT Statements

This section is overkill for experienced SQL users who already know their data. However, rookies should read this section. Here we make explicit some ideas that were implicit in previous sample queries. The following query objective is used as an example

For every part that can be purchased from some supplier, display the part's number and name, the supplier's number and name, and the name of the region where each supplier resides. Display the columns in a left-to-right sequence as PNO, PNAME, SNO, SNAME, RNAME. Sort the result by SNO within PNO.

For tutorial purposes, this query objective does not require any restrictions, grouping, or summarizing.

### Phase-I: Query Analysis

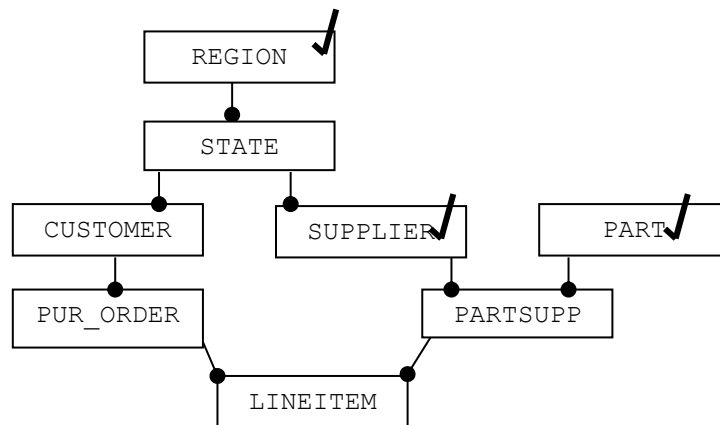
**Step-1:** Reference your data model. Check off tables that are relevant to your query objective. This query objective wants to display the following columns.

RNAME from the REGION table.

PNO and PNAME from the PART table.

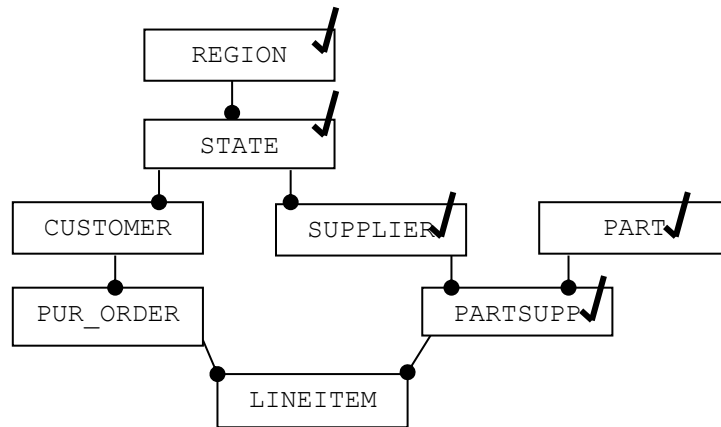
SNO and SNAME from the SUPPLIER table.

Hence, we check the REGION, SUPPLIER and PART tables



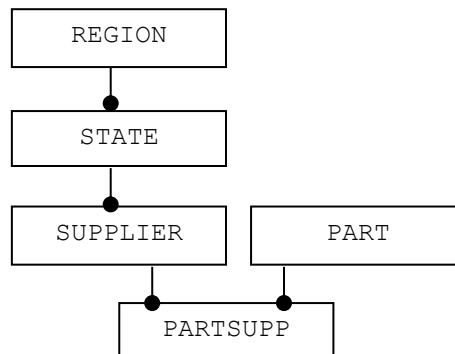


**Step-2:** Identify and check off link tables: Here the STATE and PARTSUPP tables are link tables.



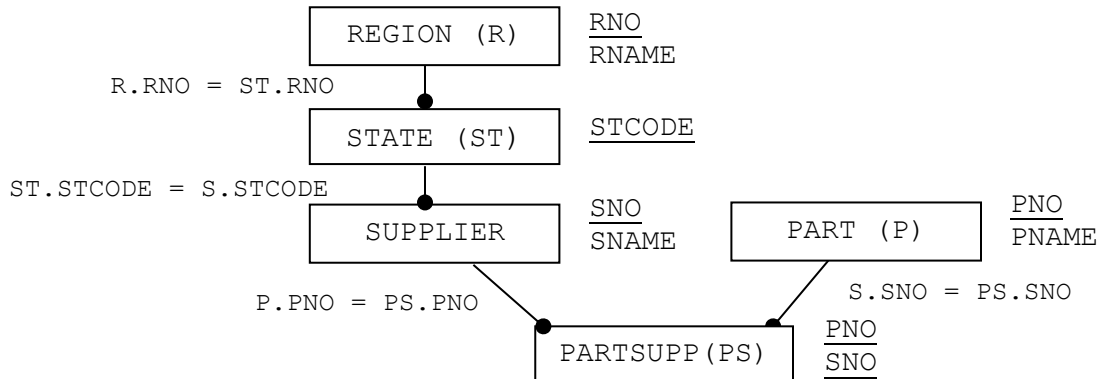
Preview: Each checkmark indicates that the table will be referenced in the FROM-clause.

**Step-3:** Simplify the data model. Only display relevant tables and relationships. Optionally reposition tables to highlight hierarchal orientation.



**Step-4:** Reference your data model (Figure 18.4) to include details about:

- table aliases
- primary-keys
- relevant columns
- join-conditions



## Phase-II: Coding SELECT Statement

Code FROM-clause: Try to follow some hierarchical or near-hierarchical coding pattern.

```
FROM REGION R,  
STATE ST,  
SUPPLIER S,  
PARTSUPP PS,  
PART P
```

Code Join-Conditions: Specify join-conditions in same order as tables specified in FROM-clause.

```
WHERE R.RNO = ST.RNO  
AND ST.STCODE = S.STCODE  
AND S.SNO = PS.SNO  
AND PS.PNO = P.PNO
```

Code SELECT-clause:

```
SELECT P.PNO, P.PNAME, S.SNO, S.SNAME, R.RNAME
```

Code ORDER BY clause:

```
ORDER BY P.PNO, S.SNO
```

Assemble the above code fragments:

```
SELECT P.PNO, P.PNAME, S.SNO, S.SNAME, R.RNAME  
FROM REGION R,  
STATE ST,  
SUPPLIER S,  
PARTSUPP PS,  
PART P  
WHERE R.RNO = ST.RNO  
AND ST.STCODE = S.STCODE  
AND S.SNO = PS.SNO  
AND PS.PNO = P.PNO  
ORDER BY P.PNO, S.SNO
```

Note: This join-result could serve as an intermediate result for a more complex query objective that requires you to specify built-in functions, restrictions, grouping, or summarizing. See the following Section-E.

## Exercises

### 3-Table Join Operations

- 18A. Display the name of every customer, followed by the name of the state, and the name of the region where the customer is located. Display the result in ascending sequence by customer name
- 18B. Display the name of every supplier, followed by the names of the region and state where the supplier is located. Display the result in ascending sequence by supplier name.
- 18C. For every supplier who can sell your organization some part, display the supplier number and name, the part number and name, and the price you will pay to the supplier for the part. Display the columns in the following left-to-right sequence: SNO, SNAME, PNO, PNAME, and PSPRICE. Sort the result by SNO, PNO.

### 4-Table Join Operations

- 18D. We are only interested in customers who have one or more purchase orders. Display the customer's number and name, followed by the name of the state and the name of the region where the customer is located, followed by the date of the purchase order. Display the result in ascending sequence by purchase order date within customer number.
- 18E. For every part that you can purchase from some supplier, display the part number and name, followed by the supplier number and name, followed by the name of the state where the supplier is located, followed by the price you will pay (PSPRICE) to the supplier for the part. Sort the result by PNO, SNO.

## 5-Table Join Operations

- 18F. We are only interested in customers who have purchased parts. (These are customers who have completed a purchase order with line items. Recall that some purchase orders may not have any line items.) Display the customer's name, followed by the name of the state and the name of the region where the customer is located, followed by the date of the purchase order, followed by the part number of the purchased part. Display the result in ascending sequence by CNAME, PODATE, PNO.
- 18G. We are only interested in parts that you can purchase from some supplier. For these parts, display the part number and name, followed by the price you will pay of the part, followed by the number and name of the supplier who will sell you this part at this price. Also include the names of the state and region where the supplier is located. Display the result in ascending sequence by price with part number.

## 6-Table and 7-Table Join Operations

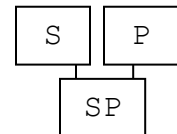
- 18H. We are only interested in customers who have purchased parts. (These are customers who have completed a purchase order with line items. Recall that some purchase orders may not have any line items.) Display the customer's name, followed by the names of the state and region where the customer is located, followed by the purchase order number, followed by the part number, line-item price (LIPRICE) and purchase price (PSPRICE) of the part. Display the result in ascending sequence by CNAME, PONO, PNO.
- 18I. This example extends the previous Exercise 18H. We are only interested in customers who have purchased parts. (These are customers who have completed a purchase order with line items. Recall that some purchase orders may not have any line items.) Display the customer's name, followed by the names of the state and region where the customer is located, followed by the purchase order number, followed by the part number *and name*, followed by the line-item price (LIPRICE) and purchase price (PSPRICE) of the part. Display the result in ascending sequence by CNAME, PONO, PNO.

## E. Join with Other Operations

Previous sample queries and exercises that specified multi-table join-operations did not include any restriction, grouping, or summary operations. Sample queries presented in this section include these operations. The following sample query joins three tables and specifies a restriction.

**Sample Query 18.12:** We are only interested in parts with a PSPRICE value that exceeds \$10.00. For every such part that you can purchase from some supplier, display the part's number and name, the supplier's number and name, and the supplier's price for the part. Display the columns in the following left-to-right sequence: PNO, PNAME, SNO, SNAME, and PSPRICE. Sort the result by SNO within PNO.

```
SELECT P.PNO, P.PNAME,
       S.SNO, S.SNAME,
       PS.PSPRICE
FROM   PART P,
       PARTSUPP PS,
       SUPPLIER S
WHERE  P.PNO = PS.PNO
AND    PS.SNO = S.SNO } ← join-conditions
AND PS.PSPRICE > 10.00 ← restriction
ORDER BY P.PNO, S.SNO
```



PNO	PNAME	SNO	SNAME	PSPRICE
P1	PART1	S2	SUPPLIER2	10.50
P1	PART1	S4	SUPPLIER4	11.00
P3	PART3	S3	SUPPLIER3	12.00
P3	PART3	S4	SUPPLIER4	12.50
P4	PART4	S4	SUPPLIER4	12.00
P5	PART5	S4	SUPPLIER4	11.00

**Syntax:** Nothing new. The restriction (`PS.PSPRICE > 10.00`) is AND-connected to the join-conditions.

**Logic:** Logically, the system performs the three-table join to form an intermediate join-result. Then it applies the restriction operation to this intermediate result to produce the final result.

The following sample query executes four join-operations to join five tables, followed by two restriction-operations.

**Sample Query 18.13:** We are only interested in pink parts that you can purchase from some supplier located in the Northeast Region. Display the part number and name, the supplier number and name, and the price you will pay to the supplier for the part. Also display the name of the state where the supplier is located. Display the columns in the following left-to-right sequence: STNAME, SNO, SNAME, PNO, PNAME, and PSPRICE. Sort the result by PNO within SNO within STNAME.

```

SELECT ST.STNAME, S.SNO, S.SNAME,
       P.PNO, P.PNAME, PS.PSPRICE

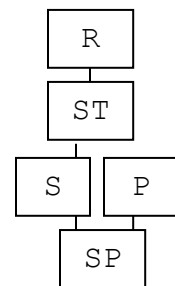
FROM   REGION R, STATE ST, SUPPLIER S,
       PARTSUPP PS, PART P

WHERE  R.RNO = ST.RNO
AND    ST.STCODE = S.STCODE } ← join-conditions
AND    S.SNO = PS.SNO
AND    PS.PNO = P.PNO

AND    P.PCOLOR = 'PINK ' } ← restrictions
AND    R.RNAME = 'NORTHEAST '

ORDER BY ST.STNAME, S.SNO, P.PNO

```



STNAME	SNO	SNAME,	PNO	PNAME	PSPRICE
CONNECTICUT	S3	SUPPLIER3	P3	PART3	12.00
MASSCHUSETTS	S2	SUPPLIER2	P7	PART7	2.00

**Syntax & Logic:** Nothing new. Notice that the query objective did not ask you to display RNAME and PCOLOR values. This means that you cannot “eyeball” the result table to verify that you only selected pink parts from the Northeast Region. When testing your statement, consider displaying the RNAME and PCOLOR columns. You can easily remove them after you are satisfied that your logic is correct. Also, many front-end query/reporting tools allow you to remove columns from a report without re-executing the SELECT statement.



## Restriction, Grouping, and Summarizing on Join Result

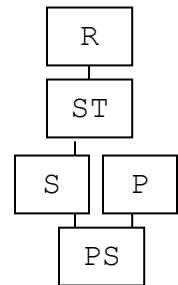
The following sample query performs a five-table join, but only displays data from two tables. It also forms groups and calculates group averages.

**Sample Query 18.14:** Exclude from consideration any pink part where the supplier's price exceeds \$11.00. For each region with a supplier who supplies at least one part, display the region number and name, followed by the average price you pay for parts available in that region.

```
SELECT R.RNO, R.RNAME, AVG (PS.PSPRICE)
FROM REGION R, STATE ST, SUPPLIER S,
      PARTSUPP PS, PART P
WHERE R.RNO = ST.RNO
AND ST.STCODE = S.STCODE
AND S.SNO = PS.SNO
AND P.PNO = PS.PNO

AND NOT (P.PCOLOR = 'PINK' AND PS.PSPRICE > 11.00)

GROUP BY R.RNO, R.RNAME
```



RNO	RNAME	AVG(PSPRICE)
1	NORTHEAST	8.12
2	NORTHWEST	3.70
3	SOUTHEAST	7.07

**Syntax:** Nothing New.

**Logic:** This modestly complex query entails a five-table join. Region numbers and names are in REGION. Supplier prices (PSPRICE) are in PARTSUPP. Figure 18.4 shows that the query path from REGION to PARTSUPP travels through STATE and SUPPLIER. Hence, we must access REGION, STATE, SUPPLIER, and PARTSUPP. Then, our navigational logic requires "one more hop" to PART because the restriction references PCOLOR.

After the applying the join and restriction operations, the system groups by RNO and RNAME and calculates the average PSPRICE for each group.

## Exercises

Begin each exercise by determining which tables your SELECT statement must reference. Then draw a skeleton data model that represents these tables.

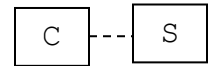
- 18J. We are only interested in customers having names that begin with the letter "B". Display the customers' name, followed by the name of the state and the name of the region where the customer is located. Display the result in ascending sequence by customer name.
- 18K. We are only interested in suppliers who are located in Florida (STCODE = 'FL') who sell parts having a weight (PWT) that is less than 20 pounds. Display each supplier's number and name, followed by the part number, name, and weight, followed by the price you will pay to the supplier for the part. Sort the result by SNO, PNO.
- 18L. The basic objective is to determine total number of parts each customer has purchased. This amount is equal to sum of the LINEITEM.QTY values for each customer. Display the customer's number followed by the total number of parts purchased. Sort the result in ascending sequence by customer number.
- 18M. This example is a minor modification to the previous exercise (18L). Along with the customer number, we also want to display the customer name and state code. (This exercise is really a review of grouping.)
- 18N. This example extends the previous exercise (18M). We only want to display information about those customers who have purchased a total of more than 100 parts.

## Join on Non-PK-FK Relationship

The following sample query executes a two-table join-operation that *is not based a primary-key/foreign-key relationship*. (This kind of join-operation is illustrated by a dashed-line in the following data model.)

**Sample Query 18.15.1:** What customers and suppliers reside in the same state? For each pair of co-located customers and suppliers, display the state code, followed by the customer's number and name, followed by the supplier's number and name. Sort the result by SNO within CNO within STCODE.

```
SELECT C.STCODE, C.CNO, C.CNAME,
       S.SNO, S.SNAME
FROM   CUSTOMER C, SUPPLIER S
WHERE  C.STCODE = S.STCODE ← join-condition
ORDER BY C.STCODE, C.CNO, S.SNO
```



STCODE	CNO	CNAME	SNO	SNAME
FL	600	BOOLE	S4	SUPPLIER4
FL	660	CANTOR	S4	SUPPLIER4
GE	700	RUSSELL	S5	SUPPLIER5
GE	770	GODEL	S5	SUPPLIER5
MA	100	PYTHAGORAS	S1	SUPPLIER1
MA	100	PYTHAGORAS	S2	SUPPLIER2
MA	110	EUCLID	S1	SUPPLIER1
MA	110	EUCLID	S2	SUPPLIER2
MA	200	HYPATIA	S1	SUPPLIER1
MA	200	HYPATIA	S2	SUPPLIER2
MA	220	ZENO	S1	SUPPLIER1
MA	220	ZENO	S2	SUPPLIER2
. . . . .				
{another 10 rows for a total of 22 rows}				

**Syntax and Logic:** Nothing new. Note that the MTPCH data model (Figure 18.4) does not show any *direct* relationship between the CUSTOMER and SUPPLIER tables. Also, neither C.STCODE nor S.STCODE is the primary-key of its table.

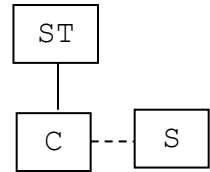
The following Sample Query 18.15.2 extends this sample query into a three-table join-operation.

The following sample query enhances the previous Sample Query 18.15.1 to display state names instead of state codes. This requires a three-table join.

**Sample Query 18.15.2:** What customers and suppliers reside in the same state? For each pair of collocated customers and suppliers, display the state's name, followed by the customer's number and name, followed by the supplier's number and name. Sort the result by SNO within CNO within STNAME.

```
SELECT ST.STNAME, C.CNO, C.CNAME, S.SNO, S.SNAME
FROM   STATE ST, CUSTOMER C, SUPPLIER S
WHERE  ST.STCODE = C.STCODE
AND    C.STCODE = S.STCODE

ORDER BY ST.STNAME, C.CNO, S.SNO
```



<u>STNAME</u>	<u>CNO</u>	<u>CNAME</u>	<u>SNO</u>	<u>SNAME</u>
FLORIDA	600	BOOLE	S4	SUPPLIER4
FLORIDA	660	CANTOR	S4	SUPPLIER4
GEORGIA	700	RUSSELL	S5	SUPPLIER5
GEORGIA	770	GODEL	S5	SUPPLIER5
MASSACHUSETTS	100	PYTHAGORAS	S1	SUPPLIER1
MASSACHUSETTS	100	PYTHAGORAS	S2	SUPPLIER2
MASSACHUSETTS	110	EUCLID	S1	SUPPLIER1
MASSACHUSETTS	110	EUCLID	S2	SUPPLIER2
MASSACHUSETTS	200	HYPATIA	S1	SUPPLIER1
MASSACHUSETTS	200	HYPATIA	S2	SUPPLIER2
MASSACHUSETTS	220	ZENO	S1	SUPPLIER1
MASSACHUSETTS	220	ZENO	S2	SUPPLIER2

. . . . .  
{another 10 rows for a total of 22 rows}

**Syntax and Logic:** This statement references the STATE table to access the STNAME column. It joins the STATE and CUSTOMER on their PK-FK relationship, and joins CUSTOMER and SUPPLIER and its common column (STCODE).

Each result table row in Sample Queries 18.15.1 and 18.15.2 displayed information about a customer and a supplier who reside in the same state. Some of these co-located pairs of customers and suppliers included a customer who never purchased a part from the supplier. The following sample query excludes such pairs of customers and suppliers.

**Sample Query 18.16:** Which customers have purchased at least one part that was sold by a supplier who resides in the same state as the customer? For each such pair of co-located customers and suppliers, display the state code, followed by the customer's number and name, followed by the supplier's number and name. Sort the result by SNO within CNO within STCODE.

```
SELECT DISTINCT C.STCODE,
               C.CNO, C.CNAME, S.SNO, S.SNAME

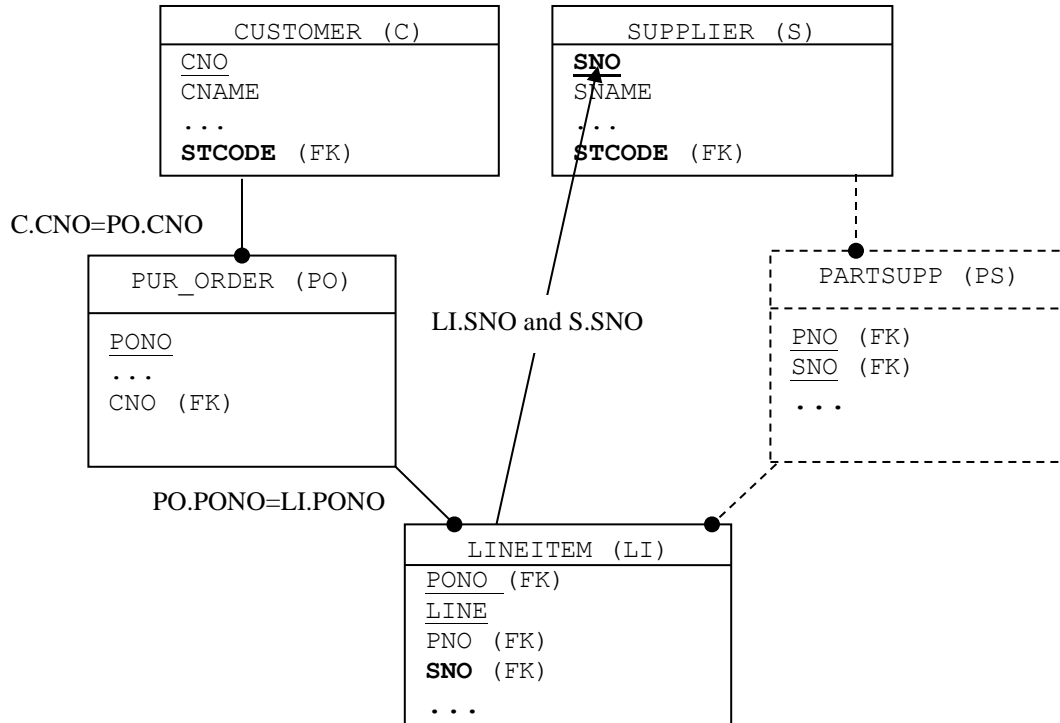
FROM   CUSTOMER C,
       PUR_ORDER PO,
       LINEITEM LI,
       SUPPLIER S

WHERE  C.CNO = PO.CNO
AND    PO.PONO = LI.PONO
AND    LI.SNO = S.SNO
AND    C.STCODE = S.STCODE

ORDER BY C.STCODE, C.CNO, S.SNO
```

<u>STCODE</u>	<u>CNO</u>	<u>CNAME</u>	<u>SNO</u>	<u>SNAME</u>
FL	600	BOOLE	S4	SUPPLIER4
FL	660	CANTOR	S4	SUPPLIER4
MA	100	PYTHAGORAS	S2	SUPPLIER2
MA	110	EUCLID	S1	SUPPLIER1
MA	110	EUCLID	S2	SUPPLIER2
MA	220	ZENO	S2	SUPPLIER2
MA	230	BOLYAI	S1	SUPPLIER1
WA	400	DECARTES	S6	SUPPLIER6
WA	440	PASCAL	S6	SUPPLIER6

**Logic:** Query analysis is described by referencing the following data model. Consider a single customer (CUSTOMER) who resides in some state (identified by a STCODE value). If this customer issued a purchase-order (PUR\_ORDER) with a line-item (LINEITEM) that specifies a combination of PNO-SNO values where the SNO value identifies a supplier (SUPPLIER) with the same STCODE value, then this customer purchased a part from a collocated supplier. Therefore, join-operations must traverse following tables



Notice there is no *direct* PK-FK relationship between the LINEITEM and SUPPLIER tables. The join-condition (AND LI.SNO = S.SNO) is based upon a valid indirect relationship between these tables. The arrow in the above figure represents this coding "shortcut" which bypasses the PARTSUPP table that is not referenced in the SELECT statement.

Be careful with shortcuts. This shortcut is valid because:

LINEITEM.SNO matches PARTSUPP.SNO which matches SUPPLIER.SNO

Also, notice that we must specify DISTINCT because the customer could make multiple purchases from the same collocated supplier.

## F. JOIN-ON Syntax

We review the JOIN-ON syntax by joining the PART and PARTSUPP tables.

```
SELECT *
FROM PART P INNER JOIN PARTSUPP PS ON P.PNO = PS.PNO
```

The JOIN-ON syntax can join any number of tables. The following statement extends the above statement to include the SUPPLIER table within a three-table join.

```
SELECT *
FROM PART P INNER JOIN PARTSUPP PS ON P.PNO = PS.PNO
          INNER JOIN SUPPLIER S   ON PS.SNO = S.SNO
```

Also, parentheses can be specified to enhance readability.

```
SELECT *
FROM (PART P INNER JOIN PARTSUPP PS ON P.PNO = PS.PNO)
     INNER JOIN SUPPLIER S   ON PS.SNO = S.SNO
```

Below, we rewrite Sample Queries 18.7 (four-table join) and 18.8 (five-table join) using the JOIN-ON syntax.

### Sample Query 18.7

```
SELECT ST.STNAME, S.SNO, S.SNAME, P.PNO, P.PNAME, PS.PSPRICE
FROM PART P INNER JOIN PARTSUPP PS ON P.PNO = PS.PNO
          INNER JOIN SUPPLIER S   ON PS.SNO = S.SNO
          INNER JOIN STATE ST     ON S.STCODE = ST.STCODE
ORDER BY ST.STNAME, S.SNO, P.PNO
```

### Sample Query 18.8

```
SELECT R.RNAME, ST.STNAME, S.SNO, S.SNAME,
       P.PNO, P.PNAME, PS.PSPRICE
FROM PARTSUPP PS INNER JOIN PART P      ON PS.PNO = P.PNO
          INNER JOIN SUPPLIER S   ON PS.SNO = S.SNO
          INNER JOIN STATE ST     ON S.STCODE = ST.STCODE
          INNER JOIN REGION R     ON ST.RNO = R.RNO
ORDER BY R.RNAME, ST.STNAME, S.SNO, P.PNO
```

### **Exercise:**

180. Rewrite Sample Queries 18.9 and 18.10 using the JOIN-ON syntax.

## Inner-Join with Restriction

The previous chapter discussed inner-join and restriction within the context of two-table join-operations. Here we extend the same concepts to multi-table join-operations.

*The following four statements are equivalent. They use the JOIN-ON syntax to satisfy Sample Query 18.12. Statement-18.12a rewrites the original FROM-WHERE statement using the JOIN-ON syntax. Statement-18.12b specifies the restriction (PS.PSPRICE > 10.00) using AND instead of WHERE. Statement-18.12c moves the restriction immediately after the join-condition for PART and PARTSUPP to “visually associate” the restriction on PARTSUPP with this join-operation. (This is more than a mere visual association. See the next page.) Statement-18.12d adds parentheses to Statement 18.12c.*

### Statement-18.12a

```
SELECT P.PNO, P.PNAME, S.SNO, S.SNAME, PS.PSPRICE
FROM   PART P INNER JOIN PARTSUPP PS ON P.PNO = PS.PNO
        INNER JOIN SUPPLIER S   ON PS.SNO = S.SNO
WHERE PS.PSPRICE > 10.00
ORDER BY P.PNO, S.SNO
```

### Statement-18.12b

```
SELECT P.PNO, P.PNAME, S.SNO, S.SNAME, PS.PSPRICE
FROM   PART P INNER JOIN PARTSUPP PS ON P.PNO = PS.PNO
        INNER JOIN SUPPLIER S   ON PS.SNO = S.SNO
AND PS.PSPRICE > 10.00
ORDER BY P.PNO, S.SNO
```

### Statement-18.12c

```
SELECT P.PNO, P.PNAME, S.SNO, S.SNAME, PS.PSPRICE
FROM   PART P INNER JOIN PARTSUPP PS ON P.PNO = PS.PNO
        AND PS.PSPRICE > 10.00
        INNER JOIN SUPPLIER S   ON PS.SNO = S.SNO
ORDER BY P.PNO, S.SNO
```

### Statement-18.12d

```
SELECT P.PNO, P.PNAME, S.SNO, S.SNAME, PS.PSPRICE
FROM   (PART P INNER JOIN PARTSUPP PS ON P.PNO = PS.PNO
        AND PS.PSPRICE > 10.00)
        INNER JOIN SUPPLIER S   ON PS.SNO = S.SNO
ORDER BY P.PNO, S.SNO
```



## ON-CLAUSE with Compound Join-Condition

The FROM-clause in the preceding Statement-18.12c specified AND PS.PSPRICE > 10.00 immediately after ON P.PNO = PS.PNO.

```
FROM PART P INNER JOIN PARTSUPP PS ON P.PNO = PS.PNO
                                AND PS.PSPRICE > 10.00 ←
INNER JOIN SUPPLIER S ON PS.SNO = S.SNO
```

On the preceding page, we stated that AND PS.PSPRICE > 10.00 is visually associated with the join of PART and PARTSUPP. This is true. However, this is more than visual association. To be precise, the ON-clause for the join of PART and PARTSUPP is a *compound join-condition*.

```
PART P INNER JOIN PARTSUPP PS
ON P.PNO = PS.PNO AND PS.PSPRICE > 10.00
```

This means that the PSPRICE > 10.00 comparison is implemented *during the join-operation*. This differs from Statements 18.12a and 18.12b where the PSPRICE > 10.00 comparison is (logically) executed *after* the PART, PARTSUPP, and SUPPLIER tables have been joined.

Efficiency Observation (Optional Reading): Again, Statements 18-12a, 18-12b, 18-12c, and 18-12d are all equivalent. However, you might observe that Statements 18.12c and 18.12d could be more efficient than Statements 18.12a and 18.12b because they execute of the PS.PSPRICE > 10.00 condition earlier, and hence they would return a smaller join-result for the join of the PART and PARTSUPP tables. A detail discussion of this efficiency consideration was presented in Appendix 17C.

**Another Example:** The following four statements are equivalent. They use the JOIN-ON syntax to satisfy Sample Query 18.13.

Statement-18.13a

```
SELECT R.RNAME, ST.STNAME, S.SNO, S.SNAME,
       P.PNO, P.PNAME, PS.PSPRICE
FROM   REGION R INNER JOIN STATE ST      ON R.RNO = ST.RNO
       INNER JOIN SUPPLIER S      ON ST.STCODE = S.STCODE
       INNER JOIN PARTSUPP PS     ON S.SNO = PS.SNO
       INNER JOIN PART P          ON PS.PNO = P.PNO
WHERE P.PCOLOR = 'PINK'
AND    R.RNAME = 'NORTHEAST'
ORDER BY R.RNAME, ST.STNAME, S.SNO, P.PNO
```

Statement-18.13b

```
SELECT R.RNAME, ST.STNAME, S.SNO, S.SNAME,
       P.PNO, P.PNAME, PS.PSPRICE
FROM   REGION R INNER JOIN STATE ST      ON R.RNO = ST.RNO
       INNER JOIN SUPPLIER S      ON ST.STCODE = S.STCODE
       INNER JOIN PARTSUPP PS     ON S.SNO = PS.SNO
       INNER JOIN PART P          ON PS.PNO = P.PNO
AND    P.PCOLOR = 'PINK'
AND    R.RNAME = 'NORTHEAST'
ORDER BY R.RNAME, ST.STNAME, S.SNO, P.PNO
```

Statement-18.13c

```
SELECT R.RNAME, ST.STNAME, S.SNO, S.SNAME,
       P.PNO, P.PNAME, PS.PSPRICE
FROM   REGION R INNER JOIN STATE ST      ON R.RNO = ST.RNO
       AND R.RNAME = 'NORTHEAST'
       INNER JOIN SUPPLIER S      ON ST.STCODE = S.STCODE
       INNER JOIN PARTSUPP PS     ON S.SNO = PS.SNO
       INNER JOIN PART P          ON PS.PNO = P.PNO
       AND P.PCOLOR = 'PINK'
ORDER BY R.RNAME, ST.STNAME, S.SNO, P.PNO
```

Statement-18.13d

```
SELECT R.RNAME, ST.STNAME, S.SNO, S.SNAME,
       P.PNO, P.PNAME, PS.PSPRICE
FROM   ((REGION R INNER JOIN STATE ST      ON R.RNO = ST.RNO
       AND R.RNAME = 'NORTHEAST')
       INNER JOIN SUPPLIER S      ON ST.STCODE = S.STCODE)
       INNER JOIN PARTSUPP PS     ON S.SNO = PS.SNO)
       INNER JOIN PART P          ON PS.PNO = P.PNO
       AND P.PCOLOR = 'PINK'
ORDER BY R.RNAME, ST.STNAME, S.SNO, P.PNO
```

## SUMMARY

This chapter did not introduce any new SQL keywords or techniques. All examples merely extended two-table join concepts and techniques to satisfy queries that required joining three or more tables.

**SQL Redundancy:** This chapter demonstrated that, given a query objective, you may be able to code many different SELECT statements to satisfy the query objective. Future chapters will present even more ways to satisfy many of this chapter's sample queries. This "SQL redundancy" can lead to maintenance problems. You will use your preferred coding style when writing your own SELECT statements. However, you might be asked to change statements written by other users. Therefore, you should understand all variations of coding SELECT statements.

## Summary Exercises

- 18P. Consider all customers. Display each customer's location (region name and state name) followed by the customer's name. Display the result in ascending sequence by customer name within region name.
- 18Q. Only consider regions that have suppliers. Display the name of the region name followed by the name of the supplier. Display the result in ascending sequence by supplier name within region name.
- 18R. Only consider Massachusetts (STCODE='MA') customers that have completed a purchase order. For each such customer, display the customer's name, and the number of purchase orders the customer has completed. (This only requires a two-table join. This exercise sets the stage for the next two exercises.)

- 18S. Only consider customers located in Region 3 (RNO=3) that have completed a purchase order. For each such customer, display the customer's name, and the number of purchase orders the customer has completed.
- 18T. Reconsider the preceding exercise. You realize that customer names (CNAME values) are not necessarily unique. Revise the query objective to state: Only consider customers located in Region 3 (RNO=3) that have completed a purchase order. For each such customer, display the customer's number and name, followed by the number of purchase orders the customer has completed.
- 18U. How many parts were sold in states that are located in the Northeast or Southeast regions? Display the region name, followed by the state name, followed by the total quantity of parts sold in the state. Sort the result in ascending sequence by state name within region name.
- 18V. Display the region name and state name, followed by the total quantity of parts sold in the state if that quantity exceeds 100. Sort the result in ascending sequence by state name within region name.
- 18W. Extend Sample Query 18.16 to display state names instead of state codes: What customers have purchased parts that were sold to your company by a supplier who resides in the same state as the customer? For each such pair of co-located customers and suppliers, display the state name, followed by the customer's number and name, followed by the supplier's number and name. Sort the result by SNO within CNO within STCODE.

## Overview of Chapter 18 Appendices

### SQL Efficiency

Appendix 18A: Efficiency - More about Optimization

Appendix 18B: Theory

Appendix 18C: Theory & Efficiency

### Accessing Data Models

Appendix 18D: Finding Your Data Model

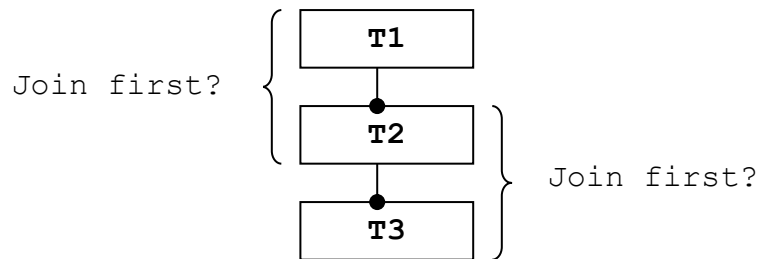
Appendix 18E: Reverse Engineering

Appendix 18F: Hierarchically Oriented Data Models

## Appendix 18A: Efficiency – More about Optimization

**Join-Sequence:** Assume the system must join three tables (T1, T2, and T3) that lie on a hierarchical path. (I.e., T2 has a foreign-key that reference T1, and T3 has a foreign-key that references T2.)

Regardless of the join-syntax, *the optimizer must decide which two tables to join first.* The basic efficiency strategy is to initially generate the smaller intermediate join-result by selecting the most efficient join-sequence. Here, the join-sequence question is: Should the optimizer tell the system to initially join tables T1 and T2, and then join table T3; or, should the optimizer tell the system to initially join tables T2 and T3, and then join table T1.



Below we consider two possible scenarios that do not involve any restriction operations. We also assume there is no significant difference in the length of the rows in the three tables.

**Scenario-1:** Assume that, for some unknown reason, the data dictionary does *not* contain statistics about the size of the tables. In this circumstance, T1 (the parent-table) will usually have fewer rows than T2 (the child table) unless there are many T1 parent rows without children. Likewise, T2 will usually have fewer rows than T3 unless there are many T2 parent rows without children. Hence, the optimizer favors initially joining the smaller tables, T1 and T2.

**Scenario-2:** Assume the optimizer *knows* the row count for each table. In this circumstance, because of the PK-FK relationships, the optimizer knows the *exact size* of both join-results.

Row Count for (T1 join T2) = Row count of T2  
Row Count for (T2 join T3) = Row count of T3

The optimizer will start with the join-operation associated with the smaller row count.

**Join Methods:** The optimizer must choose a join-method for each join-operation. (Appendix 17A discussed the Nested-Loop and the Sort-Merge methods.) For example, given a three-table join, the optimizer might decide to use the Nested-Loop Method for the first join-operation and the Sort-Merge Method for the second join-operation.

Most systems support at least four join methods. Consider a six-table join involving five join-operations. Assuming the choice of 4 methods for each of the five join-operations, the number of possible options is 20 ( $5 \times 4$ ).

**More Complexity:** The above query analysis is overly simplified. When the optimizer considers join-sequence and join-methods, it also considers restrictions against any of the tables. (We ignored these considerations.) The optimizer will also consider possible indexes that could possibly help some restriction-operations and join-operations. (We ignored these considerations.)

## Appendix 18B: Theory

We take you back to junior high school to review some mathematical buzzwords that you may not have not heard in a long time. These buzzwords are "commutative," "associative," and "transitive." Before discussing these terms in the context of SQL, we review each term in the context of everyday arithmetic.

1. **Commutative** Law of Addition: Example - If you know that  $2+3$  equals some value, then you can conclude that  $3+2$  equals the same value.

$$2 + 3 = 3 + 2$$

Generally:  $a + b = b + a$

Multiplication is also commutative. But subtraction and division are not commutative.

2. **Associative** Law of Addition: Example - Assume you calculate  $(23+1)$  and add 99 to obtain a result. The associative law says you will get the same result if you add 23 to  $(1+99)$ .

$$(23 + 1) + 99 = 23 + (1 + 99)$$

Generally:  $(a + b) + c = a + (b + c)$

Multiplication is also associative. But subtraction and division are not associative.

3. **Transitive** Law of Equality: Example - If you know that Albert (A) weighs the same as Bob (B), and Bob weighs that same as Charlie (C), then you can conclude that Albert weighs the same as Charlie.

If  $A = B$  and  $B = C$  then  $A = C$

Greater-than ( $>$ ) and Less-than ( $<$ ) are also transitive. But not-equal ( $<>$ ) is not transitive.

The following page will apply these mathematical concepts to the inner-join operation.



**Inner-Join is Commutative:** In Appendix 17A, we noted that, when joining tables T1 and T2, the optimizer could designate either table as the driving table. More formally, this means that the inner-join obeys the Commutative Law.

$$(T1 \text{ JOIN } T2) = (T2 \text{ JOIN } T1)$$

The equals (=) sign means the join operations produce the same result set (ignoring any differences in row or column sequence.)

**Inner-Join is Associative:** In the previous Appendix 18A, we noted that, when joining tables T1, T2, and T3 that lie along a hierarchy, the optimizer could initially join T1 and T2. Alternatively, the optimizer could initially join T2 and T3. More formally, this means that the inner-join obeys the Associative Law.

$$(T1 \text{ JOIN } T2) \text{ JOIN } T3 = T1 \text{ JOIN } (T2 \text{ JOIN } T3)$$

**AND-Conditions of Equality obey the Transitive Law:** Assume your SELECT statement contained the following two join-conditions.

```
SELECT *
FROM T1, T2, T3
WHERE T1.A = T2.B
AND    T2.B = T3.C
```

The optimizer could apply the transitive law to deduce that:

$$T1.A = T3.C$$

The following Appendix 18C will discuss the advantage of this kind of logical deduction.

## Appendix 18C: Theory & Efficiency

This appendix shows how the optimizer can capitalize on the Transitive Law. Consider the join-conditions in the WHERE-clause for Sample Query 18.15.2.

```
WHERE  ST.STCODE = C.STCODE
AND    C.STCODE = S.STCODE
```

The optimizer applies the Transitive Law to conclude that:

```
ST.STCODE = S.STCODE
```

This allows the optimizer to replace the original WHERE-clause with one of the following equivalent WHERE-clauses.

```
WHERE  ST.STCODE = C.STCODE
AND    ST.STCODE = S.STCODE
```

```
WHERE  C.STCODE = S.STCODE
AND    ST.STCODE = S.STCODE
```

This could be beneficial if the new condition (ST.STCODE = S.STCODE) produces a smaller intermediate join-result.

**Author Comment:** Discussion of this example has motivated some students to ask: "Should I include a redundant a join-condition in my WHERE-clause, as illustrated below?"

```
WHERE  ST.STCODE = C.STCODE
AND    C.STCODE = S.STCODE
AND    ST.STCODE = S.STCODE
```

Many years ago, when optimizers were not so smart, I answered "yes-maybe". Today, with smarter optimizers, I answer "no" because logically redundant code can be confusing, and we can assume optimizers will apply the Transitive Law to make the same deduction shown above.

**Example:** Given the following statement.

```
SELECT ENAME, DNAME
FROM   EMPLOYEE E, DEPARTMENT D
WHERE  D.DNO = E.DNO
AND    E.DNO = 20
AND    E.SALARY > 2500
```

The algebraic operations for this statement are:

```
TEMP1 ← RESTRICT EMPLOYEE WHERE DNO = 20 AND SALARY > 2500
      [ENAME, DNO]

TEMP2 ← DEPARTMENT JOIN TEMP1

RESULT ← TEMP2 [ENAME, DNAME]
```

Notice that the JOIN operation compares on all rows in the DEPARTMENT table. A more efficient plan is described below.

- - - - -

The optimizer applies the Transitive Law to deduce that D.DNO = 20, and enhances the WHERE-clause such that it looks like:

```
SELECT ENAME, DNAME
FROM   EMPLOYEE E, DEPARTMENT D
WHERE  E.DNO = E.DNO
AND    E.DNO = 20
AND    E.SALARY > 2500.00
AND    D.DNO = 20      ←
```

The algebraic operations for this statement are:

```
TEMP1 ← RESTRICT EMPLOYEE WHERE DNO = 20 AND SALARY > 2500
      [ENAME]

TEMP2 ← RESTRICT DEPARTMENT WHERE DNO = 20 [DNAME]

RESULT ← TEMP1 CROSS TEMP2
```

Notice that TEMP2 has just one row because DNO is the primary key of DEPARTMENT. (Also, the search of DEPARTMENT is fast because it uses the primary-key index.) Hence the subsequent CROSS operation involves just one TEMP2 row, and there is no need to compare on DNO values.

## Appendix 18D: Finding Your Data Model

Assume it is your first day on the job, or your first day as a consultant at some job site. Before you code your first SQL statement, you should find the relevant data model. How do you do this?

Ask someone (a colleague, supervisor, or DBA). You could be told to:

- Examine relevant "paper" documentation. Or,
- Access the database design tool that was used to design and create the database. Then print a screenshot of the data model. Or,
- Use your front-end query tool to "reverse engineer" tabular metadata into a graphical data model.

However, sometimes none of the above methods are available. The paper data model never existed or has become obsolete, or the database was not designed using a database design tool, or your query tool does not support reverse engineering. In this case you should try some do-it-yourself reverse engineering as described in the following Appendix 18E.

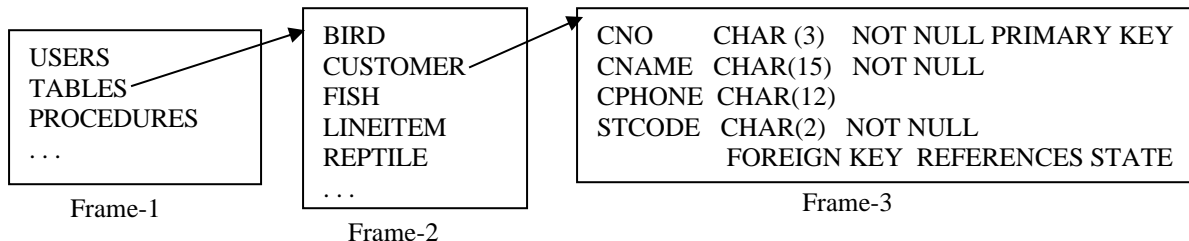
## Appendix 18E: Reverse Engineering

**Objective:** Follow some reverse engineering process to build a valid data model that represents a given set of database tables. Reverse engineering usually produces a spaghetti model that looks like Figure 18.2. The following Appendix 18F shows how to transform this spaghetti model into a hierarchical model.

**Preliminary Step:** Access the metadata that describes your database tables. Today, most front-end query tools provide some kind of "Metadata Panel" as described back in Chapter 1. Alternatively, if available, you could use the metadata embodied within the CREATE TABLE statements that created the database tables. We outline both methods.

### A. Reverse Engineering using Query Tool Meta-Data

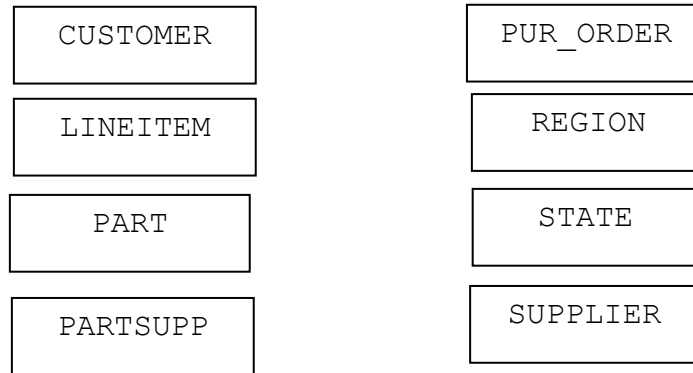
The Metadata Panel allows you to discover the names of all tables in the database, the column-names and data-types of all columns in these tables, and other relevant information, especially information about primary-keys and foreign-keys. Some metadata panels, like the following Frame-1, will list all types of database objects (e.g., tables, users, indexes, stored procedures, etc.).



Because we are interested in tables, you would click of the TABLES item. This would trigger the display of Frame-2 which shows a list of all table-names (or maybe just the names of tables that you can access). This list of tables includes the CUSTOMER table and other relevant tables in MTPC database. This list could also include the names of some irrelevant tables (e.g., BIRD, FISH, REPTILE) that were created in this database; or it could include some other important tables (e.g., DEPARTMENT, EMPLOYEE, PROJECT) that are not immediately relevant for your query objectives. This list of table-names is probably displayed in alphabetical sequence.

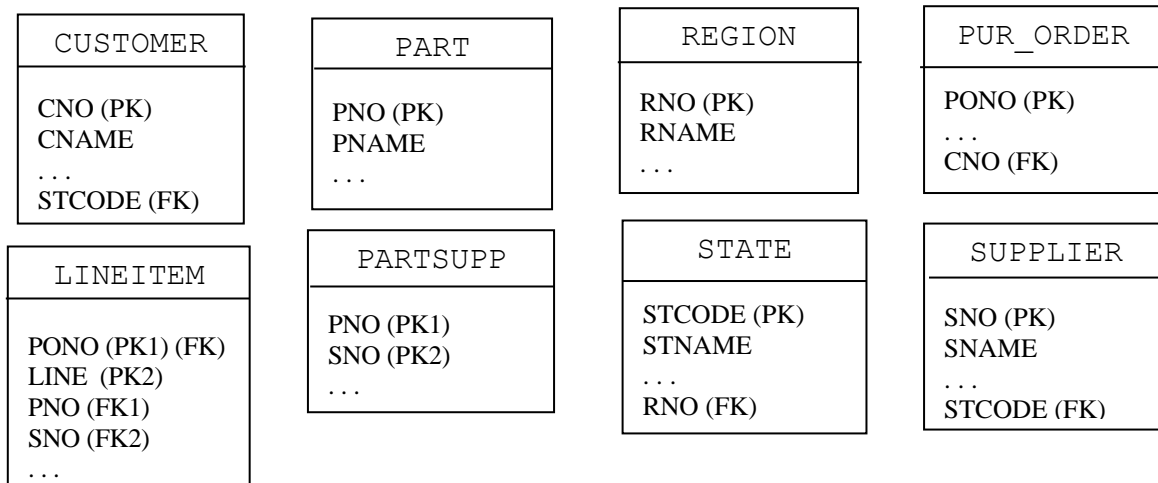
Using the above information, we start to build our data model. (Note: Reverse engineering is the inverse of the “forward engineering” design steps presented in Appendix 13B.)

Step-1: Draw rectangles. Select the desired table-names from Frame-2. Each table-name implies the designation of a corresponding rectangle.

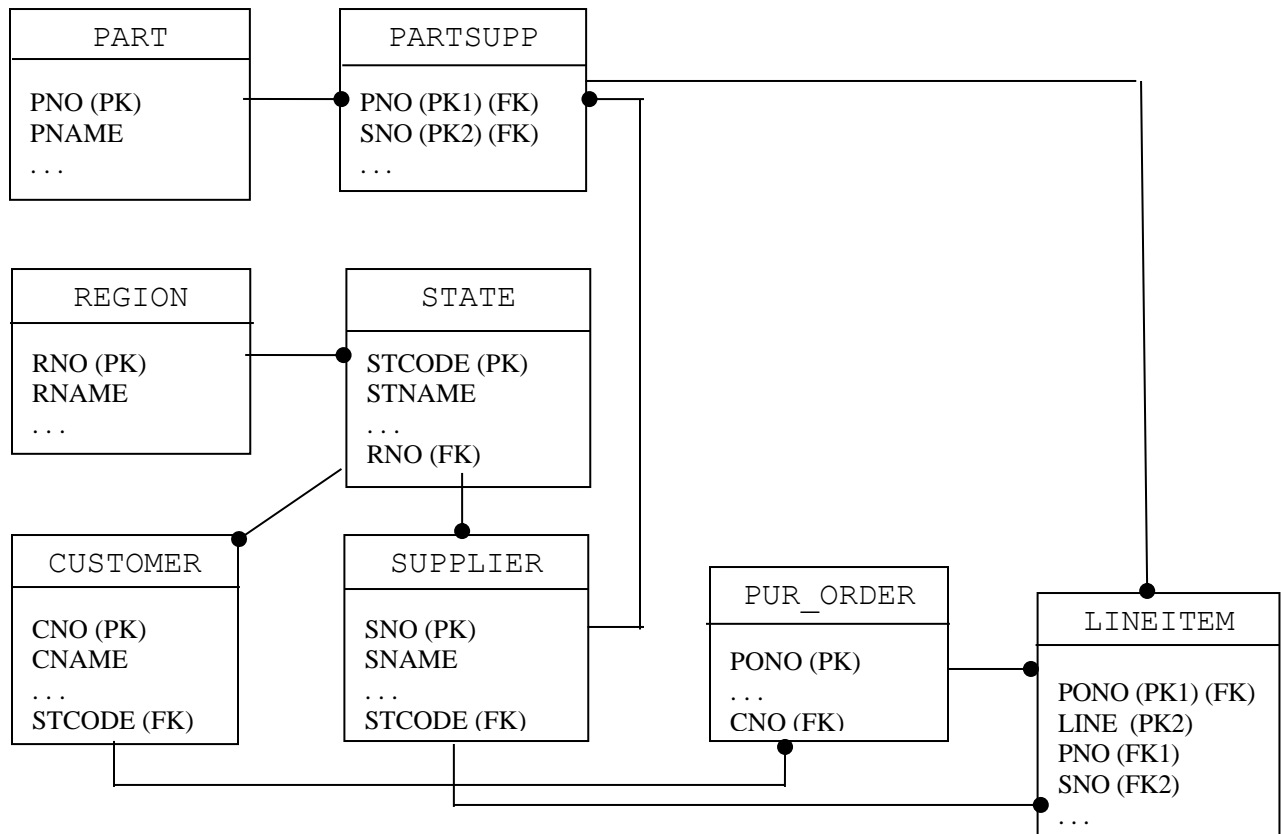


Next, in Frame-2, you click on the name of each desired table. For example, if you click on CUSTOMER, Frame-3 appears with a list of CUSTOMER’s column-names, corresponding data-types, NOT NULL indicators, and *maybe* PRIMARY KEY and FOREIGN KEY designations. (We said “*maybe*” because PRIMARY KEY and FOREIGN KEY information could be stored elsewhere. More on that below.) For the moment, we will assume that Frame-3 identifies PRIMARY KEY and FOREIGN KEY columns.

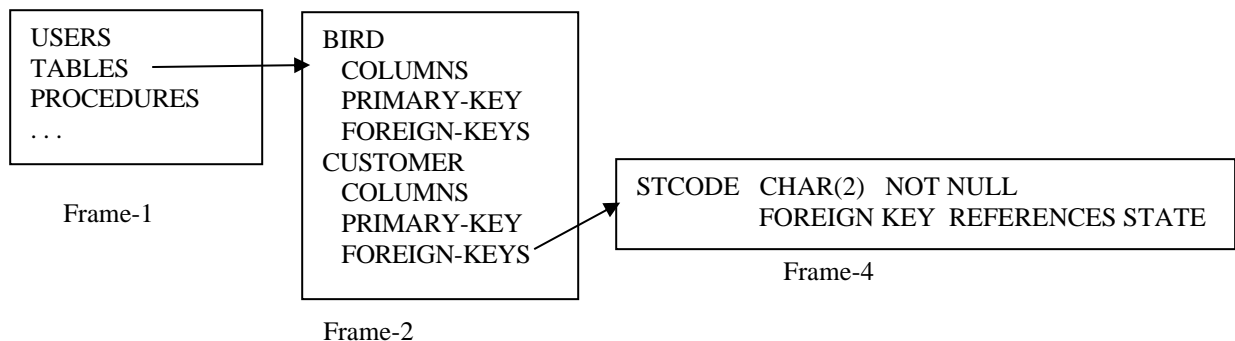
Step-2: For each table, designate its primary-key and a few popular columns. Also, include foreign-key columns without designating the tables they reference.



Step-3: Draw lines for one-to-many relationships. Relationship lines are derived from foreign-key information located in somewhere in the Metadata Panel. You will probably have to move some rectangles around to avoid intersecting lines. There is no immediate need to build a hierarchical model. (Hierarchical models are generated in the following Appendix 18F.)



Aside: Your Metadata Panel may be organized such that foreign-key information is located in a different frame as illustrated below.



## **B. Reverse Engineering using CREATE TABLE statements**

Assume you can access the CREATE TABLE statements used to create your tables. (The CREATE-ALL-TABLE file includes the CREATE TABLE statements for the MTPCH database.) The following steps will reverse engineer a collection of CREATE TABLE statements into a data model.

- For each CREATE TABLE statement, draw a rectangle to represent the table.
- Columns in this statement become attributes in the corresponding rectangle.
- The PRIMARY KEY clause designates the primary-key column.
- Draw a one-to-many relationship line to represent each FOREIGN KEY clause in the statement.

**Careful!** This process assumes that *accurate* CREATE TABLE statements are available. Unfortunately, your CREATE TABLE statements may have become obsolete because of changes made by creating or dropping tables, or adding new columns via ALTER TABLE statements.

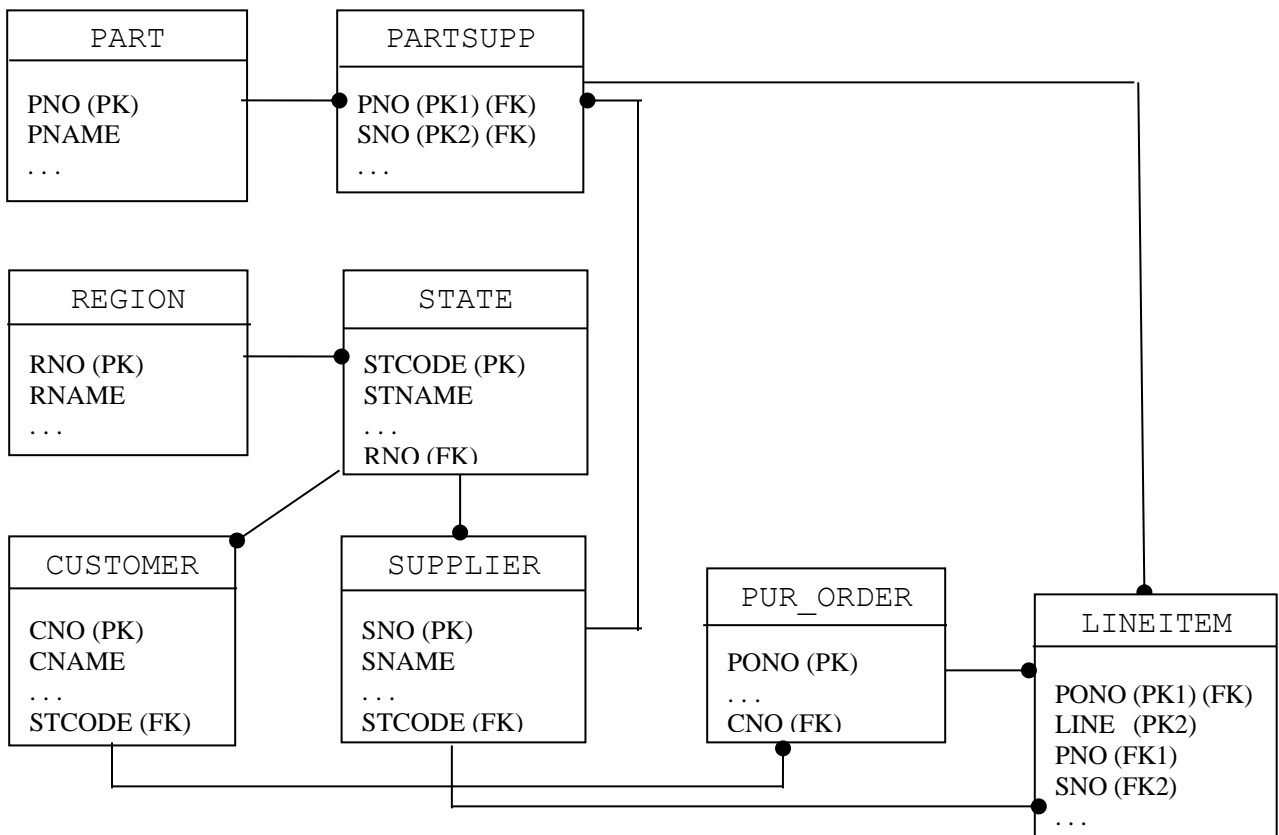
Finally, as previously stated, reverse engineering usually generates a spaghetti data model. The following Appendix 18F describes how to transform a spaghetti model into a hierarchically oriented model.



## Appendix 18F: Drawing Hierarchical Models

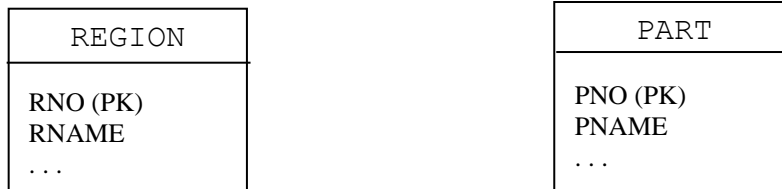
**Objective:** Transform a spaghetti data model (e.g., Figure 18.2) into a hierarchical model (e.g., Figure 18.3 or Figure 18.4).

We recommend redrawing a spaghetti data model into a hierarchically orientated model. Sometimes, you can use your intuition to achieve this objective. Alternatively, you could apply some “semi-mechanical” method as described below. Here, we start with the following spaghetti model developed in the preceding Appendix 18E.



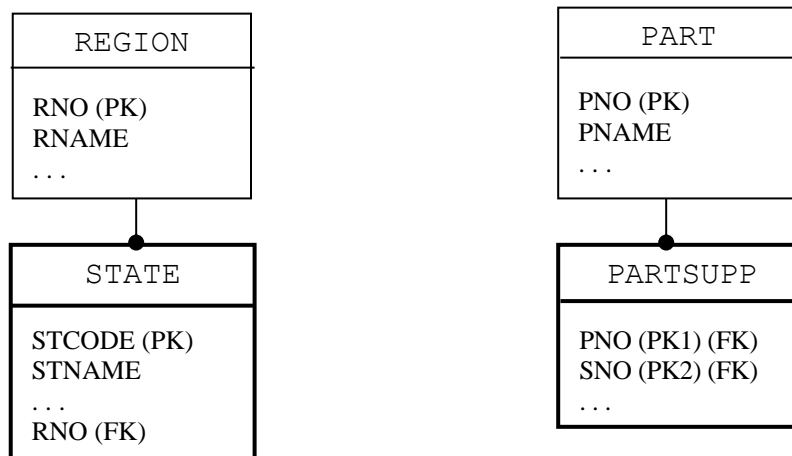
Find a big whiteboard or a large piece of paper and a pencil with a good eraser.

**Step-1:** Examine the spaghetti model to identify the top-level (root) tables and position them at the top of the page. Top-level tables do not have any foreign-key (FK) columns. This applies to the REGION and PART tables.



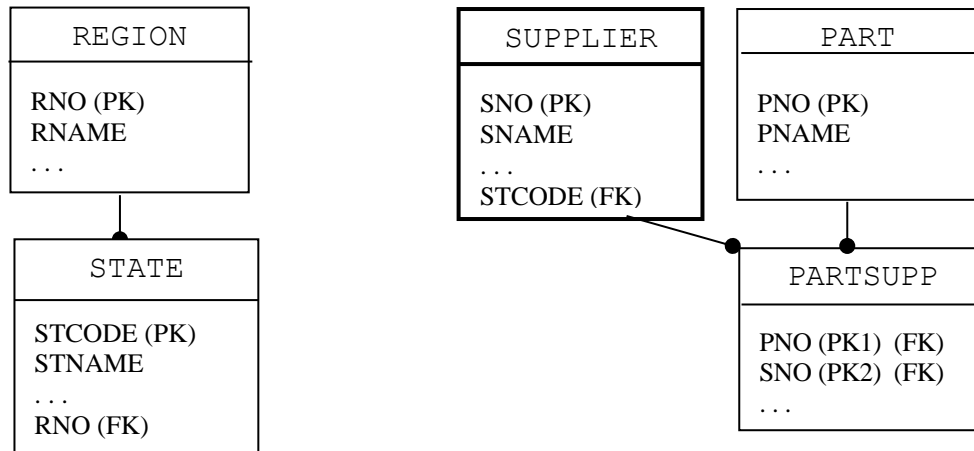
We will refer to the above intermediate design as the "current hierarchical model."

**Step-2:** Find the child-tables for these top-level tables. These child-tables have foreign-keys that reference the above REGION and PART tables. This applies to the STATE table with a foreign-key that references REGION; and it applies to the PARTSUPP table with a foreign-key that references PART.

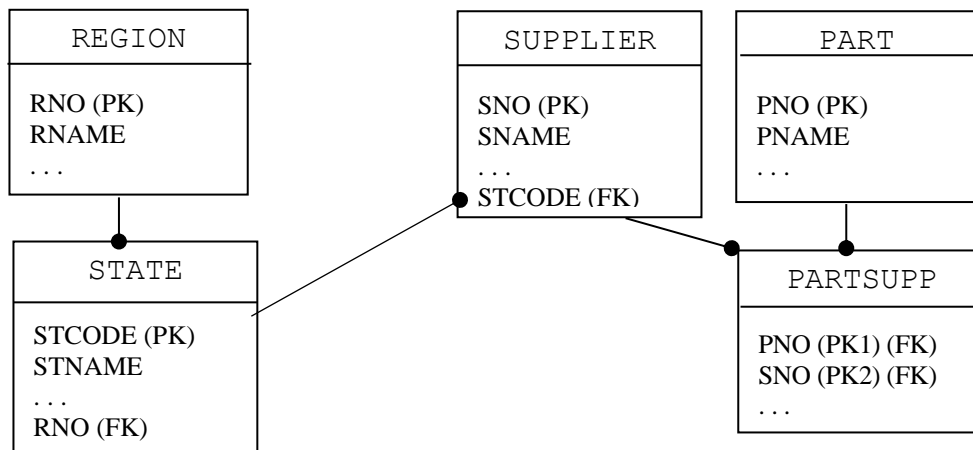


The current hierarchical model now has four tables.

**Step-3:** Ask: Do the recently added STATE and PARTSUPP tables have other foreign-keys that reference tables not in the current hierarchical model? The STATE table does not have any other foreign-keys. But, the PARTSUPP table does have another foreign-key (SNO) that references the SUPPLIER table. Hence, we must incorporate SUPPLIER into the current hierarchical model.

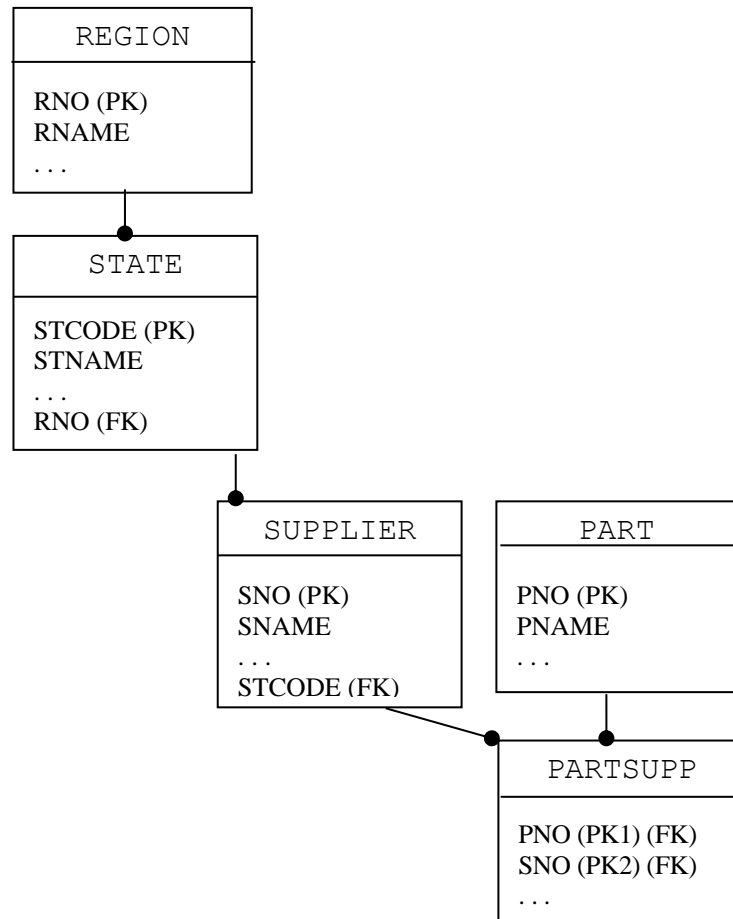


Again, does the recently added SUPPLIER table have any foreign-keys? Yes. The SUPPLIER table has a foreign-key (STCODE) that references STATE. Hence, we draw a one-to-many relationship line from STATE to SUPPLIER. (If SUPPLIER had foreign-key that referenced a table not in the current hierarchical model, we would introduce that table into the current hierarchical model.)



After drawing this new relationship line, we observe that our model no longer appears to be hierarchically oriented.

Reorganize Tables: "Move boxes and lines around" such that the design is hierarchically oriented.



The current hierarchical model now has five tables. Note that all foreign-keys reference tables that are within this model.

Note: Sometimes, the current model cannot be represented within a strictly hierarchical structure. This can happen in the special case where the design includes one or more recursive relationships (to be described in Chapter 30).

#### **Step-4: Iterate on Step-2 and Step-3**

Include new tables that are the children of the tables introduced during the preceding iteration. Draw relationship lines for these tables. [If necessary, "move boxes and lines around" such that the design becomes a hierarchy.]

If these new tables have foreign-keys that reference tables outside the current hierarchical model, incorporate these tables into the design and draw relationship lines for these new tables. [Again, if necessary, "move boxes and lines around" such that the design becomes a hierarchy.]

Etc.

We have already included child-tables for of REGION and PART.

Examining the original spaghetti model, we observe that:

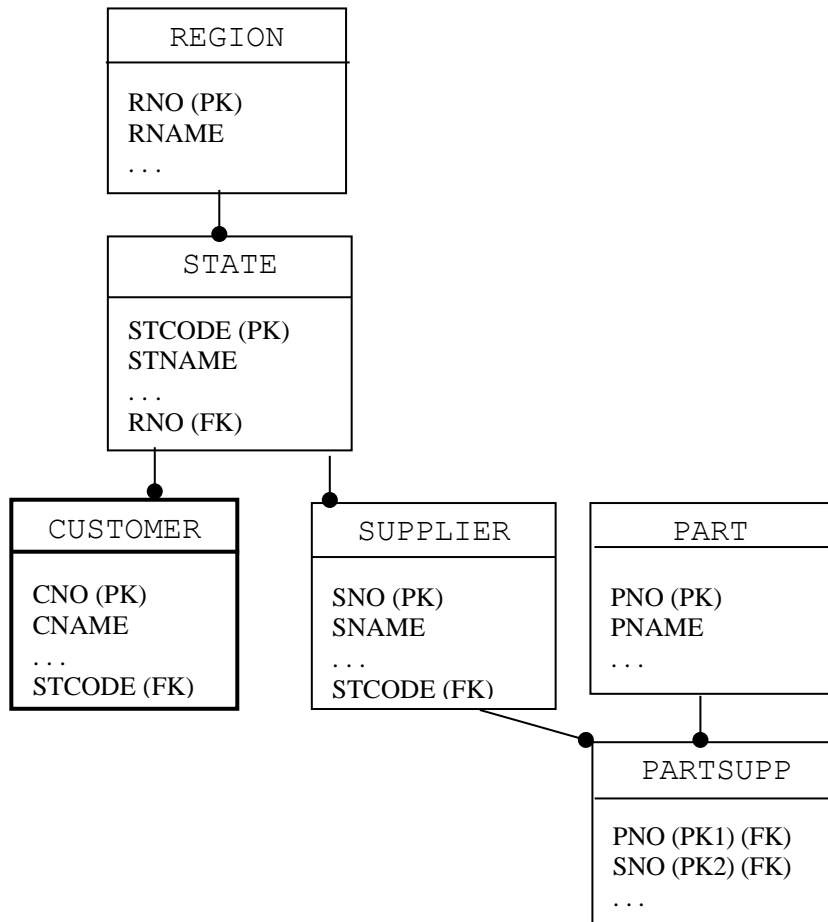
STATE has another child, CUSTOMER.

PARTSUPP has another child, LINEITEM.

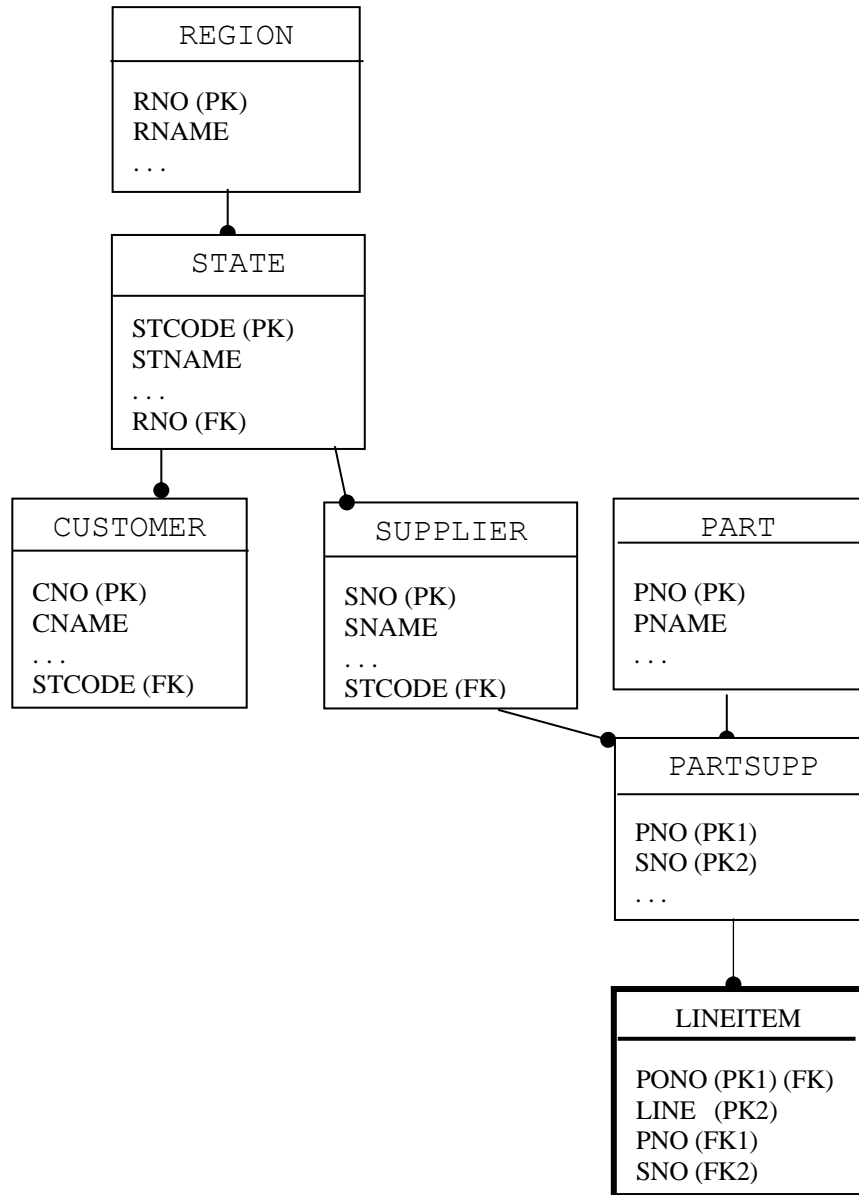
SUPPLIER does not have any other children.

Therefore, we must include CUSTOMER and LINEITEM with the current model.

Include CUSTOMER as a child of STATE, which we can do without disrupting the hierarchy. Also, observe that CUSTOMER does not have any other foreign-keys other than STCODE which references STATE.

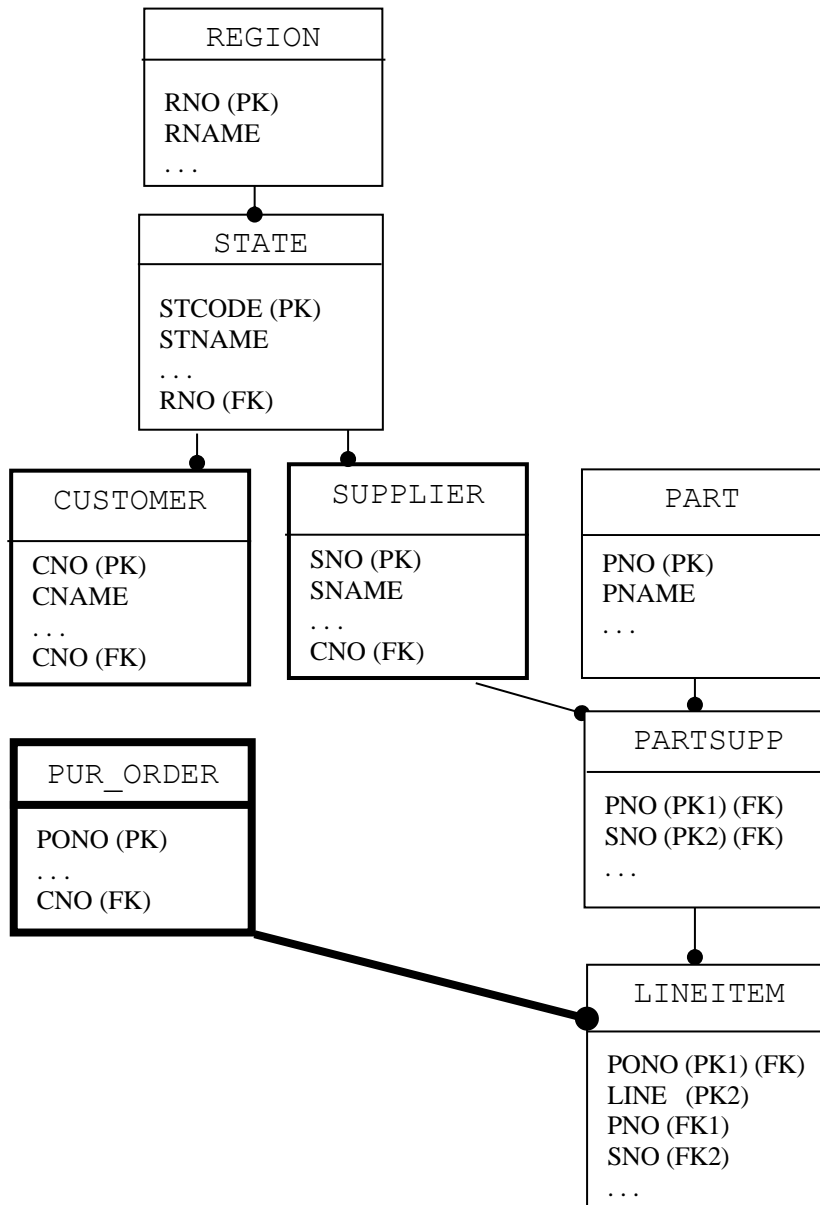


Include LINEITEM as a child of PARTSUPP, which we can do without disrupting the hierarchy.



After including LINEITEM into the current hierarchical model, observe that LINEITEM has a foreign-key that references PUR\_ORDER, a table that is not in the current model.

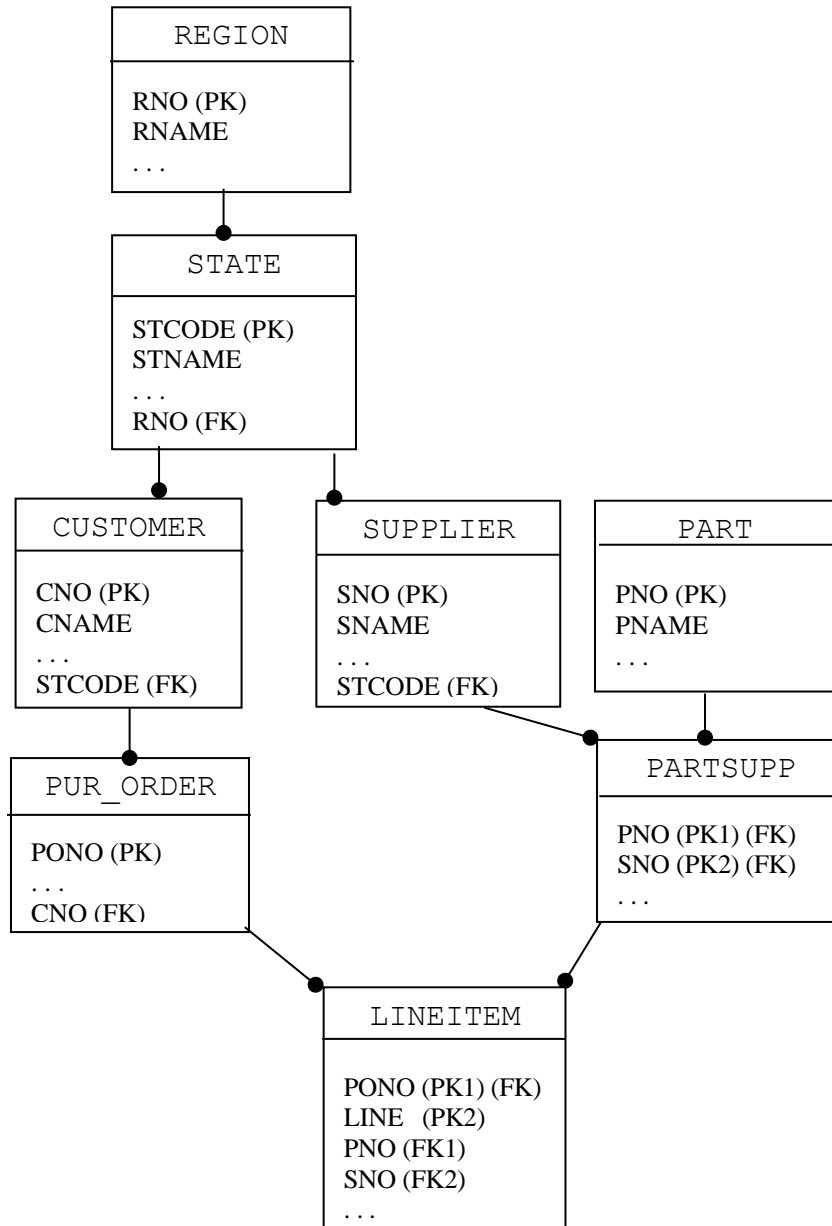
Therefore, include PUR\_ORDER into current hierarchical model and draw a one-to-many relationship line from PUR\_ORDER to LINEITEM.





After including PUR\_ORDER into the current hierarchical model, observe that it has a foreign-key that references CUSTOMER.

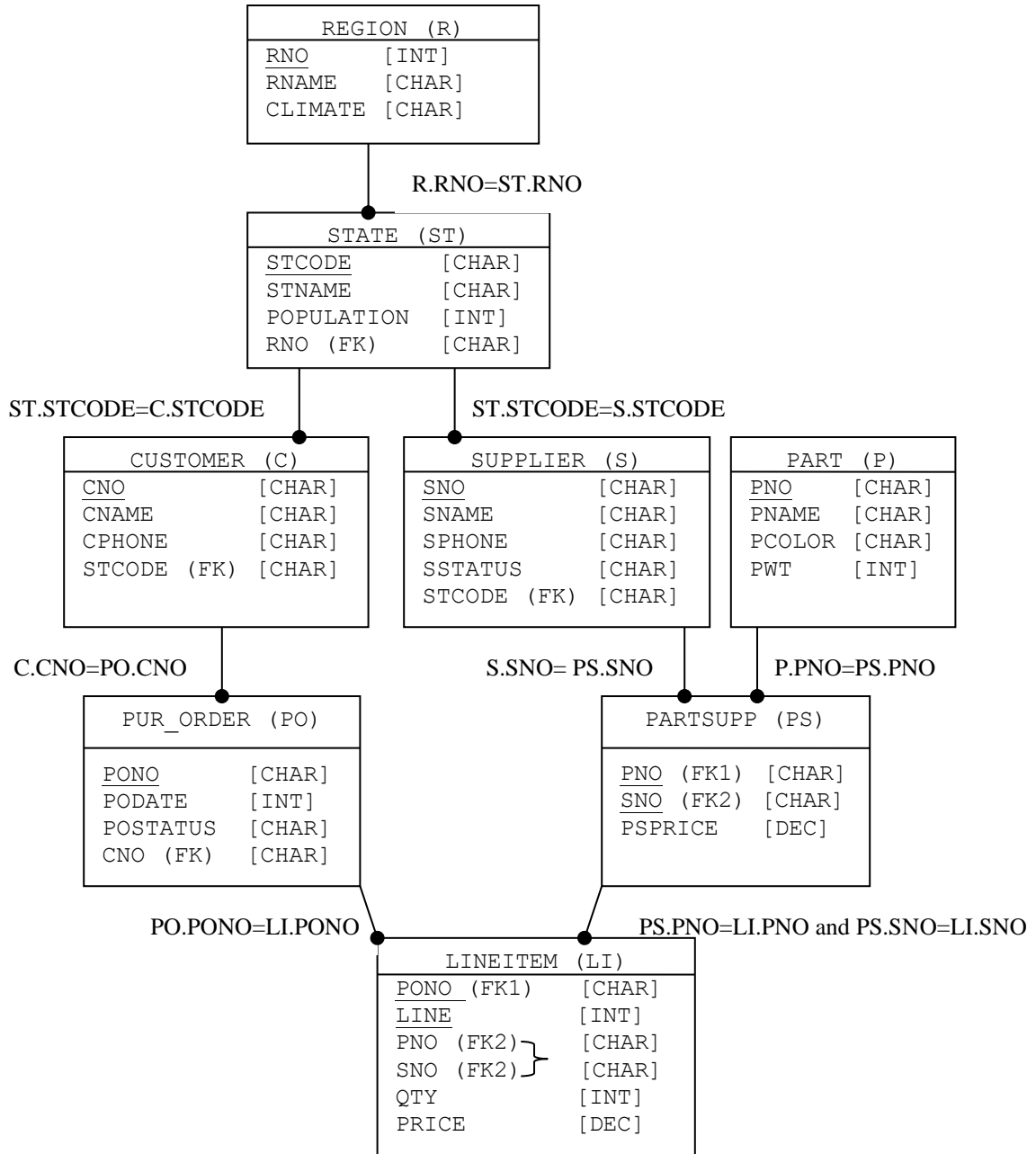
Therefore, draw a one-to-many relationship line from CUSTOMER to PUR\_ORDER.



Almost Done: The current hierarchical model now has eight tables, the same number of tables in our original spaghetti model. Hence, we can stop iterating.

Final Step:

Include the Primary-Key/Foreign-Key join-conditions as previously illustrated in Figure 18.4 (shown below). Also, if reasonable, include all columns and their data-types. Otherwise, describe columns, data-types, and null indicators in associated documentation.



## Concluding Comment

Chapter 30 will introduce recursive relationships that cannot be drawn with a strictly hierarchical orientation. This situation rarely presents a problem because many real-world applications are designed to avoid recursive relationships.

# 19

## Outer-Join: Getting Started

There are three types of outer-join operations.

1. Left Outer-Join
2. Right Outer-Join
3. Full Outer-Join

Each type of outer-join uses the JOIN-ON syntax to merge data from two tables by matching rows on a join-condition. As with the inner-join, each type of outer-join result table contains data from all matching rows. However, unlike the inner-join, *an outer-join result will also contain additional rows derived from non-matching rows*. The content of these additional rows depends upon the type of outer-join.

Outer-join logic requires that you designate one table as the "LEFT-table." The other table becomes the "RIGHT-table." Casually speaking, we note that a:

- **LEFT Outer-Join** result table contains all matching rows plus all non-matching rows from the LEFT-table only.
- **RIGHT Outer-Join** result table contains all matching rows plus all non-matching rows from the RIGHT-table only.
- **FULL Outer-Join** result table contains all matching rows, plus all non-matching rows from both tables.

The following Figure 19.1 presents an example of each type of outer-join operation.

## LEFT, RIGHT, and FULL Outer-Joins

Figure 19.1 illustrates each type of outer-join using the following MAN table (designated as the LEFT-table) and DOG table (the RIGHT-table). Observe that there is no PK-FK relationship between these tables. Each example uses the JOIN-ON syntax to compare on the MNO columns.

<u>MAN</u>		<u>DOG</u>		
MNO	MNAME	DNO	DNAME	MNO
77	MOE	1000	SPOT	99
<b>88</b>	LARRY (non-matching row)	3000	ROVER	77
99	CURLY	2000	WALLY	99
		4000	SPIKE	<b>10</b> (non-matching row)

### LEFT Outer-Join

```
SELECT *
FROM MAN LEFT OUTER JOIN DOG
ON MAN.MNO = DOG.MNO
```

MNO	MNAME	DNO	DNAME	MNO1
77	MOE	3000	ROVER	77
<b>88</b>	<b>LARRY</b>	-	-	-
99	CURLY	1000	SPOT	99
99	CURLY	2000	WALLY	99

Non-matching MAN row (MNO = 88) appears in result.

### RIGHT Outer-Join

```
SELECT *
FROM MAN RIGHT OUTER JOIN DOG
ON MAN.MNO = DOG.MNO
```

MNO	MNAME	DNO	DNAME	MNO1
77	MOE	3000	ROVER	77
99	CURLY	1000	SPOT	99
99	CURLY	2000	WALLY	99
-	-	<b>4000</b>	<b>SPIKE</b>	<b>10</b>

Non-matching DOG row (DNO = 4000) appears in result.

### FULL Outer-Join

```
SELECT *
FROM MAN FULL OUTER JOIN DOG
ON MAN.MNO = DOG.MNO
```

MNO	MNAME	DNO	DNAME	MNO1
77	MOE	3000	ROVER	77
<b>88</b>	<b>LARRY</b>	-	-	-
99	CURLY	1000	SPOT	99
99	CURLY	2000	WALLY	99
-	-	<b>4000</b>	<b>SPIKE</b>	<b>10</b>

Non-matching MAN row (MNO = 88) appears in result.  
Non-matching DOG row (DNO = 4000) appears in result.

**Figure 19.1: LEFT, RIGHT and FULL Outer-Join Operations**

[\* The front-end tool displays MNO1 as the column header for the MNO column in the DOG table.]

## Important Special Case: Primary Key – Foreign Key Relationship

On the previous page, the MAN and DOG tables were *not* related via a PK-FK relationship. In the following versions of the MAN and DOG tables, the DOG.MNO column is designated as a non-null foreign-key that references the MAN.MNO column. Again, MAN is designated as the left-table, and DOG becomes the right-table.

	<b>PK</b>			<b>FK</b>		
<u>MAN</u>	<u>MNO</u>	<u>MNAME</u>	<u>DOG</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO</u>
	77	MOE		1000	SPOT	99
	<b>88</b>	LARRY		3000	ROVER	77
	99	CURLY		2000	WALLY	99
				4000	SPIKE	99

Important Observation: All of the following outer-join result tables **contain data from all DOG rows** because the FK-constraint requires that all DOG.MNO values match on the join-condition.

<pre>SELECT * FROM MAN <b>LEFT OUTER JOIN</b> DOG <b>ON</b>   MAN.MNO = DOG.MNO</pre>	<table border="1"> <thead> <tr> <th>MNO</th> <th>MNAME</th> <th>DNO</th> <th>DNAME</th> <th>MNO1</th> </tr> </thead> <tbody> <tr> <td>77</td> <td>MOE</td> <td>3000</td> <td>ROVER</td> <td>77</td> </tr> <tr> <td><b>88</b></td> <td><b>LARRY</b></td> <td>-</td> <td>-</td> <td>-</td> </tr> <tr> <td>99</td> <td>CURLY</td> <td>1000</td> <td>SPOT</td> <td>99</td> </tr> <tr> <td>99</td> <td>CURLY</td> <td>2000</td> <td>WALLY</td> <td>99</td> </tr> <tr> <td>99</td> <td>CURLY</td> <td>4000</td> <td>SPIKE</td> <td>99</td> </tr> </tbody> </table>	MNO	MNAME	DNO	DNAME	MNO1	77	MOE	3000	ROVER	77	<b>88</b>	<b>LARRY</b>	-	-	-	99	CURLY	1000	SPOT	99	99	CURLY	2000	WALLY	99	99	CURLY	4000	SPIKE	99
MNO	MNAME	DNO	DNAME	MNO1																											
77	MOE	3000	ROVER	77																											
<b>88</b>	<b>LARRY</b>	-	-	-																											
99	CURLY	1000	SPOT	99																											
99	CURLY	2000	WALLY	99																											
99	CURLY	4000	SPIKE	99																											
<pre>SELECT * FROM MAN <b>RIGHT OUTER JOIN</b> DOG <b>ON</b>   MAN.MNO = DOG.MNO</pre>	<table border="1"> <thead> <tr> <th>MNO</th> <th>MNAME</th> <th>DNO</th> <th>DNAME</th> <th>MNO1</th> </tr> </thead> <tbody> <tr> <td>77</td> <td>MOE</td> <td>3000</td> <td>ROVER</td> <td>77</td> </tr> <tr> <td>99</td> <td>CURLY</td> <td>1000</td> <td>SPOT</td> <td>99</td> </tr> <tr> <td>99</td> <td>CURLY</td> <td>2000</td> <td>WALLY</td> <td>99</td> </tr> <tr> <td>99</td> <td>CURLY</td> <td>4000</td> <td>SPIKE</td> <td>99</td> </tr> </tbody> </table>	MNO	MNAME	DNO	DNAME	MNO1	77	MOE	3000	ROVER	77	99	CURLY	1000	SPOT	99	99	CURLY	2000	WALLY	99	99	CURLY	4000	SPIKE	99					
MNO	MNAME	DNO	DNAME	MNO1																											
77	MOE	3000	ROVER	77																											
99	CURLY	1000	SPOT	99																											
99	CURLY	2000	WALLY	99																											
99	CURLY	4000	SPIKE	99																											
<pre>SELECT * FROM MAN <b>FULL OUTER JOIN</b> DOG <b>ON</b>   MAN.MNO = DOG.MNO</pre>	<table border="1"> <thead> <tr> <th>MNO</th> <th>MNAME</th> <th>DNO</th> <th>DNAME</th> <th>MNO1</th> </tr> </thead> <tbody> <tr> <td>77</td> <td>MOE</td> <td>3000</td> <td>ROVER</td> <td>77</td> </tr> <tr> <td><b>88</b></td> <td><b>LARRY</b></td> <td>-</td> <td>-</td> <td>-</td> </tr> <tr> <td>99</td> <td>CURLY</td> <td>1000</td> <td>SPOT</td> <td>99</td> </tr> <tr> <td>99</td> <td>CURLY</td> <td>2000</td> <td>WALLY</td> <td>99</td> </tr> <tr> <td>99</td> <td>CURLY</td> <td>4000</td> <td>SPIKE</td> <td>99</td> </tr> </tbody> </table>	MNO	MNAME	DNO	DNAME	MNO1	77	MOE	3000	ROVER	77	<b>88</b>	<b>LARRY</b>	-	-	-	99	CURLY	1000	SPOT	99	99	CURLY	2000	WALLY	99	99	CURLY	4000	SPIKE	99
MNO	MNAME	DNO	DNAME	MNO1																											
77	MOE	3000	ROVER	77																											
<b>88</b>	<b>LARRY</b>	-	-	-																											
99	CURLY	1000	SPOT	99																											
99	CURLY	2000	WALLY	99																											
99	CURLY	4000	SPIKE	99																											

**Figure 19.2: Outer-Joins based on PK-FK Relationship**

## FULL OUTER JOIN (No PK-FK Relationship)

This chapter's sample queries reference the DEPARTMENT, EMPLOYEE, and EMPLOYEE3 tables illustrated in Figures 16.1a and 16.3a.

The first sample query illustrates a FULL outer-join of two tables where the join-condition is *not* based upon a PK-FK relationship. In such circumstances, it is possible for both the left-table and the right-table to contain rows that do not match on the join-condition.

**Sample Query 19.1:** Reference the DEPARTMENT and EMPLOYEE3 tables. Display all information about *all* departments and *all* employees. (I.e., perform a full outer-join of DEPARTMENT and EMPLOYEE3.) Sort the result by ENO within DNO.

```
SELECT *  
  
FROM DEPARTMENT D FULL OUTER JOIN EMPLOYEE3 E  
  
ON D.DNO = E.DNO  
  
ORDER BY D.DNO, E.ENO
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	2000	LARRY	2000.00	10
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40
-	-	-	1000	MOE	2000.00	99
-	-	-	6000	GEORGE	9000.00	-

**Syntax:** DEPARTMENT is the left-table because it is specified to the left of FULL OUTER JOIN.

**Logic:** Because this is a FULL outer-join, data from all matching and all non-matching rows appear in the result.

**Important Observation:** The result table shows that system appends null values to the non-matching rows.

[\* The front-end tool displays DNO1 as the column header for the DNO column in the EMPLOYEE table.]

## LEFT OUTER JOIN (No PK-FK Relationship)

The following sample query illustrates a LEFT outer-join where the join-condition is *not* based upon a PK-FK relationship. Again, we note that it is possible for both tables to contain rows that do not match the join-condition. This example will display all matching rows plus the non-matching rows from the left-table only.

**Sample Query 19.2:** Reference the DEPARTMENT and EMPLOYEE3 tables. Display all information about *all* departments along with all information about employees who work in those departments. (I.e., Perform a left outer-join of DEPARTMENT and EMPLOYEE3 where DEPARTMENT is designated as the left-table.) Sort the result by ENO within DNO.

```
SELECT *  
  
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE3 E  
  
ON D.DNO = E.DNO  
  
ORDER BY D.DNO, E. ENO
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	2000	LARRY	2000.00	10
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40

**Syntax:** DEPARTMENT is the left-table because it is specified to the left of LEFT OUTER JOIN.

**Logic:** This query objective requires that you designate DEPARTMENT as the left-table. The system appends null values to the non-matching row for Department 30.

**Equivalent Statement:** Exercise 19D will ask you code an equivalent statement using a RIGHT OUTER JOIN.



## RIGHT OUTER JOIN (No PK-FK Relationship)

The following sample query illustrates a RIGHT outer-join where the join condition is *not* based upon a PK-FK relationship. Again, both tables could contain rows that do not match on the join-condition. The result displays data from all matching rows, plus the non-matching rows from the right-table.

**Sample Query 19.3:** Reference the DEPARTMENT and EMPLOYEE3 tables. Display all information about *all* employees along with all information about the departments the employees work in. (I.e., Perform a right outer-join of DEPARTMENT and EMPLOYEE3 where EMPLOYEE3 is the right-table.) Sort the result by ENO within DNO.

```
SELECT *
FROM DEPARTMENT D RIGHT OUTER JOIN EMPLOYEE3 E
ON D.DNO = E.DNO
ORDER BY D.DNO, E. ENO
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	2000	LARRY	2000.00	10
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40
-	-	-	1000	MOE	2000.00	99
-	-	-	6000	GEORGE	9000.00	-

**Syntax:** EMPLOYEE3 is the right-table because it is specified to the right of RIGHT OUTER JOIN.

**Logic:** This query objective requires that you designate EMPLOYEE3 as the right-table. The system appends null values to the non-matching EMPLOYEE3 rows.

**Equivalent Statement:** The following sample query will present a better way to code this statement.

## RIGHT OUTER JOIN is Unnecessary

You *never need to code a RIGHT outer-join*. From a practical point of view, you can forget about the RIGHT outer-join because any RIGHT outer-join can always be expressed as a LEFT outer-join.

**Sample Query 19.4:** Same as preceding Sample Query 19.3. This time, perform a LEFT outer-join where EMPLOYEE3 is designated as the left-table.

```
SELECT D.DNO, D.DNAME, D.BUDGET,
       E.ENO, E.ENAME, E.SALARY, E.DNO
FROM EMPLOYEE3 E LEFT OUTER JOIN DEPARTMENT D
ON E.DNO = D.DNO
ORDER BY D.DNO, E.ENO
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	2000	LARRY	2000.00	10
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40
-	-	-	1000	MOE	2000.00	99
-	-	-	6000	GEORGE	9000.00	-

**Strong Recommendation:** Assume you intend to code an outer-join operation and you want to preserve the non-matching rows from just one table. In this circumstance, code a LEFT outer-join where the "preserve-all-rows-table" is designated as the *LEFT-table*.

## LEFT OUTER JOIN (Primary Key - Foreign Key)

The preceding Sample Queries 19.1-19.4 performed outer-join operations on tables that were not related via a PK-FK relationship. However, as with inner-joins, most outer-join operations are applied to tables that are related via a PK-FK relationship. The following sample queries reference the EMPLOYEE table with a non-null foreign key that references the DEPARTMENT table.

The following sample query illustrates the *most popular form of outer-join*. This is a LEFT OUTER JOIN where the join-condition is based upon a PK-FK relationship, and the FK is non-null. We designate the parent-table (DEPARTMENT) as the left-table, and the child-table (EMPLOYEE) becomes the right-table. In this circumstance, it is possible for the left-table to contain rows that do not match the join-condition. However, every row in the right-table must match.

**Sample Query 19.5:** Reference the DEPARTMENT and EMPLOYEE tables. Display all information about *all* departments and the employees who work in those departments. Sort the result by ENO within DNO.

```
SELECT *  
  
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E  
  
ON E.DNO = D.DNO  
  
ORDER BY D.DNO, E.ENO
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	2000	LARRY	2000.00	10
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40

**Important Observation:** Because the tables are related via a PK-FK relationship, you will get the same result if you execute a FULL outer-join. This is illustrated in the following Sample Query 19.6.

## FULL OUTER JOIN (Primary Key - Foreign Key)

The following sample query illustrates a FULL outer-join of two tables where the join-condition is based upon a PK-FK relationship, and the FK is non-null. We designate the parent-table (DEPARTMENT) as the left-table, and the child-table (EMPLOYEE) becomes the right-table. In this circumstance, it is possible for the left-table to contain rows that do not match the join condition. However, all rows in the right-table must match.

**Sample Query 19.6:** Reference the DEPARTMENT and EMPLOYEE tables. Display all information about *all* departments along with all information about employees who work in these departments. Also, include information about all employees. (Hypothetically, this includes any employee who is not assigned to some department. *But the PK-FK relationship means this is not possible.*) Sort the result by ENO within DNO.

```
SELECT *  
  
FROM DEPARTMENT D FULL OUTER JOIN EMPLOYEE E  
  
ON D.DNO = E.DNO  
  
ORDER BY D.DNO, E.ENO
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	2000	LARRY	2000.00	10
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40

**Important Observation:** This result table is identical to the previous result table.

**Conclusion:** If tables are related via a PK-FK relationship, and the parent-table is designated as the left-table, the LEFT outer-join and the FULL outer-join produce the same result. *In general, we recommend coding the LEFT outer-join.*

## Observations

The previous sample queries allow us to make some general observations. These observations imply that most query objectives can be satisfied by coding a LEFT outer-join.

**Assumption:** You want to code an outer-join operation based upon a PK-FK relationship, and the FK column is non-null.

You **only need to code a LEFT outer-join. Both the RIGHT outer-join and the FULL outer-join are unnecessary.** The following examples reference the DEPARTMENT and EMPLOYEE tables to illustrate this observation.

**RIGHT outer-join is unnecessary:** If you want the join-result to contain all rows from a specified table, designate that table as the left-table and perform a LEFT outer-join. (Sample Query 1.4 shows that this observation applies even if you are not joining on a PK-FK relationship.) The following statements are logically equivalent. The result tables contain the same data (although their row/column sequence may differ).

```
SELECT *
FROM EMPLOYEE E RIGHT OUTER JOIN DEPARTMENT D
ON   E.DNO = D.DNO

SELECT *
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E } ←Preferred
ON   D.DNO = E.DNO
```

**FULL outer-join is unnecessary:** The following statements are equivalent, and the result tables contain the same data (although their row/column sequence may differ).

```
SELECT *
FROM DEPARTMENT D FULL OUTER JOIN EMPLOYEE E
ON   D.DNO = E.DNO

SELECT *
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E } ←Preferred
ON   D.DNO = E.DNO
```

## Exercises

- 19A. Reference the REGION and STATE tables in the MTPCH database. Designate REGION as the left-table. Execute a full outer-join.
- 19B. Reference the REGION and STATE tables in the MTPCH database. Designate REGION as the left-table. Execute a left outer-join. (Observe that the result is the same as the previous exercise.)
- 19C. Reference the REGION and STATE tables in the MTPCH database. Designate STATE as the right-table. Execute a right outer-join. (Observe that the result is the same as that produced by an inner-join.)
- 19D. We generally discourage use of the right outer-join. But, for tutorial purposes only, you are asked to use the right outer-join to satisfy the query objective for Sample Query 19.2: Reference the DEPARTMENT and EMPLOYEE3 tables. Display all information about *all* departments along with all information about employees who work in those departments. (Display information about every department, even if the department does not have any employees.) Sort the result by ENO within DNO.

## Review: Inner-Join & Restriction

In Chapter 16 (Section on "INNER JOIN-ON Syntax: WHERE versus AND") we noted that, when specifying an inner-join using the JOIN-ON syntax, you can replace WHERE with AND. Therefore, the following two statements produce the same results.

### Equivalent Statements

#### Statement-1

```
SELECT *
FROM DEPARTMENT D INNER JOIN EMPLOYEE E
ON E.DNO = D.DNO
WHERE E.SALARY < 1000.00
```

#### Statement-2

```
SELECT *
FROM DEPARTMENT D INNER JOIN EMPLOYEE E
ON E.DNO = D.DNO
AND E.SALARY < 1000.00
```

Both statements produce the same result.

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40

This equivalency occurs because, with an inner-join operation, the E.SALARY < 1000.00 condition can be applied during the inner-join operation or after the inner-join operation.

### Important Observation:

Statement-1 specifies two operations: (1) An Inner-join (JOIN-ON) followed by (2) a restriction (WHERE).

Statement-2 specifies just one operation: This is an Inner-join (JOIN-ON) with a *compound-condition*. Visually, it may be easier to recognize this by rewriting Statement-2 as:

```
SELECT *
FROM DEPARTMENT D INNER JOIN EMPLOYEE E
ON E.DNO = D.DNO AND E.SALARY < 1000.00
```

## CAREFUL! Outer-Join & Restriction

With an **outer-join** operation, you **cannot** arbitrarily interchange **WHERE** and. Observe that the following two statements produce different results.

### Not Equivalent: Different Results

#### Statement-3

```
SELECT *
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E
ON E.DNO = D.DNO
WHERE D.BUDGET < 24000.00
ORDER BY D.DNO, E.ENO
```

#### Statement-4

```
SELECT *
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E
ON E.DNO = D.DNO
AND D.BUDGET < 24000.00
ORDER BY D.DNO, E.ENO
```

Statement-3 result is:

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-

The Statement-4 result is:

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	-	-	-	-
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	-	-	-	-

The following pages explain these different results.



## Restriction (WHERE): Applied *After* the Outer-Join

Consider the preceding Statement-3.

```
SELECT *
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E
ON E.DNO = D.DNO
WHERE D.BUDGET < 24000.00
ORDER BY D.DNO, E.ENO
```

**Logic:** This statement asks the system to execute two operations.

1. A left outer-join (specified with JOIN-ON syntax), and
2. A restriction is specified by the WHERE-clause

Details:

1. The system executes the LEFT OUTER JOIN to generate the following intermediate result.

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	2000	LARRY	2000.00	10
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40

Because the left outer-join result includes the non-matching DEPARTMENT 30 row, it can appear in the final result *if* it satisfies the following WHERE-clause.

2. *After* the outer-join is executed, the WHERE-clause restriction (D.BUDGET < 24000.00) is applied to the above intermediate result to produce the following final result.

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-

Observe that the non-matching DEPARTMENT 30 row appears in the final result because its BUDGET value (7000.00) satisfies the WHERE-clause.

## Compound-Condition (AND): Applied *During* the Outer-Join

Consider the preceding Statement-4.

```
SELECT *
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E
ON E.DNO = D.DNO
AND D.BUDGET < 24000.00
ORDER BY D.DNO, E.ENO
```

**Syntax:** This ON-clause specifies a compound join-condition where the `D.BUDGET < 24000.00` condition is part of the join-condition. Rewriting this statement emphasizes this logic.

```
SELECT *
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E
ON E.DNO = D.DNO AND D.BUDGET < 24000.00
ORDER BY D.DNO, E.ENO
```

**Logic:** This statement asks the system to execute one operation, a left outer-join which happens to have a compound-condition (`E.DNO = D.DNO AND D.BUDGET < 24000.00`). Hence the `D.BUDGET < 24000.00` condition is evaluated *during* the outer-join operation. As with any left outer-join, the system determines the matching rows, and then it appends the non-matching rows from the left-table. The matching rows (match both conditions in the compound-condition) are:

20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20

The non-matching left-table (DEPARTMENT) rows are:

10	ACCOUNTING	75000.00	-	-	-	-
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	-	-	-	-

Hence the outer-join result is:

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	-	-	-	-
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	-	-	-	-

## Restriction (WHERE): Applied *After* the Outer-Join

Statement-3 and Statement-4 presented the basic syntax and logic of "WHERE versus AND" within the context of an outer-join operation. The next two sample queries work backwards. We start with these statements and then present reasonable query objectives that are satisfied by the statements.

The following sample query is satisfied by Statement-3.

**Sample Query 19.7:** Reference the DEPARTMENT and EMPLOYEE tables. Display all information about *every department (including departments without employees)* which have a budget that is less than \$24,000. For all such departments, display their departmental information, followed by their employee information. Sort the result by ENO within DNO.

```
SELECT *  
  
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E  
ON   D.DNO = E.DNO  
  
WHERE D.BUDGET < 24000.00  
  
ORDER BY D.DNO, E.ENO
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-

**Logic:** A left outer-join is required because the query objective specifies "including departments without employees." This will include Department 30 in the intermediate outer-join result. Because Department 30 meets the D.BUDGET < 24000.00 condition, it appears in the final result.

## Compound-Condition (AND): Applied *During* the Outer-Join

In the following sample query, the specification of AND indicates that the D.BUDGET < 24000.00 condition is to be evaluated during the outer-join operation.

The following sample query is satisfied by Statement-4.

**Sample Query 19.8:** Reference the DEPARTMENT and EMPLOYEE tables. Display all information about all departments. Also, if a department has a budget that is less than \$24,000, display all information about that department's employees. Sort the result by ENO within DNO.

```
SELECT *  
  
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E  
ON   D.DNO = E.DNO AND D.BUDGET < 24000.00  
  
ORDER BY D.DNO, E.ENO
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	-	-	-	-
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	-	-	-	-

**Logic:** A left outer-join is required because the query objective specifies "Display all information about all departments." The following non-matching DEPARTMENT rows appear in the result because they do not match the compound ON-condition.

10	ACCOUNTING	75000.00	-	-	-	-
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	-	-	-	-

Note that information about Employees 2000, 4000, and 5000 do not appear because they work in departments which do not match the BUDGET < 24000.00 condition.

## WHERE-Condition References Column in RIGHT-Table

Note that Statement-3 and Statement-4 specified a restriction (D.BUDGET < 24000) that references a column in the LEFT-table (the parent-table). The following Statement-5 and Statement-6 specify a restriction (E.SALARY < 1000.00) that references a column from the RIGHT-table (the child-table). Again, these examples show that, with an outer-join, you cannot arbitrarily swap WHERE and.

### Not Equivalent: Different Results

#### Statement-5

```
SELECT *
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E
ON E.DNO = D.DNO
WHERE E.SALARY < 1000.00
ORDER BY D.DNO, E.ENO
```

#### Statement-6

```
SELECT *
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E
ON E.DNO = D.DNO
AND E.SALARY < 1000.00
ORDER BY D.DNO, E.ENO
```

The Statement-5 result is:

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>	<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO1</u>
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40

The Statement-6 result is:

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>	<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO1</u>
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	-	-	-	-
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40

There is nothing new about the logical behavior of these statements. However, referencing a child-column in a WHERE-clause (as in Statement-5) merits consideration.

Consider the Statement-5 showing: **WHERE E.SALARY < 1000.00**

The system initially executes the LEFT OUTER JOIN to generate the following intermediate result.

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	2000	LARRY	2000.00	10
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40

Next WHERE-clause restriction (E.SALARY < 1000.00) is applied to this intermediate result to produce the following final result.

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40

Observe that the non-matching DEPARTMENT 30 row is excluded from the final result because its null SALARY value cannot match any WHERE-condition. This example motivates the following observation.

**Important Observation:** Within the context of a left outer-join, if *WHERE-clause* that references a column from the *RIGHT-table* (e.g., *EMPLOYEE*) it can never select a non-matching row for the final result because all of its right-side columns always contain null value(s). **Hence, it makes no sense to specify an outer-join.** For Statement-5, you can (and probably should) code an inner-join operation as illustrated below.

```
SELECT *
FROM DEPARTMEN T D INNER JOIN EMPLOYEE E
ON D.DNO = E.DNO
WHERE E.SALARY < 1000.00
ORDER BY D.DNO, E.ENO
```

The above observation does not apply to Statement-6 which specifies AND (versus WHERE) for the E.SALARY < 1000.00 restriction. Notice that the Statement-6 result contains data from "non-matching" rows which cannot be realized by an inner-join.

The following sample query works backwards. It starts with Statement-6 and then presents a reasonable query objective that is satisfied by this statement. Note that, although this statement specifies a condition that references SALARY (a column from the child-table), you cannot rewrite an equivalent statement using an inner-join.

**Sample Query 19.9:** Reference the DEPARTMENT and EMPLOYEE tables. Display all information about all departments. Also, display all information about any employee whose salary is greater than \$2,000.00. Sort the result by ENO within DNO.

```
SELECT *

FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E
ON   D.DNO = E.DNO AND E.SALARY > 2000.00

ORDER BY D.DNO, E.ENO
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	-	-	-	-
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	-	-	-	-

**Logic:** During the LEFT outer-join, the matching rows (based upon the compound join-condition) are:

20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20

Three non-matching DEPARTMENT rows (with NULL values appended) are also included to produce the final LEFT outer-join result.

10	ACCOUNTING	75000.00	-	-	-	-
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	-	-	-	-

## One More Example

The following sample query brings together concepts presented in the previous sample queries. The ON-clause specifies a compound join-condition, and the WHERE-clause specifies a restriction on the outer-join result.

**Sample Query 19.10:** Reference the DEPARTMENT and EMPLOYEE tables. Display all information about every department, except DEPARTMENT 40. If any department, excluding Department 40, has a budget that is less than \$24,000.00, display all information about the department's employees. Sort the result by ENO within DNO.

```
SELECT *  
  
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E  
ON   D.DNO = E.DNO AND D.BUDGET < 24000.00  
  
WHERE D.DNO <> 40  
  
ORDER BY D.DNO, E.ENO
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	-	-	-	-
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-

**Logic:** The left outer-join produces the following intermediate result.

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	-	-	-	-
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	-	-	-	-
40	ENGINEERING	25000.00	-	-	-	-

Next, the WHERE-clause excludes the row for DEPARTMENT 40 to produce the final result.



## Summary: Outer-Join & Restriction

Consider the following skeleton-code.

```
SELECT *
FROM T1 LEFT OUTER JOIN T2
ON Condition-1
AND Condition-2
AND Condition-3
WHERE Condition-4
AND Condition-5
AND Condition-6
```

Although unnecessary, it may be helpful to rewrite this skeleton-code as:

```
SELECT *
FROM T1 LEFT OUTER JOIN T2
ON (Condition-1 AND Condition-2 AND Condition-3)
WHERE (Condition-4 AND Condition-5 AND Condition-6)
```

This statement includes two compound-conditions. The first compound-condition is specified in the ON-clause.

```
ON Condition-1 AND Condition-2 AND Condition-3
```

Because this compound-condition follows ON, it defines the outer-join comparison logic that is implemented *during* the outer-join operation. This outer-join produces an intermediate result.

The second compound-condition is specified in the WHERE-clause.

```
WHERE Condition-4 AND Condition-5 AND Condition-6
```

Because this compound-condition is specified in the WHERE-clause, it defines a restriction operation that is implemented *after* the outer-join operation has produced its intermediate result. This second compound-condition restricts this intermediate result to produce the final result.

## **Final Observation: Can You Always Avoid Null Values?**

Assume your DBA wants to avoid problems with null values. Therefore, whenever she creates a table, she always declares all columns to be NOT NULL. Then the system will reject any operation that attempts to store a null value in any table.

Question: Under this (perhaps unrealistic) assumption, can you conclude that, when coding your SELECT statements, you never have to consider null values?

Answer: No! You can avoid null values only *if* you decide to never execute an outer-join operation.

Revisit Figures 19.1 and 19.2. Notice that the MAN and DOG tables do not contain any null values. However, in these figures, four of the six result tables produced by the three kinds of outer-join operations contain null values.

Conclusion: Your DBA has prohibited null values from all *base* tables. However, assuming that some users will execute outer-join-operations, null values may appear in intermediate and final result tables.

## Important Exercises

Exercises 19E AND 19F reference the following versions of the MAN and DOG tables. These tables are related via a PK-FK relationship (DOG.MNO references MAN.MNO). These are paper-and-pencil exercises because the MAN and DOG tables were not created in the CREATE-ALL-TABLES script.

<u>MAN</u>		<u>DOG</u>		
<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO</u>
77	MOE	1000	SPOT	99
88	LARRY	3000	ROVER	77
99	CURLY	2000	WALLY	99
		4000	SPIKE	99

19E. What are the result tables produced by the following left outer-join operations?

- a. 

```
SELECT *
FROM MAN LEFT OUTER JOIN DOG
ON MAN.MNO = DOG.MNO
WHERE MAN.MNAME LIKE '%R%'
```
- b. 

```
SELECT *
FROM MAN LEFT OUTER JOIN DOG
ON MAN.MNO = DOG.MNO
AND MAN.MNAME LIKE '%R%'
```

19F. What are the result tables produced by the following left outer-join operations?

- a. 

```
SELECT *
FROM MAN LEFT OUTER JOIN DOG
ON MAN.MNO = DOG.MNO
WHERE DOG.DNAME LIKE 'S%'
```
- b. 

```
SELECT *
FROM MAN LEFT OUTER JOIN DOG
ON MAN.MNO = DOG.MNO
AND DOG.DNAME LIKE 'S%'
```

## Summary

This chapter introduced the basic concepts of the LEFT, RIGHT, and FULL outer-join operations. The syntax for these operations is a simple variation of the JOIN-ON syntax introduced in Chapter 16.

```
SELECT . . .  
FROM TABLE1  $\left( \begin{array}{l} \text{LEFT} \\ \text{RIGHT} \\ \text{FULL} \end{array} \right)$  OUTER JOIN TABLE2  
ON join-condition
```

The LEFT Outer-Join result table contains all matching rows plus all non-matching rows from the LEFT table only.

The RIGHT Outer-Join result table contains all matching rows plus all non-matching rows from the RIGHT table only.

The FULL Outer-Join result table contains all matching rows, plus all non-matching rows from both tables.

The basic syntax and logic of the outer-join operations appears to be rather straightforward. However, as the last few sample queries and exercises illustrated, sometimes the logic can become a little tricky.

## Summary Exercises

The following exercises refer to the DEPARTMENT and EMPLOYEE tables.

- 19G. Display the name and budget for *all* departments. Also display the name and salary of any employee who works in a department having a budget that exceeds \$50,000.00. The result should look like:

<u>DNAME</u>	<u>BUDGET</u>	<u>ENAME</u>	<u>SALARY</u>
ACCOUNTING	75000.00	LARRY	2000.00
ACCOUNTING	75000.00	JOE	400.00
INFO. SYS.	20000.00	-	-
PRODUCTION	7000.00	-	-
ENGINEERING	25000.00	-	-

- 19H. Display the name and budget of those departments having a budget that is less than \$50,000.00. If any such department has employees, also display name and salary of the employees. The result should look like:

<u>DNAME</u>	<u>BUDGET</u>	<u>ENAME</u>	<u>SALARY</u>
INFO. SYS.	20000.00	MOE	2000.00
INFO. SYS.	20000.00	GEORGE	9000.00
INFO. SYS.	20000.00	CURLY	3000.00
PRODUCTION	7000.00	-	-
ENGINEERING	25000.00	SHEMP	500.00

19I. Display the name and salary of any employee who earns more than \$1,000.00, along with the employee's departmental name and budget. The result looks like:

<u>ENAME</u>	<u>SALARY</u>	<u>DNAME</u>	<u>BUDGET</u>
LARRY	2000.00	ACCOUNTING	75000.00
MOE	2000.00	INFO. SYS.	20000.00
GEORGE	9000.00	INFO. SYS.	20000.00
CURLY	3000.00	INFO. SYS.	20000.00

19J. Display the name and budget for *all* departments. Also, display the name and salary of any employee who works in each department and earns more than \$1,000.00. The result should look like:

<u>DNAME</u>	<u>BUDGET</u>	<u>ENAME</u>	<u>SALARY</u>
ACCOUNTING	75000.00	LARRY	2000.00
INFO. SYS.	20000.00	MOE	2000.00
INFO. SYS.	20000.00	GEORGE	9000.00
INFO. SYS.	20000.00	CURLY	3000.00
PRODUCTION	7000.00	-	-
ENGINEERING	25000.00	-	-

19K. Consider Sample Query 19.10 shown below.

```
SELECT *
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E
ON D.DNO = E.DNO AND D.BUDGET < 24000.00
WHERE D.DNO <> 40
ORDER BY D.DNO, E.ENO
```

Assume you replaced the keyword WHERE with the keyword AND to formulate the following statement.

```
SELECT *
FROM DEPARTMENT D LEFT OUTER JOIN EMPLOYEE E
ON D.DNO = E.DNO AND D.BUDGET < 24000.00
AND D.DNO <> 40
ORDER BY D.DNO
```

Is this statement equivalent to the result shown for Sample Query 19.10? (Does it produce the same correct result?)

## Appendix 19A: Theory

Although outer-join operations can be very useful, these operations present multiple theoretical problems. (These problems may be the reason Codd did not include outer-joins in his original relational algebra.) We describe two such problems below.

**No Primary Key in Result Table:** The most *fundamental problem* is that, given two relations, the outer-join result of these relations might not be a relation. For example, consider the following LEFT OUTER JOIN of MAN and DOG where MAN is the left-relation. Assume MNO is the primary key of MAN; DNO is the primary key of DOG; and DOG.MNO is a foreign-key that references MAN.

<u>MAN</u>	<u>MNO</u>	<u>MNAME</u>	<u>DOG</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO</u>
	77	MOE		1000	SPOT	99
	88	LARRY		3000	ROVER	77
	99	CURLY		2000	WALLY	99
				4000	SPIKE	99

The following result is a table, but it is not a valid relation because no single column, or combination of columns, can serve as its primary key.

<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO1</u>
77	MOE	3000	ROVER	77
88	LARRY	-	-	-
99	CURLY	1000	SPOT	99
99	CURLY	2000	WALLY	99
99	CURLY	4000	SPIKE	99

The following observations imply that it is impossible to designate any column, or any combination of these columns, as a primary key.

- The MNO and MNAME columns from MAN, the parent-table, may contain duplicate values.
- The DNO, DNAME, and MNO1 columns from DOG, the child-table, may contain null values.

Similar problems apply to the RIGHT and FULL outer-join operations.

**Loss of Closure:** If you start with valid relation(s) and then apply any relational algebraic operation, *ideally*, the result should always be a relation. This property is called "closure." Closure applies to the RESTRICT, PROJECT, CROSS, INNER-JOIN, (and the UNION, INTERSECT, and EXCEPT operations to be presented in Chapter 21.) Unfortunately, as illustrated by the previous example, closure does not apply to the outer-join operations.

**Another Problem:** In the Chapter 16 we noted that the system produces the same result if (1) the inner-join is executed before restriction, or (2) if restriction is executed before the inner-join. The following example shows that this equivalency does not apply to the left outer-join.

Outer-Join,  
Then Restrict

1. MAN LEFT OUTER JOIN DOG ON MAN.MNO = DOG.MNO

<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO1</u>
77	MOE	3000	ROVER	77
88	LARRY	-	-	-
99	CURLY	1000	SPOT	99
99	CURLY	3000	WALLY	99
99	CURLY	4000	SPIKE	99

2. WHERE DOG.DNAME LIKE 'S%'

<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO1</u>
99	CURLY	1000	SPOT	99
99	CURLY	4000	SPIKE	99

Restrict,  
Then Outer-Join

1. WHERE DOG.DNAME LIKE 'S%'

<u>MAN</u>		<u>TEMPDOG</u>		
<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO</u>
77	MOE	1000	SPOT	99
88	LARRY	4000	SPIKE	99
99	CURLY			

2. MAN LEFT OUTER JOIN TEMPDOG  
ON MAN.MNO = TEMPDOG.MNO

<u>MNO</u>	<u>MNAME</u>	<u>DNO</u>	<u>DNAME</u>	<u>MNO1</u>
77	MOE	-	-	-
88	LARRY	-	-	-
99	CURLY	1000	SPOT	99
99	CURLY	4000	SPIKE	99

} Different Result



## Appendix 19B: Theory & Efficiency

Outer-join operations tend to be slower than inner-join operations for two reasons,

1. With inner join, the optimizer can capitalize on the efficiency technique of executing restriction before inner-join. (See the RESTRICT-PROJECT-First Procedure in Appendix 17C and Figure 17C.1.) However, as illustrated in the previous Appendix 19A, this efficiency technique cannot be used with an outer-join operation.
2. Inner-join is commutative.

$$T1 \text{ inner-join } T2 = T2 \text{ inner-join } T1$$

Therefore, when optimizing an inner-join of tables T1 and T2, the optimizer can capitalize on the option of choosing either T1 or T2 as the driving-table.

However, outer-join is not commutative.

$$T1 \text{ outer-join } T2 \neq T2 \text{ outer-join } T2$$

Therefore, when optimizing an outer-join operation, the optimizer *cannot* arbitrarily choose either T1 or T2 as the driving-table.

## Appendix 19C: De-Normalized Tables (Again)

In Appendix 16A we considered the merits of creating the de-normalized DNEmployee table by pre-joining the DEPARTMENT and EMPLOYEE tables. This pre-join operation was an inner-join and produced a table that looked like:

ENO	ENAME	SALARY	DNO	DNAME	BUDGET
1000	MOE	2000.00	20	INFO. SYS.	20000.00
2000	LARRY	2000.00	10	ACCOUNTING	75000.00
3000	CURLY	3000.00	20	INFO. SYS.	20000.00
4000	SHEMP	500.00	40	ENGINEERING	25000.00
5000	JOE	400.00	10	ACCOUNTING	75000.00
6000	GEORGE	9000.00	20	INFO. SYS.	20000.00

DNEmployee

In Appendix 16A we noted that the major problem with this de-normalized table is the Lost Information Problem. Note that DNEmployee lost information about Department 30 because it did not match on the inner-join operation.

Now, what if we attempted to resolve this problem by executing a Left Outer-Join of DEPARTMENT and EMPLOYEE (similar to Sample Query 19.6) and stored the result in a table called XDNEmployee as shown below?

DNO	DNAME	BUDGET	ENO	ENAME	SALARY
10	ACCOUNTING	75000.00	2000	LARRY	2000.00
10	ACCOUNTING	75000.00	5000	JOE	400.00
20	INFO. SYS.	20000.00	1000	MOE	2000.00
20	INFO. SYS.	20000.00	3000	CURLY	3000.00
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00
<b>30</b>	<b>PRODUCTION</b>	<b>7000.00</b>	-	-	-
40	ENGINEERING	25000.00	4000	SHEMP	500.00

XDNEmployee

Note that XDNEmployee has information about all departments and all employees (no Lost Information).

*However! Is XDNEmployee really a good idea? Probably not, because this table suffers from the "No Possible Primary Key Problem" described in Appendix 19A.*

This page is intentionally blank.

## Multi-Table Left Outer-Joins

We have already seen that a multi-table INNER JOIN is a straightforward extension of a two-table INNER JOIN. Likewise, we will see that a multi-table LEFT OUTER JOIN is (almost) a straightforward extension of a two-table LEFT OUTER JOIN.

This chapter is organized into two sections.

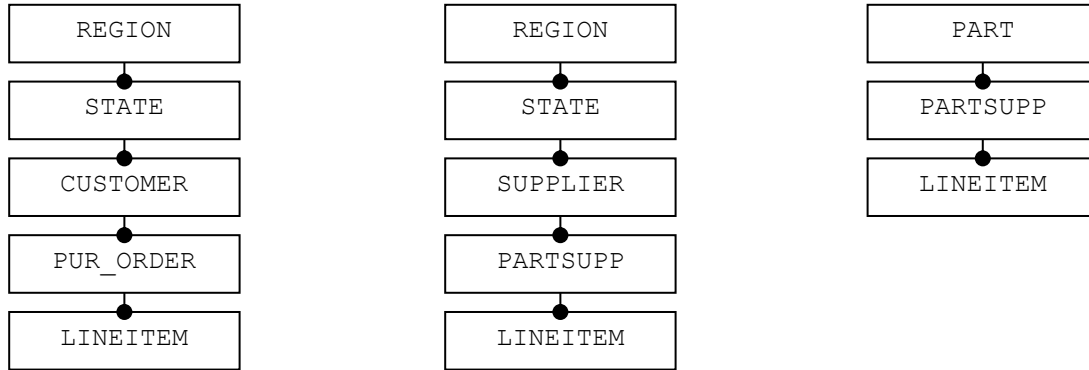
**Section-A: Top-Down Traversal of Table-Hierarchy:** This section will present SELECT statements that specify multi-table LEFT OUTER JOIN operations to select data from tables that lie along a hierarchical path. These tables will be traversed in a top-down manner.

**Section-B: Non-Top-Down Traversal of Table-Hierarchy:** As in the previous section, SELECT statements will specify multi-table LEFT OUTER JOIN operations to select data from tables that lie along a hierarchical path. However, these tables will *not* be traversed in a top-down manner. This section will be very useful in the following (optional) Chapter 20.5 where an individual SELECT statement specifies both LEFT OUTER JOIN and INNER JOIN operations.

**Terminology:** Unlike previous chapters, in this chapter whenever we say “join” we mean LEFT OUTER JOIN.

## A. Top-Down Traversal of Table-Hierarchy

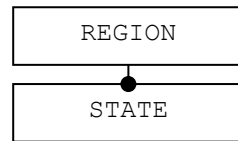
Chapter 18 presented INNER JOIN operations that traversed the following table-hierarchies in the MTPCH database. (Review the MTPCH database in Figure 18.4.)



This chapter's sample queries will specify multiple LEFT OUTER JOIN operations that also traverse these same table-hierarchies. These sample queries will traverse these hierarchies in a top-down manner, by initially joining the "top two" tables in the hierarchy.

## Review: Two-Table LEFT OUTER JOIN along Hierarchical Path

The following sample query performs a LEFT OUTER JOIN of the REGION and STATE tables based on their PK-FK relationship. Observe that these tables form a trivial hierarchy.



**Sample Query 20.1:** Display the number and name of every region followed by the number and name of every state in each region. Sort the result by STCODE within RNO.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME
FROM REGION R LEFT OUTER JOIN STATE ST ON R.RNO = ST.RNO
ORDER BY R.RNO, ST.STCODE
```

<u>RNO</u>	<u>RNAME</u>	<u>STCODE</u>	<u>STNAME</u>	
1	NORTHEAST	CT	CONNECTICUT	
1	NORTHEAST	MA	MASSACHUSETTS	
2	NORTHWEST	OR	OREGON	
2	NORTHWEST	WA	WASHINGTON	
3	SOUTHEAST	FL	FLORIDA	
3	SOUTHEAST	GE	GEORGIA	
4	SOUTHWEST	AZ	ARIZONA	
4	SOUTHWEST	NM	NEW MEXICO	
5	MIDWEST	-	-	← R-No-ST

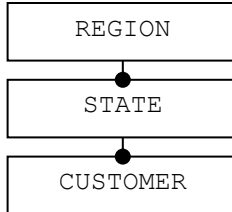
**Notation:** The (← R-No-ST) notation designates a non-matching row corresponding to a "Region with no States."

**Syntax and Logic:** Nothing New. This SELECT statement is coded to perform a top-down traversal of the REGION and STATE tables. The result table join contains *all* RNO and RNAME values because REGION is the left-table (the "top" table), and it contains *all* STCODE and STNAME values because every STATE row must match some REGION row.

Casually speaking, we "*display data from all rows in both tables.*" The next sample query will "*display data from all rows in three tables.*"

## Three-Table LEFT OUTER JOIN along Hierarchical Path

The next sample query performs a top-down traversal of all tables in the following REGION-STATE-CUSTOMER hierarchy.



**Sample Query 20.2:** Display the number and name of every region, the code and name of every state in each region, and the number and name of every customer in each state. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, CNO, and CNAME. Sort the result by CNO within STCODE within RNO.

```

SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME

FROM REGION R
     LEFT OUTER JOIN STATE ST   ON R.RNO = ST.RNO
     LEFT OUTER JOIN CUSTOMER C ON ST.STCODE = C.STCODE

ORDER BY R.RNO, ST.STCODE, C.CNO
  
```

RNO	RNAME	STCODE	STNAME	CNO	CNAME	
1	NORTHEAST	CT	CONNECTICUT	-	-	← ST-No-C
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	
1	NORTHEAST	MA	MASSACHUSETTS	230	BOLYAI	
1	NORTHEAST	MA	MASSACHUSETTS	500	HILBERT	
2	NORTHWEST	OR	OREGON	300	NEWTON	
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	
3	SOUTHEAST	FL	FLORIDA	600	BOOLE	
3	SOUTHEAST	FL	FLORIDA	660	CANTOR	
3	SOUTHEAST	GE	GEORGIA	700	RUSSELL	
3	SOUTHEAST	GE	GEORGIA	770	GODEL	
4	SOUTHWEST	AZ	ARIZONA	880	TURING	
4	SOUTHWEST	AZ	ARIZONA	890	MANDELBROT	
4	SOUTHWEST	NM	NEW MEXICO	780	CHURCH	
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	
5	MIDWEST	-	-	-	-	← R-No-ST

The result table contains data from:

- All REGION rows (including regions that do not have any states)
- All STATE rows (including states that do not have any customers)
- All CUSTOMER rows

Hence, **this result table contains some data from "all rows in all tables" in the hierarchy.** Observe that the result table contains the non-matching REGION value (MIDWEST) and the non-matching STATE value (CONNECTICUT). These rows are identified by the following notation.

← R-No-ST designates a non-matching row corresponding to a "Region with no States."

← ST-No-C designates a non-matching row corresponding to a "State with no Customers."

**"Top-Down" Join-Sequence:** Logically, the first LEFT OUTER JOIN operation joins the "top two" tables, REGION and STATE, to produce an intermediate join-result (which is the same as the final result for the preceding Sample Query 20.1). Then, the second LEFT OUTER JOIN operation joins this intermediate result (as the left-table) with the CUSTOMER table to produce the final join-result. We refer to this join-sequence as a "*top-down*" *join-sequence*. This top-down join-sequence applies to all SELECT statements presented in this Section-A.

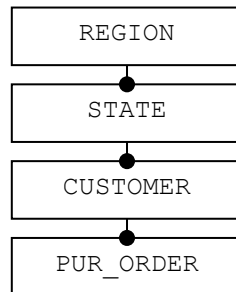
### **Exercise:**

20A. Display the number and name of every region, the number and name of every state in each region, and the number and name of every supplier in each state. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, SNO, and SNAME. Sort the result by SNO within STCODE within RNO. (Hint: Follow the REGION-STATE-SUPPLIER hierarchical path.)



## Four-Table LEFT OUTER JOIN along Hierarchical Path

The next sample query performs a top-down traversal of all tables in the following REGION-STATE-CUSTOMER-PUR\_ORDER hierarchy.



**Sample Query 20.3:** Display the number and name of every region, the code and name of every state in each region, the number and name of every customer in each state, and the purchase-order number and status of every purchase-order completed by each customer. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, CNO, CNAME, PONO, and POSTATUS. Sort the result by PONO within CNO within STCODE within RNO.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME, PO.PONO, PO.POSTATUS

FROM REGION R
     LEFT OUTER JOIN STATE ST      ON R.RNO = ST.RNO
     LEFT OUTER JOIN CUSTOMER C    ON ST.STCODE = C.STCODE
     LEFT OUTER JOIN PUR_ORDER PO  ON C.CNO = PO.CNO

ORDER BY R.RNO, ST.STCODE, C.CNO, PO.PONO
```

[Result table shown on following page]

**Syntax & Logic:** Nothing New. This statement performs three LEFT OUTER JOIN operations in a top-down join-sequence. Logically, the REGION and STATE tables are initially joined to form an intermediate join-result. This result is joined to the CUSTOMER table to form another intermediate join-result, which is joined to the PUR\_ORDER table to generate the final join-result.

RNO	RNAME	STCODE	STNAME	CNO	CNAME	PONO	POSTATUS
1	NORTHEAST	CT	CONNECTICUT	-	-	-	- ← ST-No-C
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11101	C
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11102	P
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	11108	C
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	11109	P
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	11110	C
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	11111	P
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	11120	C
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	11121	C
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	11122	P
1	NORTHEAST	MA	MASSACHUSETTS	230	BOLYAI	11124	P
1	NORTHEAST	MA	MASSACHUSETTS	500	HILBERT	11150	P
2	NORTHWEST	OR	OREGON	300	NEWTON	11130	C
2	NORTHWEST	OR	OREGON	300	NEWTON	11133	P
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	11139	C
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	11141	P
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	11142	C
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	11144	P
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	11146	C
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	11148	C
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	11149	P
3	SOUTHEAST	FL	FLORIDA	600	BOOLE	11152	C
3	SOUTHEAST	FL	FLORIDA	600	BOOLE	11153	P
3	SOUTHEAST	FL	FLORIDA	660	CANTOR	11154	C
3	SOUTHEAST	FL	FLORIDA	660	CANTOR	11155	P
3	SOUTHEAST	GE	GEORGIA	700	RUSSELL	11156	C
3	SOUTHEAST	GE	GEORGIA	770	GODEL	11157	P
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11159	C
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11160	P
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11170	P
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11198	P
4	SOUTHWEST	AZ	ARIZONA	890	MANDELBROT	-	- ← C-No-PO
4	SOUTHWEST	NM	NEW MEXICO	780	CHURCH	-	- ← C-No-PO
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	11158	C
5	MIDWEST	-	-	-	-	-	- ← R-No-ST

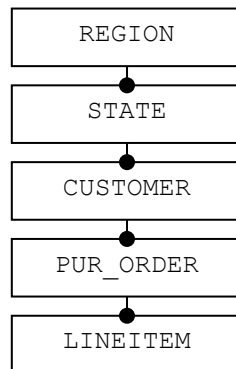
**Notation:** ← C-No-PO designates a non-matching row corresponding to a "Customer with no Purchase-Orders."

**Exercise:**

20B. Display the number and name of every region, the code and name of every state in each region, the number and name of every supplier in each state, and the part number of every part that you can purchase from these suppliers. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, SNO, SNAME, PNO. Sort the result by PNO within SNO within STCODE within RNO. (Hint: Follow the REGION-STATE-SUPPLIER-PARTSUPP hierarchical path.)

## Five-Table LEFT OUTER JOIN along Hierarchical Path

The following sample query performs a top-down traversal of all tables in the REGION-STATE-CUSTOMER-PUR\_ORDER-LINEITEM hierarchy.



**Sample Query 20.4:** Display the number and name of every region, the code of every state in each region, the number of every customer in each state, the purchase-order number of every purchase-order completed by each customer, and the part number (PNO) and supplier number (SNO) specified in the purchase-order's line-items. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, CNO, PONO, PNO, and SNO. Sort the result by PNO within PONO within CNO within STCODE within RNO.

```
SELECT R.RNO, R.RNAME, ST.STCODE, C.CNO,
       PO.PONO, LI.PNO, LI.SNO

FROM REGION R
LEFT OUTER JOIN STATE ST      ON R.RNO = ST.RNO
LEFT OUTER JOIN CUSTOMER C    ON ST.STCODE = C.STCODE
LEFT OUTER JOIN PUR_ORDER PO  ON C.CNO = PO.CNO
LEFT OUTER JOIN LINEITEM LI   ON PO.PONO = LI.PONO

ORDER BY R.RNO, ST.STCODE, C.CNO, PO.PONO, LI.PNO
```

[Following page shows some of the 67 returned rows.]

**Notation:** ← PO-No-LI designates a non-matching row corresponding to a "Purchase-Order with no Line-Items."

**Syntax & Logic:** Logically, by following a top-down join-sequence, the REGION and STATE tables are joined to form an intermediate join-result, which is joined to CUSTOMER to form another intermediate join-result, which is joined to PUR\_ORDER to form another intermediate join-result, which is joined to LINEITEM to form the final join-result.

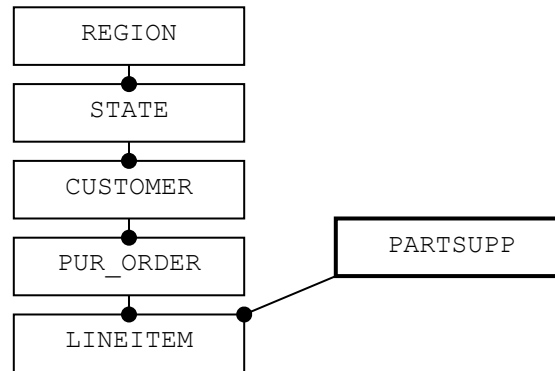
RNO	RNAME	STCODE	CNO	PONO	PNO	SNO	
1	NORTHEAST	CT	-	-	-	-	← ST-No-C
1	NORTHEAST	MA	100	11101	P1	S2	
1	NORTHEAST	MA	100	11101	P3	S3	
1	NORTHEAST	MA	100	11102	P3	S3	
1	NORTHEAST	MA	100	11102	P4	S4	
1	NORTHEAST	MA	110	11108	P5	S1	
1	NORTHEAST	MA	110	11108	P6	S4	
1	NORTHEAST	MA	110	11109	P1	S2	
1	NORTHEAST	MA	110	11109	P7	S2	
1	NORTHEAST	MA	110	11109	P8	S4	
1	NORTHEAST	MA	200	11110	P8	S4	
1	NORTHEAST	MA	200	11111	P1	S4	
1	NORTHEAST	MA	200	11111	P3	S4	
1	NORTHEAST	MA	220	11120	P4	S4	
1	NORTHEAST	MA	220	11120	P5	S2	
1	NORTHEAST	MA	220	11121	P6	S6	
1	NORTHEAST	MA	220	11121	P7	S4	
1	NORTHEAST	MA	220	11122	P1	S2	
1	NORTHEAST	MA	220	11122	P3	S3	
1	NORTHEAST	MA	230	11124	P4	S4	
1	NORTHEAST	MA	230	11124	P5	S1	
1	NORTHEAST	MA	500	11150	P3	S4	
1	NORTHEAST	MA	500	11150	P6	S4	
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
4	SOUTHWEST	AZ	880	11159	P6	S4	
4	SOUTHWEST	AZ	880	11159	P7	S2	
4	SOUTHWEST	AZ	880	11160	P1	S2	
4	SOUTHWEST	AZ	880	11160	P7	S2	
4	SOUTHWEST	AZ	880	11170	P3	S4	
4	SOUTHWEST	AZ	880	11170	P4	S4	
4	SOUTHWEST	AZ	880	11198	-	-	← PO-No-LI
4	SOUTHWEST	AZ	890	-	-	-	← C-No-PO
4	SOUTHWEST	NM	780	-	-	-	← C-No-PO
4	SOUTHWEST	NM	800	11158	P1	S2	
4	SOUTHWEST	NM	800	11158	P3	S4	
5	MIDWEST	-	-	-	-	-	← R-No-ST

**Exercise:**

20C. Display the number and name of every region, the code of every state in each region, the number and name of every supplier in each state, and the purchase-order number and part-number that appeared in every line-item for each supplier. Sort the result by PNO within PONO within SNO within STCODE within RNO. (Hints: Follow the REGION-STATE-SUPPLIER-PARTSUPP-LINEITEM hierarchy. Note that no data from the PARTSUPP table is displayed. This table serves as a link-table between the SUPPLIER and LINEITEM tables.)

## “Exiting the Hierarchy” (Careful!)

The following statement extends Sample Query 20.4. This statement involves “one more hop” to another table, the PARTSUPP table, to access and display PSPRICE values. There is nothing new from a syntax perspective. However, you must recognize that the PARTSUPP table is not a “sixth-table in a six-level hierarchy.”



**Sample Query 20.5:** Extend Sample Query 20.4. For each line-item, also display its LIPRICE value (in the LINEITEM table) and its corresponding PSPRICE value (from the PARTSUPP table).

```
SELECT R.RNO, R.RNAME, ST.STCODE, C.CNO,
       PO.PONO, LI.PNO, LI.SNO, LI.LIPRICE, PS.PSPRICE
FROM REGION R
LEFT OUTER JOIN STATE ST      ON R.RNO = ST.RNO
LEFT OUTER JOIN CUSTOMER C    ON ST.STCODE = C.STCODE
LEFT OUTER JOIN PUR_ORDER PO  ON C.CNO = PO.CNO
LEFT OUTER JOIN LINEITEM LI   ON PO.PONO = LI.PONO
LEFT OUTER JOIN PARTSUPP PS
ON LI.PNO = PS.PNO AND LI.SNO = PS.SNO
ORDER BY R.RNO, ST.STCODE, C.CNO, PO.PONO, LI.PNO
```

[Following page shows some of the 67 returned rows.]

**Logic:** Note that LINEITEM is the child (not the parent) in the one-to-many relationship between PARTSUPP and LINEITEM. Hence, we cannot assume that data from all rows in the parent PARTSUPP table will be displayed. Here, one PARTSUPP row (P7, S6) does not match a LINEITEM row.

Also, note that the one-to-many PK-FK relationship between the PARTSUPP and LINEITEM tables involves a composite key (PNO, SNO).

RNO	RNAME	STCODE	CNO	PONO	PNO	SNO	LIPRICE	PSPRICE
1	NORTHEAST	CT	-	-	-	-	-	-
1	NORTHEAST	MA	100	11101	P1	S2	11.50	10.50
1	NORTHEAST	MA	100	11101	P3	S3	12.00	12.00
1	NORTHEAST	MA	100	11102	P3	S3	13.00	12.00
1	NORTHEAST	MA	100	11102	P4	S4	13.00	12.00
1	NORTHEAST	MA	110	11108	P5	S1	11.00	10.00
1	NORTHEAST	MA	110	11108	P6	S4	5.00	4.00
1	NORTHEAST	MA	110	11109	P1	S2	11.50	10.50
1	NORTHEAST	MA	110	11109	P7	S2	3.00	2.00
1	NORTHEAST	MA	110	11109	P8	S4	6.00	5.00
1	NORTHEAST	MA	200	11110	P8	S4	6.00	5.00
1	NORTHEAST	MA	200	11111	P1	S4	12.00	11.00
1	NORTHEAST	MA	200	11111	P3	S4	13.50	12.50
1	NORTHEAST	MA	220	11120	P4	S4	13.00	12.00
1	NORTHEAST	MA	220	11120	P5	S2	11.00	10.00
1	NORTHEAST	MA	220	11121	P6	S6	5.00	4.00
1	NORTHEAST	MA	220	11121	P7	S4	4.00	3.00
1	NORTHEAST	MA	220	11122	P1	S2	11.50	10.50
1	NORTHEAST	MA	220	11122	P3	S3	13.00	12.00
1	NORTHEAST	MA	230	11124	P4	S4	13.00	12.00
1	NORTHEAST	MA	230	11124	P5	S1	11.00	10.00
1	NORTHEAST	MA	500	11150	P3	S4	13.50	12.50
1	NORTHEAST	MA	500	11150	P6	S4	5.00	4.00
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
4	SOUTHWEST	AZ	880	11159	P6	S4	5.00	4.00
4	SOUTHWEST	AZ	880	11159	P7	S2	3.00	2.00
4	SOUTHWEST	AZ	880	11160	P1	S2	12.50	10.50
4	SOUTHWEST	AZ	880	11160	P7	S2	3.00	2.00
4	SOUTHWEST	AZ	880	11170	P3	S4	12.50	12.50
4	SOUTHWEST	AZ	880	11170	P4	S4	13.00	12.00
4	SOUTHWEST	AZ	880	11198	-	-	-	-
4	SOUTHWEST	AZ	890	-	-	-	-	-
4	SOUTHWEST	NM	780	-	-	-	-	-
4	SOUTHWEST	NM	800	11158	P1	S2	11.50	10.50
4	SOUTHWEST	NM	800	11158	P3	S4	13.50	12.50
5	MIDWEST	-	-	-	-	-	-	-

**Observation:** The result table has 67 rows, the same number of rows in the previous result table for Sample Query 20.4. This occurred because each LINEITEM row matches exactly one row in the PARTSUPP table.

**Important Question:** Because PARTSUPP is not within the hierarchy of tables, could we specify an INNER JOIN (instead of a LEFT OUTER JOIN) of the LINEITEM and PARTSUPP tables? No! Try it. (See Exercise 20J.)

## Grouping and Summarizing

Chapters 9 and 9.5 introduced grouping rows and summarizing values from each group; and Chapter 18 (Multi-Table Inner-Joins) illustrated grouping and summarizing within a multi-table inner-join result. The following sample query illustrates grouping and summarizing with a multi-table outer-join result. Here, your logic must consider possible null values in the join-result produced by the LEFT OUTER JOIN operations.

**Sample Query 20.6:** Display the number and name of every region followed by the total price of all parts purchased by a customer in each region. Display the columns in the following left-to-right sequence: RNO, RNAME, and TOTALPRICE (column heading for the total price of all parts purchased in each region). Sort the result by RNO.

```
SELECT R.RNO, R.RNAME, COALESCE (SUM (LIPRICE), 0) TOTALPRICE
FROM REGION R
      LEFT OUTER JOIN STATE ST      ON R.RNO = ST.RNO
      LEFT OUTER JOIN CUSTOMER C    ON ST.STCODE = C.STCODE
      LEFT OUTER JOIN PUR_ORDER PO  ON C.CNO = PO.CNO
      LEFT OUTER JOIN LINEITEM LI   ON PO.PONO = LI.PONO
GROUP BY R.RNO, R.RNAME
ORDER BY R.RNO
```

RNO	RNAME	TOTALPRICE	
1	NORTHEAST	3110.00	
2	NORTHWEST	1790.00	
3	SOUTHEAST	2155.00	
4	SOUTHWEST	970.00	
5	MIDWEST	0.00	← R-No-ST

**Logic:** Nothing new. The join-result contains one row for the non-matching MIDWEST region with a null LIPRICE value. This row is the only row in the MIDWEST group. Because the SUM (LIPRICE) for this row is null, the COALESCE function is used to convert this null value to zero.

## Restriction (WHERE): Applied *After* the Outer-Join

Chapter 19 described potential problems associated with restrictions and null values as they relate to two-table outer-join operations. These same problems appear within multi-table outer-joins.

**Sample Query 20.7:** Display the number and name of every region except the SOUTHEAST region, the code and name of every state except those states located in the SOUTHEAST region, and the number and name of every customer except those customers located in states within the SOUTHEAST region. Sort the result by CNO within STCODE within RNO.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME
FROM REGION R
     LEFT OUTER JOIN STATE ST   ON R.RNO = ST.RNO
     LEFT OUTER JOIN CUSTOMER C ON ST.STCODE = C.STCODE
WHERE R.RNAME <> 'SOUTHEAST'
ORDER BY R.RNO, ST.STCODE, C.CNO
```

RNO	RNAME	STCODE	STNAME	CNO	CNAME	
1	NORTHEAST	CT	CONNECTICUT	-	-	← ST-No-C
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	
1	NORTHEAST	MA	MASSACHUSETTS	230	BOLYAI	
1	NORTHEAST	MA	MASSACHUSETTS	500	HILBERT	
2	NORTHWEST	OR	OREGON	300	NEWTON	
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	
4	SOUTHWEST	AZ	ARIZONA	880	TURING	
4	SOUTHWEST	AZ	ARIZONA	890	MANDELNBROT	
4	SOUTHWEST	NM	NEW MEXICO	780	CHURCH	
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	
5	MIDWEST	-	-	-	-	← R-No-ST

**Syntax & Logic:** Nothing New. The two LEFT OUTER JOIN operations produce a join-result (which is the same as final result in Sample Query 20.2). Then the WHERE-clause excludes any row with an RNAME value of SOUTHEAST. This solution is straightforward. *However, the following sample query, which is similar to this query, is not so straightforward.*



## Intentional Error

For tutorial purposes, the following SELECT statement makes an intentional error. It incorrectly specifies a WHERE-clause to exclude rows from the join-result.

**Sample Query 20.8:** Display the number and name of every region, the name of every state except MASSACHUSETTS, and the name of every customer except those customers who are located in MASSACHUSETTS. Display the columns in the following left-to-right sequence: RNO, RNAME, STNAME and CNAME. Sort the result by CNAME within STNAME within RNO.

### Incorrect Statement & Incorrect Result:

```
SELECT R.RNO, R.RNAME, ST.STNAME, C.CNAME
FROM REGION R
  LEFT OUTER JOIN STATE ST      ON R.RNO = ST.RNO
  LEFT OUTER JOIN CUSTOMER C    ON ST.STCODE = C.STCODE
WHERE ST.STNAME <> 'MASSACHUSETTS' ← Error
ORDER BY R.RNAME, ST.STNAME, C.CNAME
```

RNO	RNAME	STNAME	CNAME	
1	NORTHEAST	CONNECTICUT	-	← ST-No-C
2	NORTHWEST	OREGON	LEIBNIZ	
2	NORTHWEST	OREGON	NEWTON	
2	NORTHWEST	WASHINGTON	DECARTES	
2	NORTHWEST	WASHINGTON	PASCAL	
3	SOUTHEAST	FLORIDA	BOOLE	
3	SOUTHEAST	FLORIDA	CANTOR	
3	SOUTHEAST	GEORGIA	GODEL	
3	SOUTHEAST	GEORGIA	RUSSELL	
4	SOUTHWEST	ARIZONA	MANDELBROT	
4	SOUTHWEST	ARIZONA	TURING	
4	SOUTHWEST	NEW MEXICO	CHURCH	
4	SOUTHWEST	NEW MEXICO	VON NEUMANN	

**Logic:** This statement produces an “almost correct” (wrong) result. The WHERE-clause correctly excludes the MASSACHUSETTS rows along with related customer’s CNAME values from the final result. However, it also *incorrectly* excludes the row for the MIDWEST region.

Notice that, after executing the two LEFT OUTER JOIN operations, the MIDWEST row (denoted by the arrow) appears in the intermediate *join-result* as shown below.

RNO	RNAME	STNAME	CNAME
<b>5</b>	<b>MIDWEST</b>	-	-
1	NORTHEAST	CONNECTICUT	-
1	NORTHEAST	MASSACHUSETTS	BOLYAI
1	NORTHEAST	MASSACHUSETTS	EUCLID
1	NORTHEAST	MASSACHUSETTS	HILBERT
1	NORTHEAST	MASSACHUSETTS	HYPATIA
1	NORTHEAST	MASSACHUSETTS	PYTHAGORAS
1	NORTHEAST	MASSACHUSETTS	ZENO
2	NORTHWEST	OREGON	LEIBNIZ
2	NORTHWEST	OREGON	NEWTON
2	NORTHWEST	WASHINGTON	DECARTES
2	NORTHWEST	WASHINGTON	PASCAL
3	SOUTHEAST	FLORIDA	BOOLE
3	SOUTHEAST	FLORIDA	CANTOR
3	SOUTHEAST	GEORGIA	GODEL
3	SOUTHEAST	GEORGIA	RUSSELL
4	SOUTHWEST	ARIZONA	MANDELBROT
4	SOUTHWEST	ARIZONA	TURING
4	SOUTHWEST	NEW MEXICO	CHURCH
4	SOUTHWEST	NEW MEXICO	VON NEUMANN

Next, the WHERE ST.STNAME <> 'MASSACHUSETTS' clause removes the undesired MASSACHUSETTS rows from the final result. However, this WHERE-clause also incorrectly removes the desired MIDWEST row because it has a null STNAME value. Recall that no WHERE-condition (excluding the IS NULL condition) can "get a hit" on a comparison involving a NULL value.

## ON-Clause: Compound-Condition (AND) Applied *During* Outer-Join

We correct the previous error by recalling the difference between WHERE and AND as described in Chapter 19.

**Sample Query 20.8 (Again):** Display the number and name of every region, the name of every state except MASSACHUSETTS, and the name of every customer except those customers that are located in MASSACHUSETTS. Display the columns in the following left-to-right sequence: RNO, RNAME, STNAME and CNAME. Sort the result by CNAME within STNAME within RNO.

### Correct Statement & Result:

```
SELECT R.RNO, R.RNAME, ST.STNAME, C.CNAME
FROM REGION R
     LEFT OUTER JOIN STATE ST
           ON R.RNO = ST.RNO AND ST.STNAME <> 'MASSACHUSETTS'
     LEFT OUTER JOIN CUSTOMER C
           ON ST.STCODE = C.STCODE
ORDER BY R.RNAME, ST.STNAME, C.CNAME
```

RNO	RNAME	STNAME	CNAME	
5	MIDWEST	-	-	← R-No-ST
1	NORTHEAST	CONNECTICUT	-	← ST-No-C
2	NORTHWEST	OREGON	LEIBNIZ	
2	NORTHWEST	OREGON	NEWTON	
2	NORTHWEST	WASHINGTON	DECARTES	
2	NORTHWEST	WASHINGTON	PASCAL	
3	SOUTHEAST	FLORIDA	BOOLE	
3	SOUTHEAST	FLORIDA	CANTOR	
3	SOUTHEAST	GEORGIA	GODEL	
3	SOUTHEAST	GEORGIA	RUSSELL	
4	SOUTHWEST	ARIZONA	MANDELBROT	
4	SOUTHWEST	ARIZONA	TURING	
4	SOUTHWEST	NEW MEXICO	CHURCH	
4	SOUTHWEST	NEW MEXICO	VON NEUMANN	

**Syntax & Logic:** Nothing new. In the first LEFT OUTER JOIN, the compound join-condition (R.RNO = ST.RNO AND ST.STNAME <> 'MASSACHUSETTS') produces an intermediate result without MASSACHUSETTS rows, but includes the MIDWEST row because REGION is the left-table. The second LEFT OUTER JOIN of this intermediate join-result (as the left-table) with the CUSTOMER table preserves the MIDWEST row in the final result.

## Exercises:

- 20D. Display the number and name of every part, the supplier number of every supplier who can sell the part, and the line-item price for each sale of the part by the supplier. Display the columns in the following left-to-right sequence: PNO, PNAME, SNO, and LIPRICE. Sort the result by SNO within PNO. (Hint: Follow the PART-PARTSUPP-LINEITEM hierarchy.)
- 20E. Display the number and name of every region followed by the minimum and maximal PSPRICE values of parts sold by suppliers in each region. Display the columns in the following left-to-right sequence: RNO, RNAME, and MINPSPRICE and MAXPSPRICE values (column headings for the min and max prices of parts sold by suppliers in each region). Sort the result by RNO. Display zero for null values. (Hint: Follow the REGION-STATE-SUPPLIER-PARTSUPP hierarchy.)
- 20F. Display the number and name of every Western region. (The RNAME value ends with 'WEST'.) Also, display the code and name of every state in each Western region, and the number and name of every supplier in each Western region. Sort the result by SNO within STCODE within RNO. (Hint: Follow the REGION-STATE-SUPPLIER hierarchical path.)
- 20G. Display the number and name of every region, the code, name, and population of every state with a population over 4 million people, and the number and name of every supplier in these states. Sort the result by SNO within STCODE within RNO. (Hint: Follow the REGION-STATE-SUPPLIER hierarchical path.)]

## B. Non-Top-Down Traversal of Table-Hierarchy

In Section-A, every sample query performed a top-down traversal of table-hierarchies found in the MTPCH Database. In this section, sample queries reference the same table-hierarchies, but will not necessarily traverse these hierarchies in a top-down manner. The information presented in this section will be very useful in the following Chapter 20.5.

The following SELECT statements are equivalent. They both display data from all rows in all tables in the REGION-STATE-CUSTOMER hierarchy. Statement-1 was described in Sample Query 20.2. The syntax for Statement-2 is new and will be described below.

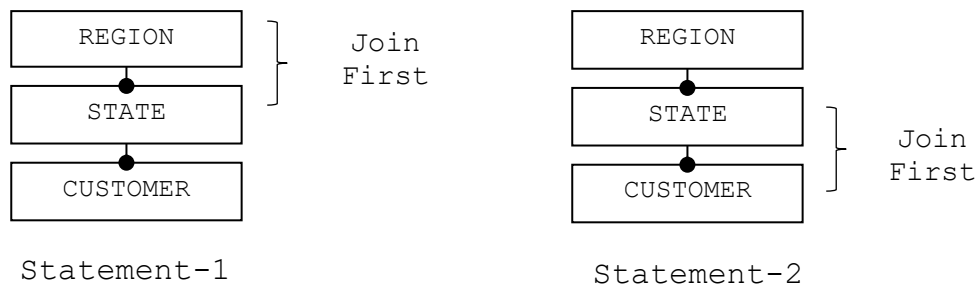
### Statement-1 (Top-Down Traversal)

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,  
       C.CNO, C.CNAME  
FROM REGION R  
       LEFT OUTER JOIN STATE ST ON R.RNO = ST.RNO  
       LEFT OUTER JOIN CUSTOMER C ON ST.STCODE = C.STCODE  
ORDER BY R.RNO, ST.STCODE, C.CNO
```

### Statement-2 (Non-Top-Down Traversal)

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,  
       C.CNO, C.CNAME  
FROM REGION R LEFT OUTER JOIN  
       STATE ST LEFT OUTER JOIN CUSTOMER C ON ST.STCODE = C.STCODE  
       ON R.RNO = ST.RNO  
ORDER BY R.RNO, ST.STCODE, C.CNO
```

Again, Statement-1 directs the system to perform a top-down traversal by initially joining the REGION and STATE tables. However, from a logical perspective, *Statement-2 directs the system to initially join the STATE and CUSTOMER tables*. These different join-sequences are illustrated in the following figure.



## General Syntax

The following page will comment on: (i) how we know that Statement-1 and Statement-2 are equivalent, and (ii) why we might want to code Statement-2 versus Statement-1. But first we address syntax.

**General Syntax for LEFT OUTER JOIN Clause:** We have already seen the basic syntax for the LEFT OUTER JOIN clause:

```
TAB1 LEFT OUTER JOIN TAB2 ON join-condition
```

where TAB1 and TAB2 are table-names.

The **general syntax** for the LEFT OUTER JOIN clause is:

```
TAB-EXP1 LEFT OUTER JOIN TAB-EXP2 ON join-condition
```

where TAB-EXP1 and/or TAB-EXP2 can be a table-name or a *table-expression* that produces an intermediate result table.

The following FROM-clause from Statement-1 conforms to this general syntax where the table-expression is underlined. This table-expression produces an intermediate-result table that becomes the left-table in the join with CUSTOMER.

```
FROM REGION R  
LEFT OUTER JOIN STATE ST ON R.RNO = ST.RNO  
LEFT OUTER JOIN CUSTOMER C ON ST.STCODE = C.STCODE
```

Likewise, the following the FROM-clause from Statement-2 conforms to this general syntax where the table-expression is underlined. This table-expression produces an intermediate-result table that becomes the right-table in the join with REGION.

```
FROM REGION R LEFT OUTER JOIN  
STATE ST LEFT OUTER JOIN CUSTOMER C ON ST.STCODE = C.STCODE  
ON R.RNO = ST.RNO
```

The following page describes why, from a logical perspective, Statement-2 implies that STATE and CUSTOMER are joined first.

Consider the FROM-clause in Statement-2.

```
FROM REGION R LEFT OUTER JOIN
  STATE ST LEFT OUTER JOIN CUSTOMER C ON ST.STCODE = C.STCODE
ON R.RNO = ST.RNO
```

Read this FROM-clause from left-to-right.

```
FROM REGION R LEFT OUTER JOIN "something"
```

The system might expect this "something" to be the name of the Right-table (e.g., STATE) as in Statement-1. However, instead of a table-name, the system encounters the following table-expression.

```
STATE ST LEFT OUTER JOIN CUSTOMER C ON ST.STCODE = C.STCODE
```

**\* The system must evaluate this table-expression first** to produce an intermediate result to serve as the right-table in the join with the REGION table. Hence, the FROM-clause effectively becomes:

```
FROM REGION R LEFT OUTER JOIN
  [Intermediate join-result on STATE and CUSTOMER]
ON R.RNO = ST.RNO
```

Consideration of this FROM-clause motivates two questions.

### Questions

1. How do we know that the FROM-clauses in Statement-1 and Statement-2 (which specify different join-sequences) are equivalent?

Answer: The Associative Law applies to LEFT OUTER JOIN operations. Details are described in Appendix 20A.

2. If we know that the simpler top-down join-sequence in Statement-1 produces the correct result, why would we want to code the alternative join-sequence specified in Statement-2?

Answer: On a rare occasion, an alternative join-sequence could possibly enhance efficiency. Again, see Appendix 20A.

## Pseudo-Code for Join-Sequences

We introduce a pseudo-code for join-sequences that will facilitate our discussion of future examples. Below it will facilitate our discussion of LEFT OUTER JOIN operations that traverse a four-level hierarchy (REGION-STATE-CUSTOMER-PUR\_ORDER).

**Pseudo-Code:** Let LOJ represent a LEFT OUTER JOIN operation, and let IJ represent an INNER JOIN operation. Similar to arithmetic expressions, parentheses are used to designate a join-sequence. For example, the join-sequences for Statement-1 and Statement-2 are shown below.

Statement-1: (REGION LOJ STATE) LOJ CUSTOMER

Statement-2: REGION LOJ (STATE LOJ CUSTOMER)

This pseudo-code does not specify ON-conditions because we assume the join-conditions reference primary-key and foreign-key columns.

Below we present pseudo-code join-sequences for Query 20.3.

**Join-Sequences for Sample Query 20.3:** This sample query referenced the REGION-STATE-CUSTOMER-PUR\_ORDER hierarchy. Here, the number of possible join-sequences increases to 5.

Join-Seq1: ((REGION LOJ STATE) LOJ CUSTOMER) LOJ PUR\_ORDER

Join-Seq2: REGION LOJ (STATE LOJ (CUSTOMER LOJ PUR\_ORDER))

Join-Seq3: REGION LOJ ((STATE LOJ CUSTOMER) LOJ PUR\_ORDER)

Join-Seq4: (REGION LOJ (STATE LOJ CUSTOMER)) LOJ PUR\_ORDER

Join-Seq5: (REGION LOJ STATE) LOJ (CUSTOMER LOJ PUR\_ORDER)

Join-Seq1 represents the top-down join-sequence specified in Sample Query 20.3. Join-Seq2 specifies a bottom-up join-sequence. Join-Seq3 and Join-Seq4 represent a "middle-join-first" sequence where two "middle tables" (STATE and CUSTOMER), are joined first. Join-Seq5 initially joins the top two tables (REGION and STATE), and then joins the bottom two tables (CUSTOMER and PUR\_ORDER), and finally joins the two intermediate join-results.



## Alternative Join-Sequences

Coding join-sequences for Join-Seq2, Join-Seq3, Join-Seq4, and Join-Seq5 can be a little tricky. Below we preview two examples. The following pages offer more advice on details.

### Example: Code Join-Seq2:

```
REGION LOJ (STATE LOJ (CUSTOMER LOJ PUR_ORDER))
```

Here, you want to initially join the CUSTOMER and PUR\_ORDER tables to produce an intermediate join-result. Then, this intermediate join-result will serve as the right-table in a left outer-join with STATE to produce a second intermediate join-result. Finally, this second intermediate join-result will serve as the right-table in a left outer-join with REGION.

The FROM-clause that specifies this join-sequence is shown below in Figure 20.1.

#### Sample Query 20.3: Join-Seq2

```
FROM REGION R LEFT OUTER JOIN
      STATE ST LEFT OUTER JOIN
            CUSTOMER C LEFT OUTER JOIN PUR_ORDER PO
                  ON C.CNO = PO.CNO
            ON ST.STCODE = C.STCODE
      ON R.RNO = ST.RNO
```

Figure 20.1: REGION LOJ (STATE LOJ (CUSTOMER LOJ PUR\_ORDER))

**Example: Code Join-Seq4:**

```
(REGION LOJ (STATE LOJ CUSTOMER)) LOJ PUR_ORDER
```

Sample Query 20.3: Join-Seq4

```
FROM REGION R LEFT OUTER JOIN
      STATE ST LEFT OUTER JOIN CUSTOMER C
      ON ST.STCODE = C.STCODE
ON R.RNO = ST.RNO
LEFT OUTER JOIN PUR_ORDER PO
ON C.CNO = PO.CNO
```

Figure 20.2: (REGION LOJ (STATE LOJ CUSTOMER)) LOJ PUR\_ORDER

**Preview Exercise:** Forthcoming Exercises 28I and 20J will ask you to code FROM-clauses for the following two pseudo-code expressions.

```
Join-Seq3: REGION LOJ ((STATE LOJ CUSTOMER) LOJ PUR_ORDER)
```

```
Join-Seq5: (REGION LOJ STATE) LOJ (CUSTOMER LOJ PUR_ORDER)
```

The following pages will help you solve these exercises.

## ON-Conditions → Join-Sequence

You can work backwards, starting with a FROM-clause (written by you or some other user) to deduce the join-sequence specified by the FROM-clause. Then you can transform this FROM-clause into pseudo-code. Below, we present a simple method to achieve this objective. Consider the following FROM-clause, and focus on the three ON-conditions.

```
FROM REGION R LEFT OUTER JOIN STATE ST
      LEFT OUTER JOIN CUSTOMER C
            ON ST.STCODE = C.STCODE           ← 1
      LEFT OUTER JOIN PUR_ORDER PO
            ON C.CNO = PO.CNO                 ← 2
      ON R.RNO = ST.RNO                       ← 3
```

List these ON-conditions in the same top-down sequence as they were specified in the FROM-clause, as shown below.

1. ON ST.STCODE = C.STCODE
2. ON C.CNO = PO.CNO
3. ON R.RNO = ST.RNO

**\*\*\* The sequence of ON-conditions dictates the join-sequence.**

Notice that table-aliases in an ON-condition identify the tables that are being joined. For example, consider the first ON-condition: ON ST.STCODE = C.STCODE

After substituting table-names for the aliases, we have:

```
ON STATE.STCODE = CUSTOMER.STCODE
```

Hence, this first ON-condition implies that the STATE and CUSTOMER tables are to be joined first. The second ON-condition implies that the CUSTOMER and PUR\_ORDER tables are to be joined second. The third ON-condition implies that the REGION and STATE tables are joined last. To summarize, we deduce that the join-sequence specified by this FROM-clause is:

<u>ON-Condition</u>	<u>Join-Sequence</u>
1. ON ST.STCODE = C.STCODE	→ 1 <sup>st</sup> Join: STATE and CUSTOMER
2. ON C.CNO = PO.CNO	→ 2 <sup>nd</sup> Join: CUSTOMER and PUR_ORDER
3. ON R.RNO = ST.RNO	→ 3 <sup>rd</sup> Join: REGION and STATE

If desired, we can transform this join-sequence into a pseudo-code expression.

First, list the referenced tables in hierarchical sequence.

```
REGION - STATE - CUSTOMER - PUR_ORDER
```

Next, examine the FROM-clause to include the specific join-operations, IJ or LOJ. (In this chapter, all examples only specify LOJ operations.)

```
REGION LOJ STATE LOJ CUSTOMER LOJ PUR_ORDER
```

Then, examine the sequence of ON-conditions (shown on the previous page) to assign the execution sequence to each LOJ operation.

```
REGION LOJ STATE LOJ CUSTOMER LOJ PUR_ORDER
      3          1          2
```

Finally, specify parentheses to represent this join-sequence.

```
REGION LOJ ((STATE LOJ CUSTOMER) LOJ PUR_ORDER)
```

**Code Verification Technique:** After you have coded a FROM-clause, you can validate the join-sequence specified by this FROM-clause by using the above method to generate the equivalent pseudo-code. This pseudo-code should correspond to your query objective.

**Exercise:**

20H. Work backwards. Describe the join-sequence for the following FROM-clause that references five tables. Then transform this sequence into pseudo-code.

```
FROM REGION R
      LEFT OUTER JOIN STATE ST
            LEFT OUTER JOIN CUSTOMER C
                  LEFT OUTER JOIN PUR_ORDER PO
                        LEFT OUTER JOIN LINEITEM LI
                              ON PO.PONO = LI.PONO
                                      ON C.CNO = PO.CNO
                                              ON ST.STCODE = C.STCODE
                                                  ON R.RNO = ST.RNO
```

## Optional Parentheses in FROM-Clauses

Our pseudo-code utilizes parentheses to designate the sequence of join-operations. This is similar to utilizing parentheses to designate the execution sequence of arithmetic operations within an arithmetic expression.

Parentheses can also be specified within a FROM-clause. **However**, unlike parentheses in our pseudo-code, *parentheses within a FROM-clause do **not** dictate the join-sequence.* Parentheses can only enhance readability.

*Important: Each pair of parentheses must enclose a complete JOIN-ON clause as illustrated in the following example.*

**Example:** Assume you examine the FROM-clause in a SQL statement (coded by some other user) and observe parentheses in the FROM-clause, as illustrated below.

```
FROM REGION R
  LEFT OUTER JOIN
    (STATE ST LEFT OUTER JOIN CUSTOMER C
      ON ST.STCODE = C.STCODE)
  ON R.RNO = ST.RNO
```

This FROM-clause is equivalent to the following FROM-clause which *simply removes the parentheses.*

```
FROM REGION R
  LEFT OUTER JOIN
    STATE ST LEFT OUTER JOIN CUSTOMER C
      ON ST.STCODE = C.STCODE
  ON R.RNO = ST.RNO
```

Author Comment: Although parentheses enhance readability, for tutorial reasons, this author did not specify parentheses in preceding FROM-clauses. Three observations justify this decision.

1. Within arithmetic expressions, there is a default sequence of arithmetic operations. For example, multiplication is executed before addition. Hence:  $2+3*4 = 14$ . However, **there is no default sequence among join-operations.** For example, the system will not automatically execute an INNER JOIN before or after a LEFT OUTER JOIN. (The following chapter will demonstrate this fact.)

2. Within an arithmetic expression, parentheses can be used to override the default sequence of arithmetic operations. For example,  $(2+3)*4 = 20$ . However, as stated above, **there is no default sequence among join-operations that can be overridden.** (In Sample Queries 20.1-20.8, we coded top-down join-sequences as our own "preferred default" join-sequence.)
3. We did not specify parentheses in the previous FROM-clauses because our intention was to avoid potential confusion associated with the *incorrect* assumption that parentheses dictate the sequence of join-operations.

**Example:** To enhance readability, we include parentheses in the FROM-clause shown below Figure 20.3a. These parentheses correspond to the parentheses specified in the following pseudo-code for this FROM-clause.

```
REGION LOJ (STATE LOJ (CUSTOMER LOJ (PUR_ORDER LOJ LINEITEM)))
```

```
FROM REGION R
  LEFT OUTER JOIN (STATE ST
    LEFT OUTER JOIN (CUSTOMER C
      LEFT OUTER JOIN (PUR_ORDER PO
        LEFT OUTER JOIN LINEITEM LI
          ON PO.PONO = LI.PONO)
        ON C.CNO = PO.CNO)
      ON ST.STCODE = C.STCODE)
    ON R.RNO = ST.RNO
```

Figure 20.3a: FROM-Clause with Parentheses

Finally, the following Figure 20.3b optionally rewrites the above FROM-clause such that each left-parentheses (indicating the beginning of a new JOIN-ON clause) starts on a new line.

```
FROM REGION R LEFT OUTER JOIN
  (STATE ST LEFT OUTER JOIN
    (CUSTOMER C LEFT OUTER JOIN
      (PUR_ORDER PO LEFT OUTER JOIN LINEITEM LI
        ON PO.PONO = LI.PONO)
      ON C.CNO = PO.CNO)
    ON ST.STCODE = C.STCODE)
  ON R.RNO = ST.RNO
```

Figure 20.3b: Optional - Parentheses start on new line

## Coding Non-Top-Down FROM-Clauses

Before coding a FROM-clause, it can be helpful to formulate a pseudo-code expression that represents your desired join-sequence (which was derived by analyzing your query objective). Then you can transform this pseudo-code into an equivalent FROM-clause.

The following examples illustrate a semi-cookbook method to transform a pseudo-code expression into a FROM-clause. (This method may be "overkill" for users with good intuition.)

**Example-1:** Transform the following pseudo-code into a FROM-clause.

```
REGION LOJ (STATE LOJ (CUSTOMER LOJ PUR_ORDER))
```

Begin by assigning sequence-numbers to each join-operation. For example, the above pseudo-code indicates that the first join operation joins the CUSTOMER and PUR\_ORDER tables. Etc.

```
REGION LOJ (STATE LOJ (CUSTOMER LOJ PUR_ORDER))
           3rd         2nd         1st
```

Next, list ON-conditions in a top-down manner corresponding to the above sequence-numbers. Here, the ON-conditions specify table-aliases and the primary-key and foreign-key columns of the tables to be joined.

```
ON C.CNO = PO.CNO
ON ST.STCODE = C.STCODE
ON R.RNO = ST.RNO
```

Then code a complete LEFT OUTER JOIN clause for each ON-condition.

1. Start with the first ON-condition: ON C.CNO = PO.CNO

```
CUSTOMER C LEFT OUTER JOIN PUR_ORDER PO
ON C.CNO = PO.CNO
```

We (optionally) enclose this complete JOIN-ON clause within parentheses.

```
(CUSTOMER C LEFT OUTER JOIN PUR_ORDER PO
ON C.CNO = PO.CNO)
```

2. Include the second ON-condition: ON ST.STCODE = C.STCODE

The pseudo-code indicates that the preceding JOIN-ON clause becomes the right-table in this second join-operation.

```
STATE ST LEFT OUTER JOIN
  (CUSTOMER C LEFT OUTER JOIN PUR_ORDER PO
   ON C.CNO = PO.CNO)
ON ST.STCODE = C.STCODE
```

Again, we optionally enclose this expanded JOIN-ON clause within parentheses.

```
(STATE ST LEFT OUTER JOIN
  (CUSTOMER C LEFT OUTER JOIN PUR_ORDER PO
   ON C.CNO = PO.CNO)
 ON ST.STCODE = C.STCODE)
```

3. Include the third ON-condition: ON R.RNO = ST.RNO

The pseudo-code indicates that the preceding JOIN-ON clause becomes the right-table in this third join-operation.

```
REGION R LEFT OUTER JOIN
  (STATE ST LEFT OUTER JOIN
   (CUSTOMER C LEFT OUTER JOIN PUR_ORDER PO
    ON C.CNO = PO.CNO)
   ON ST.STCODE = C.STCODE)
ON R.RNO = ST.RNO
```

Because this is the last JOIN-ON clause, we choose not to enclose it within the parentheses.

Finally, specify the FROM keyword before the above JOIN-ON clause.

```
FROM REGION R LEFT OUTER JOIN
  (STATE ST LEFT OUTER JOIN
   (CUSTOMER C LEFT OUTER JOIN PUR_ORDER PO
    ON C.CNO = PO.CNO)
   ON ST.STCODE = C.STCODE)
ON R.RNO = ST.RNO
```

[This FROM-clause was illustrated in Figure 20.1.]



**Example-2:** Transform the following pseudo-code into a FROM-clause.

```
(REGION LOJ (STATE LOJ CUSTOMER)) LOJ PUR_ORDER
```

Assign sequence-numbers to each join-operation.

```
(REGION LOJ (STATE LOJ CUSTOMER)) LOJ PUR_ORDER
      2nd          1st          3rd
```

List ON-conditions in a top-down manner corresponding to the above sequence-numbers

```
ON ST.STCODE = C.STCODE
ON R.RNO = ST.RNO
ON C.CNO = PO.CNO
```

Code a complete LEFT OUTER JOIN clause for first ON-condition.

```
(STATE ST LEFT OUTER JOIN CUSTOMER C
  ON ST.STCODE = C.STCODE)
```

Code a complete LEFT OUTER JOIN clause for the second ON-condition. The pseudo-code indicates that the preceding JOIN-ON clause becomes the right-table in this second join-operation.

```
(REGION R LEFT OUTER JOIN
  (STATE ST LEFT OUTER JOIN CUSTOMER C
    ON ST.STCODE = C.STCODE)
  ON R.RNO = ST.RNO)
```

Code a complete LEFT OUTER JOIN clause for the third ON-condition. The pseudo-code indicates that the preceding JOIN-ON clause becomes the left-table in this third join-operation. Also specify the FROM keyword.

```
FROM
  (REGION R LEFT OUTER JOIN
    (STATE ST LEFT OUTER JOIN CUSTOMER C
      ON ST.STCODE = C.STCODE)
    ON R.RNO = ST.RNO)
  LEFT OUTER JOIN PUR_ORDER PO
  ON C.CNO = PO.CNO
```

[This FROM-clause was illustrated in Figure 20.2.]

## Exercises:

20Ia Transform the following pseudo-code into a FROM-clause.

```
REGION LOJ ((STATE LOJ CUSTOMER) LOJ PUR_ORDER)
```

20Ib. Transform the following pseudo-code into a FROM-clause.

```
(REGION LOJ STATE) LOJ (CUSTOMER LOJ PUR_ORDER)
```

## Summary

It is not difficult, as illustrated by Sample Queries 20.1 - 20.4, to code multiple LEFT OUTER JOIN operations that join tables along a hierarchy in a top-down sequence. Sample Queries 20.5 - 20.8, which also traversed the hierarchy in a top-down sequence, were more complex because they involved exiting-the-hierarchy, restrictions, grouping, and summarizing.

Section-B illustrated that *non-top-down join-sequences produce the same result if all join-operations are LEFT OUTER JOINS*. The following Appendix 20A will describe a *potential* efficiency advantage associated with a non-top-down join-sequences. However, significant advantages of non-top-down join-sequences will be illustrated in the following Chapter 20.5.

## Summary Exercise

20J. Optional Exercise: Modify the SELECT statement for Sample Query 20.5. Change the FROM-clause. Specify an INNER JOIN operation to join the LINEITEM and PARTSUPP tables as shown below.

```
SELECT R.RNO, R.RNAME, ST.STCODE, C.CNO,
       PO.PONO, LI.PNO, LI.SNO, LI.LIPRICE, PS.PSPRICE
FROM REGION R
     LEFT OUTER JOIN STATE ST      ON R.RNO = ST.RNO
     LEFT OUTER JOIN CUSTOMER C    ON ST.STCODE = C.STCODE
     LEFT OUTER JOIN PUR_ORDER PO  ON C.CNO = PO.CNO
     LEFT OUTER JOIN LINEITEM LI   ON PO.PONO = LI.PONO
     INNER JOIN PARTSUPP PS        ON LI.PNO = PS.PNO
                                     AND LI.SNO = PS.SNO
ORDER BY R.RNO, ST.STCODE, C.CNO, PO.PONO, LI.PNO
```

Execute this statement and examine the result. All LEFT OUTER JOIN operations appear to behave like INNER JOIN operations. Why did this happen?

## Appendix 20A: Theory & Efficiency

**Review:** In Appendix 18A we discussed efficiency considerations pertaining to join-sequences for multiple INNER JOIN operations. And, in Appendix 18B we noted that the **Associative Law** applies to the INNER JOIN.

(TAB1 IJ TAB2) IJ TAB3 = TAB1 IJ (TAB2 IJ TAB3)

From an efficiency perspective, this law allows the optimizer to initially join TAB1 and TAB2, or to initially join TAB2 and TAB3. Most likely, one of these join operations would produce a smaller intermediate join result. For example, assume (TAB2 IJ TAB3) produced a join-result that was smaller than the (TAB1 IJ TAB2) join-result. Then the optimizer would be inclined to implement:

TAB1 IJ (TAB2 IJ TAB3)

**Theory:** *The Associative Law also applies to LEFT OUTER JOIN operations.*

(TAB1 LOJ TAB2) LOJ TAB3 = TAB1 LOJ (TAB2 LOJ TAB3)

As with INNER JOINS, this law can be extended to any number of LEFT OUTER JOINS. For example, Sample Query 20.3 specified:

```
FROM REGION R
LEFT OUTER JOIN STATE ST      ON R.RNO = ST.RNO
LEFT OUTER JOIN CUSTOMER C    ON ST.STCODE = C.STCODE
LEFT OUTER JOIN PUR_ORDER PO  ON C.CNO = PO.CNO
```

The pseudo-code for this FROM-clause is:

((REGION LOJ STATE) LOJ CUSTOMER) LOJ PUR\_ORDER

By repeated application of the Associative Law, we can deduce four other *equivalent* join-sequences.

(REGION LOJ (STATE LOJ CUSTOMER)) LOJ PUR\_ORDER

REGION LOJ ((STATE LOJ CUSTOMER) LOJ PUR\_ORDER)

REGION LOJ ((STATE LOJ (CUSTOMER LOJ PUR\_ORDER))

(REGION LOJ STATE) LOJ (CUSTOMER LOJ PUR\_ORDER)

**Efficiency:** Appendix 18A discussed efficiency considerations pertaining to different join-sequences within the context of multiple INNER JOIN operations. The same concepts apply to multiple LEFT OUTER JOIN operations *because both inner-joins and outer-joins obey the Associative Law.*

If a FROM-clause specifies multiple LEFT OUTER JOIN operations, a general efficiency guideline is to initially join the two tables that will produce the smallest intermediate join-result; after this join is completed, join the next two tables (including intermediate result tables) that produce the next smallest intermediate result; Etc.

For example, consider the following top-down join-sequence.

```
((REGION LOJ STATE) LOJ CUSTOMER) LOJ PUR_ORDER
```

This join-sequence initially joins the REGION and STATE tables which happen to be the two smallest tables. Then this intermediate join-result table is joined with CUSTOMER which happens to be the third smallest table. Finally, this intermediate join-result is joined with the PUR\_ORDER table which happens to be the largest of the four tables. This fortunate turn of events occurred because (i) the join-sequence followed the hierarchical path in a top-down manner, and (ii) it is usually the case a parent-table contains fewer rows than its child-table.

A top-down join-sequence may not be the most efficient sequence in two circumstances. First, sometimes a parent-table has many more rows than a child-table. Second, an ON-Clause may specify an AND-condition that significantly reduces the size of a join-result. [For tutorial reasons, this chapter's sample queries did not specify any AND-conditions in ON-clauses.] These circumstances would require the optimizer to "do its thing" to determine the optimal join-sequence.

What if the optimizer fails to generate an optimal join-sequence? Then you could try coding your desired join-sequence in the FROM-clause. This could *possibly* influence the optimizer produce a more efficient join-sequence.

# 20.5

## **“Mixing” Inner-Joins and Outer-Joins**

Each SELECT statement presented in this chapter specifies one or more INNER JOIN operations along with one or more LEFT OUTER JOIN operations. Again, all referenced tables lie along a hierarchical path in the MTPCH database.

This is an optional chapter because all of its sample queries and exercises can be satisfied by coding one or more Common Table Expressions (CTEs), a topic that will be introduced in Chapter 27. Because the “mixing” of INNER JOINS and OUTER JOINS can become a little tricky, many users will prefer to code FROM-clauses that reference CTEs.

Before reading this chapter, you should have read Section-B in the preceding Chapter 20.

## Mixing Inner-Joins & Outer-Joins

Each of this chapter's sample queries satisfy its query objective with a SELECT statement that specifies INNER JOIN and LEFT OUTER JOIN operations.

This chapter will use the pseudo-code notation that was introduced in the previous chapter. For example, the query objective for the forthcoming Sample Query 20.10 requires an INNER JOIN to be executed before a LEFT OUTER JOIN. Pseudo-code for this join-sequence is:

```
REGION LOJ (STATE IJ CUSTOMER)
```

**Unfair Preview Question:** Will the above pseudo-code produce the same result as the following pseudo-code which joins the same tables with the same join-operations, but changes the sequence of the join-operations?

```
(REGION LOJ STATE) IJ CUSTOMER
```

**Preview Answer:** No. Observe the different results for the following Sample Queries 20.10 and 20.11a.

**Possible Join Sequences:** Consider the four possible pseudo-code expressions for joining tables that along the REGION-STATE-CUSTOMER hierarchy. Assume that one join-operation is a LEFT OUTER JOIN (LOJ) and the other is an INNER JOIN (IJ). This pseudo-code references table aliases.

Expression-1: (R IJ ST) LOJ C	}	IJ executed before LOJ
Expression-2: R LOJ (ST IJ C)		
Expression-3: (R LOJ ST) IJ C	}	LOJ executed before IJ
Expression-4: R IJ (ST LOJ C)		

Expression-1: (R IJ ST) LOJ C

This expression represents a top-down join-sequence where the INNER JOIN operation is executed first. The following FROM-clause, which will be specified in Sample Query 20.9, codes this join-sequence.

```
FROM REGION R
  INNER JOIN STATE ST          ON R.RNO = ST.RNO
  LEFT OUTER JOIN CUSTOMER C    ON ST.STCODE = C.STCODE
```

Expression-2: R LOJ (ST IJ C)

This expression, like Expression-1, indicates that the INNER JOIN operation is executed first. The following FROM-clause, which will be specified in Sample Query 20.10, codes this join-sequence.

```
FROM REGION R
  LEFT OUTER JOIN
    STATE ST INNER JOIN CUSTOMER C ON ST.STCODE = C.STCODE
  ON R.RNO = ST.RNO
```

Expression-3: (R LOJ ST) IJ C

This expression represents a top-down join-sequence where the LEFT OUTER JOIN operation is executed first. The following FROM-clause, which will be specified in Sample Query 20.11a, codes this join-sequence.

```
FROM REGION R
  LEFT OUTER JOIN STATE ST ON R.RNO = ST.RNO
  INNER JOIN CUSTOMER C     ON ST.STCODE = C.STCODE
```

Expression-4: R IJ (ST LOJ C)

This expression, like Expression-3, indicates that the LEFT OUTER JOIN operation is executed first. The following FROM-clause, which will be specified in Sample Query 20.11b, codes this join-sequence.

```
FROM REGION R
  INNER JOIN STATE ST
  LEFT OUTER JOIN CUSTOMER C ON ST.STCODE = C.STCODE
  ON R.RNO = ST.RNO
```



## Expression-1: (R IJ ST) LOJ C

In the following sample query, the top-down join-sequence conforms to the query objective.

**Sample Query 20.9:** Display the number and name of every region that contains at least one state, the code and name of every state (including states without customers), and the number and name of every customer in each state. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, CNO and CNAME. Sort the result by CNO within STCODE within RNO.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME

FROM   REGION R
       INNER JOIN STATE ST           ON R.RNO = ST.RNO
       LEFT OUTER JOIN CUSTOMER C ON ST.STCODE = C.STCODE

ORDER BY R.RNO, ST.STCODE, C.CNO
```

<u>RNO</u>	<u>RNAME</u>	<u>STCODE</u>	<u>STNAME</u>	<u>CNO</u>	<u>CNAME</u>	
1	NORTHEAST	CT	CONNECTICUT	-	-	← ST-No-C
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	
1	NORTHEAST	MA	MASSACHUSETTS	230	BOLYAI	
1	NORTHEAST	MA	MASSACHUSETTS	500	HILBERT	
2	NORTHWEST	OR	OREGON	300	NEWTON	
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	
3	SOUTHEAST	FL	FLORIDA	600	BOOLE	
3	SOUTHEAST	FL	FLORIDA	660	CANTOR	
3	SOUTHEAST	GE	GEORGIA	700	RUSSELL	
3	SOUTHEAST	GE	GEORGIA	770	GODEL	
4	SOUTHWEST	AZ	ARIZONA	880	TURING	
4	SOUTHWEST	AZ	ARIZONA	890	MANDELROT	
4	SOUTHWEST	NM	NEW MEXICO	780	CHURCH	
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	

**Syntax & Logic:** This query objective wants to display just the matching rows in the REGION table. Specifying an INNER JOIN operation satisfies this objective. This inner-join produces the following intermediate join-result. (Notice that the non-matching MIDWEST region does not appear in this result.)

<u>RNO</u>	<u>RNAME</u>	<u>STCODE</u>	<u>STNAME</u>
1	NORTHEAST	CT	CONNECTICUT
1	NORTHEAST	MA	MASSACHUSETTS
2	NORTHWEST	OR	OREGON
2	NORTHWEST	WA	WASHINGTON
3	SOUTHEAST	FL	FLORIDA
3	SOUTHEAST	GE	GEORGIA
4	SOUTHWEST	AZ	ARIZONA
4	SOUTHWEST	NM	NEW MEXICO

Next, this intermediate join-result serves as the left-table in the LEFT OUTER JOIN with the CUSTOMER table. This join-operation preserves all of the above rows, including rows for states without customers. (Notice that CONNECTICUT, the only state without any customers, appears in the final result.)

**Equivalent FROM-clause:** Sample Query 20.11b will present an alternative (non-top-down) FROM-clause for this query objective. Most users will prefer the current FROM-clause because it specifies a top-down join-sequence.

**Exercise:**

20K. Display the number and name of every region that contains at least one state, the code and name of every state (including states without any suppliers), and the number and name of every supplier in each state. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, SNO, and SNAME. Sort the result by SNO within STCODE within RNO. (Hint: Follow the REGION-STATE-SUPPLIER hierarchy.)

## Expression-2: R LOJ (ST IJ C)

The following FROM-clause designates a join-sequence that does not conform to a top-down join-sequence. Here, the INNER JOIN operation is executed first.

**Sample Query 20.10:** Display the number and name of every region, the code and name of every state with at least one customer, and the number and name of every customer in these states. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, CNO, and CNAME. Sort the result by CNO within STCODE within RNO.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME

FROM REGION R
  LEFT OUTER JOIN
    STATE ST INNER JOIN CUSTOMER C ON ST.STCODE = C.STCODE
  ON R.RNO = ST.RNO

ORDER BY R.RNO, ST.STCODE, C.CNO
```

RNO	RNAME	STCODE	STNAME	CNO	CNAME	
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	
1	NORTHEAST	MA	MASSACHUSETTS	230	BOLYAI	
1	NORTHEAST	MA	MASSACHUSETTS	500	HILBERT	
2	NORTHWEST	OR	OREGON	300	NEWTON	
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	
3	SOUTHEAST	FL	FLORIDA	600	BOOLE	
3	SOUTHEAST	FL	FLORIDA	660	CANTOR	
3	SOUTHEAST	GE	GEORGIA	700	RUSSELL	
3	SOUTHEAST	GE	GEORGIA	770	GODEL	
4	SOUTHWEST	AZ	ARIZONA	880	TURING	
4	SOUTHWEST	AZ	ARIZONA	890	MANDELBROT	
4	SOUTHWEST	NM	NEW MEXICO	780	CHURCH	
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	
5	MIDWEST	-	-	-	-	← R-No-ST

**Syntax & Logic:** This query objective wants to display data about states with customers. Specifying an inner-join of STATE and CUSTOMER satisfies this objective. This inner-join produces the following intermediate join-result. (Notice that the non-matching CONNECTICT row does not appear in this result.)

STCODE	STNAME	RNO	CNO	CNAME
AZ	ARIZONA	4	880	TURING
AZ	ARIZONA	4	890	MANDELBROT
FL	FLORIDA	3	600	BOOLE
FL	FLORIDA	3	660	CANTOR
GE	GEORGIA	3	700	RUSSELL
GE	GEORGIA	3	770	GODEL
MA	MASSACHUSETTS	1	100	PYTHAGORAS
MA	MASSACHUSETTS	1	110	EUCLID
MA	MASSACHUSETTS	1	200	HYPATIA
MA	MASSACHUSETTS	1	220	ZENO
MA	MASSACHUSETTS	1	230	BOLYAI
MA	MASSACHUSETTS	1	500	HILBERT
NM	NEW MEXICO	4	780	CHURCH
NM	NEW MEXICO	4	800	VON NEUMANN
OR	OREGON	2	300	NEWTON
OR	OREGON	2	330	LEIBNIZ
WA	WASHINGTON	2	400	DECARTES
WA	WASHINGTON	2	440	PASCAL

Next, this intermediate join-result serves as the right-table in the LEFT OUTER JOIN with the REGION table. This LEFT OUTER JOIN preserves all of the above rows (because all RNO values match). It also includes information about regions without states. Notice that MIDWEST, the only region without any states, appears in the final result.

**Exercise:**

20L. Display the number and name of every region, the code and name of those states that have at least one supplier, and the number and name of these suppliers. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, SNO, and SNAME. Sort the result by SNO within STCODE within RNO. (Hint: Follow the REGION-STATE-SUPPLIER hierarchy.)

## “Questionable” Expression-3: (R LOJ ST) IJ C

**Sample Query 20.11a:** Same query objective as Sample Query 18.3 that coded two INNER JOIN operations [Access the REGION, STATE, and CUSTOMER tables. For all customers, display their CNO and CNAME values, along with their state’s STCODE and STNAME values, along with their region’s RNO and RNAME values. Display these columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, CNO and CNAME. Sort the result by CNO within STCODE within RNO.]

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME

FROM REGION R
     LEFT OUTER JOIN STATE ST ON R.RNO = ST.RNO
     INNER JOIN CUSTOMER C   ON ST.STCODE = C.STCODE

ORDER BY R.RNO, ST.STCODE, C.CNO
```

**Result Table:** Same result shown for Sample Query 18.3.

**Logic:** The final result table does not contain data about the MIDWEST region (the only region without any states), and it does not contain data about CONNECTICUT (the only state without any customers). Casually speaking, the INNER JOIN “effectively undoes part of the previous LEFT OUTER JOIN” by removing non-matching rows such that the final result corresponds to two INNER JOIN operations.

Specifically, the initial LEFT OUTER JOIN operation contains data about the MIDWEST region (because REGION is the left-table), and data about CONNECTICUT (because its foreign-key value must match some primary-key in the REGION table). However, the subsequent INNER JOIN removes the MIDWEST and CONNECTICUT rows because:

- the intermediate result row for the MIDWEST region will have a null ST.STCODE value which cannot match any C.STCODE, and
- the intermediate result row for CONNECTICUT does not match on any CUSTOMER row because Connecticut does not have any customers.

**Conclusion:** The above SELECT statement is “questionable” because it is simpler to code two INNER JOIN operations as shown in Sample Query 18.3.

## “Questionable” Expression-4: R IJ (ST LOJ C)

**Sample Query 20.11b:** Same query objective as Sample Query 20.9. [Display the number and name of every region that contains at least one state, the code and name of every state (including states without customers), and the number and name of every customer in each state. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, CNO and CNAME. Sort the result by CNO within STCODE within RNO.] Note that Sample Query 20.9. coded a top-down join-sequence.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME

FROM REGION R
     INNER JOIN STATE ST
           LEFT OUTER JOIN CUSTOMER C ON ST.STCODE = C.STCODE
           ON R.RNO = ST.RNO

ORDER BY R.RNO, ST.STCODE, C.CNO
```

**Result Table:** Same result shown for Sample Query 20.9.

**Logic:** The initial LEFT OUTER JOIN preserves information about CONNECTICUT (the state without customers). The subsequent INNER JOIN removes information about the MIDWEST Region.

The equivalent FROM-clause for Sample Query 20.9 is:

```
FROM REGION R
     INNER JOIN STATE ST           ON R.RNO = ST.RNO
     LEFT OUTER JOIN CUSTOMER C   ON ST.STCODE = C.STCODE
```

This FROM-clause may be simpler because it follows a top-down join-sequence.

**“Questionable” Expressions:** Expression-3 and Expression-4 were designated as “questionable” because each expression can be rewritten in a simpler form.

Also, note that both of these FROM-clauses executed the INNER JOIN after the LEFT OUTER, and the INNER JOIN removed part of the outer-join result. *This is not necessarily wrong.* It depends upon your query objective. However, you should be attentive when you execute an INNER JOIN after a LEFT OUTER JOIN.

The following sample query traverses the REGION-STATE-CUSTOMER-PUR\_ORDER hierarchy. The query objective conforms to a top-down join-sequence.

**Sample Query 20.12:** Display the number and name of every region with at least one state, the code and name of every state with at least one customer, the number and name of every customer in these states (including customers without purchase orders), and the date of every purchase-order completed by these customers. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, CNO, CNAME, and PODATE. Sort the result by PODATE within CNO within STCODE within RNO.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME, PO.PODATE

FROM REGION R
      INNER JOIN STATE ST          ON R.RNO = ST.RNO
      INNER JOIN CUSTOMER C       ON ST.STCODE = C.STCODE
      LEFT OUTER JOIN PUR_ORDER PO ON C.CNO = PO.CNO

ORDER BY R.RNO, ST.STCODE, C.CNO, PO.PODATE
```

[Result table shown on following page.]

**Logic:** This query objective implies the following top-down join-sequence.

```
((REGION IJ STATE) IJ CUSTOMER) LOJ PUR_ORDER
```

This FROM-clause could be coded with parentheses as shown below.

```
FROM ((REGION R
      INNER JOIN STATE ST  ON R.RNO = ST.RNO)
      INNER JOIN CUSTOMER C ON ST.STCODE = C.STCODE)
      LEFT OUTER JOIN PUR_ORDER PO ON C.CNO = PO.CNO
```

[Recall that each pair of parentheses must enclose a complete JOIN-ON clause that references two table-names or table expressions.]

Executing the first two INNER JOIN operations eliminates the non-matching regions (MIDWEST) without states and non-matching states (CONNECTICUT) without customers. Then the subsequent LEFT OUTER JOIN preserves all customers, including those customers (MANDELBROT and CHURCH) without purchase orders.

RNO	RNAME	STCODE	STNAME	CNO	CNAME	PODATE
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	1
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	3
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	47
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	49
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	20
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	21
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	5
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	22
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	23
1	NORTHEAST	MA	MASSACHUSETTS	230	BOLYAI	6
1	NORTHEAST	MA	MASSACHUSETTS	500	HILBERT	72
2	NORTHWEST	OR	OREGON	300	NEWTON	7
2	NORTHWEST	OR	OREGON	300	NEWTON	8
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	9
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	61
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	62
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	63
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	64
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	65
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	71
3	SOUTHEAST	FL	FLORIDA	600	BOOLE	73
3	SOUTHEAST	FL	FLORIDA	600	BOOLE	74
3	SOUTHEAST	FL	FLORIDA	660	CANTOR	1
3	SOUTHEAST	FL	FLORIDA	660	CANTOR	75
3	SOUTHEAST	GE	GEORGIA	700	RUSSELL	1
3	SOUTHEAST	GE	GEORGIA	770	GODEL	3
4	SOUTHWEST	AZ	ARIZONA	880	TURING	3
4	SOUTHWEST	AZ	ARIZONA	880	TURING	4
4	SOUTHWEST	AZ	ARIZONA	880	TURING	10
4	SOUTHWEST	AZ	ARIZONA	880	TURING	10
4	SOUTHWEST	AZ	ARIZONA	890	MANDELBROT	- ←C-No-PO
4	SOUTHWEST	NM	NEW MEXICO	780	CHURCH	- ←C-No-PO
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	3

**Exercise:**

20M. Display the number and name of every region with at least one state, the code and name of every state with at least one supplier, the number and name of every supplier (including suppliers who do not sell any parts), and the part numbers of parts that can be purchased from these suppliers. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, SNO, SNAME, and PNO. Sort the result by PNO within SNO within STCODE within RNO. (Hint: Traverse REGION-STATE-SUPPLIER-PARTSUPP hierarchy.)



The following sample query traverses the REGION-STATE-CUSTOMER-PUR\_ORDER hierarchy. The query objective does not conform to a top-down join-sequence.

**Sample Query 20.13:** Display the number and name of every region, the code and name of every state with at least one customer, the number and name of every customer with at least one purchase order, and the number and status of these customers' purchase-orders. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, CNO, CNAME, PONO, and POSTATUS. Sort the result by PONO within CNO within STCODE within RNO.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME, PO.PONO, PO.POSTATUS

FROM REGION R
     LEFT OUTER JOIN
       STATE ST INNER JOIN CUSTOMER C   ON ST.STCODE = C.STCODE
           INNER JOIN PUR_ORDER PO ON C.CNO = PO.CNO
       ON R.RNO = ST.RNO

ORDER BY R.RNO, ST.STCODE, C.CNO, PO.PONO
```

[Result table is shown on following page.]

**Logic:** This query objective implies the following join-sequence.

```
REGION LOJ ((STATE IJ CUSTOMER) IJ PUR_ORDER)
```

This FROM-clause could be coded with parentheses as shown below.

```
FROM REGION R
     LEFT OUTER JOIN
       ((STATE ST INNER JOIN CUSTOMER C ON ST.STCODE = C.STCODE)
        INNER JOIN PUR_ORDER PO ON C.CNO = PO.CNO)
     ON R.RNO = ST.RNO
```

Note that the two INNER JOIN operations are executed before the LEFT OUTER JOIN operation. These INNER JOINS produce an intermediate result that excludes data about the state without customers (CONNECTICUT) and the customers without purchase orders (MANDELBROT and CHURCH). This intermediate join-result result becomes the right-table in the LEFT OUTER JOIN that preserves the MIDWEST region.

RNO	RNAME	STCODE	STNAME	CNO	CNAME	PONO	POSTATUS
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11101	C
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11102	P
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	11108	C
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	11109	P
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	11110	C
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	11111	P
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	11120	C
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	11121	C
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	11122	P
1	NORTHEAST	MA	MASSACHUSETTS	230	BOLYAI	11124	P
1	NORTHEAST	MA	MASSACHUSETTS	500	HILBERT	11150	P
2	NORTHWEST	OR	OREGON	300	NEWTON	11130	C
2	NORTHWEST	OR	OREGON	300	NEWTON	11133	P
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	11139	C
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	11141	P
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	11142	C
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	11144	P
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	11146	C
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	11148	C
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	11149	P
3	SOUTHEAST	FL	FLORIDA	600	BOOLE	11152	C
3	SOUTHEAST	FL	FLORIDA	600	BOOLE	11153	P
3	SOUTHEAST	FL	FLORIDA	660	CANTOR	11154	C
3	SOUTHEAST	FL	FLORIDA	660	CANTOR	11155	P
3	SOUTHEAST	GE	GEORGIA	700	RUSSELL	11156	C
3	SOUTHEAST	GE	GEORGIA	770	GODEL	11157	P
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11159	C
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11160	P
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11170	P
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11198	P
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	11158	C
5	MIDWEST	-	-	-	-	-	- ← R-No-ST

**Exercise:**

20N: Display the number and name of every region, the code and name of every state with at least one supplier, the number and name of every supplier that sells at least one part, and the part number and PSPRICE of these parts. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, SNO, SNAME, PNO, and PSPRICE. Sort the result by PNO within SNO within STCODE within RNO.

The following sample query traverses the REGION-STATE-CUSTOMER-PUR\_ORDER hierarchy. The query objective does not conform to a top-down join-sequence.

**Sample Query 20.14:** Display the number and name of every region, the code and name of every state, the number and name of every customer that has at least one purchase order, and the number and status of these purchase orders. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, CNO, CNAME, PONO and POSTATUS. Sort the result by PONO within CNO within STCODE within RNO.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME, PO.PONO, PO.POSTATUS

FROM REGION R
     LEFT OUTER JOIN STATE ST ON R.RNO = ST.RNO
     LEFT OUTER JOIN
         CUSTOMER C INNER JOIN PUR_ORDER PO ON C.CNO=PO.CNO
         ON ST.STCODE = C.STCODE

ORDER BY R.RNO, ST.STCODE, C.CNO, PO.PONO
```

[Result table is shown on following page.]

**Logic:** This query objective implies the following join-sequence.

(REGION LOJ STATE) LOJ (CUSTOMER IJ PUR\_ORDER)

This FROM-clause could be coded with parentheses as shown below.

```
FROM (REGION R
     LEFT OUTER JOIN STATE ST ON R.RNO = ST.RNO)
     LEFT OUTER JOIN
         (CUSTOMER C INNER JOIN PUR_ORDER PO ON C.CNO=PO.CNO)
         ON ST.STCODE = C.STCODE
```

This sequence of join-operations produces two intermediate join-results. Let IRST represent the first intermediate result produced by the LEFT OUTER JOIN of REGION and STATE; and let ICPO represent the second intermediate result produced by the INNER JOIN of CUSTOMER and PUR\_ORDER. Then the final join-result is produced by: IRST LEFT OUTER JOIN ICPO.

The first LEFT OUTER JOIN produces an intermediate join-result (IRST) that includes data from the non-matching region (MIDWEST) and all states (including CONNECTICUT). The INNER JOIN produces an intermediate result (ICPO) that excludes data about customers without purchase orders (MANDELBROT and CHURCH). Then the second LEFT OUTER JOIN of IRST and ICPO, where IRST is the left-table, preserves the non-matching region (MIDWEST) and non-matching state (CONNECTICUT).

RNO	RNAME	STCODE	STNAME	CNO	CNAME	PONO	POSTATUS	
1	NORTHEAST	CT	CONNECTICUT	-	-	-	-	←ST-No-C
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11101	C	
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11102	P	
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	11108	C	
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	11109	P	
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	11110	C	
1	NORTHEAST	MA	MASSACHUSETTS	200	HYPATIA	11111	P	
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	11120	C	
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	11121	C	
1	NORTHEAST	MA	MASSACHUSETTS	220	ZENO	11122	P	
1	NORTHEAST	MA	MASSACHUSETTS	230	BOLYAI	11124	P	
1	NORTHEAST	MA	MASSACHUSETTS	500	HILBERT	11150	P	
2	NORTHWEST	OR	OREGON	300	NEWTON	11130	C	
2	NORTHWEST	OR	OREGON	300	NEWTON	11133	P	
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	11139	C	
2	NORTHWEST	OR	OREGON	330	LEIBNIZ	11141	P	
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	11142	C	
2	NORTHWEST	WA	WASHINGTON	400	DECARTES	11144	P	
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	11146	C	
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	11148	C	
2	NORTHWEST	WA	WASHINGTON	440	PASCAL	11149	P	
3	SOUTHEAST	FL	FLORIDA	600	BOOLE	11152	C	
3	SOUTHEAST	FL	FLORIDA	600	BOOLE	11153	P	
3	SOUTHEAST	FL	FLORIDA	660	CANTOR	11154	C	
3	SOUTHEAST	FL	FLORIDA	660	CANTOR	11155	P	
3	SOUTHEAST	GE	GEORGIA	700	RUSSELL	11156	C	
3	SOUTHEAST	GE	GEORGIA	770	GODEL	11157	P	
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11159	C	
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11160	P	
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11170	P	
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11198	P	
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	11158	C	
5	MIDWEST	-	-	-	-	-	-	←R-No-ST

**Exercise:**

200. Display the number and name of every region, the code and name of every state, the number and name of every supplier that sells at least one part, and the part number and PSPRICE value of these parts. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, SNO, SNAME, PNO, and PSPRICE. Sort the result by PNO within SNO within STCODE within RNO.

The following sample query traverses the REGION-STATE-CUSTOMER-PUR\_ORDER-LINEITEM hierarchy. The query objective does not conform to a top-down join-sequence.

**Sample Query 20.15:** Display the following information about regions, states, customers, purchase-orders, and line-items.

- Display the number and name of all regions, including regions without any states.
- Display the code and name of all states, including states without any customers.
- Display customer number and name for those customers that have at least one purchase-order.
- Display each of these customer's purchase-order numbers, including numbers for purchase-orders that do not have any line-items.
- Display each line-item's line-number and part-number values.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME, C.CNO, C.CNAME,
       PO.PONO, LI.LINE, LI.PNO

FROM ((REGION R
      LEFT OUTER JOIN STATE ST ON R.RNO = ST.RNO)
     LEFT OUTER JOIN
      (CUSTOMER C INNER JOIN PUR_ORDER PO ON C.CNO = PO.CNO)
      ON ST.STCODE = C.STCODE)
     LEFT OUTER JOIN LINEITEM LI ON PO.PONO = LI.PONO

ORDER BY R.RNO, ST.STCODE, C.CNO, PO.PONO, LI.LINE
```

[Result table on following page shows some of its 65 rows.]

**Logic:** This query objective implies the following pseudo-code join-sequence.

```
((REGION LOJ STATE) LOJ (CUSTOMER IJ PUR_ORDER)) LOJ LINEITEM
```

The INNER JOIN of CUSTOMER and PUR\_ORDER excludes data about those customers (CHURCH and MANDELBROT) who do not have any purchase-orders. The three LEFT OUTER JOIN operations preserve all other non-matching rows.

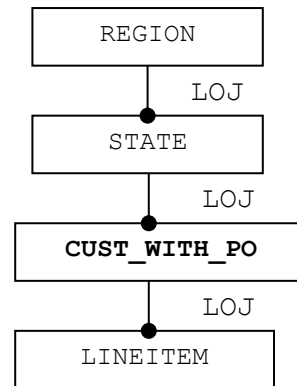
RNO	RNAME	STCODE	STNAME	CNO	CNAME	PONO	LINE	PNO
1	NORTHEAST	CT	CONNECTICUT	-	-	-	-	-
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11101	1	P1
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11101	2	P3
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11102	1	P3
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11102	2	P4
. . . . .								
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11159	1	P6
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11159	2	P7
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11160	1	P1
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11160	2	P7
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11170	1	P3
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11170	2	P4
4	SOUTHWEST	AZ	ARIZONA	880	TURING	11198	-	-
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	11158	1	P1
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	11158	2	P3
5	MIDWEST	-	-	-	-	-	-	-

**Alternative Solution:** In Chapter 27, Exercise 27R will specify a Common Table Expression to present another solution to this sample query. This alternative solution will code an INNER JOIN of CUSTOMER and PUR\_ORDER to form an intermediate result table called CUST\_WITH\_PO. The following code represents a top-down sequence of LEFT OUTER JOIN operations that traverses a four-level hierarchy which includes this CUST\_WITH\_PO intermediate result table.

```

FROM REGION R
LEFT OUTER JOIN STATE ST
ON R.RNO = ST.RNO
LEFT OUTER JOIN CUST_WITH_PO CWPO
ON ST.STCODE = CWPO.STCODE
LEFT OUTER JOIN LINEITEM LI
ON PO.PONO = LI.PONO

```



**Exercise:**

20P. Display the following information about regions, states, suppliers, and the parts that each supplier is allowed to sell and has already sold.

- Display the number and name of all regions.
- Display the code and name for all states.
- Display the supplier numbers and names for those suppliers who are allowed to sell at least one part.
- Display the part numbers of these parts.
- Display the part number and LIPRICE value of those parts these suppliers have already sold.

The next sample query traverses the REGION-STATE-CUSTOMER-PUR\_ORDER-LINEITEM hierarchy. The query objective does not conform to a top-down join-sequence.

**Sample Query 20.16:** Display the following information about regions, states, customers, purchase-orders, and line-items.

- Display the number and name of any region that has at least one state.
- Display the code and name of any state that has at least one customer.
- Display the number and name of all customers, including customers without purchase-orders.
- Display the customer's purchase-order numbers for those purchase-orders with at least one line-item.
- Display the line-number and corresponding part-number of each line-item.

```
SELECT R.RNO, R.RNAME, ST.STCODE, ST.STNAME,
       C.CNO, C.CNAME, PO.PONO, LI.LINE, LI.PNO

FROM ((REGION R
      INNER JOIN STATE ST   ON R.RNO = ST.RNO)
      INNER JOIN CUSTOMER C ON ST.STCODE = C.STCODE)
      LEFT OUTER JOIN
        (PUR_ORDER PO INNER JOIN LINEITEM LI ON PO.PONO = LI.PONO)
        ON C.CNO = PO.CNO

ORDER BY R.RNO, ST.STCODE, C.CNO, PO.PONO, LI.LINE
```

[Result table on following page shows some of its 64 rows.]

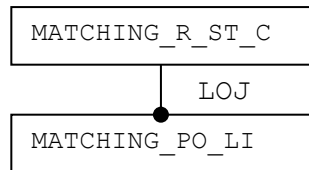
**Logic:** This query objective implies the following pseudo-code join-sequence.

```
((REGION IJ STATE) IJ CUSTOMER) LOJ (PUR_ORDER IJ LINEITEM)
```

The three INNER JOIN operations exclude data from non-matching rows that describe regions (MIDWEST) without states, states (CONNECTICUT) without customers, and purchase-orders (11198) without line-items. The LEFT OUTER JOIN preserves data about all customers, including customers (CHURCH and MANDELBROT) who do not have any purchase-orders.

RNO	RNAME	STCODE	STNAME	CNO	CNAME	PONO	LINE	PNO
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11101	1	P1
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11101	2	P3
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11102	1	P3
1	NORTHEAST	MA	MASSACHUSETTS	100	PYTHAGORAS	11102	2	P4
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	11108	1	P5
1	NORTHEAST	MA	MASSACHUSETTS	110	EUCLID	11108	2	P6
3	SOUTHEAST	GE	GEORGIA	770	GODEL	11157	2	P5
.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.
4	SOUTHWEST	AZ	ARIZONA	890	MANDELBROT	-	-	-
4	SOUTHWEST	NM	NEW MEXICO	780	CHURCH	-	-	-
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	11158	1	P1
4	SOUTHWEST	NM	NEW MEXICO	800	VON NEUMANN	11158	2	P3

**Alternative Solution:** In Chapter 27, Exercise 27S will specify two Common Table Expressions to present another solution to this sample query. Two INNER JOIN operations will join REGION, STATE, and CUSTOMER to produce an intermediate result table called MATCHING\_R\_ST\_C. Then another INNER JOIN operation will join PUR\_ORDER and LINEITEM to produce a second intermediate result table called MATCHING\_PO\_LI. Finally, a LEFT OUTER JOIN of these intermediate result tables will produce the final result.



**Exercise:**

- 20Q. Display the following information about regions, states, suppliers, and parts.
- Display the number and name of any region that has at least one state.
  - Display the code and name of any state that has at least one supplier.
  - Display the number and name of all suppliers, including those suppliers who are not yet allowed to sell any parts.
  - Display the part number and LIPRICE of each part the supplier has sold.



## Summary

This chapter introduced FROM-clauses that specified INNER JOIN and LEFT OUTER JOIN operations. Because such statements can become a little tricky, you may choose to code alternative statements using Common Table Expressions to be introduced in Chapter 27.

## Summary Exercises

Exercises 20R1, 20R2, 20S1, and 20S2 are optional exercises.

20R1. Work backwards. Transform the following FROM-clause into equivalent pseudo-code.

```
FROM REGION R
  LEFT OUTER JOIN STATE ST ON R.RNO = ST.RNO
  LEFT OUTER JOIN
    CUSTOMER C INNER JOIN PUR_ORDER PO
      ON C.CNO=PO.CNO
    ON ST.STCODE = C.STCODE
```

20R2. Work backwards. Transform the following FROM-clause into equivalent pseudo-code.

```
FROM REGION R
  LEFT OUTER JOIN STATE ST ON R.RNO = ST.RNO
  LEFT OUTER JOIN
    CUSTOMER C INNER JOIN PUR_ORDER PO
      ON C.CNO = PO.CNO
    ON ST.STCODE = C.STCODE
  LEFT OUTER JOIN LINEITEM LI ON PO.PONO = LI.PONO
```

20S1. Convert the following pseudo-code expression into a FROM-clause.

```
R LOJ (ST IJ (C LOJ PO))
```

20S2. Convert the following pseudo-code expression into a FROM-clause.

```
(R IJ ST) IJ ((C LOJ PO) LOJ LI))
```

Some of the following exercises have multiple solutions.

Suggestion: Represent query objective in pseudo-code and then transform pseudo-code to a FROM-clause.

20T. Display the number and name of every region, the code and name of every state with at least one supplier, and the number and name of every supplier in these states. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, SNO, and SNAME. Sort the result by SNO within STCODE within RNO.

20U. Display the part number of every part, the supplier number of every supplier who has sold this part, and the purchase-order number and line-item price for each sale of the part by the supplier. Display the columns in the following left-to-right sequence: PNO, SNO, PONO, and LIPRICE. Sort the result by PONO, SNO within PNO. (Hint: Follow the PART-PARTSUPP-LINEITEM hierarchy.)

20V. Display the number and name of any region that contains at least one state, the code and name of every state (including states without customers), the number and name of every customer in each state (including customers without purchase-orders), and the date of every purchase-order completed by these customers. Display the columns in the following left-to-right sequence: RNO, RNAME, STCODE, STNAME, CNO, CNAME, and PODATE. Sort the result by PODATE within CNO within STCOE within RNO.

20W. Reference the STATE-CUSTOMER-PUR\_ORDER-LINEITEM hierarchy.

- Display the RNO value of any region that has at least one state.
- Display the STCODE value of any state that has at least one customer.
- For each such state, display the CNO and CNAME values of its customers.
- For each customer with at least one purchase-order, display the customer's purchase-order numbers.
- For each purchase-order, display its LINE and corresponding PNO values even if the purchase order does not have any line-items.

20X. Reference the STATE-CUSTOMER-PUR\_ORDER-LINEITEM hierarchy.

- Display the STCODE and RNO values of any state that has at least one customer.
- For each such state, display the CNO and CNAME values of its customers, even if those customers do not have any purchase-orders.
- For each customer with at least one purchase-order that has at least one line-item, display the customer's purchase-order numbers.
- For those purchase-orders, display each line-item's LINE and PNO values.

## Appendix 20.5A: Theory & Efficiency

**Theory (Review):** Given tables (relations) T1, T2, and T3, the following logical rules apply.

The INNER JOIN obeys the Associative Law.

$$(T1 \text{ IJ } T2) \text{ IJ } T3 = T1 \text{ IJ } (T2 \text{ IJ } T3)$$

The LEFT OUTER JOIN obeys the Associative Law.

$$(T1 \text{ LOJ } T2) \text{ LOJ } T3 = T1 \text{ LOJ } (T2 \text{ LOJ } T3)$$

The INNER JOIN obeys the Commutative Law.

$$T1 \text{ IJ } T2 = T2 \text{ IJ } T1$$

Note: The LEFT OUTER JOIN does *not* obey the Commutative Law

For example, REGION LOJ STATE <> STATE LOJ REGION

**Optimizer Query Rewrite:** Assume a business user's articulation of her query objective encourages you to represent its join-sequence with the following pseudo-code expression (which references table aliases).

$$(R \text{ IJ } ST) \text{ IJ } (C \text{ LOJ } (PO \text{ LOJ } LI))$$

The optimizer could apply the above logical rules to generate seven other equivalent expressions that represent different join-sequences. The following page will present the application of these rules.

## Eight Equivalent Pseudo-Code Expressions

Exp-1: (R IJ ST) IJ (C LOJ (PO LOJ LI))    original expression

Exp-2: (ST IJ R) IJ (C LOJ (PO LOJ LI))  
by applying Commutative Law for IJ to Exp-1

Exp-3: (R IJ ST) IJ ((C LOJ PO) LOJ LI)  
by applying Associative Law for LOJ to Exp-1

Exp-4: (ST IJ R) IJ ((C LOJ PO) LOJ LI)  
by applying both Commutative Law for IJ and  
Associative Law for LOJ to Exp-1

Exp-5: (C LOJ (PO LOJ LI)) IJ (R IJ ST)  
by applying Commutative Law for IJ to Exp-1

Exp-6: (C LOJ (PO LOJ LI)) IJ (ST IJ RJ)  
by applying Commutative Law for IJ to Exp-5

Exp-7: ((C LOJ PO) LOJ LI) IJ (R IJ ST)  
by applying Associative Law for LOJ to Exp-5

Exp-8: ((C LOJ PO) LOJ LI) IJ (ST IJ R)  
by applying both Commutative Law for IJ and  
Associative Law for LOJ to Exp-5

**Optimizer Analysis:** Assuming that each parent-table is smaller than its child-table, and assuming no AND-conditions are specified in the ON-clauses, we speculate that the optimizer will implement Expression-3.

(R IJ ST) IJ ((C LOJ PO) LOJ LI))

Again, we remind you than an optimizer "has a mind of its own," and may not generate a desired application plan. In this case, you may try your hand at some logical gymnastics, deduce the most efficient join-sequence, and code this sequence within your FROM-clause.

---

# PART V

## Set Operations & CASE

This part of the book introduces SQL's set operations and CASE-expressions. These topics are important because they allow you to specify more sophisticated logic within SQL statements.

Chapter 21 introduces SQL's set operations: UNION, INTERSECT, and EXCEPT. These operations reflect SQL's mathematical heritage. (This mathematics is not difficult and will be presented in Appendix 21B.) This chapter also demonstrates how the UNION operation can be used to *indirectly* implement "If-Then" logic within a SELECT statement.

Chapter 22 introduces CASE-expressions. CASE allows you to *directly* specify If-Then logic within a SQL statement. We will see that CASE is superior to If-Then logic implemented via the UNION operation.

## Sample Tables

Chapter 21 references five new tables. Two of these tables, PROJMGR and PROJMGR2, describe project managers. These tables are almost identical. The only difference is the primary-key columns, PROJMGR.ENO and PROJMGR2.PMNO.

<u>PROJMGR</u>					<u>PROJMGR2</u>				
<u>ENO</u>	<u>PMNAME</u>	<u>MBA</u>	<u>RATE</u>	<u>DNO</u>	<u>PMNO</u>	<u>PMNAME</u>	<u>MBA</u>	<u>RATE</u>	<u>DNO</u>
1000	MOE	N	500.00	20	1500	MOE	N	500.00	20
2500	DICK	N	100.00	40	2500	DICK	N	100.00	40
6000	GEORGE	Y	10.00	20	6500	GEORGE	Y	10.00	20
4500	DON	N	70.00	40	4500	DON	N	70.00	40
Primary Key: ENO					Primary Key: PMNO				

The PROJMGR table describes project managers within an organization where *some, but not all, project managers are also employees*. Hence some PROJMGR.ENO values (e.g., 1000 and 6000) match ENO values found in the EMPLOYEE table. Furthermore, when a PROJMGR.ENO value matches an EMPLOYEE.ENO, corresponding PROJMGR.PMNAME and EMPLOYEE.ENAME values will also match. Also, notice that DICK and DON are assigned ENO numbers, even though they are not employees.

The PROJMGR2 table differs from the PROJMGR table because it describes project managers within an organization where a *project manager cannot be an employee* (even though some project managers coincidentally have the same name as an employee).

Chapter 21 also references three tables that describe parts used in projects. These tables are displayed below.

<u>PROJ1PARTS</u>				<u>PROJ2PARTS</u>			<u>PROJ3PARTS</u>			
<u>PNO</u>	<u>PNAME</u>	<u>PCOLOR</u>	<u>QTY</u>	<u>PNO</u>	<u>PNAME</u>	<u>PWT</u>	<u>PNO</u>	<u>PNAME</u>	<u>PCOLOR</u>	<u>QTY</u>
P1	PART1	RED	16	P3	PART3	20	P3	PART3	PINK	98
P2	PART2	BLUE	16	P4	PART4	10	P6	PART6	BLUE	97
P4	PART4	YELLOW	17	P5	PART5	20	P7	PART7	PINK	95
P5	PART5	RED	15	P6	PART6	12	P8	PART8	PINK	99
Primary Key: PNO				Primary Key: PNO			Primary Key: PNO			

The PROJ1PARTS table describes parts used in Project1; PROJ2PARTS describes parts used in Project2; and PROJ3PARTS describe parts used in Project3. Project1 and Project2 may use some of the same parts (e.g., P4 and P5); and, Project2 and Project3 may use some of the same parts (e.g., P3 and P6). However, if a part is used in Project1, it cannot be used in Project3; and vice versa.

## Set Operations:

# UNION, INTERSECT, and EXCEPT

This chapter introduces three set operations designated by the keywords UNION, INTERSECT, and EXCEPT. Appendix 21B describes the mathematical foundations of these operations. This appendix is optional reading, and it is not difficult. Therefore, you are encouraged (but not required) to read this appendix before starting this chapter.

The keywords UNION, INTERSECT, and EXCEPT are specified between individual "Sub-SELECTs." The following skeleton-code outlines some examples.

SELECT _____	SELECT _____	SELECT _____
FROM _____	FROM _____	FROM _____
WHERE _____	WHERE _____	WHERE _____
<b>UNION</b>	<b>INTERSECT</b>	<b>EXCEPT</b>
SELECT _____	SELECT _____	SELECT _____
FROM _____	FROM _____	FROM _____
WHERE _____	WHERE _____	WHERE _____

Each Sub-SELECT generates an intermediate result. Then a set operation processes the intermediate results to produce a final result.

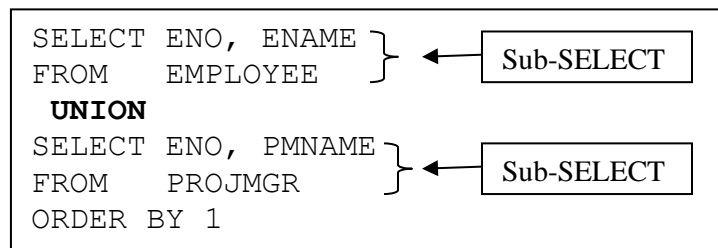




## UNION

Informally, UNION asks the system to “merge” rows from two union-compatible intermediate result tables and remove any duplicate rows from the result.

**Sample Query 21.1:** Reference the EMPLOYEE and PROJMGR tables. Display the employee numbers and names of all employees and all project managers. Sort the result table by the first column.



<u>ENO</u>	<u>ENAME</u>
1000	MOE
2000	LARRY
2500	DICK
3000	CURLY
4000	SHEMP
4500	DON
5000	JOE
6000	GEORGE

**Syntax:** UNION is specified between two Sub-SELECTs that generate union-compatible intermediate results.

**Logic:** The intermediate results are merged into the final result. *Duplicate rows are eliminated from the final result.* Specifically, rows describing Employees 1000 and 6000 are only displayed once.

**Column Headings:** If two corresponding columns have different names (e.g., ENAME and PMNAME), the front-end tool determines the column heading. Many front-end tools display the column names associated with the first Sub-SELECT as illustrated above.

**Possible Incidental Sort:** If you remove the ORDER BY clause from this statement, there is a good chance that the final result will be incidentally sorted. (See Appendix 21A.)

## Displaying “Labels”

Examine the result table in the previous sample query. Notice that you cannot determine if an individual row was derived from the EMPLOYEE table or from the PROJMGR table. If this information is important, you can display some constant value to serve as a label.

**Sample Query 21.2:** Make a simple change to the previous SELECT-statement. Display a third column containing the character-strings 'EMP' or 'PM' to indicate the source table for each row. Sort the final result table by the first column.

```
SELECT ENO, ENAME, 'EMP' SOURCETAB
FROM EMPLOYEE
UNION
SELECT ENO, PMNAME, 'PM'
FROM PROJMGR
ORDER BY 1
```

<u>ENO</u>	<u>ENAME</u>	<u>SOURCETAB</u>
1000	MOE	EMP
1000	MOE	PM
2000	LARRY	EMP
2500	DICK	PM
3000	CURLY	EMP
4000	SHEMP	EMP
4500	DON	PM
5000	JOE	EMP
6000	GEORGE	EMP
6000	GEORGE	PM

**Syntax & Logic:** Nothing New.

**Important Observation:** The result table for the previous Sample Query 21.1 has 8 rows, whereas the above result table has 10 rows. Observe that the above result table displays two rows describing Employee 1000 and two rows describing Employee 6000. Displaying the label eliminated the possibility of duplicate rows.

## INTERSECT

After executing both Sub-SELECTs, INTERSECT asks the system to return just those rows that are present in both intermediate result tables, and, if necessary, remove duplicate rows from the result.

**Sample Query 21.3:** Display the employee numbers and names of all persons who are described in both the EMPLOYEE and PROJMgr tables. Sort the result by the first column.

```
SELECT ENO, ENAME
FROM   EMPLOYEE
      INTERSECT
SELECT ENO, PMNAME
FROM   PROJMgr
ORDER BY 1
```

```
ENO  ENAME
1000  MOE
6000  GEORGE
```

**Syntax:** INTERSECT is specified between two Sub-SELECTs. The Sub-SELECTs must generate union-compatible results. (We could say the Sub-SELECTs are "intersect-compatible," but most users will say "union-compatible.")

**Logic:** The intermediate-result tables are shown below. Only rows common to both intermediate-results appear in the final result.

<u>ENO</u>	<u>ENAME</u>		<u>ENO</u>	<u>PMNAME</u>
1000	MOE	←————→	1000	MOE
2000	LARRY		2500	DICK
3000	CURLY		6000	GEORGE
4000	SHEMP		4500	DON
5000	JOE			
6000	GEORGE	←————→		

**Alternative Solutions:** Exercise 21F will invite you to code a join-operation to satisfy this query objective. Chapter 25 (Exercise 25K) will present two additional solutions.

## EXCEPT (MINUS)

EXCEPT asks the system to select those rows from the first-intermediate-result table that are *not* present in the second-intermediate-result table, and, if necessary, remove any duplicate rows from the result. [Note: ORACLE uses the keyword MINUS instead of EXCEPT.]

**Sample Query 21.4:** Display the employee number and name of every employee who is not a project manager. (I.e., Display the employee number and name of every person who is described in the EMPLOYEE table but not described in the PROJMGR table.) Sort the result by the first column.

```
SELECT ENO, ENAME
FROM   EMPLOYEE
EXCEPT
SELECT ENO, PMNAME
FROM   PROJMGR
ORDER BY 1
```

```
ENO ENAME
2000 LARRY
3000 CURLY
4000 SHEMA
5000 JOE
```

**Syntax:** EXCEPT is specified between two Sub-SELECTs. The Sub-SELECTs must generate union-compatible intermediate results.

**Logic:** The EXCEPT operation returns those rows from first intermediate-result table that are not found in the second intermediate-result table.

```
ENO ENAME                ENO PMNAME
1000 MOE                    1000 MOE
2000 LARRY ←                2500 DICK
3000 CURLY ←                6000 GEORGE
4000 SHEMA ←                4500 DON
5000 JOE ←
6000 GEORGE
```

Notice that interchanging the Sub-SELECT statements would generate a different result.

**Alternative Solutions:** Chapter 25 (Exercise 25L) will present two additional solutions.

## Exercises

Exercises 21A-21E reference the following PROJ1PARTS and PROJ2PARTS tables. Recall that some parts (e.g., P4 and P5) can be used in both projects.

<u>PROJ1PARTS</u>				<u>PROJ2PARTS</u>		
<u>PNO</u>	<u>PNAME</u>	<u>PCOLOR</u>	<u>QTY</u>	<u>PNO</u>	<u>PNAME</u>	<u>PWT</u>
P1	PART1	RED	16	P3	PART3	20
P2	PART2	BLUE	16	P4	PART4	10
P4	PART4	YELLOW	17	P5	PART5	20
P5	PART5	RED	15	P6	PART6	12

- 21A. Display the part number and name of all parts used by either Project1 or Project2.
- 21B. Display the part number and name of any part that is used in both Project1 and Project2.
- 21C. (i) Display the part number and name of any part that is used in Project1 but not used in Project2.  
(ii) Display the part number and name of any part that is used in Project2 but not used in Project1.
- 21D. The following statement produces a potentially confusing result. Why?

```
SELECT PNO, PNAME, QTY
FROM PROJ1PARTS
UNION
SELECT PNO, PNAME, PWT
FROM PROJ2PARTS
ORDER BY 1
```

- 21E. Modify the above statement to display a label to distinguish QTY values from PWT values.
- 21F. Code an alternative solution to Sample Query 21.3 using a join-operation instead of specifying INTERSECT.

## UNION ALL

Placing the keyword ALL after UNION tells the system to perform a "union" operation without removing duplicate rows from the result. The following sample query is not a very realistic, but it does illustrate the basic functionality of UNION ALL. Sample Queries 21.6 and 21.7 will present more realistic examples.

**Sample Query 21.5:** This query is a variation of Sample Query 21.1. Display the employee numbers and names of all employees and all project managers. *However, do not remove any duplicate rows from the final result.* Sort the final result by the first column.

```
SELECT ENO, ENAME
FROM   EMPLOYEE
      UNION ALL
SELECT ENO, PMNAME
FROM   PROJMgr
ORDER BY 1
```

```
ENO  ENAME
-----
1000 MOE
1000 MOE
2000 LARRY
2500 DICK
3000 CURLY
4000 SEMP
4500 DON
5000 JOE
6000 GEORGE
6000 GEORGE
```

**Syntax:** Specify UNION ALL between the individual Sub-SELECTs.

**Logic:** This example displays all rows generated by the individual Sub-SELECTs. Observe the duplicate rows that describe MOE and GEORGE. Because we discourage displaying duplicate rows, you should consider including some row label as illustrated in the Sample Query 21.2.

The next sample query references the following PROJMGR2 table.

<u>PROJMGR2:</u>	<u>PMNO</u>	<u>PMNAME</u>	<u>MBA</u>	<u>RATE</u>	<u>DNO</u>
	1500	MOE	N	500.00	20
	2500	DICK	N	100.00	40
	6500	GEORGE	Y	10.00	20
	4500	DON	N	70.00	40

Recall that PROJMGR2.PMNO values cannot match any EMPLOYEE.ENO value, and vice versa. (I.e., The primary keys values of EMPLOYEE and PROJMGR2 tables are disjoint.)

**Sample Query 21.6:** This sample query is similar to the previous sample query. Reference the EMPLOYEE and PROJMGR2 tables. Display the number and name of every employee and project manager. Sort the result by the first column.

```
SELECT ENO, ENAME
FROM   EMPLOYEE
UNION ALL
SELECT PMNO, PMNAME
FROM   PROJMGR2
ORDER BY 1
```

```
ENO  ENAME
1000  MOE
1500  MOE
2000  LARRY
2500  DICK
3000  CURLY
4000  SHEMP
4500  DON
5000  JOE
6000  GEORGE
6500  GEORGE
```

**Logic:** Design constraints imply that duplicate rows cannot appear within the intermediate-result tables. Therefore, we could have produced the same final result by specifying UNION instead of UNION ALL. However, for efficiency purposes, you might prefer UNION ALL.

**Efficiency:** Sometimes, you can improve efficiency by specifying UNION ALL instead of UNION. When you specify UNION ALL you are telling the system that it can omit any internal processing used to detect and remove duplicate rows. (Appendix 21A will comment on this matter.)



## “If-Then” Logic

UNION ALL can be used to indirectly implement some basic “If-Then” logic. Unlike previous sample queries, the following sample query specifies Sub-SELECTs that reference the *same table* (EMPLOYEE); and, unlike previous sample queries, the following sample query specifies two set operations.

**Sample Query 21.7:** Display the ENO and ENAME values from all EMPLOYEE rows. For confidentiality reasons, do not display SALARY values. Instead, for each employee, display a textual label indicating that the employee’s salary is “TOO SMALL”, “TOO BIG”, or “OK” according to the following business rules.

- If the SALARY < 1000, then display “TOO SMALL”
- If the SALARY > 3000, then display “TOO BIG”
- Otherwise, display “OK”

Sort the result by the ENAME column.

```
SELECT ENO, ENAME, 'TOO SMALL' TEXTLABEL
FROM   EMPLOYEE
WHERE  SALARY < 1000
UNION ALL
SELECT ENO, ENAME, 'TOO BIG'
FROM   EMPLOYEE
WHERE  SALARY > 3000
UNION ALL
SELECT ENO, ENAME, 'OK'
FROM   EMPLOYEE
WHERE  SALARY BETWEEN 1000 and 3000
ORDER BY 2
```

ENO	ENAME	TEXTLABEL
3000	CURLY	OK
6000	GEORGE	TOO BIG
5000	JOE	TOO SMALL
2000	LARRY	OK
1000	MOE	OK
4000	SHEMP	TOO SMALL

**Logic:** Here, the logic is straightforward because we specified the same set operation (UNION ALL) twice. The last section in this chapter will show that we must be careful when specifying multiple *different* set operations.

**Alternative Solution:** Sample Query 22.3 will present a better solution that specifies a CASE-expression.

## Cautionary Observations about UNION ALL

We reference the three previous sample queries to make some cautionary observations about UNION ALL.

- Sample Query 21.5 demonstrated that UNION ALL may return duplicate rows. *However*, we generally discourage displaying duplicate rows.
- Sample Query 21.6 noted that UNION ALL may be more efficient than UNION because it avoids the cost of finding and removing duplicate rows. *However*, from an ideal perspective, you should not have to consider efficiency.
- Sample Query 21.7 demonstrated that UNION ALL can be used to implement If-Then logic. *However*, the next chapter will show that CASE provides a more direct method of implementing If-Then logic.

**Ancient History:** Some early versions of SQL did not support UNION ALL. Users of these products complained to their database vendors about using UNION to satisfy query objectives like Sample Query 21.6 where execution time was slow because the system did extra work to identify and remove duplicate rows that the user knew could not occur. This inefficiency was one reason database vendors decided to support UNION ALL.

### Exercises:

Reference the PROJ1PARTS and PROJ3PARTS tables. Recall that Project1 and Project3 can never use the same part.

21G. Display the part number and name of all parts used by either Project1 or Project3.

21H. Reference the PROJ1PARTS table. Produce a result that displays every part number and name, followed by a character-string indicating if the QTY column contains a value that is less than, equal to, or greater than 16. Sort the result by PNO. The result should look like:

<u>PNO</u>	<u>PNAME</u>	<u>COMMENTARY</u>
P1	PART1	QTY EQUAL TO 16
P2	PART2	QTY EQUAL TO 16
P4	PART4	QTY GREATER THAN 16
P5	PART5	QTY LESS THAN 16

## INTERSECT ALL

We introduce INTERSECT ALL, but we do not present any relevant sample queries because INTERSECT ALL does not find many real-world applications.

Consider the DNO values in the EMPLOYEE and PROJMGR tables.

<u>EMPLOYEE.DNO</u>	<u>PROJMGR.DNO</u>
20	20
10	40
20	20
40	40
10	
20	

```
Execute:      SELECT DNO FROM EMPLOYEE
              INTERSECT ALL
              SELECT DNO FROM PROJMGR
```

```
Result:      DNO
              20
              20
              40
```

**Logic:** Using very casual terminology, we say that INTERSECT ALL returns *all* DNO values that “overlap” with each other.

```
EMPLOYEE.DNO: {10, 10, 20, 20, 20, 40}
                ↑   ↑   ↗
                ↓   ↓   ↘
PROJMGR.DNO:  {20, 20, 40, 40}
```

Observe that:

- The EMPLOYEE.DNO column contains three 20s. The PROJMGR.DNO column contains two 20s. Two of these 20s “overlap” each other as illustrated above. Hence, two 20 values appear in the result.
- EMPLOYEE.DNO contains one 40. PROJMGR.DNO contains two 40s. Only one 40 overlaps the other. Hence, one 40 value appears in the result.
- No other values overlap with each other.

## EXCEPT ALL

We introduce EXCEPT ALL, but we do not present any relevant sample queries because EXCEPT ALL does not find many real-world applications.

The following example illustrates EXCEPT ALL.

```
Execute:      SELECT DNO FROM EMPLOYEE
              EXCEPT ALL
              SELECT DNO FROM PROJMGR
```

```
Result:      DNO
              10
              10
              20
```

**Logic:** Using very casual terminology, we say that EXCEPT ALL returns just those EMPLOYEE.DNO values that do *not* overlap with PROJMGR.DNO values. Again, the following figure uses arrows to identify the overlapping values. The *other* (non-overlapping) EMPLOYEE.DNO values (the underlined values) are returned by the EXCEPT ALL operation.

```
EMPLOYEE.DNO:  {10, 10, 20, 20, 20, 40}
                ↑   ↑   ↑
                ↓   ↓   ↘
PROJMGR.DNO:   {20, 20, 40, 40}
```

Observe that:

- EMPLOYEE.DNO contains two 10s. PROJMGR.DNO has no 10s. Hence both of the EMPLOYEE.DNO 10s appear in the result.
- EMPLOYEE.DNO contains three 20s. PROJMGR.DNO contains two 20s. Only one EMPLOYEE.DNO value of 20 does not overlap with the 20s in PROJMGR.DNO. Hence one 20 appears in the result.
- EMPLOYEE.DNO contains one 40. PROJMGR.DNO contains two 40s. Hence, EMPLOYEE.DNO does not contain any non-overlapping 40s.

## Null Values: UNION – INTERSECT – EXCEPT

Here we describe set operations within the context of null values. The following examples reference the JUNK1 and JUNK2 tables shown below. Both tables have just one column (JNO) that contains some null values represented by hyphens.

<u>JUNK1.JNO</u>	<u>JUNK2.JNO</u>
20	20
10	40
20	20
40	40
10	-
20	30
-	
-	
-	

With UNION, INTERSECT, and EXCEPT, multiple null values are treated as duplicates, and duplicate null values do not appear in the result. (This is similar to applying DISTINCT to null values as illustrated in Sample Query 11.10.) The following examples illustrate this behavior.

<pre>SELECT * FROM JUNK1 <b>UNION</b> SELECT * FROM JUNK2 ORDER BY JNO    <u>JNO</u>    10    20    30    40    -</pre>	<pre>SELECT * FROM JUNK1 <b>INTERSECT</b> SELECT * FROM JUNK2 ORDER BY JNO    <u>JNO</u>    20    40    -</pre>
<pre>SELECT * FROM JUNK1 <b>EXCEPT</b> SELECT * FROM JUNK2 ORDER BY JNO    <u>JNO</u>    10</pre>	

## Null Values: UNION ALL – INTERSECT ALL – EXCEPT ALL

If you specify ALL with any set operation, multiple null values are not treated as duplicates. Hence multiple null values can appear in the result. The following examples illustrate this behavior.

<pre>SELECT * FROM JUNK1 <b>UNION ALL</b> SELECT * FROM JUNK2 ORDER BY JNO</pre> <p><u>JNO</u> 10 10 20 20 20 20 20 30 40 40 40 - - - -</p>	<pre>SELECT * FROM JUNK1 <b>INTERSECT ALL</b> SELECT * FROM JUNK2 ORDER BY JNO</pre> <p><u>JNO</u> 20 20 40 -</p>
<pre>SELECT * FROM JUNK1 <b>EXCEPT ALL</b> SELECT * FROM JUNK2 ORDER BY JNO</pre> <p><u>JNO</u> 10 10 20 - -</p>	

## Execution Hierarchy for Multiple Set Operations

In Chapter 4, we described the hierarchy of execution for the Boolean operations where, in the absence of parentheses, NOT is executed first, followed by AND, then followed by OR.

Later, in Chapter 7, we described the hierarchy of execution for the arithmetic operations where, in the absence of parentheses, multiplication and division are executed before addition the subtraction. Addition and subtraction are at the same level and are executed as they appear in a left-to-right reading of the expression. In a similar manner, multiplication and division are at the same level and are executed as they appear in a left-to-right reading of the expression.

In this section we consider the execution hierarchy for the set operations: UNION, INTERSECT, and EXCEPT.

**Strong Recommendation:** Specify parentheses (as previously emphasized for the Boolean and arithmetic operations). This recommendation is especially emphasized for the set operations because different database systems may default to different hierarchies! Ouch! Therefore, again, always specify parentheses to indicate your desired execution sequence.

**Execution Hierarchy (DB2 and SQL Server):** In the absence of parentheses, the execution hierarchy is:

- INTERSECT is executed first.
- UNION and EXCEPT are executed second. These operations are at the same level and are executed as they appear in a left-to-right reading of the expression.

**ORACLE:** See end of this section.

The following examples assume that you are using DB2 or SQL Server, and your SELECT statement references the following three one-column tables (SETA, SETB, and SETC).

<u>SETA</u>	<u>SETC</u>	<u>SETB</u>
<u>XNO</u>	<u>XNO</u>	<u>XNO</u>
10	10	10
20	20	30
30	40	40
50	70	60

DB2 and SQL Server: INTERSECT has precedence over UNION.

Parentheses are specified in the following Figures 21.1a and 21.1b. Hence, the execution hierarchy does not come into play.

```
(SELECT XNO FROM SETA
 UNION
 SELECT XNO FROM SETB)
 INTERSECT
 SELECT XNO FROM SETC
```

<u>XNO</u>
10
20
40

Figure 21.1a: Parentheses imply UNION executed before INTERSECT

```
SELECT XNO FROM SETA
 UNION
 (SELECT XNO FROM SETB
 INTERSECT
 SELECT XNO FROM SETC)
```

<u>XNO</u>
10
20
30
40
50

Figure 21.1b: Parentheses imply INTERSECT executed before UNION

Parentheses are not specified in the following Figure 21.1c. Hence, the execution hierarchy implies that INTERSECT is executed first (even though UNION is coded before INTERSECT). Notice that the Figure 21.1c result is the same as the Figure 21.1b result where parentheses explicitly implied that INTERSECT was executed first.

```
SELECT XNO FROM SETA
 UNION
 SELECT XNO FROM SETB
 INTERSECT
 SELECT XNO FROM SETC
```

<u>XNO</u>
10
20
30
40
50

Figure 21.1c: No Parentheses - INTERSECT executed before UNION



DB2 and SQL Server: INTERSECT has precedence over EXCEPT.

Parentheses are specified in the following Figures 21.2a and 21.2b. Hence, the execution hierarchy does not come into play.

```
(SELECT XNO FROM SETA
  EXCEPT
  SELECT XNO FROM SETB)
INTERSECT
SELECT XNO FROM SETC

XNO
 20
```

Figure 21.2a: Parentheses imply  
EXCEPT executed before INTERSECT

```
SELECT XNO FROM SETA
  EXCEPT
  (SELECT XNO FROM SETB
    INTERSECT
    SELECT XNO FROM SETC)

XNO
 20
 30
 50
```

Figure 21.2b: Parentheses imply  
INTERSECT executed before EXCEPT

Parentheses are not specified in the following Figure 21.2c. Hence the execution hierarchy implies that INTERSECT is executed first (even though EXCEPT is coded before INTERSECT). Notice that the Figure 21.2c result is the same as the Figure 21.2b result where parentheses explicitly implied that INTERSECT was executed first.

```
SELECT XNO FROM SETA
  EXCEPT
  SELECT XNO FROM SETB
    INTERSECT
    SELECT XNO FROM SETC

XNO
 20
 30
 50
```

Figure 21.2c: No Parentheses -  
INTERSECT executed before EXCEPT

DB2 and SQL Server: UNION and EXCEPT are at same level.

Parentheses are specified in the following Figures 21.3a and 21.3b. Hence, the execution hierarchy does not come into play.

```
(SELECT XNO FROM SETA
 UNION
 SELECT XNO FROM SETB)
 EXCEPT
 SELECT XNO FROM SETC

 XNO
 30
 50
 60
```

Figure 21.3a: Parentheses imply UNION executed before EXCEPT

```
SELECT XNO FROM SETA
 UNION
 (SELECT XNO FROM SETB
 EXCEPT
 SELECT XNO FROM SETC)
```

<u>XNO</u>
10
20
30
50
60

Figure 21.3b: Parentheses imply EXCEPT executed before UNION

Parentheses are not specified in the following Figures 21.3c and 21.3d. Because UNION and EXCEPT are at the same (second) level in the execution hierarchy, the left-to-right sequence of the operations dictates the execution sequence.

In Figure 21.3c UNION is executed first because it is coded before EXCEPT. (The result is the same as Figure 21.3a.) In Figure 21.3d EXCEPT is executed first because it is coded before UNION. (The result is the same as Figure 21.3b.)

```
SELECT XNO FROM SETA
 UNION
 SELECT XNO FROM SETB
 EXCEPT
 SELECT XNO FROM SETC

 XNO
 30
 50
 60
```

Figure 21.3c: No Parentheses - UNION executed before EXCEPT

```
SELECT XNO FROM SETB
 EXCEPT
 SELECT XNO FROM SETC
 UNION
 SELECT XNO FROM SETA

 XNO
 10
 20
 30
 50
 60
```

Figure 21.3d: No Parentheses - EXCEPT executed before UNION

## ORACLE: Potential Confusion

In the past, and maybe today (depending upon when you are reading this chapter), ORACLE did not implement any execution hierarchy for the set operations. All set operations were at the same level. If parentheses were not specified, the set operations were executed in the order that they appeared in a left-to-right reading of the expression.

**\*\*\* However** - Read the following statement copied from the ORACLE 10.2 reference manual.

To comply with emerging SQL standards, a future release of Oracle will give the INTERSECT operator greater precedence than the other set operators. Therefore, you should use parentheses to specify order of evaluation in queries that use the INTERSECT operator with other set operators.

Again, code parentheses to explicitly specify your desired execution sequence.

**Author Comment:** I became confused when I first executed the following SELECT statement in DB2 where I *incorrectly* assumed that UNION would be executed first.

```
SELECT XNO FROM SETA
UNION
SELECT XNO FROM SETB
INTERSECT
SELECT XNO FROM SETC
```

Why did the SQL Standards Committee decide to give INTERSECT a higher precedence? I don't know, but there may be a reasonable basis for this decision: INTERSECT is similar to AND, and UNION is similar to OR. Therefore, if AND has a higher precedence than OR, then INTERSECT should also have a higher precedence than UNION.

## Summary

**UNION:** *UNION* is fundamental in the sense that, given a query objective that requires a UNION operation, we cannot always specify some alternative SQL code to satisfy this objective.

**INTERSECT and EXCEPT:** INTERSECT and EXCEPT are useful operations, but they are not fundamental. Exercises 25K and 25L will present alternative solutions to this chapter's sample queries that specified INTERSECT and EXCEPT. (Early versions of SQL did not support INTERSECT and EXCEPT. Hence, SQL users were forced to code alternative solutions. Today, many users continue to prefer these alternative solutions.)

**Keyword ALL:** UNION ALL can be useful. However, you should be aware of the previous cautionary comments about this operation.

**Design Observation:** Examination of the EMPLOYEE and PROJMgr tables could motivate the following observation.

The PROJMgr table contains a column called ENO, implying that all project managers are assigned employee numbers, further implying that all project managers are employees. However, we stated that some project managers are not employees, and their PROJMgr.ENO values do not appear in the EMPLOYEE table. There appears to some inconsistency in this design.

The database designer might defend this design by noting that a project manager may or may not be an employee. For example, a project manager who is not an employee might be an external consultant who is assigned a "dummy" employee number, and dummy employee numbers are not stored in the EMPLOYEE table. While this dummy employee may be untidy, the EMPLOYEE and PROJMgr tables accurately model this business practice. Unfortunately, in the real-world, you will encounter untidy database designs derived from untidy business practices that complicate your efforts to know your data.

## Summary Exercises

- 21I. Reference the EMPLOYEE and PROJMGR tables. Display the employee number and name of any person who works in or manages projects for Department 20.
- 21J. Reference the EMPLOYEE and PROJMGR tables. Modify the previous exercise. Display "EMPLOYEE" or "PROJECT MANAGER", in the third column to indicate that the person is an employee or a project manager. (Two rows will be displayed for any person who is both an employee and a project manager.)
- 21K. Reference the EMPLOYEE and PROJMGR tables. Display the employee number and name of any person who is both an employee and project manager in Department 20. Sort the result by the first column.
- 21L. Reference the EMPLOYEE and PROJMGR tables. Display the employee number and name of any project manager who is not an employee.
- 21M. Reference the PROJ2PARTS table. Display every part number and name and a character-string indicating if the PWT column contains a value that is less than, equal to, or greater than 12. Sort the result by the first column. The result should look like:

```
PNO PNAME COMMENTARY
P3  PART3 WEIGHT IS GREATER THAN 12
P4  PART4 WEIGHT IS LESS THAN 12
P5  PART5 WEIGHT IS GREATER THAN 12
P6  PART6 WEIGHT IS EQUAL TO 12
```

- 21N. Reference the EMPLOYEE table. Display the department number and the total salary for each department. Also, display the final total of all salaries. Your SELECT statement should specify UNION ALL. The result should look like:

```
DNO   SUMSALARY
10      2400.00
20     14000.00
40       500.00
Final 16900.00
```

Comment: Optional Chapter 9.5 (Sample Query 9.21) describes a better method using the ROLLUP option with the GROUP BY clause.

210. Consider the following SELECT statements. Produce two results for each statement. (1) Assume that INTERSECT has precedence over UNION. (2) Assume there is no precedence among the set operations. Sometimes, both assumptions produce the same result

Statement-1: (SELECT PNO, PNAME FROM PROJ2PARTS  
UNION  
SELECT PNO, PNAME FROM PROJ3PARTS)  
INTERSECT  
SELECT PNO, PNAME FROM PROJ1PARTS

Statement-2: SELECT PNO, PNAME FROM PROJ2PARTS  
UNION  
SELECT PNO, PNAME FROM PROJ3PARTS  
INTERSECT  
SELECT PNO, PNAME FROM PROJ1PARTS

Statement-3: SELECT PNO, PNAME FROM PROJ2PARTS  
INTERSECT  
(SELECT PNO, PNAME FROM PROJ3PARTS  
UNION  
SELECT PNO, PNAME FROM PROJ1PARTS)

21P. Display the part numbers and names of any part this used in all three projects. (Trick question!)

## Appendix 21A: Efficiency

**Removing Duplicate Rows:** Coding UNION, INTERSECT, or EXCEPT (without specifying ALL) requires the system to identify and remove duplicate rows. This process, which may involve sorting, could incur an additional performance cost. Appendix 3A discussed sorting considerations associated with the DISTINCT keyword. Similar considerations apply to SQL's set operations.

**Multiple Sub-SELECTs Reference the Same Table:** Consider Sample Query 21.7 where three Sub-SELECTs referenced the same table. If this table is very large, and the optimizer decides to scan the table three times (once for each Sub-SELECT), then the total cost could be expensive. In this circumstance, specifying CASE (to be discussed in the next chapter) should be more efficient.

**Possible Do-It-Yourself Query Rewrite:** Consider the following.

```
SELECT * FROM BIG_TABLE WHERE COLX IN (2, 7, 19)
```

Assume that:

- BIG\_TABLE has a billion rows
- There is a non-unique index on COLX.
- Approximately 10 rows match the WHERE-condition.

Given the small selectivity, the optimizer should decide to use the index on COLX. However, what if your imperfect optimizer makes a poor decision and decides to scan the table? In this circumstance, you *could possibly* entice the optimizer to use the COLX index by rewriting the above SELECT statement as:

```
SELECT * FROM BIG_TABLE WHERE COLX = 2
UNION ALL
SELECT * FROM BIG_TABLE WHERE COLX = 7
UNION ALL
SELECT * FROM BIG_TABLE WHERE COLX = 19
```

This code should encourage the optimizer to use the index to satisfy each Sub-SELECT. However - Be Careful!!! This revised statement would be horribly inefficient if the system decides to perform three scans of BIG\_TABLE.

This kind of user query rewrite is an *ugly patch job*. Modern optimizers are very intelligent. They reduce, but do not always eliminate, the need for this kind of user query rewrite.

## Appendix 21B: Theory

**Mathematical Foundation:** Recall that each table (ideally) corresponds to a set where each row in the table corresponds to an element in the set. For example, consider a small school with a baseball team and a chess team. We represent the baseball team by set B, and represent the chess team by set C. Student names, the elements for each set, are listed below.

B = {Moe, Ruth, Larry, Curly, Gerig}

C = {Newton, Moe, Larry, Einstein, Curly}

From a database perspective, we can represent each set as a table (the BTEAM table and the CTEAM table) where each table has just one column.

<u>BTEAM</u>	<u>CTEAM</u>
<u>BNAME</u>	<u>CNAME</u>
Moe	Newton
Ruth	Moe
Larry	Larry
Curly	Einstein
Gerig	Curly

**Set Operations:** Reference these sets, and consider three questions (queries).

1. Assume all students from both teams attend a joint meeting. What students attend this meeting? The UNION operation answers this question.

B **UNION** C = {Moe, Larry, Curly,  
Ruth, Gerig, Newton, Einstein}

The result contains all names from both sets B and C, *without duplicate* names. (Moe, Larry, and Curly are members of both teams, but their name only appears once in the result.)



2. Assume only students who play for both teams attend another meeting. What students attend this meeting? The INTERSECT operation answers this question.

B **INTERSECT** C = {Moe, Larry, Curly}

This result contains just those names that are found in both sets.

Before discussing EXCEPT, we note that the UNION and INTERSECT operations are *commutative*. This means that:

B UNION C = C UNION B

B INTERSECT C = C INTERSECT B

3. Consider a meeting of baseball players who are not on the chess team. What students attend this meeting? The EXCEPT operation answers this question.

B **EXCEPT** C = {Ruth, Gerig}

The result contains those names from set B that are not found in set C. Note that B is specified first to the left of the keyword EXCEPT.

Now, consider a meeting of just those chess players who are not on the baseball team. What students attend this meeting? Again, the EXCEPT operation produces the answer. Here, set C is specified first to the left of the keyword EXCEPT.

C EXCEPT B = {Newton, Einstein}

Observe that, unlike UNION and INTERSECT, the EXCEPT operation is not commutative.

B EXCEPT C produces a different result than C EXCEPT B.

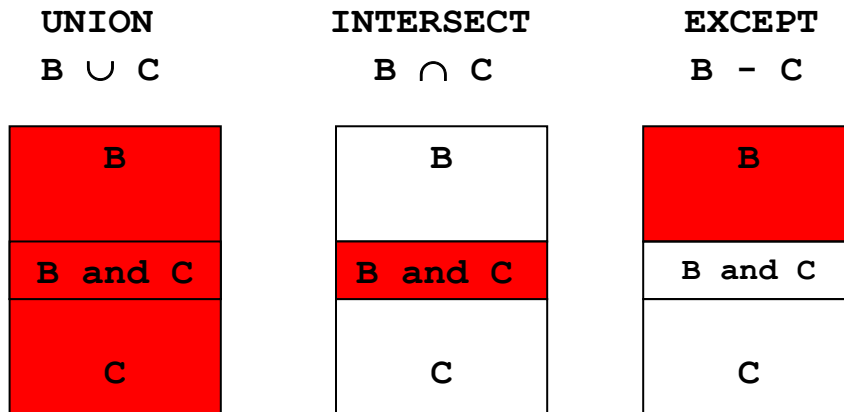
**Conventional Mathematical Notation:** Most math books use the following symbols to represent these set operations.

B  $\cup$  C represents B UNION C  
B  $\cap$  C represents B INTERSECT C  
B - C represents B EXCEPT C

**Venn Diagrams:** You may recall some teacher drawing circular diagrams to represent sets. These diagrams are called Venn Diagrams. Below we make a trivial modification by drawing rectangular Venn Diagrams to represent sets B and C.



Venn diagrams can be used to illustrate set operations. Below, each shaded area represents the result of a set operation.



**Relational Algebra:** Appendix 17B described four of the eight operations defined by Codd's Relational Algebra. These were:

- RESTRICT
- PROJECT
- JOIN
- CROSS PRODUCT

This chapter has introduced three more algebraic operations.

- UNION
- INTERSECT
- EXCEPT

You now know seven of the eight operations that Codd included in his Relational Algebra. This book does not discuss his eighth operation (DIVIDE).

## Appendix 21C: Theory & Efficiency

**Set Theory:** Appendix 4B presented some laws of logic that applied to the Boolean operations (AND, OR, and NOT). Similar laws apply to the set operations (UNION, INTERSECT, and EXCEPT). This appendix considers the two Distributive Laws.

Given three sets SA, SB, and SC.

1. Distributive Law of INTERSECT over UNION:

$$SA \cap (SB \cup SC) = (SA \cap SB) \cup (SA \cap SC)$$

2. Distributive Law of UNION over INTERSECT:

$$SA \cup (SB \cap SC) = (SA \cup SB) \cap (SA \cup SC)$$

**Observations:** A general objective is to generate small intermediate result tables. With this objective in mind, we make some observations.

- Given two arbitrary tables, SA and SB, the intersection of these tables is almost always smaller (has fewer rows) than the union of these tables.

$$\text{SIZE } [SA \cap SB] \leq \text{SIZE } [(SA \cup SB)]$$

Examination of the Venn Diagrams on the previous page illustrates this fact. A special case circumstance where the sizes are equal occurs when both tables contain the same data (SA = SB).

- If SA is smaller than SB, then

$$\text{SIZE } [SA \cap SB] \leq \text{SIZE } [SA]$$

These observations imply that an optimizer might prefer the INTERSECT (versus UNION) operation, especially when one of the tables is considerably smaller than the other table.

**Analysis:** Distributed Law of INTERSECT over UNION

$$SA \cap (SB \cup SC) = (SA \cap SB) \cup (SA \cap SC)$$

Assume SA is very large, and both SB and SC are small. Then the optimizer would probably prefer to have SA participate in just one set operation. Hence, it would decide to implement the left-side expression of this distributed law.

$$SA \cap (SB \cup SC)$$

Alternatively, assume SA is very small, and both SB and SC are large. Then, to avoid having the two large tables participate in a UNION operation that could produce a very large intermediate result, the optimizer might prefer to implement the right-side expression of this law.

$$(SA \cap SB) \cup (SA \cap SC)$$

A similar analysis can be done for the Distributed Law of UNION over INTERSECT.

**Optimizer Query Rewrite:** According to the Distributed Law of INTERSECT over UNION, the following statements are equivalent. Dependent upon the estimated size of intermediate result tables (and other factors), the optimizer may transform one of these statements into the other.

```
SELECT XNO FROM SETA
INTERSECT
(SELECT XNO FROM SETB
UNION
SELECT XNO FROM SETC)
```

```
(SELECT XNO FROM SETA
INTERSECT
SELECT XNO FROM SETB)
UNION
(SELECT XNO FROM SETA
INTERSECT
SELECT XNO FROM SETC)
```

Similar query rewrite can be done based on the Distributed Law of UNION over INTERSECT.

This page is intentionally blank.

# 22

## CASE-Expressions

CASE-expressions are important because they allow you to specify logic that cannot be (directly) specified by the WHERE, GROUP BY, and HAVING clauses, and the join and set operations. Specifically, CASE provides an explicit method for implementing “If-Then” logic. This functionality allows you to satisfy more complex query objectives.

SQL supports two kinds of CASE-expressions. These are:

1. Simple-CASE
2. Searched-CASE

This chapter’s sample queries present the syntax and logic for both variations of CASE.

A CASE-expression is usually specified within a SELECT-clause. However, a CASE-expression may also be specified within a WHERE-clause, HAVING-clause, and ORDER BY clause.

## Simple-CASE

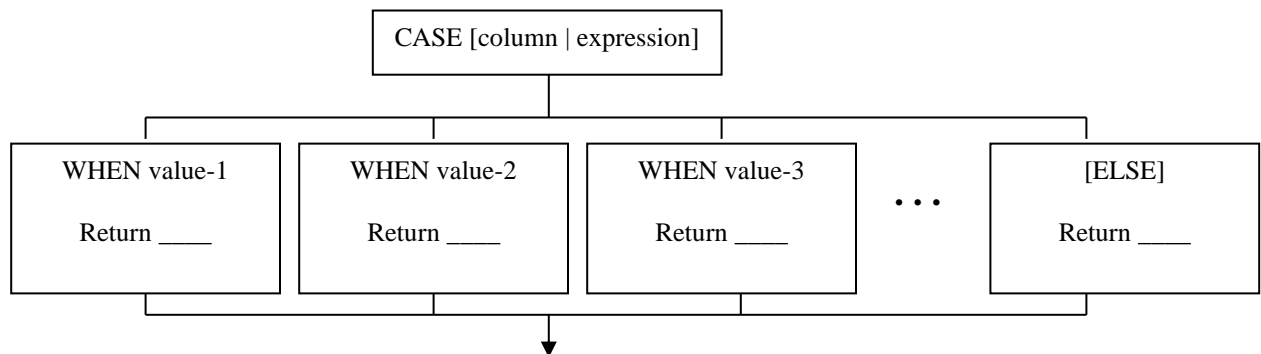
The following example introduces a Simple-CASE expression.

```
Example:  CASE SALARY
            WHEN 1000 THEN 'One Thousand'
            WHEN 2000 THEN 'Two Thousand'
            ELSE 'Not 1000 and Not 2000'
            END
```

This example asks the system to examine the value of SALARY. The first WHEN-condition asks the system to evaluate SALARY = 1000. If this condition is true, the system returns "One Thousand" and exits the CASE-Expression. Otherwise, the next WHEN-condition (SALARY = 2000) is evaluated. If this condition is true, the system returns "Two Thousand" and exits the CASE-Expression. Otherwise, the system returns the value specified in the ELSE-clause, "Not 1000 and Not 2000".

```
Syntax & Logic:  CASE [column | expression]
                   WHEN value-1 THEN return-value-1
                   WHEN value-2 THEN return-value-2
                   . . .
                   [ELSE return-value-n]
                   END
```

The keyword CASE is followed by a column-name or an expression. The value of the column/expression is compared to a series of values specified by WHEN-clauses. If the column/expression equals a WHEN-value, the corresponding return-value is returned, and the CASE-expression is terminated. If there is no match on any WHEN-clause, the return-value specified in the ELSE-clause is returned. The following diagram illustrates the logic of the Simple-CASE expression.



A null value is returned if no WHEN-clause produces a match, and the optional ELSE-clause is omitted.

## Simple-CASE (Column)

**Sample Query 22.1:** For each row in the EMPLOYEE table, display the ENO and ENAME values followed by a textual description of the SALARY value according to the following rule.

- If SALARY = 1000, then display "One Thousand"
- If SALARY = 2000, then display "Two Thousand"
- If SALARY = 9000, then display "Nine Thousand"
- Otherwise, display "Not (1000, 2000, 9000)"

Specify TEXTSALARY as a column-alias for the third column generated by a CASE-expression. Sort the result by ENO values.

```
SELECT ENO, ENAME, CASE SALARY
                        WHEN 1000 THEN 'One Thousand'
                        WHEN 2000 THEN 'Two Thousand'
                        WHEN 9000 THEN 'Nine Thousand'
                        ELSE 'Not (1000, 2000, 9000)'
                        END TEXTSALARY
FROM EMPLOYEE
ORDER BY ENO
```

ENO	ENAME	TEXTSALARY
1000	MOE	Two Thousand
2000	LARRY	Two Thousand
3000	CURLY	Not (1000, 2000, 9000)
4000	SHEMP	Not (1000, 2000, 9000)
5000	JOE	Not (1000, 2000, 9000)
6000	GEORGE	Nine Thousand

**Logic:** The system examines each row in the EMPLOYEE table. For each row, the system displays the ENO and ENAME values followed by a character-string value according the logic designated by the query objective.



## Simple-CASE (Expression)

The next sample query specifies an expression (SALARY+500) immediately after the keyword CASE.

**Sample Query 22.2:** For each row in the EMPLOYEE table, display its ENO and ENAME values, followed by a textual description according to the following rule.

- If SALARY+500 = 1500, then display "1.5K"
- If SALARY+500 = 2500, then display "2.5K"
- If SALARY+500 = 9500, then display "9.5K"
- Otherwise, display "Not (1.K, 2.5K, 9.5K)"

Specify TEXTSALARY as a column-alias for the third column generated by a CASE-expression. Sort the result by ENO values.

```
SELECT ENO, ENAME, CASE SALARY+500
                        WHEN 1500 THEN '1.5K'
                        WHEN 2500 THEN '2.5K'
                        WHEN 9500 THEN '9.5K'
                        ELSE 'Not (1.K, 2.5K, 9.5K)'
                        END TEXTSALARY
FROM EMPLOYEE
ORDER BY ENO
```

ENO	ENAME	TEXTSALARY
1000	MOE	2.5K
2000	LARRY	2.5K
3000	CURLY	Not (1.K, 2.5K, 9.5K)
4000	SHEMP	Not (1.K, 2.5K, 9.5K)
5000	JOE	Not (1.K, 2.5K, 9.5K)
6000	GEORGE	9.5K

**Logic:** The system examines each row in the EMPLOYEE table. For each row, the system displays the ENO and ENAME values, followed by a character-string value according the logic designated by the query objective.

## Searched-CASE

The *Simple-CASE* can only compare on an equals (=) condition. The Searched-CASE is more powerful than the Simple-CASE because it can express more complex conditions that can include any comparison operator (<, >, <=, >=, <>), [NOT] LIKE, [NOT] IN, NOT [BETWEEN], and Boolean connectors (AND, OR, NOT).

**Example:**

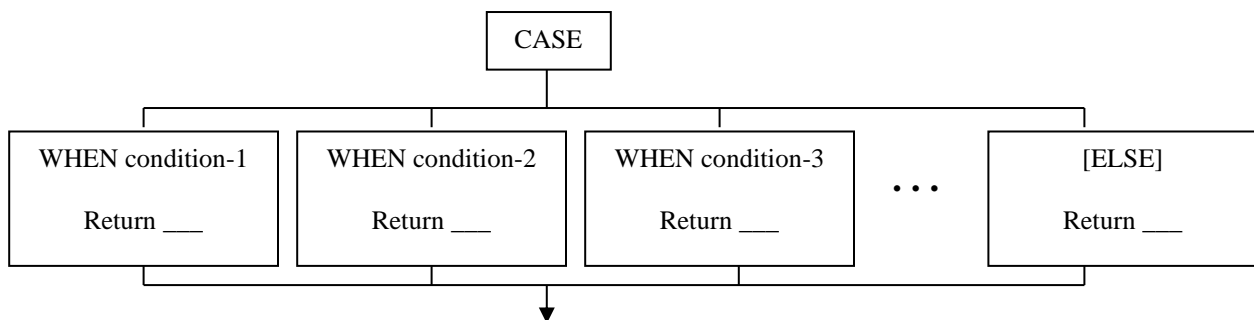
```
CASE
    WHEN SALARY < 1000 THEN 'TOO SMALL'
    WHEN SALARY > 3000 THEN 'TOO BIG'
    ELSE 'OK'
END
```

The first WHEN-clause specifies a condition (SALARY < 1000). If this condition is true, the system returns TOO SMALL and exits the CASE-Expression. Otherwise, the system evaluates the condition associated with the second WHEN-clause (SALARY > 3000). If this condition is true, the system returns TOO BIG and exits the CASE-Expression. Otherwise, the system returns OK, the value specified by the ELSE-clause.

**Syntax & Logic:**

```
CASE
    WHEN condition-1 THEN return-value-1
    WHEN condition-2 THEN return-value-2
    . . .
    [ELSE return-value-n]
END
```

Observe that the keyword CASE is *not* followed by a column-name or expression. Instead, each WHEN-clause specifies a condition. The following diagram illustrates the logic of the Searched-CASE expression.



A null value is returned if no WHEN-clause produces a match, and the optional ELSE-clause is omitted.

The following query objective is the same as Sample Query 21.7.

**Sample Query 22.3:** Display the ENO and ENAME values of all employees. For confidentiality reasons, do not display specific SALARY values. Instead, display a character-string according to the following rule.

- If the SALARY < 1000, then display "TOO SMALL"
- If the SALARY > 3000, then display "TOO BIG"
- Otherwise, display "OK"

Sort the result by ENAME values.

```
SELECT ENO, ENAME, CASE
           WHEN SALARY < 1000 THEN 'TOO SMALL'
           WHEN SALARY > 3000 THEN 'TOO BIG'
           ELSE 'OK'
        END TEXTLABEL
FROM EMPLOYEE
ORDER BY ENAME
```

<u>ENO</u>	<u>ENAME</u>	<u>TEXTLABEL</u>
3000	CURLY	OK
6000	GEORGE	TOO BIG
5000	JOE	TOO SMALL
2000	LARRY	OK
1000	MOE	OK
4000	SHEMP	TOO SMALL

**Syntax:** Unlike the Simple-CASE, the keyword CASE is not followed by a column-name or expression. Each WHEN-clause specifies a condition followed by a corresponding return-value.

**Logic:** The system examines each row in the EMPLOYEE table. For each row, the system displays its ENO and ENAME values, followed by a character-string value. If SALARY is less than 1000, then TOO SMALL is displayed. Otherwise, if SALARY is greater than 30000, then TOO BIG is displayed. Otherwise, OK is displayed.

## Simple-CASE is Unnecessary

Any if-then logic that can be expressed by a Simple-CASE expression can also be expressed by a Searched-CASE expression. We demonstrate this point by rewriting Sample Queries 22.1 and 22.2 using Searched-CASE expressions.

### Sample Query 22.1: Searched-CASE

```
SELECT ENO, ENAME, CASE
           WHEN SALARY = 1000 THEN 'One Thousand'
           WHEN SALARY = 2000 THEN 'Two Thousand'
           WHEN SALARY = 9000 THEN 'Nine Thousand'
           ELSE 'Not (1000, 2000, 9000)'
        END TEXTSALARY
FROM EMPLOYEE
```

### Sample Query 22.2: Searched-CASE

```
SELECT ENO, ENAME, CASE
           WHEN SALARY+500 = 1500 THEN '1.5K'
           WHEN SALARY+500 = 2500 THEN '2.5K'
           WHEN SALARY+500 = 9500 THEN '9.5K'
           ELSE 'Not (1.K, 2.5K, 9.5K)'
        END TEXTSALARY
FROM EMPLOYEE
```

**Conclusion:** Simple-CASE is convenient, but you don't need it.

### **Exercise:**

22A. For every row in the DEPARTMENT table, display a character-string that is derived from its DNO value according to the following rule.

- If DNO = 10, then display DEPARTMENT-10
- If DNO = 20, then display DEPARTMENT-20
- If DNO = 30, then display DEPARTMENT-30
- If DNO = 40, then display DEPARTMENT-40
- Otherwise, display "Some other department"

Also, display each department's BUDGET value. Specify DEPTNO as a column-alias for the first column generated by the CASE-expression. Code two SELECT statements using both variations of CASE.

## Careful! Sequence of WHEN-Clauses

Do not code *WHEN*-clauses in an arbitrary top-to-bottom sequence. The following sample query illustrates a *CASE*-expression that incorrectly specifies the *WHEN*-clauses in the wrong top-to-bottom sequence.

A	B
5	5
5	10
5	-
-	10
-	-

NTAB

**Sample Query 22.4:** For each row in *NTAB*, display the *A* and *B* values followed by a character-string description according the following rule:

Display "A EQUALS B" if  $A = B$   
Display "A IS BIGGER THAN B" if  $A > B$   
Display "B IS BIGGER THAN A" if  $A < B$   
Display "ONLY A IS NULL" if *A* is null and *B* is not null  
Display "ONLY B IS NULL" if *B* is null and *A* is not null  
Display "BOTH VALUES ARE NULL" if *A* is null and *B* is null  
Otherwise, display "Something Strange Happened!"

### Incorrect Solution

```
SELECT A, B,
       CASE
         WHEN A = B      THEN 'A EQUALS B'
         WHEN A > B     THEN 'A IS BIGGER THAN B'
         WHEN A < B     THEN 'B IS BIGGER THAN A'
         → WHEN A IS NULL THEN 'ONLY A IS NULL'
         WHEN B IS NULL THEN 'ONLY B IS NULL'
         WHEN A IS NULL AND B IS NULL THEN 'BOTH VALUES ARE NULL'
         ELSE 'Something Strange Happened! '
       END INCORRECT
FROM NTAB
```

A	B	INCORRECT
5	5	A EQUALS B
5	10	B IS BIGGER THAN A
5	-	ONLY B IS NULL
-	10	ONLY A IS NULL
-	-	ONLY A IS NULL ← Error

**Logic:** Observe that the last row in the result table is wrong. Both columns A and B contain null values, but the text incorrectly displays "ONLY A IS NULL".

In the Incorrect Solution, the error occurs because the last WHEN-condition (WHEN A IS NULL AND B IS NULL THEN...) is never tested. This happens because the fourth WHEN-condition (WHEN A IS NULL THEN...) evaluates to True. Hence, the fourth comment (ONLY A IS NULL) is displayed and the CASE-expression is terminated. Termination implies that the last two WHEN-conditions and the ELSE-clause are not evaluated.

The following statement specifies the WHEN-clauses in the correct top-to-bottom sequence. Here, WHEN A IS NULL AND B IS NULL is "moved up" to be specified before the WHEN A IS NULL clause and the WHEN B IS NULL clause.

### Correct Solution

```
SELECT A, B,
       CASE
         WHEN A = B      THEN 'A EQUALS B'
         WHEN A > B     THEN 'A IS BIGGER THAN B'
         WHEN A < B     THEN 'B IS BIGGER THAN A'
         → WHEN A IS NULL AND B IS NULL THEN 'BOTH VALUES ARE NULL'
         WHEN A IS NULL THEN 'ONLY A IS NULL'
         WHEN B IS NULL THEN 'ONLY B IS NULL'
         ELSE 'Something Strange Happened! '
       END CORRECT
FROM NTAB
```

<u>A</u>	<u>B</u>	<u>CORRECT</u>
5	5	A EQUALS B
5	10	B IS BIGGER THAN A
5	-	ONLY B IS NULL
-	10	ONLY A IS NULL
-	-	BOTH VALUES ARE NULL

### **Exercise:**

22B. Reference the REGION table. For each row, display a two-character code for the RNO value followed by the value of the CLIMATE column. Character codes for the RNO values are: 1 = NE, 2 = NW, 3 = SE, 4 = SW, and 5 = MW. Specify RCODE as the column-alias for the column generated by the CASE-expression. Code two SELECT statements using both variations of CASE.

## Substitute for Null Values

Sample Query 11.13a displayed the NTAB table where it used the COALESCE function to substitute a real (non-null) value for each null value. The following SELECT statement presents an alternative solution using a Searched-CASE to satisfy the same query objective.

A	B
5	5
5	10
5	-
-	10
-	-

NTAB

**Sample Query 22.5:** Display all data in the NTAB table after making the following substitutions.

- Substitute 6 for any null value in column A.
  - Substitute 9 for any null value in column B.
- Specify AA as a column alias for the first column.  
Specify BB as a column-alias for the second column.

```
SELECT CASE WHEN A IS NULL THEN 6
          ELSE A
        END AA,
       CASE WHEN B IS NULL THEN 9
          ELSE B
        END BB
FROM NTAB
```

AA	BB
5	5
5	10
5	9
6	10
6	9

**Syntax:** This statement specifies two short CASE-expressions. Sometimes a short CASE-expression can be coded on a single line as illustrated below.

```
SELECT CASE WHEN A IS NULL THEN 6 ELSE A END AA,
       CASE WHEN B IS NULL THEN 9 ELSE B END BB
FROM NTAB
```

Some users would find this code easier to understand. However, using the COALESCE function is probably simpler than any variation of CASE.

## Merging Values from Multiple Columns

The following sample query displays a single value from one of two columns according to the following rule.

**Sample Query 22.6:** For each row in NTAB, display:

- A if A is greater than or equal to B.
- B if A is less than B.
- B if A is null and B is not null
- A if B is null and A is not null
- -1 if both values are null.

Specify MERGED as a column-alias.

A	B
5	5
5	10
5	-
-	10
-	-

NTAB

```
SELECT
    CASE WHEN A >= B THEN A
         WHEN A < B THEN B
         WHEN A IS NULL AND B IS NOT NULL THEN B
         WHEN B IS NULL AND A IS NOT NULL THEN A
         ELSE -1
    END MERGED
FROM NTAB
```

```
MERGED
5
10
5
10
-1
```

**Syntax and Logic:** Nothing new.

### Exercise:

22C. Reference the NTAB table. Only consider rows where both the A and B columns contain non-null values. For each such row, display the A and B values followed by:

- "EQUAL VALUES" if A is equal to B
- "NON-EQUAL VALUES" if A is not equal to B

Specify NOTNULL as a column alias for the result which should look like:

```
A B NOTNULL
5 5 EQUAL VALUES
5 10 NON-EQUAL VALUES
```



## CASE References a Built-in Function

Both variations of CASE allow a WHEN-clause to compare a value to a result that is returned by a built-in function.

**Sample Query 22.7:** Reference the EMPLOYEE table. You are told that the sum of all SALARY values is 16,900, and you want to verify this information. Write a SELECT statement that summarizes all SALARY values. If this summary value equals 16,900, display TOTAL SALARY IS CORRECT. Otherwise, display PROBLEM WITH TOTAL SALARY.

### Simple-CASE

```
SELECT
  CASE SUM (SALARY)
    WHEN 16900 THEN 'TOTAL SALARY IS CORRECT'
    ELSE 'PROBLEM WITH TOTAL SALARY'
  END RESULT
FROM EMPLOYEE
```

### Searched-CASE

```
SELECT
  CASE
    WHEN SUM (SALARY) = 16900 THEN 'TOTAL SALARY IS CORRECT'
    ELSE 'PROBLEM WITH TOTAL SALARY'
  END RESULT
FROM EMPLOYEE
```

### RESULT

```
TOTAL SALARY IS CORRECT
```

**Simple-CASE:** The keyword CASE is immediately followed by the SUM function that returns a value. This value is referenced within the WHEN-clause.

**Searched-CASE:** The SUM function is specified within a WHEN-clause.

## CASE Specified as an Argument to a Function

A CASE-expression can be specified as an argument to a built-in function.

**Sample Query 22.8:** Display the sum of all employee salaries after making the following substitutions for SALARY values.

- Substitute 2500 for each 2000 value
- Substitute 3500 for each 3000 value
- Substitute 8500 for each 9000 value
- Leave other values unchanged

Specify ADJTOTAL as a column-alias.

### Simple-CASE

```
SELECT SUM (CASE SALARY
            WHEN 2000 THEN 2500
            WHEN 3000 THEN 3500
            WHEN 9000 THEN 8500
            ELSE SALARY
            END) ADJTOTAL
FROM EMPLOYEE
```

### Searched-CASE

```
SELECT SUM (CASE
            WHEN SALARY = 2000 THEN 2500
            WHEN SALARY = 3000 THEN 3500
            WHEN SALARY = 9000 THEN 8500
            ELSE SALARY
            END) ADJTOTAL
FROM EMPLOYEE
```

ADJTOTAL  
17900

**Syntax:** Both statements specify a CASE-expression as an argument to a SUM function.

The next two sample queries are similar to the preceding two sample queries. The only difference is that each CASE-expression references a summary total generated by a GROUP BY clause.

**Sample Query 22.9:** For each department referenced in the EMPLOYEE table, display its DNO value followed by a textual description of the department's total salary according to the following rules.

- If the total departmental salary is 14000, display "14K"
- If the total departmental salary is 2400, display "2.4K"
- Otherwise, display "NEITHER 14K NOR 2.4K"

Simple-CASE

```
SELECT DNO, CASE SUM (SALARY)
           WHEN 14000 THEN '14K'
           WHEN  2400 THEN '2.4K'
           ELSE 'NEITHER 14K NOR 2.4K'
        END TOTSAL
FROM EMPLOYEE
GROUP BY DNO
```

Searched-CASE

```
SELECT DNO, CASE
           WHEN SUM (SALARY) = 14000 THEN '14K'
           WHEN SUM (SALARY) =  2400 THEN '2.4K'
           ELSE 'NEITHER 14K NOR 2.4K'
        END TOTSAL
FROM EMPLOYEE
GROUP BY DNO
```

<u>DNO</u>	<u>TOTSAL</u>
10	2.4K
20	14K
40	NEITHER 14K NOR 2.4K

**Syntax & Logic:** Nothing new.

In the previous sample query, the CASE-expression referenced the SUM function. In the following sample query, the SUM function references a CASE-expression.

**Sample Query 22.10:** Reference the EMPLOYEE table and make the following substitutions for each SALARY value.

- Substitute 2500 for each 2000 value
- Substitute 3500 for each 3000 value
- Substitute 8500 for each 9000 value
- Substitute 1000 for any other value

Then, display the total department salary for each department.

Simple-CASE

```
SELECT DNO, SUM (CASE SALARY
                  WHEN 2000 THEN 2500
                  WHEN 3000 THEN 3500
                  WHEN 9000 THEN 8500
                  ELSE 1000
                  END) TOTSAL
FROM EMPLOYEE
GROUP BY DNO
```

Searched-CASE

```
SELECT DNO, SUM (CASE
                  WHEN SALARY = 2000 THEN 2500
                  WHEN SALARY = 3000 THEN 3500
                  WHEN SALARY = 9000 THEN 8500
                  ELSE 1000
                  END) TOTSAL
FROM EMPLOYEE
GROUP BY DNO
```

<u>DNO</u>	<u>TOTSAL</u>
10	3500
20	14500
40	1000

**Syntax & Logic:** Nothing new.

## Hiding Confidential Summary Totals

When a user examines a result table containing statistical summaries, in some circumstances, this person may be able to make deductions about the underlying raw data that was used to produce the statistical summaries. This could be problematic if the raw data is confidential. For example, assume that individual SALARY values are confidential, and consider the following statement.

```
SELECT DNO, COUNT (*) EMPCT, SUM (SALARY) TOTALSALARY
FROM EMPLOYEE
GROUP BY DNO
```

<u>DNO</u>	<u>EMPCT</u>	<u>TOTALSALARY</u>
10	2	2400.00
20	3	14000.00
40	1	500.00

Because Department 40 only has one employee, you can deduce that this employee has salary of \$500.00. Also, if you work in Department 10, and your salary is \$400.00, you can deduce that your coworker earns \$2,000.00.

We will assume (perhaps unrealistically) that you cannot deduce the salary of any employee who works in a department that has three or more employees. Below we consider three modifications to the above SELECT statement that only displays the total salary of departments with three or more employees. The third modification probably generates the most desirable result.

**Modification-1:** Use the HAVING-clause to display information about just those departments having more than two employees.

```
SELECT DNO, COUNT (*) EMPCT, SUM (SALARY) TOTALSALARY
FROM EMPLOYEE
GROUP BY DNO
HAVING COUNT (*) > 2
```

<u>DNO</u>	<u>EMPCT</u>	<u>TOTALSALARY</u>
20	3	14000.00

A potential shortcoming for this result is that it does not display any information about Departments 10 and 40. The next two modifications use CASE to display rows for Departments 10 and 40 while hiding their total departmental salaries.

**Modification-2:** Use CASE to display a total salary of 0.00 for any department with less than three employees.

```
SELECT DNO, COUNT (*) EMPCT,  
       CASE  
         WHEN COUNT (*) < 3 THEN 0.00  
         ELSE SUM (SALARY)  
       END TOTSALARY  
FROM EMPLOYEE  
GROUP BY DNO
```

<u>DNO</u>	<u>EMPCT</u>	<u>TOTSALARY</u>
10	2	0.00
20	3	14000.00
40	1	0.00

This solution may be better, but it introduces another potential problem. Someone who examines the result may conclude that Departments 10 and 40 only hire employees who work for free. ☺

**Modification-3:** Use CASE to display the "CONFIDENTIAL" message instead of the total salary for any department with less than three employees.

```
SELECT DNO, COUNT (*) EMPCT,  
       CASE  
         WHEN COUNT (*) < 3 THEN 'CONFIDENTIAL'  
         ELSE CAST (SUM (SALARY) AS CHAR(10))  
       END TOTSALARY  
FROM EMPLOYEE  
GROUP BY DNO
```

<u>DNO</u>	<u>EMPCT</u>	<u>TOTSALARY</u>
10	2	CONFIDENTIAL
20	3	14000.00
40	1	CONFIDENTIAL

The CAST function is specified to convert SUM (SALARY) to a character-string because "CONFIDENTIAL" is a character-string.

## Important Observation about CASE

Previous sample queries illustrated that the specification of CASE within a SELECT-clause allows you to specify logic that is applied *after* all other logical processing associated with the WHERE, JOIN-ON, GROUP BY, HAVING clauses and set operations.

Consider the following SELECT statement that does not contain a CASE-expression. All logic pertaining to row selection, join, and grouping operations is implemented via the WHERE, JOIN-ON, GROUP BY, and HAVING clauses.

Query Objective: Reference to the PART and PARTSUPP tables. You are only interested in parts that you can purchase from multiple suppliers. For any such part that is greater than 15 pounds, display the part number and name, followed by the maximum and minimum prices, and the difference in these prices that you could pay suppliers for the part. Do not display information about any part where there is no difference between the maximum and minimum prices.

```
SELECT P.PNO, P.PNAME,
       MAX (PS.PSPRICE) MAXPRICE,
       MIN (PS.PSPRICE) MINPRICE,
       MAX (PS.PSPRICE) - MIN (PS.PSPRICE) DIFFERENCE
FROM PART P INNER JOIN PARTSUPP PS ON P.PNO = PS.PNO
WHERE PWT >= 15
GROUP BY P.PNO, P.PNAME
HAVING COUNT (*) >= 2
AND MAX (PS.PSPRICE) > MIN (PS.PSPRICE)
ORDER BY P.PNO
```

PNO	PNAME	MAXPRICE	MINPRICE	DIFFERENCE
P1	PART1	11.00	10.50	0.50
P3	PART3	12.50	12.00	0.50
P5	PART5	11.00	10.00	1.00
P7	PART7	3.50	2.00	1.50
P8	PART8	5.00	3.00	2.00

The following extension to the logic of this query requires the specification of a CASE-expression.

Assume the business user is not interested in the specific statistical values displayed in the previous result. Instead, this user prefers to see one of three different narrative comments about the difference between maximum and minimum values, such as:

PNO	PNAME	MAXMINDIFFERENCE
P1	PART1	LESS THAN OR EQUAL TO \$1.00
P3	PART3	LESS THAN OR EQUAL TO \$1.00
P5	PART5	LESS THAN OR EQUAL TO \$1.00
P7	PART7	GREATER THAN \$1.00, BUT LESS THAN \$2.00
P8	PART8	GREATER THAN OR EQUAL TO \$2.00

Including a CASE-expression in the preceding SELECT-statement realizes this objective. (Observe there is no change to the logic embodied within the WHERE, JOIN-ON, GROUPING and HAVING clauses.)

```
SELECT P.PNO, P.PNAME,
       CASE
         WHEN MAX(P.S.PSPRICE) - MIN (P.S.PSPRICE) >= 2.00
          THEN 'GREATER THAN OR EQUAL TO $2.00'
         WHEN MAX(P.S.PSPRICE) - MIN (P.S.PSPRICE) > 1.00
          THEN 'GREATER THAN $1.00, BUT LESS THAN $2.00'
         ELSE 'LESS THAN OR EQUAL TO $1.00'
       END MAXMINDIFFERENCE
FROM PART P INNER JOIN PARTSUPP PS ON P.PNO = PS.PNO
AND PWT >= 15
GROUP BY P.PNO, P.PNAME
HAVING COUNT(*) >= 2
AND MAX(P.S.PSPRICE) > MIN(P.S.PSPRICE)
ORDER BY P.PNO
```

**Summary Observation:** Specifying CASE within a SELECT-clause allows you to implement If-Then logic that is applied *after* all other logical processing associated with row selection, join-operations, grouping, and summarizing.



## Exercises

- 22D. Reference the EMPLOYEE table. Consider the total of all SALARY values in this table. If this total is less than 10,000, display "SMALL TOTAL SALARY". If this total exceeds 20,000, display "LARGE TOTAL SALARY". Otherwise, display "OK SALARY". The result should look like:

```
TEXTMSG  
OK SALARY
```

- 22E. Make the following substitution and then calculate the total of all SALARY values in the EMPLOYEE table. For each SALARY value that is less than 1,000, substitute 1,000 for that value. The result should look like:

```
ADJUSTEDSALARY  
18000.00
```

- 22F. Reference the EMPLOYEE table. Assume that all ENAME values are unique. Display three summary totals:
- (i) The total of all employee salaries.
  - (ii) The total of all employee salaries assuming that MOE has been fired. (MOE's SALARY value is zero).
  - (iii) The total of all employee salaries assuming that both LARRY and CURLY have been fired. (Both SALARY values are zero.)

The result should look like:

```
ALLEMPLOYEES      NOMOE      NOLARRYCURLY  
16900.00  14900.00      11900.00
```

- 22G. Reference the PRESERVE table. Display the state code and total acreage for all preserves in any state having a total acreage that exceeds 15,000 acres. If a state has less than or equal to 15,000 acres, display the state code followed by a character-string stating "LESS THAN OR EQUAL TO 15000 ACRES". The result should look like:

```
STATE      SUM (ACRES)  
AZ          51360  
MA          LESS THAN OR EQUAL TO 15000 ACRES  
MT          16931
```

## CASE in the WHERE-Clause

A CASE-expression is usually specified within a SELECT-clause. There are not many situations where you will need to specify a CASE-expression in a WHERE-clause. However, the following sample query presents an example where specifying CASE in a WHERE-clause is useful.

**Sample Query 22.11:** Reference the EMPLOYEE table. Do not display information about any employee with a SALARY value of 2000.00. For other employees, display the ENO, ENAME, SALARY, and ratio of SALARY/(SALARY-2000.00) if this ratio is greater than or equal to 2.00. (Note: There is a divide-by-zero problem when a SALARY value equals 2000.00.)

```
SELECT ENO, ENAME, SALARY, SALARY/(SALARY - 2000.00) RATIO
FROM EMPLOYEE
WHERE (CASE
      WHEN SALARY = 2000.00 THEN 0.0
      ELSE SALARY/(SALARY - 2000.00)
      END) >= 2.00
```

ENO	ENAME	SALARY	RATIO
3000	CURLY	3000.00	3.00

**Logic:** For each EMPLOYEE row, the CASE-expression returns some value. The WHEN-condition returns a value of 0.0 when a SALARY equals 2000.00; otherwise, it returns the ratio of SALARY/(SALARY - 2000.00). If the returned value exceeds 2.00, the corresponding ENAME, SALARY, and ratio values are displayed.

For example, consider the two rows describing MOE and LARRY who have SALARY values of 2000.00. These rows match the WHEN-condition and return 0.0, a value that is less than 2.00. Hence, data about MOE and LARRY are not displayed.

Now consider the other four rows describing employees with SALARY values that are not equal to 2000.00. These rows are tested by the ELSE-clause. Only CURLY's salary of 3000.00 causes the system to evaluate  $3000.00/(3000.00 - 2000.00) = 3.0$  which exceeds 2.00. Hence, data about CURLY is displayed.

**Alternative Solutions:** To be presented in Exercises 26Q and 27Q.

## CASE in the ORDER BY Clause

A CASE-expression can be specified within an ORDER BY clause.

Assume your system does not support the NULLS FIRST or NULLS LAST options for the ORDER BY clause as described at the end of Chapter 11. Therefore, when you specify an ORDER BY clause, you have to adopt a do-it-yourself approach to position null values at the top/bottom of a row sequence.

**Sample Query 22.12a:** Assume you are using a system (e.g., DB2) where, by default, null values sort high. Display all information in the NTAB table. Sort the result by Column A where null values will sort low.

A	B
5	5
5	10
5	-
-	10
-	-

NTAB

```
SELECT A, B
FROM   NTAB
ORDER BY CASE WHEN A IS NULL THEN 0
          ELSE 1
          END,
A;
```

A	B
-	10
-	-
5	5
5	10
5	-

**Logic:** The ORDER BY clause references two columns. The CASE-expression generates the first column that will contain a 0 or 1 where a 0 is associated with a null value in column A. (These 0/1 values are not displayed in the result.) The second column is Column A. Hence, the sort sequence is based upon:

0/1	A
0	-
0	-
1	5
1	5
1	5

**Sample Query 22.12b** Assume you are using a system (e.g., SQL Server) where, by default, null values sort low. Display all information in the NTAB2 table. Sort the result by Column A where null values will sort high.

A	B
10	-
15	10
-	30
-	10
40	40
-	-

**NTAB2**

```
SELECT A, B
FROM NTAB2
ORDER BY CASE WHEN A IS NULL THEN 1
           ELSE 0
           END,
         A;
```

A	B
10	-
15	10
40	40
-	30
-	10
-	-

**Logic:** Same as preceding sample query. Here, a value of 1 is associated with each null A value.

## Summary

CASE provides an explicit method for implementing "If-Then" logic. This is an important feature and significantly enhances the power of SQL.

There are two kinds of CASE-expressions: (1) the Simple-CASE and (2) the Searched-CASE. The Searched-CASE is more useful because it can express more complex conditions that may include any comparison operator (<, >, <=, >=, <>), [NOT] LIKE, [NOT] IN, [NOT] BETWEEN, or Boolean connector (AND, OR, NOT).

## Summary Exercises

Specify CASE-Expressions to satisfy the following query objectives.

22H. This exercise has the same query objective as Exercise 21H. Reference the PROJ1PARTS1 table. Produce a result that displays every part number and name, followed by a character-string indicating if the QTY column contains a value that is less than, equal to, or greater than 16. Sort the result by PNO. The result should look like:

<u>PNO</u>	<u>PNAME</u>	<u>SIZE</u>
P1	PART1	EQUAL TO 16
P2	PART2	EQUAL TO 16
P4	PART4	GREATER THAN 16
P5	PART5	LESS THAN 16

22I. This exercise has the same query objective as Sample Query 11.13b. Reference the NTAB table. Calculate the grand total of all values using the two cross-tabulation patterns. (1) Summarize the subtotals of column values. (2) Summarize the subtotals of row values. Substitute 6 for any null value in column A, and substitute 9 for any null value in column B. The result should look like:

<u>GRANDTOTAL1</u>	<u>GRANDTOTAL2</u>
70	70

22J. This exercise is a variation on Exercise 22F. For each department referenced in the EMPLOYEE table, display the department number followed by three summary totals: (i) The total of each departmental salary assuming that MOE will be fired. (ii) The total of each departmental salary assuming that LARRY will be fired. (iii) The total of each departmental salary assuming that CURLY will be fired. The result should look like:

<u>DNO</u>	<u>SUMWITHOUTMOE</u>	<u>SUMWITHOUTLARRY</u>	<u>SUMWITHOUTCURLY</u>
10	2400.00	400.00	2400.00
20	12000.00	14000.00	11000.00
40	500.00	500.00	500.00

22K. This exercise extends the preceding Exercise 22J. Display a final row in the result table that contains the grand totals of all salaries. The result should look like:

<u>DNO</u>	<u>WITHOUTMOE</u>	<u>WITHOUTLARRY</u>	<u>WITHOUTCURLY</u>
10	2400.00	400.00	2400.00
20	12000.00	14000.00	11000.00
40	500.00	500.00	500.00
TOTAL	14900.00	14900.00	13900.00

Hint: Consider the UNION ALL operation. Also, regarding the first column, note that DNO contains integer values, but "TOTAL" is a character string.

22L. Reference the EMPLOYEE table. For each department that has at least one employee, display the department number and average salary followed by a comment that indicates if this departmental average is less than, equal to, or greater than the overall average salary of all employees. Sort the result by department numbers. The result should look like:

<u>DNO</u>	<u>AVGSAL</u>	<u>COMMENTARY</u>
10	1200.00	LESS THAN OVERALL DEPARTMENTAL AVERAGE
20	4666.66	GREATER THAN OVERALL DEPARTMENTAL AVERAGE
40	500.00	LESS THAN OVERALL DEPARTMENTAL AVERAGE

22M. This exercise is a variation of the preceding Exercise 22L. Address the circumstance where a department may have only one or two employees, allowing for the deduction of confidential individual salaries. For each department that has at least one employee, display the department number and a count of the number of employees who work in the department. If the department has more than two employees, display a comment indicating if the departmental average is less than, equal to, or greater than the overall average salary of all employees. Otherwise, if the department only has one or two employees, the comment should state "CONFIDENTIAL". The result should look like:

<u>DNO</u>	<u>EMPCT</u>	<u>COMMENTARY</u>
10	2	CONFIDENTIAL
20	3	GREATER THAN OVERALL DEPARTMENTAL AVERAGE
40	1	CONFIDENTIAL

22N. Pivot a table: This is an optional and very challenging exercise. This exercise asks you to use CASE to "pivot" (or "rotate") tabular data into a spreadsheet format. Again, we recommend using your front-end tool for this kind of report formatting. Also, some database vendors provide special purpose built-in functions (e.g., PIVOT) that can pivot tabular data. [These functions are not covered in this book. They may be presented in a future edition.]

Query Objective: Represent the following PARTSUPP table in a spreadsheet format as illustrated below. Assume you know that supplier numbers range from S1 to S8.

PARTSUPP Table

Spreadsheet Format

<u>PNO</u>	<u>SNO</u>	<u>PSPRICE</u>		<u>S1</u>	<u>S2</u>	<u>S3</u>	<u>S4</u>	<u>S5</u>	<u>S6</u>	<u>S7</u>	<u>S8</u>
P5	S1	10.00	P1	0.00	10.50	0.00	11.00	0.00	0.00	0.00	0.00
P1	S2	10.50	P3	0.00	0.00	12.00	12.50	0.00	0.00	0.00	0.00
P5	S2	10.00	P4	0.00	0.00	0.00	12.00	0.00	0.00	0.00	0.00
P7	S2	2.00	P5	10.00	10.00	0.00	11.00	0.00	0.00	0.00	0.00
P3	S3	12.00	P6	0.00	0.00	0.00	4.00	0.00	4.00	0.00	4.00
P1	S4	11.00	P7	0.00	2.00	0.00	3.00	3.50	3.50	0.00	0.00
P3	S4	12.50	P8	0.00	0.00	0.00	5.00	0.00	4.00	0.00	3.00
P4	S4	12.00									
P5	S4	11.00									
P6	S4	4.00									
P7	S4	3.00									
P8	S4	5.00									
P7	S5	3.50									
P6	S6	4.00									
P7	S6	3.50									
P8	S6	4.00									
P6	S8	4.00									
P8	S8	3.00									

Hint: Form groups of PNO values. Display PNO followed by eight summaries generated by eight SUM functions, one for each SNO value. Each SUM function should be similar to that shown below.

```
SUM (CASE WHEN SNO = 'S1' THEN PSPRICE ELSE 0 END) S1
```



This page is intentionally blank.

---

# PART VI

## Sub-SELECTs

In Chapter 21, the UNION, INTERSECT, and EXCEPT keywords were specified between two Sub-SELECTs. In the following Chapters 23-28, sample queries will illustrate that a Sub-SELECT can be specified within other SQL clauses as illustrated below.

```
SELECT _____  
FROM _____  
WHERE _____ = (SELECT _____  
                   FROM _____  
                   WHERE _____) } Sub-SELECT
```

```
SELECT _____  
FROM (SELECT _____  
      FROM _____  
      WHERE _____) } Sub-SELECT  
WHERE _____
```

Again, a Sub-SELECT generates an intermediate result table that is referenced by the containing "Outer-SELECT" statement.

Comment: Application developers who have written programs in traditional programming languages (e.g., COBOL, C++, JAVA) should notice that a Sub-SELECT is analogous to a subprogram.

## Part VI: Chapter Topics

Chapter 23 - Regular Sub-SELECT: This chapter introduces the "regular" Sub-SELECT. We use of the term "regular" to indicate that the Sub-SELECT is not a special kind of Sub-SELECT called a "correlated" Sub-SELECT (to be introduced in Chapter 25).

Chapter 24 - Sub-SELECT in DML: This chapter is a continuation of Chapter 15 which introduced the DML statements: INSERT, UPDATE, and DELETE. This chapter will illustrate the specification of Sub-SELECTs within DML statements.

Chapter 25 - Correlated Sub-SELECT: This chapter introduces a special variation of Sub-SELECT called a "correlated" Sub-SELECT. Unlike regular Sub-SELECTs, which are relatively straightforward, correlated Sub-SELECTs involve more complex logic.

In Chapters 23-25, Sub-SELECTs return intermediate result tables that are not assigned names. In Chapters 26-27, Sub-SELECTs return intermediate result tables that are assigned names. These names are referenced in the Outer-SELECT.

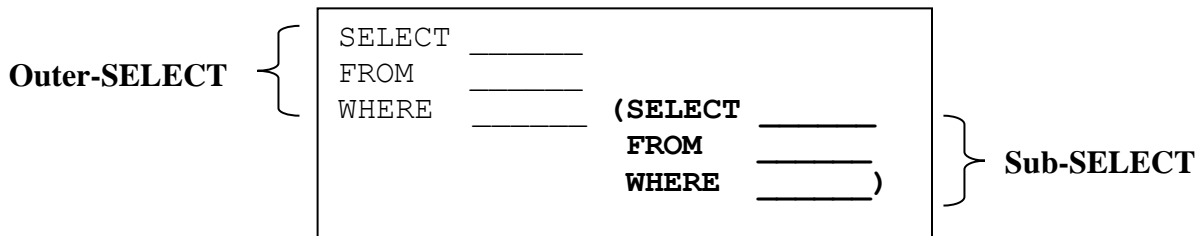
Chapter 26 - Inline Views: An inline View is a Sub-SELECT that is specified within a FROM-clause.

Chapter 27 - WITH-Clause: A WITH-Clause specifies a Sub-SELECT to define a "Common Table Expression" (CTE). The WITH-clause provides the same functionality as an inline view, plus additional functionality that will be described in this chapter.

Chapter 28 - CREATE VIEW Statement: This chapter introduces the CREATE VIEW Statement which is part of SQL's Data Definition Language.

## “Regular” Sub-SELECT

This chapter introduces the regular Sub-SELECT. “Regular” is an unofficial term used to indicate that a Sub-SELECT is not a “correlated” Sub-SELECT (to be introduced in Chapter 25.) In this chapter, each Sub-SELECT is nested within an “Outer-SELECT” as illustrated below.



We begin by previewing Sample Query 23.1 which executes:

```

SELECT *
FROM EMPLOYEE
WHERE SALARY = (SELECT MIN (SALARY) FROM EMPLOYEE)
  
```

We suspect that your intuition will guide you to the correct interpretation of this statement. You are invited to predict the result before reading Sample Query 23.1.

Although a Sub-SELECT is usually specified within a WHERE-clause, it can also be specified within other clauses.

Sample Query 23.15 will specify a Sub-SELECT within a HAVING-clause that looks like:

```
SELECT _____  
FROM _____  
GROUP BY _____  
HAVING _____ (SELECT _____  
                    FROM _____  
                    WHERE _____)
```

Sample Query 23.16 will specify a Sub-SELECT within a SELECT-clause that looks like:

```
SELECT _____, (SELECT _____  
                  FROM _____  
                  WHERE _____)  
FROM _____  
WHERE _____
```

Sample Query 23.17 will specify multiple Sub-SELECTs within a CASE-expression that looks like:

```
SELECT _____,  
       CASE  
         WHEN _____ = (SELECT _____  
                          FROM _____  
                          WHERE _____)  
         THEN _____  
         WHEN _____ = (SELECT _____  
                          FROM _____  
                          WHERE _____)  
         THEN _____  
         ELSE _____  
       END  
FROM _____  
WHERE _____
```

## “Shape” of Intermediate Result Table

Executing a Sub-SELECT generates an intermediate result. We are very interested in the “shape” (another unofficial term) of this intermediate result. An intermediate result can look like one of three general shapes as illustrated below.

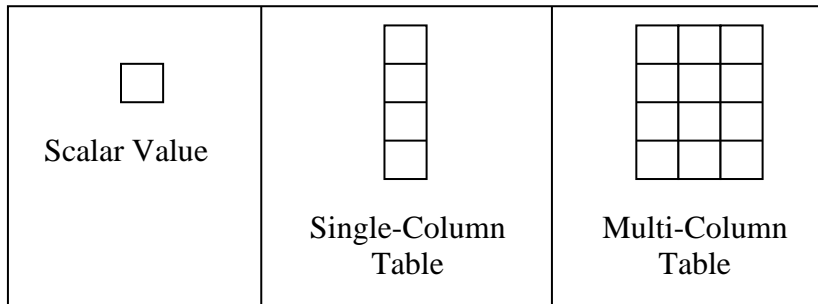


Figure 23.1: “Shapes” of Intermediate Results

Sample Queries 23.1-23.4 will illustrate Sub-SELECTs that return a single (scalar) value as an intermediate result. Although this intermediate result looks like a single value, it is really a table with just one row and one column.

Sample Queries 23.5-23.12 will illustrate Sub-SELECTs that return multiple values. This “list of values” is really a table with a single column.

Sample Queries 23.13-23.14 will illustrate Sub-SELECTs that return a multi-column table as an intermediate result.

## Sub-SELECT Returns a Single (Scalar) Value

The next four sample queries illustrate Sub-SELECTs that return a single value as an intermediate result. This value is referenced by a WHERE-clause in the Outer-SELECT.

**Sample Query 23.1:** Reference the EMPLOYEE table. Display all information about any employee who earns the lowest salary.

```
SELECT *  
  
FROM EMPLOYEE  
  
WHERE SALARY = (SELECT MIN (SALARY) FROM EMPLOYEE)
```

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
5000	JOE	400.00	10

**Syntax:** The Sub-SELECT must be enclosed within parentheses. In general, a Sub-SELECT may be as complex as any SELECT statement.

**Logic:** The Sub-SELECT is executed and returns the minimum SALARY (400.00) as an intermediate result. Then the Outer-SELECT reduces to:

```
SELECT *  
FROM EMPLOYEE  
WHERE SALARY = 400.00
```

Execution of this statement returns the final result. The final result would contain multiple rows if multiple employees had the same minimum salary of \$400.00.

This query objective required the Outer-SELECT and Sub-SELECT to reference the same table (EMPLOYEE). Forthcoming query objectives will require the Outer-SELECT and Sub-SELECT to reference different tables.

### Exercise:

23A. Display all information about any employee who earns the largest salary.

**Sample Query 23.2:** Reference the EMPLOYEE table. Display all information about any employee whose salary is less than the overall average salary.

```
SELECT *  
  
FROM EMPLOYEE  
  
WHERE SALARY < (SELECT AVG (SALARY) FROM EMPLOYEE)
```

ENO	ENAME	SALARY	DNO
1000	MOE	2000.00	20
2000	LARRY	2000.00	10
4000	SHEMP	500.00	40
5000	JOE	400.00	10

**Syntax:** Nothing new. In this example, the WHERE-clause specifies a less-than (<) comparison operator.

**Logic:** The Sub-SELECT is executed and returns a single-value (2816.66) as an intermediate result. Then the Outer-SELECT reduces to:

```
SELECT *  
FROM EMPLOYEE  
WHERE SALARY < 2816.66
```

Execution of this statement produces the final result.

**Special-Case Scenario:** Recall that aggregate functions (e.g., SUM, MAX, MIN, AVG) return a null value if the function references a column containing all null values. Therefore, for the previous two examples, you should ask: Is SALARY declared as NOT NULL? If yes, then you do not have to worry about null-value problems. If no, then you should determine if there are any circumstances where all SALARY values could be null. If yes, then the function would return a null value leading to "no rows returned" as a final result.

**Exercise:**

23B. Display all information about any employee whose salary exceeds the overall average salary.



**Sample Query 23.3:** Reference the EMPLOYEE table. Display all information about any employee who works in Department 10 and earns the largest salary in that department.

```
SELECT *  
  
FROM EMPLOYEE  
  
WHERE DNO = 10  
  
AND SALARY = (SELECT MAX (SALARY)  
  
FROM EMPLOYEE  
  
WHERE DNO = 10)
```

ENO	ENAME	SALARY	DNO
2000	LARRY	2000.00	10

**Syntax:** Nothing new.

**Logic:** The Sub-SELECT is executed and returns a single value (2000.00). Then the Outer-SELECT reduces to:

```
SELECT *  
FROM EMPLOYEE  
WHERE DNO = 10  
AND SALARY = 2000.00
```

Execution of this statement returns the final result. Note that multiple EMPLOYEE rows would be displayed if multiple employees in Department 10 earned the same largest salary.

**Important Exercise:**

23C. Be careful with your logic. Note that the above Sample Query 23.3 specified the same WHERE-clause in the Sub-SELECT and the Outer-SELECT. Is this an unnecessary redundancy?

- (a) What is the final result if you *only* specify "WHERE DNO=10" in the Sub-SELECT?
- (b) Also, what is the final result if you *only* specify "WHERE DNO=10" in the Outer-SELECT?

The next sample query illustrates that a Sub-SELECT and Outer-SELECT can reference different tables.

**Sample Query 23.4:** Display all information out any department having a budget that exceeds the total salary of all employees.

```
SELECT *  
  
FROM DEPARTMENT  
  
WHERE BUDGET > (SELECT SUM (SALARY) FROM EMPLOYEE)
```

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>
10	ACCOUNTING	75000.00
20	INFO. SYS.	20000.00
40	ENGINEERING	25000.00

**Syntax:** Nothing new.

**Logic:** The Sub-SELECT returns a single value (16,900.00). Then, the outer-SELECT reduces to:

```
SELECT *  
FROM DEPARTMENT  
WHERE BUDGET > 16900.00
```

Execution of this statement returns the final result.

**Exercise:**

23D. Display the name and salary of any employee who has a salary that exceeds the smallest BUDGET value in the DEPARTMENT table.

## Bottom-Up versus Top-Down Reading of Sub-SELECTs

Assume you are about to analyze a Sub-SELECT that was written by another user. There are two basic approaches to reading this Sub-SELECT, bottom-up and top-down. Understanding both approaches can help you understand another user's Sub-SELECT and also help you code your own Sub-SELECTs.

Narrative descriptions of previous sample queries presented a bottom-up analysis because *our initial focus was on the Sub-SELECT*.

**Bottom-Up Analysis:** Review our analysis of the following statement presented in Sample Query 23.3. Here, the Sub-SELECT initially produced the largest SALARY (2000.00) for employees in Department 10. Note that our initial focus was on the Sub-SELECT. Thereafter our focus shifted to the Outer-SELECT.

```
SELECT *
FROM EMPLOYEE
WHERE DNO = 10
AND SALARY =
      ↑ (SELECT MAX (SALARY)
        FROM EMPLOYEE
        WHERE DNO = 10)
```

**Top-Down Analysis:** A top-down analysis begins with the Outer-SELECT. The above statement is read as:

```
↓ SELECT *
   FROM EMPLOYEE
   WHERE DNO = 10
   AND SALARY = "some unknown value"
```

A Sub-SELECT is coded to return the desired unknown value.

Either the bottom-up or top-down approach can be used to analyze a regular Sub-SELECT. Chapter 25 will show that the top-down approach is usually more relevant with correlated Sub-SELECTs.

## Know Shape of Intermediate Result

Consider the following query objective: Display all information about LARRY's department. (There is a potential problem with this query objective. What is it?) Given the current contents of the EMPLOYEE table, executing the following statement produces the correct result.

```
SELECT * FROM DEPARTMENT
WHERE DNO = (SELECT DNO FROM EMPLOYEE
            WHERE ENAME = 'LARRY')
```

However, we emphasize that, at some point in the future, this same statement could produce an error. Examination of the EMPLOYEE table shows that LARRY works in Department 10. Consider what would happen if the organization hired another LARRY who is assigned to Department 30. In this circumstance, the Sub-SELECT would return two different DNO values. Then the WHERE-clause in the Outer-SELECT would reduce to:

```
WHERE DNO = (10, 30)      ← Error
```

This is an *invalid* WHERE-clause because it specifies an equals sign (=) to compare DNO with multiple values. Therefore, when you analyze this query objective, you should ask if two employees could have the same name. You should presume this could happen because the ENAME column is not defined as PRIMARY KEY or UNIQUE. You could resolve this problem by re-articulating the query objective to search for a specific LARRY by referencing his primary-key value (2000).

```
SELECT * FROM DEPARTMENT
WHERE DNO = (SELECT DNO FROM EMPLOYEE
            WHERE ENO = 2000)
```

What if you don't know the ENO value? Then you could revise the query objective to display all information about every department that hires an employee named LARRY. This change requires the Outer-SELECT to specify the keyword IN.

```
SELECT * FROM DEPARTMENT
WHERE DNO IN (SELECT DNO FROM EMPLOYEE
            WHERE ENAME = 'LARRY')
```

The next three sample queries specify IN within the Outer-SELECT because the Sub-SELECT could return multiple values.

## IN: Sub-SELECT Returns a Column of Values

The following seven sample queries illustrate Sub-SELECTs that return a single column with multiple values. Under this circumstance, the WHERE-clause in the Outer-SELECT must specify IN or NOT IN as outlined below.

```
WHERE _____ [NOT] IN (Sub-SELECT)
```

**Sample Query 23.5:** Reference the DEPARTMENT and EMPLOYEE tables. Display all information about every department that has at least one employee.

```
SELECT *  
  
FROM DEPARTMENT  
  
WHERE DNO IN (SELECT DNO FROM EMPLOYEE)
```

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>
10	ACCOUNTING	75000.00
20	INFO. SYS.	20000.00
40	ENGINEERING	25000.00

**Syntax:** The keyword IN is specified before the Sub-SELECT. Specifying an equals-sign (=) would cause an error.

**Logic:** The Sub-SELECT returns multiple values. If you were to execute the Sub-SELECT as an independent statement, the result would look like:

```
DNO  
20  
10  
20  
40  
10  
20
```

After substituting these values for the Sub-SELECT, the Outer-SELECT reduces to:

```
SELECT *  
FROM DEPARTMENT  
WHERE DNO IN (20, 10, 20, 40, 10, 20)
```

Observe that some values (20 and 10) have duplicates. Because these duplicate values are meaningless, the system effectively removes them from the intermediate result. Then the Outer-SELECT reduces to:

```
SELECT *
FROM DEPARTMENT
WHERE DNO IN (20, 10, 40)
```

Execution of this statement returns the final result. Observe that the result does not include Department 30, the only department without any employees.

**Alternative Solution:** The following statement uses a join-operation to satisfy this query objective. DISTINCT is specified because duplicate DNO, DNAME, and BUDGET values occur for those departments that have multiple employees.

```
SELECT DISTINCT D.DNO, D.DNAME, D.BUDGET
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.DNO = D.DNO
```

**Exercise:**

23E. Reference the REGION and STATE tables in the MTPCH database. Display the name of every REGION that is related to some row in the STATE table. Specify a Sub-SELECT in your solution.

## IN versus Inner-Join

The following query objective is the same as Sample Query 16.6 that specified a join-operation.

**Sample Query 23.6:** Display the employee number and name of every employee who works for a department having a budget that is greater than or equal to \$25,000.00.

```
SELECT ENO, ENAME
FROM EMPLOYEE
WHERE DNO IN (SELECT DNO
              FROM DEPARTMENT
              WHERE BUDGET >= 25000.00)
```

```
ENO  ENAME
-----
2000 LARRY
4000 SHEMP
5000 JOE
```

**Syntax & Logic:** Nothing new. The Sub-SELECT returns (10, 40). Then the Outer-SELECT reduces to:

```
SELECT ENO, ENAME
FROM EMPLOYEE
WHERE DNO IN (10, 40)
```

**Alternative Solution:** Shown in Sample Query 16.6.

```
SELECT E.ENO, E.ENAME
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.DNO = D.DNO
AND D.BUDGET >= 25000.00
```

Form your own opinion about the relative friendliness of the coding a Sub-SELECT versus a join-operation. (Appendix 23A will discuss efficiency tradeoffs between Sub-SELECTs and join-operations.)

### Exercise:

23F. Reference the REGION and STATE tables. Display the name of any state that is located in the NORTHEAST region. Specify a Sub-SELECT in your solution.

23G. Review: Use join-operations to solve the preceding Exercises 23E and 23F.

**Sample Query 23.7:** Only consider departments that have at least one employee. Display the DNO, DNAME, and BUDGET values for any such department with a budget that is greater than or equal to \$25,000.00.

```
SELECT DNO, DNAME, BUDGET
FROM DEPARTMENT
WHERE BUDGET >= 25000.00
AND DNO IN (SELECT DNO FROM EMPLOYEE)
```

DNO	DNAME	BUDGET
10	ACCOUNTING	75000.00
40	ENGINEERING	25000.00

**Syntax & Logic:** Nothing new.

**Alternative Solution:** The following solution specifies a join-operation. DISTINCT is specified because duplicate DNO, DNAME, and BUDGET values occur for those departments that have multiple employees.

```
SELECT DISTINCT D.DNO, D.DNAME, D.BUDGET
FROM EMPLOYEE E, DEPARTMENT D
WHERE E.DNO = D.DNO
AND D.BUDGET >= 25000.00
```

**Exercise:**

23H. Reference the REGION and STATE tables. Display the name of any region with a CLIMATE of "Hot" and is related to some state. Code two solutions using (i) a Sub-SELECT and (ii) a join-operation.

**Important Exercise:**

23I. In the commentary for Sample Query 17.3.2, we considered the following query objective and concluded that it could not be satisfied by coding a join-operation:

Reference the DEPARTMENT and EMPLOYEE tables. Display the overall total budget of those departments which have at least one employee.

Satisfy this query objective by coding a Sub-SELECT.



## NOT IN

The next sample query specifies "NOT IN (Sub-SELECT)".

**Sample Query 23.8:** Display the DNO, DNAME and BUDGET values for any department that does not have any employees.

```
SELECT DNO, DNAME, BUDGET
FROM DEPARTMENT
WHERE DNO NOT IN (SELECT DNO FROM EMPLOYEE)
```

DNO	DNAME	BUDGET
30	PRODUCTION	7000.00

**Syntax:** Nothing new.

**Logic:** The Sub-SELECT returns all DNO values from the EMPLOYEE.DNO column. After removing duplicate values, the Outer-SELECT becomes:

```
SELECT DNO, DNAME, BUDGET
FROM DEPARTMENT
WHERE DNO NOT IN (10, 20, 40)
```

Execution of this statement returns the final result.

**Alternative Solutions:** Sample Query 25.5 will present another solution using NOT EXISTS.

Also, for tutorial reasons, Exercise 23.Zg will invite you to code a very roundabout (and obviously inefficient) solution that specifies a join-operation and a set-operation.

### Exercise:

23J. Reference the REGION and STATE tables. Display the name of any region that is not associated with a state.

**Sample Query 23.9:** Display the ENO, ENAME, SALARY, and DNO values of any employee who does not work in a department with a budget that is greater than \$22,000.00.

```
SELECT ENO, ENAME, SALARY, DNO
FROM EMPLOYEE
WHERE DNO NOT IN (SELECT DNO
                  FROM DEPARTMENT
                  WHERE BUDGET > 22000.00)
```

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
1000	MOE	2000.00	20
3000	CURLY	3000.00	20
6000	GEORGE	9000.00	20

**Syntax & Logic:** Nothing new. The Sub-SELECT returns 10 and 40. The outer-SELECT reduces to:

```
SELECT ENO, ENAME, SALARY, DNO
FROM EMPLOYEE
WHERE DNO NOT IN (10, 40)
```

Execution of this statement returns the final result.

**Alternative Solution:** This query objective can be articulated in a more positive form: "Display the ENAME, SALARY, and DNO values of any employee who works for a department having a budget that is less than or equal to \$22,000.00." This observation leads us to another solution.

```
SELECT ENO, ENAME, SALARY, DNO FROM EMPLOYEE
WHERE DNO IN (SELECT DNO FROM DEPARTMENT
             WHERE BUDGET <= 22000.00)
```

**Exercise:**

23K. Reference the EMPLOYEE table. You are asked to display all information about any employee who is not assigned to some department. The following statement produces the correct result.

```
SELECT * FROM EMPLOYEE
WHERE DNO NOT IN (SELECT DNO FROM DEPARTMENT)
```

However, why does this statement constitute a "silly" solution?

## Sub-SELECTs and Join-Operations

The next two sample queries demonstrate that a Sub-SELECT and Outer-SELECT can specify a join-operation.

**Sample Query 23.10:** Reference the PART, SUPPLIER, and PARTSUPP tables in the MTPCH database. Display the part number and name of any part that you can purchase from SUPPLIER2 (i.e., SNAME value is SUPPLIER2). Code a Sub-SELECT that specifies a join-operation.

```
SELECT PNO, PNAME
FROM   PART
WHERE  PNO IN
      (SELECT PS.PNO
       FROM  SUPPLIER S, PARTSUPP PS
       WHERE S.SNO = PS.SNO
       AND   S.SNAME = 'SUPPLIER2')
```

<u>PNO</u>	<u>PNAME</u>
P1	PART1
P5	PART5
P7	PART7

**Sample Query 23.11:** Reference the PART, SUPPLIER, and PARTSUPP tables in the MTPCH database. Display the part number, name, and price for any part that you can purchase from SUPPLIER2. Code a join-operation in the Outer-SELECT.

```
SELECT P.PNO, P.PNAME, PS.PSPRICE
FROM  PARTSUPP PS, PART P
WHERE PS.PNO = P.PNO
AND   PS.SNO IN (SELECT SNO
                 FROM  SUPPLIER
                 WHERE  SNAME = 'SUPPLIER2')
```

<u>PNO</u>	<u>PNAME</u>	<u>PSPRICE</u>
P1	PART1	10.50
P5	PART5	10.00
P7	PART7	2.00

**Important Observation:** The SNAME column is not declared as a UNIQUE column. Hence, the above statements coded IN to account for the possibility that two or more suppliers could have the same name.

**Alternate Solutions:** Exercise 23.Zf invites you to code three-table join solutions for the above sample queries.

## Multi-level (Nested) Sub-SELECTs

A Sub-SELECT may specify another Sub-SELECT.

**Sample Query 23.12:** Same as Sample Query 23.10. Display the part number and name of any part that you can purchase from SUPPLIER2.

```
SELECT PNO, PNAME
FROM   PART
WHERE  PNO IN
      (SELECT PNO
       FROM   PARTSUPP
       WHERE  SNO IN
            (SELECT SNO
             FROM   SUPPLIER S
             WHERE  SNAME = 'SUPPLIER2') )
```

<u>PNO</u>	<u>PNAME</u>
P1	PART1
P5	PART5
P7	PART7

**Logic:** The system executes the lowest level (innermost) Sub-SELECT, returning the SNO value (S2) for SUPPLIER2. (Again, we coded IN because we cannot assume that SNAME is unique.) The statement now reduces to:

```
SELECT PNO, PNAME
FROM   PART
WHERE  PNO IN (SELECT PNO
              FROM   PARTSUPP
              WHERE  SNO IN ('S2') )
```

After executing the above Sub-SELECT, the statement becomes:

```
SELECT PNO, PNAME
FROM   PART
WHERE  PNO IN ('P1', 'P5', 'P7')
```

Execution of this SELECT produces the final result.

## Exercises:

- 23L. Reference the STATE and CUSTOMER tables in the MTPCH database. Display the name of any state that does not have at least one customer.
- 23M. Reference the CUSTOMER and PUR\_ORDER tables in the MTPCH database. Display the number and name of any customer who has not purchased any parts (i.e., is not related to any purchase orders).
- 23N. Reference the STATE, CUSTOMER, and PUR\_ORDER tables in the MTPCH database. Display the name of any state that has a customer who has not purchased any parts.
- 23O. Reference the PART, SUPPLIER, and PARTSUPP tables in the MTPCH database. Display the supplier number and name of any supplier who can sell you PART5 (i.e., PNAME value is "PART5"). Code four solutions.
- (a) Code a Sub-SELECT where the Sub-SELECT specifies a two-table join. (Similar to Sample Query 23.10)
  - (b) Code a Sub-SELECT where the Outer-SELECT specifies a two-table join. (Similar to Sample Query 23.11)
  - (c) Code a Sub-SELECT nested within another Sub-SELECT. (Similar to Sample Query 23.12)
  - (d) For review purposes, code a three-table join.
- 23P. Reference the PART, SUPPLIER, and PARTSUPP tables in the MTPCH database. Display the supplier number and name of any supplier who can sell you PART8. (i.e., PNAME value is "PART8".) Also display the price (PSPRICE) the supplier charges for this part. Code two solutions.
- (a) Code a Sub-SELECT where the Outer-SELECT specifies a two-table join. (Similar to Sample Query 23.11)
  - (b) For review purposes, code a three-table join.

## Sub-SELECT Returns Multiple Columns

The following sample query illustrates that a Sub-SELECT may return an intermediate result table with more than one column. [This feature might not be supported on your system.] This sample query references the PROJMGR table described in the beginning of Part V.

<u>PROJMGR</u>				
<u>ENO</u>	<u>PMNAME</u>	<u>MBA</u>	<u>RATE</u>	<u>DNO</u>
1000	MOE	N	500.00	20
2500	DICK	N	100.00	40
6000	GEORGE	Y	10.00	20
4500	DON	N	70.00	40

**Sample Query 23.13:** Display all information about any EMPLOYEE row with ENAME and DNO values that are equal to the corresponding PMNAME and DNO values in PROJMGR.

```
SELECT *
FROM EMPLOYEE
WHERE (ENAME, DNO) IN (SELECT PMNAME, DNO
                       FROM PROJMGR)
```

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
1000	MOE	2000.00	20
6000	GEORGE	9000.00	20

**Syntax:** The Sub-SELECT returns two columns. Hence, the WHERE-clause in the Outer-SELECT must specify two columns that are enclosed within parentheses. The corresponding columns must have compatible data-types.

**Logic:** The Sub-SELECT returns a two-column intermediate result that looks like:

<u>PMNAME</u>	<u>DNO</u>
MOE	20
DICK	40
GEORGE	20
DON	40

The Outer-SELECT produces a final result by comparing each EMPLOYEE row's ENAME and DNO values to the above intermediate result.

Chapters 25-27 will present three alternative solutions for the following sample query. However, after examining these alternative solutions, you might conclude that the following solution is the simplest.

**Sample Query 23.14** Display all information about the highest paid employee in every department that has at least one employee.

```
SELECT *
FROM EMPLOYEE
WHERE (DNO, SALARY) IN (SELECT DNO, MAX (SALARY)
                        FROM EMPLOYEE
                        GROUP BY DNO)
```

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
2000	LARRY	2000.00	10
4000	SHEMP	500.00	40
6000	GEORGE	9000.00	20

**Syntax & Logic:** This Sub-SELECT returns an intermediate result table with two columns that are compatible with the Outer-SELECTs DNO and SALARY data-types. This intermediate result looks like:

<u>DNO</u>	<u>MAX (SALARY)</u>
10	2000.00
20	9000.00
40	500.00

For each EMPLOYEE row selected by the Outer-SELECT, the system compares each pair of DNO and SALARY values to the above pairs of DNO and MAX (SALARY) values. If there is a match, all data from the EMPLOYEE row are displayed.

**Alternative Solutions:** Alternative solutions will be presented in Sample Queries 25.1, 26.3, and 27.3.

## HAVING-Clause Specifies Sub-SELECT

The following sample query illustrates that a Sub-SELECT can be specified within a HAVING-clause.

**Sample Query 23.15:** Reference the EMPLOYEE table. Display the DNO and average departmental salary of any department having an average departmental salary that exceeds the overall average salary for all employees.

```
SELECT DNO, AVG (SALARY)
FROM EMPLOYEE
GROUP BY DNO
HAVING AVG (SALARY) > (SELECT AVG (SALARY)
                        FROM EMPLOYEE)
```

<u>DNO</u>	<u>AVG (SALARY)</u>
20	4666.66

**Logic:** The Sub-SELECT is executed and returns an overall average of 2816.66. The Outer-SELECT then becomes:

```
SELECT DNO, AVG (SALARY)
FROM EMPLOYEE
GROUP BY DNO
HAVING AVG (SALARY) > 2816.66
```

Executing this Outer-SELECT returns the final result.

### Exercises:

23Q: Display all information about the lowest paid employee in each department that has at least one employee.

23R. Reference the EMPLOYEE table. Only consider employees who earn less than \$5,000.00. Display the DNO and minimum employee salary in those departments having a minimal employee salary that exceeds the overall average salary for all employees under consideration.



## SELECT-Clause Specifies Sub-SELECT

The following sample query illustrates a Sub-SELECT specified within a SELECT-clause.

**Sample Query 23.16:** Reference the EMPLOYEE table. Consider the impact of adjusting each employee's salary to a value that is equal to the overall average of all current salaries plus 5% of the employee's current salary. Display each employee's number, name, current salary, and adjusted salary.

```
SELECT ENO, ENAME, SALARY,
       (SELECT AVG (SALARY) FROM EMPLOYEE) + (.05*SALARY) ADJSAL
FROM EMPLOYEE
```

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>ADJSAL</u>
1000	MOE	2000.00	2916.66
2000	LARRY	2000.00	2916.66
3000	CURLY	3000.00	2966.66
4000	SHEMP	500.00	2841.66
5000	JOE	400.00	2836.66
6000	GEORGE	9000.00	3266.66

**Logic:** For each selected row, the Sub-SELECT is executed and returns the overall average salary of 2816.66. This value becomes an intermediate result that is referenced by the Outer-SELECT as shown below.

```
SELECT ENO, ENAME, SALARY, 2816.66 + (.05 * SALARY)
FROM EMPLOYEE
```

Execution of this statement returns the final result.

**Alternative Solutions:** To be presented in Sample Queries 26.5 and 27.5.

### Exercise:

23S. Consider changing each department's BUDGET value to a value that is equal to the overall largest BUDGET value minus 10% of the department's current BUDGET value. Display each department's number, name, current budget, and the adjusted budget.

## CASE-Expression Specifies Sub-SELECTs

The following sample query illustrates a SELECT-clause that contains a CASE-expression where the CASE-expression specifies multiple Sub-SELECTs.

**Sample Query 23.17:** For each department that has at least one employee, display its department number and its average departmental salary followed a textual comment indicating that the departmental average is less than, greater than, or equal to the overall average salary.

```
SELECT DNO, AVG (SALARY) AVGSALARY,
       CASE
         WHEN AVG (SALARY) < (SELECT AVG(SALARY) FROM EMPLOYEE)
          THEN 'LESS THAN OVERALL AVERAGE SALARY'
         WHEN AVG (SALARY) = (SELECT AVG(SALARY) FROM EMPLOYEE)
          THEN 'EQUAL TO OVERALL AVERAGE SALARY'
         ELSE      'GREATER THAN OVERALL AVERAGE SALARY'
       END TEXTCOMMENT
FROM EMPLOYEE
GROUP BY DNO
```

```
DNO AVGSALARY TEXTCOMMENT
---
10  1200.00 LESS THAN OVERALL AVERAGE SALARY
20  4666.66 GREATER THAN OVERALL AVERAGE SALARY
40   500.00 LESS THAN OVERALL AVERAGE SALARY
```

### Exercise:

23T. For each department that has at least one employee, display its department number and maximum departmental salary followed textual comment indicating that departmental maximum value is less than or equal to the overall maximum salary.

## IN and NOT IN with Null Values

Here we go again! Null values make life interesting. Previous sample queries referenced the DEPARTMENT and EMPLOYEE tables where the foreign key (EMPLOYEE.DNO) was declared to be NOT NULL. The absence of null values simplified our logic. We encourage you to review your understanding of null values by trying the following exercise. This exercise references the EMPLOYEE3 table which has one null DNO value.

<u>DEPARTMENT</u>			<u>EMPLOYEE3</u>			
DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO
10	ACCOUNTING	75000.00	1000	MOE	2000.00	99
20	INFO. SYS.	20000.00	2000	LARRY	2000.00	10
30	PRODUCTION	7000.00	3000	CURLY	3000.00	20
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40
			5000	JOE	400.00	10
			6000	GEORGE	9000.00	- ←

**Important Review Exercise:** Predict the result for each of the following four statements. Then, execute each statement. If any prediction is wrong, you are encouraged to read the next two pages very closely.

Hints: Statements S1 and S2 are straightforward. They specify a Sub-SELECT that cannot return a null value because DEPARTMENT.DNO is a primary key. Statements S3 and S4 are more interesting because they contain a Sub-SELECT that returns a null value. Statement S4 is especially tricky.

```
S1:  SELECT ENAME, SALARY
      FROM    EMPLOYEE3
      WHERE   DNO IN (SELECT DNO FROM DEPARTMENT)
```

```
S2:  SELECT ENAME, SALARY
      FROM    EMPLOYEE3
      WHERE   DNO NOT IN (SELECT DNO FROM DEPARTMENT)
```

```
S3:  SELECT DNO, DNAME, BUDGET
      FROM    DEPARTMENT
      WHERE   DNO IN (SELECT DNO FROM EMPLOYEE3)
```

```
S4:  SELECT DNO, DNAME, BUDGET
      FROM    DEPARTMENT
      WHERE   DNO NOT IN (SELECT DNO FROM EMPLOYEE3)
```

**Answers:** Most readers will correctly predict the results for statements S1, S2, and S3. However, many readers will make an incorrect prediction for statement S4.

S1: SELECT ENAME, SALARY, DNO FROM EMPLOYEE3  
WHERE DNO **IN** (SELECT DNO FROM DEPARTMENT)

<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
LARRY	2000.00	10
CURLY	3000.00	20
SHEMP	500.00	40
JOE	400.00	10

S2: SELECT ENAME, SALARY, DNO FROM EMPLOYEE3  
WHERE DNO **NOT IN** (SELECT DNO FROM DEPARTMENT)

<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
MOE	2000.00	99

Note that GEORGE (with null DNO) is not displayed.

S3: SELECT DNO, DNAME, BUDGET FROM DEPARTMENT  
WHERE DNO **IN** (SELECT DNO FROM EMPLOYEE3)

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>
10	ACCOUNTING	75000.00
20	INFO. SYS.	20000.00
40	ENGINEERING	25000.00

S4: SELECT DNO, DNAME, BUDGET  
FROM DEPARTMENT  
WHERE DNO NOT IN (SELECT DNO FROM EMPLOYEE3)

*\*\*\* Result table is empty! - "no rows returned."*

Many readers expect the result table to contain a row for the PRODUCTION department (DNO = 30), the only department without employees. What happened? The following page presents the logic of this statement.

## Careful! NOT IN with Null Values

Consider Statement S4 shown below.

```
SELECT DNO, DNAME, BUDGET
FROM   DEPARTMENT
WHERE  DNO NOT IN (SELECT DNO FROM EMPLOYEE3)
```

After executing the Sub-SELECT, the statement reduces to:

```
SELECT DNO, DNAME, BUDGET
FROM   DEPARTMENT
WHERE  DNO NOT IN (99, 10, 20, 40, null)
```

The DEPARTMENT.DNO column contains 10, 20, 30, and 40. The 10, 20, and 40 values fail to match on the NOT IN condition. Now consider what happens when the system considers the DNO value of 30. Recall that a NOT IN condition implies that each comma is an "implied-AND." (See Chapter 5 - Logic: IN and NOT IN.) Hence, when comparing on 30, the WHERE-clause equates to the following compound-condition.

```
WHERE  30 <> 99      → TRUE
AND    30 <> 10      → TRUE
AND    30 <> 20      → TRUE
AND    30 <> 40      → TRUE
AND    30 <> null    → UNKNOWN ← "the problem"
```

This compound WHERE-clause reduces to UNKNOWN because, in order to be TRUE, all individual conditions must be TRUE. Hence, the row for Department 30 is not selected.

**Conclusion:** \*\*\* *Whenever a Sub-SELECT generates a null value, the NOT IN condition always evaluates to UNKNOWN, implying that the Outer-SELECT always yields a "no rows returned" result.*

If you want to display information about Department 30, you could modify S4 as shown below.

```
SELECT DNO, DNAME, BUDGET
FROM   DEPARTMENT
WHERE  DNO NOT IN (SELECT DNO
                  FROM   EMPLOYEE3
                  WHERE  DNO IS NOT NULL)
```

**Exercise:**

23U. a. Rewrite the following join-operation using a Sub-SELECT.

```
SELECT E3.ENAME, E3.SALARY
FROM   EMPLOYEE3 E3, DEPARTMENT D
WHERE  E3.DNO = D.DNO
```

b. Is the following statement equivalent to the above statement?

```
SELECT ENAME, SALARY FROM EMPLOYEE3
```

## ANY or ALL Reference Sub-SELECT

The keywords ANY and ALL can be used with a comparison operand to compare a value to an intermediate result produced by a Sub-SELECT. However, before presenting examples, we emphasize that *there is no good reason to use these keywords. You can always code an alternative statement that is easier to understand (and probably more efficient)*. We present these keywords in case you encounter them when examining another user's code.

Example-1: Display the DNAME value of any department with a DNO value that is *equal to any* DNO value in the EMPLOYEE table. (I.e., Display the DNAME value of any department that has at least one employee.)

```
SELECT DNAME FROM DEPARTMENT
WHERE DNO = ANY (SELECT DNO FROM EMPLOYEE)
```

Alternative solutions are:

```
SELECT DNAME FROM DEPARTMENT
WHERE DNO IN (SELECT DNO FROM EMPLOYEE);
```

```
SELECT DISTINCT D.DNAME
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.DNO = E.DNO;
```

Example-2: Display the DNAME of any department with a BUDGET value that is *greater than all* EMPLOYEE.SALARY values.

```
SELECT DNAME, BUDGET FROM DEPARTMENT
WHERE BUDGET > ALL (SELECT SALARY FROM EMPLOYEE)
```

If a BUDGET value exceeds all SALARY values, it must exceed the largest value. Hence, an alternative solution is:

```
SELECT DNAME, BUDGET FROM DEPARTMENT
WHERE BUDGET > (SELECT MAX (SALARY) FROM EMPLOYEE)
```

## Summary

The specification of a Sub-SELECT allows you to generate an intermediate result that can be referenced within an Outer-SELECT. This chapter's sample queries illustrate that Sub-SELECTs can help a user satisfy more complex query objectives. Sample queries illustrated that a Sub-SELECT may be specified within a WHERE-clause, HAVING-clause, SELECT-clause, or CASE-expression.

From both a syntax and logic perspective, you must understand the "shape" of the intermediate result generated by the Sub-SELECT.

1. If the Sub-SELECT generates a **scalar result**, the WHERE-clause can specify any of the basic comparison operators (=, <, <=, >, >=, <>).

```
Ex:  SELECT ...
      FROM   ...
      WHERE  COL <= (SELECT AVG(X) FROM ...)
```

2. If the Sub-SELECT generates a **single-column result**, the WHERE-clause must specify IN or NOT IN.

```
Ex:  SELECT ...
      FROM   ...
      WHERE  COLA [NOT] IN (SELECT COLX FROM ...)
```

3. If the Sub-SELECT generates a **multi-column result**, the Outer-SELECT's WHERE-clause must specify IN or NOT IN and the column-names in the Outer-SELECT must be enclosed within parentheses.

```
Ex:  SELECT ...
      FROM   ...
      WHERE  (COLA, COLB) [NOT] IN
              (SELECT COLX, COLY FROM ...)
```



## Summary Exercises

Code Sub-SELECTs for the following Exercises 23V - 23Ze which reference tables in the MTPCH sample database.

- 23V. Display all information about the state with the largest population.
- 23W. Display all information about any state having a population that is less than the overall average population.
- 23X. (i) Display the number and name of every supplier who sells part P6.  
(ii) Display the number and name of every supplier who does not sell part P6.
- 23Y. (i) Display the number and name of every supplier who sells at least one pink part.  
(ii) Display the number and name of every supplier who does not sell any pink parts.
- 23Za. For each region with at least one state, display all information about the state with the lowest population in the region.
- 23Zb. Consider the state with the smallest population in each region that has at least one state. Display the region number and its smallest state population if that population value is less than the overall average population for all states.
- 23Zc. Reference the PARTSUPP table. Determine the overall average PSPRICE value. For each row, display its SNO, PNO, and PSPRICE values, followed by the difference between the PSPRICE and the overall average PSPRICE value. Sort the result by the fourth column. Observe that the fourth column will contain negative values for PSPRICE values that are less than the average. (Hint: Consider specifying a Sub-SELECT in the main SELECT-clause.)
- 23Zd. Modify the above Exercise 23Zc. Only display rows for where the PSPRICE exceeds the average PSPRICE. (Hint: Consider specifying the same Sub-SELECT in the main SELECT-clause and the WHERE-clause.)

23Ze. Reference the DEPARTMENT and EMPLOYEE tables. Revisit Sample Query 17.3.2 where you were asked to summarize a numeric parent-column for the parent-table participating in a parent-child join operation: Only consider those departments that have employees and have a budget that is less than or equal to \$50,000.00. Display the total budget for these departments.

Sample Query 17.3.2 considered following the following "almost correct" (i.e., wrong) "solution."

```
SELECT SUM (DISTINCT D.BUDGET)
FROM DEPARTMENT D, EMPLOYEE E
WHERE D.DNO = E.DNO
AND D.BUDGET <= 50000.00

SUM (DISTINCT D.BUDGET)
45000.00
```

This statement "got lucky" because no two DEPARTMENT rows happened to have the same BUDGET value. Code a SELECT statement that constitutes a correct solution.

The following exercises are presented for review purposes.

23Zf. Review Exercise: Satisfy Sample Queries 23.10 and 23.11 using join-operations.

23Zg. Optional Review Exercise: This is a strange tutorial exercise. Assume you simply did not want to write a statement that contains NOT IN. You are invited you to code a very inconvenient, rather roundabout (and obviously inefficient) solution to Sample Query 23.8 (Display the DNO, DNAME and BUDGET values for any department that does not have any employees.) Generate two intermediate results. The first has the DNO, DNAME and BUDGET values of all departments. The second has the same values for those departments that have employees. Then use EXCEPT to "subtract" the second intermediate result from the first.

The following exercises address some previously described SQL challenges.

23Zh. Review Sample Query 8.3 which described a common error shown below.

```
SELECT *
FROM PRESERVE
WHERE FEE > AVG (FEE)
```

Code a correct SELECT statement to satisfy this query objective.

23Zi. Review the page after Sample Query 7.6 which discussed a potential problem of dividing-by-zero in a calculated condition. There we described two potentially problematic statements.

```
Statement-A:  SELECT PNAME, ACRES/FEE
              FROM   PRESERVE
              WHERE  FEE <> 0 AND ACRES/FEE > 200.00
```

```
Statement-B:  SELECT PNAME, ACRES/FEE
              FROM   PRESERVE
              WHERE  ACRES/FEE > 200.00 AND FEE <> 0
```

Code an alternative equivalent SELECT statement that satisfies this query objective where you are asked to display the PNAME and ratio ACRES/FEE for all preserves where this ratio exceed 200.00.

## Appendix 23A: Efficiency

**Size of Sub-SELECT (Intermediate) Result:** Assume a Sub-SELECT references the TAB2 table which is a very large table. Efficiency problems may occur if this Sub-SELECT returns a large intermediate result. This result may have to be written to a temporary disk area (without an index), and the Outer-SELECT would have to search this disk area.

Good News Scenarios: Although TAB2 is very large, the following three statements specify Sub-SELECTs that return small intermediate results.

-----  
Statement-1:     SELECT \*  
                  FROM TAB1  
                  WHERE COLA = (SELECT SUM (COLX) FROM TAB2)

The intermediate result produced by the SUM function in the Sub-SELECT is only a single scalar value. This tiny intermediate result will incur a trivial storage and processing cost.

-----  
Statement-2:     SELECT \*  
                  FROM TAB1  
                  WHERE COLA IN (SELECT COLB FROM TAB2  
                                  WHERE COLX > 1000)

The intermediate result would be small if the Sub-SELECT's WHERE-clause has good selectivity and only returns a few rows.

-----  
Statement-3:     SELECT \*  
                  FROM TAB1  
                  WHERE COLA IN (SELECT COLX FROM TAB2)

Assume the data dictionary indicates that column TAB2.COLX has many duplicate values. Then the intermediate result could be small.

### Bad News Scenario:

```
Statement-4:  SELECT COLA, COLB, COLC
              FROM  TAB1
              WHERE COLA IN (SELECT COLX FROM TAB2)
```

Assume the data dictionary indicates that column COLX has very few duplicate values. Then the intermediate result will be large.

In this circumstance, your optimizer might be able to avoid storing and processing a large intermediate result by rewriting the statement to use a join-operation.

```
Statement-4a: SELECT T1.COLA, T1.COLB, T1.COLC
              FROM TAB1 T1, TAB2 T2
              WHERE T1.COLA = T2.COLX
```

[Note: Assume TAB2 was a small table. Then the Sub-SELECT in Statement-4 would be fast, and it would produce a small intermediate result. Hence, Statement-4a could be slower than Statement-4.]

**IN versus Join:** Many sample queries in this chapter illustrated an alternative solution that specified an inner-join operation. An inner-join may be more efficient if the Sub-SELECT produces a large intermediate result, and the system can use some fast join method. (Review Appendices 17A, 17B, and 7C which describe efficiency considerations for two-table inner-join operations.)

**Query Rewrite:** In some circumstance, if advantageous, the optimizer may rewrite a Sub-SELECT as a join-operation. Should the user consider a do-it-yourself query rewrite. Usually No. Ideally, the optimizer considers all reasonable options and makes the correct decision regarding query rewrite. However, as previously noted, optimizers are not perfect. Therefore, there may be a few occasions where a user can improve query performance by coding an alternative SELECT statement.

## Sub-SELECT in DML

This chapter describes the specification of Sub-SELECTs within the INSERT, UPDATE, and DELETE statements. (These DML statements were introduced in Chapter 15.) This chapter is optional reading for those users who will only retrieve and never modify data in a table.

If you intend to execute the sample statements presented in this chapter, you must create a table called MYEMP by executing the following CREATE TABLE statement. (This table is not created in the CREATE-ALL-TABLES Scripts.)

```
CREATE TABLE MYEMP
(MYENAME  VARCHAR (25),
 MYSALARY DECIMAL (7,2),
 MYDNO    INTEGER)
```

The MYENAME, MYSALARY, and MYDNO columns have the same data-types as the corresponding ENAME, SALARY, and DNO columns in the EMPLOYEE table. For tutorial reasons, this CREATE TABLE statement does not specify a PRIMARY KEY clause or a UNIQUE clause. Commentary will discuss potential problems that can occur when a table that does not have any unique column(s).

## Sub-SELECT Specified in DML Statements

A Sub-SELECT can be specified within the INSERT, UPDATE, and DELETE statements. The basic syntax for specifying a Sub-SELECT within an INSERT statement is outlined in the following skeleton-code.

```
INSERT INTO _____  
  SELECT _____  
  FROM   _____  
  WHERE  _____
```

Within an UPDATE statement, a Sub-SELECT can be specified within a SET-clause or a WHERE-clause (or both). An outline of the basic syntax is shown below.

```
UPDATE _____  
SET   _____ = (SELECT _____  
                        FROM   _____  
                        WHERE  _____ )  
WHERE _____
```

```
UPDATE _____  
SET   _____ = _____  
WHERE _____ = (SELECT _____  
                        FROM   _____  
                        WHERE  _____ )
```

Within a DELETE statement, a Sub-SELECT can be specified within the WHERE-clause. An outline of the basic syntax is shown below.

```
DELETE FROM _____  
WHERE _____ = (SELECT _____  
                        FROM   _____  
                        WHERE  _____ )
```

## INSERT Specifies Sub-SELECT

Assume you have just created the MYEMP table, and this table is empty. The following INSERT statement copies three columns from all EMPLOYEE rows into MYEMP.

**Sample Statement 24.1:** Copy the ENAME, SALARY, and DNO values from all rows in the EMPLOYEE table into the corresponding columns in the MYEMP table.

```
INSERT INTO MYEMP (MYENAME, MYSALARY, MYDNO)
      SELECT ENAME, SALARY, DNO
      FROM EMPLOYEE
```

**System Response:** The system should respond with a message implying the successful insertion of 6 rows into MYEMP. (On some systems, this response might not designate the number of affected rows.) You can verify the successful INSERT operation by executing:

```
SELECT * FROM MYEMP
```

**Syntax:** INSERT INTO MYEMP is followed by column-names within parentheses, which is followed by the Sub-SELECT. The Sub-SELECT can be any valid SELECT statement (except that some systems will reject an ORDER BY clause). Because this INSERT statement specifies three columns, the Sub-SELECT must specify three columns. The ENAME and MYENAME columns must have compatible data-types; likewise for the SALARY and MYSALARY columns, and the DNO and MYDNO columns.

**Logic:** The Sub-SELECT selects three columns from all EMPLOYEE rows and inserts them into MYEMP. None of the selected columns is declared to be unique. Hence, it is possible that duplicate rows could be stored in MYEMP.



## UPDATE: Sub-SELECT Specified within the SET-Clause

MYEMP now contains:

MYENAME	MYSALARY	MYDNO
MOE	2000.00	20
LARRY	2000.00	10
CURLY	3000.00	20
SHEMP	500.00	40
JOE	400.00	10
GEORGE	9000.00	20

The following UPDATE statement modifies some of these rows.

**Sample Statement 24.2:** Update the MYEMP table. Change the MYENAME and MYSALARY values of all rows having an MYDNO value of 20. Each new MYENAME value should be set to JOSEPHINE. Each new MYSALARY value should be set to the average of all current MYSALARY values.

```
UPDATE MYEMP
SET   MYENAME = 'JOSEPHINE',
      MYSALARY = (SELECT AVG (MYSALARY) FROM MYEMP)
WHERE MYDNO = 20
```

**System Response:** The system should respond with a message implying the successful update of 3 rows. You can verify successful execution of this UPDATE operation by executing:

```
SELECT * FROM MYEMP
```

**Syntax:** Sample Statement 15.4 discussed variations of the SET-clause. This example shows that a column can be set to a value that is returned by a Sub-SELECT.

```
SET Column = (SELECT . . .)
```

The Sub-SELECT must return a single value that is compatible with the data-type of the specified column.

**Logic:** MOE, CURLY, and GEORGE work in Department 20. Hence their names are changed to JOSEPHINE. The Sub-SELECT generates a value of 2816.66. Hence, their MYSALARY values are changed to 2816.66. The following page shows the current contents of the MYEMP table.

## UPDATE: Sub-SELECT Specified within the WHERE-Clause

```
MYEMP now contains:  MYENAME      MYSALARY      MYDNO
                     JOSEPHINE      2816.66       20
                     LARRY          2000.00       10
                     JOSEPHINE      2816.66       20
                     SHEMP          500.00        40
                     JOE            400.00        10
                     JOSEPHINE      2816.66       20
```

Observe that the previous UPDATE operation produced three duplicate rows in MYEMP. The following UPDATE statement makes additional changes to this table. This query objective requires the nesting of Sub-SELECTs.

**Sample Statement 24.3:** Update the MYEMP table. Determine the overall minimal salary and those departments with an employee who earns this minimal salary. Then change the salary of all employees who work in these departments to \$1,500.00. Your statement should account for the possibility that two or more employees who work in different departments may earn the same minimal salary.

```
UPDATE MYEMP
SET   MYSALARY = 1500.00
WHERE MYDNO IN (SELECT MYDNO
                FROM MYEMP
                WHERE MYSALARY =
                    (SELECT MIN (MYSALARY)
                     FROM MYEMP))
```

**System Response:** The system should respond with a message implying the successful update of 2 rows.

**Syntax and Logic:** Nothing New. The one employee (JOE) with lowest salary (\$400.00) works in Department 10. Therefore, the bottom-level Sub-SELECT returns 400.00 causing the upper-level Sub-SELECT to return 10. Hence, the MYSALARY values are change for all (2) employees in Department 10. The code specifies IN because, if multiple employees earn the same minimal salary, and these employees work in different departments, then multiple MYDNO values would be returned.

The following page shows that LARRY and JOE, who work in Department 10, have their salaries changed to 1500.00.

## DELETE: Sub-SELECT Specified within the WHERE-Clause

MYEMP now contains:

<u>MYENAME</u>	<u>MYSALARY</u>	<u>MYDNO</u>
JOSEPHINE	2816.66	20
LARRY	1500.00	10
JOSEPHINE	2816.66	20
SHEMP	500.00	40
JOE	1500.00	10
JOSEPHINE	2816.66	20

The following DELETE statement deletes some rows from MYEMP. These rows are identified by MYDNO values that are returned by a Sub-SELECT.

**Sample Statement 24.4:** Delete the MYEMP row for every employee who works in a department where some departmental employee earns more than \$1,000.00.

```
DELETE FROM MYEMP
WHERE MYDNO IN (SELECT MYDNO
                FROM MYEMP
                WHERE MYSALARY > 1000.00)
```

**System Response:** The system should respond with a message implying successful deletion of 5 rows.

**Syntax:** Nothing New.

**Logic:** The Sub-SELECT returns 10 and 20. Hence, this statement deletes all rows for employees who work in Department 10 and Department 20. After deleting these rows, only one row remains in MYEMP, corresponding to SHEMP, the only employee who works in Department 40. MYEMP table now looks like:

<u>MYENAME</u>	<u>MYSALARY</u>	<u>MYDNO</u>
SHEMP	500.00	40

**Design Comment:** Consider the MYEMP table shown at the top of this page. Observe that you could not delete or change just one of the three rows describing JOSEPHINE because there is no column or combination of columns that is unique. Therefore, whenever you create a table, you are strongly encouraged to designate some column or combination of columns as its primary-key.

## Summary

The specification of a Sub-SELECT within a DML statement is a relatively straightforward extension of the concepts and techniques introduced in Chapter 15 (DML Statements) and Chapter 23 (Regular Sub-SELECTs).

You should be extremely careful with your logic when you execute a DML statement. Storing bad data in a table can become very problematic and especially embarrassing if other users subsequently access this bad data.

Finally, if you are an applications developer who executes DML statements within a production environment, you are *strongly encouraged* to read Chapter 29 on Transaction Processing.

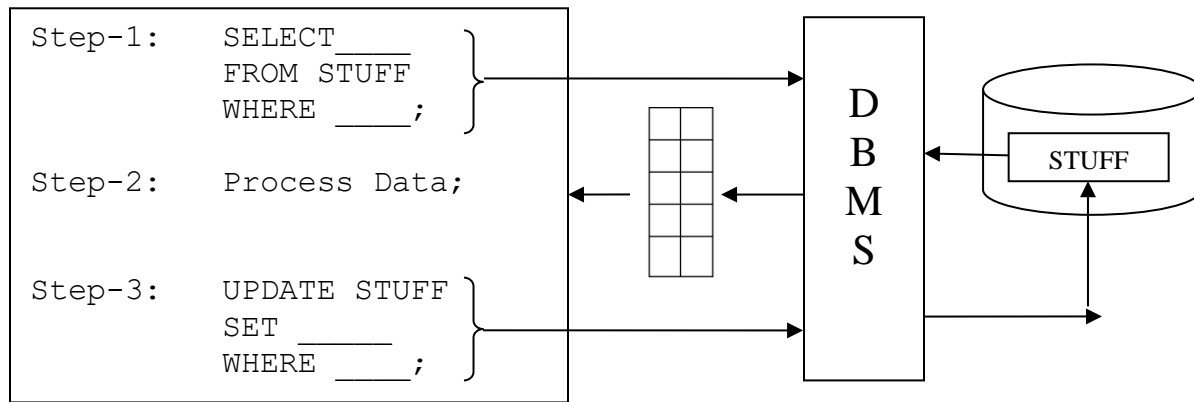
## Summary Exercises

- 24A. Delete all rows from the MYEMP table.
- 24B. Copy the ENAME, SALARY and DNO values from EMPLOYEE into MYEMP. Only copy rows for employees having a salary that is less \$8,000.00.
- 24C. Update the MYEMP table. Change the MYENAME values of all rows having an MYDNO value of 10. All modified MYENAME values should be the same as the name of the MYEMP employee having the largest salary.
- 24D. Delete MYEMP rows corresponding to employees who have the same name as the highest paid employee. Assume that multiple employees can have the same largest salary.
- 24E. Drop the MYEMP table.

## Appendix 24A: Efficiency

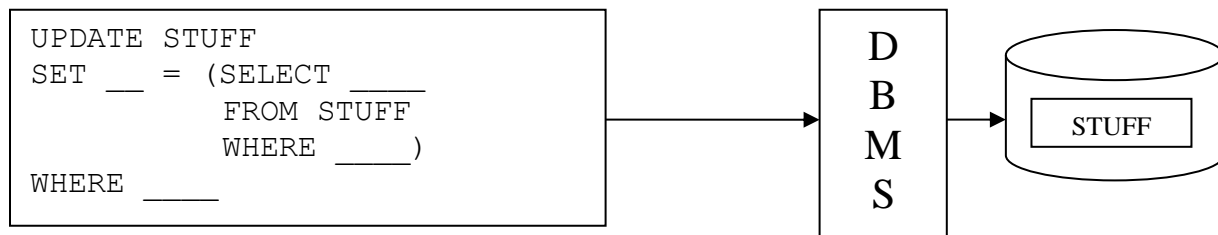
In a production environment, DML statements are usually embedded within application programs or stored procedures written by application developers. The following discussion shows that (sometimes) you can significantly improve efficiency by coding a Sub-SELECT to embody a program's logic within a single SQL statement. Consider the following example where you want to update a table called STUFF.

**Method-A:** The following skeleton-code is specified within an application program or stored procedure.



The above procedure shows three steps: (1) Execute a SELECT statement that returns data to the procedure. (2) Process this data to produce some result. (3) Store this result in the database via an UPDATE operation. This method could be inefficient because (i) two SQL statements are executed, and (ii) a large result could be returned in Step-1 and subsequently processed in Step-2.

**Method-B:** *Sometimes* (but not always) you can embody all the logic and processing within a single UPDATE statement that specifies a Sub-SELECT.



This approach is more efficient because only one SQL statement is executed, all logic and processing are implemented within the database engine, and no data is transferred from the STUFF table to the front-end tool.

## Appendix 24B: Accuracy of Dictionary Statistics

We have seen that the optimizer examines the data dictionary to learn the number of rows in a table and other relevant statistics. Thus far, we have assumed these statistics are accurate. In this appendix, we consider inaccurate statistics, approximately accurate statistics, and statistics that are 100% accurate.

The accuracy of dictionary statistics is considered here because this chapter's sample queries illustrated that a single DML statement can make significant changes to a table. For example, a single INSERT statement can insert multiple rows into a table. (Also, Chapter 15 showed how an UPDATE statement can modify multiple columns in multiple rows, and a DELETE statement can delete multiple rows.) Significant changes to a table, when recorded in the data dictionary, will influence the optimizer. This raises the question about *when* such changes are reflected in dictionary statistics.

In principle, dictionary statistics could be automatically updated after the execution of each INSERT, UPDATE, and DELETE statement. However, this could hurt efficiency. For example, after each INSERT and DELETE operation, the table row-count (and other statistics) would have to be adjusted; and, column statistics would have to be updated for each column, especially if a column is described by a histogram. For this reason, most systems do not update dictionary statistics on a real-time basis. Instead, the DBA periodically executes a utility program that scans database tables and indexes and writes relevant statistical information into the data dictionary.

The following pages present three scenarios where dictionary statistics are (1) inaccurate, (2) approximately accurate, and (3) 100% accurate. Each scenario references one of the following dictionary statistics.

- ROWCOUNT contains the number of rows in a designated table.
- DISTINCTVAL contains the number of distinct values in a designated column.
- MAXCOLVAL contains the largest value in a designated column.

The above statistics have generic names. Your reference manual will present the proper names for your system. This manual will also show that your data dictionary stores many statistical facts. A complete discussion of dictionary statistics is beyond the scope of this book.

## Scenario-1: Inaccurate Statistics

The following scenario illustrates how an inaccurate ROWCOUNT value can influence an optimizer to generate an inefficient application plan.

For tutorial reasons, we assume the optimizer will decide to scan a table if its ROWCOUNT value is less than 200. (This is an oversimplification. The optimizer will also consider the number of rows stored on a physical page and how these pages are distributed across a disk.)

Consider table TAB1 with columns COLA, COLB, and COLC where there is a unique index (XCOLA) on column COLA. Assume TAB1 has 50 rows.

Time-1: The DBA executes a utility program that reads TAB1 and updates relevant data dictionary statistics. The ROWCOUNT value for TAB1 is set to 50.

Time-2: A user executes the following SELECT statement.

```
SELECT * FROM TAB1 WHERE COLA = 25
```

The optimizer decides to scan TAB1 because it is small table. (Its ROWCOUNT value of 50 is less than 200.) This is a good decision that should produce an efficient application plan.

Time-3 - Time-999: Multiple application programs execute INSERT statements that insert a total of 40,000 new rows. *These changes are not recorded in the data dictionary.*

Time-1000: A user executes the SELECT statement similar to that executed at Time-2.

```
SELECT * FROM TAB1 WHERE COLA = 100
```

Again, the optimizer decides to scan TAB1 because it *incorrectly* believes TAB1 is small table. (The ROWCOUT value is still 50.) This is a bad decision that produces an inefficient application plan. If the ROWCOUNT value were set to the actual number of rows (40,050), the optimizer would decide to use the XCOLA index to directly access the desired row. The overall performance penalty could be costly if many similar SELECT statements were executed after Time-1000.

[If reasonable, the DBA should have executed the dictionary update utility program after Time-999.]

## **Scenario-2: Approximately Accurate Statistics**

*In most circumstances, the optimizer only requires data dictionary statistics to be approximately accurate (good enough).*

Consider the MYCUSTOMER table where column CNO is the primary key. The other columns contain demographic data. Three of these other columns are the FOOTSIZE, SEX, and STATECODE columns. The only index on this table is a unique index (XCNO) on the CNO column. Also, the data dictionary does not contain a histogram of values for any column in this table.

Example-1: The optimizer examines a ROWCOUNT statistic.

Time-1: The DBA executes a utility program that updates dictionary statistics for the MYCUSTOMER table. Assume the ROWCOUNT value for MYCUSTOMER is set to 14 million.

Time-2: A user executes the following SELECT statement.

```
SELECT CNO, SEX, FOOTSIZE
FROM MYCUSTOMER
WHERE CNO = 22222222
```

Because the ROWCOUNT value is large (14000000), the optimizer decides to generate an (efficient) application plan that uses the XCNO index to retrieve the one row for Customer 22222222.

Time-3 - Time-999: Many application programs execute INSERT statements that cause the MYCUSTOMER table to grow to about 15 million rows. *This statistical change is not recorded in the data dictionary.*

Time-1000: A user executes the SELECT statement similar to that executed at Time-2.

```
SELECT CNO, FOOTSIZE, STATECODE
FROM MYCUSTOMER
WHERE CNO = 88888888
```

Because the ROWCOUNT value is unchanged (14000000), the optimizer again generates an application plan that uses the XCNO index to retrieve the desired row. This is a good plan that provides the same efficiency advantage as the Time-2 query.



Example-2: The optimizer examines DISTINCTVAL statistics.

Time-1: The DBA executes a utility program that updates data dictionary statistics for the MYCUSTOMER table. Assume that the following statistics are set as:

```
DISTINCTVAL for SEX: 2
DISTINCTVAL for STATECODE: 25
```

Time-2: A user executes a SELECT statement with a WHERE-clause that specifies the following compound-condition.

```
STATECODE = 'NY' OR SEX = 'F'
```

After examining DISTINCTVAL values for the SEX and STATECODE columns, the optimizer deduces that there is a 50% (1/2) chance a hit on the SEX = 'F' condition and only a 4% (1/25) chance of a hit on the STATECODE = 'NY' condition. Hence, the optimizer decides to apply Logical Law 2a [C1 OR C2 = C2 OR C1] to rewrite the above compound-condition. (You may wish to review Appendix 4C.) The above compound-condition is rewritten as:

```
SEX = 'F' OR STATECODE = 'NY'
```

Time-3 - Time-999: Over time, the MYCUSTOMER table grows. Many application programs execute INSERT statements. After Time-9999, the SEX column now has 5 distinct values, and the STATECODE column now has 33 distinct values. *These statistical changes are not recorded in the data dictionary.*

Time-1000: A user executes a SELECT statement with a compound-condition that is similar to the Time-2 condition.

```
STATECODE = 'RI' OR SEX = 'M'
```

The optimizer examines the (unchanged) DISTINCTVAL values for the SEX and STATECODE columns and again, following the same logic, decides to rewrite the above compound-condition as:

```
SEX = 'M' OR STATECODE = 'RI'
```

This rewritten compound-condition provides the same efficiency advantages as the Time-2 condition. If the optimizer had access to the actual number of distinct column values, it would make the same decision because there is a 20% (1/5) chance a hit on the SEX = 'M' condition and only a 3% (1/33) chance of a hit on the STATECODE = 'RI' condition.

### **Scenario-3: Statistic Must be 100% Accurate**

In a few circumstances, the optimizer requires a dictionary statistic to be 100% accurate. One circumstance involves the application of Logical Law 2c [C1 OR TRUE = TRUE] that was described in Appendix 4C.

Time-1: The DBA executes a utility program that updates data dictionary statistics for the MYCUSTOMER table. Assume that the following statistic is set as:

```
MAXCOLVAL for FOOTSIZE: 16.0
```

Time-2: A user executes a SELECT statement with a WHERE-clause that specifies the following compound-condition.

```
SEX = 'F' OR FOOTSIZE <= 17.0
```

After examining the MAXCOLVAL for FOOTSIZE, the optimizer deduces that FOOTSIZE <= 17.0 must be true for all rows. Therefore, the optimizer applies Logical Law 2c to rewrite the compound-condition as:

```
SEX = 'F' OR FOOTSIZE <= 17.0
SEX = 'F' OR TRUE
TRUE
```

Hence, the optimizer formulates an application plan that directs the system to retrieve all MYCUSTOMER rows without expending any effort to examine the SEX and FOOTSIZE values.

Time-3: An application program executes an SQL statement on MYCUSTOMER. The data dictionary is not updated; hence the MAXCOLVAL for FOOTSIZE remains unchanged.

Time-4: A user executes the same Time-2 SELECT statement with the same compound-condition:

```
SEX = 'F' OR FOOTSIZE <= 17.0
```

In this circumstance, the optimizer decides *not* to apply Logical Law 2c because it *may or may not* be the case that, at Time-3, an INSERT or UPDATE operation caused some FOOTSIZE value to become larger than 17.0. Hence the FOOTSIZE <= 17.0 condition may or may not be true for all rows, implying that Law 2c should not be applied.

Conclusion: Before the optimizer can apply Logical Law 2c, it must know that the MAXCOLVAL value for the FOOTSIZE column is 100% accurate. This is not possible if the MYCUSTOMER table could possibly be changed after relevant dictionary statistics have been updated.

This conclusion restricts the optimizer's use of Logical Law 2c. However, there are some important special case circumstances where the optimizer can apply this logical law (and any other logical law that requires a dictionary statistic to be 100% accurate). These circumstances that are described below.

- The optimizer knows that a table is a read-only table, or the entire database is read-only (e.g., data warehouse). Here all DML operations are applied in an off-line environment, and dictionary statistics are updated after these operations. Thereafter, no on-line operation is allowed to change a table.
- The DBA specifies a CHECK clause in the CREATE TABLE statement that specifies the valid values of a column. (Chapter 13 described the CHECK clause.) For example, the following CHECK clause guarantees that every FOOTSIZE value must be less than or equal to 16.0.

```
CHECK (FOOTSIZE <= 16.0)
```

- Some database systems allow the DBA to *optionally* designate the real-time update of dictionary statistics.

## Correlated Sub-SELECTs

This chapter introduces a variation of the Sub-SELECT called the *correlated Sub-SELECT*. *Correlated Sub-SELECTs behave significantly different than regular Sub-SELECTs*. Correlated Sub-SELECTs can be very useful, but their logic is more complex. For this reason, many users attempt to code alternative solutions using other SQL techniques that will be described in the next few chapters. Despite the possibility of coding alternative solutions, you should understand correlated Sub-SELECTs.

We begin by presenting some important preliminary observations that should help you understand the basic syntax and logic of a correlated Sub-SELECT.

Comment for Application Developers: We previously noted that a Sub-SELECT is analogous to a subprogram. Now we note that a regular Sub-SELECT is analogous to a subprogram that does not have an argument. In this chapter, we will see that a correlated Sub-SELECT is analogous to a subprogram with an argument.

## Three Important Preliminary Observations

Consider the following two statements.

```
Regular          SELECT *  
Sub-SELECT:    FROM EMPLOYEE  
                  WHERE SALARY = (SELECT MAX (SALARY)  
                                  FROM EMPLOYEE)
```

```
Correlated     SELECT *  
Sub-SELECT:   FROM EMPLOYEE EX  
                  WHERE SALARY = (SELECT MAX (SALARY)  
                                  FROM EMPLOYEE  
                                  WHERE DNO = EX.DNO)
```

You should already understand the first statement (described in Sample Query 23.1) that specifies a regular Sub-SELECT. The second statement specifies a correlated Sub-SELECT. For the moment, don't worry about its query objective. (Don't try to "understand" it.) Instead, we want to focus on the "mechanical" differences between these two statements.

**Observation-1:** Consider each Sub-SELECT isolated from its Outer-SELECT.

```
Regular Sub-SELECT:    SELECT MAX (SALARY)  
                        FROM EMPLOYEE)
```

```
Correlated Sub-SELECT:  SELECT MAX (SALARY)  
                        FROM EMPLOYEE  
                        WHERE DNO = EX.DNO
```

The above regular Sub-SELECT (not embedded in an Outer-SELECT) will successfully execute and return a result. However, if you attempt to execute the above correlated Sub-SELECT (not embedded in an Outer-SELECT), it will fail because the system will not understand EX.

**\*\*\* Important!** *From a purely mechanical viewpoint, this is how you can always distinguish a regular Sub-SELECT from a correlated Sub-SELECT. A regular Sub-SELECT can always be extracted and independently executed. (This was true for every Sub-SELECT presented in the previous two chapters.) However, if you extract and attempt to execute a correlated Sub-SELECT, it will fail. This observation applies to all correlated Sub-SELECTs presented in this book.*

**Observation-2:** Consider each \*Outer-SELECT isolated from its Sub-SELECT.

Regular Outer-SELECT:           SELECT \*  
                                  FROM EMPLOYEE  
                                  WHERE SALARY = \_\_\_\_\_

Correlated Outer-SELECT:       SELECT \*  
                                  FROM EMPLOYEE **EX**  
                                  WHERE SALARY = \_\_\_\_\_

Within the context of the correlated Sub-SELECT, the EX (table alias) in the Outer-SELECT's FROM-clause is a "correlation-name." We recommend that you always specify a correlation-name whenever you code a correlated Sub-SELECT. However, there are circumstances (Sample Query 25.3) where you can ignore this recommendation.

**Observation-3:** We ask an important question that relates to Sub-SELECT logic: "How many times is the Sub-SELECT executed?"

Regular Sub-SELECT: A regular Sub-SELECT, as illustrated in the previous two chapters, is executed once.

Correlated Sub-SELECT: Logically\*, a correlated Sub-SELECT is executed for each selected row in the outer-table. For example, if the Outer-SELECT retrieves a million rows, then (in principle, and maybe in fact) the correlated Sub-SELECT is executed a million times! Hence, a correlated Sub-SELECT behaves very differently than a regular Sub-SELECT.

\*Physically (under-the-hood), the system may be able to satisfy the query objective without implementing the repetitive execution of the correlated Sub-SELECT. (Appendix 25A offers more insight into this matter.)

## Correlated Sub-SELECT

The following sample query has the same query objective as Sample Query 23.14.

**Sample Query 25.1:** Display all information about the highest paid employee in every department that has at least one employee.

```
SELECT *  
  
FROM EMPLOYEE EX  
  
WHERE SALARY = (SELECT MAX (SALARY)  
  
                FROM EMPLOYEE  
  
                WHERE DNO = EX.DNO)
```

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
2000	LARRY	2000.00	10
4000	SHEMP	500.00	40
6000	GEORGE	9000.00	20

**General Logic:** The system considers the table that is referenced in the Outer-SELECT. This is the EMPLOYEE table shown below. (This same EMPLOYEE table happens to be referenced in the Sub-SELECT. Sample Query 23.3 will show that the Outer-SELECT and Sub-SELECT may reference different tables.)

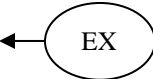
<u>EMPLOYEE</u>			
<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
1000	MOE	2000.00	20
2000	LARRY	2000.00	10
3000	CURLY	3000.00	20
4000	SHEMP	500.00	40
5000	JOE	400.00	10
6000	GEORGE	9000.00	20

For each EMPLOYEE row, the Outer-SELECT's WHERE-clause effectively asks: "Is the SALARY value in *this* row equal to the highest salary for this employee's department?" (The Sub-SELECT finds the highest salary for the department.) If the answer is "yes," the row is placed in the result table. Otherwise, the row is not placed in the result table. The following pages present a row-by-row detail description of this process.

**Detail Logic:** We walk through the processing of each row in the EMPLOYEE table. We begin by noting that the first row is the "current row" being "pointed at" by EX.

- Consider the 1<sup>st</sup> row: Does this row represent the highest paid employee in his department? Specifically, is MOE the highest paid employee in Department 20?

ENO	ENAME	SALARY	DNO
1000	MOE	2000.00	20



To answer this question the system must determine the maximum salary for employees who work in Department 20. The correlated Sub-SELECT returns this maximum salary.

The current row (the first EMPLOYEE row) is "pointed to" by EX. Therefore, EX.DNO represents the current DNO value. In this case its value is 20. After substituting 20 for EX.DNO, the Sub-SELECT becomes:

```
SELECT MAX (SALARY)
FROM   EMPLOYEE
WHERE  DNO = 20
```

Execution of this Sub-SELECT returns 9000.00, the largest SALARY value for Department 20. Hence the Outer-SELECT reduces to:

```
SELECT *
FROM   EMPLOYEE EX
WHERE  SALARY = 9000.00
```

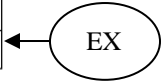
Within the context of a correlated Sub-SELECT, *this Sub-SELECT asks the system to only display the current row (the one pointed at by EX) if its SALARY value is 9000.00.* This is not true. MOE's salary is 2000.00, not 9000.00. Hence MOE's row does not appear in the result table. This completes processing for the first row. The system now iterates such that EX references the next (the second) row.

Important (Again): The above Outer-SELECT does **not** ask the system to: "Display all rows where SALARY is 9000.00." The specification of EX asks the system to display just the current row if its SALARY value is 9000.00.



- Consider the 2<sup>nd</sup> row: Does this row represent the highest paid employee in his department? Specifically, is LARRY the highest paid employee in Department 10?

ENO	ENAME	SALARY	DNO
2000	LARRY	2000.00	10



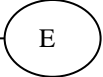
Because the second row is the current row, EX.DNO now has a value of 10. Hence, the Sub-SELECT becomes:

```
SELECT MAX (SALARY)
FROM   EMPLOYEE
WHERE  DNO = 10
```

Execution of this Sub-SELECT returns 2000.00, the largest SALARY for Department 10. Hence, the Outer-SELECT reduces to:

```
SELECT *
FROM   EMPLOYEE EX
WHERE  SALARY = 2000.00
```

LARRY's salary is 2000.00. Therefore, his row is placed in the result table. The system now iterates such that EX references the next (the third) row.

- Consider the 3<sup>rd</sup> row:
- | ENO  | ENAME | SALARY  | DNO |
|------|-------|---------|-----|
| 3000 | CURLY | 3000.00 | 20  |
- 

Because the third row is the current row, EX.DNO has a value of 20. The Sub-SELECT becomes:

```
SELECT MAX (SALARY)
FROM   EMPLOYEE
WHERE  DNO = 20
```

The Sub-SELECT returns 9000.00. The Outer-SELECT becomes:

```
SELECT *
FROM   EMPLOYEE EX
WHERE  SALARY = 9000.00
```

CURLY's salary is not 9000.00. Hence CURLY's row is not placed in the result table. The system now iterates such that EX references the next (the fourth) row.

Similar processing occurs for the remaining rows.

- 4<sup>th</sup> row:


ENO	ENAME	SALARY	DNO
4000	SHEMP	500.00	40



SHEMP works in Department 40. After substituting 40 for EX.DNO, the Sub-SELECT returns 500.00. SHEMP's salary is 500.00. Hence SHEMP's row is placed into result table.

- 5<sup>th</sup> row:


ENO	ENAME	SALARY	DNO
5000	JOE	400.00	10



JOE works in Department 10. After substituting 10 for EX.DNO, the Sub-SELECT returns 2000.00. JOE's salary is 400.00. Hence JOE's row is not placed into result table.

- 6<sup>th</sup> row:

ENO	ENAME	SALARY	DNO
6000	GEORGE	9000.00	20



GEORGE works in Department 20. After substituting 20 for EX.DNO, the Sub-SELECT returns 9000.00. GEORGE's salary is 9000.00. Hence GEORGE's row is placed into result table.

In general, this query objective requires the comparison of each employee's salary to the highest salary for his department. Therefore, the Sub-SELECT specified EX.DNO to reference the DNO value of each EMPLOYEE row.

Observe that the Outer-SELECT referenced the EMPLOYEE table which has 6 rows. Hence, the correlated Sub-SELECT was executed 6 times. (This repeated execution of a correlated Sub-SELECT has obvious efficiency implications that will be discussed in Appendix 25A.)

**Alternative Solutions:** See Sample Queries 23.14, 26.3, and 27.3.

**Exercises:**

- 25A. Display all information about the lowest paid employee in each department.
- 25B. Display the name and salary of any employee whose salary is less than the average employee salary for his department.

**Sample Query 25.2:** Reference the PARTSUPP table in the MTPCH database. Recall that PARTSUPP.PSPRICE represents the price paid to a supplier for a part. The objective is to find the supplier(s) with the lowest price for each part. Specifically, for every part that can be purchased from some supplier, display the PNO, SNO, and PSPRICE values corresponding to the lowest PSPRICE. Observe that, for a given part, multiple suppliers may offer the same lowest price. Sort the result by SNO within PNO.

```

SELECT *
FROM PARTSUPP PSX
WHERE PSPRICE = (SELECT MIN (PSPRICE)
                 FROM PARTSUPP
                 WHERE PNO = PSX.PNO)
ORDER BY PNO, SNO

```

PNO	SNO	PSPRICE
P1	S2	10.50
P3	S3	12.00
P4	S4	12.00
P5	S1	10.00
P5	S2	10.00
P6	S4	4.00
P6	S6	4.00
P6	S8	4.00
P7	S2	2.00
P8	S8	3.00

**Syntax & Logic:** The system iterates over every row in the table referenced by the Outer-SELECT (PARTSUPP). For each such row, the system substitutes the current PNO value for PSX.PNO and then executes the Sub-SELECT. The Sub-SELECT returns the lowest PSPRICE value for the current PNO. If the current PSPRICE matches the lowest price, the current PNO, SNO, and PSPRICE values are placed in the result table.

Observe that P2 is missing because no supplier supplies this part. Also observe that, for parts P5 and P6, multiple suppliers offer the same lowest price.

## Sometimes the Correlation-Name is Optional

In the previous two sample queries, the Outer-SELECT and the Sub-SELECT referenced the same tables. When this occurs, to avoid ambiguity, you must specify an explicit correlation-name (e.g., EX and PSX). Frequently, the Outer-SELECT and Sub-SELECT will reference different tables. When this occurs, the specification of an explicit correlation-name becomes optional. Then, if you do not specify an explicit correlation-name, the table-name serves as the correlation-name.

**Sample Query 25.3:** Reference the PARTSUPP and LINEITEM tables in the MTPCH database. Recall that the LINEITEM.LIPRICE represents the price that a customer paid for a part. Display all information in every PARTSUPP row with a PSPRICE that exceeds the average LIPRICE for the corresponding part.

```
SELECT *
FROM PARTSUPP
WHERE PSPRICE > (SELECT AVG (LIPRICE)
                 FROM LINEITEM
                 WHERE PNO = PARTSUPP.PNO)
```

PNO	SNO	PSPRICE
P8	S4	5.00

**Syntax:** The Sub-SELECT and Outer-SELECT reference different tables. Here, the Outer-SELECT's table-name (PARTSUPP) serves as the correlation-name. We generally recommend specifying an explicit correlation-name as illustrated below.

```
SELECT *
FROM PARTSUPP PSX
WHERE PSPRICE > (SELECT AVG (LIPRICE)
                 FROM LINEITEM
                 WHERE PNO = PSX.PNO)
```

**Logic:** Nothing new. Notice that each line-item is associated with the supplier (SNO) who supplied the part (PNO). This result implies that Supplier S4 may be charging too much for Part P8.

## EXISTS

The following sample query specifies "EXISTS (correlated Sub-SELECT)". We will see that EXISTS is logically equivalent to IN. (But don't jump to the erroneous conclusion that NOT EXISTS is always equivalent to NOT IN. This issue will be considered in Sample Query 25.6.) If you are inclined to avoid correlated Sub-SELECTs, then you might also be inclined to avoid EXISTS and simply code a statements that specifies IN within the context of a regular Sub-SELECT. This is not a bad idea, but we strongly encourage you to understand EXISTS and NOT EXISTS for reasons that will be described later in this chapter.

**Sample Query 25.4:** Same as Sample Query 23.5. Reference the DEPARTMENT and EMPLOYEE tables. Display all information about every department that has at least one employee.

```
SELECT *
FROM DEPARTMENT DX
WHERE EXISTS (SELECT 'X'
              FROM EMPLOYEE
              WHERE DNO = DX.DNO)
```

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>
10	ACCOUNTING	75000.00
20	INFO. SYS.	20000.00
40	ENGINEERING	25000.00

**Logic:** The system examines every row in the DEPARTMENT table. For each DEPARTMENT row, the system (in principle) executes the correlated Sub-SELECT to determine if *there exists* any EMPLOYEE row having the same DNO value as the current DEPARTMENT row.

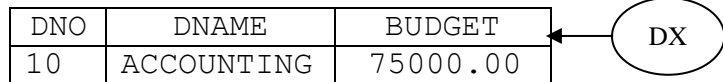
Specifically, for each DEPARTMENT row, the Sub-SELECT returns some value ('X') to represent a "hit," meaning that some EMPLOYEE row has a DNO value equal to the current DEPARTMENT.DNO value (DX.DNO). Under this circumstance, the current DEPARTMENT row is displayed. If the Sub-SELECT does not return an 'X' (a "no-hit"), the current DEPARTMENT row is not displayed.

Comment: The letter 'X' (the hit indicator) is an arbitrary value. Some users code "SELECT \*" in the Sub-SELECT.

**Detail Logic:** We walk through this process for the first three DEPARTMENT rows.

- Assume the 1<sup>st</sup> row is:

DNO	DNAME	BUDGET
10	ACCOUNTING	75000.00



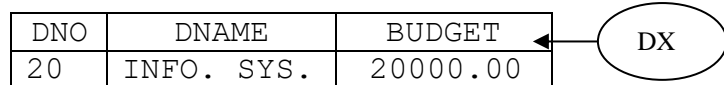
The current DEPARTMENT row is "pointed at" by DX. Hence DX.DNO has value of 10. After substituting 10 for DX.DNO, the Sub-SELECT becomes:

```
SELECT 'X' FROM EMPLOYEE WHERE DNO = 10
```

Executing the Sub-SELECT produces a "hit" on some EMPLOYEE row, either LARRY or JOE. The Sub-SELECT returns some value. (The fact that it returns "X" is not relevant.) Hence, the current row is placed into the result table.

- Assume the 2<sup>nd</sup> row is:

DNO	DNAME	BUDGET
20	INFO. SYS.	20000.00



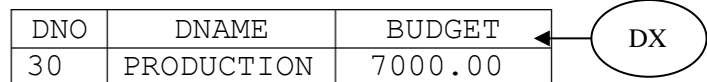
After substituting 20 for DX.DNO, the Sub-SELECT becomes:

```
SELECT 'X' FROM EMPLOYEE WHERE DNO = 20
```

Executing the Sub-SELECT produces a "hit." Hence the current row is placed into the result table.

- Assume the 3<sup>rd</sup> row is:

DNO	DNAME	BUDGET
30	PRODUCTION	7000.00



After substituting 30 for DX.DNO, the Sub-SELECT becomes:

```
SELECT 'X' FROM EMPLOYEE WHERE DNO = 30
```

Executing this Sub-SELECT produces a "no hit." Hence the current row is not placed into the result table.

### Important Exercise:

25C. Code two alternative solutions for this Sample Query 25.4. The first solution should specify IN. The second solution should specify a join-operation.

## NOT EXISTS

The next sample query specifies "NOT EXISTS (correlated Sub-SELECT)". In this sample query, NOT EXISTS is logically equivalent to NOT IN. However, Sample Query 25.6 will show that this equivalency does not always apply.

**Sample Query 25.5:** Same as Sample Query 23.8. Reference the DEPARTMENT and EMPLOYEE tables. Display the DNAME and BUDGET values for any department that does not have any employees.

```
SELECT DNAME, BUDGET
FROM DEPARTMENT DX
WHERE NOT EXISTS (SELECT 'X'
                  FROM EMPLOYEE
                  WHERE DNO = DX.DNO)
```

<u>DNAME</u>	<u>BUDGET</u>
PRODUCTION	7000.00

**Logic:** NOT EXISTS evaluates to True if the Sub-SELECT returns a "no hit." For example, when the system considers Department 30, the Sub-SELECT becomes:

```
SELECT 'X' FROM EMPLOYEE WHERE DNO = 30
```

This Sub-SELECT returns a "no hit". Hence, the NOT EXISTS evaluates to True, and the name and budget of Department 30 are placed in the result table. Conversely, execution of the Sub-SELECT for Departments 10, 20, and 40 returns X (a hit); the NOT EXISTS comparisons evaluate to False; and, data about these departments are not displayed.

**Alternative Solution:** Many users are inclined to avoid correlated Sub-SELECTs. They prefer to code the solution shown in Sample Query 23.8.

```
SELECT DNAME, BUDGET
FROM DEPARTMENT
WHERE DNO NOT IN (SELECT DNO FROM EMPLOYEE)
```

## Important! NOT EXISTS is Not Equivalent to NOT IN

Once again, we must consider null values. The previous Sample Query 25.5 illustrated an example where NOT EXISTS and NOT IN produced the same result. However, *given the presence of null values, the following sample query will illustrate that NOT IN is not equivalent as NOT EXISTS.*

**Sample Query 25.6:** Reference the DEPARTMENT and EMPLOYEE3 tables. Display all information about any employee assigned to a department that is not represented in the DEPARTMENT table. (This includes any employee with a null DNO value.)

```
SELECT *
FROM EMPLOYEE3 EX
WHERE NOT EXISTS (SELECT 'X'
                  FROM DEPARTMENT
                  WHERE DNO = EX.DNO)
```

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
1000	MOE	2000.00	99
6000	GEORGE	9000.00	-

**Logic:** MOE appears in the result because his DNO value (99) is not in the DEPARTMENT.DNO column. GEORGE also appears in the result because his DNO value (null) is not in the DEPARTMENT.DNO column. When EX is pointing at GEORGE's row, the Sub-SELECT returns a "no hit," and therefore the NOT EXISTS condition evaluates to True, selecting GEORGE's row for display. *Notice the difference between this logic and the following statement that specifies NOT IN.*

```
SELECT * FROM EMPLOYEE3
WHERE DNO NOT IN (SELECT DNO FROM DEPARTMENT)
```

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
1000	MOE	2000.00	99

This statement does not display GEORGE's row. The regular SUB-SELECT returns (10, 30, 30, 40), and GEORGE's null DNO value does not match any value on this list. (You may want to review NOT IN logic in Chapter 5.)

**Conclusion:** NOT IN will not match on a null value. NOT EXISTS will match on a null value.

**Alternative Solution:** See solution for Exercise 25Q.



## Select Rows with Unique Values in a Column

The following sample query uses a correlated Sub-SELECT to compare a table with itself.

**Sample Query 25.7:** Reference the EMPLOYEE table. Display all information about any employee whose salary is unique. This means that no other employee earns the same salary. (Recall that SALARY is not defined as a PRIMARY KEY or UNIQUE column. Hence the SALARY column may contain some duplicate values.)

```
SELECT *
FROM EMPLOYEE EX
WHERE NOT EXISTS (SELECT 'X'
                  FROM   EMPLOYEE
                  WHERE  SALARY = EX.SALARY
                  AND    ENO <> EX.ENO)
```

ENO	ENAME	SALARY	DNO
3000	CURLY	3000.00	20
4000	SHEMP	500.00	40
5000	JOE	400.00	10
6000	GEORGE	9000.00	20

**Logic:** Consider why the Sub-SELECT must specify "AND ENO <> EX.ENO". Without this condition, each EMPLOYEE row would match itself. This example shows the advantage of knowing about primary-key and unique columns. Because ENO is the primary-key, we know that its value cannot be equal any another ENO value within the EMPLOYEE table.

### Exercises:

- 25D. Reference the STATE and CUSTOMER tables in the MTPCH database. Write three different statements to display all information about every state that has at least one customer. The first statement should specify EXISTS; the second statement should specify IN; the third statement should specify a join-operation.
- 25E. Reference the STATE and CUSTOMER tables in the MTPCH database. Write two different statements to display all information about every state that does not have any customers. The first statement should specify NOT EXISTS, and the second statement should specify NOT IN.

## Correlated Sub-SELECT Specified within SELECT-Clause

The next sample query is a variation of Sample Query 23.16 which illustrated a regular Sub-SELECT specified within a SELECT-clause. The following sample query illustrates a correlated Sub-SELECT specified within a SELECT-clause.

**Sample Query 25.8:** Reference the EMPLOYEE table. Consider adjusting each employee's salary to a value that is equal to the employee's *departmental* average salary plus 5% of the employee's current salary. Display each employee number, name, and current salary, followed by the adjusted salary.

```
SELECT ENO, ENAME, SALARY,
       (SELECT AVG (SALARY) FROM EMPLOYEE WHERE DNO = E.DNO)
       + (.05 * SALARY) ADJSALARY
FROM EMPLOYEE E
```

ENO	ENAME	SALARY	ADJSALARY
1000	MOE	2000.00	4766.66
2000	LARRY	2000.00	1300.00
3000	CURLY	3000.00	4816.66
4000	SHEMP	500.00	525.00
5000	JOE	400.00	1220.00
6000	GEORGE	9000.00	5116.66

**Syntax:** The FROM-clause in the Outer-SELECT specifies E as a correlation-name. The Sub-SELECT uses this correlation-name to reference a specific DNO value.

**Logic:** The Sub-SELECT is executed for each EMPLOYEE row. It returns the departmental average salary for the current row's DNO value (the value pointed to by E.DNO). This departmental average is added to 5% of the employee's current salary.

**Alternative Solutions:** See Exercises 26N and 27N.

## Ancient History: Do-It-Yourself LEFT OUTER JOIN

In early versions of SQL, before relational database systems directly supported OUTER JOIN, users had to code do-it-yourself logic to implement an outer-join operation. The following statement illustrates an example which is presented for tutorial purposes only.

**Sample Query 25.9:** Similar to Sample Query 19.5 which specified LEFT OUTER JOIN. Reference the DEPARTMENT and EMPLOYEE tables. Display all information about every department along with all information about employees who work in those departments. Sort the result by the first column.

```
SELECT D.DNO, D.DNAME, D.BUDGET,
       E.ENO, E.ENAME, E.SALARY, E.DNO
FROM   DEPARTMENT D, EMPLOYEE E
WHERE  D.DNO = E.DNO

UNION ALL

SELECT DNO, DNAME, BUDGET,
       'No Emp ', ' ', 0, 0
FROM   DEPARTMENT DX
WHERE  NOT EXISTS (SELECT 'X'
                  FROM EMPLOYEE
                  WHERE DNO = DX.DNO)

ORDER BY 1
```

DNO	DNAME	BUDGET	ENO	ENAME	SALARY	DNO1
10	ACCOUNTING	75000.00	2000	LARRY	2000.00	10
10	ACCOUNTING	75000.00	5000	JOE	400.00	10
20	INFO. SYS.	20000.00	1000	MOE	2000.00	20
20	INFO. SYS.	20000.00	3000	CURLY	3000.00	20
20	INFO. SYS.	20000.00	6000	GEORGE	9000.00	20
30	PRODUCTION	7000.00	<b>No Emp</b>		<b>0.00</b>	<b>0</b>
40	ENGINEERING	25000.00	4000	SHEMP	500.00	40

### Optional Exercise:

25F. Write an ancient history solution for a full outer-join of the DEPARTMENT and EMPLOYEE3 tables.

## "FOR ALL" (Double NOT EXISTS)

The following sample query presents the **most complex SELECT statement in this book!** Fortunately, you can skip this example without any loss of continuity. However, this example is interesting.

Consider *all* parts (P1-P8) described in the PART table. Examination of the PARTSUPP table shows that no individual supplier sells *all* parts. However, Supplier S4 sells all parts except Part P2. To allow Supplier S4 to sell all parts, we must insert a new row in PARTSUPP table by executing the following statement.

```
INSERT INTO PARTSUPP VALUES ('P2', 'S4', 12.00)
```

You should execute this INSERT statement if you intend to execute the following SELECT statement. However, remember to delete this new row afterwards.

**Sample Query 25.10:** Display the supplier number and name of any supplier who sells all parts described in the PART table.

```
SELECT SNO, SNAME
FROM SUPPLIER S
WHERE NOT EXISTS

    (SELECT 'X'
     FROM PART P
     WHERE NOT EXISTS

        (SELECT 'X'
         FROM PARTSUPP PS
         WHERE PS.PNO = P.PNO
              AND PS.SNO = S.SNO))
```

```
SNO  SNAME
S4    SUPPLIER4
```

This result shows that only Supplier S4 sells all parts.

**Syntax:** Nothing New.

**Logic:** We want to code a SELECT statement that embodies the logical notion of "FOR ALL." The problem is that SQL does not provide a keyword that directly supports the FOR ALL concept. However, it is possible to write an equivalent statement by coding two correlated Sub-SELECTs that specify NOT EXISTS before each Sub-SELECT. This approach specifies a "double-negative" logic. (Example: "I ain't got no money" implies that I do have some money.)

We begin our explanation of this "Double-NOT-EXISTS" code by asking you to temporarily ignore the outermost SELECT. Focus on the Sub-SELECTs and make two changes.

(i). Substitute PNO for 'X' in the first Sub-SELECT

(ii). Substitute 'S1' for S.SNO in the second Sub-SELECT

The modified Sub-SELECTs look like:

```
SELECT PNO                                ← change (i)
FROM PART P
WHERE NOT EXISTS
      (SELECT 'X'
       FROM PARTSUPP PS
       WHERE PS.PNO = P.PNO
       AND   PS.SNO = 'S1')              ← change (ii)
```

This code asks the system to display the PNO values of all parts not sold by supplier S1. Executing this statement would produce:

```
PNO
P1
P2
P3
P4
P6
P7
P8
```

This intermediate result tells us that Supplier S1 does not sell parts P1-P4 and P6-P8. (Supplier S1 only sells part P5.) If you substitute any SNO value, *except S4*, into the above statement and execute it, you will always get a "hit" representing a list of parts that are not sold by the supplier.

However, if you substitute S4, you get a "no-hit," implying that Supplier S4 sells all parts. More explicitly, substituting S4 for S.SNO we have.

```
SELECT PNO
FROM PART P
WHERE NOT EXISTS
  (SELECT 'X'
   FROM PARTSUPP PS
   WHERE PS.PNO = P.PNO
   AND PS.SNO = 'S4')
```

Executing this statement produces a "no-hit."

Double-Negative Logic: For a given supplier, if there are **no** parts that are **not** sold by the supplier, then the supplier sells **all** parts.

Consider the outermost SELECT and assume that S points to the SUPPLIER row for S4. Because the Sub-SELECTs generate a no-hit, we have:

```
SELECT SNO, SNAME
FROM SUPPLIER S
WHERE NOT EXISTS (no-hit)
```

The NOT EXISTS (no-hit) returns True. Hence, the SNO and SNAME values for Supplier S4 will be displayed.

In general, the outermost SELECT asks the system to iterate through all SUPPLIER rows, substituting each row's SNO value for S.SNO on each iteration. With the exception of S4, the nested Sub-SELECTs always produce a hit, implying those rows are not selected by the outermost-SELECT. Every SUPPLIER row, except the S4 row, fails the outermost NOT EXISTS condition.

**Finally, Don't Forget:** Delete the previously inserted PARTSUPP row by executing:

```
DELETE FROM PARTSUPP WHERE PNO = 'P2' AND SNO = 'S4'
```

## Summary

**Avoid correlated Sub-SELECTs?** Most users feel that correlated Sub-SELECTs are more complex than regular Sub-SELECTs. They also contend that the iterative execution of a correlated Sub-SELECT implies that a correlated Sub-SELECT incurs a higher performance cost. Therefore, they avoid correlated Sub-SELECTs and attempt to formulate alternative solutions. We briefly comment on these issues below.

1. Complexity: Correlated Sub-SELECTs are usually more complex than regular Sub-SELECTs. With the exception of Sample Query 25.10 (double NOT EXISTS), future chapters will present alternative solutions for all other sample queries presented in this chapter. Most users would prefer these alternative solutions.
2. Efficiency: Many SQL reference manuals present efficiency guidelines, and some of these manuals discourage correlated Sub-SELECTs. However, while this may be a pretty good guideline, there are exceptions to this guideline. Appendix 25A will discuss these exceptions.

## Summary Exercises

Code solutions that specify correlated Sub-SELECTs unless directed otherwise.

- 25G. Reference the PRESERVE table. Determine the largest preserve (greatest number of acres) in each state. Display the state code followed the preserve number, name, and acreage.
- 25H. Code an alternative solution to the preceding Exercise 25G. Specify a regular Sub-SELECT that returns multiple columns. Hint: Review Exercise 23.14. [Skip this exercise if your system does not allow regular Sub-SELECTs to return multiple columns.]
- 25I. Reference the PARTSUPP table in the MTPCH database. The basic objective is to display information about each part having a price that is less than the average price for the part. Specifically, for every part that you can purchase from some supplier, display the PNO, SNO, and PSPRICE values for any part having a price that is less than the average price for the part. Sort the result by SNO within PNO.
- 25J. Reference the PARTSUPP and SUPPLIER tables in the MTPCH database. Modify the above Exercise 25I to include the name of the supplier.
- 25K. Sample Query 21.3 specified an INTERSECT operation (shown below) to display the employee numbers and names of all persons who are described in both the EMPLOYEE and PROJMGR tables.

```
SELECT ENO, ENAME
FROM   EMPLOYEE
INTERSECT
SELECT ENO, PMNAME
FROM   PROJMGR
ORDER BY 1
```

- a. Code an alternative solution using EXISTS.
- b. Code another alternative solution using IN.



25L. Sample Query 21.4 specified an EXCEPT operation (shown below) to display the employee number and name of every employee who is not a project manager.

```
SELECT ENO, ENAME
FROM   EMPLOYEE
EXCEPT
SELECT ENO, PMNAME
FROM   PROJMGR
ORDER BY 1
```

- a. Code an alternative solution using NOT EXISTS.
- b. Code another alternative solution using NOT IN.
- c. Important Question: How do you know that, in this circumstance, the NOT EXISTS and NOT IN solutions are equivalent to each other?

25M1. Reference the DEPARTMENT and EMPLOYEE tables. Assume that management is considering adjusting each department's budget. Each new departmental budget might be changed to twice the total salary of all employees who work in the department. Before implementing this change, management asks you to produce a report that displays each department's number, name, current budget, and the proposed new budget. If a department does not have any employees, then display a null value for the proposed new budget. The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>	<u>NEWDBUDGET</u>
10	ACCOUNTING	75000.00	4800.00
20	INFO. SYS.	20000.00	28000.00
30	PRODUCTION	7000.00	-
40	ENGINEERING	25000.00	1000.00

Your solution should specify a correlated Sub-SELECT within the SELECT-clause as shown in Sample Query 25.8. (The following Exercise 25M2 suggests an alternative solution.)

25M2. This is an optional exercise. Code an alternative solution for the preceding Exercise 25M1. Instead of coding a Sub-SELECT, your solution should specify a left outer-join operation and group by the DNO, DNAME, and BUDGET columns.

25N. Exercise 23I asked you to code a regular Sub-SELECT to satisfy the query objective: Reference the DEPARTMENT and EMPLOYEE tables. Display the overall total budget of those departments which have at least one employee. Code another solution using a correlated Sub-SELECT. The result should look like:

```
TOTBUDGET
120000.00
```

25O. This exercise modifies Exercise 25M1. The user does not want to see any null values in the report. Therefore, if a department does not have any employees, the new budget should be the same as the current budget. The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>	<u>NEWBUDGET</u>
10	ACCOUNTING	75000.00	4800.00
20	INFO. SYS.	20000.00	28000.00
30	PRODUCTION	7000.00	<b>7000.00</b> ←
40	ENGINEERING	25000.00	1000.00

Code two solutions, each having the same basic structure as the solution for Exercise 25Ma.

The first should use the COALESCE function to substitute the current BUDGET value for a null value in the NEWBUDGET column. The basic structure of the SELECT-clause is:

```
SELECT ... COALESCE ((correlated Sub-SELECT...), BUDGET)
                NEWBUDGET
```

The second solution should specify a CASE-expression to substitute the current BUDGET value for a null value in the NEWBUDGET column. The basic structure of the CASE-expression is:

```
CASE (SELECT COUNT(*) FROM EMPLOYEE
      WHERE DNO=D.DNO)
      WHEN 0 THEN . . .
      ELSE (correlated Sub-SELECT . . .)
END NEWBUDGET
```

25P. Review Exercise: Same as for Sample Query 25.7. Reference the EMPLOYEE table. Display all information about any employee whose salary is unique. This means that no other employee earns the same salary.

Do not specify a correlated Sub-SELECT. Code a regular Sub-SELECT that joins the EMPLOYEE table with itself to retrun ENO values of any employee who has the same salary as another employee.

25Q. Review Exercise: Same as for Sample Query 25.6. Reference the DEPARTMENT and EMPLOYEE3 tables. Display all information about any employee assigned to a department that is not represented in the DEPARTMENT table. (This includes any employee with a null DNO value.)

Code a roundabout solution that specifies NOT IN and UNION ALL.

## Appendix 25A: Efficiency

A correlated Sub-SELECT is (in principle) executed multiple times, once for each row selected by the Outer-SELECT. This observation leads to a general recommendation that you should avoid correlated Sub-SELECTs. However, this appendix will present some caveats to this recommendation.

**Case Study:** Reconsider the following Statement-4 presented in Appendix 23A. There we assumed that:

- TAB2 was a very large table, and
- The regular Sub-SELECT produced a large intermediate result

```
Statement-4:  SELECT COLA, COLB, COLC
              FROM TAB1
              WHERE COLA IN (SELECT COLX FROM TAB2)
```

In Appendix-23A, these assumptions motivated us to consider rewriting Statement-4 using an inner-join. Here we consider rewriting this statement using a correlated Sub-SELECT that specifies EXISTS.

```
Statement-5:  SELECT COLA, COLB, COLC
              FROM TAB1 T1
              WHERE EXISTS (SELECT 'X'
                           FROM TAB2
                           WHERE COLA = T1.COLA)
```

Query Rewrite: The following page describes circumstances where Statement-5 may be more efficient than Statement-4. Again, we remind you that, *ideally*, your optimizer should automatically rewrite a SELECT statement into the most efficient form. The ideal optimizer is not influenced by the idiosyncratic coding style of the user. However, because real-world optimizers are not perfect, occasionally you may have to consider adopting a do-it-yourself approach.

## **Correlated Sub-SELECTs: Efficiency Considerations**

Below we analyze the preceding Statement-5 to illustrate potential efficiency advantages for a correlated Sub-SELECT.

Size of TAB1: When analyzing Statement-4, the size of the TAB1, the table referenced in the Outer-SELECT, was not a factor because a regular Sub-SELECT is only executed once. With Statement-5, the size of TAB1 becomes a factor because the correlated Sub-SELECT is executed for each row in TAB1. Therefore, the overall cost for this correlated Sub-SELECT will be less expensive if TAB1 only has a few rows.

Cost for Each Execution of a Correlated Sub-SELECT: Even if the correlated Sub-SELECT is executed many times, each execution could be fast because:

1. Unlike a regular Sub-SELECT, each execution of a correlated Sub-SELECT is not required to retrieve and save any intermediate results. EXISTS simply looks for a "hit or no-hit" response.
- 2a. Assume there is no index on TAB2.COLA. Then the Sub-SELECT has to scan the TAB2 table. But it rarely has to scan all TAB2 rows. The EXISTS condition is only looking for a hit on some COLA value. The entire TAB2 table is only scanned when it searches for a T1.COLA value that is not present in TAB2.
- 2b. Alternatively, assume there is an index on TAB2.COLA. (This is likely because TAB2.COLA is a frequently primary key or a foreign-key.) This index could be very helpful. The correlated Sub-SELECT would directly search of the index to quickly conclude hit or no-hit. There is no need to follow index pointers to retrieve rows from the TAB2 table.

**Conclusion:** In some circumstances, correlated Sub-SELECTs that specify EXISTS or NOT EXISTS can be very efficient.

## Appendix 25B: Theory

We offer a few brief comments for those readers who have taken a mathematics course that covered the predicate calculus. Other readers can skip this appendix without any loss of continuity.

In Appendix 1B, we noted that SQL is derived from Codd's Relational Calculus and his Relational Algebra. Previous theory appendices focused on associating SQL with the Relational Algebra. Having completed this chapter, we can now associate SQL with some concepts and symbols that are used within the Relational Calculus.

The Relational Calculus inherits two symbols from the Predicate Calculus. These symbols are the backwards-E that represents "there exists" and the upside-down-A that represents "for all."

$\exists$  (There Exists)

$\forall$  (For All)

- The EXISTS condition (Sample Query 25.4) is based on the  $\exists$  symbol from the Relational Calculus. The NOT EXISTS condition (Sample Query 25.5) corresponds to the  $\sim\exists$  symbol (there does not exist). Some mathematics books represent NOT EXISTS by specifying a slash (/) overlaid on top of  $\exists$ .
- The narrative for Sample Query 25.10 noted that SQL does not have a keyword for the  $\forall$  symbol. This situation required coding a more complex SELECT statement that specified two  $\sim\exists$  comparisons.

Suggestion: If you intend to investigate the Relational Calculus, you are encouraged to do some preliminary reading about the Predicate Calculus.

This page is intentionally blank.

## Inline Views

This chapter introduces a variation of the Sub-SELECT called an *inline view*, also known as a *dynamic view*, *table-expression*, or *derived table*.

From a syntax perspective, the basic observation is that an inline view is a Sub-SELECT that is specified within a FROM-clause as illustrated below.

```
SELECT _____  
  
FROM (SELECT _____  
      FROM _____  
      WHERE _____) AS ____  
  
WHERE _____
```

The system executes the Sub-SELECT to derive an intermediate-result table (the inline view). The keyword AS is used to assign a name to this table. Then the Outer-SELECT will use this name to reference the intermediate-result table.

The *inline* view disappears after the Outer-SELECT completes execution.

This chapter begins with an overly simplistic tutorial sample query. Subsequent sample queries will be more realistic.



## Mundane Tutorial Example

**Sample Query 26.1:** Reference the EMPLOYEE table. Only consider employees who work in Department 20 and earn less than \$8,000.00. Display each employee's name followed by an amount equal to his salary plus \$250.00.

For tutorial purposes, we use a roundabout method to satisfy this query objective. Generate an intermediate-result table (an inline view) called TEMP20 that contains the ENAME and SALARY values of every employee who works in Department 20. Then reference TEMP20 to display the ENAME and SALARY + \$250.00 values for every row having a SALARY that is less than \$8,000.00.

```
SELECT TEMP20.ENAME, TEMP20.SALARY+250.00

FROM  (SELECT ENAME, SALARY
        FROM  EMPLOYEE
        WHERE DNO = 20) AS TEMP20

WHERE TEMP20.SALARY < 8000.00
```

<u>ENAME</u>	<u>SALARY + 250.00</u>
MOE	2250.00
CURLY	3250.00

**Syntax & Logic:** The Sub-SELECT is specified in the FROM-clause and is enclosed within parentheses. It produces an intermediate-result table (the inline view) called TEMP20 that looks like:

<u>TEMP20</u>	
<u>ENAME</u>	<u>SALARY</u>
MOE	2000.00
CURLY	3000.00
GEORGE	9000.00

Then the Outer-SELECT reduces to:

```
SELECT TEMP20.ENAME, TEMP20.SALARY+250.00
FROM  TEMP20
WHERE TEMP20.SALARY <= 8000.00
```

Execution of this statement produces the final result. Then TEMP20 "goes away" when the statement terminates.

## Minimum of Maximum Values (“Mini-Max” Value)

Sample Query 9.13 noted that some systems do not allow you to nest an aggregate function within another aggregate function. For example, these systems would reject an aggregate function that looks like: MIN (MAX (Column)). Specifying an inline view allows you to bypass this limitation.

**Sample Query 26.2:** Reference the EMPLOYEE table. Determine the maximum salary in each department. Then display the smallest of these maximum values. (I.e., Display the “min of the maxes.”)

```
SELECT MIN (MAXSAL) MINIMAX
FROM   (SELECT DNO, MAX (SALARY) MAXSAL
        FROM   EMPLOYEE
        GROUP BY DNO) AS TMAXES
```

<u>MINIMAX</u>
500.00

**Syntax:** Within an inline view, you must assign a column alias to any column that is derived by an expression or function. For this reason, we assigned MAXSAL as the column-alias for the MAX (SALARY) column.

**Logic:** The Sub-SELECT returns an intermediate-result table (the inline view) called TMAXES that looks like:

<u>TMAXES</u>	
<u>DNO</u>	<u>MAXSAL</u>
10	2000.00
20	9000.00
40	500.00

Then the Outer-SELECT reduces to:

```
SELECT MIN (MAXSAL) MINIMAX
FROM   TMAXES
```

Execution of this statement produces the final result. Thereafter TMAXES is no longer available.

**Sample Query 26.3:** Display all information about the highest paid employee in every department that has at least one employee. (Same as Sample Queries 23.14 and 25.1.)

```

SELECT E.ENO, E.ENAME, E.SALARY, E.DNO
FROM EMPLOYEE E,
     (SELECT DNO, MAX (SALARY) MAXSAL
      FROM EMPLOYEE
      GROUP BY DNO) AS TMAXES
WHERE E.DNO      = TMAXES.DNO
AND   E.SALARY = TMAXES.MAXSAL

```

ENO	ENAME	SALARY	DNO
2000	LARRY	2000.00	10
4000	SHEMP	500.00	40
6000	GEORGE	9000.00	20

**Logic:** This is the same inline view (TMAXES) that was specified in the previous Sample Query 26.2. It returns the same intermediate result table called that looks like:

<u>TMAXES</u>	
<u>DNO</u>	<u>MAXSAL</u>
10	2000.00
20	9000.00
40	500.00

Then the EMPLOYEE TABLE is joined with TMAXES.

```

SELECT E.ENO, E.ENAME, E.SALARY, E.DNO
FROM EMPLOYEE E, TMAXES
WHERE E.DNO = TMAXES.DNO
AND   E.SALARY = TMAXES.MAXSAL

```

If an EMPLOYEE row has a DNO value that equals TMAXES.DNO and a SALARY value that equals TMAXES.MAXSAL, then this row corresponds to the highest paid employee in the DNO department.

**Alternative Solutions:** Sample Queries 23.14, 25.1, and 27.3.

## Exercises

Solve the following exercises by coding inline views. These exercises reference the EMPLOYEE table

26A. Determine the total salary for each department. Then display the largest of these totals. The result should look like:

```
LARGESTTOTAL
14000.00
```

26B. Determine the average salary for each department. Then display the smallest of these averages. The result should look like:

```
SMALLESTAVG
500.00
```

26C. Display all information about the lowest paid employee in each department. The result should look like:

```
ENO  ENAME      SALARY  DNO
1000 MOE          2000.00  20
4000 SEMP        500.00   40
5000 JOE         400.00   10
```

26D. For each department, display all information about every departmental employee who has a salary that is greater than or equal to the average salary for the department. The result should look like:

```
ENO  ENAME      SALARY  DNO
2000 LARRY     2000.00  10
4000 SEMP        500.00   40
6000 GEORGE    9000.00  20
```

## Multiple Inline Views

The following sample query specifies two inline views, TMAXES and DMIN.

**Sample Query 26.4:** Whenever a department's highest paid employee has a salary that exceeds the smallest departmental budget, display both DNO values (with column headings that distinguish each value) along with the corresponding maximum salary and minimum budget values.

```
SELECT TMAXES.DNO MAXSALDEPT, TMAXES.MAXSAL,
       DMIN.DNO MINBUDGETDEPT, DMIN.BUDGET MINBUDGET

FROM (SELECT DNO, MAX (SALARY) MAXSAL
      FROM EMPLOYEE
      GROUP BY DNO) AS TMAXES,

      (SELECT DNO, BUDGET
      FROM DEPARTMENT
      WHERE BUDGET = (SELECT MIN (BUDGET)
                     FROM DEPARTMENT)) AS DMIN

WHERE TMAXES.MAXSAL > DMIN.BUDGET
```

<u>MAXSALDEPT</u>	<u>MAXSAL</u>	<u>MINBUDGETDEPT</u>	<u>MINBUDGET</u>
20	9000.00	30	7000.00

**Syntax:** The FROM-clause defines two inline views called TMAXES and DMIN. A comma must separate the Sub-SELECT specification of each inline view.

```
FROM (SELECT . . .) AS TMAXES,
      (SELECT . . .) AS DIM
```

**Logic:** The idea is to generate two intermediate-result tables (two inline views) and then join them by matching on a greater-than comparison.

The Sub-SELECT for the first inline view is:

```
(SELECT DNO, MAX (SALARY) MAXSAL
 FROM EMPLOYEE
 GROUP BY DNO) AS TMAXES
```

It creates an intermediate-result table that looks like:

<u>TMAXES</u>	
<u>DNO</u>	<u>MAXSAL</u>
10	2000.00
20	9000.00
40	500.00

The Sub-SELECT for the second inline view is:

```
(SELECT DNO, BUDGET
FROM DEPARTMENT
WHERE BUDGET = (SELECT MIN (BUDGET)
FROM DEPARTMENT)) AS DMIN
```

It creates an intermediate-result table that looks like:

<u>DMIN</u>	
<u>DNO</u>	<u>BUDGET</u>
30	7000.00

Then, the system produces the final result by joining the two intermediate-result tables with the join-condition:

```
TMAXES.MAXSAL > DMIN.BUDGET
```

### **Exercises:**

26E1. Will this SELECT statement work if two departments have the same minimal budget? Will it work if two departmental employees have the same maximum salary?

26E2. Reference the PARTSUPP and LINEITEM tables. For each part sold, the actual selling price (LIPRICE) is always greater than or equal to the part's purchase price (PSPRICE). Hence, a part's average selling price is always greater than or equal to its average purchase price. Display information about any part where the difference between these averages is less than 75 cents. For any such part, display its part number followed by its average purchase price and average selling price. The result should look like:

<u>PNO</u>	<u>AVGPS</u>	<u>AVGLI</u>
P7	3.00	3.50

## Inline View with One-Row and One-Column

The following inline view generates an intermediate-result table (TEMP) with just one row and one column.

**Sample Query 26.5:** Same as Sample Query 23.16. Reference the EMPLOYEE table. Consider the impact of adjusting each employee's salary to a value that is equal to the overall average of all current salaries plus 5% of the employee's current salary. Display each employee number, name, current salary, and adjusted salary.

```
SELECT ENO, ENAME, SALARY,  
       TEMP.EMPAVGSAL + (.05*SALARY) ADJSAL  
FROM EMPLOYEE,  
     (SELECT AVG (SALARY) EMPAVGSAL FROM EMPLOYEE) AS TEMP
```

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>ADJSAL</u>
1000	MOE	2000.00	2916.66
2000	LARRY	2000.00	2916.66
3000	CURLY	3000.00	2966.66
4000	SHEMP	500.00	2841.66
5000	JOE	400.00	2836.66
6000	GEORGE	9000.00	3266.66

**Logic:** The Sub-SELECT returns an intermediate-result table with one row and one column that looks like:

<u>TEMP</u>
<u>EMPAVGSAL</u>
2816.66

The Outer-SELECT does not specify a join-condition. Hence the system executes a cross-product on EMPLOYEE and TEMP to produce the following intermediate cross-product result:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>	<u>EMPAVGSAL</u>
1000	MOE	2000.00	20	2816.66
2000	LARRY	2000.00	10	2816.66
3000	CURLY	3000.00	20	2816.66
4000	SHEMP	500.00	40	2816.66
5000	JOE	400.00	10	2816.66
6000	GEORGE	9000.00	20	2816.66

Finally, the Outer-SELECT clause produces the final result.

**Alternative Solutions:** Sample Queries 23.16 and 27.5.

The following sample query uses coding methods introduced in the previous two sample queries. It specifies two inline views, TEMP1 and TEMP2, where TEMP2 has just one row with one column.

**Sample Query 26.6:** Same as Sample Query 23.17. For each department referenced in the EMPLOYEE table, display its department number and its average departmental salary followed by a comment indicating that the departmental average is less than, greater than, or equal to the overall average salary.

```
SELECT TEMP1.DNO, TEMP1.DEPTAVGSAL,
       CASE
         WHEN TEMP1.DEPTAVGSAL < TEMP2.EMPAVGSAL
           THEN 'LESS THAN OVERALL AVERAGE SALARY'
         WHEN TEMP1.DEPTAVGSAL = TEMP2.EMPAVGSAL
           THEN 'EQUAL TO OVERALL AVERAGE SALARY'
         ELSE   'GREATER THAN OVERALL AVERAGE SALARY'
       END COMMENT
FROM
  (SELECT DNO, AVG (SALARY) DEPTAVGSAL
   FROM EMPLOYEE
   GROUP BY DNO) AS TEMP1,
  (SELECT AVG (SALARY) EMPAVGSAL FROM EMPLOYEE) AS TEMP2
```

<u>DNO</u>	<u>DEPTAVGSAL</u>	<u>COMMENT</u>
10	1200.00	LESS THAN OVERALL AVERAGE SALARY
20	4666.66	GREATER THAN OVERALL AVERAGE SALARY
40	500.00	LESS THAN OVERALL AVERAGE SALARY

**Syntax and Logic:** Nothing New.

**Alternative Solutions:** Sample Query 23.17 and 27.6



## Identical Sub-SELECTs

The following sample query specifies two inline views, TMAXES1 and TMAXES2, with identical Sub-SELECTs that generate identical intermediate-result tables. Redundant inline views are usually written by a person who does not know about the WITH-clause that will be introduced in the following chapter.

**Sample Query 26.7:** Extend Sample Query 26.2 which asked you to display the "min of the maximum" departmental salaries. Also display the DNO value of the department that has this "mini-max" value.

```
SELECT TMAXES1.DNO, TMAXES1.MAXSAL MINIMAX

FROM (SELECT DNO, MAX (SALARY) MAXSAL
      FROM EMPLOYEE
      GROUP BY DNO) AS TMAXES1

WHERE TMAXES1.MAXSAL =
      (SELECT MIN (MAXSAL)
       FROM (SELECT DNO, MAX (SALARY) MAXSAL
             FROM EMPLOYEE
             GROUP BY DNO) AS TMAXES2)
```

<u>DNO</u>	<u>MINIMAX</u>
40	500.00

**Syntax & Logic:** Nothing new. Both inline views happen to return intermediate-result tables with the same data.

**Common Error:** To avoid coding redundant Sub-SELECTs, some users make a common error by attempting to execute.

```
SELECT DNO, MAXSAL
FROM (SELECT DNO, MAX (SALARY) MAXSAL
      FROM EMPLOYEE
      GROUP BY DNO) AS TMAXES
WHERE MAXSAL = (SELECT MIN (MAXSAL) FROM TMAXES)
```

This statement will generate an error. The Sub-SELECT specified within the WHERE-clause is not allowed to reference TMAXES (even though it appears to be reasonable).

**Alternative (Preferred) Solution:** Sample Query 27.7.

## Syntax Variations

When coding an inline view, some systems allow the following variations in syntax.

You do not need to specify the keyword AS. For example, in Sample Query 26.1, TEMP20 could be specified as shown below.

```
SELECT TEMP20.ENAME, TEMP20.SALARY+250.00
FROM (SELECT ENAME, SALARY
      FROM EMPLOYEE
      WHERE DNO = 20) TEMP20
WHERE TEMP20.SALARY < 8000.00
```

Furthermore, sometimes, an view does not need to have a name, as illustrated below.

```
SELECT ENAME, SALARY+250.00
FROM (SELECT ENAME, SALARY
      FROM EMPLOYEE
      WHERE DNO = 20)
WHERE SALARY < 8000.00
```

We generally *discourage* these syntactical variations. It is conceptually cleaner to explicitly assign a name to an inline view using the keyword AS.

### Exercise:

26F. Reference the EMPLOYEE table. Display the department number and total salary of the department having the largest total salary. The result should look like:

<u>DNO</u>	<u>LARGESTTOTAL</u>
20	14000.00

## Summary

**Primary Advantage:** Inline views are very useful. Sometimes, when you analyze a query objective, you might say to yourself "This SELECT statement would simple if I had a table that looked like:"


Therefore, although you do not have any such table, you might be able to generate the desired table as an intermediate-result table by coding an inline view.

**Inline View Specifies a Correlated Sub-SELECT:** For tutorial purposes, this chapter's sample queries illustrated inline views that specified regular (non-correlated) Sub-SELECTs. The following statement illustrates that you can code an inline view (DMAXSAL) that specifies a correlated Sub-SELECT. (The code for DMAXSAL was explained in Sample Query 25.1.)

```
SELECT DMAXSAL.ENO, DMAXSAL.ENAME, DMAXSAL.SALARY
FROM (SELECT *
      FROM EMPLOYEE EX
      WHERE SALARY = (SELECT MAX (SALARY)
                     FROM EMPLOYEE
                     WHERE DNO = EX.DNO)) AS DMAXSAL
WHERE DMAXSAL.SALARY > 5000.00
```

**Terminology:** We have been very casual in our use the term "inline view." Sometimes we use this term to refer to the Sub-SELECT per se. Other times we use this term to refer to the intermediate-result table that is generated by the Sub-SELECT.

**"Temporary Table:"** You might be inclined to think of an inline view as a "temporary table." While this term may be conceptually valid, we do not use this term because some systems use it to refer to a different (but similar) type of table. (Applications developers should be interested in temporary tables. See Appendix 28B.)

## Summary Exercises

Solutions for the following exercises should specify inline views.

26G. Reference the EMPLOYEE table. Determine the average salary in each department. Then display the largest of these averages. The result should look like:

```
MAXAVGSAL
4666.66
```

26H. Reference the PRESERVE table. For each state, display the state code and preserve's number, name, acreage for every preserve that is larger than the average preserve acreage for the state. The result should look like:

<u>STATE</u>	<u>PNO</u>	<u>PNAME</u>	<u>ACRES</u>
AZ	7	MULESHOE RANCH	49120
MA	9	DAVID H. SMITH	830
MA	12	MOUNT PLANTAIN	730
MT	2	PINE BUTTE SWAMP	15000

26I. This exercise has the same query objective as Exercise 25Ma. Your solution should specify an inline view.

Reference the DEPARTMENT and EMPLOYEE tables. Assume that management is considering adjusting each department's budget. Each new departmental budget might be changed to twice the total salary of all employees who work in the department. Before implementing this change, management asks you to produce a report that displays each department's number, name, current budget, and the proposed new budget. If a department does not have any employees, then display a null value for the proposed new budget. The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>	<u>NEWDBUDGET</u>
10	ACCOUNTING	75000.00	4800.00
20	INFO. SYS.	20000.00	28000.00
30	PRODUCTION	7000.00	-
40	ENGINEERING	25000.00	1000.00

26J. This exercise modifies the preceding Exercise 26I. (It also has same query objective as Exercise 25N.) The user does not want to see any null values in the report. Therefore, if a department does not have any employees, the new budget should be the same as the current budget. The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>	<u>NEWBUDGET</u>
10	ACCOUNTING	75000.00	4800.00
20	INFO. SYS.	20000.00	28000.00
30	PRODUCTION	7000.00	7000.00
40	ENGINEERING	25000.00	1000.00

Code two solutions which specify inline views.

The first solution should use the COALESCE function to substitute the current BUDGET value for a null value in the NEWBUDGET column.

The second solution should specify a CASE-expression to substitute the current BUDGET value for a null value in the NEWBUDGET column.

26K. Extend Sample Query 26.3. (Display all information about the highest paid employee in each department that has employees.) Also display the department name along with the department number. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>	<u>DNAME</u>
2000	LARRY	2000.00	10	ACCOUNTING
4000	SHEMP	500.00	40	ENGINEERING
6000	GEORGE	9000.00	20	INFO. SYS.

26L. Same query objective as Exercise 23S. Consider changing each DEPARTMENT.BUDGET value to a value that is equal to the largest BUDGET value minus 10% of the department's current BUDGET value. Display each department number, name, current budget, and the adjusted budget. (Hint: Review Sample Query 26.5) The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>	<u>ADJBUDGET</u>
10	ACCOUNTING	75000.00	67500.00
20	INFO. SYS.	20000.00	73000.00
30	PRODUCTION	7000.00	74300.00
40	ENGINEERING	25000.00	72500.00

26M. Reference the PARTSUPP and LINEITEM tables. For each part, display its part number, its largest purchase price, and its lowest selling price, if this largest purchase price is greater than its lowest selling price. The result should look like:

<u>PNO</u>	<u>MAXPAID</u>	<u>MINSOLD</u>
P3	12.50	12.00
P7	3.50	3.00
P8	5.00	4.00

Hint: Specify two inline views that look like:

<u>BOUGHT</u>		<u>SOLD</u>	
<u>PNO</u>	<u>MAXPS</u>	<u>PNO</u>	<u>MINLI</u>
P1	11.00	P1	11.50
P3	12.50	P3	12.00
P4	12.00	P4	13.00
P5	11.00	P5	11.00
P6	4.00	P6	5.00
P7	3.50	P7	3.00
P8	5.00	P8	4.00

26N. Specify an inline view to satisfy Sample Query 25.8. Reference the EMPLOYEE table. Consider adjusting each employee's salary to a value that is equal to the employee's *departmental* average salary plus 5% of the employee's current salary. Display each employee number, name, and current salary, followed by the adjusted salary. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>ADJUSTEDSALARY</u>
1000	MOE	2000.00	4766.66
2000	LARRY	2000.00	1300.00
3000	CURLY	3000.00	4816.66
4000	SHEMP	500.00	525.00
5000	JOE	400.00	1220.00
6000	GEORGE	9000.00	5116.66

- 26O. Use an inline view to enhance Sample Query 23.16. Reference the EMPLOYEE table. Consider the impact of adjusting each employee's salary to a value that is equal to the overall average of all current salaries plus 5% of the employee's current salary. Display each employee number, name, current salary, and adjusted salary. Also, display a narrative label "SALARY INCREASED" or "SALARY DECREASED" or "NO CHANGE" in the last column in result table. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>ADJSAL</u>	<u>NARRATIVE</u>
1000	MOE	2000.00	2916.66	SALARY INCREASED
2000	LARRY	2000.00	2916.66	SALARY INCREASED
3000	CURLY	3000.00	2966.66	SALARY DECREASED
4000	SHEMP	500.00	2841.66	SALARY INCREASED
5000	JOE	400.00	2836.66	SALARY INCREASED
6000	GEORGE	9000.00	3266.66	SALARY DECREASED

- 26P. Reference the PRESERVE table. For each row, if its FEE value is not zero, calculate the ratio of ACRES divided by FEE. The result should look like:

<u>PNAME</u>	<u>RATIO</u>
HASSAYAMPA RIVER	220.00
RAMSEY CANYON	126.66
PAPAGONIA-SONOITA CREEK	400.00

Review the page after Sample Query 7.6 and Exercise 23Zi. Your solution should specify an inline view.

- 26Q. Code an alternative solution for Sample Query 23.11. Do not display information about any employee with a SALARY value of 2000.00. For other employees, display the ENO, ENAME, SALARY, and ratio of SALARY/(SALARY-2000.00) if this ratio is greater than or equal to 2.00. (Notice that, when a SALARY value equals 2000.00, we have a divide-by-zero problem.)

## WITH-Clause: Common Table Expressions

This chapter introduces the WITH-clause which defines a *Common Table Expression (CTE)*.

A CTE offers the same basic functionality as an inline view. To demonstrate this point, the first seven sample queries in this chapter have the same query objectives as the first seven sample queries in the preceding Chapter 26. Furthermore, Sample Query 27.7 will show that a CTE offers an additional advantage not offered by an inline view.

The following skeleton-code illustrates a WITH-clause that defines a CTE called HAPPY.

<pre>WITH HAPPY AS   (SELECT _____    FROM   _____    WHERE  _____) SELECT _____ FROM   HAPPY WHERE  _____</pre>	<pre>} } }</pre>	<pre>CTE "Main-SELECT"</pre>
--	------------------	------------------------------

The WITH-clause is specified *before* the Main-SELECT. The above WITH-clause generates a CTE (an intermediate-result table) called HAPPY. HAPPY is subsequently referenced within the Main-SELECT, and, like an inline view, disappears when the statement terminates.



## Mundane Tutorial Example

Sample Query 26.1 defined an intermediate-result table (TEMP20) by coding an inline view where the Sub-SELECT was coded in a FROM-clause. The following sample query defines the same intermediate-result table with the same name (TEMP20) by coding the same Sub-SELECT within a WITH-clause.

**Sample Query 27.1:** Same as Sample Query 26.1. Generate a CTE (an intermediate-result table) called TEMP20 that contains the ENAME and SALARY values of every employee who works in Department 20. Then reference TEMP20 to display the ENAME and SALARY + \$250.00 values for every row with a SALARY value that is less than \$8,000.00.

```
WITH TEMP20 AS
  (SELECT ENAME, SALARY
   FROM EMPLOYEE
   WHERE DNO = 20)

SELECT ENAME, SALARY + 250.00
FROM TEMP20
WHERE SALARY < 8000.00
```

<u>ENAME</u>	<u>SALARY + 250.00</u>
MOE	2250.00
CURLY	3250.00

**Syntax:** The basic syntax of the WITH-clause is:

```
WITH CTE-name AS (SELECT . . .)
```

Here, TEMP20 inherits its column-names (ENAME and SALARY) from the Sub-SELECT. The following Sample Query 27.2 will show how to explicitly assign column-names to a CTE.

**Logic:** The intermediate-result generated by the CTE looks like:

<u>TEMP20</u>	
<u>ENAME</u>	<u>SALARY</u>
MOE	2000.00
CURLY	3000.00
GEORGE	9000.00

Execution of the Main-SELECT that references TEMP20 produces the final result.

## Minimum of Maximum Values (“Mini-Max” Value)

**Sample Query 27.2:** Same as Sample Query 26.2. Reference the EMPLOYEE table. Determine the maximum salary in each department. Then display the smallest of these maximum values. (I.e., Display the “min of the maxes.”)

```
WITH TMAXES (DNO, MAXSAL) AS
  (SELECT DNO, MAX (SALARY)
   FROM EMPLOYEE
   GROUP BY DNO)
SELECT MIN(MAXSAL) MINMAX
FROM TMAXES
```

```
MINMAX
500.00
```

**Syntax:** Unlike the preceding sample query, the CTE name (TMAXES) is followed by column names (DNO, MAXSAL) specified within parentheses. In particular, *if* the Sub-SELECT generates a derived value (e.g., MAX (SALARY)), then a column-name *must* be assigned. Alternatively, a column-name can be assigned within the Sub-SELECT as shown below.

```
WITH TMAXES AS
  (SELECT DNO, MAX (SALARY) MAXSAL
   FROM EMPLOYEE
   GROUP BY DNO)
SELECT MIN(MAXSAL) MINMAX
FROM TMAXES
```

**Logic:** The intermediate-result generated by the CTE looks like:

```
TMAXES
DNO  MAXSAL
10   2000.00
20   9000.00
40   500.00
```

Executing the Main-SELECT against TMAXES produces the final result.

**Alternative Solutions:** Sample Queries 23.17 and 26.2.

The following sample query has already been solved by coding a regular Sub-SELECT that returns multiple columns (Sample Query 23.14), a correlated Sub-SELECT (Sample Query 25.1), and an inline view (Sample Query 26.3). Many users would consider the following statement that codes a CTE to be the simplest solution.

**Sample Query 27.3:** Same as Sample Query 26.3. Display all information about the highest paid employee in each department that has at least one employee.

```

WITH TMAXES (DNO, MAXSAL) AS
  (SELECT DNO, MAX (SALARY)
   FROM EMPLOYEE
   GROUP BY DNO)

SELECT ENO, ENAME, SALARY, EMPLOYEE.DNO
FROM EMPLOYEE, TMAXES
WHERE EMPLOYEE.DNO = TMAXES.DNO
AND EMPLOYEE.SALARY = TMAXES.MAXSAL

```

ENO	ENAME	SALARY	DNO
2000	LARRY	2000.00	10
4000	SHEMP	500.00	40
6000	GEORGE	9000.00	20

**Logic:** This is the CTE (TMAXES) that was specified in the previous Sample Query 27.2. The Sub-SELECT returns an intermediate result table called TMAXES that looks like:

TMAXES	
DNO	MAXSAL
10	2000.00
20	9000.00
40	500.00

TMAXES is joined with EMPLOYEE. If an EMPLOYEE row has a DNO value that equals TMAXES.DNO, and a SALARY value that equals TMAXES.MAXSAL, then this EMPLOYEE row corresponds to the highest paid employee in the DNO department.

**Alternative Solution:** Sample Query 26.3.

## Exercises

The following exercises have the same query objectives as Exercises 26A-26D. Solve by coding WITH-clauses. These exercises reference the EMPLOYEE table.

27A. Determine the total salary for each department. Then display the largest of these totals. The result should look like:

```
LARGESTTOTAL  
14000.00
```

27B. Determine the average salary for each department. Then display the smallest of these averages. The result should look like:

```
SMALLESTAVG  
500.00
```

27C. Display all information about the lowest paid employee in each department. The result should look like:

```
ENO  ENAME      SALARY  DNO  
1000 MOE          2000.00  20  
4000 SHEMP      500.00   40  
5000 JOE         400.00   10
```

27D. For each department, display all information about every departmental employee who has a salary that is greater than or equal to the average salary for the department. The result should look like:

```
ENO  ENAME      SALARY  DNO  
2000 LARRY      2000.00  10  
4000 SHEMP      500.00   40  
6000 GEORGE     9000.00  20
```

## WITH-Clause Generates Multiple CTEs

The following WITH-clause specifies multiple CTEs.

**Sample Query 27.4:** Same as Sample Query 26.4. Whenever a department's highest paid employee has a salary that exceeds the smallest departmental budget, we want to display both DNO values along with the corresponding maximum salary and minimum budget values.

```
WITH
  TMAXES (DNO, MAXSAL) AS
    (SELECT DNO, MAX (SALARY)
     FROM EMPLOYEE
     GROUP BY DNO),
  DMIN (DNO, BUDGET) AS
    (SELECT DNO, BUDGET
     FROM DEPARTMENT
     WHERE BUDGET = (SELECT MIN (BUDGET)
                     FROM DEPARTMENT))
SELECT TMAXES.DNO MAXSALDEPT, TMAXES.MAXSAL,
       DMIN.DNO MINBUDGETDEPT, DMIN.BUDGET MINBUDGET
FROM TMAXES, DMIN
WHERE TMAXES.MAXSAL > DMIN.BUDGET
```

<u>MAXSALDEPT</u>	<u>MAXSAL</u>	<u>MINBUDGETDEPT</u>	<u>MINBUDGET</u>
20	9000.00	30	7000.00

**Syntax & Logic:** The WITH-clause specifies two CTEs called TMAXES and DMIN. A comma must separate the specification of each CTE. Then the Main-SELECT joins these CTEs.

### Exercise:

27E. Same as Exercise 26E2. Reference the PARTSUPP and LINEITEM tables. For each part sold, the actual selling price (LIPRICE) is always greater than or equal to the part's purchase price (PSPRICE). Hence, a part's average selling price is always greater than or equal to its average purchase price. Display information about any part where the difference between these averages is less than 75 cents. For any such part, display its part number followed by its average purchase price and average selling price. The result should look like:

<u>PNO</u>	<u>AVGPS</u>	<u>AVGLI</u>
P7	3.00	3.50

## CTE with One-Row and One-Column

The following WITH-clause generates a CTE called TEMP with just one row and one column.

**Sample Query 27.5:** Same as Sample Query 26.5. Reference the EMPLOYEE table. Consider the impact of adjusting each employee's salary to a value that is equal to the overall average of all current salaries plus 5% of the employee's current salary. Display each employee number, name, current salary, and adjusted salary.

```
WITH TEMP (EMPAVGSAL) AS
  (SELECT AVG (SALARY) FROM EMPLOYEE)
SELECT ENO, ENAME, SALARY,
       TEMP.EMPAVGSAL + (.05*SALARY) ADJSAL
FROM EMPLOYEE, TEMP
```

ENO	ENAME	SALARY	ADJSAL
1000	MOE	2000.00	2916.66
2000	LARRY	2000.00	2916.66
3000	CURLY	3000.00	2966.66
4000	SHEMP	500.00	2841.66
5000	JOE	400.00	2836.66
6000	GEORGE	9000.00	3266.66

**Logic:** The intermediate-result generated by the CTE looks like:

<u>TEMP</u>
<u>EMPAVGSAL</u>
2816.66

The Main-SELECT does not specify a join-condition. Hence the system executes a cross-product on EMPLOYEE and TEMP to produce the following cross-product intermediate result:

ENO	ENAME	SALARY	DNO	EMPAVGSAL
1000	MOE	2000.00	20	2816.66
2000	LARRY	2000.00	10	2816.66
3000	CURLY	3000.00	20	2816.66
4000	SHEMP	500.00	40	2816.66
5000	JOE	400.00	10	2816.66
6000	GEORGE	9000.00	20	2816.66

Finally, the Main-SELECT produces the final result.

**Alternative Solutions:** Sample Queries 23.16 and 26.5.

The following sample query uses coding techniques that were introduced in the previous two sample queries. It specifies a WITH-clause that defines two common table expressions, TEMP1 and TEMP2. Note that TEMP2 has just one row with one column.

**Sample Query 27.6:** Same as Sample Query 26.6. For each department referenced in the EMPLOYEE table, display its department number and its average departmental salary followed a comment indicating that the departmental average is less than, greater than, or equal to the overall average salary.

```
WITH TEMP1 (DNO, DEPTAVGSAL) AS
      (SELECT DNO, AVG (SALARY)
       FROM EMPLOYEE
       GROUP BY DNO),
      TEMP2 (EMPAVGSAL) AS
      (SELECT AVG (SALARY) FROM EMPLOYEE)
SELECT TEMP1.DNO, TEMP1.DEPTAVGSAL,
CASE
  WHEN TEMP1.DEPTAVGSAL < TEMP2.EMPAVGSAL
    THEN 'LESS THAN OVERALL AVERAGE SALARY'
  WHEN TEMP1.DEPTAVGSAL = TEMP2.EMPAVGSAL
    THEN 'EQUAL TO OVERALL AVERAGE SALARY'
  ELSE   'GREATER THAN OVERALL AVERAGE SALARY'
END COMMENTS
FROM TEMP1, TEMP2
```

<u>DNO</u>	<u>DEPTAVSAL</u>	<u>COMMENTS</u>
10	1200.00	LESS THAN OVERALL AVERAGE SALARY
20	4666.66	GREATER THAN OVERALL AVERAGE SALARY
40	500.00	LESS THAN OVERALL AVERAGE SALARY

**Syntax and Logic:** Nothing New.

**Alternative Solutions:** Sample Queries 23.17 and 26.6.

## Multiple References to Same CTE

The two inline views in Sample Query 26.7 coded identical Sub-SELECTs that generated identical intermediate-result tables. The following WITH-clause allows you to avoid this undesirable redundancy.

**Sample Query 27.7:** Same as Sample Query 26.7. Display the “minimum of the maximum” departmental salaries and the DNO value of the department that has this “mini-max” value.

```
WITH TMAXES AS
    (SELECT DNO, MAX (SALARY) MAXSAL
     FROM EMPLOYEE
     GROUP BY DNO)

SELECT DNO, MAXSAL MINIMAX
FROM TMAXES
WHERE MAXSAL = (SELECT MIN (MAXSAL)
                FROM TMAXES)
```

<u>DNO</u>	<u>MINIMAX</u>
40	500.00

**Syntax & Logic:** The WITH-clause codes a CTE called TMAXES that is referenced twice in the Main-SELECT.

**Alternative Solution:** Sample Query 26.7.

### Exercise:

27F. Same as Exercise 26F. Display the department number and total salary of the department having the largest total salary. The result should look like:

<u>DNO</u>	<u>LARGESTTOTAL</u>
20	14000.00



## Summary

When compared to an inline view, a Common Table Expression offers four potential advantages.

1. The Main-SELECT can specify multiple references to same CTE. This advantage was illustrated in previous Sample Query 27.7. The WITH-clause defined TMAXES that was referenced twice in the Main-SELECT. This advantage does not apply to inline views.
2. CTEs may be friendlier: Defining an intermediate-result table just once at the beginning of a statement seems to be conceptually tidier than defining an inline view in a FROM-clause "somewhere in the middle" of a statement. Also, regarding the previous advantage, the SELECT statement for Sample Query 27.7 is smaller and simpler than the statement for Sample Query 26.7 because the CTE is only specified once.
3. Possible efficiency benefits: Sample Query 26.7 defined two identical Sub-SELECTs. Some systems might execute both of these Sub-SELECTs. This double execution of the same Sub-SELECT is obviously inefficient. This possible inefficiency would not apply to a CTE where its Sub-SELECT is (presumably) only executed once.
4. Recursive CTEs: The WITH-clause can be used to define a "recursive" CTE, a topic that will be introduced in Chapter 30.

Finally, you must understand both inline views and common table expressions, especially if you will read SELECT statements that were written by other users.

## Summary Exercises

The following Exercises 27G-27Q have the same query objectives as Exercises 26G-26Q. Utilize the WITH-clause to satisfy these query objectives.

27G. Reference the EMPLOYEE table. Determine the average salary in each department. Then display the largest of these averages. The result should look like:

```
MAXAVGSAL
4666.66
```

27H. Reference the PRESERVE table. For each state, display the state code and preserve's number, name, acreage for every preserve that is larger than the average preserve acreage for the state. The result should look like:

<u>STATE</u>	<u>PNO</u>	<u>PNAME</u>	<u>ACRES</u>
AZ	7	MULESHOE RANCH	49120
MA	9	DAVID H. SMITH	830
MA	12	MOUNT PLANTAIN	730
MT	2	PINE BUTTE SWAMP	15000

27I. This exercise has the same query objective as Exercises 25M1 and 26I. Reference the DEPARTMENT and EMPLOYEE tables. Assume that management is considering adjusting each department's budget. Each new departmental budget might be changed to twice the total salary of all employees who work in the department. Before implementing this change, management asks you to produce a report that displays each department's number, name, current budget, and the proposed new budget. If a department does not have any employees, then display a null value for the proposed new budget. The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>	<u>NEWDBUDGET</u>
10	ACCOUNTING	75000.00	4800.00
20	INFO. SYS.	20000.00	28000.00
30	PRODUCTION	7000.00	-
40	ENGINEERING	25000.00	1000.00

27J. This exercise modifies the preceding Exercise 27I. (It also has same query objective as Exercise 25N.) The user does not want to see any null values in the report. Therefore, if a department does not have any employees, the new budget should be the same as the current budget. The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>	<u>NEWBUDGET</u>
10	ACCOUNTING	75000.00	4800.00
20	INFO. SYS.	20000.00	28000.00
30	PRODUCTION	<b>7000.00</b>	<b>7000.00</b>
40	ENGINEERING	25000.00	1000.00

Code two solutions which specify WITH-clauses.

The first solution should use the COALESCE function to substitute the current BUDGET value for a null value in the NEWBUDGET column.

The second solution should specify a CASE-expression to substitute the current BUDGET value for a null value in the NEWBUDGET column.

27K. Extend Sample Query 27.3. (Display all information about the highest paid employee in each department that has at least one employee.) Also display the department name along with the department number. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>	<u>DNAME</u>
2000	LARRY	2000.00	10	ACCOUNTING
4000	SHEMP	500.00	40	ENGINEERING
6000	GEORGE	9000.00	20	INFO. SYS.

27L. Consider changing each DEPARTMENT.BUDGET value to a value that is equal to the largest BUDGET value minus 10% of the department's current BUDGET value. Display each department number, name, current budget, and the adjusted budget. (Hint: Review Sample Query 27.5) The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>BUDGET</u>	<u>ADJBUDGET</u>
10	ACCOUNTING	75000.00	67500.00
20	INFO. SYS.	20000.00	73000.00
30	PRODUCTION	7000.00	74300.00
40	ENGINEERING	25000.00	72500.00

27M. Reference the PARTSUPP and LINEITEM tables. For each part, display its part number, its largest purchase price, and its lowest selling price, if this largest purchase price is greater than its lowest selling price. The result should look like:

<u>PNO</u>	<u>MAXPAID</u>	<u>MINSOLD</u>
P3	12.50	12.00
P7	3.50	3.00
P8	5.00	4.00

Hint: Specify two common table expressions for the following tables that look like:

<u>BOUGHT</u>		<u>SOLD</u>	
<u>PNO</u>	<u>MAXPS</u>	<u>PNO</u>	<u>MINLI</u>
P1	11.00	P1	11.50
P3	12.50	P3	12.00
P4	12.00	P4	13.00
P5	11.00	P5	11.00
P6	4.00	P6	5.00
P7	3.50	P7	3.00
P8	5.00	P8	4.00

27N. Specify a WITH-clause to satisfy Sample Query 25.8. Reference the EMPLOYEE table. Consider adjusting each employee's salary to a value that is equal to the employee's *departmental* average salary plus 5% of the employee's current salary. Display each employee number, name, and current salary, followed by the adjusted salary. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>ADJUSTEDSALARY</u>
1000	MOE	2000.00	4766.66
2000	LARRY	2000.00	1300.00
3000	CURLY	3000.00	4816.66
4000	SHEMP	500.00	525.00
5000	JOE	400.00	1220.00
6000	GEORGE	9000.00	5116.66

270. Reference the EMPLOYEE table. Consider the impact of adjusting each employee's salary to a value that is equal to the overall average of all current salaries plus 5% of the employee's current salary. Display each employee number, name, current salary, and adjusted salary. Also, display a narrative label "SALARY INCREASED" or "SALARY DECREASED" or "NO CHANGE" in the last column in result table. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>ADJSAL</u>	<u>NARRATIVE</u>
1000	MOE	2000.00	2916.66	SALARY INCREASED
2000	LARRY	2000.00	2916.66	SALARY INCREASED
3000	CURLY	3000.00	2966.66	SALARY DECREASED
4000	SHEMP	500.00	2841.66	SALARY INCREASED
5000	JOE	400.00	2836.66	SALARY INCREASED
6000	GEORGE	9000.00	3266.66	SALARY DECREASED

27P. Reference the PRESERVE table. For each row, if its FEE value is not zero, calculate the ratio of ACRES divided by FEE. The result should look like:

<u>PNAME</u>	<u>RATIO</u>
HASSAYAMPA RIVER	220.00
RAMSEY CANYON	126.66
PAPAGONIA-SONOITA CREEK	400.00

Hint: Review the page after Sample Query 7.6 and Exercise 26P. .

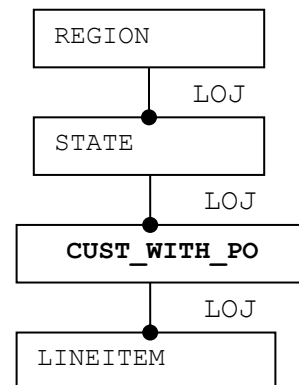
27Q. Code an alternative solution for Sample Query 23.11 (and Exercise 26Q). Do not display information about any employee with a SALARY value of 2000.00. For other employees, display the ENO, ENAME, SALARY, and ratio of SALARY/(SALARY-2000.00) if this ratio is greater than or equal to 2.00. (Notice that, when a SALARY value equals 2000.00, we have a divide-by-zero problem.)

27R. Same as Sample Query 20.15: Display the following information about regions, states, customers, purchase-orders, and line-items.

- Display the region number and name of all regions, including regions without any states.
- Display the code and name for all states, including states without any customers.
- Display customer number and name for those customers that have at least one purchase-order.
- Display each customer's purchase-order numbers, including numbers for purchase-orders that do not have any line-items.
- Display each line-item's line-number and part-number values.

Specify a CTE called `CUST_WITH_PO` which executes an INNER JOIN to join the `CUSTOMER` and `PUR_ORDER` tables. Then the following code would represent a sequence of join-operations that traverse a five-level hierarchy.

```
FROM REGION R  
LEFT OUTER JOIN STATE ST  
ON R.RNO = ST.RNO  
LEFT OUTER JOIN CUST_WITH_PO CWPO  
ON ST.STCODE = CWPO.STCODE  
LEFT OUTER JOIN LINEITEM LI  
ON PO.PONO = LI.PONO
```

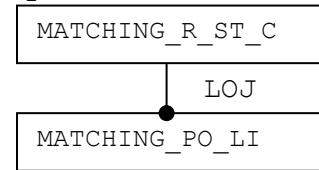


27S. Same as Sample Query 20.16: Display the following information about regions, states, customers, purchase-orders, and line-items.

- Display the region number of any region that has at least one state.
- Display the code of any state that has at least one customer.
- Display the number and name of all customers, including customers without purchase-orders.
- Display each customer's purchase-order numbers if the purchase-order has at least one line-item.
- Display the line-number and corresponding part-number of each line-item.

Think of the INNER JOIN operations forming two intermediate join-results in tables called MATCHING\_R\_ST\_C and MATCHING\_PO\_LI. Then the following code would represent a sequence of LEFT OUTER JOIN operations that traverse a four-level hierarchy.

```
FROM MATCHING_R_ST_C RSTC  
LEFT OUTER JOIN MATCHING_PO_LI POLI  
ON RSTC.CNO = POLI.CNO
```



## CREATE VIEW Statement

This is an optional chapter that is primarily directed towards application developers. However, super-users are invited to read on.

Assume you find yourself frequently coding the same Sub-SELECT to specify an inline view or a common table expression. It may be more convenient to execute a CREATE VIEW statement that permanently defines the Sub-SELECT and assigns it a name for future reference. For example, assume you have a table called JUNK. To create a view on this table called HAPPY, you could execute a CREATE VIEW statement that looks like:

```
CREATE VIEW HAPPY
AS SELECT A, B, C
FROM JUNK
WHERE X = 100
```

Thereafter, you can execute any number of SELECT statements that reference the same HAPPY view as shown below.

```
SELECT * FROM HAPPY;

SELECT * FROM HAPPY WHERE B = C;

SELECT A, C FROM HAPPY WHERE B = 9;
```

These SELECT statements illustrate reuse of the HAPPY table (view), versus recoding the same inline view or CTE in multiple SELECT statements.

Your DBA might not grant you permission to execute a CREATE VIEW statement. Instead, she may offer to create the HAPPY view for you. This could be productive because the DBA may want to allow other users to access HAPPY.



## CREATE VIEW Statement

The following CREATE VIEW statement specifies the Sub-SELECT that defined the TEMP20 inline view in Sample Query 26.1.

**Sample Statement 28.1:** Create a view called TEMP20 that contains the ENAME, and SALARY columns from the EMPLOYEE table and only contains those rows with a DNO value of 20.

```
CREATE VIEW TEMP20
  AS   SELECT ENAME, SALARY
        FROM EMPLOYEE
        WHERE DNO = 20
```

**System Response:** The system should respond with a message indicating the successful creation of a view. Verify the creation of this view by executing: SELECT \* FROM TEMP20. The result table should look like:

<u>ENAME</u>	<u>SALARY</u>
MOE	2000.00
CURLY	3000.00
GEORGE	9000.00

**Syntax:** The basic CREATE VIEW syntax is:

```
CREATE VIEW view-name AS
  SELECT ...
  FROM ...
  WHERE ...
```

In this example, the TEMP20 view inherits its column-names (ENAME, and SALARY) from the EMPLOYEE table. Sample Statement 28.4 will show how to explicitly assign names to a view's columns.

**Logic:** The CREATE VIEW statement assigns a name (TEMP20) to the Sub-SELECT and saves the named Sub-SELECT in the system's data dictionary.

**Terminology:** A "base table" is created by a CREATE TABLE statement. A "view" is created by a CREATE VIEW statement. A view is a "virtual table;" it is not a base table. Where appropriate, we will use the generic term "table" to encompass both base tables and views.

## SELECT Statement References a View

The following two sample queries demonstrate that a SELECT statement references a view just like any other table.

**Sample Query 28.2:** Reference the TEMP20 table (view). Display the ENAME column followed by SALARY + 250.00 for any row with a SALARY value that is less than 8,000.00. (Same query objective as Sample Queries 26.1 and 27.1).

```
SELECT ENAME, SALARY+250.00
FROM   TEMP20
WHERE  SALARY < 8000.00
```

<u>ENAME</u>	<u>SALARY+250.00</u>
MOE	2250.00
CURLY	3250.00

**Sample Query 28.3:** Reference TEMP20. Display the ENAME column followed by SALARY+100.00 for any row where its ENAME value ends with the letter E.

```
SELECT ENAME, SALARY+100.00
FROM   TEMP20
WHERE  ENAME LIKE '%E'
```

<u>ENAME</u>	<u>SALARY+100.00</u>
MOE	2100.00
GEORGE	9100.00

**Logic:** When TEMP20 was created, the system did *not* immediately execute its Sub-SELECT and save the result table. Instead, when the above sample queries were executed, the system accessed the data dictionary to retrieve the Sub-SELECT associated with TEMP20, and then it executed this Sub-SELECT to produce the TEMP20 intermediate-result table.

*The system executed the view's Sub-SELECT twice, once for each of the above sample queries. It did not execute the Sub-SELECT once and save the result. This approach is necessary because the underlying base table (EMPLOYEE) could have been changed sometime after you executed Sample Query 28.2 but before you executed Sample Query 28.3.*

**Comment:** Unless told otherwise, some users may (incorrectly) think that TEMP20 is just another base table. This should not cause any problems.

## Renaming Columns in Views

The CREATE VIEW statement shown in Sample Statement 28.1 illustrated that, sometimes, you do not need to explicitly assign column-names in a view. The next example illustrates a circumstance where you *must* explicitly assign column-names.

**Sample Statement 28.4:** Create a view called DEPTSTATSV. For each department that has at least one employee, DEPTSTATSV should contain the department's DNO value, a column called MAXSAL with the largest departmental salary, a column called MINSAL with the smallest departmental salary, and a column called TOTALSAL with the sum of departmental salaries.

```
CREATE VIEW DEPTSTATSV (DNO, MAXSAL, MINSAL, TOTALSAL)
AS
SELECT  DNO, MAX(SALARY), MIN(SALARY), SUM(SALARY)
FROM    EMPLOYEE
GROUP BY DNO
```

**System Response:** The system should respond with a message indicating the successful creation of a view. Verify this by executing: SELECT \* FROM DEPTSTATSV. The result should look like:

DNO	MAXSAL	MINSAL	TOTALSAL
10	2000.00	400.00	2400.00
20	9000.00	2000.00	14000.00
40	500.00	500.00	500.00

**Syntax:** CREATE VIEW \_\_\_\_\_ (COL1, COL2, COL3, ...)  
AS SELECT...  
FROM ...  
WHERE ...

This view *requires* the explicit assignment of column-names because some of the view's columns (e.g., MAXSAL, MINSAL, and TOTALSAL) contain derived data (i.e., data that is derived by executing an expression or function).

**View Naming Convention:** Notice the letter V in DEPTSTATSV. Some designers will specify the letter V as the first or last letter in a view's name. This convention is optional. Notice that we did not follow this convention when we created the DEPT20 view.

The following sample queries reference the DEPTSTATSV view.

**Sample Query 28.5:** Reference DEPTSTATSV. Display the smallest MAXSAL value. (I.e., Display the "mini-max" of departmental salaries. This is the same query objective as Sample Queries 26.2 and 27.2)

```
SELECT MIN (MAXSAL) FROM DEPTSTATSV
```

```
MINMAX  
500.00
```

**Sample Query 28.6:** Reference DEPTSTATSV. Display the mini-max departmental salary along with DNO value of the department that has this mini-max value. (Same query objective as Sample Queries 26.5 and 27.5).

```
SELECT DNO, MAXSAL MINIMAX  
FROM DEPTSTATSV  
WHERE MAXSAL = (SELECT MIN (MAXSAL)  
                FROM DEPTSTATSV)
```

```
DNO  MINIMAX  
40   500.00
```

**Sample Query 28.7:** Reference the EMPLOYEE and DEPTSTATSV tables. Display all information about the highest paid employee in each department. (Same query objective as Sample Queries 26.3 and 27.3).

```
SELECT E.ENO, E.ENAME, E.SALARY, E.DNO  
FROM EMPLOYEE E, DEPTSTATSV D  
WHERE E.DNO = D.DNO  
AND E.SALARY = D.MAXSAL
```

```
ENO  ENAME      SALARY  DNO  
2000 LARRY       2000.00  10  
4000 SEMP      500.00   40  
6000 GEORGE    9000.00  20
```

## View Definition Can Specify *Almost* Any SQL Operation

The following CREATE VIEW statement effectively pre-joins the EMPLOYEE and DEPARTMENT tables to represent data about employees and related departments.

**Sample Statement 28.8:** Create a view called EMPDEPTV that contains data from the EMPLOYEE and DEPARTMENT tables. The view should contain the employee number, name, and salary of every employee along with the number, name, and budget of the employee's department.

```
CREATE VIEW EMPDEPTV
AS
SELECT E.ENAME, E.ENO, E.SALARY, E.DNO,
       D.DNAME, D.BUDGET
FROM   EMPLOYEE E, DEPARTMENT D
WHERE  E.DNO = D.DNO
```

**System Response:** The system should respond with a message indicating the successful creation of the view. Verify the creation of this view by executing: SELECT FROM EMPDEPTV. The result should look like:

ENAME	ENO	SALARY	DNO	DNAME	BUDGET
MOE	1000	2000.00	20	INFO. SYS.	20000.00
LARRY	2000	2000.00	10	ACCOUNTING	75000.00
CURLY	3000	3000.00	20	INFO. SYS.	20000.00
SHEMP	4000	500.00	40	ENGINEERING	25000.00
JOE	5000	400.00	10	ACCOUNTING	75000.00
GEORGE	6000	9000.00	20	INFO. SYS.	20000.00

**Syntax:** A view's Sub-SELECT can specify most SELECT clauses (e.g., WHERE, HAVING, INNER JOIN, OUTER JOIN, UNION, etc.). However, there is an issue regarding the ORDER BY clause. In Chapter 1 we noted that tables do not have any predefined sort sequence; and, a view is a virtual table. Hence, in principle, the CREATE VIEW statement should not specify an ORDER BY clause. *But!* Some systems (e.g., ORACLE and SQL Server) allow you to specify an ORDER BY clause in a CREATE VIEW statement. Other systems (e. g., DB2) will reject the ORDER BY clause.

**Logic:** The following page illustrates multiple SELECT statements that reference EMPDEPTV. These statements are smaller and simpler than the corresponding SELECT statements shown in Chapters 16 and 17 because they do not have to specify join-operations.

## Friendlier SELECT Statements

The following SELECT statements do not specify join-operations because EMPDEPTV has effectively pre-joined the DEPARTMENT and EMPLOYEE tables.

**Sample Query 16.4:** For each employee earning a salary that is less than \$3,000.00, display the name of his department followed by his name and salary.

```
SELECT DNAME, ENAME, SALARY FROM EMPDEPTV
WHERE SALARY < 3000.00
```

**Sample Query 16.5:** Only consider employees having a salary less than \$999.00. If any such employee works for a department having a budget that is less than or equal to \$75,000.00, display the department's DNO and BUDGET values along with the employee's ENAME and SALARY values.

```
SELECT DNO, BUDGET, ENAME, SALARY FROM EMPDEPTV
WHERE SALARY < 999.00 AND BUDGET <= 75000.00
```

**Sample Query 16.7:** Only consider departments with employees. Display the DNAME and BUDGET values for any such department having a budget that is greater than or equal to \$25,000.00.

```
SELECT DISTINCT DNAME, BUDGET FROM EMPDEPTV
WHERE BUDGET >= 25000.00
```

**Sample Query 17.2:** For all employees, display their ENO and SALARY values along with the DNO and BUDGET values of the department they work for. Also display the ratio of each employee's salary to his department's budget.

```
SELECT ENO, SALARY, DNO, BUDGET, SALARY/BUDGET
FROM EMPDEPTV
```

**Sample Query 17.4.1:** For each department that has employees, display the department name and total salary of all employees who work in that department.

```
SELECT DNAME, SUM (SALARY) FROM EMPDEPTV
GROUP BY DNAME
```

**Sample Query 17.9:** Does any employee have a salary that exceeds one third of his own departmental budget? If yes, display the employee's name, salary, and department number, followed by the budget amount for his department.

```
SELECT ENAME, SALARY, DNO, BUDGET FROM EMPDEPTV
WHERE SALARY > (BUDGET * 0.333)
```

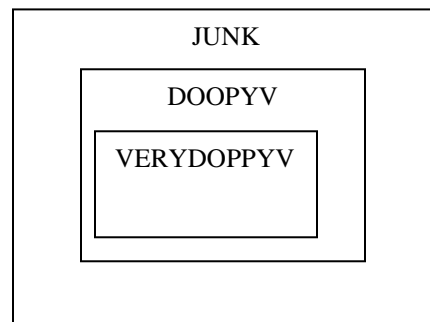
## Other View Topics

**Views Defined on Views:** A view can be defined on another view.

Assume you have created a base table called JUNK with columns A, B, C, and X. The following CREATE VIEW statements create two views. The first CREATE VIEW statement creates the DOOPYV view that references the JUNK table. *The second CREATE VIEW statement creates the VERYDOOPYV view that references the first view (DOOPYV).*

```
CREATE VIEW DOOPYV AS
  SELECT A, B, C
  FROM JUNK
  WHERE X = 100;

CREATE VIEW VERYDOOPYV AS
  SELECT A, C
  FROM DOOPYV
  WHERE B = 2000,
```



A SELECT statement that references the VERYDOOPYV view indirectly references the DOOPYV view which then indirectly references the JUNK table.

**View Dependencies:** This is another know-your-data consideration. In general, if you have permission to create/drop tables and views, then you must be aware of any view dependencies (i.e., which views depend upon which tables, and which views depend upon other views).

If you drop a base table, then any view which is directly or indirectly dependent on this table becomes invalid. Hence, all SELECT statements that reference this view would fail. For example, if you dropped the above JUNK table, then any reference to the VERYDOOPYV or DOOPYV views would return an error. (Information about view dependencies is stored in the system's data dictionary.)

**DROP VIEW:** The DROP VIEW statement is similar to DROP TABLE. To drop a view, you execute:

```
DROP VIEW view-name
```

After dropping a view, any attempt to execute a SELECT statement that directly or indirectly references the view will return an error.

## Summary

The CREATE VIEW statement assigns a name to a Sub-SELECT and saves the Sub-SELECT in the system's data dictionary. (The CREATE VIEW statement does *not* execute the Sub-SELECT and save the result table.)

Subsequently, when a SQL statement references a view, the view's Sub-SELECT is executed to generate an intermediate-result table. Discussion of Sample Queries 28.2 and 28.3 noted that, although this intermediate-result disappears when the statement terminates, the view's definition remains in the system's data dictionary for future reuse.

**Advantages of Views:** Views offer many advantages. Some (not all) of these advantages are described below.

**Simplification of User Queries:** A designer can create views that contain derived data such as a statistical summary (Sample Statement 28.4) or pre-joined data from multiple tables (Sample Statement 28.8). Then users can formulate relatively simple queries against these views. Frequently, users who write queries against a "table" are unaware that the table is really a view.

**Support Database Security:** A DBA might not give every departmental manager access all EMPLOYEE data because it contains confidential data (SALARY) about all employees. Assume a department manager should only be allowed to access data about employees who work in his department. In this circumstance, the DBA could deny all managers access to the EMPLOYEE table. Instead, the DBA would create a view for each department (e.g., DEPT20 view) and grant access privileges on each view to the appropriate manager.

**Avoid De-normalized Base Tables:** (Optional Reading): Appendix 16A described de-normalized tables and problems associated with executing INSERT, UPDATE, and DELETE statements against such tables. Therefore, designers generally avoid creating de-normalized base tables. Instead, to facilitate friendlier user queries, some designers create de-normalized views (e.g., Sample Statement 28.8).



**Potential Disadvantages of Views:** Again, no free lunch.

**Knowing-Your-Data:** Creating too many views can challenge knowing-your-data because the same data may be stored in multiple "tables." For example, the ENO, ENAME, and SALARY values are (logically) stored in both the EMPLOYEE table and the EMPDEPTV "table." Also, from the DBA's perspective, too many views increase the complexity of managing view dependency.

**View Update Problem:** This chapter does not illustrate any INSERT, UPDATE, or DELETE statements that reference a view. In some circumstances, you can successfully execute these statements against a view, with the intention of indirectly modifying an underlying base table. However, in other circumstances, executing an INSERT, UPDATE, or DELETE statement against a view may produce an error. In practice, most DBAs require all INSERT, UPDATE, or DELETE statement to reference base tables. (Theory Comment: This book does not discuss the "view update problem.")

### **Comments for Application Developers**

The CREATE VIEW statement is part of SQL's Data Definition Language. Therefore, you might not have permission to execute this statement in a production environment. If you think that a view would be helpful, you could ask your DBA to create one for you. If your request is denied, you can still specify an inline view or a common table expression in your SELECT statement. Alternatively, you might create a "Temporary Table" that will be described in Appendix 28B. You are encouraged to read this appendix. It describes how a temporary table can preserve an intermediate-result for a "little while longer" so that you can reference it in multiple SQL statements.

## Summary Exercises

Exercises 28A - 28F assume that the DEPTSTATSV and EMPDEPTV tables (views) already exist because Sample Statements 28.4 and 28.8 have been executed.

28A. Same query objective as Exercise 27A. Reference the DEPTSTATSV table. Determine the total salary for each department. Then display the largest of these totals. The result should look like:

<u>LARGESTTOTAL</u>
14000.00

28B. Same query objective as Exercise 27C. Reference the EMPLOYEE and DEPTSTATSV tables. Display all information about the lowest paid employee in each department. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>DNO</u>
1000	MOE	2000.00	20
4000	SHEMP	500.00	40
5000	JOE	400.00	10

28C. Same query objective as Exercise 27E. Reference the DEPTSTATSV table. Display the department number and total salary of the department having the largest total salary. The result should look like:

<u>DNO</u>	<u>LARGESTTOTAL</u>
20	14000.00

28D. Expand upon the previous Exercise 28C. Reference the DEPARTMENT and DEPTSTATSV tables. Display the department name along with the department number. The result should look like:

<u>DNO</u>	<u>DNAME</u>	<u>LARGESTTOTAL</u>
20	INFO. SYS.	14000.00

28E. Reference the EMPDEPTV table. Display the employee number and name of any employee whose salary exceeds \$2,000.00 and works in the Accounting Department. The result should look like:

```

ENO  ENAME
2000  LARRY

```

28F. Reference the EMPDEPTV and DEPTSTATSV tables. For any department where the difference between the largest and smallest employee salaries exceeds \$3,000.00, display the department name, followed by the name and salary of each of its employees. The result should look like:

```

DNAME      ENAME      SALARY
INFO. SYS.   MOE           2000.00
INFO. SYS.   CURLY         3000.00
INFO. SYS.   GEORGE        9000.00

```

28G. (a) The DEPTSTATSV view does not contain the average salary for each department. Create another view called DEPTSTATSV2 that contains the same data as DEPTSTATSV plus another column called AVGSAL that contains the average salary for each department.

(b) Reference the above DEPTSTATSV2. Display the smallest of average departmental salaries. The result should look like:

```

MINAVG
500.00

```

(c) Reference the EMPLOYEE and DEPTSTATSV2 tables. For each department, display all information about every departmental employee who has a salary that is greater than or equal to the average salary for the department. The result should look like:

```

ENO  ENAME  SALARY  DNO
2000  LARRY   2000.00  10
4000  SEMP    500.00   40
6000  GEORGE  9000.00  20

```

(d) Drop the DEPTSTATSV2 view.

28H. This exercise is a variation of Exercise 27M.

Reference the PARTSUPP and LINEITEM tables to create a view called BOUGHT\_SOLD\_STATS. This view contains each part number, its largest purchase price (MAXPAID), and its lowest selling price (MINSOLD). Hint: Modify the solution to Exercise 27M. Data for the BOUGHT\_SOLD\_STATS view should look like:

<u>PNO</u>	<u>MAXPAID</u>	<u>MINSOLD</u>
P1	11.00	11.50
P3	12.50	12.00
P4	12.00	13.00
P5	11.00	11.00
P6	4.00	5.00
P7	3.50	3.00
P8	5.00	4.00

Display any row in BOUGHT\_SOLD\_STATS where this largest purchase price is greater than its lowest selling price. The result should look like:

<u>PNO</u>	<u>MAXPAID</u>	<u>MINSOLD</u>
P3	12.50	12.00
P7	3.50	3.00
P8	5.00	4.00

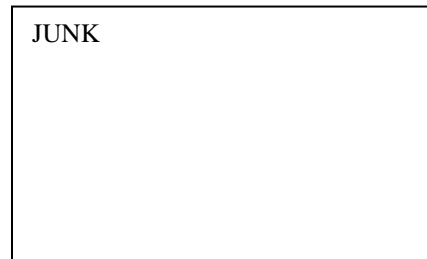
Drop the BOUGHT\_SOLD\_STATS view

## Appendix 28A: Efficiency & Theory

This appendix examines query rewrite in the context of view processing. Previous commentary indicated that, when a SELECT statement references a view, the system accesses and executes the Sub-SELECT associated with the view to generate an intermediate-result. While this is logically correct, the system may be able to improve efficiency by rewriting the SELECT statement before execution. We describe this process below.

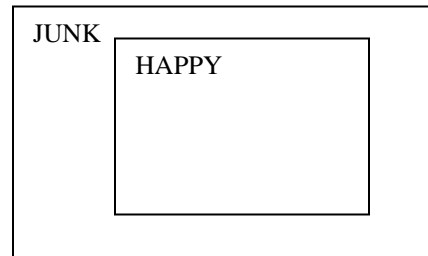
### **Theory:** Sets and Subsets

Assume that JUNK is a valid base table (a set).



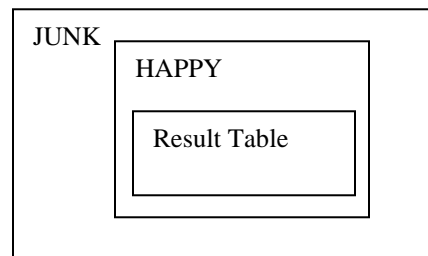
**View Definition:** Assume the DBA created the following view called HAPPY that is based on JUNK. Notice that HAPPY is a subset of JUNK.

```
CREATE VIEW HAPPY
AS SELECT A, B, C
   FROM   JUNK
   WHERE  X = 100
```



**User Query:** Assume a user executes a SELECT-statement that references HAPPY. Because the result table is a subset of HAPPY, it is also a subset of JUNK.

```
SELECT A, B
FROM HAPPY
WHERE C = 9
```



**Optimizer Query Rewrite:** Given the previous SELECT-statement, the optimizer modifies it by changing its FROM-clause and WHERE-clause. It substitutes JUNK for HAPPY in the FROM-clause; and then it modifies the WHERE-clause by AND-connecting the view's WHERE-condition (X = 100). The rewritten SELECT statement looks like:

```
Rewritten Statement:      SELECT A, B
                           FROM  JUNK      ←
                           WHERE C = 9
                           AND X = 100    ←
```

**Potential Efficiency:** Sometimes, view processing can improve efficiency. For example, assume that:

JUNK is very large (millions of rows)  
90% of these rows have an X value of 100.  
There is no index on column X.  
1% of the rows have a C value of 9.  
There is an index on column C.

Without Query Rewrite: Re-consider the Sub-SELECT specified in the CREATE VIEW statement.

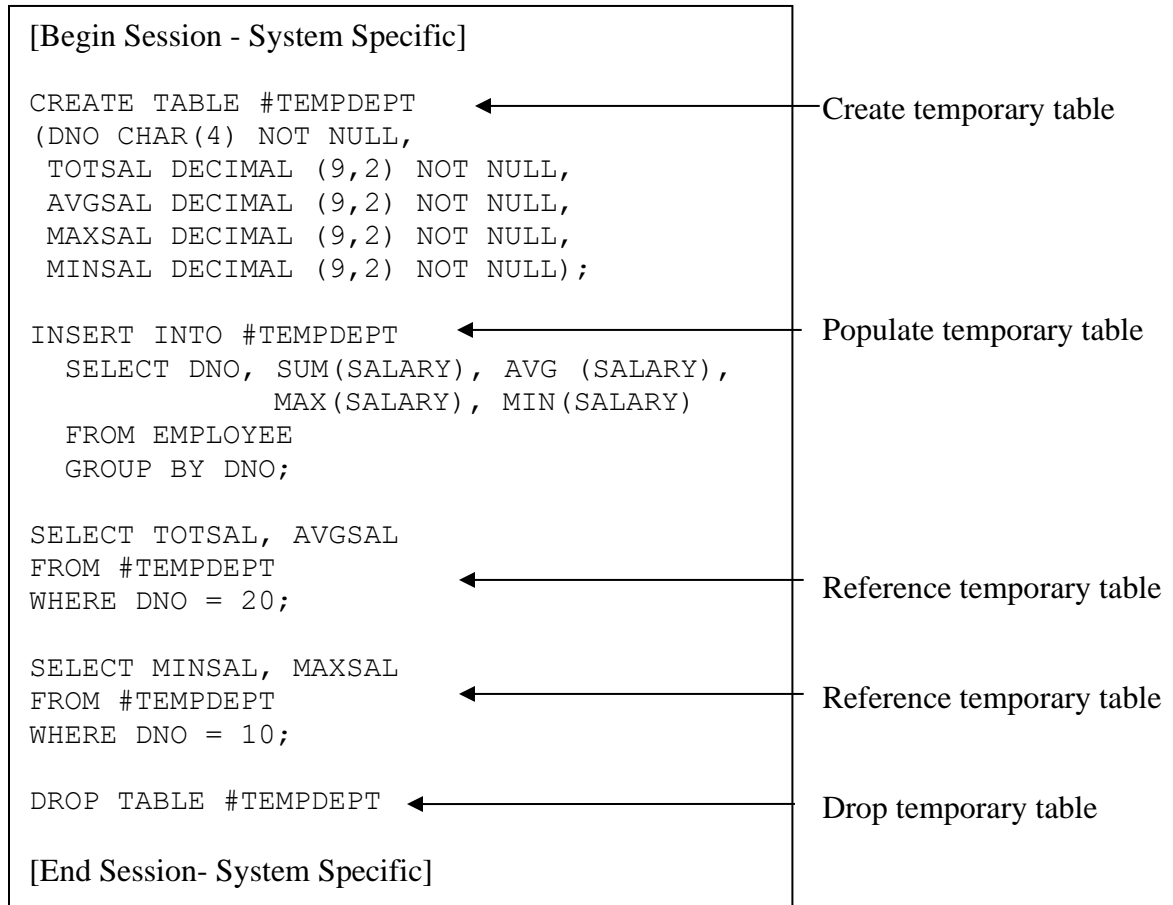
```
SELECT A, B, C
FROM JUNK
WHERE X = 100
```

If the system initially executed this statement, it would build HAPPY as an intermediate-result table by scanning the very large JUNK table to find and save 90% of its rows where X = 100. HAPPY would be very large and would not have any index on it. Thereafter, the system would scan this large intermediate-result to find all rows where C = 9.

With Query Rewrite: When the system executes the rewritten statement (shown above), it would use the index on column C to directly access the matching 1% of rows where C = 9. As each row is retrieved, the system examines its X value. If this X value is 100, the row placed in the final result table. This plan is more efficient because there is no scanning of a large table, and no storing and subsequent scanning of a large intermediate-result table.



**Sample Session:** The following figure outlines the creation of a temporary table. Note that multiple SELECT-statements reference this table. This example uses SQL Server syntax where the names of temporary tables begin with the # symbol.



### Characteristics of Temporary Tables:

- A temporary table is private to the user's session. No other user can access this table.
- Unlike creating a base table, no metadata describing a temporary table is stored in the system's data dictionary.
- Executing INSERT, UPDATE, and DELETE statements against a temporary table is usually efficient because the system incurs less overhead cost associated with transaction processing. The following chapter on Transaction Processing (COMMIT & ROLLABCK) introduces this topic.

**Recommendation:** We have only outlined basic concepts. If you intend to utilize temporary tables, you must read specific details in your SQL Reference Manual.



## Concluding Efficiency Appendices: 28C - 28D - 28E

Previous efficiency appendices presented “bits and pieces” of information about query performance and optimization. The following three appendices (especially Appendix 28E) bring these bits and pieces together into a more coherent framework. (These appendices are not explicitly related to the content of Chapter 28. They are identified as Appendix 28C-28E only because they follow Appendices 28A-28B.)

### Appendix 28C: Explaining an Application Plan

Previous efficiency appendices have described how physical design decisions and SQL code can influence an optimizer to include specific operations (e.g., table scan) within an application plan. However, we have not yet shown how you can obtain a complete description of all operations within an application plan. This appendix will do so. It describes how you can ask your optimizer to generate an “explanation” of your application plan.

### Appendix 28D: Optimizer Hints

After viewing an explanation of your application plan, you might conclude that this plan should be modified to improve efficiency. For this reason, most systems provide some method for a user to specify a “hint.” A hint encourages the optimizer to generate a different (presumably more efficient) application plan. This appendix will describe hints along with their advantages and disadvantages.

### Appendix 28E: Tuning SELECT Statements

This final chapter appendix concludes by presenting a general strategy for tuning SELECT-statements. It incorporates the material presented in the previous two appendices.

Note: Book-Appendix-III presents a brief summary of this book’s chapter appendices. This summary also presents additional commentary on efficiency considerations.

## Appendix 28C: Explaining an Application Plan

Assume that you have just executed a `SELECT` statement that satisfied your query objective. Now you might ask: What is the application plan that was generated by the optimizer for this statement? You might ask this question because your `SELECT` statement had a slow response time, and you want to tune it. Or, you are simply curious. Explaining an application plan can be a productive learning experience.

Many front-end tools provide an "Explain-Button" that produces a graphical/textual explanation of an application plan. *However, you must have a general understanding of the optimization process before you can interpret this explanation.* This book's efficiency appendices offered a *starting point*. With this basic knowledge, you should be able to read and understand your system's reference manuals that discuss query performance, query optimization, and the explanation of application plans.

The explanation of an application plan describes a sequence of steps where each step identifies an internal operation with related information about:

- A table scan.
- An index access. If an index was used, was it used to provide direct access to rows in a table, support an index-only search, or return rows in some useful row sequence?
- Join-methods. For each join-operation, the plan indicates the join-method (e.g., Nested-Loop, Match-Merge).
- Join-sequence. If three or more tables were joined, the plan identifies which two tables were joined first, second, etc.

Also, if the `SELECT`-statement was executed, an explanation might contain statistics indicating the processing time for each step in the plan. (E.g., How long did it take to scan a table? )

**Graphical Notation:** Most Explain-Methods produce some graphical representation of an application plan. In the following graphs, a table is represented by a rectangle; a table scan is represented by an oval; a sort operation is represented by a trapezoid; an index access is represented by triangle; and a join-operation is represented by a diamond. These graphical plans should be read in a bottom-up, left-to-right manner.

**Sample Application Plan:**

```
SELECT *
FROM T1
WHERE COLX = 100
```

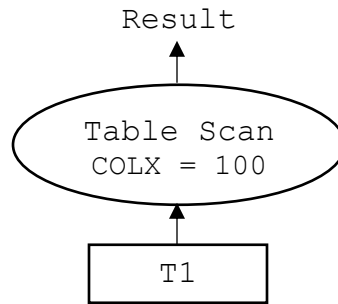
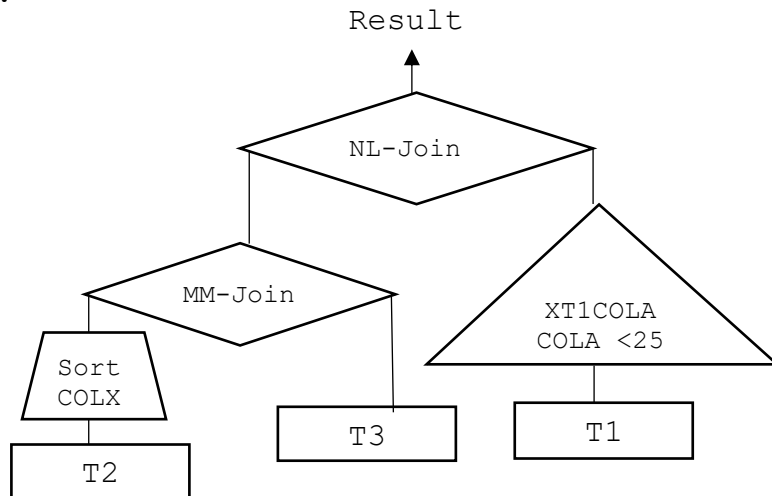


Table T1 is accessed by a Table Scan operation to retrieve rows with a COLX value of 100.

**Sample Application Plan:**

```
SELECT *
FROM T1, T2, T3
WHERE T1.COLA = T2.COLB
AND T2.COLX = T3.COLY
AND T1.COLA < 25
```



The optimizer decides to initially join Tables T2 and T3 using a Match-Merge (MM) join-method. But first, Table T2 must be sorted by COLX to facilitate the logic of the match-merge join.

The next step uses a Nested-Loop (NL) join-method to join the T2-T3 join-result to Table T1 by using the XT1COLA index to directly access rows in T1.

**"EXPLAIN" Statements:** The activation of the Explain-Button will (under-the-hood) execute a SQL statement that generates the explanation. Examples of these statements are:

- DB2: EXPLAIN statement
- ORACLE: EXPLAIN PLAN statement
- SQL Server: SHOWPLAN\_ALL statement

You can directly execute these statements. However, these statements are relatively complex, have many options, and may return their explanations in a rather unfriendly format, typically as rows in a special "Plan-Table." Hence, it is usually easier to use your front-end tool.

**Setup of Explain-Feature:** Some front-end tools only require that you activate the Explain-Button immediately before you execute a SELECT statement. Other front-end tools require you to follow some multi-step process to setup an explanation. See your reference manual for details.

**Optional Exercise:** Explain a few SELECT-statements from this book. (Note: Because all sample tables are very small, it is highly unlikely that the plan will utilize an index for a direct-access search. Also, indexes have only been created for the primary-key and other unique columns.)

## Appendix 28D: Optimizer Hints

Assume you want to tune the following SELECT statement because its execution time is very slow.

```
SELECT T1.PK, T1.COLA, T1.COLB
FROM   T1, T2
WHERE  T1.PK = T2.FK
AND    T2.FK > 100
```

You have already examined the metadata information about tables T1 and T2 and concluded that the optimizer is using accurate statistics. Also, you have determined that there is a unique index (XT1PK) on the primary-key T1.PK column, and there is a non-unique index (XT2FK) on the foreign-key T2.FK column.

You have explained this SELECT statement, and the explanation shows that the optimizer decided *not* to use the XT2FK index. Then, after “playing optimizer” in your own mind, you conclude that using this index could enhance efficiency. You believe this index should improve the T2.FK > 100 condition, and it may also improve the join-operation.

Next you formulate and explain multiple equivalent SELECT statements. However, you become grumpy because all application plans fail to utilize this presumably useful XT2FK index. You curse your (apparently) stupid optimizer! Then, as a *last act of desperation*, you decide to specify a “hint.”

**Hints:** A hint is a “directive” that a user can present to the optimizer telling it to include a specific action in an application plan. Here, you want to tell the optimizer to utilize the XT2FK index.

This directive is called a “hint” because, sometimes, the optimizer may not follow the directive. For example, the DBA can setup the optimizer to ignore user-specified hints; or, unknown to the user, the DBA has dropped the XT2FK index.

**Coding Hints:** Different systems provide different methods for specifying a hint. The next page illustrates the specification of hints (in bold font) using ORACLE and SQL Server.

```
ORACLE:          SELECT /*+ INDEX(T2 XT2FK) */  
                  T1.PK, T1.COLA, T1.COLB  
FROM    T1, T2  
WHERE   T1.PK = T2.FK  
AND     T2.FK > 100
```

```
SQL Server:     SELECT T1.PK, T1.COLA, T1.COLB  
FROM    T1, T2 WITH (INDEX(XT2FK))  
WHERE   T1.PK = T2.FK  
AND     T2.FK > 100
```

After specifying this hint, an explanation should show that the optimizer decides to use the XT2FK index. Hopefully, this hint improves efficiency.

Your SQL reference manual will describe all hints provided by your system.

#### **Cautionary Comments about Hints:**

- What if the DBA drops the XT2FK index before this SELECT statement is executed? In this circumstance, the optimizer will ignore the hint and re-optimize the statement.
- Without dropping the XT2FK index, assume the DBA creates another index that provides better performance than the XT2FK index. The optimizer would still obey the hint. In this circumstance, the hint becomes a directive to do the wrong thing.
- If users specify too many hints, the overall effectiveness of the DBA'S physical database design could be compromised.
- Finally, a new version of your database system may include an improved optimizer that could render your hint irrelevant.

**Conclusion:** It's fun to "play optimizer." But, for very practical reasons, you should think twice before specifying a hint in a production environment.

**Final Comment:** Hits are philosophically repugnant. An ideal optimizer should not require any performance hints from a user. However, occasionally, a hint can be useful.

## Appendix 28E: Tuning SELECT Statements

Users rarely have permission to take all possible actions that can improve the efficiency of a SELECT statement. However, occasionally, a user can take an action that may be helpful. Below we describe these actions in a *conceptual overview* of tuning a SELECT statement.

Assume your SELECT statement has a slow response time.

Step-1: Only tune a SELECT-statement if the payoff could be significant. Ask: Do I have a real problem? For example, assume your SELECT statement is just a "little bit" slow. Then, most likely, your statement is "good enough - hence no problem." Alternatively, what if this SELECT statement is embedded within an application program or stored procedure that is executed many times every minute/second? This could be a real problem. Although a slow response time may not be a problem for the single execution of a statement, the total cost for multiple executions could be significant.

Step-2: Don't bother the DBA unless it becomes necessary. If necessary, collect relevant information to present to her. The following steps identify this information.

Step-3: Collect statistics about the size of relevant tables. Although you may have an accurate estimate about the size of each table, the optimizer might not have access to this information. Examine the statistical information stored in your system's dictionary tables to verify that the stored statistics are realistic estimates. (The MetaData Panel in your front-end tool might be able to display this information.) If these statistics are not realistic estimates, you may have discovered the source of your problem. Ask the DBA to update these statistics. Then re-execute your SELECT-statement. Proceed to the following Step-4 if the response time is still too slow.

Step-4: Explain your SELECT statement. If you think the optimizer did a good job, proceed to Step-7. Otherwise, if you think the optimizer could have generated a more efficient plan, proceed to the following Step-5.

Step-5: Rewrite your SELECT statement as one or more equivalent SELECT statements. Then, explain each statement. (In this book, we have suggested alternative statements for many sample queries. Also, your SQL reference manual may identify problematic SQL code and propose alternative coding solutions.) Execute an alternative statement if its application plan differs from the original problematic plan. If response time has improved, you may have solved your problem. Otherwise, proceed to the following Step-6.

Step-6: If all else fails, and you are desperate, consider coding a hint. Then explain and test your statement with the hint. However, even if the response time improves, do not commit to implementing the hint in a production environment until after you have moved onto the following Step-7.

Step-7: Describe your problem to the DBA. Present the following information to her.

- Size of relevant tables and other statistics (e.g., histograms of relevant column values).
- Description of relevant indexes.
- Equivalent SELECT statements with corresponding explanations, and response times for those statements with different application plans.
- Hints that improved response time.
- Finally, you might consider "playing DBA." For example, although you are not authorized to create a new index, you can still speculate about the benefit of a new index and present your speculation to your DBA.



## What can the DBA do to improve my response time?

A lot!

The DBA controls many performance factors that *have not been addressed in this book*. These include creating a special table structure (e.g., clustered table), a special type of index (e.g., bitmap index), increasing the size of a memory buffer, assigning multiple tables to the same tablespace, and partitioning a table to facilitate parallel processing. Furthermore, in some circumstances, the DBA may be able to store your data on a faster data storage device. (See the following page.) Finally, the DBA will be aware of potential problems outside the database system per se, such as bottlenecks associated with your communications network or operating system.

Tunning a single SELECT statement is a local concern (your concern) whereas physical database design is a global concern. Physical database design attempts to produce an overall efficient design for all applications which specify many different types of SQL statements. The DBA must consider overall thruput, not just the response times for a specific application program. For example, Appendix 2A presented a design scenario where the DBA, after considering overall cost/benefit tradeoffs, rejected a user's apparently reasonable request to create a new index.

Finally, your DBA might agree with your suggestion to create a new index to help your SELECT statement. This new index may also help other SQL applications. Therefore, your DBA might say: "Good idea. Let's do it and see what happens."

## Data Storage Devices

In this book, we have assumed that database data is stored on conventional disk, which is relatively slow because of mechanical disk arm movement and rotational delay. Some modern database systems offer the following more efficient data storage options.

- Massively Parallel Processing (MPP): Data is stored on arrays of conventional disks, controlled by multiple controllers that can process data in parallel.
- Solid State Drive (SSD): An SSD does not suffer the mechanical drawbacks associated with conventional disk. Today, SSD has become very common and is found in personal computers.
- Main Memory Database: MMP storage and SSD are external to main memory. Like conventional disk, data must be transferred between these storage devices and main memory. A Main Memory Database is very fast because it eliminates this data transfer time by keeping most (maybe all) database data in a special type of main memory where data is not lost when the system crashes.

Data storage technology will continue to improve into the foreseeable future, and this technology will entail different methods of physical design and query optimization. This observation supports the following conclusion.

## Conclusion

Blazing fast data storage technology  
+  
Good physical database design  
+  
Very smart optimizers

*imply that, in the near future, most applications developers who design and implement conventional business applications will encounter fewer SQL tuning problems.*

This conclusion justifies this book's focus on correctness.

[However, more challenging application domains present caveats to this conclusion. See Book Appendix IV.]

This page is intentionally blank.

---

# PART VII

## Special Topics

This part of the book includes two chapters on two special topics, Transaction Processing and Recursive Queries. After reading the following overview, you may decide to bypass one or both of the following chapters.

### **Chapter 29 - Transaction Processing: COMMIT & ROLLBACK**

Chapter 15 introduced SQL's DML statements (INSERT, UPDATE, and DELETE), and Chapter 24 showed that these statements may contain Sub-SELECTs. Those chapters, like all previous chapters, presented sample statements that were executed using an interactive front-end tool. However, in most real-world applications, DML statements are executed as embedded statements within an application program or stored procedure. Within this context, the idea of "transaction processing" becomes relevant.

This chapter is a "must read" if you are an applications developer who intends to write programs/procedures with embedded SQL. (Super-users are invited to read this chapter because the basic concepts are not complex, and the chapter is relatively short.) This chapter introduces transaction processing concepts and two relevant SQL statements, COMMIT and ROLLBACK.

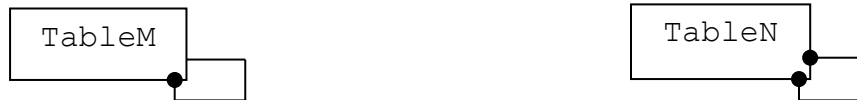
The optional appendices are rather long and extend our discussion of transaction processing beyond the COMMIT and ROLLBACK statements.

## Chapter 30 - Recursive Queries

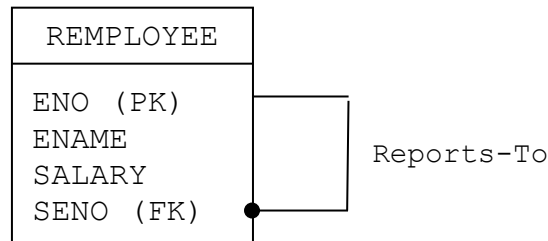
Thus far, this book has implicitly assumed that a database relationship is a relationship between two *different* tables as illustrated below.



Chapter 30 will introduce "Recursive Queries" that reference "Recursive Tables" which are tables that participate in "Recursive Relationships." A recursive relationship relates a table to itself as illustrated below.



Section-A presents sample queries that reference the recursive REEMPLOYEE table shown below. This table is recursive because its SENO column contains the employee number of an employee's supervisor, who is another employee and therefore is represented by another row in the *same* REEMPLOYEE table. The SENO column is a foreign-key that references the primary-key (ENO) in some other REEMPLOYEE row.



Section-B presents a recursive design that specifies a many-to-many recursive relationship. Finally, Section-C presents some special case scenarios where a recursive query can be satisfied by coding a non-recursive SELECT statement.

# 29

## Transaction Processing: COMMIT and ROLLBACK

This short chapter is primarily directed towards application developers who embed DML statements (INSERT, UPDATE, and DELETE) within application programs and stored procedures.

This chapter begins by describing the concept of a *transaction* as a “logical unit of work.” In this book, a transaction will always contain one or more SQL statements.

Casually speaking, we will see that: the COMMIT statement tells the system to terminate the transaction and make its database changes permanent; and the ROLLBACK statement tells the system to terminate the transaction and undo its changes to the database.

The following pages present four database update scenarios where each scenario executes a COMMIT or ROLLBACK statement. The first three scenarios illustrate COMMIT and/or ROLLBACK statements embedded within a program/procedure. The fourth scenario demonstrates that you can execute a COMMIT or ROLLBACK statement using an interactive front-end tool.

Transaction processing also supports *database concurrency and database recovery*. Database concurrency is introduced in Appendix 29B.

## What is a Transaction?

A *transaction* is a logical unit of work. The classic example of a transaction is the transfer of funds from one bank account to another. A business user would consider this transfer of funds to be a single business transaction. However, from a SQL perspective, this transaction involves the execution two UPDATE statements against two different tables. For example, the following statements transfer \$100.00 from a checking account to a savings account.

```
UPDATE CHECKING_ACCT
SET CK_BALANCE = CK_BALANCE - 100.00
WHERE CK_ACCOUNT_NO = 123;
```

```
UPDATE SAVINGS_ACCT
SET SAVE_BALANCE = SAVE_BALANCE + 100.00
WHERE SAVE_ACCOUNT_NO = 456;
```

To force these two UPDATE statements to operate as a single logical unit of work, we need some way to bundle both statements within a transaction. Specifically, there must be some way to designate a transaction's starting point and its termination point.

### Start-Transaction

```
UPDATE CHECKING_ACCT
SET CK_BALANCE = CK_BALANCE - 100.00
WHERE CK_ACCOUNT_NO = 123;
```

```
UPDATE SAVINGS_ACCT
SET SAVE_BALANCE = SAVE_BALANCE + 100.00
WHERE SAVE_ACCOUNT_NO = 456;
```

### Terminate-Transaction

**Start-Transaction:** Some systems (e.g., SQL Server) support a BEGIN TRANSACTION statement that explicitly designates the start of a transaction. The following scenarios do not specify this statement. Instead, we will assume that an internal Start-Transaction operation automatically occurs just before the first SQL statement is executed.

**Terminate-Transaction:** Execution of a COMMIT or ROLLBACK statement terminates a transaction. If neither statement is executed, and the program successfully terminates, the system will generate an *automatic-commit*. Alternatively, if the program terminates because of an error, the system will generate an *automatic-rollback*.

## Scenario-1: COMMIT

The following figure illustrates a section of code within a program/procedure. This code illustrates a transaction that contains two UPDATE statements. Here we assume an "all-went-well" scenario where all SQL and non-SQL statements execute without problems.

```
1.  UPDATE CHECKING_ACCT
     SET BALANCE = BALANCE - 100.00
     WHERE CK_ACCOUNT_NO = 123;

2:   Other non-SQL processing/logic;

3.  UPDATE SAVINGS_ACCT
     SET BALANCE = BALANCE + 100.00
     WHERE SAVE_ACCOUNT_NO = 456;

4:   Other non-SQL processing/logic;

5.  COMMIT;
```

Here, a transaction starts when the database system is asked to execute the first SQL statement. This means that an *implicit "start-transaction"* is executed just before the system executes the first UPDATE statement.

After successful executions of the above Steps 1-4, the COMMIT statement in Step-5 tells the system to:

- Make all changes permanent, and then
- Terminate the transaction.

The COMMIT statement will also return some system-specific return code that implies a "successful commit" operation.

If, after executing COMMIT, the program/procedure looped back to the first UPDATE statement, then, because COMMIT terminated the transaction, this UPDATE statement designates the beginning of a new transaction.

Comment: Executing COMMIT causes the system to execute many internal operations that are not described in this chapter. (Some of these operations are described in the following Appendix 29B.)



## Scenario-2: Automatic Rollback

The following example attempts to execute the same code shown in Scenario-1. However, this example illustrates a "something-went-wrong" scenario where a program error or some system-level error (e.g., main memory is lost) causes the program or the entire system to "crash."

1.	UPDATE CHECKING_ACCT SET CK_BALANCE = CK_BALANCE - 100.00 WHERE CK_ACCOUNT_NO = 123;	} This code is successfully executed
2:	Other non-SQL processing/logic;	<b>Error ("Crash!")</b>
3.	UPDATE SAVINGS_ACCT SET SAVE_BALANCE = SAVE_BALANCE + 100.00 WHERE SAVE_ACCOUNT_NO = 456;	} This code is <b>not</b> executed
4:	Other non-SQL processing/logic;	} ↓
5.	COMMIT	

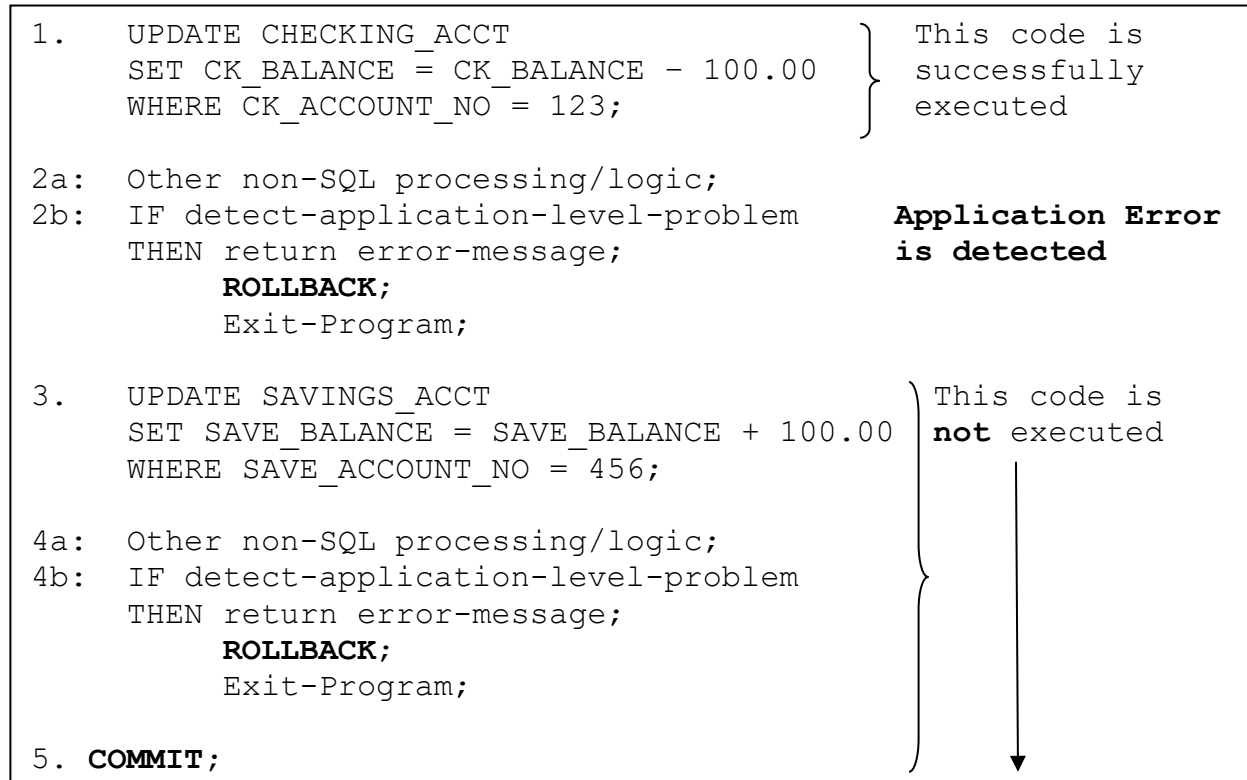
Assume the system did not take any special action after the program crashed. Then \$100.00 would have been deducted from the checking account, but \$100.00 would not have been added to the savings account. The database would not be consistent. To avoid this "database inconsistency" problem, after the crash event, the system automatically initiates a *database recovery* process which includes an *automatic-rollback operation*.

This auto-rollback operation "undoes" the effect of the first UPDATE statement. In effect, the first UPDATE statement was never executed. Also, all users/programs/procedures are prevented from accessing any corrupted data. Therefore, this rollback operation supports the desirable property of database **consistency**. If the first UPDATE were not rolled back, the database would be in an inconsistent state.

Also, observe that both Scenario-1 and Scenario-2 demonstrate that a transaction is **atomic**. This means that a transaction enforces the "all-or-nothing" execution of all statements within the scope of the transaction. All statements were executed in the previous Scenario-1. No statements were (effectively) executed in the above Scenario-2.

### Scenario-3: Explicit User-Specified ROLLBACK

The following example illustrates another "something-went-wrong" scenario that involves the explicit coding of a ROLLBACK statement. This scenario illustrates an application-level problem that is detected by the user's code.



Step-2b specifies code that tests for a detectable application-level error. For example, this code might ask if the new updated CK\_BALANCE value contains a negative value. If it does, the program returns some error-message (e.g., "Insufficient Funds"), executes the ROLLBACK statement that undoes the preceding UPDATE statement, and terminates the transaction. Note this is not a "crash" scenario. The insufficient funds scenario is not unusual, and this program proceeds to a normal termination.

Again, we note that this transaction is atomic. Both of the UPDATE statements are executed, or neither of them is executed. Either way, after the transaction terminates, the database is in a consistent state.

## Scenario-4: Interactive COMMIT/ROLLBACK

We have already noted that COMMIT and ROLLBACK statements are usually embedded within an application program/procedure. However, you can also execute these statements within an interactive environment. This capability can be useful when testing INSERT, UPDATE, or DELETE statements.

By default, most front-end tools automatically commit successful INSERT, UPDATE, or DELETE statements. However, these tools usually provide some mechanism that allows you to temporarily disable this auto-commit behavior. This allows you to execute an INSERT, UPDATE, or DELETE statement, observe the changes made by the statement, and then ask the system to commit or rollback these changes. In the following scenario, Step-1 disables the auto-commit operation. The following steps make changes to the DEMO2 table and then explicitly rollback these changes.

1. SET AUTOCOMMIT OFF; ←\*\*\*\* this code varies by system

2. SELECT \* FROM DEMO2;

<u>I1</u>	<u>D1</u>	<u>V1</u>	<u>F1</u>
-10	-8.82	Julie Martyn	HELLO
-5	-5.28	JESSIE MARTYN	GOOD
0	0.00	Janet Martyn	BY
2	6.42	Frank	BYE
9	9.98	Wally	HYY

3. DELETE FROM DEMO2 WHERE I1 <> 0;

4. SELECT \* FROM DEMO2;

<u>I1</u>	<u>D1</u>	<u>V1</u>	<u>F1</u>
0	0.00	Janet Martyn	BY

5. ROLLBACK;

6. SELECT \* FROM DEMO2;

<u>I1</u>	<u>D1</u>	<u>V1</u>	<u>F1</u>
-10	-8.82	Julie Martyn	HELLO
-5	-5.28	JESSIE MARTYN	GOOD
0	0.00	Janet Martyn	BY
2	6.42	Frank	BYE
9	9.98	Wally	HYY

- Step-1: Each front-end tool has its own specific method for disabling and enabling an auto-commit operation. (The illustrated SET AUTOCOMMIT statement works in ORACLE's SQL\*Plus.)
- Step-2: Display the DEMO2 table before changes are made.
- Step-3: Delete some rows from DEMO2. (You could also execute multiple DML statements to make additional changes to DEMO2.)
- Step-4: Display DEMO2 to observe the changes made by the preceding DELETE statement.
- Step-5: ROLLBACK undoes the effect of the preceding DELETE statement. (If Step-3 had executed multiple DML statements, ROLLBACK would undo all of these DML statements.)
- Step-6: Display DEMO2 to observe that all changes made during Step-3 were undone.

Alternatively, at Step-5, if you were pleased with the contents of DEMO2 as shown in Step-4, you could have executed a COMMIT statement to commit the changes.

Finally, after executing Step-6, you may wish to re-establish the auto-commit behavior. Again, each front-end tool has its own specific method for enabling auto-commit. (For example, SET AUTOCOMMIT ON would work in ORACLE's SQL\*Plus.)

## **Summary**

Transaction processing is a large and complex topic. The preceding four scenarios only illustrated some basic concepts along with the COMMIT and ROLLBACK statements. The following optional appendices offer a little more insight into this topic. Application developers are encouraged to read both of these appendices.

## Appendix 29A: Theory

This appendix assumes that your transactions will execute concurrently with other transactions, and these other transactions may attempt to access the same row(s) that your transaction will access. In this context, we consider the correctness criterion and four desired properties of concurrent transactions.

**Correctness:** The execution of multiple concurrent transactions is considered to be correct if each transaction produces a result that would be produced under some sequential execution of the transactions. I.e., The transactions are *serializable*.

For example, assume three transactions (T1, T2, and T3) are executing concurrently. The result of executing these transactions is considered to correct if it corresponds to the result produced by any of the following six serial execution sequences.

T1-T2-T3, T1-T3-T2, T2-T1-T3, T2-T3-T1, T3-T1-T2, T3-T2-T1

For a more concrete example, consider two concurrent transactions, TA and TB. These transactions update the same row which has a column with a value of 100. The first transaction (TA) adds 50 to this value. The second transaction (TB) doubles this value. The system will consider *both of the following scenarios to be correct*.

1. TA-TB: TA executes, commits, and terminates before TB starts. Then TB executes, commits, and terminates. In this case the final value is  $(100+50) * 2 = 300$ .
2. TB-TA: TB executes, commits, and terminates before TA starts. Then TA executes, commits, and terminates. In this case the column value is  $(100*2) + 50 = 250$ .

*Again, the transaction processing system will consider either 300 or 250 to be a correct result. This implies two different correct results! However, if you want the result to be 300, then you must (somehow) explicitly execute TA and have it committed before you start TB. Alternatively, if you want the result to be 250, then you must (somehow) explicitly execute TB and have it committed before you start TA.*

**Transaction Properties:** At a conceptual level, the transaction system should support four desirable properties. These properties are represented by the "ACID" acronym; a transaction should be **Atomic, Consistent, Isolated, and Durable.**

ACID merely designates these properties; it does not describe how the system should support these properties. Different systems utilize different techniques to support ACID. The following discussion describes the ACID properties without presenting implementation techniques.

## **1. Atomic**

A transaction is an "all-or-nothing" process. Either all of its SQL statements are successfully executed, or none of its SQL statements are executed.

## **2. Consistent**

Assume the database is in a consistent state when a transaction starts. The system guarantees that the database will be in a consistent state when the transaction terminates. In the previous Scenario-2, the system would have been in an inconsistent state if it had not undone the first UPDATE statement. Data inconsistency was avoided because the system performed an automatic-rollback after the crash event.

Observe that the system will allow some data values to be *temporarily* inconsistent. For example, in the previous "all-went-well" Scenario-1, there was a temporary data inconsistency during the short time period after the first UPDATE operation completed and before the start of the second UPDATE operation. During this time period, the system automatically prevented any other user, program, or procedure from accessing the temporarily inconsistent data.

### 3. Isolated

This property relates to database concurrency where multiple concurrent transactions want to access the same row in a table. Isolation implies that each transaction should be able to execute its SQL statements without considering the presence of any other concurrent transaction. In principle, your concurrent transaction is completely isolated from all other concurrent transactions. In practice, the system prevents potential problems associated with transaction isolation. These problems will be described in the following Appendix 29B.

### 4. Durable

Reconsider the all-went-well Scenario-1 where the program successfully executed and committed the two UPDATE statements. In this scenario we assumed that the modified data was physically written to a *durable* database storage device (presumably a disk) before the transaction terminated. This assumption is correct. However, because data durability should be obvious, why state it?

Answer: When a COMMIT statement is executed, you can *logically* conclude that the modified data is physically written to a durable storage device. But, under-the-hood, for efficiency reasons, the system may undertake some different actions.

In Scenario-1, the database changes produced by the two UPDATE statements may be temporally stored in a (non-durable) memory buffer with the expectation that this buffered data will subsequently be written within a batch of records to the durable database. (This is a common internal efficiency technique.) But what if the system crashes and loses its internal memory before the buffer's contents are written to the durable database? In this circumstance, the database changes would not be durable. Therefore, the system never lets this happen.

To maintain durability, when a COMMIT statement is executed, the system writes the database changes to some durable "*backup log file*" before the transaction terminates. Then, if no system crash occurs, the memory buffer is subsequently written to the durable database. But, if a system crash does occur, a database recovery operation will copy the changed data from the durable log file to the durable database.

## **Appendix 29B: Data Isolation & Efficiency**

You usually do not have to worry about how your system supports the ACID properties. However, in some circumstances, you might be able to reduce your transaction's response-time. This opportunity to improve response-time pertains to factors associated with transaction isolation. This appendix introduces this topic which is organized into the following sections.

### Sec B.1 - Data Isolation Problems

This section describes four data isolation problems that can occur if the system does not take any special action to prevent them.

### Sect B.2 - System Prevents Data Isolation Problems

This section outlines how a system can prevent each of the four data isolation problems. It describes *Locking*, a traditional method used to support data isolation. (An alternative method, *Multi-Versioning*, is not described in this book.)

### Sec B.3 - Adjusting Isolation Levels to Improve Efficiency

The preceding section describes how the system automatically prevents data isolation problems. Therefore, you can ignore these problems. However! In some circumstances, the system may have to expend considerable effort to prevent a data isolation problem, and this could have a negative impact on response-time. This section will describe how an applications developer can, in special case circumstances, reduce efficiency costs associated with data isolation.

### Section B.4 - More about Locking

This section concludes by describing additional locking considerations.



## **B.1. Data Isolation Problems**

We describe four problems associated with data isolation.

1. Lost-Update Problem
2. Dirty-Read Problem
3. Non-Repeatable-Read Problem
4. Phantom-Row Problem

The first problem, the Lost-Update Problem, can occur when multiple concurrent transactions want to *update* the same row.

The other three problems can occur when a transaction wants to update a row that is currently being read by some other transaction; or, conversely, a transaction wants to read a row that is currently being updated by some other transaction.

The following pages present examples of these problems where two concurrently executing transactions, Tran-A and Tran-B, conflict with each other. This section merely describes the problems without describing any method to prevent them. The following Section B.2 will describe how the system can prevent these problems.

Good News - Special "No Problem" Scenario: The above problems cannot occur within a read-only database environment where all concurrent transactions only retrieve data from tables. (I.e., All transactions only execute SELECT statements; no DML statements.) This is analogous to a classroom where all students simultaneously read the same white board or a projected image of the instructor's computer screen. A read-only database is frequently associated with data warehouse or data mining applications.

### B.1.a. Lost-Update Problem

Consider the following scenario where two concurrent transactions attempt to update the same row.

Time1 -	Tran-A reads a row with a column value of 100.
Time2 -	Tran-B reads the same row and finds the same value (100).
Time3 -	Tran-A changes its copy of the value by adding 50; then it writes the changed value (100+50=150) to the database, commits, and terminates.
Time4 -	Tran-B doubles its copy of the value, writes the updated value (2*100=200) to the database, commits, and terminates.

Figure 29A.1: Lost-Update Problem (Assume *no* isolation protection)

**Problem:** After Time4, the final value is 200. Observe that the change made at by Tran-A at Time3 is lost. The final value (200) is wrong according the serializable criterion. Consider the following serial execution sequences of these transactions.

- Assume Tran-A executes, commits, and terminates. Then Tran-B starts, commits, and terminates. In this case the final value is  $(100+50)*2 = 300$ .
- Alternatively, assume Tran-B executes, commits, and terminates. Then Tran-A starts, commits, and terminates. Here, the final value is  $(100*2)+50 = 250$ .

*Serializable requires that the final value must be 300 or 250 (not 200).* If the result should be 300, then you must explicitly schedule Tran-A such that it executes and commits before Tran-B starts; alternatively, if the result should be 250, then you must explicitly schedule Tran-B such that it executes and commits before Tran-A starts.

**Negative Domino Effects:** *The lost-update problem is a very serious problem because it writes garbage into the database.* This garbage could be read by subsequent transactions that may write additional garbage into the database. Section B.2.a will show how this lost-update problem can be prevented.

### B.1.b. Dirty-Read Problem

With the lost-update problem, both Tran-A and Tran-B wanted to update the same row. With the dirty-read problem (and the following two problems) only one transaction wants to update a row. In the following scenario, Tran-A updates the row, and then Tran-B reads the same row before Tran-A terminates. Then, a dirty-read problem occurs because Tran-A terminates by executing a ROLLBACK operation.

```
*** Assume Preserve 7 has a FEE value of 0.00

Time1 -   Tran-A executes UPDATE (without COMMIT):

          UPDATE PRESERVE
          SET FEE = 9.00
          WHERE PNO = 7;

Time2 -   Tran-B executes:

          SELECT PNO, FEE FROM PRESERVE WHERE PNO = 7;

          PNO   FEE
          ---   ---
           7   9.00

Time3 -   Tran-A executes:

          ROLLBACK

Time4 -   Tran-B continues to execute under the mistaken
          assumption that FEE is 9.00 (not 0.00).
```

Figure 29A.2: Dirty-Read Problem (Assume *no* isolation protection)

**Problem:** At Time2, Tran-B does a dirty-read by reading a modified but uncommitted FEE value (9.00). Then, at Time3, Tran-A returns the FEE value to 0.00. Thereafter, Tran-B continues to use the incorrect FEE value (9.00) which may produce an incorrect result.

*Observation:* In principle, the dirty-read problem is bad, but not as bad as the lost-update problem. A lost-update stores "garbage" in the database which could be read by subsequent transactions. With the dirty-read, only Tran-B suffers. However, more serious problems would occur if the program/procedure containing Trans-B subsequently writes data derived from the bad FEE (9.00) to the database.

### B.1.c. Non-Repeatable-Read Problem

Sometimes a transaction reads a row and then re-reads the same row. Assuming no isolation protection, the re-read operation could return the same row with different value(s). Consider the following scenario.

Time1 -	Tran-A executes:				
	SELECT PNO, FEE FROM PRESERVE WHERE PNO = 7;				
	<table><thead><tr><th><u>PNO</u></th><th><u>FEE</u></th></tr></thead><tbody><tr><td>7</td><td>0.00</td></tr></tbody></table>	<u>PNO</u>	<u>FEE</u>	7	0.00
<u>PNO</u>	<u>FEE</u>				
7	0.00				
Time2 -	Tran-B executes:				
	UPDATE PRESERVE				
	SET FEE = FEE + 5.00				
	WHERE PNO = 7;				
	COMMIT;				
Time3 -	Tran-A re-executes the same statement:				
	SELECT PNO, FEE FROM PRESERVE WHERE PNO = 7;				
	<table><thead><tr><th><u>PNO</u></th><th><u>FEE</u></th></tr></thead><tbody><tr><td>7</td><td>5.00</td></tr></tbody></table>	<u>PNO</u>	<u>FEE</u>	7	5.00
<u>PNO</u>	<u>FEE</u>				
7	5.00				

Figure 29A.3: Non-Repeatable-Read Problem (Assume *no* isolation protection)

**Problem:** At Time-1, Tran-A reads a row with a FEE value of 0.00. Then, at Time-3, it re-reads the same row which now has a FEE value of 5.00. This occurred because, at Time-2, Tran-B (unknown to Tran-A) modified the FEE value and committed it. This scenario is overly simplistic because Tran-A executes identical SELECT statements twice. A more realistic example will be presented later.

[Alternative Scenario: Tran-B executes a DELETE statement at Time-2, and Tran-A gets a "no hit" at Time-3.]

Observation: The non-repeatable-read problem is bad, but not as bad as the previous dirty-read problem because, in principle, Tran-A could detect and resolve this problem. However, writing extra code to detect and resolve this problem may not be reasonable.

### B.1.d. Phantom-Row Problem

A phantom-row problem occurs when a transaction retrieves a result table and subsequently retrieves the "same" result table which has a new row. This new row is called a "phantom-row." Consider the following scenario.

Time1 - Tran-A executes:									
SELECT PNO, PNAME, ACRES FROM PRESERVE WHERE ACRES > 40000									
<table><thead><tr><th>PNO</th><th>PNAME</th><th>ACRES</th></tr></thead><tbody><tr><td>7</td><td>MULESHOE RANCH</td><td>49120</td></tr></tbody></table>	PNO	PNAME	ACRES	7	MULESHOE RANCH	49120			
PNO	PNAME	ACRES							
7	MULESHOE RANCH	49120							
Time2 - Tran-B executes:									
INSERT INTO PRESERVE VALUES (99, 'HAPPY VALLY', 'AZ', 50000, 0.00); COMMIT;									
Time3 - Tran-A re-executes the same statement:									
SELECT PNO, PNAME, ACRES FROM PRESERVE WHERE ACRES > 40000									
<table><thead><tr><th>PNO</th><th>PNAME</th><th>ACRES</th></tr></thead><tbody><tr><td>7</td><td>MULESHOE RANCH</td><td>49120</td></tr><tr><td>99</td><td>HAPPY VALLY</td><td>50000</td></tr></tbody></table>	PNO	PNAME	ACRES	7	MULESHOE RANCH	49120	99	HAPPY VALLY	50000
PNO	PNAME	ACRES							
7	MULESHOE RANCH	49120							
99	HAPPY VALLY	50000							

Figure 29A.4: Phantom-Row Problem (Assume *no* isolation protection)

**Problem:** At Time-1 and Time-3, Trans-A's SELECT-statement specified the same condition (ACRES > 40000), but an additional "phantom-row" was returned at Time-3. This occurred because, at Time-2, Tran-B (unknown to Tran-A) inserted and committed a new row with an ACRES value (50000) which matched the ACRES > 40000 condition. (Again, this scenario is overly simplistic because Tran-A executes the same SELECT statement twice.)

Observation: As with the non-repeatable-read problem, the phantom-row problem is not as bad as the dirty-read problem because, in principle, Tran-A could detect and resolve this problem. However, as with the non-repeatable-read problem, detecting and resolving the phantom-row problem may not be reasonable.

## **B.2. System Prevents Data Isolation Problems**

Database systems can utilize *Locking* to prevent data isolation problems.

Preliminary Comment: A comprehensive discussion of locking is a rather complex topic that is beyond the scope of this book.

Another Preliminary Comment: There are other methods that prevent data isolation problems (e.g., Multi-Versioning) that are beyond the scope of this book. However, even if you know that your system is using some other method, many of the concepts presented in the remainder of this appendix are relevant.

**Locking:** Whenever your transaction wants to access a "chunk of data," the system (using its own internal bookkeeping scheme) will "lock" this data such that no other transaction can access it until your transaction terminates. After your transaction terminates, the system will unlock your transaction's data such that other transactions can access it.

Observe that, before the system can award your transaction a lock on some chunk of data, it must verify that another transaction has not already locked this data. If another transaction has already locked this data, your transaction may have to wait until this other transaction terminates.

**Types of Locks:** We discuss the two basic types of locks: Exclusive-Lock (X-Lock) and Share-Lock (S-Lock). As their names imply, if a transaction is awarded an X-Lock on a chunk of data, then no other transaction is allowed any kind of access to this data; and, if a transaction is awarded an S-Lock, then another transaction is allowed to read, but not modify, this data.

**Lock Assignment:** "Writer-transactions" are assigned X-Locks, and "Reader-transactions" are assigned S-Locks. Hence:

- Writers will block readers and other writers
- Readers will block writers
- Readers will not block other readers

The following examples will illustrate this behavior to demonstrate how locking can prevent data isolation problems.

### B.2.a. Prevent Lost-Update Problem

Basic Idea: Tran-A is awarded an X-lock which blocks Tran-B from obtaining an X-lock.

Revise Figure 29A.1:

Time1 - Tran-A asks the system to update a row. The system places an X-lock on the row and delivers it (with a column value of 100) to Tran-A. Then Tran-A continues.

Time2 - Tran-B asks the system to update the same row. (Hence Tran-B needs an X-lock on this row.) The system detects that the row is already locked with an X-lock and suspends Tran-B until Tran-A terminates. Tran-B waits.

Time3 - Tran-A updates the row by changing the column value from 100 to 150, writing the row to the database, and committing. This commit operation causes the system to remove Tran-A's X-lock.

Time4 - The system restarts Tran-B, places an X-lock on the row, and delivers it to Tran-B. Tran-B updates the recently modified column value from 150 to 300, writes the row to the database, and commits, and removes Tran-B's X-lock.

**Figure 29A.5: Locking Prevents Lost-Update Problem**

Locking prevented the lost-update problem. At Time-1, the system assigned an X-lock to the row on behalf of Tran-A. This prevented this system from giving any kind of lock to Tran-B at Time-2. Hence Tran-B was forced to wait.

Serializability was enforced. The results were the same as if Tran-A started and terminated before Tran-B started. (Note: This same sequence, Tran-A-before-Tran-B, occurs in the following three examples. Other circumstances, not described in this book, might produce a Tran-B-before-Tran-A sequence.)

Efficiency Cost: Because Tran-B is forced to wait, it suffers a longer response-time. This could extend the wait-time of other transactions that are also waiting for the same row.

### B.2.b. Prevent Dirty Read Problem

Basic Idea: Tran-A is awarded an X-lock which blocks Tran-B from obtaining an S-lock.

Revise Figure 29A.2:

Time1 - Tran-A asks the system to update a row. The system places an X-lock on the row and delivers it to Tran-A. Tran-A updates the row by changing a column value of 0.00 to 9.00. Tran-A continues to operate *without committing*.

Time2 - Tran-B asks the system to read the same row. (Hence Tran-B needs an S-lock on this row.) The system detects that Tran-A has an X-lock and suspends Tran-B until Tran-A terminates. Tran-B waits.

Time3 - Tran-A does a rollback operation returning its column value to 0.00. This rollback operation causes the system to unlock the row.

Time4 - The system restarts Tran-B, places an S-lock on the row for Tran-B, and delivers it (with the original column value of 0.00) to Tran-B. Tran-B continues using this value.

**Figure 29A.7: Locking Prevents Dirty-Read Problem**

Locking prevented the dirty-read problem. At Time-1, the system assigned an X-lock to the row on behalf of Tran-A. This prevented this system from giving any kind of lock to Tran-B at Time-2. Hence Tran-B was forced to wait.

Serializability was enforced. The results were the same as if Tran-A started and terminated before Tran-B started.

Efficiency Cost: Because Tran-B is forced to wait, it suffers a longer response-time. This could extend the wait-time of other transactions that are also waiting for the same row.



### B.2.c. Prevent Non-Repeatable-Read Problem

Basic Idea: Tran-A is awarded an S-lock which prevents Tran-B from obtaining an X-lock.

Revise Figure 29A.3:

- Time1 - Tran-A asks the system to read a row. The system places an S-lock the row and delivers it to Tran-A. Assume some column has a value of 0.00.
- Time2 - Tran-B asks the system to update the same row. (Hence Tran-B needs an X-lock on this row.) The system detects that Tran-A already has an S-lock and suspends Tran-B. Tran-B waits.
- Time3 - Tran-A *re-reads* the same row with the (unmodified) column value of 0.00. Tran-A continues until it does a commit or rollback operation which will cause the system to remove its lock.
- Time4 - The system restarts Tran-B.

Figure 29A.8: Locking Prevents Non-Repeatable-Read Problem

Locking prevented the non-repeatable-read problem. At Time-1, the system applied a S-lock on the row on behalf of Tran-A. This prevented the system from giving a X-lock to Tran-B at Time-2.

Serializability was enforced. The results were the same as if Tran-A started and terminated before Tran-B started.

Efficiency Cost: Again, because Tran-B is forced to wait, it suffers a longer response-time. This could extend the wait-time of other transactions that are also waiting for the same row.

Note that the system assigns an S-Lock at Time-1 which allows other read-only transactions to read the locked data. This is "friendlier" than the previous two locking scenarios where the system assigned an X-Lock at Time-1 which denied access to all other transactions. (A similar observation will apply to the following phantom-row scenario.)

#### B.2.d. Prevent Phantom-Row Problem

Basic Idea: Assign a *condition-lock* to Tran-A to prevent Tran-B from inserting rows that match the condition.

Revise Figure 29A.4:

Time1 - Tran-A asks the system to retrieve all PRESERVE rows where ACRES > 40000. The system applies a condition-lock on *all* rows that match this condition.

This kind of *condition-lock* (also called a "range-lock") locked *all* rows that matched the ACRES > 40000 condition. A condition-lock is different than a lock which locks rows that are stored in the database. A condition-lock locks existing rows and also (somehow) locks any "future inserted rows" that match the condition.

Time2 - Tran-B asks permission to insert a new row with an ACRES value of 50000 (which matches the ACRES > 4000 condition). The condition-lock implies this new row could become a phantom-row. Hence, the system suspends Tran-B until the Tran-A terminates. Tran-B waits.

Time3 - Tran-A *re-reads* the PRESERVE table with the same condition and returns the same rows (without any phantom-rows). Tran-A subsequently does a commit or rollback operation, causing the system to remove the condition-lock.

Time4 - The system restarts Tran-B.

**Figure 29A.9: Locking Prevents Phantom-Row Problem**

At Time-1, the system assigned a condition-lock on behalf of Tran-A. This lock caused the system to suspend Tran-B because it wants to insert a row that matches the condition.

Again, serializability was enforced (as if Tran-A started and terminated before Tran-B started).

Again, because Tran-B is forced to wait, it suffers a longer response-time, and this may extend the wait-time of other transactions that are also waiting for the same row.

### **B.3. Adjusting Isolation Levels**

Although locking prevents transaction isolation errors, it has a negative impact on response-time by forcing some transactions to wait for other transactions to terminate. In some circumstances, you can improve efficiency by adjusting your transaction's isolation level. In these circumstances, you effectively tell the system:

"I want to reduce the response-time for my transaction. So, don't bother protecting my transaction against some isolation problems. I know what I am doing. I know that a specific isolation problem cannot occur. Or, if such a problem can occur, I can resolve it, or I will simply tolerate its negative consequences."

Most systems support SQL's **SET TRANSACTION** statement which can be used to adjust a transaction's "isolation level." (See the following Figure 29A.10.) Four isolation levels can be specified: **SERIALIZABLE**, **REPEATABLE READ**, **READ COMMITTED**, and **READ UNCOMMITTED**. This figure describes the meaning of each isolation level by designating, for each isolation level, which isolation problems will be prevented and which problems will be allowed to occur.

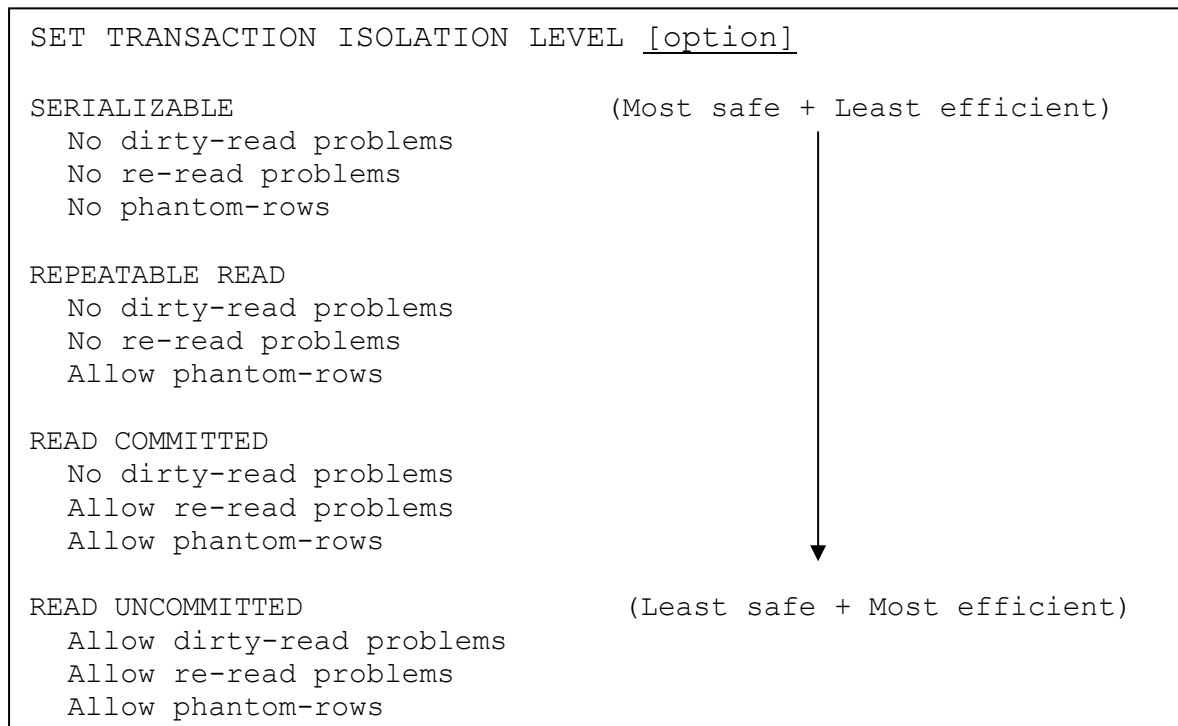


Figure 29A.10: Adjusting Isolation Level Protection

Your DBA will set a *default isolation level that applies to all transactions*. Executing a SET TRANSACTION ISOLATION LEVEL statement allows you to change the isolation level for your transaction *only*.

Also note that the SET TRANSACTION ISOLATION LEVEL does not specify an option like "Allow Lost-Updates." The system will always prevent lost-updates.

**Variation among different systems:** *Different systems support different variations of the SET TRANSACTION statement, and these systems offer different options for setting the isolation level. We briefly comment on four database systems.*

**MYSQL:** MYSQL supports the SET TRANSACTION ISOLATION LEVEL statement with the four options described in Figure 29A.10.

**SQL Server:** SQL supports the SET TRANSACTION ISOLATION LEVEL statement which includes the four options described in Figure 29A.10, plus another option.

**ORACLE:** ORACLE supports a SET TRANSACTION ISOLATION LEVEL statement that can specify two of the four options described in in Figure 29A.10.

**DB2 (Windows):** DB2 uses the term "unit of work" (UOW) instead of "transaction." DB2 does not support the SET TRANSACTION ISOLATION LEVEL statement. Instead, DB2 supports a similar SET CURRENT ISOLATION TO statement which assigns a value to a system register. The options (UR, CS, RS, and RR) are similar to but not identical to those described in Figure 29A.10.

Given the variation across different systems, you should consider the following discussion to be a *conceptual* introduction to setting isolation levels. You *must* consult your reference manual to learn details about setting isolation levels on your system.

## Efficiency Considerations

**Terminology - On-line versus Off-line:** - We use the term "on-line" processing to imply the concurrent execution of SQL transactions that read *and modify* a shared collection of tables. The previously described isolation level problems apply within an on-line environment.

The term "off-line" processing refers to a processing environment that executes a "batch" of non-conflicting update programs. These programs "run a midnight" so they do not compete with on-line programs that execute during the business day. Isolation level problems do not occur in an off-line environment.

[Today, sophisticated 24x7 "non-stop" database systems support on-line transactions that automatically address isolation level problems. Most applications do not run on such sophisticated systems.]

**Isolation Levels:** Below we examine each isolation level by working backwards, from the weakest level of protection to the strongest.

### READ UNCOMMITTED ISOLATION LEVEL

- Allow dirty-reads
- Allow non-repeatable-reads
- Allow phantom-rows

This is the weakest (least safe) isolation level and should only be specified in special case circumstances. Good news! The following two special case circumstances are very common.

**1. Read-Only Databases** - Sometimes, *all* application tables are read-only. A common example is a data warehouse application. Also, the DBA may create a read-only database for testing purposes.

**2. Read-Only Tables:** Your on-line system might execute many concurrent transactions that modify tables. But it may be your good fortune that your transaction happens to read a read-only table. Here you know that no other concurrent transaction can possibly change this table. Therefore, *your transaction cannot encounter any isolation problems.*

**Example-1: "Code" Tables** - A code table is usually a small table that is rarely modified. The STATE table in our MTPC database is a code table where the STCODE column contains the codes. In real-world applications, STATE would be populated with fifty rows, one for each state in the USA. Political circumstances imply that it will be a very long time before any INSERT and DELETE operations will be applied to this table. Regarding UPDATE operations, it will be a long time before any state changes its STCODE and STNAME values. But there will be occasional changes to the POPULATION column. Therefore, the DBA might prohibit all *on-line DML* operations against the STATE table and move infrequent updates of the POPULATION column off-line.

**Example-2: An "Almost-Read-Only" Table** - Within our MTPC database, the REGION table could be setup as read-only within an on-line environment. Most users would not consider REGION to be a code table. However, changes to this table would be infrequent, maybe a few times a year. The DBA could prohibit all *on-line DML* operations against the REGION table, and move infrequent DML operations off-line.

**Conclusion:** Always ask: "Do my SELECT statements only access read-only tables?" If yes, consider setting your transaction's isolation level to READ UNCOMMITTED.

[Note: Practically all read-only tables are "effectively" read-only. It is hard to image any table that is absolutely read-only. Perhaps a chemistry application could include an absolutely read-only table that represents the unchanging Periodic Table. However, it is impossible to imagine an absolutely read-only table within a business application. Therefore, in this book, read-only always implies effectively read-only where DML operations are pushed off-line.]

## READ COMMITTED ISOLATION LEVEL

- No dirty-reads
- Allow non-repeatable-reads
- Allow phantom-rows

The option is the second weakest isolation level. You should consider specifying READ COMMITTED *if your transaction never re-reads the same data*. Therefore, your transaction would not need protection from non-repeatable-reads and phantom-rows.

**Example-1: Never Re-Read a Row** - A transaction has just one SELECT statement. Hence, no re-reads. (Because this is a very common circumstance, the DBA might set READ COMMITTED as the default isolation level.)

### **Example-2: STATE Table (As an "Almost-Read-Only" Table)**

- On-line INSERTs are prohibited (no new states), and
- On-line DELETEs are prohibited (no leaving USA), and
- On-line UPDATEs are limited to the POPULATION column

Here:

- READ COMMITTED isolation prevents dirty-reads.
- Prohibiting on-line INSERTs prevent phantom-rows.
- Prohibiting on-line DELETE prevents non-repeatable-reads caused by this operation.
- And, *if your transaction's SELECT statement(s) only re-read non-updateable columns (STCODE and STNAME)*, you cannot encounter non-repeatable-reads caused by another transaction's UPDATE operation.

## REPEATABLE READ ISOLATION LEVEL

- No dirty-reads
- No non-repeatable-reads
- Allow phantom-rows

This option is the second strongest isolation level. We describe two examples where REPEATABLE READ could be helpful.

**Example-1: "Rarely-Inserted" Tables** - These are tables where:

- On-line INSERTs are prohibited, and
- On-line DELETES are allowed, and
- On-line UPDATES are allowed.

Hence:

- REPEATABLE READ isolation prevents dirty-reads.
- REPEATABLE READ isolation prevents non-repeatable-reads
- You only have to worry about phantom-rows. Fortunately, on-line INSERT operations are executed off-line.

Consider the SUPPLIER table in the following context. Some organizations impose a rigorous security process on new suppliers. This reduces the number of new suppliers and limits the number of INSERT operations into the SUPPLIER table. Because INSERT operations are rare, they can be executed off-line. DELETE and UPDATE operations are protected by the REPEATABLE READ isolation level.

**Example-2: "Append-Only" Tables** - These are tables where:

- On-line DELETES are allowed, and
- On-line UPDATES are allowed.
- On-line INSERTs are restricted to inserts at the "end of the table."

Hence:

- REPEATABLE READ isolation prevents dirty-reads.
- REPEATABLE READ isolation prevents non-repeatable-reads
- You avoid phantom-rows by not selecting rows from "the end of the table."

An append-only table is logically (and maybe physically) sorted by some column, usually a date-time column. All on-line INSERT operations must (somehow) insert rows where the sort-column value is always greater than the previously inserted row. (The "new row" is inserted after the "last row.")



This scenario could apply to the PUR\_ORDER table. Here, transactions that insert rows would guarantee that the PODATE value for each newly inserted row is always greater than or equal to the PODATE value for the previously inserted row. [Note: Real-world purchase-order tables would probably store date-time values. The same append-only concepts apply.]

When coding SELECT statements, you must (1) know about this append-only behavior, and (2) code WHERE-clauses that reject the "last row(s)" that could set the stage for phantom-rows. For example, assume today's PODATE value is 204. Then your SELECT statements would contain an WHERE-clause that looks like:

```
SELECT * FROM PUR_ORDER
WHERE PSTATUS = 'P'
AND    PODATE < 204      ←
```

An alternative approach allows the system to automatically prevent phantom-rows. The DBA could create a view (discussed in Chapter 28) called PUR\_ORDERV which specifies a condition like **PODATE < TODAY**. (TODAY is a keyword containing today's date. This keyword varies across different database systems.)

The CREATE VIEW statement would look like:

```
CREATE VIEW PUR_ORDERV AS
  SELECT * FROM PUR_ORDER
  WHERE PODATE < TODAY
```

Then, your transaction could specify:

```
SELECT * FROM PUR_ORDERV
WHERE PSTATUS = 'P'
```

The system substitutes today's date (204) for TODAY. Then it generates and executes the following desired statement.

```
SELECT * FROM PUR_ORDER
WHERE PSTATUS = 'P'
AND    PODATE < 204
```

## SERIALIZABLE ISOLATION LEVEL

- No dirty-reads
- No non-repeatable-reads
- No phantom-rows

SERIALIZABLE is the safest isolation level because it prohibits all transaction isolation problems. However, this option could generate long response-times for some transactions.

In an ideal-world, SERIALIZABLE would be the default isolation level; and, many DBAs will designate SERIALIZABLE as the default isolation level. However, if slow response-time becomes a problem, some DBA's might choose another isolation level as the default. As previously mentioned, some DBAs specify READ COMMITTED as the default isolation level. Therefore, we emphasize that: *SERIALIZABLE might not be your default isolation-level.*

**Conclusion: A Positive Domino Effect:** Earlier we stated that:

*"I want to reduce the response-time for my transaction. So, don't bother protecting my transaction against one or more specific isolation problems."*

We really have a "Win-Win" situation. Notice that, when I reduce my transaction's execution time, I "get in and get out." When I "get out" earlier, my locks are released earlier, thereby allowing other transactions quicker access to the data that I had locked. *Hence, by reducing my response-time, I indirectly reduce the response time of other concurrent transactions.*

#### **B.4. More About Locking**

The plot thickens.

**Size of Lock:** In previous examples we indicated that the system locked one row or a "chunk" of data. More accurately, real-world systems can lock a single row, a physical block (page) consisting of multiple rows, a partition (a collection of physically contiguous blocks), a complete table, or a "tablespace" which consists of multiple blocks containing all rows from one or more tables. Choosing the optimal lock size is not a simple task.

Various database systems offer different ways to set the size of a lock. For example, in DB2, the DBA can specify a LOCKSIZE parameter in the CREATE TABLESPACE and ALTER TABLESPACE statements that set the default lock size for all tables in a tablespace. And, in DB2, applications developers (with appropriate privileges) can execute a LOCK TABLE statement to lock a table (e.g., LOCK TABLE JUNK IN SHARE MODE, LOCK TABLE JUNK IN EXCLUSIVE MODE).

**Efficiency Tradeoffs:** A larger lock size implies fewer locks which imply more efficient internal bookkeeping. Assume your transaction will retrieve 1000 rows from a JUNK table. A worst-case scenario with row-level locking could involve 1000 executions of: [ask-for-lock, wait-to-get-lock, get-and-process-row, release-lock]. Alternatively, with table-level locking, your transaction does a one-time [ask-for-lock, wait-to-get-lock, get-and-process-1000-rows, release-lock]. Locking an entire table sounds great because your transaction "gets in and gets out.". BUT, before the system can award you a table-level lock, your [wait-to-get-lock] request could take a long time because you have to wait for *all* X-locks on JUNK to be released. Furthermore, after your transaction gets an S-Lock on the JUNK table, you force *all* other transactions that want an X-lock on any row in this table to wait.

Hints: Some systems allow an application developer to specify hints that set lock size and influence locking behavior. For example, SQL Server provides PAGLOCK, TABLOCK, NOLOCK, and other locking hints.

**Inconsistent-Analysis Problem:** Consider the following scenario. Assume row-level locking. Your SELECT statement executes SUM(AMT) over five rows with AMT values of 10, 5, 15, 30, 20. The correct result should be 80. After the system summarizes the third row, it has an intermediate running total of 30. Then, before your transaction can lock the fourth row, another transaction changes its AMT value from 30 to 60 and *commits*. Then, your transaction reads the fourth row AMT (value of 60), and then the fifth row AMT (value of 20), and produces an incorrect total value of 110. A table-level lock could have prevented this problem.

Author Comment: Consider the inconsistent-analysis problem in terms of setting isolation levels. (Note: This problem is not directly protected by the SET TRANSACTION ISOLATION LEVEL statement.) Assume you are not allowed to execute a LOCK TABLE statement. What isolation level do you choose? Candidly, I don't know, and the answer could vary across database systems. I have read that READ COMMITTED, which only protects you from dirty-reads, will not work. This is reasonable because the inconsistent-analysis problem is not caused by a dirty-read. And, with row-level locks, READ COMMITTED only cares about the integrity of the current row. So, we jump to the next level of protection, REPEATABLE READ. I have read that this isolation level will prevent inconsistent-analysis problem. But I am uncertain. What about phantom-rows? So, to be safe, we jump to SERIALIZABLE. If this is unacceptable, consult your reference manual and/or ask your DBA. Good luck.

**Types Locks:** For tutorial purposes, we have only described the major lock types, the S-lock and the X-lock. This is good enough for presenting a conceptual overview. However, commercial systems are not limited to these two types of locks. For example, DB2 mainframe provides more than 10 different lock types.

**Releasing Locks:** After awarding a lock, the system must decide when to release the lock. General Objective: Release your locks early so they can be given to other transactions. But, do not release locks too early, which may allow isolation problems to occur. Releasing a lock can be done at different times.

- After a transaction terminates.  
[Locks held for the longest time.]
- After a statement terminates. For example, assume a transaction has two SELECT statements which reference different tables. In some circumstances, locks assigned for the first statement can be released after this statement completes, before the second statement starts.
- After a row has been retrieved.  
[Locks held for the shortest time.]  
Assume a transaction has a SELECT statement that executes SUM(AMT) over multiple rows. With row-level locking, the lock on each row can be released after each row has been accessed and summarized. Notice that, with the previous inconsistent analysis problem, the problem occurred because an AMT value was changed by another transaction *before* your transaction accessed it. Here, each row-level lock is released *after* the transaction has accessed a row. This is a good idea, *unless* your transaction will re-read some of the same rows.

The system usually decides when to release locks. However, some systems (e.g., SQL Server) provide an UNLOCK TABLE statement.

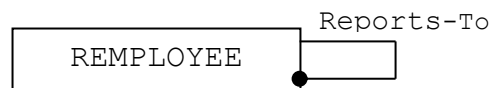
## Appendix Summary

Again, we emphasize that our discussion of locking and isolation levels was conceptual. We only examined the tip of the iceberg. Applications developers are strongly encouraged to consult their reference manuals and the multitude of web sites which present system specific details.

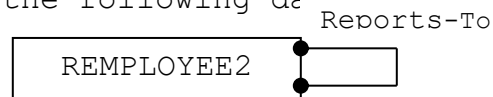
## Recursive Queries

This (rather long) chapter introduces a special kind of query called a "recursive query." This type of query references a "recursive table" that is *related to itself* via a "recursive relationship." Unlike other relationships previously described in this book, a recursive relationship *relates objects of the same type*. Casually speaking, recursion involves a form of "self-reference."

A common example of recursion is a business organization where every employee (except the "big boss") directly reports to exactly one supervisor who is also an employee; and, a supervisor may directly supervise one or more employees. This example of a one-to-many recursive relationship is illustrated by the following data model. -



A many-to-many relationship can also be recursive. For example, some organizations allow "matrix management" where an employee may directly supervise multiple employees, and an employee may have multiple direct supervisors. A recursive many-to-many relationship is illustrated by the following data model.



Section A presents sample queries against a table called `EMPLOYEE` that is related to itself via a recursive one-to-many relationship. Section B presents sample queries against a table called `EMPLOYEE2` that is related to itself via a recursive many-to-many relationship. Section C concludes with some special case recursive queries.

## A. Recursive One-to-Many Relationships

This section describes a recursive one-to-many relationship where each employee (except the "big boss") reports to exactly one supervisor; and each employee who is a supervisor may supervise multiple employees. The following Figure 30.1a illustrates a data model with a recursive relationship (Reports\_To). Within the corresponding CREATE TABLE statement, this relationship is implemented by a FOREIGN KEY clause designating the SENO column (the supervisor's ENO value) as a *foreign-key which references EMPLOYEE, the same table that is being created.*

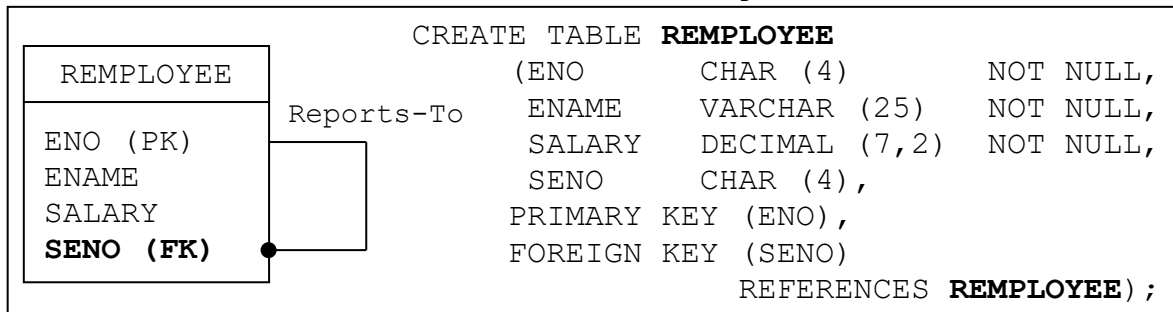


Figure 30.1a: Recursive One-to-Many Relationship

Sample data for the EMPLOYEE table are shown below. The null SENO value in the first row implies that Employee 1000 is the big boss who does not have a supervisor.

Figure 30.1b:  
EMPLOYEE Table

ENO	ENAME	SALARY	SENO
1000	MOE	2000.00	-
2000	JANET	2000.00	1000
3000	LARRY	3000.00	1000
4000	JULIE	500.00	2000
4500	JOHNNY	2000.00	4000
4600	ELEANOR	3000.00	4000
4700	ANDY	2000.00	4600
4800	MATT	3000.00	4600
5000	JESSIE	400.00	2000
5500	HANNAH	4000.00	5000
6000	FRANK	9000.00	2000
6500	CURLY	8000.00	3000
7500	SHEMP	9000.00	6500
8000	JOE	8000.00	1000
8500	GEORGE	7000.00	8000
8600	DICK	6000.00	8500
8700	HANK	6000.00	8500

## Recursive One-to-Many Relationship → Tree Diagram

Rows within the recursive REEMPLOYEE table can be represented by the following tree diagram.

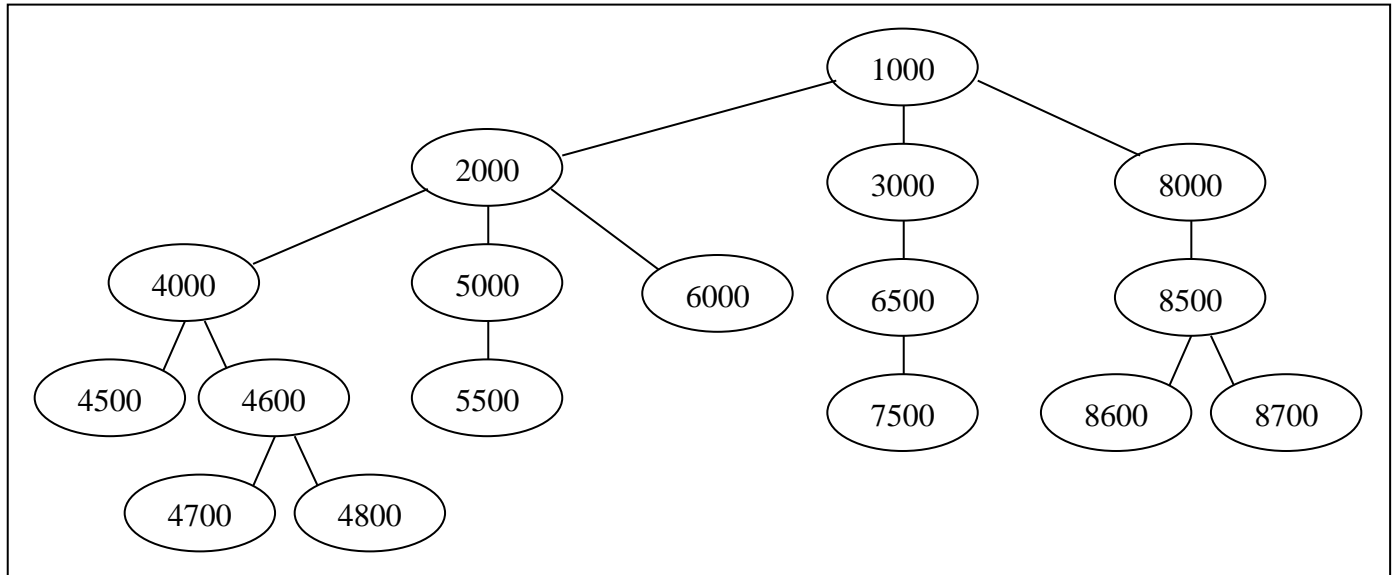


Figure 30.1c: Tree Diagram Represents REEMPLOYEE Table

Each REEMPLOYEE row is represented by a circular node containing its primary-key (ENO). For example, Node-2000 represents the row with the ENO value of 2000. Each row's SENO value is represented by an upward line to the node representing the employee's direct supervisor. For example, the line above Node-2000 indicates that Employee 2000 reports to Employee 1000.

**Terminology:** Conventional genealogical terms (*parent*, *child*, *ancestor*, and *descendant*) describe relationships between nodes. Examples: Node-2000 is the parent of Node-4000, Node-5000, and Node-6000. Node-5000 is a child of Node-2000. Node-2000 is an ancestor of all nodes below it; and, Node-4600 is a descendant of Node-4000, Node-2000, and Node-1000. The *root* node is the node at the top of the tree (Node-1000) and is the only node without a parent. The corresponding row is the only row with a null SENO value. Nodes at the bottom of the tree (4500, 4700, 4800, 5500, 6000, 7500, 8600, and 8700) which do not have any children are called *leaf* nodes.

Finally, observe that there is only one path from the root node to any other node. (This observation will not apply to Network Diagrams to be introduced in Section B.)



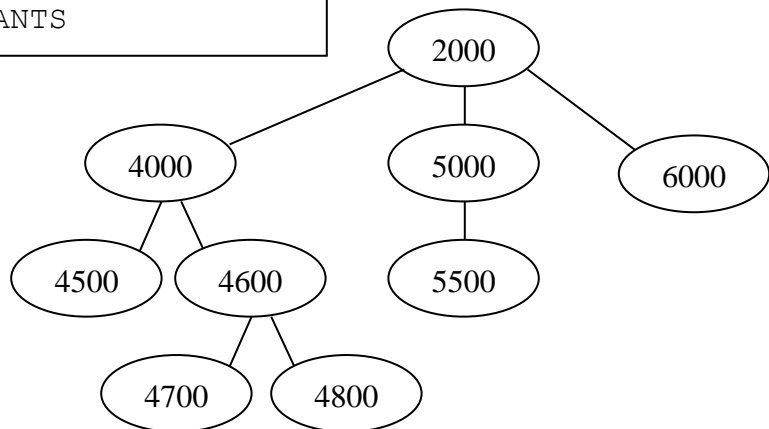
## Recursive Query: Traverse Down a Tree

A recursive query asks the system to traverse up or down a tree. The following sample query asks the system to traverse down a tree. Because recursive SQL code is not self-evident, we begin by introducing a relatively simple example before presenting a comprehensive explanation of syntax and logic. From a graphical perspective, the following sample query asks the system to start at Node-2000 and traverse down the tree.

**Sample Query 30.1:** Reference the REMPLOYEE table. Display ENO, ENAME, and SENO values for Employee 2000 and all employees who directly or indirectly work for this employee. (I.e., Display data about Employee 2000 and all her descendants.) Also draw a sub-tree that represents the result table.

```
WITH DESCENDANTS (ENO, ENAME, SENO)
AS
(SELECT   ENO, ENAME, SENO
 FROM     REMPLOYEE
 WHERE    ENO = '2000'
  UNION ALL
 SELECT   R.ENO, R.ENAME, R.SENO
 FROM     DESCENDANTS D, REMPLOYEE R
 WHERE    D.ENO = R.SENO
 )
SELECT * FROM DESCENDANTS
```

ENO	ENAME	SENO
2000	JANET	1000
4000	JULIE	2000
5000	JESSIE	2000
6000	FRANK	2000
4500	JOHNNY	4000
4600	ELEANOR	4000
5500	HANNAH	5000
4700	ANDY	4600
4800	MATT	4600



**Important Observation:** No ORDER BY clause is specified. Hence, again, you cannot make any assumptions about row sequence in the result table. However, the above result table is *incidentally* displayed in a "hierarchical sequence." (This may not occur on your system.) A following section will say more about Hierarchical Sequences.

## Dissecting a Recursive Query: Syntax

The logic of a recursive query can be a little tricky, especially for rookie users. However, for the moment, disregard logic and focus on syntax and related terminology.

**SELECT Keyword:** Skeleton-code for Sample Query 30.1 shows that the keyword SELECT is specified three times.

```
WITH DESCENDANTS (. . .)
AS  (SELECT ...
     UNION ALL
     SELECT ...
     ...
     ...
  )
SELECT ...
```

The first two SELECTs are Sub-SELECTs specified within a WITH-clause that defines a Common Table Expression (CTE) called DESCENDANTS. (You might want to review Chapter 27 which introduced CTEs.) These Sub-SELECTs will select data from the REMPLOYEE table to populate DESCENDANTS.

1<sup>st</sup> Sub-SELECT: The **Initialization Sub-SELECT**.

```
SELECT ENO, ENAME, SENO
FROM   REMPLOYEE
WHERE  ENO = '2000'
```

The Initialization Sub-SELECT is usually quite simple. Here, data from the row describing Employee 2000 is placed into the DESCENDANTS table.

2<sup>nd</sup> Sub-SELECT: The **Recursive Sub-SELECT**.

```
SELECT R.ENO, R.ENAME, R.SENO
FROM   DESCENDANTS D, REMPLOYEE R
WHERE  D.ENO = R.SENO
```

The Recursive Sub-SELECT is not so simple and will be explained on the following pages. This explanation will describe the process whereby data about all descendants of Employee 2000 are placed into the DESCENDANTS table.

3<sup>rd</sup> SELECT: The **Main SELECT**.

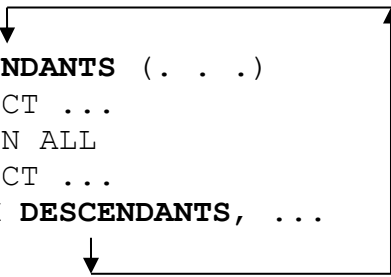
```
SELECT * FROM DESCENDANTS
```

All rows in the DECSENDANTS table are selected for display.

As this example illustrates, the Initialization Sub-SELECT and the Main SELECT are usually quite simple. (You can understand this code after reading Chapter 1.) However, the Recursive Sub-SELECT is another story. This recursive Sub-SELECT “does all the hard work” that requires a rather long explanation. We begin with a critical observation about the definition of DESCENDANTS.

**Critical Observation:** Notice the “self-reference” within this WITH-clause. This WITH-clause defines a CTE called DESCENDANTS; and, most importantly, note that *this definition of DESCENDANTS references DESCENDANTS*.

```
WITH DESCENDANTS (. . .)
AS (SELECT ...
    UNION ALL
    SELECT ...
    FROM DESCENDANTS, ...
    ...
)
SELECT ...
```



This self-reference implies that DESCENDANTS is a recursive-CTE, not a conventional (non-recursive) CTE as described in Chapter 27.

Note: Some systems (e.g., SQL Server) *require* specifying the RECURSIVE keyword to code a recursive CTE as illustrated below:

```
WITH RECURSIVE DESCENDANTS (. . .)
AS (SELECT ...
    UNION ALL
    SELECT ...
    FROM DESCENDANTS, ...
    ...
)
SELECT ...
```

## Dissecting a Recursive Query: Logic

The following pages present a step-by-step tutorial walk-through of the logic for this sample query.

[Note: This description of recursive logic is a logical description. The system may utilize internal efficiency techniques not described here.]

### 1. Initialization Sub-SELECT.

```
SELECT ENO, ENAME, SENO
FROM   REMPLOYEE
WHERE  ENO ='2000'
```

This Sub-SELECT places the following row into DESCENDANTS.

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
2000	JANET	1000

This row corresponds to the first-level node in the sub-tree diagram that represents the DESCENDANTS table. (Note that this row corresponds to a second-level node in the tree diagram illustrated in Figure 30.1c.)

### 2. Recursive Sub-SELECT.

```
SELECT R.ENO, R.ENAME, R.SENO
FROM   DESCENDANTS D, REMPLOYEE R
WHERE  D.ENO = R.SENO
```

This Sub-SELECT will be executed multiple times. Each execution will retrieve the children of parent-rows that were retrieved during the previous execution. The first execution will place second-level rows (children of Employee 2000) into DESCENDANTS; the second execution will place third-level rows (grandchildren of Employee 2000) into DESCENDANTS. Etc. Execution stops after all rows corresponding to descendants of Node-2000 have been retrieved.

The following pages examine this recursive Sub-SELECT in greater detail.

1<sup>st</sup> Execution: The recursive Sub-SELECT returns the second-level rows. Because the DESCENDANTS table currently has one row with an ENO value of 2000, the join-operation matches this row with the three REMPLOYEE rows with a SENO value of 2000 and returns the following three rows corresponding to the children of Employee 2000.

4000	JULIE	2000
5000	JESSIE	2000
6000	FRANK	2000

These "new rows" (designated by ←<sub>new</sub>) are placed into the DESCENDANTS table which now looks like:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>	
2000	JANET	1000	
4000	JULIE	2000	← <sub>new</sub>
5000	JESSIE	2000	← <sub>new</sub>
6000	FRANK	2000	← <sub>new</sub>

2<sup>nd</sup> Execution: The recursive Sub-SELECT is executed *again* to return the next (third-level) rows.

*\*Important Observation: Only the new rows in DESCENDANTS are joined with REMPLOYEE. Hence, the second execution of this Sub-SELECT returns the following result.*

4500	JOHNNY	4000
4600	ELEANOR	4000
5500	HANNAH	5000

These rows are placed into DESCENDANTS, and they become the new rows. DESCENDANTS now looks like:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>	
2000	JANET	1000	
4000	JULIE	2000	
5000	JESSIE	2000	
6000	FRANK	2000	
4500	JOHNNY	4000	← <sub>new</sub>
4600	ELEANOR	4000	← <sub>new</sub>
5500	HANNAH	5000	← <sub>new</sub>

3<sup>rd</sup> Execution: The recursive Sub-SELECT is executed again to return the next (fourth-level) rows.

4700	ANDY	4600
4800	MATT	4600

These rows are placed into DESCENDANTS, and they become the new rows. DESCENDANTS now looks like:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
2000	JANET	1000
4000	JULIE	2000
5000	JESSIE	2000
6000	FRANK	2000
4500	JOHNNY	4000
4600	ELEANOR	4000
5500	HANNAH	5000
4700	ANDY	4600 ←new
4800	MATT	4600 ←new

4<sup>th</sup> Execution: The recursive Sub-SELECT is executed again. This time, joining the new rows in DESCENDANTS with REMPLOYEE produces a "no hit" because none of the new (fourth-level) rows have children. Graphically speaking, DESCENDANTS already contains all leaf-nodes under Node-2000. This no-hit event terminates execution of the recursive Sub-SELECT. DESCENDANTS remains unchanged:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
2000	JANET	1000
4000	JULIE	2000
5000	JESSIE	2000
6000	FRANK	2000
4500	JOHNNY	4000
4600	ELEANOR	4000
5500	HANNAH	5000
4700	ANDY	4600
4800	MATT	4600

**3. Main SELECT.** Finally, the Main SELECT is executed:

```
SELECT * FROM DESCENDANTS
```

This statement displays the final result which in this case constitutes all rows in the DESCENDANTS table.

## Important “Getting Started” Exercises

Although we have only presented one recursive sample query, you should be able to utilize this example to code solutions for the following exercises. Do not specify an ORDER BY clause for any of these exercises. (Optionally, you are invited to detect a special kind of row sequence in the result tables. This sequence will be discussed later in this section.)

30A1. Reference the REMPLOYEE table. Display ENO, ENAME, and SENO values for Employee 8000 and all employees who directly or indirectly work for this employee. I.e., Display data about Employee 8000 and all his descendants. The result table should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
8000	JOE	1000
8500	GEORGE	8000
8600	DICK	8500
8700	HANK	8500

Hint: This exercise only requires one trivial modification to the solution for Sample Query 30.1.

30A2. Enhance the previous Exercise 30A1 to also display SALARY values. The result table should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>SENO</u>
8000	JOE	8000.00	1000
8500	GEORGE	7000.00	8000
8600	DICK	6000.00	8500
8700	HANK	6000.00	8500

30B1. Reference the REMPLOYEE table. Traverse its tree from top to bottom. Start with the row for Employee 1000 (root node). Display all data about this employee and all employees who directly or indirectly work for this employee. (I.e., Display the entire tree.)

30B2. The solution for the previous Exercise 30B1 assumes you know that the root node has an ENO value of 1000. Assume you do not have this knowledge. Code an alternative solution that satisfies the same query objective. Start at the root node (without knowing its ENO value) and display all information about all its descendants.

30C. Consider the recursive RDEMO1 table shown below in Figure 30.2. All columns contain integer values. PKEY is the primary-key. FKEY is a foreign-key that references PKEY. (The rows happen to be displayed in PKEY sequence.)

PKEY	CODE	FKEY
10	0	-
15	1000	25
20	0	10
25	0	20
30	0	15
35	0	30
40	0	25
50	0	40

Figure 30.2: RDEMO1 Table

Draw a tree diagram for the above RDEMO1 table.

Display the PKEY, CODE, and FKEY values for the rows with a PKEY value of 25 and all its descendant rows. The result should contain the following rows (without regard to sequence).

PKEY	CODE	FKEY
25	0	20
15	1000	25
40	0	25
30	0	15
50	0	40
35	0	30

Hints: Code a CTE called DESCENDANTS.

- The initialization Sub-SELECT should retrieve the row with a PKEY value of 25.
- The recursive Sub-SELECT should join DESCENDANTS with RDEMO1 by matching DESCENDANTS's primary-key with RDEMO1's foreign-key.

```
DESCENDANTS.PKEY = RDEMO1.FKEY
```

- The Main SELECT should display all information in DESCENDANTS.



## General Syntax and Logic

The following Figure 30.3 outlines a code-pattern for a recursive query that traverses down the REMPLOYEE tree to retrieve descendant rows.

```
WITH DESCENDANTS (ENO, . . . , SENO)
AS
(
  SELECT ENO, . . . , SENO
  FROM   REMPLOYEE
  [WHERE condition-1]
  UNION ALL
  SELECT R.ENO, . . . , R.SENO
  FROM   DESCENDANTS D, REMPLOYEE R
  WHERE  D.ENO = R.SENO
  [AND  condition-2]
)
SELECT . . .
FROM   DESCENDANTS
[WHERE condition-3]
[ORDER BY . . .]
```

Figure 30.3: Recursive Query “Down a Tree”

The WITH-clause defines and populates a CTE called DESCENDANTS. DESCENDANTS is not a reserved word. However, subsequent sample queries that traverse down a tree will specify DESCENDANTS as the name of the recursive CTE. (We will specify ANCESTORS for upward tree traversals.)

The Initialization Sub-SELECT is a conventional (non-recursive) Sub-SELECT; the Recursive Sub-SELECT implements the recursive logic which does “most of the hard work” to place descendant rows into the DESCENDANTS table; and, the Main-SELECT is a conventional (non-recursive) SELECT-statement that displays some or all DESCENDANTS rows.

### Initialization Sub-SELECT

```
SELECT ENO, . . . , SENNO  
FROM   REMPLOYEE  
[WHERE condition-1]
```

This Sub-SELECT initializes the DESCENDANTS table. It usually specifies a WHERE-clause that selects one or more rows from the recursive table (REMPLOYEE). Graphically speaking, this indicates that you want to start at one or more nodes and traverse the tree to retrieve the descendants (or ancestors) of each node. In Sample Query 30.1 the initialization Sub-SELECT returned just one row. Sample Query 30.2a will demonstrate an example where this Sub-SELECT returns multiple rows.

### Recursive Sub-SELECT

```
SELECT R.ENO, . . . , R.SENNO  
FROM   DESCENDANTS D, REMPLOYEE R  
WHERE  D.ENO = R.SENNO  
[AND  condition-2]
```

This Sub-SELECT specifies the recursive logic. Two operations require your attention.

- Join Operation:

**Important:** The join-condition dictates the direction of tree traversal, down the tree from parent to child, or up the tree from child to parent. The above join-condition (D.ENO = R.SENNO) indicates downward traversal. We will say more about this issue on the following pages.

- Restriction: (AND condition-2)

You can specify a restriction operation within a recursive Sub-SELECT. Note that a "no hit" terminates the search for further descendants/ancestors. Be careful when coding this condition because a logical error will produce a "too early" or "too late" termination of the tree traversal. Sample Query 30.2b will offer a detail discussion of this logic.

## Main-SELECT

```
SELECT      . . .
FROM        DESCENDANTS
[WHERE      condition-3]
[ORDER BY  . . .]
```

This SELECT produces the final result by selecting some or all rows from DESCENDANTS. Two operations require your attention.

- Restriction (WHERE condition-3)

This is usually a relatively simple WHERE-clause. Sample Query 30.2c will specify a WHERE-clause that selects a subset of DESCENDANTS's rows.

- ORDER BY clause

Consistent with prior recommendations, we usually encourage you to specify ORDER BY clauses in SELECT-statements. However, before we present sample queries that specify an ORDER BY clause in the Main SELECT, we must address the notion of a "Hierarchical Sequence" on the following page. For the moment, we simply note (without explanation) that most systems do not allow you to specify an ORDER BY in the initialization and recursive Sub-SELECTs. All systems allow you to specify an ORDER BY in the Main-SELECT. Consult your SQL Manual to determine your system's rules.

- Other Operations

The Main-SELECT can specify other SQL operations (e.g., join-operations, Sub-SELECTs, grouping, etc.), but it cannot specify another recursive operation.

## Hierarchal Sequences

Sample Query 30.1 did not specify an ORDER BY clause. However, the result table was *incidentally* displayed in a hierarchical sequence called the "Breadth-First" hierarchical sequence. The following Figure 30.4a illustrates a breadth-first hierarchical sequence by ENO values.

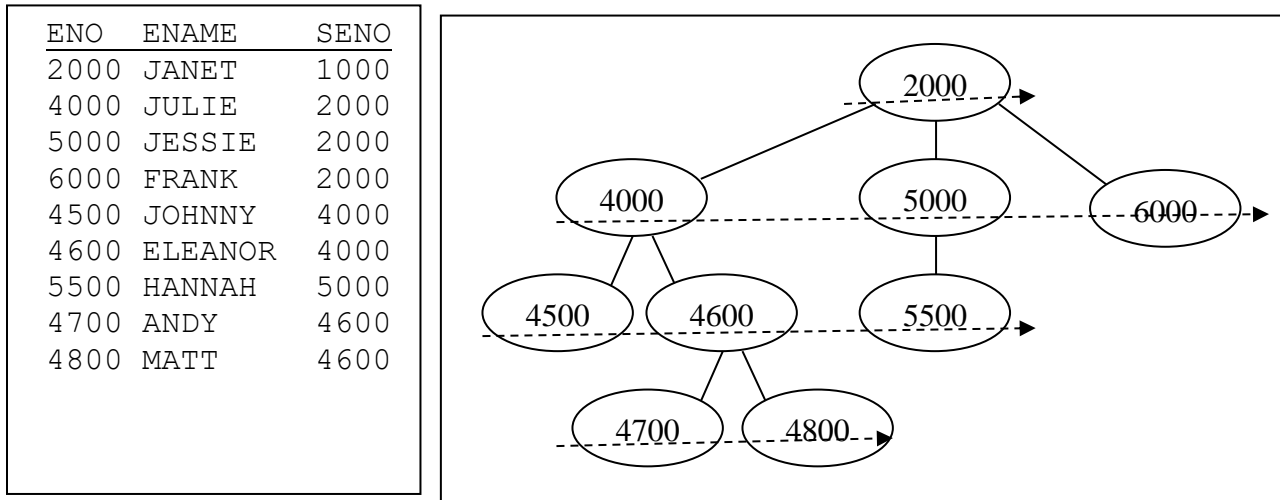


Figure 30.4a: Breadth-First Hierarchical Sequence

Casually speaking, the Breadth-First sequence can be described as: "First left-to-right, then top-to-bottom."

Another hierarchical sequence, the "Depth-First" Hierarchical Sequence, is illustrated below in Figure 30.4b. A casual description of this sequence is: "First top-to-bottom, then left-to-right." The following page describes a depth-first traversal starting at Node-2000.

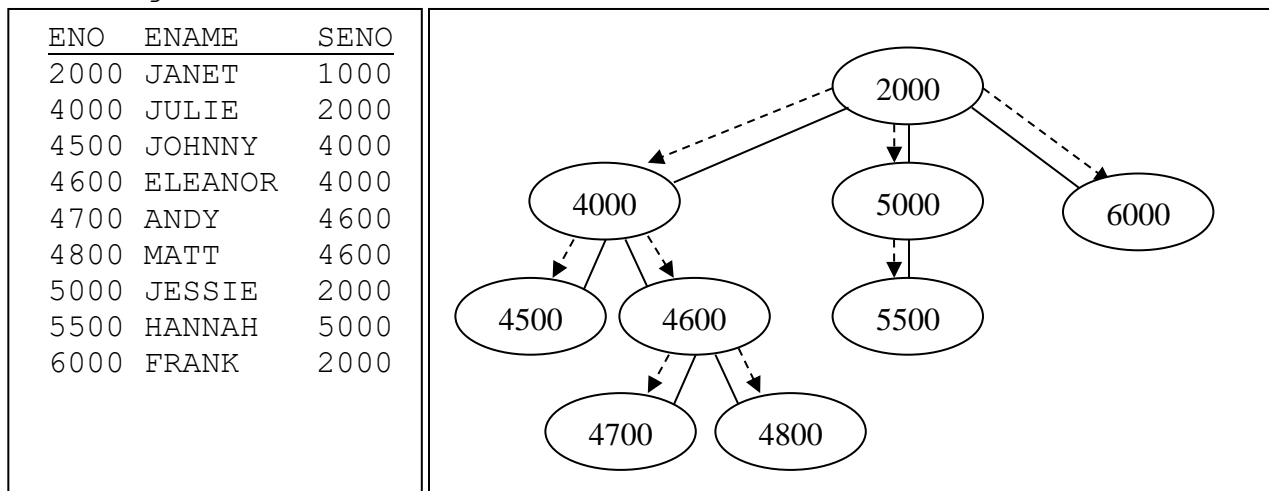


Figure 30.4b: Depth-First Hierarchical Sequence

Start at Node-2000. Because this node has multiple children, we accessed its leftmost child, Node-4000. Because this node has multiple children, we accessed its leftmost child, Node-4500. Because this node does not have any children, we returned to its parent (Node-4000) and accessed the next (not yet retrieved) leftmost child, Node-4600. Because this node has multiple children, we accessed its leftmost child, Node-4700. Because this node does not have any children, we returned to its parent (Node-4600) and accessed the next (not yet retrieved) leftmost child, Node-4800. Because this node does not have any children, we returned to its parent (Node-4600) and then its grandparent (Node-4000). Because all descendants of Node-4000 have been retrieved, we returned to Node-2000 and retrieved its (not yet retrieved) leftmost child, Node-5000. Etc.

**\*\*\* Important:** For tutorial reasons, many sample queries in this chapter will disregard our recommendation to always specify an ORDER BY clause to return a result in a desired row sequence. (Do not rely on an incidentally sorted result.) The Summary to this chapter will address this issue.

**\*\*\* Also:** Many sample queries in this chapter will incidentally default to a breath-first hierarchical sequence. (Currently this default sequence applies to DB2 and ORACLE, but this could change.) A following section will discuss the explicit specification of a desired hierarchical sequence.

### **Exercises:**

- 30D1. Reconsider the rows in the RDEMO1 table shown in Figure 30.2. Using pencil and paper, display all these rows in Breadth-First Hierarchical Sequence.
- 30D2. Append ORDER BY ENO to the Main-SELECT in Sample Query 30.1. Execute the statement. Observe that the rows are no longer displayed in breadth-first hierarchical sequence.
- 30D3: Reconsider the RDEMO1 table shown in Figure 30.2. Using pencil and paper, display its rows in depth-first hierarchical sequence.

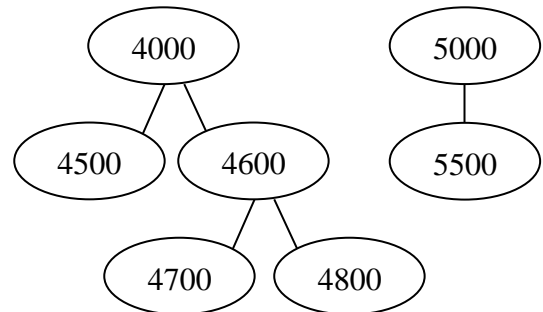
## Restriction within the Initialization Sub-SELECT

The initialization Sub-SELECT can place multiple rows into DESCENDANTS, as illustrated below.

**Sample Query 30.2a:** Reference the REMPLOYEE table. Display ENO, ENAME, and SENO values about Employees 4000 and 5000 and all their descendants.

```
WITH DESCENDANTS (ENO, ENAME, SENO)
AS
(SELECT   ENO, ENAME, SENO
 FROM     REMPLOYEE
 WHERE    ENO IN ('4000', '5000')
 UNION ALL
 SELECT   R.ENO, R.ENAME, R.SENO
 FROM     DESCENDANTS D, REMPLOYEE R
 WHERE    D.ENO = R.SENO
 )
SELECT * FROM DESCENDANTS
```

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
4000	JULIE	2000
5000	JESSIE	2000
4500	JOHNNY	4000
4600	ELEANOR	4000
5500	HANNAH	5000
4700	ANDY	4600
4800	MATT	4600



**Syntax and Logic:** Nothing New. The initialization Sub-SELECT places the following two rows into DESCENDANTS.

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
4000	JULIE	2000
5000	JESSIE	2000

Thereafter, the recursive Sub-SELECT and Main-SELECT operate as previously described.

**Observation:** The final result table does not contain any duplicate rows. This happened because the nodes for the initial two rows (Node-4000 and Node-5000) are not on the same hierarchical path.

**Duplicate Rows in Final Result:** Assume the initialization Sub-SELECT specified the following WHERE-clause with *two ENO values (4000 and 4600) that are on the same path.*

```
WHERE ENO IN ('4000', '4600')
```

This would produce the adjacent result table that contains duplicate rows. We could specify DISTINCT in the Main-SELECT to remove duplicate rows. However, DISTINCT may have an unwanted side-effect that will be discussed later.

ENO	ENAME	SENO
4000	JULIE	2000
4600	ELEANOR	4000
4500	JOHNNY	4000
4600	ELEANOR	4000
4700	ANDY	4600
4800	MATT	4600
4700	ANDY	4600
4800	MATT	4600

### Exercises:

30E1. Consider the RDEMO1 table shown in Figure 30.2. What is the result of executing the following statement? Execute the statement to verify your answer.

```
WITH DESCENDANTS (PKEY, CODE, FKEY)
AS
(SELECT PKEY, CODE, FKEY
 FROM RDEMO1
 WHERE PKEY IN (15, 40)
 UNION ALL
 SELECT R.PKEY, R.CODE, R.FKEY
 FROM RDEMO1 R, DESCENDANTS D
 WHERE R.FKEY = D.PKEY
 )
SELECT * FROM DESCENDANTS
```

30E2. Again, consider the RDEMO1 table. What is the result of executing the following statement? Observe and explain the presence of duplicate rows in the result. Execute the statement to verify your answer.

```
WITH DESCENDANTS (PKEY, CODE, FKEY)
AS
(SELECT PKEY, CODE, FKEY
 FROM RDEMO1
 WHERE PKEY IN (25, 40)
 UNION ALL
 SELECT R.PKEY, R.CODE, R.FKEY
 FROM RDEMO1 R, DESCENDANTS D
 WHERE R.FKEY = D.PKEY
 )
SELECT * FROM DESCENDANTS
```

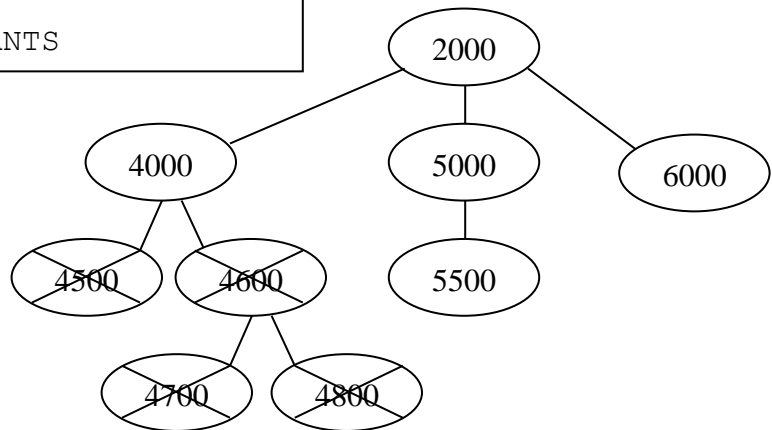
## Restriction within the Recursive Sub-SELECT

The recursive Sub-SELECT in the following sample query specifies a downward tree traversal with an additional restriction (`D.ENO <> '4000'`). This restriction will trim all nodes in all branches in the sub-tree below Node-4000.

**Sample Query 30.2b:** Display ENO, ENAME, and SENO values about Employee 2000 and her descendants. However, exclude any row describing a descendent of Employee 4000. (The row describing Employee 4000 should be included in the result.)

```
WITH DESCENDANTS (ENO, ENAME, SENO)
AS
(SELECT   ENO, ENAME, SENO
 FROM     REMPLOYEE
 WHERE    ENO = '2000'
  UNION ALL
 SELECT   R.ENO, R.ENAME, R.SENO
 FROM     DESCENDANTS D, REMPLOYEE R
 WHERE    D.ENO = R.SENO
  AND     D.ENO <> '4000'
 )
SELECT * FROM DESCENDANTS
```

ENO	ENAME	SENO
2000	JANET	1000
4000	JULIE	2000
5000	JESSIE	2000
6000	FRANK	2000
5500	HANNAH	5000



**Syntax:** Nothing New.

**Logic:** The `D.ENO <> '4000'` restriction prevents all descendants of Node-4000 (Node-4500, Node-4600, Node-4700, and Node-4800) from being placed into DESCENDANTS. Observe that the row for Node-4000 is in the result table. A step-by-step description follows.



The initialization Sub-SELECT places the following row in DESCENDANTS.

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
2000	JANET	1000

The first execution of the recursive Sub-SELECT retrieves the three descendants of Employee 2000 and places them into the DESCENDANTS table. These rows become the new rows. DESCENDANTS now looks like:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
2000	JANET	1000
4000	JULIE	2000 ←new
5000	JESSIE	2000 ←new
6000	FRANK	2000 ←new

Note: Row for Employee 4000 is in this intermediate result.

The second execution of the recursive Sub-SELECT retrieves the descendants of the new rows, but the D.ENO <> '4000' condition prevents the four descendants of Employee 4000 from being selected and placed into DESCENDANTS. DESCENDANTS now looks like:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
2000	JANET	1000
4000	JULIE	2000
5000	JESSIE	2000
6000	FRANK	2000
5500	HANNAH	5000 ←new

Note that rows for Employees 4500 and 4600, the children of Employee 4000, are not in this result. Therefore, during subsequent executions of the recursive Sub-SELECT, no other descendants of Employee 4000 (e.g., Employees 4700 and 4800) will be placed into DESCENDANTS. *The D.ENO <> '4000' condition has effectively trimmed all branches of the tree below node-4000.*

The next execution of the recursive Sub-SELECT attempts to retrieve descendants of Employee 5500 but encounters a "no hit" event which terminates the recursive execution.

Finally, the Main-SELECT displays all rows in DESCENDANTS.

## Restriction within the Main-SELECT

After you populate the DESCENDANTS table, you might want to display just some subset of rows from this table by specifying a WHERE-clause in the Main-SELECT.

**Sample Query 30.2c:** This query objective is similar to Sample Query 30.1: Display ENO, ENAME, and SENO values for Employee 2000 and all her descendants with the exception of the row describing Employee 4000. (We still want to display data about the descendants of Employee 4000.)

```
WITH DESCENDANTS (ENO, ENAME, SENO)
AS
(SELECT   ENO, ENAME, SENO
 FROM     REMPLOYEE
 WHERE    ENO = '2000'
  UNION ALL
 SELECT   R.ENO, R.ENAME, R.SENO
 FROM     DESCENDANTS D, REMPLOYEE R
 WHERE    D.ENO = R.SENO
 )
SELECT * FROM DESCENDANTS
WHERE   ENO <> '4000'
```

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
2000	JANET	1000
5000	JESSIE	2000
6000	FRANK	2000
4500	JOHNNY	4000
4600	ELEANOR	4000
5500	HANNAH	5000
4700	ANDY	4600
4800	MATT	4600

**Syntax & Logic:** Nothing New. Compared to the previous Sample Query 30.2b, this statement simply moved the ENO <> '4000' condition from the recursive Sub-SELECT to the Main-SELECT.

**Important Observation:** The Main-SELECT selects all DESCENDANTS rows except the row for Employee 4000. Note that rows describing descendants of Employee 4000 (Employees 4500, 4600, 4700 and 4800) are displayed.

## Exercises

30F. This exercise focuses on the significant difference between specifying a restriction in the recursive Sub-SELECT versus the Main SELECT. Consider the following statements which reference the RDEMO1 table shown in Figure 30.2. What is the result of executing each statement? Execute each statement to verify your answers.

```
WITH DESCENDANTS (PKEY, CODE, FKEY)
AS
(SELECT      PKEY, CODE, FKEY
 FROM        RDEMO1
 WHERE       PKEY = 20
  UNION ALL
 SELECT      R.PKEY, R.CODE, R.FKEY
 FROM        RDEMO1 R, DESCENDANTS D
 WHERE       R.FKEY = D.PKEY
 AND        R.CODE = 0      ←
 )
SELECT * FROM DESCENDANTS
```

```
WITH DESCENDANTS (PKEY, CODE, FKEY)
AS
(SELECT      PKEY, CODE, FKEY
 FROM        RDEMO1
 WHERE       PKEY = 20
  UNION ALL
 SELECT      R.PKEY, R.CODE, R.FKEY
 FROM        RDEMO1 R, DESCENDANTS D
 WHERE       R.FKEY = D.PKEY
 )
SELECT * FROM DESCENDANTS
WHERE       CODE = 0      ←
```

30G. Modify Exercise 30A which displayed the ENO, ENAME, SALARY, and SENO values for Employee 8000 and all employees who directly or indirectly work for this employee. This time only display information about an employee who directly or indirectly works for Employee 8000 if the employee's salary exceeds \$6500.00. The result should look like:

ENO	ENAME	SALARY	SENO
8000	JOE	8000.00	1000
8500	GEORGE	7000.00	8000

30H. Display the ENO, ENAME, and SENO values for Employee 2000 and all her descendants. However, exclude the row describing Employee 4000 and all descendants of this employee. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
2000	JANET	1000
5000	JESSIE	2000
6000	FRANK	2000
5500	HANNAH	5000

Code two solutions.

Solution-1 should specify the same restriction in both the recursive Sub-SELECT and the Main-SELECT. The recursive Sub-SELECT stores a row for Employee 4000 into DESCENDANTS but eliminates all its descendants. The Main-SELECT eliminates the row for Employee 4000.

Solution-2 specifies just one restriction in the recursive Sub-SELECT which prevents the row for EMPLOYEE 4000 from being placed into DESCENDANTS. Hence, all of its descendants will not be placed into DESCENDANTS.

30I. Display the ENO, ENAME, SALARY, and SENO values of Employee 2000. Also display these values for any employee who directly or indirectly works for this employee with the following exception. Do not display information about an employee *and his dependents* if the employee earns less than \$1000.00. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>SENO</u>
2000	JANET	2000.00	1000
6000	FRANK	9000.00	2000

Code two solutions similar to the two solutions for the preceding Exercise 30H.

30J. What is total salary of all employees who report to Employee 8000? The result should look like:

<u>TOTSAL</u>
19000.00

Hint: You only need to modify the Main-SELECT in the solution for Exercise 30A2.

The following two exercises do *not* require you to code recursive SQL. Careful! Observe the null value in the SEN0 column.

30K1. Display the ENO, ENAME, SALARY values for all supervisors. Sort the result by ENO values. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>
1000	MOE	2000.00
2000	JANET	2000.00
3000	LARRY	3000.00
4000	JULIE	500.00
4600	ELEANOR	3000.00
5000	JESSIE	400.00
6500	CURLY	8000.00
8000	JOE	8000.00
8500	GEORGE	7000.00

Note: This result is not in hierarchical sequence.

30K2. Display the ENO, ENAME, SALARY values for all non-supervisors. Sort the result by ENO values. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>
4500	JOHNNY	2000.00
4700	ANDY	2000.00
4800	MATT	3000.00
5500	HANNAH	4000.00
6000	FRANK	9000.00
7500	SHEMP	9000.00
8600	DICK	6000.00
8700	HANK	6000.00

Note: This result is not in hierarchical sequence.

## Summarizing Down Each Path

A calculation can be performed during each iteration of a recursive Sub-SELECT. In the following sample query, a running total of SALARY values is calculated along each hierarchical path.

**Sample Query 30.3:** Display the ENO, ENAME, SALARY, and SENO values for Employee 2000 and all her descendants. Also, display a running total (TOTPATH) of each employee's salary plus the total salary of her direct and indirect supervisors.

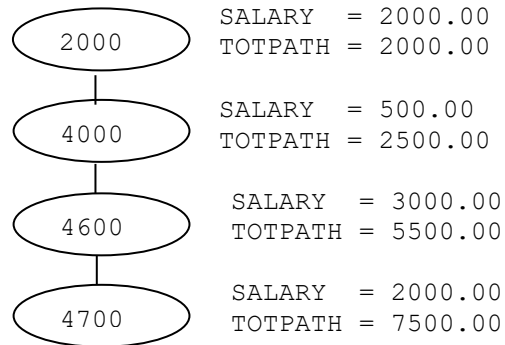
```

WITH DESCENDANTS (ENO, ENAME, SALARY, TOTPATH, SENO)
AS
(SELECT  ENO, ENAME, SALARY, SALARY, SENO
 FROM    REMPLOYEE
 WHERE   ENO = '2000'
  UNION ALL
 SELECT  R.ENO, R.ENAME,
        R.SALARY, R.SALARY + D.TOTPATH, R.SENO
 FROM    DESCENDANTS D, REMPLOYEE R
 WHERE   D.ENO = R.SENO
 )
SELECT * FROM DESCENDANTS

```

ENO	ENAME	SALARY	TOTPATH	SENO
2000	JANET	2000.00	2000.00	1000
4000	JULIE	500.00	2500.00	2000
5000	JESSIE	400.00	2400.00	2000
6000	FRANK	9000.00	11000.00	2000
4500	JOHNNY	2000.00	4500.00	4000
4600	ELEANOR	3000.00	5500.00	4000
5500	HANNAH	4000.00	6400.00	5000
4700	ANDY	2000.00	7500.00	4600
4800	MATT	3000.00	8500.00	4600

**Logic:** As data from each new REMPLOYEE row is placed into DESCENDANTS, its SALARY value is added the TOTPATH value of its parent row. This creates a running total down each path. For example, the adjacent diagram illustrates the running total of SALARY values on the path from Node-2000 to Node-4700.



## Concatenation Down Each Path

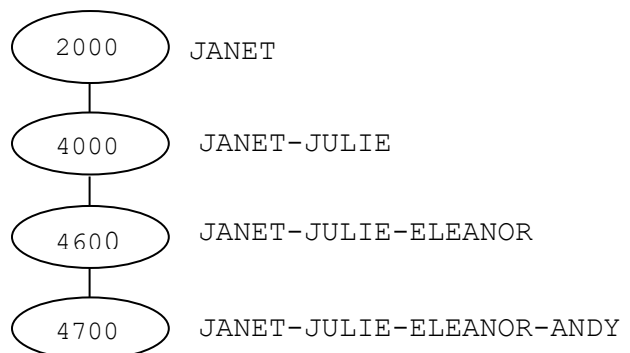
The following sample query is similar to the previous sample query. The only difference is that a concatenation operation is performed instead of an addition operation.

**Sample Query 30.4:** Display the ENO, ENAME, and SNO values of Employee 2000 and all her descendants. Also, for each employee, display a list of names (NAMELIST) containing the employee's name and the names of the employee's direct and indirect supervisors.

```
WITH DESCENDANTS (ENO, ENAME, SENO, NAMELIST)
AS
(SELECT   ENO, ENAME, SENO, ENAME
 FROM     REMPLOYEE
 WHERE    ENO = '2000'
  UNION ALL
 SELECT   R.ENO, R.ENAME, R.SENO, D.NAMELIST || '-' || R.ENAME
 FROM     DESCENDANTS D, REMPLOYEE R
 WHERE    D.ENO = R.SENO
 )
SELECT * FROM DESCENDANTS
```

ENO	ENAME	SENO	NAMELIST
2000	JANET	1000	JANET
4000	JULIE	2000	JANET-JULIE
5000	JESSIE	2000	JANET-JESSIE
6000	FRANK	2000	JANET-FRANK
4500	JOHNNY	4000	JANET-JULIE-JOHNNY
4600	ELEANOR	4000	JANET-JULIE-ELEANOR
5500	HANNAH	5000	JANET-JESSIE-HANNAH
4700	ANDY	4600	JANET-JULIE-ELEANOR-ANDY
4800	MATT	4600	JANET-JULIE-ELEANOR-MATT

**Logic:** As data from each new REMPLOYEE row is placed into DESCENDANTS, its ENAME value is concatenated to the NAMELIST value of its parent row. This creates a growing list of ENAME values. For example, the adjacent diagram illustrates the concatenation of ENAME values on the path from Node-2000 to Node-4700.



## Recursive Query: Traverse Up a Tree

The syntax and logic for traversing up a tree is similar to traversing down a tree. The only difference pertains to the join-condition in the recursive Sub-SELECT. Also, although the CTE name is specified by the user, in this book, when performing an upward traversal, we will always specify ANCESTORS as the name of the recursive CTE.

**Sample Query 30.5:** Start with Employee 4600. Display the ENO, ENAME, and SENO values for this employee and her direct and indirect supervisors.

```
WITH ANCESTORS (ENO, ENAME, SENO)
AS
(SELECT ENO, ENAME, SENO
 FROM   REMPLOYEE
 WHERE  ENO = '4600'
 UNION ALL
 SELECT R.ENO, R.ENAME, R.SENO
 FROM   ANCESTORS A, REMPLOYEE R
 WHERE A.SENO = R.ENO ←
 )
SELECT * FROM ANCESTORS
```

ENO	ENAME	SENO
4600	ELEANOR	4000
4000	JULIE	2000
2000	JANET	1000
1000	MOE	-

**Syntax:** Nothing New.

**Logic:** You want to insert the ancestor rows for Employee 4600 into ANCESTORS. The following step-by-step walkthrough illustrates how the A.SENO = R.ENO join-condition realizes this objective.

**Observation:** It is easier to verify the correctness of this result because, in an upward traversal of a tree, each node (except the root node) has exactly one parent.



**Step-by-Step Walk-Through:** The initial Sub-SELECT stores data about Employee 4600 into the ANCESTORS table.

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
4600	ELEANOR	4000

The first execution of the recursive Sub-SELECT will store data about the supervisor of Employee 4600 into the ANCESTORS table. The SENNO value (4000) identifies this supervisor. Hence, the A.SENNO = R.ENO join-condition reduces to 4000 = R.ENO, and data about Employee 4000 is stored into the ANCESTORS table. The ANCESTORS table now looks like:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
4600	ELEANOR	4000
4000	JULIE	2000 ← new

The second execution of the recursive Sub-SELECT retrieves the parent of Employee 4000, which is Employee 2000. This new row is placed into ANCESTORS which now looks like:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
4600	ELEANOR	4000
4000	JULIE	2000
2000	JANET	1000 ← new

The third execution of the recursive Sub-SELECT retrieves the parent of Employee 2000, which is Employee 1000. This new row is placed into ANCESTORS which now looks like:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
4600	ELEANOR	4000
4000	JULIE	2000
2000	JANET	1000
1000	MOE	- ← new

The last execution of the recursive Sub-SELECT attempts to find the parent of Employee 1000. But the null SENNO value produces a "no hit." Therefore, no new rows are placed in ANCESTORS, and the recursive execution terminates.

Finally, the Main-SELECT displays the ANCESTORS table.

## Recap: Recursive Join-Condition → Direction of Tree Traversal

**Downward Traversal:** In Sample Query 30.1, after the initialization Sub-SELECT stored the row for Employee 2000 into DESCENDANTS, the recursive Sub-SELECT specified a downward traversal by coding D.ENO = R.SENO as the join-condition.

FROM DESCENDANTS D, REMPLOYEE R						
WHERE D.ENO = R.SENO						
DESCENDANTS			REMPLOYEE			
ENO	ENAME	SENO	ENO	ENAME	SALARY	SENO
2000	LARRY	1000	4000	SHEMP	500.00	2000
			5000	JOE	400.00	2000
			6000	GEORGE	9000.00	2000

Here, we want to insert the children of Employee 2000 into DESCENDANTS. Within REMPLOYEE, these children have SENO values of 2000. In general, downward traversal is realized by building a DESCENDANTS table where the join-condition is:

DESCENDANTS.PRIMARYKEY = RECURSIVETABLE.FOREIGNKEY

**Upward Traversal:** In Sample Query 30.5, after the initialization Sub-SELECT stored the row for Employee 4600 into ANCESTORS, the recursive Sub-SELECT specified an upward traversal by coding A.SENO = R.ENO as the join-condition.

FROM ANCESTORS A, REMPLOYEE R						
WHERE A.SENO = R.ENO						
ANCESTORS			REMPLOYEE			
ENO	ENAME	SENO	ENO	ENAME	SALARY	SENO
4600	JESSIE	4000	4000	SHEMP	500	4600

Here, you want to insert the parent of Employee 4600 into ANCESTORS. Within REMPLOYEE, this parent has an ENO value of 4000. In general, upward traversal is realized by building a ANCESTORS table where the join-condition is:

ANCESTORS.FOREIGNKEY = RECURSIVETABLE.PRIMARYKEY

## Duplicate Rows in Result Table

The following sample query is similar to the preceding Sample Query 30.5. The only difference is that the initialization Sub-SELECT selects two rows. This causes the result table to contain some duplicate rows.

**Sample Query 30.6:** Start with Employees 4500 and 4600. Display the ENO, ENAME, and SENO values of these employees and all their direct or indirect supervisors.

```
WITH ANCESTORS (ENO, ENAME, SENO)
AS
(SELECT ENO, ENAME, SENO
 FROM   REMPLOYEE
 WHERE ENO IN ('4500', '4600')
 UNION ALL
 SELECT R.ENO, R.ENAME, R.SENO
 FROM   ANCESTORS A, REMPLOYEE R
 WHERE A.SENO = R.ENO
 )
SELECT * FROM ANCESTORS;
```

ENO	ENAME	SENO
4500	JOHNNY	4000
4600	ELEANOR	4000
4000	JULIE	2000
4000	JULIE	2000
2000	JANET	1000
2000	JANET	1000
1000	MOE	-
1000	MOE	-

**Logic:** Because the initialization Sub-SELECT selected two rows, there are two upward paths from each corresponding node to the root-node. These are:

```
Path-1: Node-4500 → Node-4000 → Node-2000 → Node-1000
Path-2: Node-4600 → Node-4000 → Node-2000 → Node-1000
```

Duplicate rows appear in the result table because both of these paths encounter the same three nodes. If desired, you could specify DISTINCT in the Main-SELECT to eliminate this duplication. However, DISTINCT has a potentially unwanted side-effect that will be discussed later.

## Exercises

30L. Display the ENO, ENAME, SALARY, and SENO values for Employee 4000 and all her direct and indirect supervisees. Also, for each employee, display a running total of the employee's salary plus the total salary of all her direct and indirect supervisors. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>TOTPATH</u>	<u>SENO</u>
4000	JULIE	500.00	500.00	2000
4500	JOHNNY	2000.00	2500.00	4000
4600	ELEANOR	3000.00	3500.00	4000
4700	ANDY	2000.00	5500.00	4600
4800	MATT	3000.00	6500.00	4600

Hint: This exercise is similar to Sample Query 30.3.

30M. Start with Employee 5500. Display all data about this employee and all data about her direct or indirect supervisors. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>SENO</u>
5500	HANNAH	4000.00	5000
5000	JESSIE	400.00	2000
2000	JANET	2000.00	1000
1000	MOE	2000.00	-

30N. Reference the RDEMO1 table described in Exercise 30C. Within the context of upward tree traversal, this exercise focuses on the specification a restriction in the recursive Sub-SELECT and Main-SELECT. Consider the following three statements. What is the result of executing each statement? Execute each statement to verify your answer.

```
Statement 30N-1
WITH ANCESTORS (PKEY, CODE, FKEY)
AS
(SELECT      PKEY, CODE, FKEY
 FROM        RDEMO1
 WHERE      PKEY = 35
  UNION ALL
 SELECT      R.PKEY, R.CODE, R.FKEY
 FROM        ANCESTORS A, RDEMO1 R
 WHERE      A.FKEY = R.PKEY
)
SELECT * FROM ANCESTORS
WHERE      CODE = 0           ←
```

```
Statement 30N-2
WITH ANCESTORS (PKEY, CODE, FKEY)
AS
(SELECT      PKEY, CODE, FKEY
 FROM        RDEMO1
 WHERE      PKEY = 35
  UNION ALL
 SELECT      R.PKEY, R.CODE, R.FKEY
 FROM        ANCESTORS A, RDEMO1 R
 WHERE      A.FKEY = R.PKEY
 AND        A.CODE = 0       ←
)
SELECT * FROM ANCESTORS
```

```
Statement 30N-3
WITH ANCESTORS (PKEY, CODE, FKEY)
AS
(SELECT      PKEY, CODE, FKEY
 FROM        RDEMO1
 WHERE      PKEY = 35
  UNION ALL
 SELECT      R.PKEY, R.CODE, R.FKEY
 FROM        ANCESTORS A, RDEMO1 R
 WHERE      A.FKEY = R.PKEY
 AND        R.CODE = 0       ←
)
SELECT * FROM ANCESTORS
```

## Displaying “Level Numbers” for Downward Tree Traversal

The following sample query, which does a downward tree traversal, assigns a level number (LVL) to each selected row. Level number 1 is assigned to the row selected by the initialization Sub-SELECT.

**Sample Query 30.7a:** Enhance Sample Query 30.1 such that it assigns and displays level numbers to rows in the result table.

```
WITH DESCENDANTS (LVL, ENO, ENAME, SENO)
AS
(SELECT 1, ENO, ENAME, SENO
 FROM   REMPLOYEE
 WHERE  ENO = '2000'
  UNION ALL
 SELECT D.LVL+1, R.ENO, R.ENAME, R.SENO
 FROM   DESCENDANTS D, REMPLOYEE R
 WHERE  D.ENO = R.SENO
 )
SELECT * FROM DESCENDANTS
```

<u>LVL</u>	<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
1	2000	JANET	1000
2	4000	JULIE	2000
2	5000	JESSIE	2000
2	6000	FRANK	2000
3	4500	JOHNNY	4000
3	4600	ELEANOR	4000
3	5500	HANNAH	5000
4	4700	ANDY	4600
4	4800	MATT	4600

**Syntax:** LVL is not a reserved word. It is just another user-specified name. (ORACLE supports the pseudo-column LEVEL. See ORACLE SQL manual.)

**Logic:** An LVL value of 1 is assigned to the first selected row. Thereafter, LVL is incremented at each execution of the recursive Sub-SELECT. We emphasize that LVL is *not* an absolute level number in the tree that represents the entire table (as shown in Figure 30.1c). In this example, the first retrieved row for Employee 2000 is assigned an LVL of 1, but this row corresponds to a second-level node in tree shown in Figure 30.1c.

## Limiting Levels in Downward Traversal

Sample Query 30.2b limited the downward tree traversal by specifying a WHERE-clause in the recursive Sub-SELECT. This WHERE-clause effectively "trimmed some branches" of the tree. Sometimes, a "tree trimming" objective is simpler. You simply want to limit the number of levels the system will traverse in its downward traversal. The following example limits the downward traversal to three levels.

**Sample Query 30.7b:** Make a minor enhancement to the previous Sample Query 30.7a by limiting the downward traversal to three levels.

```
WITH DESCENDANTS (LVL, ENO, ENAME, SENO)
AS
(SELECT 1, ENO, ENAME, SENO
 FROM   REMPLOYEE
 WHERE  ENO = '2000'
 UNION ALL
 SELECT D.LVL+1, R.ENO, R.ENAME, R.SENO
 FROM   DESCENDANTS D, REMPLOYEE R
 WHERE  D.ENO = R.SENO
 AND   D.LVL+1 <= 3
 )
SELECT * FROM DESCENDANTS
```

<u>LVL</u>	<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
1	2000	JANET	1000
2	4000	JULIE	2000
2	5000	JESSIE	2000
2	6000	FRANK	2000
3	4500	JOHNNY	4000
3	4600	ELEANOR	4000
3	5500	HANNAH	5000

**Syntax & Logic:** Nothing new. Specifying "D.LVL+1 <= 3" in the recursive Sub-SELECT limits the downward traversal to three levels.

## Displaying “Level Numbers” for Upward Tree Traversal

The following sample query, which does an upward tree traversal, assigns level numbers (LVL) to selected rows. Level number 1 is assigned to the row selected by the initialization Sub-SELECT.

**Sample Query 30.8a:** Enhance Sample Query 30.5. Start with Employee 4600. Display the ENO, ENAME, and SENO values for this employee and all her direct and indirect supervisors. Assign and display level numbers in the result table.

```
WITH ANCESTORS (LVL, ENO, ENAME, SENO)
AS
(SELECT 1, ENO, ENAME, SENO
 FROM   REMPLOYEE
 WHERE  ENO = '4600'
 UNION ALL
 SELECT A.LVL+1, R.ENO, R.ENAME, R.SENO
 FROM   ANCESTORS A, REMPLOYEE R
 WHERE  A.SENO = R.ENO
 )
SELECT * FROM ANCESTORS;
```

<u>LVL</u>	<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
1	4600	ELEANOR	4000
2	4000	JULIE	2000
3	2000	JANET	1000
4	1000	MOE	-

**Logic:** An LVL value of 1 is assigned to the first selected row. Thereafter, LVL is incremented at each execution of the recursive Sub-SELECT.

**Important Observation:** In this example, the first retrieved row for Employee 4600 is assigned an LVL of 1. Observe that this row corresponds to a fourth-level node in tree shown in Figure 30.1c. A user might find this assignment of level numbers to be somewhat counterintuitive. A following section (Modifying Level Numbers) will address this issue.



## Limiting Levels of Upward Traversal

The following sample query limits an upward tree traversal to three levels.

**Sample Query 30.8b:** Revise Sample Query 30.5 which displayed data about Employee 4600 and her supervisors. Display level numbers followed by the ENO, ENAME, and SENO values of this employee, her supervisor, and her supervisor's supervisor (i.e., limit the upward traversal to three levels).

```
WITH ANCESTORS (LVL, ENO, ENAME, SENO)
AS
• (SELECT 1, ENO, ENAME, SENO
  FROM   REMPLOYEE
  WHERE  ENO = '4600'
  UNION ALL
  SELECT A.LVL+1, R.ENO, R.ENAME, R.SENO
  FROM   ANCESTORS A, REMPLOYEE R
  WHERE  A.SENO = R.ENO
  AND    A.LVL+1 <= 3
 )
SELECT * FROM ANCESTORS
```

<u>LVL</u>	<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
1	4600	ELEANOR	4000
2	4000	JULIE	2000
3	2000	JANET	1000

**Syntax and Logic:** Nothing new.

## Do-It-Yourself “Poor Man’s” Formatting via Indentation

The following sample query demonstrates a common technique to illustrate hierarchical relationships among rows within a result table.

**Sample Query 30.9:** This query has the same objective as Sample Query 30.1: Display data about Employee 2000 and all her descendants. Here, you should display leading spaces in front of each row to help users visualize the hierarchical structure of the data.

```
WITH DESCENDANTS (LVL, ENO, ENAME, SENO)
AS
(SELECT 0, ENO, ENAME, SENO
 FROM   REMPLOYEE
 WHERE  ENO = '2000'
 UNION ALL
 SELECT D.LVL+1, R.ENO, R.ENAME, R.SENO
 FROM   DESCENDANTS D, REMPLOYEE R
 WHERE  D.ENO = R.SENO
 )
SELECT
(SUBSTR ('          ', 1, LVL*2)||ENO) ENO, ENAME, SENO
FROM DESCENDANTS
```

ENO	ENAME	SENO
2000	JANET	1000
4000	JULIE	2000
5000	JESSIE	2000
6000	FRANK	2000
4500	JOHNNY	4000
4600	ELEANOR	4000
5500	HANNAH	5000
4700	ANDY	4600
4800	MATT	4600

**Syntax and Logic:** Nothing new. (You may want to review the concatenation operation (||) and SUBSTR function in Chapter 10.) Recall that different systems support similar but different string processing functions. For example, A SQL Server user would code the keyword SUBSTRING (versus SUBSTR) and specify the + symbol (versus ||) for string concatenation. SQL Server also supports a SPACE function which could simplify the specification of leading spaces.

## Exercises

3001. Reference the RDEMO1 table. Display the PKEY, CODE, and FKEY values for the row with a PKEY value of 25 and all its descendants. Also display the level number for each row. The result should look like:

<u>LVL</u>	<u>PKEY</u>	<u>CODE</u>	<u>FKEY</u>
1	25	0	20
2	15	1000	25
2	40	0	25
3	30	0	15
3	50	0	40
4	35	0	30

3002. Modify the previous query objective such that downward traversal is restricted to three levels. The result should look like:

<u>LVL</u>	<u>PKEY</u>	<u>CODE</u>	<u>FKEY</u>
1	25	0	20
2	15	1000	25
2	40	0	25
3	30	0	15
3	50	0	40

30P1. Reference the RDEMO1 table. Display the PKEY, CODE, and FKEY values for the row with a PKEY value of 40 and all its ancestors. Also display the level number for each row. The result should look like:

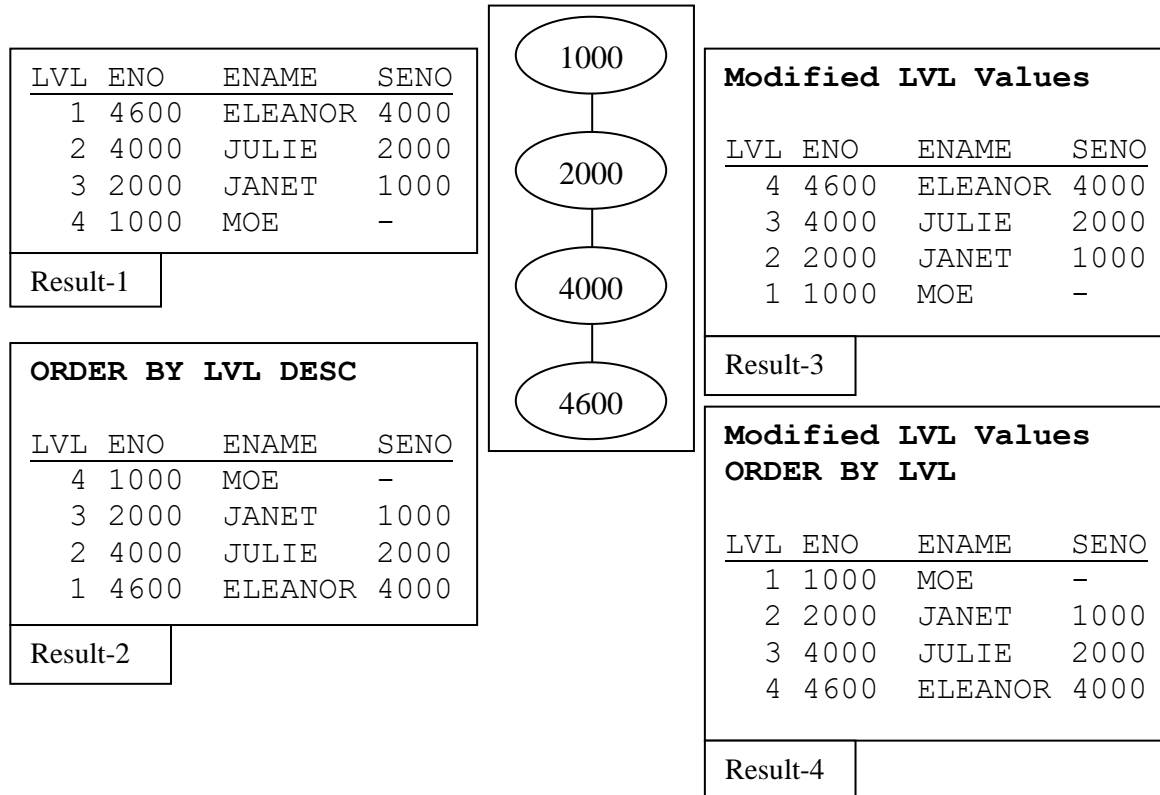
<u>LVL</u>	<u>PKEY</u>	<u>CODE</u>	<u>FKEY</u>
1	40	0	25
2	25	0	20
3	20	0	10
4	10	0	-

30P2. Modify this query objective such that upward traversal is restricted to three levels. The result should look like:

<u>LVL</u>	<u>PKEY</u>	<u>CODE</u>	<u>FKEY</u>
1	40	0	25
2	25	0	20
3	20	0	10

## Modifying Level Numbers

Consider the following Result-1 produced in Sample Query 30.8a. Here, an LVL value of 1 was assigned to the fourth-level Node-4600 shown in the adjacent sub-tree. This occurred because, in the upward traversal, the row for Employee 4600 was the first row retrieved. You might prefer to display the Result-2, Result-3, or Result-4 result tables shown below.



Result-2 is easily realized by appending ORDER BY LVL DESC to the Main-SELECT. However, note that rows are still assigned the same (perhaps counterintuitive) LVL values. Result-3 modifies the LVL values shown in Result-1. Producing Result-3 requires some coding gymnastics which will be shown in the solution to the following Exercise 30Q. Producing Result-4 simply requires appending ORDER BY LVL to the statement that produced Result-3.

### Exercise:

30Q: Modify the Main-SELECT in Sample Query 30.8a such that the result looks like Result-3. Then, append an ORDER BY clause to produce Result-4.

## Explicit Specification of Hierarchical Sequence

Previous sample queries defaulted to a breath-first hierarchical sequence. Here we consider producing a result table with a depth-first hierarchical sequence. *Currently, not all systems directly support this sequence.*

ORACLE is one of the systems that allows you to explicitly designate a specific hierarchical sequence. ORACLE provides a SEARCH-clause that is used in conjunction with the ORDER BY clause to produce either of the two hierarchical sequences. The following example produces the depth-first sequence shown in Figure 30.4b.

WITH DESCENDANTS (ENO, ENAME, SENO)	<b><u>ORACLE</u></b>
AS	
(SELECT ENO, ENAME, SENO	
FROM REEMPLOYEE	
WHERE ENO = '2000'	
UNION ALL	
SELECT R.ENO, R.ENAME, R.SENO	
FROM DESCENDANTS D, REEMPLOYEE R	
WHERE D.ENO = R.SENO	
)	
<b>SEARCH DEPTH FIRST BY ENO SET MYHSORTCOL</b>	←
SELECT * FROM DESCENDANTS	
<b>ORDER BY MYHSORTCOL</b>	←

The SEARCH DEPTH FIRST BY ENO clause designates a depth-first sequence based upon the ENO column. (Each ENO value identifies a node in the tree.) The SET MYHSORTCOL clause designates a column (MYHSORTCOL) to represent this depth-first sequence. This column contains a sequence of integer values starting at 1 and increasing by 1 (i.e., 1,2,3...). These values correspond to the order in which the nodes were visited in its depth-first traversal. The following ORDER BY clause directs the system to use the MYHSORTCOL values to display the rows according to the depth-first sequence. If you want to specify a breadth-first sequence, you would substitute BREADTH FIRST for DEPTH FIRST in the SEARCH clause.

**Suggestion:** Consult your SQL reference manual to determine if your system offers some method to generate a depth-first hierarchical sequence.

## Potential Problem: Losing Default Hierarchical Sequence

Sometimes a SQL keyword or clause can indirectly undo a default hierarchical sequence in a result table. For example, review the result for Sample Query 30.6 and note that it contains duplicate rows. Below, we modify this statement by specifying `DISTINCT` in the Main-`SELECT` and display its final result without duplicate rows.

```
Sample Query 30.6 with DISTINCT

WITH ANCESTORS (ENO, ENAME, SENO)
AS
(SELECT ENO, ENAME, SENO
 FROM   REMPLOYEE
 WHERE  ENO IN ('4500','4600')
 UNION ALL
 SELECT R.ENO, R.ENAME, R.SENO
 FROM   ANCESTORS A, REMPLOYEE R
 WHERE  A.SENO = R.ENO
 )
SELECT DISTINCT * FROM ANCESTORS
```

ENO	ENAME	SENO
2000	JANET	1000
4000	JULIE	2000
4600	ELEANOR	4000
4500	JOHNNY	4000
1000	MOE	-

**Potential Problem:** While `DISTINCT` removes duplicate rows, it may cause a potentially undesirable side-effect. Note that *the above result table is no longer sorted in a breath-first hierarchical sequence.*

Prior to this chapter, we did not care if some operation like `DISTINCT` (or grouping or join) generated rows in some incidental sort sequence. You could simply specify an `ORDER BY` clause to produce the desired sequence. (Appendix 3A explains how `DISTINCT` could produce an incidentally sorted result.) *Here, in the context of hierarchical queries, if some operation in the Main-`SELECT` disrupts a desired hierarchical sequence, you may be out of luck.*

The *traditional* `ORDER BY` clause does not allow you to specify a desired hierarchical sequence. Therefore, *hopefully*, your system supports code similar to the `SEARCH-ORDER BY` clauses described on the preceding page.

**Questionable Alternative Solution?** Why not specify `DISTINCT` in the recursive Sub-`SELECT`? Seems like good idea. However, similar to the `ORDER BY` clause, some systems prohibit specifying `DISTINCT` in the recursive Sub-`SELECT`.

## Again, Possible Loss of Default Hierarchical Sequence

Assume we want to enhance the result table for Sample Query 30.1 to display the supervisor's ENAME value along with the supervisor's ENO value.

**Sample Query 30.10:** Start with Employee 2000 and include all employees who directly or indirectly work for this employee. For each employee, display the employee's ENO, and ENAME values, followed by the ENO and ENAME values of their supervisor.

```
WITH DESCENDANTS (ENO, ENAME, SENO)
AS
(SELECT      ENO, ENAME, SENO
 FROM        REMPLOYEE
 WHERE       ENO = '2000'
  UNION ALL
 SELECT      R.ENO, R.ENAME, R.SENO
 FROM        DESCENDANTS D, REMPLOYEE R
 WHERE       D.ENO = R.SENO
 )
SELECT D.ENO, D.ENAME, R.ENO BOSSENO, R.ENAME BOSSENAME
FROM DESCENDANTS D, REMPLOYEE R
WHERE D.SENO = R.ENO
```

<u>ENO</u>	<u>ENAME</u>	<u>BOSSENO</u>	<u>BOSSENAME</u>
2000	JANET	1000	MOE
4000	JULIE	2000	JANET
5000	JESSIE	2000	JANET
6000	FRANK	2000	JANET
4500	JOHNNY	4000	JULIE
4600	ELEANOR	4000	JULIE
5500	HANNAH	5000	JESSIE
4700	ANDY	4600	ELEANOR
4800	MATT	4600	ELEANOR

**Syntax and Logic:** Nothing new. Note that, after executing the recursive Sub-SELECT, the DESCENDANTS table does not contain ENAME values. Therefore, the Main-SELECT joins DESCENDANTS with REMPLOYEE to include the supervisor's ENAME values in the final result.

**Important Observation:** The above result table happens to be in hierarchical sequence. However, some future execution of this same statement could produce a different row sequence. This could happen because of a change in the internal processing for the join-operation. (See Appendix 17A.)

## “Parent-Oriented” Design

Given a one-to-many recursive application, a rookie designer might consider designing a table where each row describes a parent-row corresponding to the parent-side of a one-to-many (parent-child) relationship. Given this perspective, the designer might consider creating the following `REMPLOYEE_V2` table. In this table, unlike the `REMPLOYEE` table, the `SENO` column designates an employee’s supervisee (not supervisor). A null value in the `SENO` column implies that the employee is not a supervisor.

REMPLOYEE_V2			
<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>SENO</u>
1000	MOE	2000.00	2000
1000	MOE	2000.00	3000
1000	MOE	2000.00	8000
2000	JANET	2000.00	4000
2000	JANET	2000.00	5000
2000	JANET	2000.00	6000
3000	LARRY	3000.00	6500
4000	JULIE	500.00	4500
4000	JULIE	500.00	4600
4500	JOHNNY	2000.00	-
4600	ELEANOR	3000.00	4700
4600	ELEANOR	3000.00	4800
4700	ANDY	2000.00	-
4800	MATT	3000.00	-
5000	JESSIE	400.00	5500
5500	HANNAH	4000.00	-
6000	FRANK	9000.00	-
6500	CURLY	8000.00	7500
7500	SHEMP	9000.00	-
8000	JOE	8000.00	8500
8500	GEORGE	7000.00	8600
8500	GEORGE	7000.00	8700
8600	DICK	6000.00	-
8700	HANK	6000.00	-

`REMPLOYEE_V2` is superficially similar to `REMPLOYEE`. All its columns have the same names and data-types. But there is an important semantic difference. In `REMPLOYEE`, the `SENO` column designates an employee’s supervisor. Whereas, in `REMPLOYEE_V2`, the `SENO` column designates an employee’s supervisee. Also, note that *some employees have multiple supervisees*.



## Avoid “Parent-Oriented” Designs

Within the context of a one-to-many recursive design, there are many problems with a parent-oriented design. First, observe that `REMPLOYEE_V2` has more rows than `REMPLOYEE`. This occurs because rows for those employees who supervise multiple supervisees have duplicate `ENO`, `ENAME` and `SALARY` values. However, this duplication is not the major problem. The real problem with `REMPLOYEE_V2` is that it is not possible to designate a valid primary-key for this table. Notice that:

- The `ENO` column cannot be the primary-key because it has duplicate values.
- The `SENO` column cannot be the primary-key because it has null values.
- Each of the `ENO`, `ENAME`, and `SALARY` columns contain duplicate values, and any two-way or three-way combination of these columns would also contain duplicates.

Therefore, you will probably not encounter a parent-oriented recursive table within a real-world application.

Having disparaged `REMPLOYEE_V2` as a base table, we note that some users may wish to display `REMPLOYEE_V2` as a result table. The following non-recursive `SELECT` statement satisfies this query objective.

```
SELECT  R1.ENO, R1.ENAME, R1.SALARY, R2.ENO
FROM    REMPLOYEE R1 LEFT OUTER JOIN REMPLOYEE R2
        ON R1.ENO = R2.SENO
ORDER BY R1.ENO
```

[Note: The following Section B will show that a parent-oriented design may not be problematic within a many-to-many recursive design.]

37R. Optional Exercise: Assume that many users would like the `REMPLOYEE_V2` table. For this reason, you decide to create a view called `REMPLOYEE_V2` that looks like the `REMPLOYEE_V2` table. Create this view, and then execute `SELECT * FROM REMPLOYEE_V2` to display its contents.

## B. Recursive Many-to-Many Relationships

In this section your attention is directed towards coding SELECT statements against tables that represent a recursive many-to-many relationship.

Assume a business organization follows a matrix management policy where an individual employee may directly supervise multiple employees, and an individual employee may have multiple direct supervisors. Hence, we have a recursive many-to-many relationship. The following Figure 30.5a represents this relationship via the REPORTS-TO relationship in the data model and the corresponding REPORTS\_TO table.

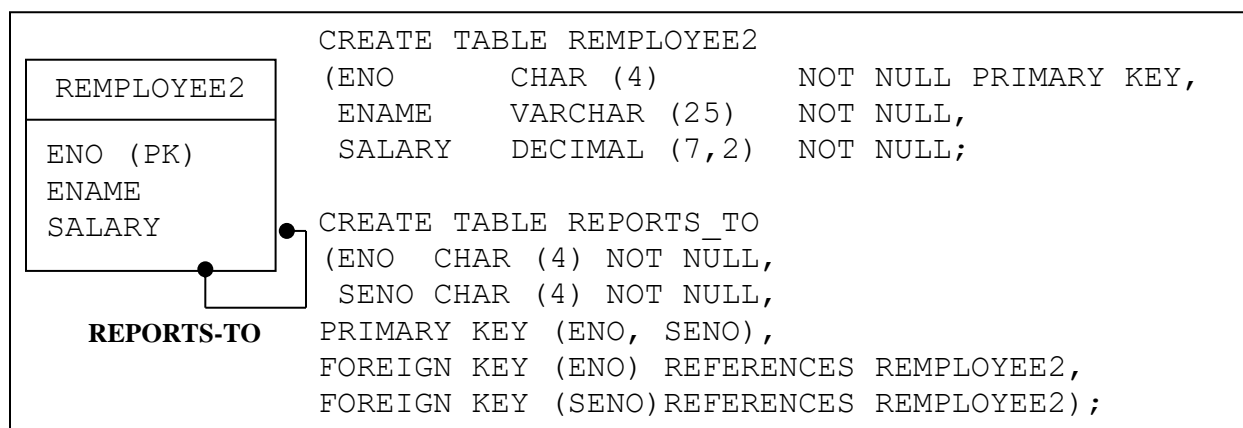


Figure 30.5a: Recursive Many-to-Many Relationship

**EMPLOYEE2 Table:** Like the EMPLOYEE table (Figure 30.1a), the EMPLOYEE2 table contains a primary-key column (ENO) and descriptive columns (ENAME, and SALARY). Unlike the EMPLOYEE table, EMPLOYEE2 does not specify a foreign-key to designate an employee's supervisor because an individual employee may have multiple supervisors.

**REPORTS\_TO Table:** A many-to-many relationship is represented by a separate table. (Review Chapter 13 and Appendix 13A.) Here, the REPORTS-TO relationship is represented by the REPORTS\_TO table. Within this table, the ENO column identifies an employee who is directly supervised by some other employee identified by the SENO column. Direct supervisee-supervisor relationships are represented by pairs of (ENO, SENO) values. Recursive logic applied to these values allows you to deduce all indirect supervisee-supervisor relationships.

## Sample Data for Recursive Many-to-Many Relationship

The following Figure 30.5b presents sample data that will be accessed by subsequent sample queries. We make some observations about this data.

**EMPLOYEE2 Table:** With one exception, all EMPLOYEE2 rows have the same ENO, ENAME, and SALARY values found in the EMPLOYEE table. This exception pertains to the row with an ENO value of 0000 which describes a fictional "DUMMY" employee. Justification for this row is presented below.

**REPORTS\_TO Table:** The first row in the REPORTS\_TO table shows that Employee 1000 (the big-boss) reports to Employee 0000, a fictitious "big-big-boss." This fiction is necessary because (ENO, SENO) is a composite primary-key, and each component of this composite primary-key must contain a non-null value, such as 0000. Now, because the SENO column is a foreign-key, 0000 must match some primary-key value in the REMPLYEE2 table. Hence, EMPLOYEE2 contains a DUMMY row with an ENO value of 0000.

EMPLOYEE2			REPORTS_TO	
ENO	ENAME	SALARY	ENO	SENO
<b>0000</b>	<b>DUMMY</b>	0000.00	1000	<b>0000</b>
1000	MOE	2000.00	2000	1000
2000	JANET	2000.00	3000	1000
3000	LARRY	3000.00	8000	1000
8000	JOE	8000.00	4000	2000
4000	JULIE	500.00	5000	2000
5000	JESSIE	400.00	6000	2000
6000	FRANK	9000.00	6500	3000
6500	CURLY	8000.00	8500	8000
8500	GEORGE	7000.00	4500	4000
4500	JOHNNY	2000.00	4600	4000
4600	ELEANOR	3000.00	5500	5000
5500	HANNAH	4000.00	7500	6500
7500	SHEMP	9000.00	8600	8500
8600	DICK	6000.00	8700	8500
8700	HANK	6000.00	4700	4600
4700	ANDY	2000.00	4800	4600
4800	MATT	3000.00	4600	5000
			4800	5500
			4800	6000

Figure 30.5b: EMPLOYEE2 and REPORTS\_TO Tables

## Recursive Many-to-Many Relationship → Network Diagram

A Network Diagram can be used to represent a many-to-many recursive relationship. The following Figure 30.5c illustrates a network diagram where each `REMPLOYEE2` row (excluding the DUMMY 0000 row) is represented by a node; and, each `REPORTS_TO` row (excluding the row referencing 0000) is represented by a line between corresponding supervisee and supervisor nodes.

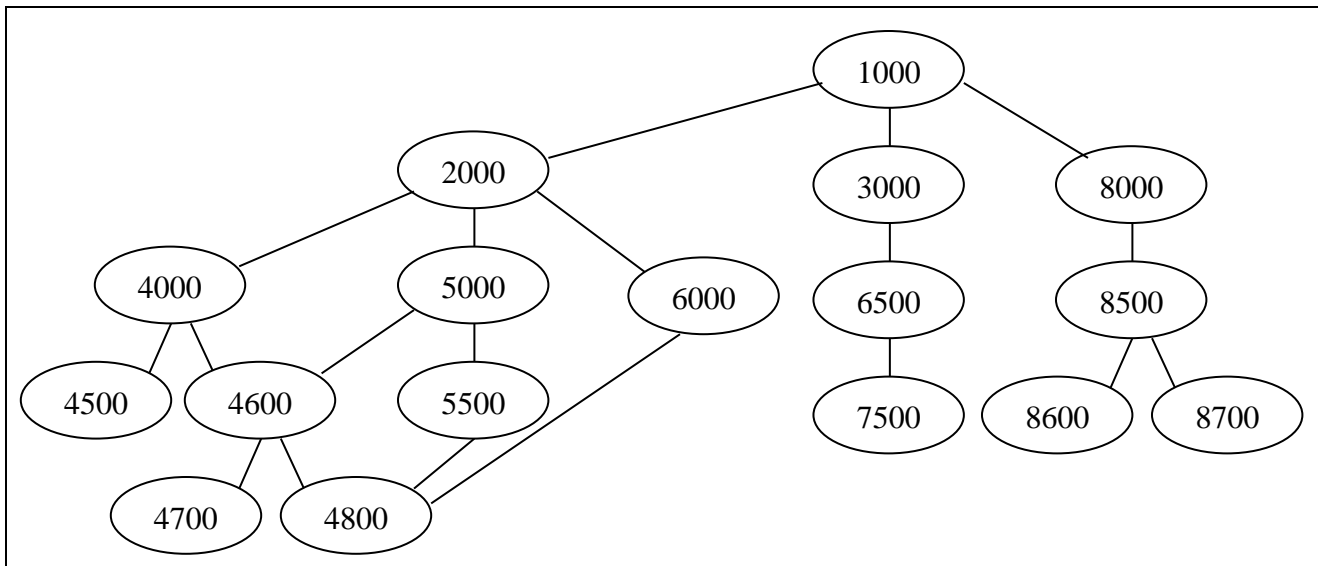


Figure 30.5c: Network Representation of `REMPLOYEE2` and `REPORTS_TO` Tables

The above network diagram is similar to the tree diagram illustrated in Figure 30.1c. It is basically the same diagram plus *three additional lines corresponding to three additional supervisee-supervisor relationships*. These additional lines connect: Node-4600 to Node-5000, Node-4800 to Node-5500, and Node-4800 to Node-6000. *This diagram is not a tree because some nodes (Node-4600 and Node-4800) have multiple parents.*

Relating this network diagram to the `REPORTS_TO` table (Figure 30.5b), observe that the `ENO` column contains some duplicate values indicating that an employee reports to (is supervised by) multiple employees. (E.g., Employee 4800 reports to Employees 4600, 5500, and 6000, and the network diagram shows three lines above Node-4800.) Also, observe that the `SENO` column contains some duplicate values indicating that an employee may supervise multiple employees. (E.g., Employee 2000 supervises Employees 4000, 5000, and 6000; hence this network diagram shows three lines below Node-2000.)

Within a tree diagram, there is only one path between a designated ancestor node and a designated descendant node. However, within a network diagram, there may be multiple paths between such nodes. For example, observe that there are two paths between Node-2000 and the intermediate (non-leaf) Node-4600.

Node-2000 → Node-4000 → Node-4600

Node-2000 → Node-5000 → Node-4600

Also, observe that there are four paths between Node-2000 and leaf Node-4800.

Node-2000 → Node-4000 → Node-4600 → Node-4800

Node-2000 → Node-5000 → Node-4600 → Node-4800

Node-2000 → Node-5000 → Node-5500 → Node-4800

Node-2000 → Node-6000 → Node-4800

**Network Represented as a Tree with Duplicate Nodes:** Sometimes, it can be helpful to represent a section of a network diagram as an equivalent tree diagram. The following tree diagram (Figure 30.5d) represents the nodes under Node-2000. Notice the (logical) redundancy in this tree diagram. It shows two copies of Node-4600, two copies of Node-4700, and four copies of Node-4800.

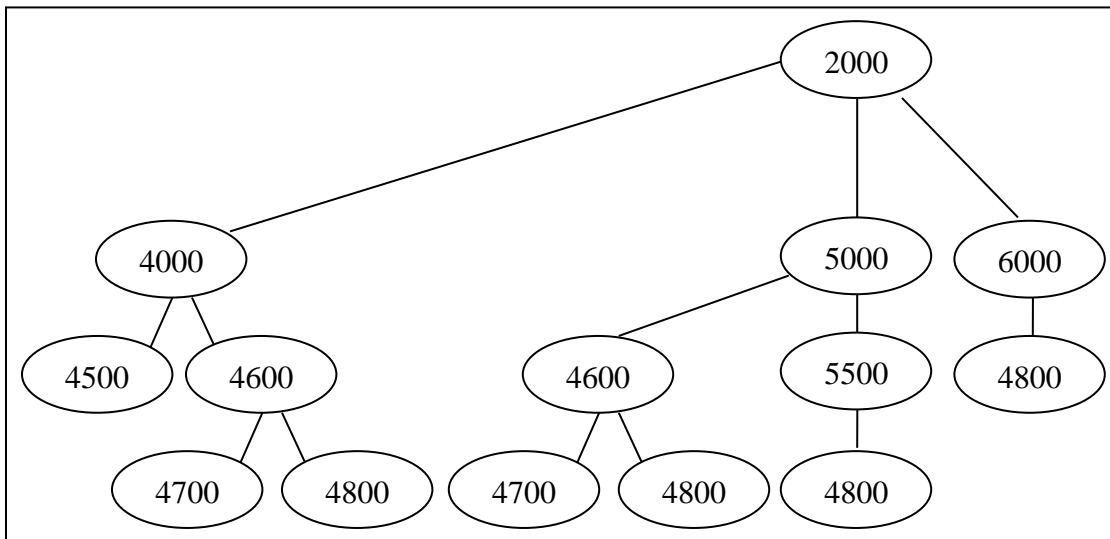


Figure 30.5d: Tree Representation of Network under Node-2000

## Traverse Down a Network

Traversing down a network utilizes the same syntax and logic as traversing down a tree. Three equivalent solutions are presented for the following sample query.

**Sample Query 30.11:** Reference the `REMPLOYEE2` and `REPORTS_TO` tables. Display the `ENO`, `ENAME`, `SALARY`, and `SENO` values for Employee 2000 and all employees who directly or indirectly work for her. The result should look like:

ENO	ENAME	SALARY	SENO		
2000	JANET	2000.00	1000	}	Level 1
4000	JULIE	500.00	2000		
5000	JESSIE	400.00	2000	}	Level 2
6000	FRANK	9000.00	2000		
4500	JOHNNY	2000.00	4000	}	Level 3
4600	ELEANOR	3000.00	4000		
4600	ELEANOR	3000.00	5000		
5500	HANNAH	4000.00	5000		
4800	MATT	3000.00	6000	}	Level 4
4700	ANDY	2000.00	4600		
4800	MATT	3000.00	4600		
4700	ANDY	2000.00	4600		
4800	MATT	3000.00	4600		
4800	MATT	3000.00	5500		

**Solution-1** (Two "Early-Joins" involving `REMPLOYEE2`)

```
WITH DESCENDANTS (ENO, ENAME, SALARY, SENO)
AS
(SELECT   R.ENO, R2.ENAME, R2.SALARY, R.SENO
 FROM     REPORTS_TO R, REMPLOYEE2 R2
 WHERE    R.ENO = R2.ENO
 AND      R.ENO = '2000'
  UNION ALL
 SELECT   R.ENO, R2.ENAME, R2.SALARY, R.SENO
 FROM     DESCENDANTS D, REPORTS_TO R, REMPLOYEE2 R2
 WHERE    R.ENO = R2.ENO
 AND      D.ENO = R.SENO
 )
SELECT * FROM DESCENDANTS
```

**Syntax & Logic:** This solution does an "early-join" (another unofficial term) by specifying `REMPLOYEE2` within the CTE Sub-SELECTs. All desired data are placed into `DESCENDANTS`. This allows coding a very simple Main-SELECT.

**Duplicate Rows in Result Table:** This result table (correctly) includes some duplicate rows. Duplicate rows will be discussed later.

**Row Sequence:** This result is in breath-first hierarchical sequence because the recursive Sub-SELECT returned the DESCENDANTS rows in this sequence, and no ORDER BY clause changed this sequence.

**Verify Result:** Examine the following Figure 30.5e which enhances the tree diagram shown in Figure 30.5d by designating level numbers on the left side of the figure. Then, by following a breadth-first hierarchical sequence, you will obtain the row sequence shown in the result table.

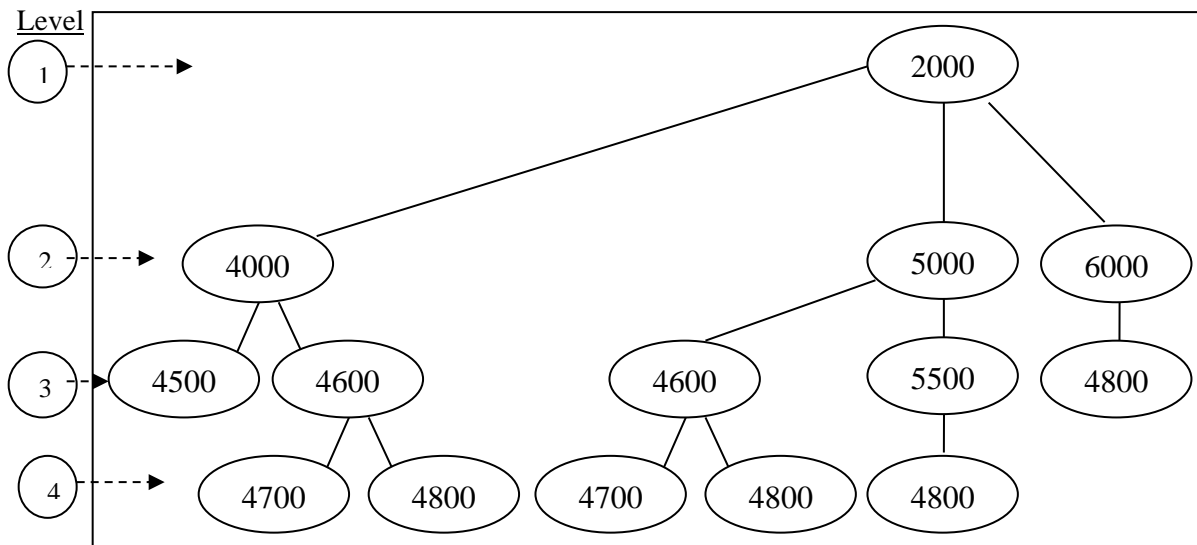


Figure 30.5e: Same as Figure 30.5d plus Level Numbers

**Exercise:**

30S1. Reference the REPORTS\_TO and REMPLOYEE2 tables. Apply the Soutlion-1 code-pattern to display the ENO, ENAME, SALARY, and SENO values for Employee 5000 and all employees who directly or indirectly work for her. The result should look like:

ENO	ENAME	SALARY	SENO
5000	JESSIE	400.00	2000
4600	ELEANOR	3000.00	5000
5500	HANNAH	4000.00	5000
4700	ANDY	2000.00	4600
4800	MATT	3000.00	4600
4800	MATT	3000.00	5500

The following Solution-2 creates two CTEs. (You might want to review Sample Query 27.4 which introduces the specification of two CTEs within a single WITH-clause.) The first CTE, called FULLTAB, pre-joins the REEMPLOYEE2 and REPORTS\_TO tables. The second CTE, DESCENDANTS, references FULLTAB to implement the recursive logic.

**Solution-2** (One "Early-Join" involving REEMPLOYEE2)

```
WITH
FULLTAB (ENO, ENAME, SALARY, SENO)
AS
  (SELECT RT.ENO, R2.ENAME, R2.SALARY, RT.SENO
   FROM REPORTS_TO RT, REEMPLOYEE2 R2
   WHERE RT.ENO = R2.ENO),

DESCENDANTS (ENO, ENAME, SALARY, SENO)
AS
  (SELECT ENO, ENAME, SALARY, SENO
   FROM FULLTAB
   WHERE ENO = '2000'
  UNION ALL
   SELECT F.ENO, F.ENAME, F.SALARY, F.SENO
   FROM   DESCENDANTS D, FULLTAB F
   WHERE  D.ENO = F.SENO
  )
SELECT * FROM DESCENDANTS
```

**Syntax & Logic:** Nothing new. Like the Solution-1, this solution allows us to code a very simple Main-SELECT to display DESCENDANTS which contains all the desired data.

**Observation:** FULLTAB has the same columns as REEMPLOYEE. Hence, the recursive code used to populate the DESCENDANTS table has a similar code-pattern for traversing a tree. (Review Sample Query 30.1.)

**Exercise:**

30S2. Apply the Solution-2 code-pattern to code an equivalent solution for Exercise 30S1.



The following Solution-3 delays joining the REMPLOYEE2 table until the Main-SELECT.

**Solution-3** ("Late-Join" involving REMPLOYEE2)

```
WITH DESCENDANTS (ENO, SENO)
AS
(SELECT   ENO, SENO
 FROM     REPORTS_TO
 WHERE    ENO = '2000'
  UNION ALL
 SELECT   R.ENO, R.SENO
 FROM     DESCENDANTS D, REPORTS_TO R
 WHERE    D.ENO = R.SENO
 )
SELECT   D.ENO, E.ENAME, E.SALARY, D.SENO
FROM     DESCENDANTS D, REMPLOYEE2 E
WHERE    D.ENO = E.ENO
```

**Syntax & Logic:** Nothing new. The initialization and recursive Sub-SELECTs do not need to access REMPLOYEE2 because the ENAME and SALARY values do not participate in the recursive logic. After DESCENDANTS is populated with the desired (ENO, SENO) values, the third Sub-SELECT joins DESCENDANTS with REMPLOYEE2 to access related ENAME and SALARY values.

**Best Solution?** Solution-3 appears to be the simplest solution. However, the join-operation in the Main-SELECT could undo a desired hierarchical sequence.

You might prefer the code-pattern in Solution-2 if you want your result to be in hierarchical sequence, and your system does not support some code like the previously described SEARCH-ORDER BY clauses. Solution-1, which specifies two early-joins, appears to be the least desirable code-pattern.

For tutorial purposes, future sample queries will not favor any particular code-pattern.

**Exercise:**

30S3. Apply the Solution-3 code-pattern to code another equivalent solution for Exercise 30S1.

## Duplicate Rows in the Result Table

We present another network design to discuss duplicate rows in a result table. The following RDEMO2 and RDEMO2MM tables (all integer columns) and corresponding network and tree diagrams represent a recursive many-to-many design.

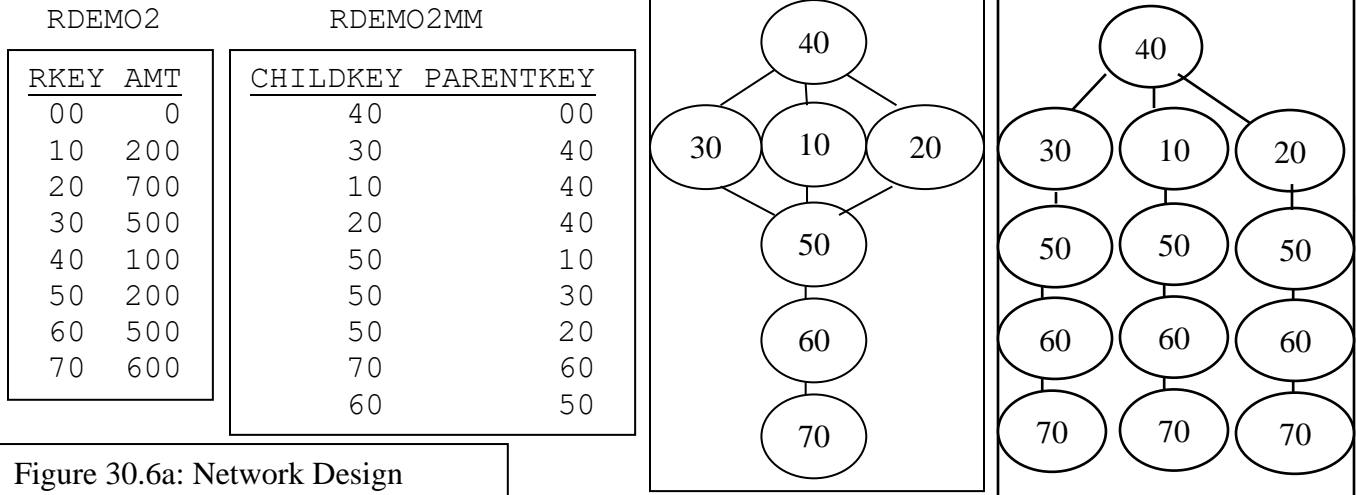


Figure 30.6a: Network Design

Note that the RDEMO2MM rows are not listed in any specific row sequence. (Rows were inserted into RDEMO2MM in the above sequence. However, execution of "SELECT \* FROM RDEMO2MM" might display these rows in a different sequence.) The following statement traverses down the network starting at Node-40. Important observations are presented on the following page.

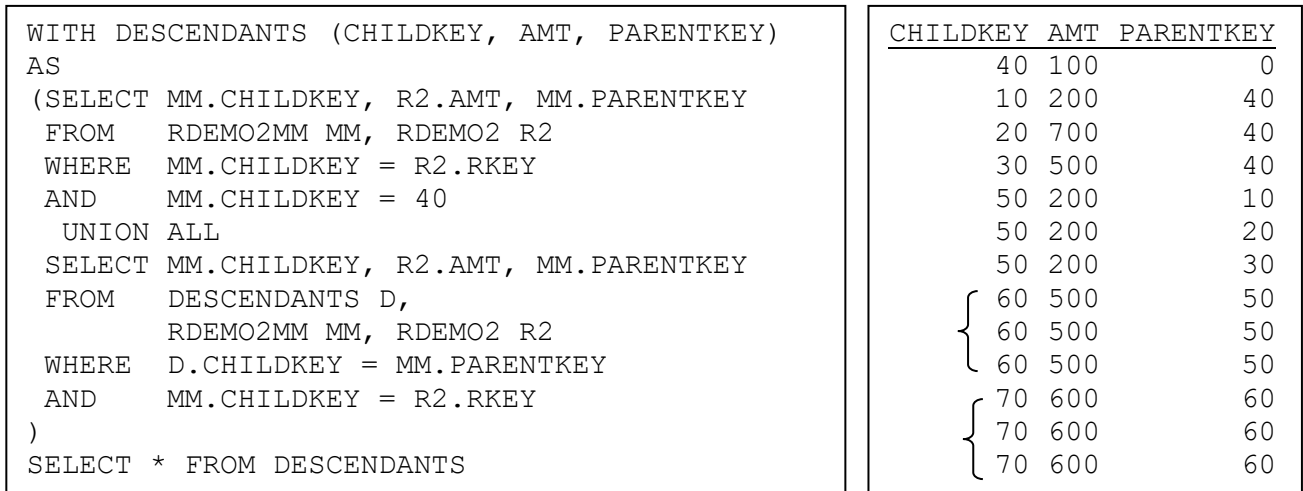


Figure 30.6b: Duplicate Rows in Result

**Duplicate Rows:** The tree diagram in Figure 30.6a illustrates redundant nodes. Hence, traversing a network, unlike a tree, can produce duplicate rows in a result table. In Figure 30.6b, duplicate rows appear for CHILDKEY values of 60 and 70. (Rows with CHILDKEY of 50 are not duplicates because these rows have different PARENTKEY values.)

**DISTINCT Eliminates Duplicate Rows:** We have already noted that specification of DISTINCT may indirectly cause the loss of hierarchical sequence. This can happen with network traversal. The following Figure 30.6c specifies DISTINCT in the Main-SELECT which removes duplicate rows from a result table. Notice than an incidental sort was based on the PARENTKEY column. This incidental sort undid the hierarchical sequence.

```
WITH DESCENDANTS (CHILDKEY, AMT, PARENTKEY)
AS
(SELECT MM.CHILDKEY, R2.AMT, MM.PARENTKEY
 FROM  RDEMO2MM MM, RDEMO2 R2
 WHERE MM.CHILDKEY = R2.RKEY
 AND   MM.CHILDKEY = 40
 UNION ALL
 SELECT MM.CHILDKEY, R2.AMT, MM.PARENTKEY
 FROM  DESCENDANTS D,
       RDEMO2MM MM, RDEMO2 R2
 WHERE D.CHILDKEY = MM.PARENTKEY
 AND   MM.CHILDKEY = R2.RKEY
 )
SELECT DISTINCT * FROM DESCENDANTS
```

CHILDKEY	AMT	PARENTKEY
40	100	0
50	200	10
50	200	20
50	200	30
10	200	40
30	500	40
20	700	40
60	500	50
70	600	60

Figure 30.6c: DISTINCT Removes Duplicate Rows (with Side-Effect)

**Another (Incidental) Side-Effect:** As expected, the result table in Figure 30.6b shows a breadth-first hierarchical sequence. Now, observe the second-level nodes in the network/tree diagrams. These nodes show a left-to-right node sequence of 30-10-20 which corresponds to the row sequence of CHILDKEY values in the RDEMO2MM table. However, the corresponding rows in this result table appear in ascending 10-20-30 sequence. (Techie Observation: Blame the optimizer.)

## Grouping and Counting Duplicate Rows

Another way to avoid row duplication is to group and count duplicate rows as illustrated in the following Figure 30.6d. This is done by specifying a GROUP BY clause and the COUNT(\*) function in the Main-SELECT.

```
WITH DESCENDANTS (CHILDKEY, AMT, PARENTKEY)
AS
(SELECT MM.CHILDKEY, R2.AMT, MM.PARENTKEY
 FROM   RDEMO2MM MM, RDEMO2 R2
 WHERE  MM.CHILDKEY = R2.RKEY
 AND    MM.CHILDKEY = 40
 UNION ALL
 SELECT MM.CHILDKEY, R2.AMT, MM.PARENTKEY
 FROM   DESCENDANTS D,
        RDEMO2MM MM, RDEMO2 R2
 WHERE  D.CHILDKEY = MM.PARENTKEY
 AND    MM.CHILDKEY = R2.RKEY
 )
SELECT CHILDKEY, AMT, PARENTKEY,
       COUNT(*) CNT
FROM   DESCENDANTS
GROUP BY CHILDKEY, AMT, PARENTKEY
```

CHILDKEY	AMT	PARENTKEY	CNT
40	100	0	1
50	200	10	1
50	200	20	1
50	200	30	1
10	200	40	1
30	500	40	1
60	500	50	3
70	600	60	3
20	700	40	1

Figure 30.6d: GROUP BY Removes Duplicate Rows (with Side-Effect)

An internal operation (for grouping) disrupted the hierarchical sequence. Furthermore, the absence of an ORDER BY clause in the Main-SELECT means that the final row sequence is unpredictable.

**Code Limitations within Recursive Sub-SELECT:** Previous examples have shown that specifying GROUP BY, join-operations, and ORDER BY (without SEARCH) in the Main-SELECT can undo a hierarchical sequence. This issue (most likely) is one reason why your SQL reference manual will describe some limitations on coding recursive Sub-SELECTs. For example, ORACLE forbids the specification of DISTINCT, ORDER BY, GROUP BY, and aggregate functions (e.g., SUM, AVG, MAX, and MIN) within a recursive Sub-SELECT.

## Exercises

30T. Code three SELECT-statements to satisfy the following query objective. Each statement should be similar in structure to those statements presented in Solution-1, Solution-2, and Solution-3 for Sample Query 30.11.

Reference the RDEMO2 and RDEMO2MM tables. Display the CHILDKEY, AMT, and PARENTKEY values for CHILDKEY 10 and all its descendants. The result should look like:

<u>CHILDKEY</u>	<u>AMT</u>	<u>PARENTKEY</u>
10	200	40
50	200	10
60	500	50
70	600	60

30U. Modify Sample Query 30.11 to remove duplicate rows from the result by grouping and counting the number of duplicate rows. Show this count value in the CNT column. The result will contain the following rows, but these rows might appear in a different sequence.

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>SENO</u>	<u>CNT</u>
2000	JANET	2000.00	1000	1
5000	JESSIE	400.00	2000	1
4000	JULIE	500.00	2000	1
6000	FRANK	9000.00	2000	1
4500	JOHNNY	2000.00	4000	1
4600	ELEANOR	3000.00	4000	1
4700	ANDY	2000.00	4600	2
4800	MATT	3000.00	4600	2
4600	ELEANOR	3000.00	5000	1
5500	HANNAH	4000.00	5000	1
4800	MATT	3000.00	5500	1
4800	MATT	3000.00	6000	1

## Restriction in Recursive Sub-SELECT

The following sample query requires a restriction in the recursive Sub-SELECT.

**Sample Query 30.12:** Display ENO, ENAME, SALARY, and SENO values for Employee 2000 and all her descendants. However, if an employee's salary is less than \$500.00, then exclude all descendants of this employee.

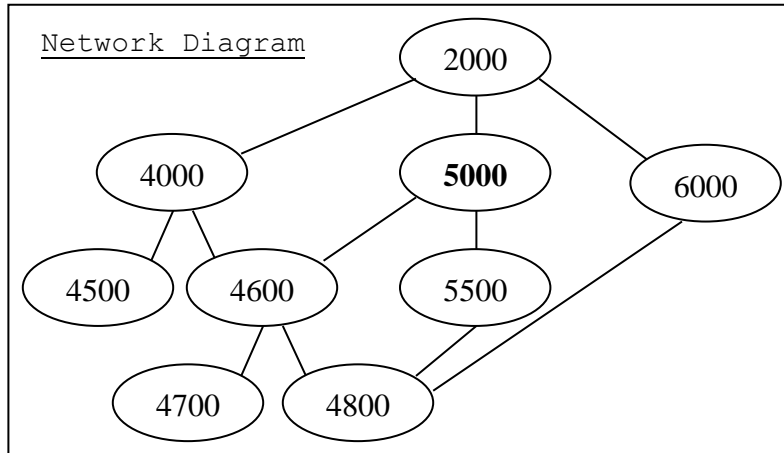
```
WITH
FULLTAB (ENO, ENAME, SALARY, SENO) AS
  (SELECT RT.ENO, R2.ENAME, R2.SALARY, RT.SENO
   FROM REPORTS_TO RT, REMPLOYEE2 R2
   WHERE RT.ENO = R2.ENO),
DESCENDANTS (ENO, ENAME, SALARY, SENO) AS
  (SELECT ENO, ENAME, SALARY, SENO
   FROM FULLTAB
   WHERE ENO = '2000'
   UNION ALL
   SELECT F.ENO, F.ENAME, F.SALARY, F.SENO
   FROM DESCENDANTS D, FULLTAB F
   WHERE D.ENO = F.SENO
   AND D.SALARY >= 500.00 ←
  )
SELECT * FROM DESCENDANTS
```

ENO	ENAME	SALARY	SENO
2000	JANET	2000.00	1000
4000	JULIE	500.00	2000
5000	JESSIE	400.00	2000
6000	FRANK	9000.00	2000
4500	JOHNNY	2000.00	4000
4600	ELEANOR	3000.00	4000
4800	MATT	3000.00	6000
4700	ANDY	2000.00	4600
4800	MATT	3000.00	4600

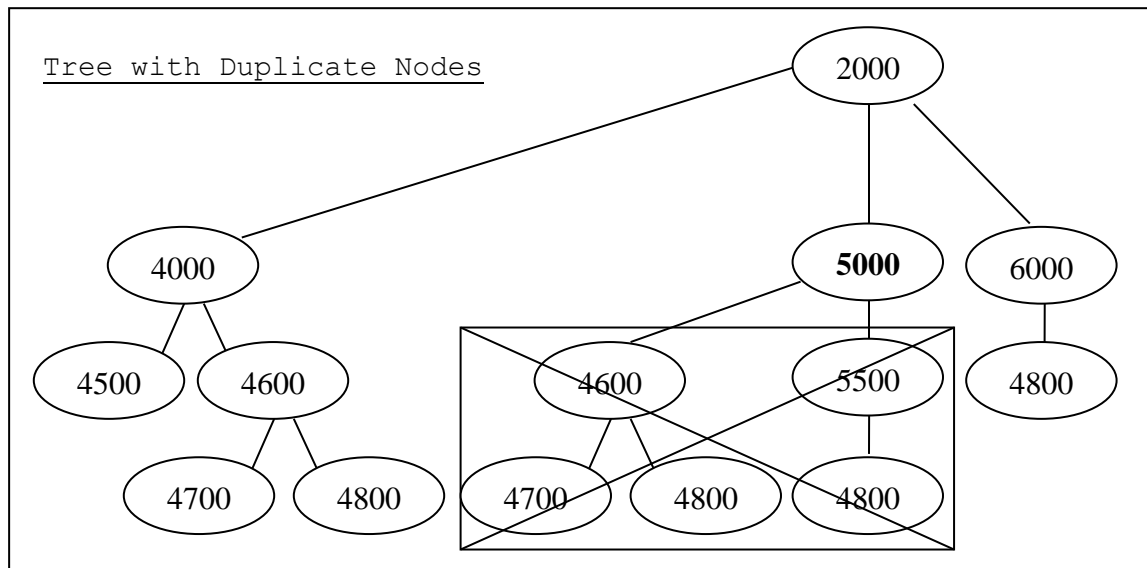
**Syntax and Logic:** This statement requires an "early-join" because the D.SALARY>=500.00 condition requires access to the SALARY column which resides in the REMPLOYEE2 table.

Employee 5000 fails the D.SALARY >= 500.00 condition. This condition is specified in the recursive Sub-SELECT because the query objective requires "trimming the branches" of the sub-tree corresponding to the descendants of Employee 5000.

**Network and Tree Diagrams:** Assume the recursive Sub-SELECT did not include the D.SALARY >= 500.00 condition. Then, the following network diagram would represent the result.



After transforming this network into a tree and eliminating descendants of Node-5000 (shown in the crossed-out box), the following tree represents the final result.



**Three Observations:** (1) Node-4600 appears twice in this tree. However, only one of these nodes is "crossed out." Hence, only one row for Employee 4600 appears in the final result. (2) Node-4700 appears twice in this tree. However, only one of these nodes is crossed out. Hence, only one row for Employee 4700 appears in the final result. (3) Node-4800 appears four times in this tree. However, two of these nodes are crossed out. Hence, only two rows for Employee 4800 appear in the final result.

## Traverse Up a Network

The following sample query traverses up a network. This sample query is similar to Sample Query 30.5 which traversed up a tree.

**Sample Query 30.13:** Start with Employee 4600. Display the ENO, ENAME, and SENO values for this employee and all of her direct and indirect supervisors (ancestors).

```
WITH ANCESTORS (ENO, ENAME, SENO)
AS
(SELECT R.ENO, R2.ENAME, R.SENO
 FROM   REPORTS_TO R, REMPLOYEE2 R2
 WHERE  R.ENO = R2.ENO
 AND    R.ENO = '4600'
 UNION ALL
 SELECT R.ENO, R2.ENAME, R.SENO
 FROM   ANCESTORS A, REPORTS_TO R, REMPLOYEE2 R2
 WHERE  R.ENO = R2.ENO
 AND    A.SENO = R.ENO
 )
SELECT * FROM ANCESTORS
```

ENO	ENAME	SENO
4600	ELEANOR	4000
4600	ELEANOR	5000
4000	JULIE	2000
5000	JESSIE	2000
2000	JANET	1000
2000	JANET	1000
1000	MOE	0000
1000	MOE	0000

**Syntax and Logic:** Nothing New. The join-condition in the recursive Sub-SELECT dictates upward traversal. Recall that a single node may have multiple parents. Hence, duplicate rows appear in the result because the same ancestor nodes appear in the two upward paths from Node-4600 to Node-1000.

Path-1: Node-4600 → Node-4000 → Node-2000 → Node-1000

Path-2: Node-4600 → Node-5000 → Node-2000 → Node-1000

Again, you might consider specifying DISTINCT in the Main-SELECT to remove duplicate rows.



## Exercises

30V. Reference RDEMO2 and RDEMO2MM. Display the CHILDKEY, AMT, and PARENTKEY values for Node-30 and its descendants. However, if a descendant's AMT is greater than or equal to 600, then exclude all descendants of this descendant. The result should look like:

<u>CHILDKEY</u>	<u>AMT</u>	<u>PARENTKEY</u>
30	500	40
50	200	30
60	500	50

30W. Reference the RDEMO2 and RDEMO2MM tables. Start with Node-60. Display the CHILDKEY, AMT, and PARENTKEY values for this node and all of its direct and indirect ancestors. The result should look like:

<u>CHILDKEY</u>	<u>AMT</u>	<u>PARENTKEY</u>
60	500	50
50	200	10
50	200	20
50	200	30
10	200	40
20	700	40
30	500	40
40	100	0
40	100	0
40	100	0

30X. Reference the REPORTS\_TO and REMPLOYEE2 tables. Display the ENO, ENAME, and SENO values for Employee 4000 and all his direct or indirect supervisors. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
4000	JULIE	2000
2000	JANET	1000
1000	MOE	0000

## Display Level Numbers: Traverse Down a Network

**Sample Query 30.14:** Extend Sample Query 30.11. Display the ENO, ENAME, SALARY, and SENO values for Employee 2000 and all employees who directly or indirectly work for her. Also display level numbers.

```
WITH DESCENDANTS (LVL, ENO, ENAME, SALARY, SENO)
AS
(SELECT   1, R.ENO, R2.ENAME, R2.SALARY, R.SENO
 FROM     REPORTS_TO R, REEMPLOYEE2 R2
 WHERE    R.ENO = R2.ENO
 AND      R.ENO = '2000'
 UNION ALL
 SELECT   LVL+1, R.ENO, R2.ENAME, R2.SALARY, R.SENO
 FROM     DESCENDANTS D, REPORTS_TO R, REEMPLOYEE2 R2
 WHERE    R.ENO = R2.ENO AND D.ENO = R.SENO
 )
SELECT * FROM DESCENDANTS
```

<u>LVL</u>	<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>SENO</u>
1	2000	JANET	2000.00	1000
2	4000	JULIE	500.00	2000
2	5000	JESSIE	400.00	2000
2	6000	FRANK	9000.00	2000
3	4500	JOHNNY	2000.00	4000
3	4600	ELEANOR	3000.00	4000
3	4600	ELEANOR	3000.00	5000
3	5500	HANNAH	4000.00	5000
3	4800	MATT	3000.00	6000
4	4700	ANDY	2000.00	4600
4	4800	MATT	3000.00	4600
4	4700	ANDY	2000.00	4600
4	4800	MATT	3000.00	4600
4	4800	MATT	3000.00	5500

**Syntax & Logic:** Nothing New. Observe that data about Employee 4800 appears in three rows with an LVL value of 4 and in one row with an LVL value of 3. This is illustrated in Figures 30.5c, 30.5d, and 30.5e.

### Exercise:

30Y1. Modify the above Sample Query 30.14 to limit the downward traversal to three levels. (Hint: Review Sample Query 30.7b.)

## Display Level Numbers: Traverse Up a Network

**Sample Query 30.15:** Extend Sample Query 30.13. Start with Employee 4600. Display the ENO, ENAME, SALARY, and SENO values for this employee and all her direct or indirect supervisors. Also display level numbers.

```
WITH ANCESTORS (LVL, ENO, ENAME, SENO)
AS
(SELECT 1, R.ENO, R2.ENAME, R.SENO
 FROM   REPORTS_TO R, REEMPLOYEE2 R2
 WHERE  R.ENO = R2.ENO
 AND    R.ENO = '4600'
 UNION ALL
 SELECT LVL+1, R.ENO, R2.ENAME, R.SENO
 FROM   ANCESTORS A, REPORTS_TO R, REEMPLOYEE2 R2
 WHERE  R.ENO = R2.ENO
 AND    A.SENO = R.ENO
 )
SELECT * FROM ANCESTORS
```

<u>LVL</u>	<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>SENO</u>
1	4600	ELEANOR	3000.00	4000
1	4600	ELEANOR	3000.00	5000
2	4000	JULIE	500.00	2000
2	5000	JESSIE	400.00	2000
3	2000	JANET	2000.00	1000
3	2000	JANET	2000.00	1000
4	1000	MOE	2000.00	0000
4	1000	MOE	2000.00	0000

**Syntax & Logic:** Nothing New.

### Exercise:

30Y2. Modify the above Sample Query 30.15 to (i) limit the upward traversal to three levels, (ii) remove duplicate rows from the result table, and (iii) modify the level numbers such that the result looks like:

<u>LVL</u>	<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>SENO</u>
2	2000	JANET	2000.00	1000
3	5000	JESSIE	400.00	2000
3	4000	JULIE	500.00	2000
4	4600	ELEANOR	3000.00	4000
4	4600	ELEANOR	3000.00	5000

## Variations of Many-to-Many Recursive Designs

Within the context of an “employees-report-to-employees” application, we have utilized the following design to represent a recursive many-to-many relationship.

REMPLOYEE2	<u>ENO</u>	ENAME	SALARY	REPORTS_TO	<u>ENO</u>	<u>SENO</u>
------------	------------	-------	--------	------------	------------	-------------

Below we present three similar design scenarios that require modifications to this design.

### Design Scenario-1: Intersection Data within Recursive Table

Assume employees are not paid a fixed salary. Instead, each employee may work a different number of hours and be paid a different hourly pay-rate by each supervisor. To represent this change, the REMPLOYEE3 table is formed by removing SALARY from REMPLOYEE2; and the REPORTS\_TO2 table is formed by extending REPORTS\_TO to include the HOURS and PAYRATE columns (the intersection data).

REMPLOYEE3	<u>ENO</u>	ENAME	REPORTS_TO2	<u>ENO</u>	<u>SENO</u>	HOURS	PAYRATE
------------	------------	-------	-------------	------------	-------------	-------	---------

Sample data for these tables are shown below.

<b>Figure 30.7:</b> <b>REMPLOYEE3 and</b> <b>REPORTS_TO2 Tables</b>	REMPLOYEE3		REPORTS_TO2			
	<u>ENO</u>	<u>ENAME</u>	<u>ENO</u>	<u>SENO</u>	HOURS	PAYRATE
	0000	DUMMY	1000	0000	0	0
	1000	MOE	2000	1000	40	80
	2000	JANET	3000	1000	40	70
	3000	LARRY	8000	1000	40	80
	8000	JOE	4000	2000	40	60
	4000	JULIE	5000	2000	40	60
	5000	JESSIE	6000	2000	40	70
	6000	FRANK	6500	3000	40	60
	6500	CURLY	8500	8000	20	90
	8500	GEORGE	4500	4000	40	60
	4500	JOHNNY	4600	4000	20	50
	4600	ELEANOR	5500	5000	40	50
	5500	HANNAH	7500	6500	30	50
7500	SHEMP	8600	8500	40	40	
8600	DICK	8700	8500	40	50	
8700	HANK	4700	4600	20	40	
4700	ANDY	4800	4600	10	50	
4800	MATT	4600	5000	20	50	
		4800	5500	20	20	
		4800	6000	10	60	

This design does not require any significant coding adjustments. Consider the following query objective that is similar to Sample Query 30.11. It displays HOURS and PAYRATE values instead of SALARY values.

**Sample Query 30.16:** Reference the REMPLOYEE3 and REPORTS\_TO2 tables. For Employee 2000 and all her descendants, display each employee's ENO and ENAME values plus the supervisor's SENO value and corresponding HOURS and PAYRATE values.

```

WITH DESCENDANTS (ENO, ENAME, SENO, HOURS, PAYRATE)
AS
(SELECT  REMP3.ENO, REMP3.ENAME,
        RT2.SENO, RT2.HOURS, RT2.PAYRATE
FROM    REPORTS_TO2 RT2, REMPLOYEE3 REMP3
WHERE   REMP3.ENO = RT2.ENO AND REMP3.ENO = '2000'
UNION ALL
SELECT  REMP3.ENO, REMP3.ENAME,
        RT2.SENO, RT2.HOURS, RT2.PAYRATE
FROM    DESCENDANTS D, REPORTS_TO2 RT2, REMPLOYEE3 REMP3
WHERE   REMP3.ENO = RT2.ENO AND D.ENO = RT2.SENO
)
SELECT * FROM DESCENDANTS

```

ENO	ENAME	SENO	HOURS	PAYRATE
2000	JANET	1000	40	80
4000	JULIE	2000	40	60
5000	JESSIE	2000	40	60
6000	FRANK	2000	40	70
4500	JOHNNY	4000	40	60
4600	ELEANOR	4000	20	50
5500	HANNAH	5000	40	50
4600	ELEANOR	5000	20	50
4800	MATT	6000	10	60
4700	ANDY	4600	20	40
4800	MATT	4600	10	50
4800	MATT	5500	20	20
4700	ANDY	4600	20	40
4800	MATT	4600	10	50

**Syntax and Logic:** Nothing new.

**Another very simple design scenario:** Assume business users only care about intersection data (HOURS and PAYRATE) and have no interest in ENAME values. Then, the DBA would not need to create the REMPLOYEE3 table.

## Design Scenario-2: A "Good" Parent-Oriented Design

In Section-A, we discouraged creating a parent-oriented recursive table for a recursive one-to-many relationship. However, the following Figure 30.8 illustrates that a *parent-oriented design can be reasonable for a many-to-many recursive relationship*.

Reconsider the REPORTS\_TO table. *In principle, this table is neither child-oriented nor parent-oriented*. However, we adopted a child-oriented perspective of REPORTS\_TO by reading its column names from left-to-right: "ENO (child) REPORTS\_TO SENO (parent)." This table could be replaced by the following SUPERVISES table which can be seen as parent-oriented by reading its column names from left-to-right: "SENO (parent) supervises ENO (child)."

REPORTS_TO		SUPERVISES	
<u>ENO</u>	<u>SENO</u>	<u>SENO</u>	<u>ENO</u>
1000	0000	0000	1000
2000	1000	1000	2000
3000	1000	1000	2000
8000	1000	1000	8000
4000	2000	2000	4000
5000	2000	2000	5000
6000	2000	2000	6000
6500	3000	3000	6500
8500	8000	8000	8500
4500	4000	4000	4500
4600	4000	4000	4600
5500	5000	5000	5500
7500	6500	6500	7500
8600	8500	8500	8600
8700	8500	8500	8700
4700	4600	4600	4700
4800	4600	4600	4800
4600	5000	5000	4600
4800	5500	5500	4800
4800	6000	6000	4800

Figure 30.8: Logically Equivalent Tables

The REPORTS\_TO and SUPERVISES tables are logically equivalent tables. They represent the same semantic concepts. (The SUPERVISES table merely swaps the left-to-right column sequence in REPORT\_TO.) A database designer could create either table depending upon her mindset.

Assume a designer created SUPERVISES instead of REPORTS\_TO. This design change would not have any significant impact on coding recursive queries. Consider the following sample query.

**Sample Query 30.17:** Same as Sample Query 30.11. Reference the SUPERVISES and REMPLOYEE2 tables. Display the ENO, ENAME, SALARY, and SENO values for Employee 2000 and all employees who directly or indirectly work for her.

```

WITH DESCENDANTS (ENO, SENO)
AS
(SELECT   ENO, SENO
 FROM     SUPERVISES
 WHERE    ENO = '2000'
 UNION ALL
 SELECT   R.ENO, R.SENO
 FROM     DESCENDANTS D, SUPERVISES R
 WHERE    D.ENO = R.SENO
 )
SELECT   D.ENO, E.ENAME, E.SALARY, D.SENO
FROM     DESCENDANTS D, REMPLOYEE2 E
WHERE    D.ENO = E.ENO

```

ENO	ENAME	SALARY	SENO
2000	JANET	2000.00	1000
4000	JULIE	500.00	2000
5000	JESSIE	400.00	2000
6000	FRANK	9000.00	2000
4500	JOHNNY	2000.00	4000
4600	ELEANOR	3000.00	4000
4600	ELEANOR	3000.00	5000
5500	HANNAH	4000.00	5000
4800	MATT	3000.00	6000
4700	ANDY	2000.00	4600
4800	MATT	3000.00	4600
4700	ANDY	2000.00	4600
4800	MATT	3000.00	4600
4800	MATT	3000.00	5500

**Syntax and Logic:** Nothing new. This statement merely substitutes "SUPERVISES" for "REPORTS\_TO" in Solution-3 for Sample Query 30.11.

### Design Scenario-3: Cyclic Network

The previous two scenarios did not make recursive queries any more complex. However, this third scenario is another story. Reconsider our original many-to-many recursive design.

EMPLOYEE2 

<u>ENO</u>	ENAME	SALARY
------------	-------	--------

      REPORTS\_TO 

<u>ENO</u>	<u>SENO</u>
------------	-------------

This third scenario does not make any structural changes to the above design (same tables with same columns.) However, *it removes the following important design constraint that we have implicitly assumed throughout this chapter.*

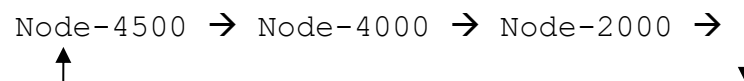
**Previous Design Constraint - Acyclic Network:** The database designer specifies an integrity constraint for a recursive table (e.g., REPORTS\_TO) such that this table represents an *acyclic* network. This means that you can never start at some node, follow some path, and return to the same starting node. All previously described network diagrams have been acyclic.

**Eliminate Design Constraint - Cyclic Network:** Within a cyclic network, it is possible to start at some node, follow some path, and return to the same starting node. We present two examples of a cyclic network.

**Unrealistic Example of a Cyclic Network:** If the system did not enforce the acyclic constraint on the REPORTS\_TO table, then the following row could be inserted into this table.

<u>ENO</u>	<u>SENO</u>
2000	4500

Inserting this row makes it possible to traverse the following path.



This path forms a cycle (also called a "loop"). This cycle implies that Employees 4500, 4000, and 2000 indirectly report to themselves, which is why we say this is an unrealistic example.

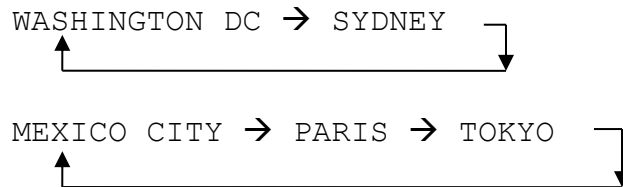


**Realistic Example of Cyclic Network:** Assume the designer created a table called FLIGHTS (shown in following Figure 30.9a) where each row describes an airline flight from an origin city (STARTCITY) to a destination city (STOPCITY). This table stores the PRICE of each flight and its travel time in HOURS. In this application, it is reasonable to have a flight from City-1 to City-2 and another flight from City-2 to City-1. Or, there could be intermediate flights that allow a passenger to *indirectly* travel from a given city and return to the same city. Consider the sample data in the following FLIGHTS table.

FLIGHTS				
STARTCITY	STOPCITY	PRICE	HOURS	
WASHINGTON DC	SYDNEY	2500.00	16.00	
WASHINGTON DC	MEXICO CITY	1000.00	6.50	
LONDON	PARIS	500.00	1.25	
PARIS	TOKYO	1200.00	9.25	
LONDON	TOKYO	2000.00	10.50	
MEXICO CITY	PARIS	1500.00	9.00	
TOKYO	MEXICO CITY	2000.00	17.00	
SYDNEY	WASHINGTON DC	2600.00	18.00	

Figure 30.9a: Cyclic Table

Inspection of the FLIGHTS table shows two cycles.



The presence of any cycle means that the network is cyclic. The network diagram in the following Figure 30.9b makes it easier to detect the cycles.

## “No Top – No Bottom”

Previous tree and network diagrams without cycles presumed a top-to-bottom orientation. In those diagrams, we illustrated supervisor-nodes “above” supervisee-nodes. This top-to-bottom orientation does not apply within a cyclic network. Consider the following network diagram for the FLIGHTS table.

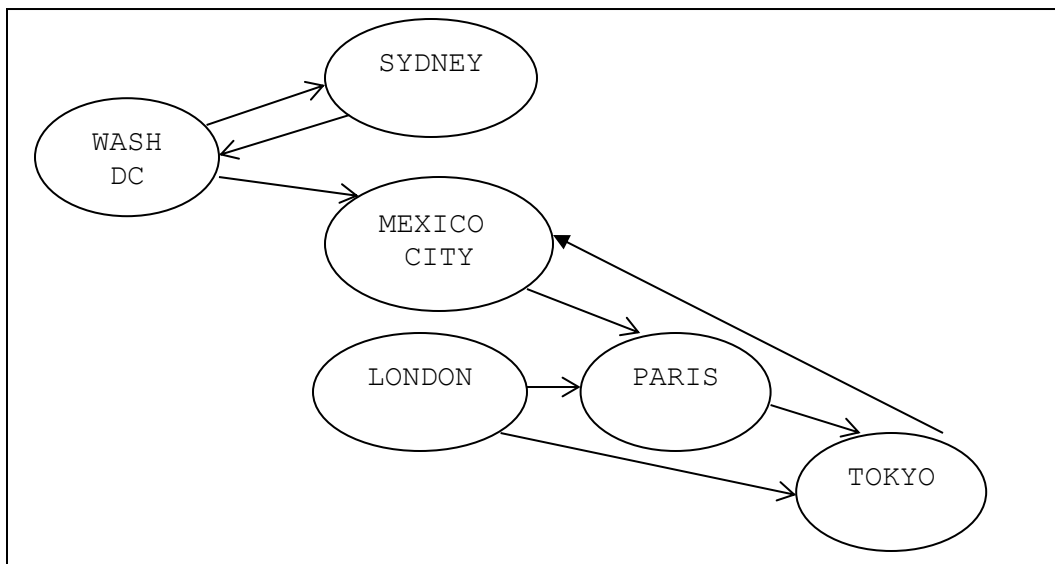


Figure 30.9b: Cyclic Network

Examination of the above network diagram shows that there are no absolute top (root) nodes or bottom (leaf) nodes. For example, another valid illustration of this network could show the SYDNEY Node below the WASH DC Node.

**Bad News – Possible “Infinite” Traversal:** Recursive queries against a cyclic table become more complex because your code must detect and prohibit the “infinite” traversal of a cycle. Some systems do not offer a direct method to satisfy this objective. (One exception is ORACLE which supports a CYCLE-SET clause. Again, consult your SQL reference manual for help on this matter.) We do not present an example of this cyclic-detection-prevention code because it is usually specified by procedural logic within a stored procedure or application program.

## C. Recursive Queries without Recursive CTEs

The following Figures 30.10a and 30.10b show result tables that are produced by queries against the `REMPLOYEE` table. Both result tables display data about a supervisor (Employee 2000) and her immediate supervisees. The “horizontal” format of Figure 30.10b differs from Figure 30.10a and previous result tables shown in this chapter. Here the `PENO` and `PENAME` columns contain a parent’s number and name, and the `CENO` and `CENAME` columns contain the numbers and names of children. Because the corresponding parents and children appear in the same row, there is no need to display `SENO` values.

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
2000	JANET	1000
4000	JULIE	2000
5000	JESSIE	2000
6000	FRANK	2000

Figure 30.10a: Two-level Vertical

<u>PNO</u>	<u>PENAME</u>	<u>CENO</u>	<u>CENAME</u>
2000	JANET	4000	JULIE
2000	JANET	5000	JESSIE
2000	JANET	6000	FRANK

Figure 30.10b: Two-level Horizontal

The following Figures 30.11a and 30.11b extend the above result tables by displaying data from a three-level traversal of the `REMPLOYEE` table. The objective is to display data about a supervisor (Employee 2000), her direct supervisees (children), and their direct supervisees (grandchildren). In Figure 30.11b, the `GCENO` and `GCENAME` columns contain the numbers and names of the grandchildren.

<u>ENO</u>	<u>ENAME</u>	<u>SENO</u>
2000	JANET	1000
4000	JULIE	2000
5000	JESSIE	2000
6000	FRANK	2000
4500	JOHNNY	4000
4600	ELEANOR	4000
5500	HANNAH	5000

Figure 30.11a: Three-level Vertical

<u>PENO</u>	<u>PENAME</u>	<u>CENO</u>	<u>CENAME</u>	<u>GCENO</u>	<u>GCENAME</u>
2000	JANET	4000	JULIE	4500	JOHNNY
2000	JANET	4000	JULIE	4600	ELEANOR
2000	JANET	5000	JESSIE	5500	HANNAH
2000	JANET	6000	FRANK	-	-

Figure 30.11b: Three-level Horizontal

Some users will prefer result tables with a horizontal format that look like Figures 30.10b and 30.11b. The following sample queries produce similar result tables by coding `SELECT` statements *that do not require specifying recursive CTEs*.

The following pages present sample queries against the REMPLOYEE table where each SELECT statement will join the REMPLOYEE table with itself. (Review Sample Queries 17.10.1 - 17.10.3 which introduced joining a table with itself.) Each sample query illustrates a query-pattern that can be described according to three categories. Each category has two options, producing a total of eight (2x2x2) query-patterns. The three categories are described below.

### 1. Number of Levels

- Two-Level tree traversal
- Three-Level tree traversal

### 2. Column Orientation of Result Table

- Parent-Oriented (Parent-data in leftmost cols)

<u>PNO</u>	<u>PENAME</u>	<u>CENO</u>	<u>CENAME</u>
2000	JANET	4000	JULIE

- Child-Oriented (Child-data in leftmost cols)

<u>CENO</u>	<u>CENAME</u>	<u>PENO</u>	<u>PENAME</u>
4000	JULIE	2000	JANET

### 3. Matching Objective

- Non-Matching: The result table includes data from all selected parent-rows, including parents without children. For example, the row for Employee 8700, an employee without children, is included in the following result.

<u>PNO</u>	<u>PENAME</u>	<u>CENO</u>	<u>CENAME</u>
2000	JANET	4000	JULIE
8700	HANK	-	-

- Matching: The result table excludes rows for parents without children. For example, the following result does not include a row for Employee 8700.

<u>PNO</u>	<u>PENAME</u>	<u>CENO</u>	<u>CENAME</u>
2000	JANET	4000	JULIE

In preparation of the next eight sample queries, we review Figure 30.1c which illustrates the tree diagram for the REMPLOYEE table. This figure highlights (in bold font) four nodes (1000, 4700, 3000, and 8500). We make pertinent observations about these nodes below.

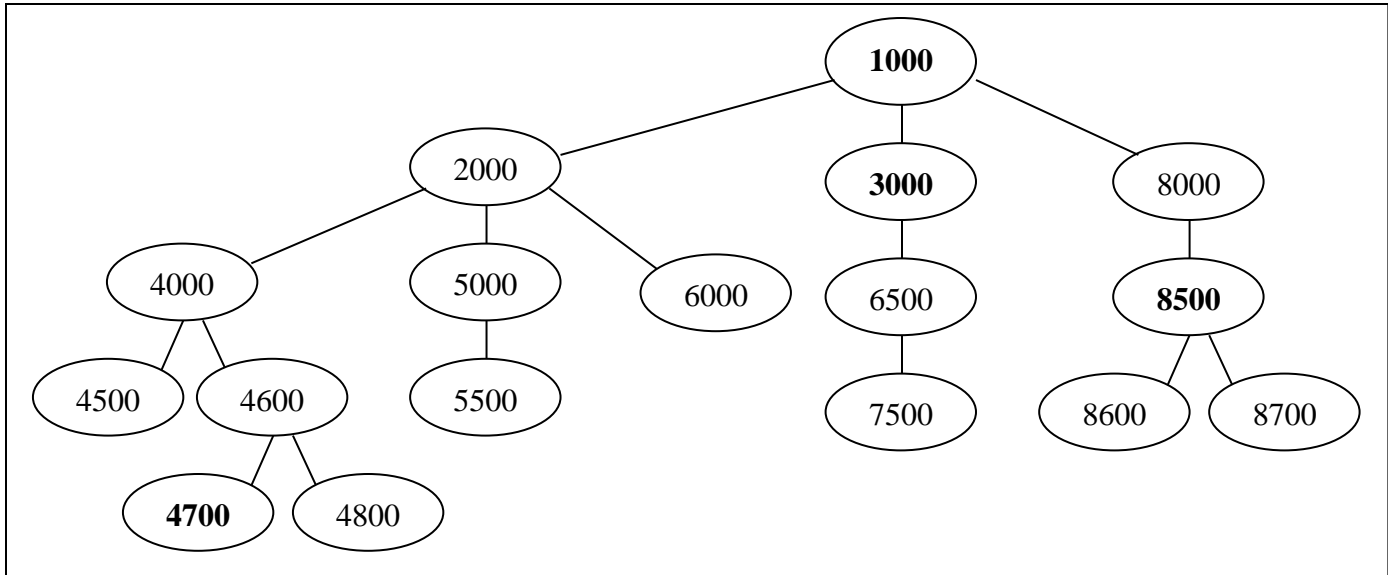


Figure 30.1c: Tree Diagram Represents REMPLOYEE Table

- Employee 1000: Root Node  
No parent node, hence no grandparent node
- Employee 4700: Leaf Node  
No child node, hence no grandchild nodes
- Employee 3000: Intermediate Node  
Parent node, but no grandparent node
- Employee 8500: Intermediate Node  
Child nodes, but no grandchild nodes

Sample Queries 30.18.1 - 30.18.4 will traverse two levels in this tree.

Sample Queries 30.19.1 - 30.19.4 will traverse three levels in this tree.

## Query-Pattern-1: Two-Level, Parent-Oriented, Matching

The next four sample queries display data about parents and children in the tree representing the REMPLOYEE table. These queries reference three employees: Employee 1000, the only node without a parent; Employee 4700, a leaf node without any children; and Employee 8500, a node with children.

The following sample query accesses two levels, is parent-oriented (i.e., data from the parent-table appears in the leftmost columns in the result table), and only displays matching rows for parents who have at least one child.

**Sample Query 30.18.1:** Consider Employees 1000, 8500, and 4700. If any of these employees is a supervisor, display his employee number, name, and salary followed by the number, name, and salary of each immediate supervisee. Sort the result by the supervisee's ENO value within the supervisor's ENO value.

```
SELECT PARENT.ENO BOSSENO,
       PARENT.ENAME BOSSENAME,
       PARENT.SALARY BOSSSALARY,
       CHILD.ENO,
       CHILD.ENAME,
       CHILD.SALARY

FROM   REMPLOYEE PARENT INNER JOIN REMPLOYEE CHILD
      ON PARENT.ENO = CHILD.SENO

AND    PARENT.ENO IN ('1000', '8500', '4700')

ORDER BY PARENT.ENO, CHILD.ENO
```

<u>BOSSENO</u>	<u>BOSSENAME</u>	<u>BOSSSALARY</u>	<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>
1000	MOE	2000.00	2000	JANET	2000.00
1000	MOE	2000.00	3000	LARRY	3000.00
1000	MOE	2000.00	8000	JOE	8000.00
8500	GEORGE	7000.00	8600	DICK	6000.00
8500	GEORGE	7000.00	8700	HANK	6000.00

**Syntax & Logic:** Nothing new. This result table shows three rows for Employee 1000 because he supervises three employees; and two rows for Employee 8500 because he supervises two employees. Employee 4700 does not appear in the result because he is not a supervisor. (His row corresponds to a leaf node.)

## Query-Pattern-2: Two-Level, Parent-Oriented, Non-Matching

The following sample query is similar to the preceding sample query. It accesses two levels and is parent-oriented. Unlike the preceding sample query, this sample query displays both matching and non-matching rows.

**Sample Query 30.18.2:** Consider Employees 1000, 8500, and 4700. Display the employee number, name, and salary of these employees. Also, if any of these employees is a supervisor, display the number, name, and salary of each immediate supervisee. Sort the result by the supervisee's ENO value within the supervisor's ENO value.

```
SELECT PARENT.ENO BOSSENO,
       PARENT.ENAME BOSSENAME,
       PARENT.SALARY BOSSSALARY,
       CHILD.ENO,
       CHILD.ENAME,
       CHILD.SALARY

FROM   REMPLOYEE PARENT LEFT OUTER JOIN REMPLOYEE CHILD
       ON PARENT.ENO = CHILD.SENO

WHERE  PARENT.ENO IN ('1000', '8500', '4700')

ORDER BY PARENT.ENO, CHILD.ENO
```

<u>BOSSENO</u>	<u>BOSSENAME</u>	<u>BOSSSALARY</u>	<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>
1000	MOE	2000.00	2000	JANET	2000.00
1000	MOE	2000.00	3000	LARRY	3000.00
1000	MOE	2000.00	8000	JOE	8000.00
4700	ANDY	2000.00	-	-	-
8500	GEORGE	7000.00	8600	DICK	6000.00
8500	GEORGE	7000.00	8700	HANK	6000.00

**Syntax & Logic:** Nothing new. Unlike the previous sample query that coded an INNER JOIN operation, this statement coded a LEFT OUTER JOIN with the parent-table as the left-table. This result is the same as the previous result table plus one more row for Employee 4700, an employee who is not a supervisor (corresponding to a leaf node).

### Query-Pattern-3: Two-Level, Child-Oriented, Matching

The following sample query accesses two levels, is child-oriented (i.e., data from the child-table appears in the leftmost columns in the result table), and displays matching rows corresponding to children with parents (which is every row except Employee 1000, the big-boss).

**Sample Query 30.18.3:** Consider Employees 1000, 8500, and 4700. For each employee who has a supervisor, display the employee's number, name, and salary followed by the number, name and salary of their immediate supervisor. Sort the result by the supervisee's ENO value.

```
SELECT CHILD.ENO,  
        CHILD.ENAME,  
        CHILD.SALARY,  
        PARENT.ENO BOSSENO,  
        PARENT.ENAME BOSSENAME,  
        PARENT.SALARY BOSSSALARY  
  
FROM    REMPLOYEE CHILD INNER JOIN REMPLOYEE PARENT  
        ON CHILD.SENO = PARENT.ENO  
  
AND     CHILD.ENO IN ('1000', '8500', '4700')  
  
ORDER BY CHILD.ENO
```

ENO	ENAME	SALARY	BOSSENO	BOSSENAME	BOSSSALARY
4700	ANDY	2000.00	4600	ELEANOR	3000.00
8500	GEORGE	7000.00	8000	JOE	8000.00

**Syntax & Logic:** Nothing new. Observe that the row for Employee 1000 does not appear in the result because this employee is the big-boss who does not have a supervisor. (His row corresponds to the root node.)

**Observation:** Examination of this result table shows that a "child-oriented" result is analogous to an upward tree traversal.



## Query-Pattern-4: Two-Level, Child-Oriented, Non-Matching

The following sample query is similar to the preceding sample query. It accesses two levels and is child-oriented. Unlike the preceding sample query, this sample query displays both matching and non-matching rows.

**Sample Query 30.18.4:** Consider Employees 1000, 8500, and 4700. Display each employee's number, name, and salary. Also, if the employee has a supervisor, display the number, name, and salary of the employee's immediate supervisor. Sort the result by the supervisee's ENO value.

```
SELECT CHILD.ENO,  
        CHILD.ENAME,  
        CHILD.SALARY,  
        PARENT.ENO BOSSENO,  
        PARENT.ENAME BOSSENAME,  
        PARENT.SALARY BOSSSALARY  
  
FROM    REMPLOYEE CHILD LEFT OUTER JOIN REMPLOYEE PARENT  
        ON CHILD.SENO = PARENT.ENO  
  
WHERE   CHILD.ENO IN ('1000', '8500', '4700')  
  
ORDER  BY CHILD.ENO
```

ENO	ENAME	SALARY	BOSSENO	BOSSENAME	BOSSSALARY
1000	MOE	2000.00	-	-	-
4700	ANDY	2000.00	4600	ELEANOR	3000.00
8500	GEORGE	7000.00	8000	JOE	8000.00

**Syntax & Logic:** Nothing new. This statement specified a LEFT OUTER JOIN with the child-table as the left-table. This result is the same as the previous result table plus one more row for Employee 1000 (the big-boss) who does not have a supervisor.

## Query-Pattern-5: Three-Level, Parent-Oriented, Matching

The next four sample queries describe parents, children, and grandchildren in a tree. The following sample query traverses three levels, is parent-oriented, and only displays matching rows. This matching applies to parents who have at least one child *where each child also has at least one child*.

**Sample Query 30.19.1:** Consider Employees 1000, 3000, 8500, and 4700. If any of these employees is a supervisor who supervises an employee who is also a supervisor, display the numbers and names of these employees. (I.e., Describe data for parents, children, and grandchildren.) Sort the result by GRANDCHILD.ENO within CHILD.ENO within PARENT.ENO

```
SELECT PARENT.ENO          PARENTENO,
       PARENT.ENAME        PARENTNAME,
       CHILD.ENO           CHILDENO,
       CHILD.ENAME         CHILDNAME,
       GRANDCHILD.ENO      GRANDCHILDENO,
       GRANDCHILD.ENAME    GRANDCHILDNAME

FROM REMPLOYEE PARENT
     INNER JOIN REMPLOYEE CHILD
              ON PARENT.ENO = CHILD.SENO
     INNER JOIN REMPLOYEE GRANDCHILD
              ON CHILD.ENO = GRANDCHILD.SENO

AND PARENT.ENO IN ('1000', '3000', '8500', '4700')

ORDER BY PARENT.ENO, CHILD.ENO, GRANDCHILD.ENO
```

<u>PARENTENO</u>	<u>PARENTNAME</u>	<u>CHILDENO</u>	<u>CHILDNAME</u>	<u>GRANDCHILDENO</u>	<u>GRANDCHILDNAME</u>
1000	MOE	2000	JANET	4000	JULIE
1000	MOE	2000	JANET	5000	JESSIE
1000	MOE	2000	JANET	6000	FRANK
1000	MOE	3000	LARRY	6500	CURLY
1000	MOE	8000	JOE	8500	GEORGE
3000	LARRY	6500	CURLY	7500	SHEMP

**Syntax & Logic:** Observe that: Employee 1000 appears in five rows because he has five grandchildren; Employee 3000 appears in one row because he has one grandchild. Employee 4700 does not appear in the result table because he has no children (and hence on grandchildren); and Employee 8500 does not appear in the result table because he has no grandchildren, even though he does have two children.

## Query-Pattern-6:Three-Level, Parent-Oriented, Non-Matching

The following sample query is similar to the preceding sample query. It accesses three levels and is parent-oriented. Unlike the preceding sample query, this sample query displays both matching and non-matching rows.

**Sample Query 30.19.2:** Display the numbers and names of Employees 1000, 3000, 8500, and 4700. If any of these employees is a supervisor, display each of their supervisee's number and name; and, if any of these supervisees is also a supervisor, display each of these supervisee's number and name. Sort the result by GRANDCHILD.ENO within CHILD.ENO within PARENT.ENO

```
SELECT PARENT.ENO      PARENTENO,
       PARENT.ENAME    PARENTNAME,
       CHILD.ENO       CHILDENO,
       CHILD.ENAME     CHILDNAME,
       GRANDCHILD.ENO GRANDCHILDENO,
       GRANDCHILD.ENAME GRANDCHILDNAME

FROM REMPLOYEE PARENT
   LEFT OUTER JOIN REMPLOYEE CHILD
                   ON PARENT.ENO = CHILD.SENO
   LEFT OUTER JOIN REMPLOYEE GRANDCHILD
                   ON CHILD.ENO = GRANDCHILD.SENO

WHERE PARENT.ENO IN ('1000', '3000', '8500', '4700')

ORDER BY PARENT.ENO, CHILD.ENO, GRANDCHILD.ENO
```

<u>PARENTENO</u>	<u>PARENTNAME</u>	<u>CHILDENO</u>	<u>CHILDNAME</u>	<u>GRANDCHILDENO</u>	<u>GRANDCHILDNAME</u>
1000	MOE	2000	JANET	4000	JULIE
1000	MOE	2000	JANET	5000	JESSIE
1000	MOE	2000	JANET	6000	FRANK
1000	MOE	3000	LARRY	6500	CURLY
1000	MOE	8000	JOE	8500	GEORGE
3000	LARRY	6500	CURLY	7500	SHEMP
4700	ANDY	-	-	-	-
8500	GEORGE	8700	HANK	-	-
8500	GEORGE	8600	DICK	-	-

**Syntax & Logic:** The two LEFT OUTER JOIN operations preserve non-matching rows. Employee 4700 appears in the result table even though he has no children (and hence no grandchildren). Employee 8500 appears in the result even though he does not have any grandchildren.

## Query-Pattern-7: Three-Level, Child-Oriented, Matching

The following sample query traverses three levels, is child-oriented (really grandchild-oriented), and only displays data from matching rows.

**Sample Query 30.19.3:** Consider Employees 1000, 3000, 8500, and 4700. If any of these employees is supervised by a supervisor who is also supervised by a supervisor, then display the number and name of these employees. (I.e., Display grandchild, child, and parent data.) Sort the result by GRANDCHILD.ENO.

```
SELECT GRANDCHILD.ENO      GRANDCHILD.ENO,
        GRANDCHILD.ENAME   GRANDCHILDNAME,
        CHILD.ENO          CHILD.ENO,
        CHILD.ENAME        CHILDNAME,
        PARENT.ENO         PARENT.ENO,
        PARENT.ENAME       PARENTENAME

FROM EMPLOYEE PARENT
     INNER JOIN EMPLOYEE CHILD
           ON PARENT.ENO = CHILD.SENO
     INNER JOIN EMPLOYEE GRANDCHILD
           ON CHILD.ENO = GRANDCHILD.SENO

AND GRANDCHILD.ENO IN ('1000', '3000', '8500', '4700')

ORDER BY GRANDCHILD.ENO
```

<u>GRANDCHILDENO</u>	<u>GRANDCHILDNAME</u>	<u>CHILDENO</u>	<u>CHILDNAME</u>	<u>PARENTENO</u>	<u>PARENTNAME</u>
4700	ANDY	4600	ELEANOR	4000	JULIE
8500	GEORGE	8000	JOE	1000	MOE

**Syntax & Logic:** The objective is to display data for each employee who is a grandchild. Data about Employee 1000 does not appear in the result table because he is the big-boss (not a child; hence not a grandchild). Data about Employee 3000 does not appear in this result because, although he is child, he is not a grandchild.

**Restate Previous Observation:** Examination of this result table shows that a "child-oriented" result is analogous to an upward tree traversal.

## Query-Pattern-8: Three-Level, Child-Oriented, Non-Matching

The following sample query is similar to the preceding sample query. It traverses three levels and is child (really grandchild) oriented. Unlike the preceding sample query, it displays both matching and non-matching rows.

**Sample Query 30.19.4:** Display the numbers and names of Employees 1000, 3000, 8500, and 4700. For each of these employees who has a supervisor, display this supervisor's number and name; and, if one of these supervisors is also a supervisor, display this supervisor's number and name. Sort the result by GRANDCHILD.ENO.

```
SELECT      GRANDCHILD.ENO GRANDCHILDENO,
            GRANDCHILD.ENAME GRANDCHILDNAME,
            CHILD.ENO CHILDENO,
            CHILD.ENAME CHILDNAME,
            PARENT.ENO PARENTENO,
            PARENT.ENAME PARENTENAME

FROM REMPLOYEE GRANDCHILD
        LEFT OUTER JOIN REMPLOYEE CHILD
            ON GRANDCHILD.SENO = CHILD.ENO
        LEFT OUTER JOIN REMPLOYEE PARENT
            ON CHILD.SENO = PARENT.ENO

WHERE GRANDCHILD.ENO IN ('1000', '3000', '8500', '4700')

ORDER BY GRANDCHILD.ENO
```

<u>GRANDCHILDENO</u>	<u>GRANDCHILDNAME</u>	<u>CHILDENO</u>	<u>CHILDNAME</u>	<u>PARENTENO</u>	<u>PARENTNAME</u>
1000	MOE	-	-	-	-
3000	LARRY	1000	MOE	-	-
4700	ANDY	4600	ELEANOR	4000	JULIE
8500	GEORGE	8000	JOE	1000	MOE

**Syntax & Logic:** This result is the same as the previous result table plus two additional non-matching rows. Data about Employee 1000 appears in the result table even though he is the big-boss without a supervisor. Data about Employee 3000 appears in the result table even though his supervisor (the big-boss) does not have a supervisor.

## Exercises

30Z1. Consider Employees 3000 and 8600. Display the number, name, and salary for these employees. Also, if either of these employees is a supervisor, display the number, name, and salary of each immediate supervisee. The result should look like:

<u>BOSSENO</u>	<u>BOSSENAME</u>	<u>BOSSSALARY</u>	<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>
3000	LARRY	3000.00	6500	CURLY	8000.00
8600	DICK	6000.00	-	-	-

30Z2. Consider Employees 3000 and 8600. Display each employee's number, name, and salary followed by the number, name and salary of the employee's immediate supervisor. The result should look like:

<u>ENO</u>	<u>ENAME</u>	<u>SALARY</u>	<u>BOSSENO</u>	<u>BOSSENAME</u>	<u>BOSSSALARY</u>
3000	LARRY	3000.00	1000	MOE	2000.00
8600	DICK	6000.00	8500	GEORGE	7000.00

30Z3. Display the numbers and names of Employees 3000, 5000, and 8000. If any of these employees is a supervisor, display each supervisee's number and name; and, if any of these supervisees is also a supervisor, display each of these supervisee's number and name. Sort the result by the supervisor's (the parent's) ENO value. The result should look like:

<u>PARENTENO</u>	<u>PARENTNAME</u>	<u>CHILDENO</u>	<u>CHILDNAME</u>	<u>GRANDCHILDENO</u>	<u>GRANDCHILDNAME</u>
3000	LARRY	6500	CURLY	7500	SHEMP
5000	JESSIE	5500	HANNAH	-	-
8000	JOE	8500	GEORGE	8600	DICK
8000	JOE	8500	GEORGE	8700	HANK

30Z4. Consider Employees 3000, 6000, and 8500. If any of these employees is supervised by a supervisor who is also supervised by a supervisor, then display the number and name of all such employees. The result should look like:

<u>GRANDCHILDENO</u>	<u>GRANDCHILDNAME</u>	<u>CHILDENO</u>	<u>CHILDNAME</u>	<u>PARENTENO</u>	<u>PARENTNAME</u>
6000	FRANK	2000	JANET	1000	MOE
8500	GEORGE	8000	JOE	1000	MOE

## Many-to Many Relationship

The preceding query-patterns can be applied to the `REMPLOYEE2` and `REPORTS_TO` tables that represent a recursive many-to-many relationship. In preparation of the next two sample queries, we review Figure 30.1c which illustrates the network diagram for the `REPORTS_TO` table. This figure highlights (in bold font) five nodes (1000, 2000, 3000, 4700, and 8500) which will be referenced in the next two sample queries.

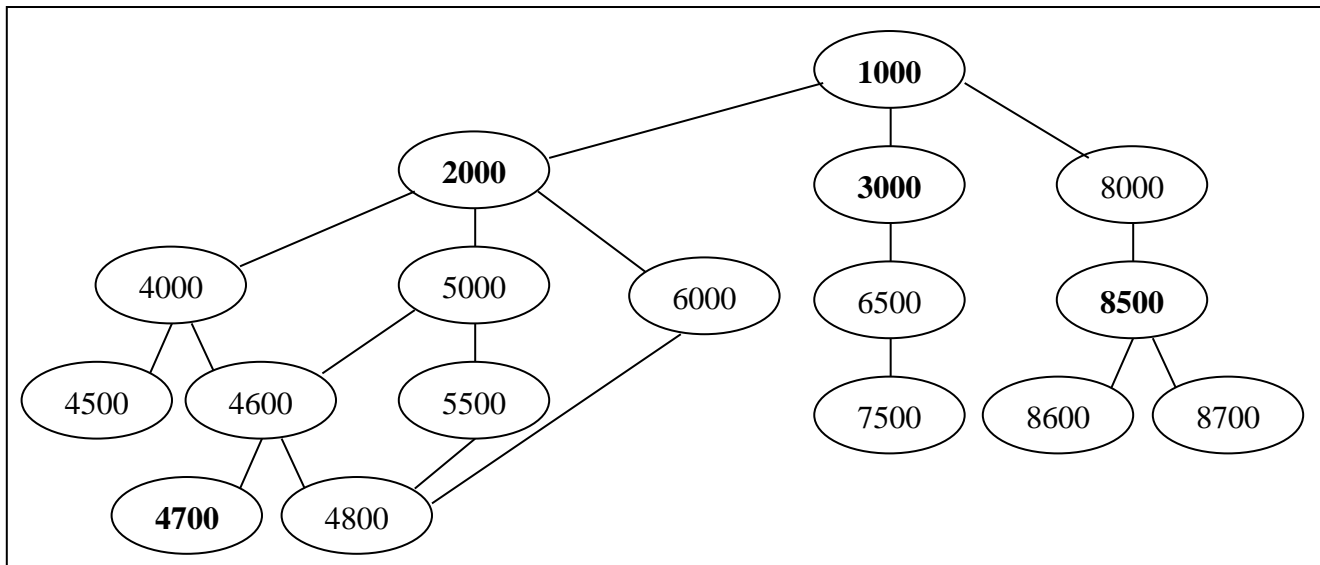


Figure 30.5c: Network Representation of `REMPLOYEE2` and `REPORTS_TO` Tables

The following Sample Queries 30.20.1 and 30.20.2 specify a non-recursive CTE called `REMPLOYEEEX` which is derived by joining the `REMPLOYEE2` and `REPORTS_TO` tables.

```
WITH REMPLOYEEX (ENO, ENAME, SALARY, SENO)  
AS  
(SELECT RT.ENO, R2.ENAME, R2.SALARY, RT.SENO  
FROM REPORTS_TO RT, REMPLOYEE2 R2  
WHERE RT.ENO = R2.ENO)
```

Sample Query 30.20.1 will code a three-level network traversal starting at nodes for Employees 1000, 3000, 8500, and 4700. In this example, only one path leads from a parent-node to a grandchild-node.

Sample Query 30.20.2 is similar to Sample Query 30.20.1. It will code a three-level network traversal starting at the node for Employee-2000. This example shows two paths leading from a parent-node to a grandchild-node.

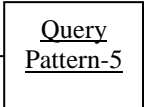
**Sample Query 30.20.1:** Same as Sample Query 30.19.1 (Query- Pattern-5). Consider Employees 1000, 3000, 8500, and 4700. If any of these employees is a supervisor who supervises an employee who is also a supervisor, display the numbers and names of these employees. (I.e., Describe data for parents, children, and grandchildren.) Sort the result by GRANDCHILD.ENO within CHILD.ENO within PARENT.ENO

```

WITH EMPLOYEEEX (ENO, ENAME, SALARY, SENO)
AS
(SELECT RT.ENO, R2.ENAME, R2.SALARY, RT.SENO
FROM REPORTS_TO RT, EMPLOYEE2 R2
WHERE RT.ENO = R2.ENO)

SELECT PARENT.ENO          PARENTENO,
       PARENT.ENAME        PARENTNAME,
       CHILD.ENO           CHILDENO,
       CHILD.ENAME         CHILDNAME,
       GRANDCHILD.ENO     GRANDCHILDENO,
       GRANDCHILD.ENAME   GRANDCHILDNAME
FROM EMPLOYEEEX PARENT
     INNER JOIN EMPLOYEEEX CHILD
               ON PARENT.ENO = CHILD.SENO
     INNER JOIN EMPLOYEEEX GRANDCHILD
               ON CHILD.ENO = GRANDCHILD.SENO
AND PARENT.ENO IN ('1000', '3000', '8500', '4700')
ORDER BY PARENT.ENO, CHILD.ENO, GRANDCHILD.ENO

```



PARENTENO	PARENTNAME	CHILDENO	CHILDNAME	GRANDCHILDENO	GRANDCHILDNAME
1000	MOE	2000	JANET	4000	JULIE
1000	MOE	2000	JANET	5000	JESSIE
1000	MOE	2000	JANET	6000	FRANK
1000	MOE	3000	LARRY	6500	CURLY
1000	MOE	8000	JOE	8500	GEORGE
3000	LARRY	6500	CURLY	7500	SHEMP

**Syntax & Logic:** Nothing new. Examination of the network diagram (Figure 30.5C) shows that the above five rows correspond to the five paths from Node-1000 to its five grandchild-nodes. *Note: that the five GRANDCHILD.ENO values are distinct. Hence, there is only one path to each grandchild.* This does not occur in following the sample query.

Also, observe that rows for Node-8500 and Node-4700 do not appear in the result table because they do not have any grandchildren.



The following sample query illustrates that, *within a network, there may be multiple paths from a starting node to a descendant node.*

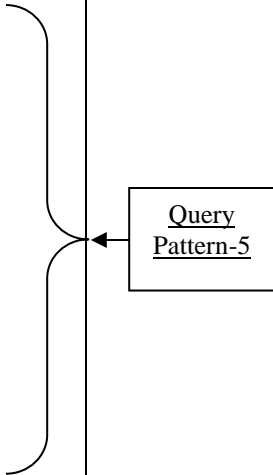
**Sample Query 30.20.2:** Same logic as preceding Sample Query 30.20.1. The only difference is that we want to display data about the children and grandchildren of just one employee, Employee 2000.

```

WITH REMPLOYEEX (ENO, ENAME, SALARY, SENO)
AS
(SELECT RT.ENO, R2.ENAME, R2.SALARY, RT.SENO
 FROM REPORTS_TO RT, REMPLOYEE2 R2
 WHERE RT.ENO = R2.ENO)

SELECT PARENT.ENO      PARENTENO,
       PARENT.ENAME    PARENTNAME,
       CHILD.ENO       CHILDENO,
       CHILD.ENAME     CHILDNAME,
       GRANDCHILD.ENO GRANDCHILDENO,
       GRANDCHILD.ENAME GRANDCHILDNAME
FROM REMPLOYEEX PARENT
     INNER JOIN REMPLOYEEX CHILD
           ON PARENT.ENO = CHILD.SENO
     INNER JOIN REMPLOYEEX GRANDCHILD
           ON CHILD.ENO = GRANDCHILD.SENO
AND PARENT.ENO = '2000'
ORDER BY PARENT.ENO, CHILD.ENO, GRANDCHILD.ENO

```



PARENTENO	PARENTNAME	CHILDENO	CHILDNAME	GRANDCHILDENO	GRANDCHILDNAME
2000	JANET	4000	JULIE	4500	JOHNNY
2000	JANET	4000	JULIE	4600	ELEANOR
2000	JANET	5000	JESSIE	4600	ELEANOR
2000	JANET	5000	JESSIE	5500	HANNAH
2000	JANET	6000	FRANK	4800	MATT

**Syntax & Logic:** Examination of the network diagram (Figure 30.5C) shows that the above five rows correspond to the five paths from Node-2000 to its *four* grandchild nodes. *It is important to note that there are two paths from Node-2000 to Node-4600.*

## Exercise

30Z5. Reference the `RERORTS_TO` and the `REMPLOYEE2` tables (representing a recursive many-to-many relationship). Display the numbers and names of Employees 3000, 5000, and 8500. If any of these employees is a supervisor, display each supervisee's number and name; and, if any of these supervisees is also a supervisor, display each of these supervisee's number and name. Sort the result by the supervisor's (the parent's) `ENO` value. The result should look like:

<u>PARENTENO</u>	<u>PARENTNAME</u>	<u>CHILDENO</u>	<u>CHILDNAME</u>	<u>GRANDCHILDENO</u>	<u>GRANDCHILDNAME</u>
3000	LARRY	6500	CURLY	7500	SHEMP
5000	JESSIE	4600	ELEANOR	4700	ANDY
5000	JESSIE	4600	ELEANOR	4800	MATT
5000	JESSIE	5500	HANNAH	4800	MATT
8500	GEORGE	8700	HANK	-	-
8500	GEORGE	8600	DICK	-	-

## Self-Joins versus Recursive CTE

**Self-Join Disadvantages:** Self-joins have two obvious limitations when compared to recursive CTEs. First, the self-join method does not return rows in some hierarchical sequence. Second, with the self-join method, you are limited to some fixed number of levels. This section's self-join examples could be extended to a fourth-level, fifth-level, etc. In practice, there is some reasonable (unspecified) limit to this kind of extension.

**Self-Join Advantages:** Some users may feel that coding a self-join is simpler than coding a recursive CTE; and, horizontally-oriented result table are easier to understand.

This focus on friendlier SELECT statements and friendlier result-tables may appear to conflict with our previous recommendation that you should focus on coding correct SELECT statements and use front-end tools for report-formatting. Again, the gospel is: Code correct statements. If a query objective can be satisfied by using either method, friendliness is your choice. However, recall that, at some time in the future, your code may have to be modified by some other user.

## Summary

**ORDER BY Clauses:** Many sample queries in this chapter disregarded our strong recommendation to: "Do not rely on an incidentally sorted result. Always code an ORDER BY clause to return your result in the desired row sequence." We address this issue by considering two possible circumstances.

- 1. Your database system supports an ORDER BY clause that allows you to specify a hierarchical sequence,** such as the SEARCH-ORDER BY clauses described after Sample Query 30.9. In this circumstance, the above recommendation applies.
- 2. Your database system does not (yet) support an ORDER BY clause that allows you to specify a hierarchical sequence.** In this circumstance, within the context of recursive queries, the above recommendation becomes problematic. In this chapter, some sample queries did not code an ORDER BY clause and relied on the default breath-first hierarchical sequence. This could be considered an incidental sort. Also, other sample queries included operations (e.g., DISTINCT, GROUP BY) in the Main-SELECT that disrupted a default hierarchical sequence. Some users will have to deal with these issues until a new version of their database provides code that allows the ORDER BY clause to specify a desired hierarchical sequence.

## Appendix 30A: Theory

Recursion is a very interesting topic that appears in many disciplines, including mathematics, software engineering, and philosophy. Our description of these topics is very brief. Fortunately, a web search for each topic will return many hits.

### A. Mathematics

Most textbooks on discrete mathematics discuss recursive functions. A recursive function is defined in terms of itself. Below we present recursive definitions for (i) Factorial Numbers and (ii) Fibonacci Numbers.

Two methods will be used to calculate each function. The first method is an iterative method which most users prefer as the "easy way" to calculate the function. The second method is a recursive method that conforms to the recursive definition.

#### Factorial (!) Function

##### Iterative (Easy Way) Method

$$n! = n * (n-1) * (n-2) * (n-3) * \dots * 1$$

$$\text{Examples: } 5! = 5*4*3*2*1 = 120$$

$$7! = 7*6*5*4*3*2*1 = 5,040$$

##### Recursive Method

$$n! = n * (n-1)!, \text{ where } 0! = 1$$

Example:

$$\begin{aligned} 5! &= 5 * (4!) \\ &= 5 * 4 * (3!) \\ &= 5 * 4 * 3 * (2!) \\ &= 5 * 4 * 3 * 2 * (1!) \\ &= 5 * 4 * 3 * 2 * 1 * (0!) \\ &= 5 * 4 * 3 * 2 * 1 * 1 \\ &= 5 * 4 * 3 * 2 * 1 \\ &= 5 * 4 * 3 * 2 \\ &= 5 * 4 * 6 \\ &= 5 * 24 \\ &= 120 \end{aligned}$$

## Fibonacci Numbers

### Iterative ("Easy Way") Method

Fibonacci numbers are produced by generating a sequence of values:

$F_0, F_1, F_2, F_3, F_4, F_5, F_6, F_7, F_8 \dots$

Start with  $F_0 = 0$  and  $F_1 = 1$ .

Calculate the next number by adding the previous two numbers.

Example: Find the value of  $F_6$

Calculate  $F_2$ : Add the first two numbers, producing 1

$F_0, F_1, F_2$

0, 1, 1  
└─┘

Calculate  $F_3$ : Add two previous numbers, producing 2

$F_0, F_1, F_2, F_3$ ,

0, 1, 1, 2  
└─┘

Calculate  $F_4$ : Add two previous numbers, producing 3

$F_0, F_1, F_2, F_3, F_4$

0, 1, 1, 2, 3  
└─┘

Calculate  $F_5$ : Add two previous numbers, producing 5

$F_0, F_1, F_2, F_3, F_4, F_5$

0, 1, 1, 2, 3, 5  
└─┘

Calculate  $F_6$ : Add two previous numbers, producing 8

$F_0, F_1, F_2, F_3, F_4, F_5, F_6$

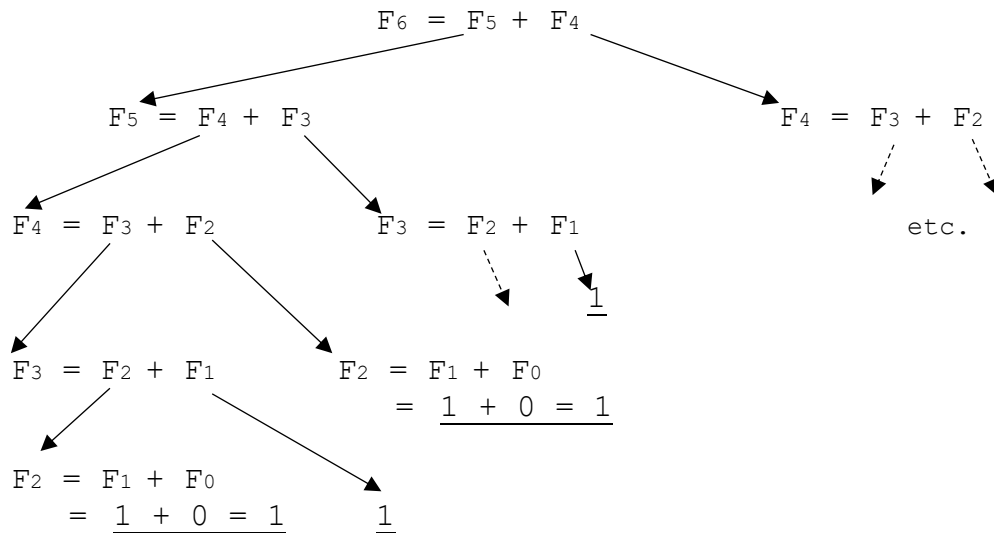
0, 1, 1, 2, 3, 5, 8  
└─┘

Stop.  $F_6 = 8$

## Fibonacci - Recursive Definition

$$F_n = F_{(n-1)} + F_{(n-2)} \quad \text{where } F_0 = 0 \text{ and } F_1 = 1$$

The following figure illustrates some (not all) of the recursive executions of the  $F_n$  function. You are invited to add more details to complete this figure.



## B. Software Engineering

A recursive program will call itself. Some, but not all, programming languages (e.g., C++, Java, Python) support recursion.

Imagine that you were asked to write programs to generate Factorial numbers and Fibonacci numbers. Most users would be inclined to code "easy way" iterative solutions. But, for academic reasons, you are asked to write a recursive program for each programming task.

The following Java skeleton-code illustrates recursive code for the Factorial function.

```
Fact (int n)
{
    if (n >= 1) return n * Fact (n-1);
    else return 1;
}
```

The following Java skeleton-code illustrates recursive code for Fibonacci numbers.

```
Fib (int n)
{
    if (n == 0) {return 0;}

    if (n == 1) || (n == 2) {return 1;}

    return Fib (n-2) + Fib (n-1);
}
```

A web search will turn up many complete programs in Java and other languages for the above functions.

These programming exercises may be academic. However, in some circumstances (unlike the Factorial and Fibonacci examples), **a recursive program can be much smaller and much simpler than an iterative program** that satisfies the same objective. The following pages identifies three examples (The first two examples assume that you understand arrays. The third example assumes you understand using pointers to build a tree structure.) Complete solutions for these examples can be found on the web.



Knight's Tour: Assume you understand how a knight moves on a chessboard. Represent the chessboard as a two-dimensional array. Then, write a recursive program to display a *Knight's Tour*. Begin by placing a knight on any square on the chessboard. Then code a program to discover and display how the knight can make 63 consecutive moves to visit all of the other 63 squares exactly once.

Eight Queens Problem: Assume you understand how a queen moves on a chessboard. Represent the chessboard as a two-dimensional array. Write a recursive program to solve the Eight Queens Problem. This program should discover and display how eight queens can be placed on the chessboard such that no two queens can threaten each other.

Search Binary Tree: Assume you have a collection of records organized as a binary tree. Write a recursive program to traverse all nodes in the tree to display a specified field in each record.

## **C. Philosophy: Logic**

There are many logical paradoxes associated with self-reference. We present two classical examples and one modern variation.

### Liar's Paradox

Consider the following statement.

"This statement is False."

Is this statement True or False?

If this statement true, then it's false, and  
If this statement false, then it's true.

Hence, we have a contradiction.

### Epimenides Paradox

Epimenides lived in Crete.

He said: "All Cretans are liars."

Did he speak the truth?

### Dilbert

A new study says that "all studies are misleading."

## **D. Philosophy: Mathematics – Russell’s Paradox**

In the early 1900’s, Bertrand Russell presented his famous paradox that had a profound impact on the philosophy of mathematics. This paradox is outlined below.

### **Sets of Sets**

Sometimes a set has elements that are also sets.

Example-1: Consider some sets involving athletic teams in a large university.

WBBALL = set of players on women’s basketball team

MBBALL = set of players on men’s basketball team

WTENNIS = set of players on women’s tennis team

MTENNIS = set of players on men’s tennis team

WGOLF = set of players on women’s golf team

MGOLF = set of players on men’s golf team

Etc.

The following two sets, WTEAMS and MTEAMS, are sets of sets.

WTEAMS = {WBBALL, WTENNIS, WGOLF, . . .}

MTEAMS = {MBBALL, MTENNIS, MGOLF, . . .}

Example-2: A relational database is a set of relations (tables) where each relation is a set of n-tuples (rows).

## Normal versus Abnormal Sets

All sets are either (1) normal or (2) abnormal where:

A set is *normal* if it is not an element of itself.

A set is *abnormal* if it is an element of itself.

Normal Sets: Most sets are normal. Consider sets:

S1 = positive even integers: {2, 4, 6, 8, ...}

S2 = the seven days in a week: {Monday, Tuesday, ...}

S3 = 64 cells on a chessboard

S4 = 52 cards in a conventional deck of cards

Demonstrate that S1 is normal.

Assume S1 is abnormal.

Then S1 would equal {S1, 2, 4, 6, 8, ...}.

This is a contradiction because S1 only contains even integers, and S1 is not an integer.

Hence, S1 is normal.

Demonstrate that S2 is normal.

Assume S2 is abnormal.

Then S2 equals {S2, Monday, Tuesday, Wednesday,  
Thursday, Friday, Saturday, Sunday}

This is a contradiction because S2 is not a day of the week.

Hence, S2 is normal.

Similar logic applies to sets S3, S4, and probably most other sets that you can imagine.

## Abnormal Sets

Consider the following B3PLUS set.

B3PLUS = Set of all sets that have more than 3 elements.

The above sets S1, S2, S3, and S4 have more than 3 elements.

So, these sets are elements of B3PLUS.

B3PLUS = {S1, S2, S3, S4, ...}

And, because B3PLUS has more than three elements, B3PLUS is an element of B3PLUS.

B3PLUS = {B3PLUS, S1, S2, S3, S4, ...}

Hence, B3PLUS is abnormal because it is an element of itself.

## Russell's Paradox

Russell considered the set R where:

R = set of all normal sets

(R contains *all* sets that not elements of themselves.)

Russell asked: *Is R a normal set?*

Answering this question leads to a paradox.

Russell assumed that:

(1) R was normal, and then deduced that R was abnormal.

Then he assumed that:

(2) R was abnormal, and then deduced that R was normal.

*Contradiction!*

Details:

- Assume R is normal.  
Hence, R is not an element of itself.  
Now, because R contains *all* sets that not elements of themselves, R must contain R.  
Hence, R is abnormal.
- Assume R is abnormal.  
Hence R is an element of itself.  
Now, because R *only* contains sets that not elements of themselves, R is not an element of R.  
Hence, R is normal.

-----  
Final Thought: The philosophy of mathematics may be more interesting than SQL. 😊

## Book Appendices

The first two appendices are intended to be read sometime after you have read Chapters 0 and 1 in The Free SQL Book. The other three appendices are intended to be read after you have read most of this book.

### I. Create Sample Tables for The Free SQL Book

This appendix assumes that you have already obtained access to some relational database system. (Read the following Book-Appendix-II if this does not apply.) Creating the sample tables for the FREESQL database is straightforward. You execute a script found at [www.freesqlbook.com](http://www.freesqlbook.com) website. This appendix also presents a tutorial introduction to SQL Scripts for rookie users.

### II. Obtain Access to a Relational Database System

Gaining access to some relational database system (e.g., DB2, SQL Server, ORACLE, MYSQL) may be easy, or it may require some effort. This appendix will outline multiple ways to satisfy this objective.

### III. Summary of Chapter Appendices

Many chapter appendices present by-the-way commentary on topics pertaining to the SQL statements introduced in the corresponding chapter. This appendix organizes and summarizes these topics.

### IV. Post-Relational Database Systems & NoSQL

Codd proposed his Relational Database Model over 50 years ago! Thereafter, database researchers proposed many other *post-relational database models*. This appendix comments on some of these post-relational models.

### V. Abbreviated Bibliography

A few of this author's favorite books.

This page is intentionally blank.

## **Book-Appendix-I**

### **Create Sample Tables for The Free SQL Book**

#### **Prerequisite Knowledge**

Rookies: Before reading this appendix, you should have read Chapter 0 in The Free SQL Book. Specifically, review the narrative about Figure 0.2 which introduces the CREATE TABLE statement for the PRESERVE table.

#### **Appendix Objectives**

This appendix is organized into three sections.

- A. Tutorial on SQL Scripts: Rookie users should read this tutorial. Experienced users who already have a basic understanding of SQL scripts can jump to Section B.
- B. CHPT-1-5 Script: This section describes the CHPT-1-5 script (found at the [www.freesqlbook.com](http://www.freesqlbook.com) website). This script creates and populates the only two tables referenced in the first five chapters of The Free SQL Book. Executing this (optional) script will allow rookie users to learn their front-end tool and its script processing facilities as they work your way through the first five chapters.
- C. Create All Sample Tables: All users eventually execute a script (found at the [www.freesqlbook.com](http://www.freesqlbook.com) website) to *create and populate all sample tables* referenced in The Free SQL Book. Users of DB2, ORACLE, and SQL Server users will execute one of the following scripts.

CREATE-ALL-TABLES-DB2

CREATE-ALL-TABLES-ORACLE

CREATE-ALL-TABLES-SQLServ

If you are using some other database system (e.g. MYSQL), you should be able to use the DB2 script with very few changes. (There is a good chance that you will have to edit your script to account for DATE data in the DEMO3 table.)



## **A. Tutorial on SQL Scripts**

A SQL script is a collection of SQL statements where each statement is terminated by a semicolon. (Most sample queries in this book only specify a single SELECT statement. Hence there is no need to terminate the statement with a semicolon.)

The following Figure I.1 illustrates a script called BIRD-Script. This script includes three SQL statements (DROP TABLE, INSERT, and COMMIT) that were not introduced in Chapter 0. These statements will be described below. Notice that a semicolon terminates each statement.

```
DROP TABLE BIRD;

CREATE TABLE BIRD
(SPECIES CHAR(10) NOT NULL UNIQUE,
 BLENGTH INTEGER NOT NULL);

INSERT INTO BIRD VALUES ('ROBIN', 11);
INSERT INTO BIRD VALUES ('SPARROW', 6);
INSERT INTO BIRD VALUES ('BLUEBIRD', 7);

COMMIT;
```

Figure I.1: BIRD-Script

### DROP TABLE BIRD;

This statement removes the BIRD table from the database. Because the BIRD table has not yet been created, this DROP TABLE statement will return some kind of “No-BIRD-table-found” error-message. The absence of a BIRD table means that the following CREATE TABLE statement will execute without any kind of “Table-already-exists” error message.

[Occasionally you want to drop a table just before you (re-)create and (re-)populate the same table. For example, after creating and populating the BIRD table, for tutorial reasons, you might make changes to this table. If you do this, you can easily return the BIRD table to its original state by (re-)executing the BIRD-Script.]

```
CREATE TABLE BIRD . . .;
```

This statement creates an empty table called BIRD which has two columns. The SPECIES column identifies a bird's species. SPECIES is a fixed-length 10-character column that is declared to be UNIQUE. The BLENGTH column describes the average length of the bird measured in inches. BLENGTH is an integer column. Both columns are specified as NOT NULL.

```
INSERT INTO BIRD . . .;
```

Each of the three INSERT statements inserts a row into the previously created BIRD table. *You can ignore syntactical details for these INSERT statements.* These details are described in Chapter 15.

```
COMMIT;
```

COMMIT is *optional* within the context of this script. Without explanation, COMMIT asks the system to make the preceding INSERT operations "permanent." Chapter 29 will discuss the COMMIT statement. (Note: SQL Server users should remove this COMMIT statement from the BIRD-Script.)

**Creating a Script File:** You can use a conventional text editor to create and save a script file. Some observations follow.

- Usually, no special file type is required. However, many users specify "sql" as the file type for a file that contains SQL statements (e.g., BIRD-Script.sql).
- To get started, you can save your script in any personal folder/directory. Then copy-and-paste the script into a SQL Panel when you want to execute it. (Your front-end tool should also provide alternative methods for creating, saving, and executing scripts.)
- If you examine a script written by your DBA or another user, you may observe comments (to be described in the following Section B). You may also observe other non-SQL "commands" that are understood by your database system or front-end tool. Such commands are not included in this book's scripts

## Optional Exercise-1: Execute Individual Statements in the BIRD-Script

For tutorial reasons, before executing the BIRD-Script, this exercise invites you to execute each statement as an individual statement.

1. Copy-and-paste just the DROP TABLE BIRD statement into an empty SQL Panel in your front-end tool. Then execute this statement. The system will return some kind of "No-BIRD-table-found" error-message in (or near) the Result Panel.
2. Copy-and-paste just the CREATE TABLE BIRD statement into an empty SQL Panel. Then execute this statement. The system should return some kind of success-message in (or near) the Result Panel.
3. Copy-and-paste the first INSERT statement into an empty SQL Panel. Then execute this statement. The system should return some kind of success-message in (or near) the Result Panel.
4. Execute: `SELECT * FROM BIRD`  
Observe a one-row result table in the Result Panel.
5. Copy-and-paste the second INSERT statement into an empty SQL Panel. Then execute this statement. The system should return some kind of success-message in (or near) the Result Panel.
6. Execute: `SELECT * FROM BIRD`  
Observe a two-row result table in the Result Panel.
7. Copy-and-paste the third INSERT statement into an empty SQL Panel. Then execute this statement. The system should return some kind of success-message in (or near) the Result Panel.
8. Execute: `SELECT * FROM BIRD`  
Observe a three-row result table in the Result Panel.
9. Finally, execute: `DROP TABLE BIRD`  
The system will return some kind of success-message in (or near) the Result Panel.

## Optional Exercise-2: Execute (all statements in) the BIRD-Script

Create and execute an SQL script.

1. Save the BIRD-Script into some file (e.g., BIRD-Script.sql).
2. Copy-and-paste all statements from the BIRD-Script file into an *empty* SQL Panel. [SQL Server users should remove the COMMIT statement.]
3. *Execute the BIRD-Script. Because the SQL Panel contains multiple statements terminated by semicolons, the system will execute each statement from top to bottom.*
4. **Examine Results:** The system will return success-messages and/or error-messages. Specific messages will vary among different systems.

What happens if one of the statements causes an error? Most systems will ignore the erroneous statement and try to execute the following statements.

After you execute the BIRD-Script, you should see:

- A "No-BIRD-table-found" error-message for the DROP TABLE BIRD statement because the BIRD table does not exist. (It was dropped in Step-9 in the preceding Exercise-1.)
- One success-message for the CREATE TABLE statement, and one success-message for each of the three INSERT statements.
- DB2 and ORACLE users will see a success-message for the COMMIT statement.

Alternatively, your front-end tool might respond with just one "all-went-well" success-message, or one "something went-wrong" error-message.

Finally, verify all-went-well by executing: `SELECT * FROM BIRD`

## **B. CHPT-1-5-Script**

When starting this book, you may prefer to execute a small script that creates the only two tables that are referenced in the first five chapters. This will allow you time to learn your front-end tool and its script processing facilities as you work your way through the first five chapters.

Figure I.2 (on the following page) displays the CHPT-1-5-Script. This script creates and populates the PRESERVE table (described in Chapter 0) and the EMPLOYEE table (described in the Summary Exercises for Chapter 1). These are the only two tables that are referenced in Chapters 1-5. Copy-and-paste this script into the SQL Panel and execute it. Verify success by executing the SELECT statement for any sample query in Chapters 1-5.

**Comments in Scripts:** For documentation purposes, you can include comments in a script. A double-dash (--) is used to indicate a comment which is ignored by the system. For example, the first few lines in the CHPT-1-5-Script specify comments.

```
-- This script creates two tables, the PRESERVE and EMPLOYEE tables.  
-- These are the only tables that will be referenced in  
-- Chapters 1-5 in The FREE SQL Book.  
-- SQL Server users should remove the two COMMIT statements from this script.
```

Here, because a double-dash appears at the beginning of a line, the entire line is a comment. You can also append a comment at the end of a SQL statement as illustrated below.

```
COMMIT;      -- SQL Server users should remove this statement
```

**Executing a Script:** It is easy to copy-and-paste all SQL statements from the small CHPT-1-5-Script into the SQL Panel. You can also copy-and-paste a larger CREATE-ALL-TABLES script into the SQL Panel. Also, most front-end tools provide other methods to create, save, and execute scripts. Assuming the user has already coded and saved a script within a text file, this method usually involves:

- (1) The user clicks on some kind of "Execute-Script" button.
- (2) The tool asks the user to identify a file containing a script.
- (3) The tool responds by placing the script into the SQL Panel.
- (4) The user executes the script.

```

-- This script is found at www.freesqlbook.com
-- This script creates two tables, the PRESERVE and EMPLOYEE tables.
-- These are the only tables that will be referenced in Chapters 1-5.

-- SQL Server users should remove the two COMMIT statements from this script.

DROP TABLE PRESERVE;
DROP TABLE EMPLOYEE;

CREATE TABLE PRESERVE
(PNO    INTEGER          NOT NULL UNIQUE,
 PNAME VARCHAR (25)     NOT NULL,
 STATE CHAR (2)         NOT NULL,
 ACRES  INTEGER          NOT NULL,
 FEE    DECIMAL (5,2)    NOT NULL);

INSERT INTO PRESERVE VALUES (5, 'HASSAYAMPA RIVER', 'AZ', 660, 3.00);
INSERT INTO PRESERVE VALUES (3, 'DANCING PRAIRIE', 'MT', 680, 0.00);
INSERT INTO PRESERVE VALUES (7, 'MULESHOE RANCH', 'AZ', 49120, 0.00);
INSERT INTO PRESERVE VALUES (40, 'SOUTH FORK MADISON', 'MT', 121, 0.00);
INSERT INTO PRESERVE VALUES (14, 'MCELWAIN-OLSEN', 'MA', 66, 0.00);
INSERT INTO PRESERVE VALUES (13, 'TATKON', 'MA', 40, 0.00);
INSERT INTO PRESERVE VALUES (9, 'DAVID H. SMITH', 'MA', 830, 0.00);
INSERT INTO PRESERVE VALUES (11, 'MIACOMET MOORS', 'MA', 4, 0.00);
INSERT INTO PRESERVE VALUES (12, 'MOUNT PLANTAIN', 'MA', 730, 0.00);
INSERT INTO PRESERVE VALUES (1, 'COMERTOWN PRAIRIE', 'MT', 1130, 0.00);
INSERT INTO PRESERVE VALUES (2, 'PINE BUTTE SWAMP', 'MT', 15000, 0.00);
INSERT INTO PRESERVE VALUES (80, 'RAMSEY CANYON', 'AZ', 380, 3.00);
INSERT INTO PRESERVE VALUES (10, 'HOFT FARM', 'MA', 90, 0.00);
INSERT INTO PRESERVE VALUES (6, 'PAPAGONIA-SONOITA CREEK', 'AZ', 1200, 3.00);

COMMIT; -- SQL Server users should remove this statement

CREATE TABLE EMPLOYEE
(ENO    CHAR (4)         NOT NULL UNIQUE,
 ENAME  VARCHAR(25)     NOT NULL,
 SALARY DECIMAL (7,2)   NOT NULL,
 DNO    INTEGER          NOT NULL);

INSERT INTO EMPLOYEE VALUES ('1000', 'MOE', 2000.00, 20);
INSERT INTO EMPLOYEE VALUES ('2000', 'LARRY', 2000.00, 10);
INSERT INTO EMPLOYEE VALUES ('3000', 'CURLY', 3000.00, 20);
INSERT INTO EMPLOYEE VALUES ('4000', 'SHEMP', 500.00, 40);
INSERT INTO EMPLOYEE VALUES ('5000', 'JOE', 400.00, 10);
INSERT INTO EMPLOYEE VALUES ('6000', 'GEORGE', 9000.00, 20);

COMMIT; -- SQL Server users should remove this statement

```

Figure I.2: CHPT-1-5-Script

## **C. Create All Sample Tables**

Users of DB2, ORACLE, and SQL Server will execute one of the following scripts (found at the [www.freesqlbook.com](http://www.freesqlbook.com) website) to *create and populate all sample database tables* referenced in The FREESQL Book. If you have executed the CHPT-1-5-Script, the following scripts will drop and re-create the PRESERVE and EMPLOYEE tables.

```
CREATE-ALL-TABLES-DB2
```

```
CREATE-ALL-TABLES-ORACLE
```

```
CREATE-ALL-TABLES-SQLServer
```

Each script creates the same 40 tables. (These tables are very small. The largest table has 62 rows.) There are only a few differences among these three scripts.

1. The CREATE TABLE and INSERT statements that create and populate the DEMO3 table will differ. These statements differ because they reference a column (BDDATE) with a DATE data-type, and date-time processing differs across DB2, ORACLE, and SQL Server.
2. The COMMIT statement is included in the DB2 and ORACLE scripts, but it is not included in the SQL Server script.

Other Database Systems? If you are using some other database system, you might be able to use the DB2 script. However, there is a good chance that you will have to edit your script to account for DATE data in the DEMO3 table.

## Book Appendix-II

### Access a Relational Database System

**Prerequisite Knowledge:** Before reading this appendix, you should have read Chapter 0 and Chapter 1 in The Free SQL Book. Specifically, review the narrative about Figure 0.2 which introduces the CREATE TABLE statement for the PRESERVE table. Also review the discussions of system architecture in Chapter 0 and metadata in Chapter 1.

**Objective:** Help SQL users gain access to: (1) a relational database management system using (2) some front-end tool.

1. Relational Database Management System (RDBMS): This RDBMS could be DB2, SQL Server, ORACLE, MYSQL, SQLite, etc.

Most users do *not* have to learn much about their RDBMS per se. Instead, these users must learn SQL to ask the RDBMS to take some action on their behalf. They will use a front-end tool to submit SQL statements to the RDBMS and display results.

2. Front-End Tool: A front-end tool is a graphical interface that allows a user to execute SQL statements and observe the results. Also, this tool usually allows a user to display metadata about the data stored in the database. The following Figure 1.1 (from Chapter 1) outlines the SQL-Page displayed by a typical front-end tool.

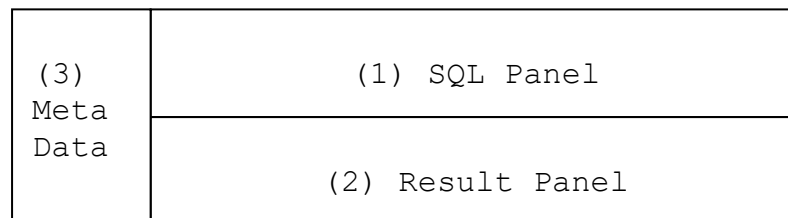


Figure 1.1: SQL-Page

The SQL-Page in most front-end tools will be more complex because such tools usually support other activities (e.g., application programming with embedded SQL) that are beyond the scope of this book. Hence, new users must take some time to learn the basic organization of their front-end tool.

The following section on System Architecture describes how a front-end tool can interact with an RDBMS.

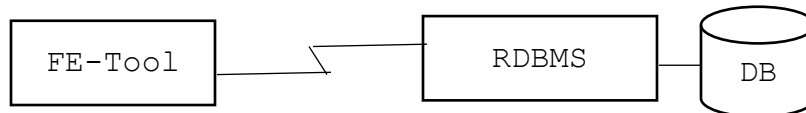


## System Architecture

The RDBMS and front-end tool (FE-Tool) are usually installed within one of the following three system architectures.

### 1. Client-Server System

Here, the front-end-tool and RDBMS reside on different computers. The front-end-tool executes on a "client computer," and the RDBMS executes on a different computer called a "database server." The client and server are connected via a communications network which may be the Internet.



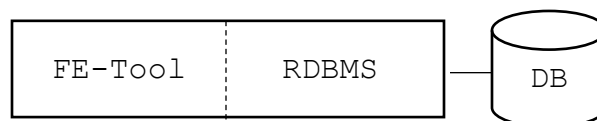
Client-server systems are common within business organizations where many employees have a client computer sitting on their own desk. Client-server systems are also common within schools where students share client machines located in a computer-lab. (Sometimes, an employee/student will also install a front-end tool on their own personal computer at home.)

Advantages: The RDBMS which has already been installed and is managed by a (presumably friendly) DBA; and, most likely, some other system techie has already installed a front-end tool on the client computers.

Disadvantage: Users have limited privileges on the RDBMS.

### 2. Stand-Alone Systems

Here, the front-end tool and the RDBMS reside on the same computer. We will assume this is the user's personal computer.

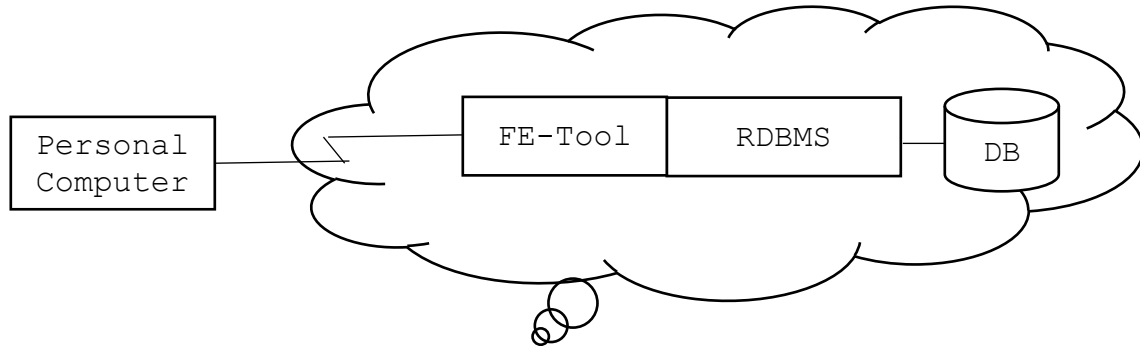


Advantage: Users have all privileges on the RDBMS.

Disadvantage: The user must download and install both the RDBMS and the front-end tool.

### 3. Cloud-Based Systems

Some companies (e.g., Amazon, ORACLE) sell access to cloud-based computing resources, including database systems. Here, the cloud provides a web-based front-end tool and RDBMS. Your personal computer can access the cloud's front-end tool via the Internet. (Alternatively, if you have installed a front-end tool on your personal computer, you may be able use it to directly connect to the RDBMS.)



Advantage: Users do not have to install anything.

Disadvantages: Users must to pay a modest fee to access the cloud-based system, and they have limited privileges on the RDBMS.

### Databases

We present a more precise description of a "database."

Chapter 0 indicated that a database is the collection of all tables stored within a relational database system. To be more precise, we note that the typical database system usually contains multiple databases. For example, a small college might have three applications-level databases: the ACADEMIC database, the ATHLETIC database, and the ALUMNI database. These databases could contain the following tables.

- ACADEMIC Database: STUDENT, COURSE, and FACULTY tables
- ATHLETIC Database: TEAM, ATHLETE, and EVENT tables
- ALUMNI Database: GRADUATE and JOB tables

In this book, we assume that the DBA has created a database called FREESQLDB that will contain the PRESERVE table and all other tables referenced in The Free SQL Book. After you connect your front-end tool to your RDBMS, you will connect to the FREESQLDB Database.

## Installation Scenarios

The following pages describe five scenarios. Each scenario offers a **conceptual overview** that should help you achieve your particular objective. (These scenarios do not present details about the installation of any specific RDBMS or front-end tool.) All users should read Scenario-1A and Scenario-1B.

### Scenario-1A: Client-Server (No Installs Required)

The user does not have to install an RDBMS or front-end tool. But the user must learn the basic features of the front-end tool. In this scenario, the user is granted powerful DBA-privileges on the FREESQLDB database.

### Scenario-1B: Client-Server (No Installs Required)

Similar to the previous scenario, the user does not have to install an RDBMS/front-end tool and must learn the basic features of the front-end tool. Here the user is only granted SELECT privileges on sample tables in the FREESQLDB database.

### Scenario-2: Client-Server (Install Front-End Tool)

The user is granted access to a pre-installed RDBMS on a database server. Here, the user wants to access this RDBMS by installing a front-end tool on her own personal computer.

### Scenario-3: Install RDBMS on a Personal Computer

This is the most complex scenario because the user wants to install an RDBMS (and perhaps also install a front-end tool) on her own personal computer.

### Scenario-4: Access RDBMS in the Cloud (No Installs Required)

Consider this scenario if none of the above scenarios apply to your personal situation.

## **Scenario-1A: Client-Server (No Installs Required)**

Author Comment: The following scenario frequently applied when I presented a live SQL course at a customer's location.

The customer used a client-server system where an RDBMS was installed on a database server, and a front-end tool was installed on client computers in a classroom.

1. I contacted the customer's DBA and asked her to:
  - Assign me an id/password so that I could access the RDBMS.
  - Create a database named FREESQLDB.
  - Grant me DBA-privileges on the new FREESQLDB database. These privileges will allow me to: (i) create the FREESQL sample tables in this database, (ii) grant students access to these tables, and (iii) create other types of database objects (e. g., views and indexes).
2. I contacted the educational coordinator to obtain the id/password for the instructor's client computer in the classroom.
3. Sometimes I had to learn how to find and navigate the SQL-Page for the customer's front-end tool. In this circumstance, I asked the educational coordinator to provide a knowledgeable SQL user to give me a 10-minute tutorial sometime before the class started. I watched this person:
  - Start-up the front-end tool, usually by clicking on some icon.
  - Use the front-end tool to connect to the RDBMS.
  - Navigate to the SQL-Page.
  - Use the MetaData Panel to connect to my FREESQLDB database.

Then I asked the tutor to "let me drive" so that I could create the FREESQL sample tables in the FREESQLDB (as described in Book Appendix-I).

[Also, if I had no prior experience with a customer's front-end tool, I usually undertook two other preliminary learning experiences. (1) I watched a web-based video that demonstrated the customer's front-end tool, and (2) I downloaded and installed the front-end tool to my personal computer. (See Scenario-2.)]

## **Scenario-1B: Client-Server (No Installs Required)**

This scenario is similar to the previous Scenario-1A. Assume you are a student at a school, or you are employed by an organization, where the school/organization allows you to use their client-server system to learn SQL. Again, both the RDBMS and front-end tool have already been installed.

Like the previous scenario, the DBA creates the FREESQLDB database. Then she gives you an user id/password to access the RDBMS and grants you some privileges on the FREESQL database.

Unlike the previous scenario, you send a copy of your CREATE-ALL-TABLES script to the DBA and ask her to create the FREESQL sample tables in FREESQLDB database. Here, the DBA does not grant you powerful DBA-privileges that, in the previous scenario, were given to the SQL instructor. You are only given SELECT privileges on the sample tables. Also, you may or may not be given CREATE TABLE and CREATE INDEX privileges so that you try the optional exercises in this book's Part III on Data Definition and Data Manipulation.

If necessary, obtain an id/password for a client computer. Again, you must learn to use your front-end tool. If possible, find a friendly student/co-worker who will give you a short tutorial similar to that described in the previous scenario. Otherwise, you can watch web-based videos about your front-end tool. Assuming ABC is the name of your front-end tool, do a web-search that looks like: Video on getting started with ABC.

This video should provide information about connecting to an RDBMS. However, in this scenario, connecting to your RDBMS should not be a problem because you are presumably using your employer/school's front-end tool where a connection has already been created. (The following Scenario-2 will comment on connecting to an RDBMS using a front-end tool that you have installed on your personal computer.)

[Potential Problems with Web-Based Videos: Sometimes, finding an informative web-based video can be a challenge. A video may describe an older/newer version of a software product that differs from the version that you will use. (Your web-search should reference the specific version that you want to use.) Also, many videos for front-end tools provide too much information by describing features that transcend the basic execution of SQL statements.]

## **Scenario-2: Client-Server (Install Front-End Tool)**

Downloading and installing a front-end tool should be relatively straightforward. However, a rookie user who has never downloaded and installed any software product on her personal computer should seek help from a more experienced friend.

Again, we assume you are a student at a school, or you are employed by an organization, that allows you to access an RDBMS on a database server. Here, for the sake of convenience, you want to install a front-end tool on your personal computer.

As in the previous scenarios, assume the DBA has:

- Given you an id/password to access the RDBMS.
- Created the FREESQLDB Database. (Alternatively, the DBA may want to store your FREESQL sample tables in some other database.)
- Given you the CREATE TABLE privilege on the FREESQLDB database. Then you will execute the appropriate CREATE-ALL-TABLES script to create the sample tables in the database. (Alternatively, the DBA may create the sample tables for you. Then the DBA will give you SELECT privileges on these sample tables.)

### **Choosing a Front-End Tool**

First, if your school/organization uses the ABC front-end tool, then consider installing the same ABC front-end-tool on your computer.

Alternatively, consider another approach for choosing a front-end tool. All database vendors provide a "default" (unofficial term) front-end tool for their RDBMS. For example:

- IBM provides Data Studio Client for DB2
- ORACLE provide SQL Developer for ORACLE
- Microsoft provides Management Studio for SQL Server

If you know that you will be using the XYZ database system, consider installing the default front-end tool for this XYZ system.

Again, before you install the ABC front-end tool, you are encouraged to search the web for videos about downloading and *installing* the front-end tool." For example, search on: Tutorial for downloading and installing ABC.

## **Download & Install a Front-End Tool**

Preliminary Comment: If you intend to install both an RDBMS and a front-end tool, you may want to initially install the RDBMS because, as described in the following Scenario-3, the RDBMS install process may include the option to also install a front-end tool. This combined install of the RDBMS and front-end tool may be simpler than the separate install of each software product.

Find a download web site: Usually there are multiple web sites that allow you to download a given front-end tool. Do a web search for: "Download ABC" where ABC identifies your desired front-end tool.

Which download site? In general, if ABC is the default front-end tool for the XYZ database system, you should consider downloading this front-end tool from an XYZ web site. For example, if you want to download IBM's Data Studio Client, consider downloading this software from an IBM DB2 web site.

The download and install processes may be bundled together. The download process returns a compressed zip-file which you will unzip, usually by just clicking on it. This process will save one or more files on your computer. You may be asked if you want to start the install process, and you should (most likely) say yes. This should start the install process. Alternatively, one of the download files is a "setup" or "install" file. Clicking on this setup/install file will start the install process. Then you should proceed as with any install process.

Next

- Start up your front-end tool.
- Use your database id/password to connect the RDBMS.
- Navigate to the SQL-Page.

Next, there are some preliminary "first-time, one-time" actions you must take before you can execute SQL statements.

## **First-Time - One-Time Actions**

Create connection to the desired database: The process of creating a new connection involves clicking on a "new connection" button usually found in the MetaData Panel. This causes the front-end tool to display a panel inviting you to enter information describing the new connection. Using this panel, you will assign a name to the new connection (e.g., FREESQLCONN), identify your database (e.g., FREESQLDB), and enter the address of this database. You must obtain this database address from your DBA (or some other user may be able to provide this information).

Details for creating a connection vary among different front-end tools and database systems. Again, you are encouraged to do a web search like: Use the ABC front-end tool to connect to the XYZ database.

After creating a connection to the FREESQLDB, you will see the connection's name on the list of connections. Clicking on this connection-name will connect your front-end tool to the FREESQLDB database.

Verify the new connection: You should verify that your new database connection is working. Click on the connection-name for this database.

After connecting to the database, if the DBA should have created the FREESQL sample tables, execute: `SELECT * FROM PRESERVE`. If this statement fails, contact your DBA.

Otherwise, if the DBA should have given you the CREATE TABLE privilege on the database, verify this privilege by: (i) creating a STUFF table with just one row with one column, (ii) insert a row into this table, (iii) display this table, and then (iv) drop it, as shown below.

```
CREATE TABLE STUFF (COLXXX INTEGER);
INSERT INTO STUFF VALUES (999);
SELECT * FROM STUFF;
DROP TABLE STUFF;
```

Contact your DBA if the above CREATE TABLE statement fails with some kind of "insufficient privileges" message.



### **Scenario-3: Install RDBMS on a Personal Computer**

This scenario offers a *conceptual* overview of installing an RDBMS on a personal computer. In this scenario, we assume a stand-alone environment where you want to install an RDBMS (and maybe a front-end tool) on your personal computer. Installing an RDBMS may require some effort, and unlike the previous scenarios, you may not have a friendly co-worker/student to help you with any problems.

Most application developers and super-users have the background to “muddle through” the RDBMS installation. Also, rookie users may attempt to install an RDBMS and be successful. However, sometimes the install process can become confusing. Therefore, rookies should be prepared to utilize a cloud-based RDBMS as described in the following Scenario-4.

#### **System Administrator**

We introduce a job title, the *system administrator*, which we have not used before. As already indicated, a database administrator is a person who has full control over one or more databases such as the FREESQLDB Database. The more powerful system administrator has complete control over all RDBMS components, including all databases. The system administrator can create databases, create users, assign ids/passwords to users, and grant/revoke database privileges. In particular, the system administrator can designate a user to be the database administrator for a specific database. (Sometimes, on a small system, the same person serves as the system administrator and database administrator for all databases.)

We comment on the system administrator role because, if you install an RDBMS on your personal computer, you become the system administrator. You have all database privileges.

## **Which RDBMS?**

This book presents SQL statements that are generally compatible with DB2, ORACLE, and SQL Server. These database systems are the most popular systems used in major business organizations. The good news is that IBM, ORACLE, and Microsoft offer *free* versions of their database systems. *These versions have "Express" in their title.*

If your work at an organization or attend a school where ORACLE is used, then you probably want to install ORACLE on your personal computer. Assume your computer is running Windows 10. To get started, you should do a web search that looks like: Tutorial overview for installing ORACLE Express on Windows 10.

Alternatively, you may want to consider installing an open-source RDBMS such as MYSQL or SQLite.

## **Download the RDBMS**

You must find a download site to download an RDBMS before you can install it. Do a web search like: Download XYZ for ZZZ where XYZ identifies the RDBMS, and ZZZ identifies operating system (e.g., Windows 10).

This search will produce multiple hits. Which download site should you choose? In general, if you want to download DB2, you should probably download DB2 from an IBM web site. Likewise, for the other database systems. After selecting and visiting a download web site, click on its Download-Button. This usually returns a compressed zip-file.

Unzip the zip-file. (Usually just click on file.) The unzip process will store multiple files in some folder/directory on your computer. Look for an executable file with a name similar to "setup" or "install".

## Install the RDBMS

Start (click on) the setup/install file. This action might not directly start the install process. Instead, it may start an installer-program which does a few things before the installation process begins. This installer-program may ask you to:

- Give up your email address.
- Accept a license agreement.
- Optionally join some vendor organization.
- Designate a folder/directory for the RDBMS files.

After the installation process starts, you may have to wait some time for it to finish. During this process, you will be asked some questions and be presented with some important information that you should remember. These questions and returned information may include the following.

Id/Password: The installation process will ask you to designate an id/password. With this id/password you become the *systems administrator*. Do NOT lose this id/password. A lost id/password will require a re-install.

Connection Information: The installation process will provide the information used by a front-end tool to connect the RDBMS. Again, do not lose this information.

Optionally, Install the RDBMS Vendor's Front-End Tool: In addition to installing the RDBMS, the installation process may ask if you want to want to install the database vendor's front-end tool. For example, when installing DB2, you may be asked if you want to install Data Studio. The front-end tool could be large because its functionality transcends the basic execution of SQL statements. Therefore, you might want to respond "no" and install some other front-end tool. However, one reason to install the vendor's front-end tool during the RDBMS installation is that the connection is automatically setup. Also, you may want to explore the front-end tool's advanced features.

Automatic Start of RDBMS: The installation process may ask if you want to immediately start the RDBMS so that you can get right to work. (Most likely answer "yes.") Also, the installation process may ask if you want to keep the RDBMS running as a background process after you exit from your front-end tool. (Most likely respond "yes," unless you know your utilization of the RDBMS will be infrequent.)

Vendor's Sample Database & Tables: The RDBMS installation process may ask if you want to create the vendor's sample database including its sample tables. You could respond "no" because you intend to create the FREESQLDB database and store the FREESQL sample tables in this database. Nevertheless, you might also want to create the vendor's sample tables because these tables are referenced in the vendor's SQL reference manuals.

Careful! Some of the FREESQL sample tables have very common table-names (e.g., EMPLOYEE, DEPARTMENT, PROJECT); and the vendor's sample database may have one or more tables with the same table-names. Therefore, to avoid problems, we recommend storing the vendor's sample tables and the FREESQL sample tables in different databases. This is not difficult. After an installation that creates the vendor's sample database, you can create the FREESQLDB database, connect to this database, and create the FREESQL tables in this database.

### **After Installing the RDBMS**

Start your front-end tool and connect to the RDBMS using the system administrator's id/password that was specified during the installation process.

Next, you will want to create the FREESQL sample tables as described in Book-Appendix-I. Before doing this, you can (optionally) create the FREESQLDB database to hold these sample tables, and then connect to this database. Otherwise, if you do not explicitly create these tables in the FREESQLDB database, the system will create them in some other "default" or "system" database. This may not be a problem unless another table with the same name has already been created in same default/system database. For this reason, we recommend storing all the FREESQL sample tables in its own database.

## Scenario-4: Access RDBMS in the Cloud

Assume you do not have access to any RDBMS on your employer/school's computer, and assume you do not (yet) have the technical ability to install an RDBMS on your personal computer. Then, your next option is to utilize an RDBMS in the cloud.

Some major cloud web-sites (e.g., IBM, ORACLE, AWS) provide free access (for a trial-period) to open source and commercial RDBMS systems. Although accessing one of these databases still requires some getting-started effort, you can obtain help from the site's chat room if necessary. Below, we describe some of the typical getting-started steps to access a cloud-based RDBMS.

1. Find a cloud-site: If you want access to the XYZ database system in the cloud, do a web-search like:

Free or trial access to XYZ in the cloud

2. After you find and access the desired web-site, look for a link that takes you to a list of free offerings. This list will include multiple computing resources, including the desired RDBMS. Clicking on a "free/trial-access" link should initiate the following step.
3. You will be asked to become a recognized user of the cloud-site by entering your name, email address, etc. Also, you might be asked for credit card information in case you exceed the trial period's time limit, or you store too much data in the database. (This will not a problem when storing the very small FREESQL sample tables). Then you will be asked to specify an id/password that you will use to access the cloud-site.
4. A menu of computing resources will be displayed. You should select the RDBMS that you want to use.
5. Information about the RDBMS will be displayed. This RDBMS is usually a *virtual* RDBMS that has most of the features found in a real RDBMS. You will be asked to specify another id/password to be used to access this RDBMS. This second id/password will give you powerful privileges over the virtual RDBMS. You should be able to execute all SQL statements found in The Free SQL Book.

6. Use your database id/password to access your RDBMS. The system will respond by displaying multiple options that include database administration actions, application programming actions, and executing SQL statements.
7. Select the option that allows you to execute SQL statements. This should initiate a front-end tool. Navigate to the SQL-Page that should display a SQL Panel, Result Panel, and maybe a Metadata Panel.

Note: Your web browser becomes your front-end tool that is controlled by the cloud web-site. However, you may be able to use another front-end tool that has been installed on your personal computer to connect to a cloud-based RDBMS.

8. Optionally, verify that you have successfully accessed the RDBMS by creating a STUFF table with one column, inserting one row, displaying, and then dropping this table, as illustrated below.

```
CREATE TABLE STUFF (COLXXX INTEGER);
INSERT INTO STUFF VALUES (999);
SELECT * FROM STUFF;
DROP TABLE STUFF;
```

9. Assuming the above statements do not fail, you can create the FREESQL sample tables as described in Book-Appendix-I.

Finally: Note that, although using a cloud database requires some getting-started effort, you do not have to install a front-end tool and RDBMS. Hence you can quickly jump into learning SQL. However, because the cloud-site's trial period may not be very long, you may eventually want to gain access to some other RDBMS using one of the previously described methods.

## **Book-Appendix-III**

### **Summary of Chapter Appendices**

This book's chapter appendices, found at the end of some chapters, introduced topics that are *optional* within the context of a *tutorial* introduction to SQL. Each chapter appendix presented by-the-way commentary on topics pertaining to the SQL statements presented in the corresponding chapter.

This Book-Appendix-III organizes and summarizes these optional topics. It will also present some additional material not found in the chapter appendices. Before proceeding, we should revisit the difference between the terms "database model" and "data model" as used in this book.

**Database Model:** In this book we referred to Codd's Relational Model as a "database" model, not a "data" model. IBM, ORACLE, Microsoft, and other database vendors based their database systems (e.g., DB2, SQL Server) on Codd's Relational (*Database*) Model.

**Data Model:** Database analysts/designers formulate conceptual and logical data models to describe specific *applications*. For example, Figure 13.9 illustrated a DEPARTMENT-EMPLOYEE-PROJECT conceptual data model, and Figure 13.10 illustrated the corresponding logical data model.

**Post-Relational Database Models:** Codd proposed his database model over 50 years ago! Thereafter, database researchers have proposed many other database models, which we collectively refer to as *Post-Relational Database Models*. (The following Book-Appendix-IV will identify some of these post-relational database models.)

## History: Codd's Relational Database Model

Appendix 1B presented a very incomplete description of Codd's Relational Database Model. Below we review and add a little more substance to this description.

Codd's Relational Database Model specified three major components: (1) Data Structure, (2) Language, and (3) Integrity.

**1. Data Structure:** Appendix 1B noted that Codd defined a relational database as a set of sets where each set is a relation (table). The elements in each relation are n-tuples (rows) consisting of attributes (column values).

Appendix 1B did not mention that an attribute is defined in terms of another set called a "domain." For example, consider the CHECK-clause for the JCODE column shown in the CREATE TABLE statement for the TESTEMP table (Figure 13.2).

```
JCODE INTEGER NOT NULL CHECK (JCODE IN (1,3,7,11))
```

Within the Codd's Model, a domain is a set of values that is assigned a name (e.g., VALID\_JCODE) as shown below.

```
DOMAIN VALID_JCODE = {1,3,7,11}
```

After defining the VALID\_JCODE domain, the JCODE attribute could be specified as:

```
JCODE VALID_JCODE
```

Specifying this domain is useful if other attributes also contain VALID\_JCODE values (even if these attributes are not named JCODE).

**2. Language:** Appendix 1B noted that Codd proposed two languages, the relational calculus and the relational algebra. This book illustrated that SQL inherited some features from both languages.

**3. Integrity:** Appendix 1B was silent about database integrity. However, Codd's Relational Model did include primary-key, foreign-key, and NOT NULL integrity constraints. Codd's model also included "DOMAIN integrity constraints". For example, assume some user attempted to insert a row into the TESTEMP table with a JCODE value of 5. The system would detect that JCODE is defined on the VALID\_JCODE domain, and this domain does not include 5. Hence, the system would reject this insert operation.



## **History: Codd's Normalization Theory**

In addition to defining his Relational Model, Codd also proposed formal criteria for the specification of a "good" table design. These criteria were defined in his *Normalization Theory*. This appendix, including the chapter appendices, is silent about this theory. However, without describing normalization theory per se, Appendix 16A made *some informal observations* about tabular design that were derived from this theory.

[Codd's normalization theory defined multiple levels of "normal forms." These were 1<sup>st</sup> Normal Form, 2<sup>nd</sup> Normal Form, 3<sup>rd</sup> Normal Form, and Boyce-Codd Normal Form. Other researchers added 4<sup>th</sup> Normal Form and 5<sup>th</sup> Normal Form. Book-Appendix-V identifies two textbooks that present normalization theory.]

In Appendix 16A, we noted that all descriptive (non-key) columns within a normalized (good) table describe just one type of object/entity. Notice that all descriptive DEPARTMENT columns (DNAME and BUDGET) only describe departments; and, all descriptive EMPLOYEE columns (ENAME and SALARY) only describe employees. Therefore, we noted that both the DEPARTMENT and EMPLOYEE tables are normalized.

Then we noted that the DNEMPLOYEE table was de-normalized (not so good) because its descriptive columns describe different object/entity types. Within the DNEMPLOYEE table, some columns (DNAME and BUDGET) describe departments whereas other columns (ENAME and SALARY) describe employees. Appendix 16A showed that this "mixing of apples and oranges" can lead to problems. Hence, in most (but not all) circumstances, database designers do not create de-normalized tables.

Appendix 19C reconsidered de-normalized tables in the context of a LEFT OUTER JOIN operation.

The following commentary on Logical Database Design will associate normalized tables with the database design methodology described in Appendix 13B.

## Database Analysis, Design, and Implementation

Database Design is a large topic that cannot be covered in a few small appendices. Appendix 13B only introduced core concepts and presented a graphical overview (Figure 13.9) of a typical methodology for Database Analysis-Design-Implementation.

Working backwards, Implementation involves the coding of CREATE TABLE statements. Implementation is not difficult because it is derived (in a cookbook manner) from the **Logical Data Model** produced by the preceding Design process. This Logical Data Model was, in turn, derived (in a cookbook manner) from a **Conceptual Data Model** produced by the preceding Analysis process. (Formulating a Conceptual Data Model is another important and very challenging process that is beyond the scope of this book.)

Below we review three observations about Logical Data Models.

1. A detailed logical data model (e.g., Figure 18.4) contains practically all information needed to code CREATE TABLE statements. This allows design tools to automatically generate CREATE TABLE statements. For prototyping and testing purposes, these CREATE TABLE statements can be executed "as is," and sample data can be inserted into these tables. However, within a production environment, the DBA may include other efficiency related clauses in the CREATE TABLE statements before executing them. These clauses were not presented in this book.
2. All tables in our MTPCH database are normalized. In general, any collection of tables derived from a well-designed logical data model will produce normalized tables. We note that learning Codd's Normalization Theory can help analysts/designers evaluate the "goodness" of each table's design.
3. Finally, from a know-your-data perspective, the most important observation is that *a logical data model can help users formulate correct SELECT statements*, especially when a SELECT statement references multiple tables. This explains why the logical data model for the MTPCH Database (Figure 18.4) was frequently referenced in our discussion of sample queries throughout Chapters 18-28.

## **Efficiency: Database Indexes**

Database indexes were the only *physical* (under-the-hood) database structures described in this book.

Appendix 1A introduced the system's utilization of a database index to directly access rows identified by a WHERE-clause. An analogy was drawn between a database index and an index of topics located at the end of a history book.

Appendix 2A described how the system could use an index to satisfy an ORDER BY clause. This appendix also introduced overall cost-benefit tradeoffs associated with database indexes.

Appendix 3A described how the system could use an index to satisfy the DISTINCT keyword.

Appendix 4A introduced query optimization. Examples illustrated how the optimizer would consider selectivity to decide to use or not use an index to satisfy a WHERE-clause.

Appendix 8A described how an index-only search could satisfy some query objectives.

Chapter 14 introduced the CREATE INDEX statement. Appendix 14A presented some general guidelines for index design, including the design of composite indexes.

Appendix 17A observed that an index can improve the efficiency of a Nested-Loop join-operation.

## **Efficiency: Query Optimization**

Appendix 4A introduced query optimization by drawing an analogy with self-driving cars. This appendix introduced a high-level overview of the optimizer's "thought process" within the context of a SELECT statement that accessed a single table.

Appendices 4B and 4C introduced optimizer query rewrite within the context of WHERE-clauses that specified compound-conditions with Boolean Connectors (AND, OR, NOT).

Appendix 6A described a historical scenario where, unfortunately, the user was required to code a do-it-yourself query rewrite.

Appendix 18A said more about optimization within the context of inner-join operations. It described two join-methods (Match-Merge and Nested-Loop); and it described the importance of join-sequence in the context of joining three or more tables.

Appendices 23A and 25A described optimizer query rewrite within the context of join-operations, regular Sub-SELECTs, and correlated Sub-SELECTs.

Appendix 24B described three scenarios where the optimizer referenced dictionary statistics that were: (i) inaccurate, (ii) approximately accurate, or (iii) 100% accurate.

Appendix 28C described how a user can ask the system to generate the explanation of an application plan.

Appendix 28D described how a user can code a "hint." A hint asks the optimizer to take some action that presumably helps it generate a more efficient application plan.

After reading these appendices, you should appreciate that:

- SQL is a declarative language. Your SQL code describes "what" you want to do; the optimizer decides "how to" do it.
- The optimizer utilizes the relational algebra and Boolean Logic to construct an application plan.

## **Efficiency: Tuning a SELECT Statement**

Appendix 28E presented a general method for tuning such a SELECT statement. If you have to tune a SELECT statement, one of two things went wrong.

1. Your optimizer generated a sub-optimal application plan.
2. There was a problem with the physical database design as it relates to your SELECT statement.

Appendix 28E addressed both circumstances.

## Book-Appendix-IV

### Post-Relational Database Systems & NoSQL

At the end of Appendix 28 (Tunning SELECT Statements), we stated that: (1) good physical database design, plus (2) very smart optimizers, plus (3) blazingly fast data storage technology imply that future application developers should not encounter many SQL tuning problems. We also noted that this statement applied to traditional business information systems (e.g., the MTPCH database) which have the following characteristics.

- The data is *structured*, and it fits nicely into “flat” tables.
- The data is *not extremely large*. Row length is short because most column data-types contain short values (numbers, dates, and fixed-length character-strings) plus a few longer variable-length character-strings. Some tables have millions of rows, a few tables have billions of rows, but only a very few tables exceed a trillion rows.

These observations may not apply when we step away from traditional database applications to consider applications where data is not structured, is extremely large, and resides on some unknown computer located anywhere in the world (e.g., the Web).

Consider unstructured data that do not *conveniently* fit into *traditional* relational database tables. This data includes photographs, maps, videos, engineering drawings, documents, textbooks, web pages, email messages, text messages, tweets, etc. Some of this data is partially structured (e.g., document) and is called “semi-structured.” Post-Relational database systems have been designed to store and query this unstructured and semi-structured data.

## Abbreviated History of Post-Relational Database Systems

**Object-Oriented Database Systems (OODBMS):** In the 1970's, the maturity of object-oriented programming languages (e.g., C++) encouraged IT professionals to design object-models with object types/classes. In the 1980's, database researchers proposed object-oriented database management systems that offered many of the goodies associated with object-oriented programming languages.

One goal was to use object types/classes to store unstructured and semi-structured data. An object-oriented database system could *directly* store and query a large complex object. Compared to a relational database, this was more convenient than decomposing a large object into smaller components to facilitate storage within multiple database tables, and subsequently coding a multi-table join-operation to reconstruct the large object.

**Object-Relational Databases:** As object-orientation gained traction in the database world, relational database vendors (IBM, ORACLE, Microsoft) began to include OO features in their database products. This OO functionality can be implemented by storing unstructured or semi-structured objects in a column specified as a BLOB (Binary Large Object) data-type or a CLOB (Character Large Object) data-type.

These object-relational systems also provided a CREATE TYPE statement to create user-defined abstract data-types. (Appendix 10.5A noted that SQL's DATE data-type is an example of a built-in abstract data-type.) Today, all major relational database systems are really hybrid systems that include relational and object-oriented functionality.

**AI and Deductive Databases:** In the late 1970's database researchers began to apply deductive logic to database data. They developed Datalog to query a *deductive database*. Datalog was derived from PROLOG (PROgramming in LOGic), a logic-based declarative language that was developed in the early days of research into artificial intelligence.

Today, relational database systems utilize artificial intelligence to improve query optimization. These systems also offer built-in data mining functions.

**Temporal Databases:** Relational and non-relational data models have been proposed to address the inherent complexity of date-time processing. Within the relational world, one temporal-relational model created three-dimensional tables where the third dimension captured historical data.

On one prototype system, the SELECT statement was enhanced to include a WHEN-clause to specify a time-travel query that retrieved historical data. For example, the following statement displayed the number and name of all employees who worked in Department 30 anytime between 2000 and 2005.

```
SELECT ENO, ENAME
FROM EMPLOYEE
WHERE DNO = 30
WHEN BETWEEN 01-01-2000 AND 12-31-2005
```

If a WHEN-clause was not specified, the system would only display data about current employees.

Today, all major relational database systems support some form of temporal database functionality.

[Chapter Appendix 10.5A says more about Temporal Databases.]

### **And then came the Web**

**XML Databases:** Some application developers want to store XML documents in a database and query these documents. These developers have two basic options:

- (1) Store the XML documents in a "native" XML database (e.g., BaseX, Berkley DB), and use the XQUERY language to query these documents.
- (2) Within a relational database, store the XML documents in a column defined as an XML data-type. Then use XQUERY-like built-in functions to query the XML documents.



## **NoSQL Databases**

Beyond storing unstructured and semi-structured data, some major web-based organizations (e.g., Google, Amazon) wanted to store and query data that is: (1) outrageously large (Big Data) and (2) distributed across the world. Also, some scientific applications must store and query very large amounts of data that are generated by sensor devices. These requirements led to the development of NoSQL databases. NoSQL originally meant "No SQL." Today, most NoSQL databases fit the "**Not Only SQL**" interpretation of NoSQL, meaning that SQL is used along with a non-relational language.

To satisfy extreme data requirements, some web-based organizations started from scratch and built their own "home grown" NoSQL database systems. These systems were based upon non-relational data models that could be considered to be special-purpose data models. The NoSQL literature describes four such models: wide-column, key-value pairs, document, and graph.

Today, NoSQL has become common within the database world. Independent software vendors, along with the major commercial relational database vendors (e.g., IBM, ORACLE, Microsoft), sell NoSQL products. Most of these products provide some method to interact with a relational database.

Currently, all NoSQL systems have limitations when compared to conventional relational database systems. For example, most NoSQL languages are not declarative, and most NoSQL databases do not support ACID level transactions (as described in Appendices 29A and 29B).

### **Concluding "Philosophical" Questions**

Theory Question: Is there a single database model that can effectively represent and process all kinds of data? If yes, what is it? This is the holy grail of database theory.

Practical Question: Are there a small number of database models that can collectively support all kinds of database applications? If yes, what are these models, and how can these models be integrated?

[A comprehensive overview of topics mentioned in this appendix can be found in the C. J. Date textbook and the Elmasri & Navathe textbook referenced in the following Book-Appendix-V.]

## **Book-Appendix-V**

### **Abbreviated Bibliography**

Preliminary Comment: Unlike all other database textbooks, this book does not contain many references. The primary reason is that most significant references can be found in two books by (i) C. J. Date and (ii) Elmasri & Navathe. Both books are described below.

#### **Database Concepts and Facilities**

The following excellent textbooks cover many database topics beyond SQL. These topics include the Relational Model, logical and physical database design, normalization theory, query optimization, distributed databases, concurrency and recovery, and post-relational database systems (e.g., Object-Oriented, Temporal, and XML databases).

1. C. J. Date, An Introduction to Database System (8<sup>th</sup>), Addison-Wesley, Reading, MA (2004).

Date has written many other database books. A comprehensive list can be found on the web. Do a web search for: Books by C. J. Date at [www.goodreads.com](http://www.goodreads.com)

2. Ramez Elmasri & Shamkant Navathe, Fundamentals of Database Systems (7<sup>th</sup>), Pearson (2017).

This more recent textbook also covers NoSQL databases.

#### **SQL References**

You should not need to purchase any other SQL book that covers the same SQL topics presented in this book. This does not imply the absence of many high-quality SQL books to be found in the marketplace. Instead, we mean that, after getting started with this Free SQL Book, you should be able to understand your SQL reference manuals that describe *all SQL statements* supported by your RDBMS. Also, many authors have published a wide variety of excellent SQL articles on the web.

## Theory

The following book contains all important concepts and references (as of 2009) about SQL and relational database theory.

**C. J. Date, *SQL and Relational Theory: How to Write Accurate SQL Code*, O'Reilly Media, Boston, MA (2009).**

*Buy this book!* As its subtitle ("How to Write Accurate SQL Code") indicates, the primary objective of this book, like all other SQL books, is to write correct SQL. However, unlike all other SQL books, *this book presents relational theory as the foundational cornerstone for coding correct SQL.* (Whereas this Free SQL Book merely presents snippets of theory within a few optional appendices.)

In his book, Date describes Codd's original Relational Model. Then he presents revisions and extensions to this model, followed by a summary of the current version of this model. Throughout his book, Date presents many SQL examples to demonstrate that "theory is practical."

Date also makes negative comments about SQL's failure to be completely faithful the Relational Model. Consider the attention-grabbing second sentence in his Preface where he states that:

**"SQL is hard to use: It's complicated, confusing, and error prone - much more so, I venture to suggest than its apologists would have you believe."**

Martyn Comment: I confess to being one of those SQL apologists. In Chapter 0, with the intention of motivating the reader, I may have told a little white lie when I stated that "SQL is easy."

In his Appendix A, Date concludes by saying:

**"SQL is incapable of providing the kind of firm foundation for future growth and development."**

Date's rationale is thought-provoking. By reading his book the reader gains a deeper understanding of SQL.

## Two Other Free SQL Textbooks

1. Developing Time-Oriented Database Applications in SQL by Richard Snodgrass, Morgan Kaufmann (1999).

This book offers the reader a solid foundation for using SQL to work with temporal data. You can obtain a free PDF version of this book by visiting Richard Snodgrass's web site at the University of Arizona (or you can purchase this book from Amazon and other on-line book sellers).

*-- Thanks, and regards to Richard!*

2. DB2 SQL Cookbook. This was the first free on-line comprehensive textbook about SQL (DB2 version). It was written by Graeme Birchall who has apparently disappeared from the world of the Web. Currently, Rodney Crick is maintaining this book. Krick writes:

The last version of the book I am aware of was published in 16 August 2011, based on version 9.7 of Db2 LUW. In the past years I've googled sometimes to check if there was a new version of the book and if someone decided to maintain it. I didn't find any new version and, as far as I know, Graeme Birchall deleted everything he had (his homepage, where the book was published and every link that he maintained)... Because I've learned a lot and I found the book very good to help people that are starting with SQL, I decided to take the contents of the book as they were in the last published version and use it to initiate a new version of the book.

An HTML version of this book can be found at [db2-sql-cookbook.org](http://db2-sql-cookbook.org). This web site will direct you to a PDF version of the book.

Martyn Comments: (1) This book is especially useful for DB2 users because it covers DB2 specific functions, DB2 temporary tables, and many other DB2 SQL features not addressed in this Free SQL Book. (2) I did a web search on "where is Graeme Birchall?" and got some interesting hits on a person involved with astronomy and urban kayaking. I think this person could be the same "missing" Graeme Birchall. Anyways,

*-- Thanks, and regards to Graeme (wherever you are)!*