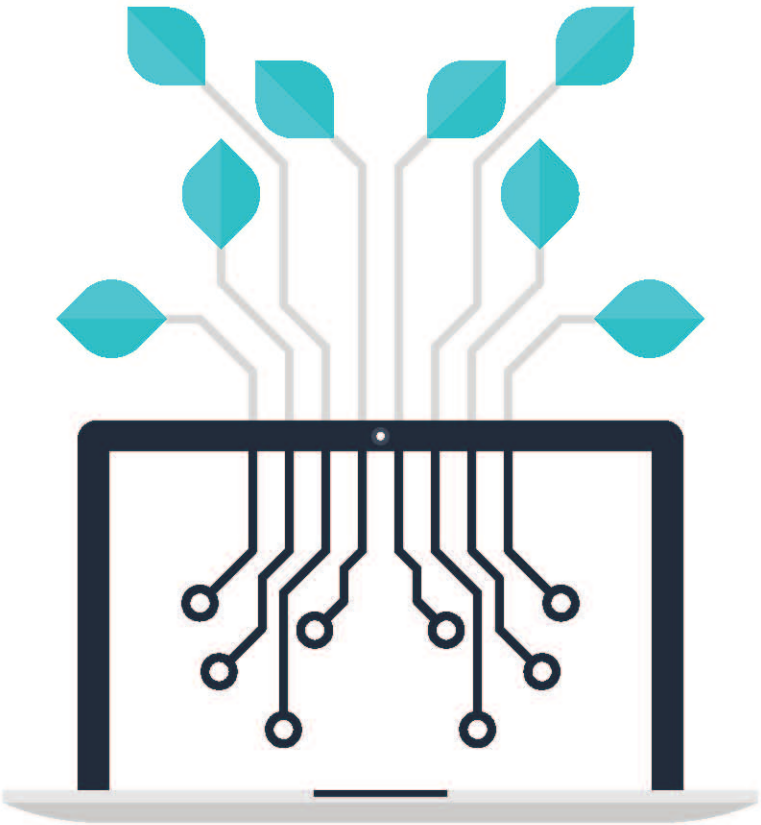
SECOND EDITION

Database Systems Using Oracle®

A Simplified Guide to SQL and PL/SQL



Nilesh Shah

ALWAYS LEARNING

PEARSON

Database Systems Using Oracle

A Simplified Guide to SQL and PL/SQL

Second Edition

(Updated for Oracle9i)

Nilesh Shah

Associate Professor, CIS Department
DeVry University, North Brunswick, New Jersey
Senior Systems Analyst, IT Department
Monroe College, Bronx, New York

PEARSON

ISBN 978-93-325-4972-2
eISBN 978-93-325-7309-3

Head Office: A-8 (A), 7th Floor, Knowledge Boulevard, Sector 62, Noida 201 309, Uttar Pradesh, India.
Registered Office: 4th Floor, Software Block, Elnet Software City, TS-140, Block 2 & 9, Rajiv Gandhi Salai, Taramani, Chennai 600 113, Tamil Nadu, India.
Fax: 080-30461003, Phone: 080-30461060
www.pearson.co.in, Email: companysecretary.india@pearson.com

To my two special boys,
Naman, 12
(an honor student and an excellent basketball player)
and
Navan (Jinku), 6
(for reading my book at the age of 4)

To my wife,
Prena
(for her support)

To my parents,
Dhiraj and Hansa
(for their sacrifice)

This page is intentionally left blank

Contents

Foreword

Alex Ephrem, Ph.D.

Computer science educators and IT administrators are—and, traditionally, have been—faced with a common problem. In an industry characterized by rapid and dramatic changes, the manager must determine how he or she can maintain state-of-the-art skills among the IT staff. In a similar vein, the educator, must be able to judge how students can be best prepared to work as professionals in a field that may have gone through revolutionary transitions between the time that student first entered college and the time that he or she graduates.

Certainly, a technical education must incorporate a strong foundation in the core concepts of operating systems, file or database structure, computer architecture, and general programming theory. The difficulty arises when the educator seeks to select an application or a development platform to use for introducing these concepts and for providing students with practical, functional, and marketable hands-on skills. As the ones responsible for such preparation, we often look for a package that not only will give students the most vivid demonstration of the theoretical concepts we are attempting to portray but will also offer students an opportunity to use that knowledge almost immediately in a variety of environments. In addition, we seek packages that are in heavy and common demand, with a "track record" of success, reliability, and longevity.

The area of relational database management systems (RDBMS) is crowded with a vast number of quality RDBMS products. Only one, however, addresses the many concerns the educator has for students. That product, of course, is Oracle. It has been on the market for more than 20 years, and it holds a major portion of the market share, which accounts for Larry Ellison's position among the world's wealthiest men and Oracle's position among the largest global corporations. There are versions of the product for nearly every hardware platform, from personal computers through minicomputers and supercomputers, and for operating systems from DOS, Windows, and Linux through MVS, OS/400, PICK, and the multitudinous flavors of Unix. Of all the RDBMS systems available, Oracle is the one the student is most likely to encounter on the job—and the one in which employers most eagerly seek expert applicants and employees.

From an educator's perspective, Oracle, as a truly relational database, incorporates virtually all the relational operations that any database theory course must encompass. This allows the student to actually see the results of such operations.

Similarly, the instructor retains the flexibility to design customized exercises that combine one, several, or all the standardized operations discussed in lectures on relational theory. In addition, Oracle's ease of use allows the instructor to concentrate on the purpose of the course rather than on how to utilize the RDBMS software.

Computer science has never been one of the "pure" sciences, concerned solely with theoretical constructs. Like engineering, its concern and preparation are directly and fully aimed at the practical application of knowledge. In today's economy, a comprehensive grasp of database design, use, and implementation is a basic skill required of IT professionals, and as an educator and CIO, it is my opinion that any university course or professional training seminar focusing on database concepts that does not also provide the student with at least an introduction to Oracle is deficient.

Nilesh Shah's *Database Systems Using Oracle* includes everything that both the educator seeking to present essential database concepts and the student wishing to learn Oracle, either in a guided classroom or an independent study approach, would need. It is organized so that the beginner is presented with enough background to quickly progress to a functional mastery of the more complex material, and the progression of topics and degree of coverage are comprehensive enough to meet the needs of the demanding professional. In recognition of the necessity to go beyond theory, numerous hands-on exercises are included, and examples are given of features, such as Web interfaces to Oracle tables, from Oracle's most recent versions.

For those of you who are first entering the world of RDBMS, the Shah text is a reliable vehicle that will assist you in meeting your objectives and assure that you finish with confidence in your ability.

To the database professional, the Shah text will give you a reference and guide that you will use frequently.

To the university professor or professional trainer, the Shah text has given you a uniquely flexible educational tool. With it, you can develop, plan, and implement your course in a manner that will give your students the necessary academic understanding of core database concepts while simultaneously teaching them a hugely marketable skill within the computer industry.

As a professor of computer science at Monroe College as well as that college's Senior Systems Analyst, Nilesh Shah has demonstrated the dual abilities of fully comprehending the broad range of complexities involved in database management as well as the gift of presenting complex subject matter in an easily understandable format. To the reader's benefit, these abilities come across clearly in the text before you.

This new and expanded edition provides even greater depth of instruction in the use of those elements that have resulted in the Oracle's huge popularity. Additionally, it adeptly covers a range of Oracle's newest features and capabilities. Few texts can genuinely be considered both a useful learning and reference tool for the experienced professional as well as a comprehensive and understandable introduction for the database beginner. With this text, Dr. Shah appears to have achieved that difficult union.

Alex Ephrem, Ph.D.
Senior Vice-President and Chief Information Officer
Monroe College

Foreword

John W. Weber

Rarely does a text come along in the IT field that effectively blends the theoretical framework of a topic with its practical application. Either a text focuses on purely theoretical concepts that leave students lacking in real-world application, or it covers the "how-to" of a tool without stressing the theoretical foundation so important to the students' ultimate mastery of the tool.

In the field of relational database management systems (RDBMS), one book stands out in its ability to blend theory and application. That book is Nilesh Shah's *Database Systems Using Oracle*. This book covers key foundational concepts, such as the relational database model, entity–relationship modeling, relationship types, exposure to both relational algebra and relational calculus concepts, and normalization. This text also effectively provides the necessary application to creating, maintaining, and querying a database through the Structured Query Language (SQL). Professor Shah's book provides students with thorough examples, tables of key commands and functions, data types and their uses, as well as other key information. The appendices provide a good reference for novices and experienced users alike in regards to SQL syntax.

This second edition also provides more extensive coverage of topics that relate to the Oracle9i database environment. Two new topics have been introduced: embedded SQL, and Java/Oracle connectivity through Java Database Connectivity (JDBC) and SQLj. A new chapter that addresses nested queries has also been added, as well as an appendix that introduces object relational database management systems (ORDBMS). In addition, the treatment of various topics has been enhanced since the last edition, particularly in the area of PL/SQL.

Professor Shah's teaching style is particularly well received by his students because of his effective ability to take complex ideas and present them in an understandable manner. His book has been designed in this same style. His explanations are clear, concise, and presented in a way that even the most inexperienced database user can understand. This is further evidenced by the "In a Nutshell . . . " section of each chapter, which succinctly presents each key concept in a series of bullet points.

Similar to his classroom environment, Prof. Shah has also incorporated activities that are crucial to the students' application of the material. Each chapter contains lab activities that allow each student to apply the concepts covered in the

Oracle 9i database environment, thus giving each student practice in using those skills that are in demand in the workplace today.

As an educator, one is not concerned solely if a teacher is teaching but, rather, if the students are learning. Professor Shah is consistently interested in the progress of his students, which is evident in this book. As a user of this book, I am sure you also will find it to be a contributor to your success in your database coursework.

JOHN W. WEBER
Dean, Information Systems Programs
DeVry University

Preface

THE READER

The Relational Database Management System (RDBMS) is the most popular database model today. Oracle Corporation has established the Oracle database product as the prime database package in the world. Structured Query Language (SQL) is the universal query language for relational databases. Programming Language extension to SQL (PL/SQL), an Oracle extension to SQL, brings all the benefits and capabilities of a high-level programming language to the database environment.

This book is designed for use as a primary text in a database course at the college level or as a self-study guide for the information systems or business professional. With its in-depth coverage of relational database concepts, SQL* (Oracle's version of SQL), and PL/SQL, the text serves as an introductory guide as well as a future reference resource. The proper term for Oracle SQL is SQL*, but throughout this text, the term Oracle SQL is simply referred to as SQL. The text can be used in a course that concentrates on database design and utilizes SQL to complement it. It also makes a perfect textbook with which to teach SQL only. Another use of this book is for an advanced database management system course, in which more advanced features of SQL, PL/SQL, connectivity through Java, and database administration are emphasized. In a classroom environment, it is not possible to cover all 16 chapters in one semester. At our campus, we cover Chapters 1 through 9 in the introductory database systems course. In the advanced database course, the SQL portion is reviewed, and then Chapters 10 through 16 are covered. The book serves as a great resource to expand on the topics learned in the classroom.

THE TEXT

The second edition contains more examples, added screen shots, a new chapter on Java with Oracle, new SQL and PL/SQL topics/statements (e.g., MERGE, INSERT ALL, INSERT FIRST, correlated subqueries, CASE, Searched CASE, and INSTEAD OF trigger), and more built-in functions. The first part of the book provides adequate

knowledge of relational concepts and database design techniques to allow students to design and implement accurate and effective database systems. The second part concentrates on the primary nonprocedural relational database language, SQL, which is supported by most relational database software packages. The book primarily concentrates on Oracle9i and points out those features that were not available in previous releases. (In reality, the SQL part of the book can be utilized in Oracle release 7.x, 8, 8i, or 9i.) The third part of the textbook is devoted to the procedural language PL/SQL, which is Oracle's proprietary language extension to SQL. PL/SQL features data encapsulation, error handling, and information hiding, which are typical capabilities of a high-level programming language. The fourth part of the textbook introduces the architecture and administration of Oracle9i as well as connectivity to Oracle from Java.

Throughout the text, the general syntax of SQL and PL/SQL are supplemented by simple examples, screen captures, and illustrations. Each chapter includes a brief summary, exercise questions, and lab activities. The textbook is supported by sample databases—one a typical college's student database with demographic, schedule, and registration information, and the other a corporation's employee database with employees' demographic and job-related data. In most cases, the examples are based on one of the sample databases and the lab activity on the other, to test a student's ability to apply queries in a different scenario. A separate section with a third sample database is included at the end of the SQL portion of the textbook, which summarizes most SQL statements covered in Chapters 3 through 9.

Because the book is primarily designed as a college text, it also includes (exclusively for instructors) answers to the exercise questions as well as SQL queries and PL/SQL blocks for the lab activities. The script to create both sample databases is also included for the instructors.

THE SOFTWARE

Oracle comes in many flavors. At your business or college laboratory, Oracle might be implemented in a Windows, Unix, Linux, Solaris, or Novell Netware environment. The version of Oracle might vary from 8 to 8i to 9i. The beauty of this text is its versatility. The SQL and PL/SQL features covered here work with all versions, and the exceptions are pointed out in the individual topics wherever necessary.

The reader is advised to join the Oracle Technology Network (OTN) at *www.otn.oracle.com*. One of the benefits of being a registered OTN user is access to free downloads of Enterprise and a personal version of Oracle9i software. Oracle support, however, is not free!

USING THE TEXT

This text is designed for sequential reading from Chapter 1 through Chapter 16. If you are familiar with relational database concepts, you may skip the first two chapters.

From my personal experience with students, Chapter 2, on data modeling and normalization, helps students tremendously in designing effective databases. You will need access to a computer system to practice SQL statements and PL/SQL programs from Chapters 3 through 14. The fourth part of the text contains material on the architecture and administration of Oracle and on creating Java applications/applets with connectivity to an Oracle database. Many popular SQL* Plus commands are also covered in Appendix C. Appendix E details what is new in 9i SQL and PL/SQL. The reader must perform exercise questions and labs at the end of each chapter before moving on to the next chapter. Whether a programming language is procedural or nonprocedural, you cannot learn it just by reading about it. You need to practice to master the material.

ACKNOWLEDGMENTS

I would like to thank Kate Hargett and her staff for their support in publishing this text. It is my pleasure to work with a prestigious publishing company such as Prentice Hall. I thank Petra Recter for her help and guidance with the first edition. Good luck with your new position, Petra!

I am also grateful to my employers—DeVry University, North Brunswick, New Jersey; and Monroe College, Bronx, New York. I would like to mention three individuals in particular: Dean Bhupinder Sran at DeVry, who asked me to write my first book when we could not find a suitable text for our database systems course; Dr. Alex Ephrem at Monroe, who supported and encouraged me throughout the text's development process (Dr. Ephrem has the Japanese-language copy of this text on his desk as an exhibit!); and John Weber, Dean of Information Systems at DeVry, for taking the time to write an excellent foreword for the text.

I would like to thank all my students, past and present, for being themselves. Without my students, I would not be in a position to write a book. Their enthusiasm in the classroom, respect toward me, and desire to learn inspired me to take up this project. I would like to single out two students at DeVry; James McClaran, for creating ERD for the sample databases used in this text, and Heillyn Viquez, for retyping almost 200 coding examples from existing screenshots.

I also thank all the reviewers, especially Richard J. Staron, for their honest comments, which enabled me to address deficiencies of the first edition and to create a better text.

Finally, I would like to thank my family for their understanding and patience during the entire process: my 12-year-old son, Naman, an honor student and an excellent basketball player, who possesses the first copy of my book; my 6-year-old son, Navan, a login helper, who wants to move to Philadelphia after the book is finished so he can watch Allen Iverson play basketball everyday; my wife, Prena, for putting up with me; and last but not least, my parents, Dhiraj and Hansa, for the sacrifice they have made in their life by sending their only son to the United States for a better future.

NILESH SHAH

PART 1
DATABASE
CONCEPTS

1

Database Concepts: A Relational Approach

IN THIS CHAPTER . . .

- You will learn about basic database terminology.
- Relational database concepts are covered.
- The Database Management System (DBMS) and its functions are outlined.
- Integrity rules and types of relationships are explained.
- Two theoretical relational languages for data retrieval, relational algebra and relational calculus, are introduced.

DATABASE: AN INTRODUCTION

A database is an electronic store of data. It is a repository that stores information about different "things" and also contains relationships among those different "things." Let us examine some of the basic terms used to describe the structure of a database:

- A person, place, event, or item is called an **entity.**
- The facts describing an entity are known as **data.** For example, if you were a registrar in a college, you would like to have all the information about the students. Each student is an entity in such a scenario.

- Each entity can be described by its characteristics, which are known as **attributes.** For example, some of the likely attributes for a college student are student identification number, last name, first name, phone number, Social Security number, gender, birthdate, and so on.
- All the related entities are collected together to form an **entity set.** An entity set is given a singular name. For example, the STUDENT entity set contains data about students only. All related entities in the STUDENT entity set are students. Similarly, a company keeps track of all its employees in an entity set called EMPLOYEE. The EMPLOYEE entity set does not contain information about the company's customers, because it wouldn't make any sense.
- A **database** is a collection of entity sets. For example, a college's database may include information about entities such as student, faculty, course, term, course section, building, registration information, and so on.
- The entities in a database are likely to interact with other entities. The interactions between the entity sets are called **relationships.** The interactions are described using active verbs. For example, a student *takes* a course section (CRSSECTION), so the relationship between STUDENT and CRSSECTION is *takes*. A faculty member *teaches* in a building, so the relationship between FACULTY and BUILDING is *teaches*.

RELATIONSHIPS

The database design requires you to create entity sets, each describing a set of related entities. The design also requires you to establish all the relationships between the entity sets within the database. The different database management software packages handle the creation and use of relationships in different manners. Depending on the type of interaction, the relationships are classified into three categories:

1. **One-to-one relationship:** A one-to-one relationship is written as **1:1** in short form. It exists between two entity sets, *X* and *Y*, if an entity in entity set *X* has only one matching entity in entity set *Y*, and vice versa. For example, a department in a college has one chairperson, and a chairperson chairs one department in a college. An employee manages one department in a company, and only one employee manages a department.
2. **One-to-many relationship:** A one-to-many relationship is written as **1:M.** It exists between two entity sets, *X* and *Y*, if an entity in entity set *X* has many matching entities in entity set *Y* but an entity in entity set *Y* has only one matching entity in entity set *X*. In such a situation, a 1:M relationship exists between entity sets *X* and *Y*. For example, a faculty teaches for one division in a college, but a division has many faculty members. The relationship

between DIVISION and FACULTY is 1:M. An employee works in a department, but a department has many employees. The relationship between DEPARTMENT and EMPLOYEE is 1:M.

3. **Many-to-many relationship:** A many-to-many relationship is written as **M:N** or **M:M.** It exists between two entity sets, *X* and *Y*, if an entity in entity set *X* has many matching entities in entity set *Y* and an entity in entity set *Y* has many matching entities in entity set *X*. For example, a student takes many courses, and many students take a course. An employee works on many projects, and a project has many employees.

Many times, students find it difficult to determine the type of a relationship. You need to ask the following two questions to make the determination:

1. Does an entity in entity set *X* have more than one matching entity in entity set *Y*?
2. Does an entity in entity set *Y* have more than one matching entity in entity set *X*?

If your answers to both questions are "No," the relationship is a 1:1 relationship. If one of the answers is "Yes" and the other answer is "No," it is a 1:M relationship. If both answers are "Yes," you have an M:N relationship. Later, you will see that the M:N relationship is not easy to implement and is decomposed into two 1:M relationships.

DATABASE MANAGEMENT SYSTEM (DBMS)

The database system consists of the following components (see Fig. 1-1):

- A database management System (DBMS) software package such as Microsoft Access, Visual Fox Pro, Microsoft SQL-Server, or Oracle.
- A user-developed and implemented database or databases that includes tables, a data dictionary, and other database objects.
- Custom applications such as data-entry forms, reports, queries, blocks, and programs.
- Computer hardware—personal computers, minicomputers, and mainframes in a network environment.
- Software—an operating system and a network operating system.
- Personnel—a database administrator, a database designer/analyst, a programmer, and end users.

Data are the raw materials. Information is processed, manipulated, collected, or organized data. The information is produced when a user uses the applications to

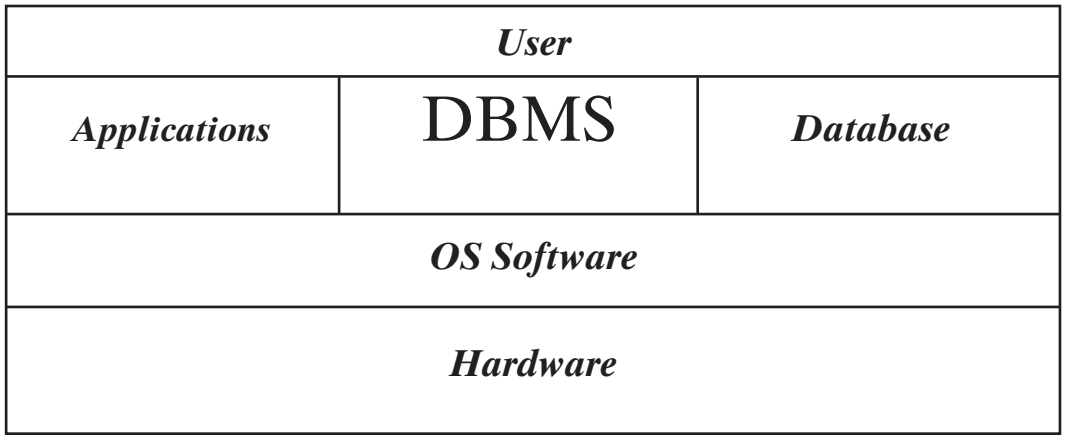


Figure 1-1 Database system.

transform data managed by the DBMS. The database system is utilized as a decision-making system and is also referred to as an information system (IS).

A DBMS based on the relational model is also known as a Relational Database Management System (RDBMS). An RDBMS not only manages data but is also responsible for other important functions:

- It manages the data and relationships stored in the database. It creates a Data Dictionary as a user creates a database. The Data Dictionary is a system structure that stores **Metadata** (data about data). The Metadata include table names, attribute names, data types, physical space, relationships, and so on.
- It manages all day-to-day transactions.
- It performs bookkeeping duties, so the user has data independence at the application level. The applications do not have information about data characteristics.
- It transforms logical data requests to match physical data structures. When a user requests data, the RDBMS searches through the Data Dictionary, filters out unnecessary data, and displays the results in a readable and understandable form.
- It allows users to specify validation rules. For example, if only M and F are possible values for the attribute gender, users can set validation rules to keep incorrect values from being accepted.
- It secures access through passwords, encryption, and restricted user rights.
- It provides backup and recovery procedures for physical security of data.
- It allows users to share data with data-locking capabilities.
- It provides import and export utilities to use data created in other database or spreadsheet software or to use data in other software.

- It enables users to join tables to view information stored in different tables within the database. The user is able to design a database with less redundancy, which means fewer data-entry errors, fewer data corrections, better data integrity, and a more efficient database.

THE RELATIONAL DATABASE MODEL

The need for data is always present. In the computer age, the need to represent data in an easy-to-understand, logical form has led to many different models, such as the relational model, the hierarchical model, the network model, and the object model. Because of its simplicity in design and ease in retrieval of data, the relational database model has been very popular, especially in the personal computer environment.

E. F. Codd developed the relational database model in 1970. The model is based on mathematical set theory, and it uses a **relation** as the building block of the database. The relation is represented by a two-dimensional, flat structure known as a **table.** The user does not have to know the mathematical details or the physical aspects of the data, but the user views the data in a logical, two-dimensional structure. The database system that manages a relational database environment is known as a Relational Database Management System (RDBMS). Some of the popular relational database systems are Oracle9i by Oracle Corporation, Microsoft Access 2000, and Microsoft Visual Fox Pro 6.0.

A table is a matrix of rows and columns in which each row represents an entity and each column represents an attribute. In other words, a table represents an entity set as per database theory, and it represents a relation as per relational database theory. In daily practice, the terms *table, relation*, and *entity set* are used interchangeably.

Figure 1-2 shows six relational tables—PROJ2002, PROJ2003, PRJPARTS, PARTS, DEPARTMENT, and EMPLOYEE. PROJ2002 has three columns and five entities. PROJ2003 contains three columns and four entities. PRJPARTS has three columns and five entities. In relational terminology, a row is also referred to as a **tuple**. It rhymes with couple. In a relational database, it is easy to establish relationships between tables. For example, it is possible to find the name of the vendor who supplies parts for a project.

Each column in a relation or a table corresponds to a column of the relation, and each row corresponds to an entity. The number of columns in a table is called the **degree** of the relation. For example, if a table has four columns, then the table is of degree 4.

It is assumed that there is no predefined order to rows of a table and that no two rows have the exact same set of values. The order of columns is also immaterial, but correct order is used in the illustrations.

The set of all possible values that a column may have is called the **domain** of that column. Two domains are the same only if they have the same meaning and use. ProjNo, PartNo, DeptNo, and EmpNo are columns with numeric values, but their domains are different.

PROJ2002

ProjNo	Loc	Customer
1	Miami	Stocks
3	Trenton	Smith
5	Phoenix	Robins
6	Edison	Shaw
7	Seattle	Douglas

PROJ2003

ProjNo	Loc	Customer
1	Miami	Stocks
2	Orlando	Allen
3	Trenton	Smith
4	Charlotte	Jones

PRJPARTS

ProjNo	PartNo	Qty
1	11	20
2	33	5
3	11	7
1	22	10
2	11	3

PARTS

PartNo	PartDesc	Vendor	Cost
11	Nut	Richards	19.95
22	Bolt	Black	5.00
33	Washer	Mobley	55.99

DEPARTMENT

DeptNo	DeptName
10	Production
20	Supplies
30	Marketing

EMPLOYEE

EmpNo	Ename	DeptNo	ProjNo	Salary
101	Carter	10	1	25000
102	Albert	20	3	37000
103	Breen	30	6	50500
104	Gould	20	5	23700
105	Barker	10	7	75000

Figure 1-2 Relational database tables.

Relational Terminology	File System Terminology
Entity Set or Table or Relation	File
Entity or Row or Tuple	Record
Attribute or Column	Field

Figure 1-3 Terminology comparison.

Terms like tuple and degree are used here because they are relational database terms, but in reality, these terms are not used in workplace.

Figure 1-3 shows a simple comparison between terminology used in relational databases and file systems. Many times, terms are borrowed from the file system for the relational field, and vice versa.

A **key** is a minimal set of columns used to uniquely define any row in a table. If a single column can be used to describe each row, there is no need to use two columns as a key. For example, in PROJ2002, ProjNo uniquely defines each row, and in PARTS, PartNo uniquely defines each row. In PRJPARTS, none of the columns defines each row uniquely by itself. The column ProjNo is not unique, and PartNo is not unique either. In such a table, a combination of columns can be used as a key. For example, ProjNo and PartNo together make a key for PRJPARTS table. When a single column is used as a unique identifier, it is known as a **primary key.** When a combination of columns is used as a unique identifier, it is known as a **composite primary key** or, simply, as a **composite key.**

Sometimes, a more human approach is used to identify or retrieve a row from a table because it is not possible to remember primary key values such as the employee number, part number, department number, and so on. For example, a vendor's name, an employee's last name, a book's title, or an author's name can be used for the data retrieval. Such a key is known as a **secondary key.**

If none of the columns is a candidate for the primary key in a table, sometimes database designers use an extra column as a primary key instead of using a composite key. Such a key is known as a **surrogate key.** For example, columns such as customer identification number, term identification number, or vendor number can be added in a table to describe a customer, term, or vendor, respectively.

In a relational database, tables are related to each other through a common column. A column in a table that references a column in another table is known as a **foreign key.** For example, the PartNo column in PRJPARTS is a foreign key column that references the PartNo column in PARTS.

Figure 1-4 shows typical illustrations showing the notation used for tables in a relational database. The table name is followed by a list of columns within parentheses. The primary key or composite primary key columns are underlined. In Oracle, the primary key, composite key, or surrogate key is defined as a primary key only, and a foreign key in a table can reference a primary key column only.

Oracle uses key words PRIMARY KEY to define a primary, composite, or surrogate key. In Oracle tables, only primary and foreign keys are defined. Secondary key is not part of Oracle's table structure, but it is a column used in search operations. Later, you will learn to use Oracle's Data Dictionary to find table keys and other table information.

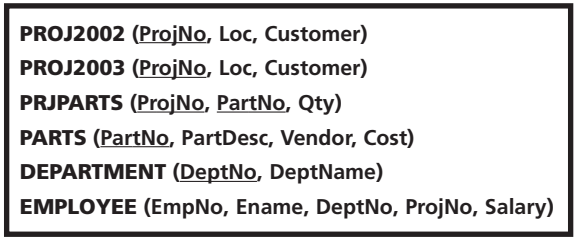


Figure 1-4 Notation used for tables.

INTEGRITY RULES

In any database managed by an RDBMS, it is very important that the data in the underlying tables be consistent. If consistency is compromised, the data are not usable. This need led the pioneers of database field to formulate two integrity rules:

1. ***Entity integrity:*** No column in a primary key may be null. The primary key provides the means of uniquely identifying a row or an entity. A null value means a value that is not known, not entered, not defined, or not applicable. A zero or a space is not considered to be a null value. If the primary key value is a null value in a row, we do not have enough information about the row to uniquely identify it. The RDBMS software strictly follows the entity integrity rule and does not allow users to enter a row without a unique value in the primary key column.
2. ***Referential integrity:*** A foreign key value may be a null value, or it must exist as a value of a primary key in the referenced table.

Referential integrity is not fully supported by all commercially available systems, but Oracle supports it religiously! Oracle does not allow you to declare a foreign key if it does not exist as a primary key in another table. It allows you to leave the foreign key column value as a null. If a user enters a value in the foreign key column, Oracle cross-references the referenced primary key column in the other table to confirm the existence of such a value.

It is not a good practice to use null values in any non–primary key columns, because this results in extra overhead on the system's part in search operations. The programmers or query users have to add extra measures to include or exclude rows with null values. In certain cases, it is not possible to avoid null values. For example, an employee does not have a middle initial, an employee is hired but does not have an assigned department, or a student's major is undefined. In Oracle, a default value can be assigned to a column, and a user does not have to enter a value for that column.

THEORETICAL RELATIONAL LANGUAGES

E. F. Codd suggested two theoretical relational languages to use with the relational model:

1. ***Relational algebra,*** a procedural language.
2. ***Relational calculus,*** a nonprocedural language.

Third-generation high-level compiler languages can be used to manipulate data in a table, but they can only work with one row at a time. In contrast, the relational languages can work on the entire table or on a group of rows. The multiple-row

manipulation does not even need a looping structure! The relational languages provide more power with a very little coding. Codd proposed these languages to embed them in other host languages for more processing capability and more sophisticated application development. In the database systems available today, nonprocedural Structured Query Language (SQL) is used as a data-manipulation sublanguage. The theoretical languages have provided the basis for SQL.

Relational Algebra

Relational algebra is a procedural language, because the user accomplishes desired results by using a set of operations in a sequence. It uses set operations on tables to produce new resulting tables. These resulting tables are then used for subsequent sequential operations. In Oracle, all operation names are not actually used as programming terms, and most of these operations do not create a new resulting table, as shown in the following examples using relational algebra.

The nine operations used by relational algebra are:

1. Union.
2. Intersection.
3. Difference.
4. Projection.
5. Selection.
6. Product.
7. Assignment.
8. Join.
9. Division.

Union. The union of two tables results in retrieval of all rows that are in one or both tables. The duplicate rows are eliminated from the resulting table. The resulting table does not contain two rows with identical data values. There is a basic requirement to perform a union operation on two tables:

- Both tables must have the same degree.
- The domains of the corresponding columns in two tables must be same.

Such tables are said to be ***union compatible***. In mathematical set theory, a union can be performed on any two sets, but in relational algebra, a union can be performed only on union-compatible tables.

Suppose we want to see all the projects from years 2002 and 2003. We obtain it by performing a union (∪) on the PROJ2002 and PROJ2003 tables as given in Figure 1-2.

If we call the resulting table TABLE_A, the operation can be denoted by

TABLE_A = PROJ2002 ∪ PROJ2003

TABLE_A

ProjNo	Loc	Customer
1	Miami	Stocks
2	Orlando	Allen
3	Trenton	Smith
4	Charlotte	Jones
5	Phoenix	Robins
6	Edison	Shaw
7	Seattle	Douglas

Intersection. The intersection of two tables produces a table with rows that are in both tables. The two tables must be union compatible to perform an intersection on them.

If we use the same two tables that were used in the union operation, the intersection will give us the projects that appear in the year 2002 and in the year 2003. Let us call the resulting table, which is produced by the intersection (Ω) operation, TABLE_B:

TABLE_B = PROJ2002 Ω PROJ2003

TABLE_B

ProjNo	Loc	Customer
1	Miami	Stocks
3	Trenton	Smith

Difference. The difference of two tables produces a table with rows that are present in the first table but not in the second table. The difference can be performed on union-compatible tables only.

If we find the difference (−) of the same two tables used in the previous operations and create TABLE_C, it will have projects for the year 2002 that are not projects for the year 2003:

TABLE_C = PROJ2002 – PROJ2003

TABLE_C

ProjNo	Loc	Customer
5	Phoenix	Robins
6	Edison	Shaw
7	Seattle	Douglas

Now, just as in mathematics, $A - B$ is not equal to $B - A$. If we perform the same operation to find projects from the year 2003 that did not exist in year 2002,

TABLE_D = PROJ2003 – PROJ2002

the resulting TABLE_D will look like this:

TABLE_D

ProjNo	Loc	Customer
2	Orlando	Allen
4	Charlotte	Jones

Projection. The projection operation allows us to create a table based on desirable columns from all existing columns in a table. The undesired columns are ignored. The projection operation returns the "vertical slices" of a table. The projection is indicated by including the table name and a list of desired columns:

TABLE_E = PARTS (PartDesc, Cost)

TABLE_E

PartDesc	Cost
Nut	19.95
Bolt	5.00
Washer	55.99

Selection. The selection operation selects rows from a table based on a condition or conditions. The conditional operators (=, <>, >, >=, <, <=) and the logical operators (AND, OR, NOT) are used along with columns and values to create conditions. The selection operation returns "horizontal slices" from a table.

Let us apply the selection (**Sel**) operation to the PARTS table:

TABLE_F = Sel (PARTS: Cost>10.00)

TABLE_F

PartNo	PartDesc	Vendor	Cost
11	Nut	Richards	19.95
33	Washer	Mobley	55.99

The resulting table has the same number of columns as the original table but fewer rows. The rows that satisfy the given condition are returned.

Product. A product of two tables is a combination everything in both tables. It is also known as a **Cartesian product**. It can cause huge results with big tables. If the first table has *x* rows and the second table has *y* rows, the resulting product has

$x \cdot y$ rows. If the first table has m columns and the second table has n columns, the resulting product has $m + n$ columns.

For simplicity, let us take two tables with one column each and perform the product ($\cdot$) operation on them:

DEPARTMENT

DeptName
Production
Supplies
Marketing

EMPLOYEE

Ename
Carter
Albert

TABLE_G = EMPLOYEE • DEPARTMENT

TABLE_G

Ename	DeptName
Carter	Production
Carter	Supplies
Carter	Marketing
Albert	Production
Albert	Supplies
Albert	Marketing

In this example, EMPLOYEE has two rows and DEPARTMENT three rows, so TABLE_G has $2 \cdot 3 = 6$ rows. EMPLOYEE has one column and DEPARTMENT one column, so TABLE_G has $1 + 1 = 2$ columns.

Assignment. This operation creates a new table from existing tables. We have been doing it throughout all the other operations. Assignment (=) gives us an ability to name new tables that are based on other tables. Note that assignment is not an Oracle term.

For example,

TABLE_A = PROJ2002 ∪ PROJ2003
TABLE_C = PROJ2002 – PROJ2003

Join. The join is one of the most important operations because of its ability to get related data from a number of tables. The join is based on common set of values, which does not have to have the same name in both tables but does have to have the same domain in both tables. When a join is based on equality of value, it is

known as a **natural join**. In Oracle, you will learn about the natural join, or **equijoin**, and also about other types of joins, such as **outer join**, **nonequijoin**, and **self-join**, that are based on the operators other than the equality operator.

For example, if we are interested in employee information along with department information, a join can be carried out using the EMPLOYEE and DEPARTMENT tables shown in Figure 1.2. The DeptNo column is the common column in both tables and will be used for the join condition:

TABLE_H = join (EMPLOYEE, DEPARTMENT : DeptNo = DeptNo)

TABLE_H

EmpNo	Ename	DeptNo	ProjNo	Salary	DeptName
101	Carter	10	1	25000	Production
102	Albert	20	3	37000	Supplies
103	Breen	30	6	50500	Marketing
104	Gould	20	5	23700	Supplies
105	Barker	10	7	75000	Production

The expression is read as "join a row in the EMPLOYEE table with a row in the DEPARTMENT table, where the DeptNo value in the EMPLOYEE table is equal to the DeptNo value in the DEPARTMENT table."

The join operation is an overhead on the system, because it is accomplished using a series of operations. A product is performed first, which results in $5 \cdot 3 = 15$ rows. A selection is performed next to select rows where the DeptNo values are equal. Finally, a projection is performed to eliminate duplicate DeptNo columns.

Division. The division operation is the most difficult operation to comprehend. It is not as simple as division in mathematics. In relational algebra, it identifies rows in one table that have a certain relationship to all rows in another table. Let us consider the following two tables:

PROJ

ProjNo
1
2
3

PRJPARTS

ProjNo	PartNo
1	11
2	33
3	11
1	22
2	11

Suppose we want to find out which parts are used in every project. We have to divide (/) PRJPARTS by PROJ:

```
TABLE_I = PRJPARTS / PROJ
```

TABLE_I

PartNo
11

The columns of TABLE_I are those from the dividend PRJPARTS that are not in the divisor PROJ. The rows of TABLE_I are a subset of the projection PRJPARTS (PartNo). The row (PartNo) is in TABLE_I if and only if (ProjNo, PartNo) is in the dividend PRJPARTS for every value of (ProjNo) in the divisor PROJ.

Summary. The nine operations provide users with a sufficient set of operations to work with the relational databases. Some of the operations are combinations of other operations, as we saw in the case of the join operation, but such operations are very useful in actual practice. In later chapters on Oracle, you will find the actual query statements used to accomplish the different operations outlined here. You will learn to perform these operations using Oracle's SQL.

Applications of Relational Algebra

Relational algebra is a procedural language in the sense that a user is required to use a series of operations to obtain a certain result. This language has its capabilities and limitations.

Problem 1

Referring to the tables in Figure 1.2, which employee is working on a project in Miami during the year 2003?

Solution

```
A = join(PROJ2003, EMPLOYEE : ProjNo = ProjNo)
B = Sel(A : Loc = 'Miami')
C = B(Ename)
```

Alternative solution

```
A = Sel(PROJ2003 : Loc = 'Miami')
B = join(EMPLOYEE, A : ProjNo = ProjNo)
C = B(Ename)
```

In these solutions, Table C will have one entry, *Carter,* an employee who works on project 1 in Miami. Relational algebra is called a procedural language, because a user has to perform a series of operations to achieve the desired result.

Problem 2

Referring to the tables in Figure 1.2, who has supplied parts for the project in Trenton?
Solution

```
D = PROJ2002 ∪ PROJ2003
E = Sel(D : Loc = 'Trenton')
F = join(E, PRJPARTS : ProjNo = ProjNo)
G = join(F, PARTS : PartNo = PartNo)
H = G(Vendor)
```

The solution uses four tables from the database. First, the union operation is performed to put all projects together. Then, the selection operation is performed to find all rows for projects in Trenton. Next, the result is joined with the PRJPARTS table to merge part number with project information. Then, the PARTS table is merged with the resulting table to get part–vendor information. At last, the name of the vendor is retrieved using projection, which returns *Richards*.

The solutions illustrated here show a fundamental weakness in relational algebra as a programming language. For users who have not come across problems like these before, the solutions are difficult to develop and comprehend. Relational algebra cannot group related information together. Neither can it perform calculations on numeric values or sort rows in any particular order. Printing information with formatting is out of the question! The actual implementation of relational languages is an integral part of fourth-generation query languages. These languages are supported by many other tools, which provide users with full application-development capabilities.

Relational Calculus

Relational calculus is a nonprocedural language. The programmer specifies the data requirement, and the system generates the operations needed to produce a table with the required data. In this section, we will try to understand relational calculus briefly with sample examples using the general syntax

```
Result = (column list) : Expression
```

The list of columns is on the left of the colon, and the expressions (and conditions) are on the right.

Problem 3

Referring to the tables in Figure 1.2, find projects where part number 11 is used.
Solution

```
(r.ProjNo) : r in PRJPARTS and r.PartNo = 11
```

In this expression, *r* is known as a **row variable.** The expression is read as "ProjNo in row *r*, where *r* is a row in the PRJPARTS table and PartNo in row *r* is 11." Each row

in PRJPARTS is examined using the condition to the right of the colon. The resulting table will contain project numbers 1, 3, and 2:

ProjNo
1
3
2

The solution for the same problem in relational algebra would look like this:

```
M = Sel (PRJPARTS : PartNo = 11)
N = M (ProjNo)
```

Problem 4

Referring to the tables in Figure 1.2, find employee names and salary for employees who work in Production.

Solution

```
(r.Ename, r.Salary) : r in EMPLOYEE and
                      s in DEPARTMENT and
                      s.DeptName = 'Production' and
                      r.DeptNo = s.DeptNo
```

RESULT

Ename	Salary
Carter	25000
Barker	75000

Problem 5

Referring to the tables in Figure 1.2, find employee names, department names, and locations for projects in the year 2003 for all employees who are working on project 1.

Solution

```
(r.Ename, s.DeptName, t.Loc) : r in EMPLOYEE and
                               s in DEPARTMENT and
                               t in PROJ2003 and
                               r.ProjNo = 1 and
                               r.ProjNo = t.ProjNo and
                               r.DeptNo = s.DeptNo
```

RESULT

Ename	DeptName	Loc
Carter	Production	Miami

The theoretical relational languages discussed in this chapter are the basis for the commercially available relational languages. SQL is a nonprocedural query language based on relational calculus that is supported by many relational database systems.

Final Note

A relational database is an electronic data repository that is supposed to satisfy users' data requests correctly, quickly, and efficiently. Most end users execute applications that work on databases. The most important task for the database designer is to create a properly designed database. If the database is not designed well, it will not be implemented well. The result is a nightmare for application developers. Even if the application is well written, a defective database may result in incorrect and sometimes meaningless results.

In the first half of Chapter 2, we will study the fundamentals of database design and modeling techniques. In the second half, we will learn the normalization process to control data redundancies. If you have learned a programming language before, you should be familiar with the tools used in the program development cycle. If you use a shortcut and create a wrong algorithm for a problem, you have to start the cycle all over again to rectify your logic. Database design is also like programming. The designing tools are aids for creating a "good" database. Remember that the database is a collection of tables. Do not build individual tables; rather, design a database as a whole.

IN A NUTSHELL . . .

- A database is an electronic store of data.
- An entity is a person, place, event, or item.
- Data are the facts describing an entity.
- An entity's characteristics are known as columns.
- An entity set is a collection of related entities.
- A database is a collection of entity sets.
- Relationships are interactions between entity sets.
- Three types of relationships are one-to-one (1:1), one-to-many (1:M), and many-to-many (M:N or M:M).
- In a relational model, a row is known as a tuple.
- The degree is the number of columns in a table, and the domain is a set of all possible values for a column.
- A primary key is a minimal set of columns used to uniquely define a row. When a single column is used as a key, it is known as a primary key. When a

combination of columns is used as a key, it is known as a composite primary key or a composite key.

- The foreign key is a column in a table that references a primary key in another table.
- Two integrity rules of relational model are entity integrity (the primary key may not be null) and referential integrity (the foreign key value may be null or must exist as a primary key value in another table).
- Relational algebra is a theoretical procedural language for data retrieval. It provides users with a set of operations such as union, intersection, difference, selection, projection, product, join, assignment, and division.
- Relational calculus is a nonprocedural relational language, which is the basis for today's popular relational database language Structured Query Language (SQL).

EXERCISE QUESTIONS

1. Define the following terms:
   - **a.** Entity.
   - **b.** Entity set.
   - **c.** Attribute.
   - **d.** Tuple.
   - **e.** Domain.
   - **f.** Key.
   - **g.** Null.
2. What are two integrity rules of the relational model?
3. What are different types of keys? What is their use?
4. Identify the primary key and foreign key for the following tables. Also, specify the table referenced by the foreign key. If a table does not have a foreign key, leave the entry blank. (*Note:* Some tables have a composite primary key. Identify all composite key columns for such tables.)

STUDENT (StudentId, Last, First, StartTerm, Birthdate, FacultyId, MajorId, Phone)
FACULTY (FacultyId, Name, RoomId, Phone, DeptId)
COURSE (CourseId, Title, Credits)
CRSSECTION (CsId, CourseId, Section, TermId, FacultyId, Day, RoomId)
REGISTRATION (StudentId, CsId, Midterm, Final)
ROOM (RoomType, RoomDesc)
TERM (TermId, TermDesc, StartDate, EndDate)
LOCATION (RoomId, Building, RoomNo, Capacity, RoomType)
MAJOR (MajorId, MajorDesc)
DEPARTMENT (DeptId, DeptName, FacultyId)

Table	Primary Key	Foreign Key	Tables Referenced
STUDENT			
FACULTY			
COURSE			
CRSSECTION			
REGISTRATION			
ROOM			
TERM			
LOCATION			
MAJOR			
DEPARTMENT			

5. Discuss different types of relationships, and provide examples.
6. What do we mean by union compatible? Which operations require tables to be union compatible with each other?
7. State the difference between the following:
   a. Union and intersection.
   b. Product and join.
   c. Selection and projection.
8. Using the tables given in Figure 1-2, the relational database notation of tables is

PROJ2002 (ProjNo, Loc, Customer)
PROJ2003 (ProjNo, Loc, Customer)
PARTS (PartNo, Vendor, Cost)
PRJPARTS (ProjNo, PartNo, Qty)
DEPARTMENT (DeptNo, DeptName)
EMPLOYEE (EmpNo, Ename, DeptNo, ProjNo, Salary)

Retrieve the following information by using a series of relational algebraic operations and also by using a relational calculus statement:

   a. All employee names.
   b. All employees working in department 20.
   c. All employees who are making $50,000 or more.
   d. All employees who are working in department 20 and also making more than $25,000.
   e. Vendors who supplied parts for the project in Miami during the year 2003.

2

Database Design: Data Modeling and Normalization

IN THIS CHAPTER . . .

- You will learn about database modeling techniques.
- You will work with symbols and E-R diagrams (ERD) for representation of entities and relationships.
- Types of dependencies within a table are examined and illustrated by using dependency diagrams.
- Reduction of data redundancy and the process of normalization are covered.

In Chapter 1, you learned about relational database management system (RDBMS) concepts. You also learned about theoretical languages and operations on tables. The relational model is very popular because of its simplicity. It shows data to the user in a very simple, logical view as a two-dimensional table. Anyone can create tables, but the strongest characteristic of the relational model is its ability to establish relationships among tables, which helps to reduce redundancy. Your queries are as good as the database you create. The first and foremost step in database creation is database design, which involves a certain degree of common sense. If the given list of columns describes different entities, you would create a separate table for each entity type. You would use foreign keys to establish relationships. To join two tables, you need at least one common or redundant column in both tables. All situations are not the same. There are complex cases in which common sense

does not do the job. Many proven modeling and design tools are available for a better database design. In this chapter, you will learn about different pictorial methods, techniques, and concepts to create a "near-perfect" database.

DATA MODELING

A model is a simplified version of real-life, complex objects. Databases are complex, and data modeling is a tool to represent the various components and their relationships. The entity-relationship (E-R) model is a very popular modeling tool among many such tools available today. Many tools are available for data modeling with E-R. All tools have some variations in representation of components. The E-R model provides:

- An excellent communication tool.
- A simple graphical representation of data.

The E-R model uses **E-R diagrams (ERD)** for graphical representation of the database components. An **entity** (or an entity set) is represented by a rectangle. The name of the entity (set) is written within the rectangle. Some tools prefer to use uppercase letters only for entities. The name of an entity set is a singular noun. For example, EMPLOYEE, CUSTOMER, and DEPARTMENT are singular entity set names (see Fig. 2-1).

A line represents relationship between the two entities. The name of the relationship is an active verb in lowercase letters. For example, *works, manages,* and *employs* are active verbs. Passive verbs can be used, but active verbs are preferable (see Fig. 2-2).

Figure 2-1 Entity representation in an E-R diagram.

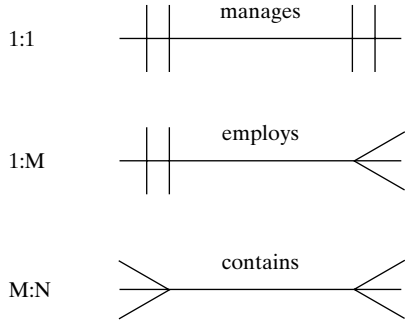


Figure 2-2 Representation of relationship in an E-R diagram.

The types of relationships (1:1, 1:M, and M:N) between entities are called **connectivity** or **multiplicity.** The connectivity is shown with vertical or angled lines next to each entity, as shown in Figure 2-2. For example, an EMPLOYEE supervises a DEPARTMENT, and a DEPARTMENT has one EMPLOYEE supervisor. A DIVISION contains many FACULTY members, but a FACULTY works for one DIVISION. An INVOICE contains many ITEMs, and an ITEM can be in more than one INVOICE.

Let us put everything together and represent these scenarios with the E-R diagram. Figure 2-3 shows entities, relationships, and connectivity.

The relationship between two entities can be given using the lower and upper limits. This information is called the **cardinality.** The cardinality is written next to each entity in the form *(n, m),* where *n* is the minimum number and *m* is the maximum number. For example, (1,1) next to EMPLOYEE means that an employee can supervise a minimum of one and a maximum of one department. Similarly, (1,1) next to DEPARTMENT says that one and only one employee supervises the department. The value (1,N) means a minimum of one and a maximum equal to any number (see Fig. 2-4). Some modern tools do not show cardinality in an E-R diagram.

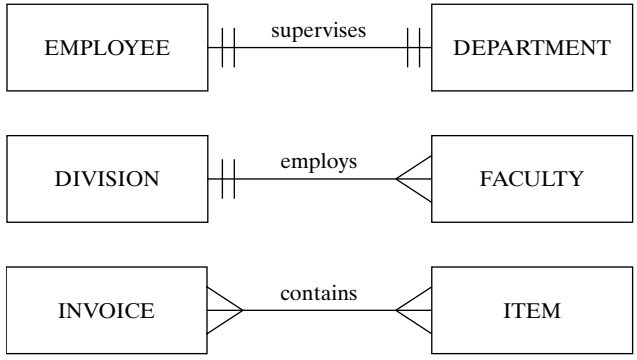


Figure 2-3 Entity, relationship, and connectivity.

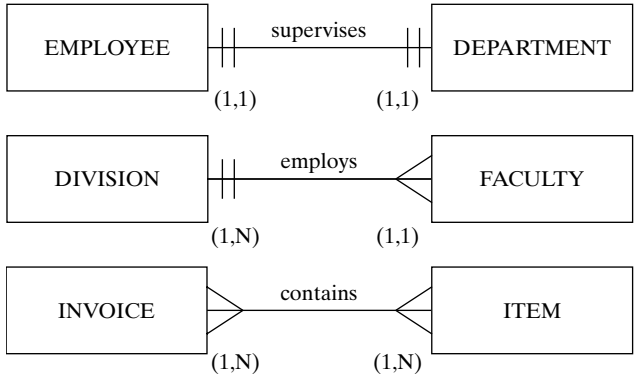


Figure 2-4 Cardinality.

In reality, corporations set rules for the minimum and maximum values for cardinality. A corporation may decide that a department must have a minimum of 10 employees and a maximum of 25 employees, which results in cardinality of (10,25). A college decides that a computer-science course section must have at minimum 5 students to recover the cost incurred and at maximum 35 students, because the computer lab contains only 35 terminals. An employee can be part of zero or more than one department, and an item may not be in any invoice! These types of decisions are known as **business rules.**

Figure 2-4 shows the E-R diagram with added cardinality. In real life, it is possible to have an entity that is not related to another entity at all times. The relationship becomes optional in such a case. In the example of a video rental store, a customer can rent video movies. In this case, there are times when the customer has not rented any movie, and there are times when the customer has rented one or more movies. Similarly, there can be a movie in the database that is or is not rented at a particular time. These are called **optional relationships** and are shown with a small circle next to the optional entity (see Fig. 2-5). The optional relationship can occur in 1:1, 1:M, or M:N relationships, and it can occur on one or both sides of the relationship.

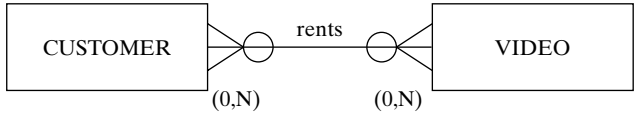


Figure 2-5 Optional relationships.

In relational databases, many-to-many (M:N) relationships are allowed, but they are not easy to implement. For example, an invoice has many items, and an item can be in many invoices. Refer to the INVOICE and ITEM relationship in Figure 2-4. At this point, you will be introduced to the relational schema, a graphical representation of tables, their column names, key components, and relations between the primary key in one table and the foreign key in another. You will also see the decomposition of an M:N relationship into two 1:M relationships. The decomposition from M:N to 1:M involves a third entity, known as a **composite entity** or an **associative entity.** The composite entity is created with the primary key from both tables with M:N relationships. The new entity has a composite key, which is a combination of primary keys from the original two entities. In the E-R diagram, a composite entity is drawn as a diamond within a rectangle (see Fig. 2-6). The composite

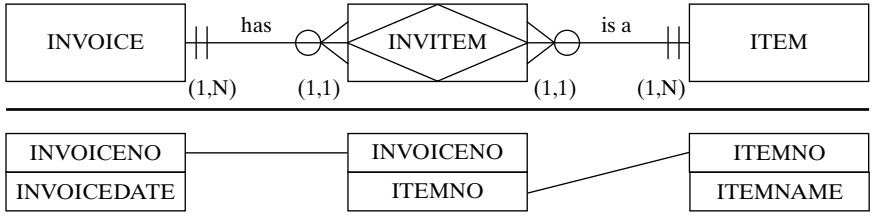


Figure 2-6 Composite entity and relational schema.

entity has a composite primary key with two columns, each of them being foreign keys referencing the other two entities in the database. For example, the foreign key INVOICENO in the INVITEM table references the INVOICENO column in the INVOICE table, and the foreign key ITEMNO in the INVITEM table references the ITEMNO column in the ITEM table.

In a database, there are entities that cannot exist by themselves. Such entities are known as **weak entities.** In Chapter 3, you will be introduced to two different sample databases. In the employee database of that chapter, there is an entity called EMPLOYEE with employees' demographic information and another entity called DEPENDENT with information about each employee's dependents. The DEPENDENT entity cannot exist by itself. There are no dependents for an employee who does not exist. In other words, you need the existence of an employee for his or her dependent to exist in the database. The weak entities are shown by double-lined rectangles (see Fig. 2-7).

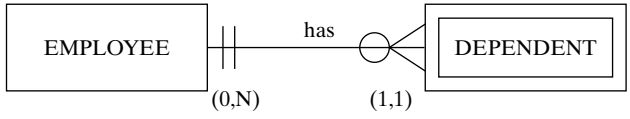


Figure 2-7 Weak entity.

Some of the other elements considered in the database design are:

- **Simple attributes**—attributes that cannot be subdivided; for example, last name, city, or gender.
- **Composite attributes**—attributes that can be subdivided, into atomic form; for example, a full name can be subdivided into the last name, first name, and middle initial.
- **Single-valued attributes**—attributes with a single value; for example, Employee ID, Social Security number, or date of birth.
- **Multivalued attributes**—attributes with multiple values; for example, degree codes or course registration. The multivalued attributes have to be given special consideration. They can be entered into one attribute with a value separator mark, or they can be entered in separate attributes with names like Course1, Course2, Course3, and so on. Alternatively, a separate, composite entity can be created.

DEPENDENCY

In Chapter 1, you learned that the primary key in a table identifies an entity. Every table in the database should have a primary key, which uniquely identifies an entity. For example, PartNo is a primary key in the PARTS table, and DeptNo is a primary key in the DEPARTMENT table. In Oracle, if you create a table and do not define

its primary key, Oracle does not consider it to be an error. You should define a primary key for all tables for integrity of data. Each table has other columns that do not make up the primary key for the table. Such columns are called the nonkey columns. The nonkey columns are functionally dependent on the primary key column. For example, PartDesc and Cost in the PARTS table are dependent on the primary key PartNo, and DeptName is dependent on the primary key DeptNo in the DEPARTMENT table.

Now, let us take a scenario as shown in Figure 2-8. The INVOICE table in Figure 2-8 does not have any single column that can uniquely identify an entity. The first choice would be InvNo. It is not a unique value in the table, however, because an invoice may contain more than one item and there may be more than one entry for an invoice. CustNo cannot be the primary key, because there can be many invoices for a customer and CustNo does not identify an invoice. ItemNo cannot be the primary key either, because an item may appear in more than one invoice and ItemNo does not describe an invoice. The table has a composite primary key, which consists of InvNo and ItemNo. InvNo and ItemNo together make up unique values for each row. All other columns that do not constitute the primary key are nonkey columns, and they are dependent on the primary key.

INVOICE

InvNo	InvDate	CustNo	ItemNo	CustName	ItemName	ItemPrice	Qty
1001	04/14/03	212	1	Starks	Screw	$2.25	5
1001	04/14/03	212	3	Starks	Bolt	$3.99	5
1001	04/14/03	212	5	Starks	Washer	$1.99	9
1002	04/17/03	225	1	Connors	Screw	$2.25	2
1002	04/17/03	225	2	Connors	Nut	$5.00	3
1003	04/17/03	239	1	Kapur	Screw	$2.25	7
1003	04/17/03	239	2	Kapur	Nut	$5.00	1
1004	04/18/03	211	4	Garcia	Hammer	$9.99	5

Figure 2-8 INVOICE table and its columns.

There are three types of dependencies in a table:

1. ***Total or full dependency:*** A nonkey column dependent on all primary key columns shows total dependency.
2. ***Partial dependency:*** In partial dependency, a nonkey column is dependent on part of the primary key.
3. ***Transitive dependency:*** In transitive dependency, a nonkey column is dependent on another nonkey column.

For example, in the INVOICE table, ItemName and ItemPrice are nonkey columns that are dependent only on a part of the primary key column ItemNo. They

are not dependent on the InvNo column. Similarly, the nonkey column InvDate is dependent only on InvNo. They are *partially dependent* on the primary key columns. The nonkey column CustName is not dependent on any primary key column but is dependent on another nonkey column, CustNo. It is said to have *transitive dependency*. The nonkey column Qty is dependent on both InvNo and ItemNo, so it is said to have *full dependency*.

DATABASE DESIGN

Relational database design involves an attempt to *synthesize* the database structure to get the "first draft." The initial draft goes through an *analysis* phase to improve the structure. More formal techniques are available for the analysis and improvement of the structure. In the synthesis phase, entities and their relationships are identified. The characteristics or the columns of all entities are also identified, and the designer defines the domains for each column. The candidate keys are picked, and primary keys are selected from them. The minimal set of columns is used as a primary key. If one column is sufficient to uniquely identify an entity, there is no need to select two columns to create a composite key. Avoid using names as primary keys, and break down composite columns into separate columns. For example, a name should be split into last name and first name. Once entities, columns, domains, and keys are defined, each entity is synthesized by creating a table for it. A process called **normalization** analyzes tables created by the synthesis process.

NORMAL FORMS

In Figure 2-8, data are repeated from row to row. For example, InvDate, CustNo, and CustName are repeated for same InvNo. The ItemName is entered repeatedly from invoice to invoice. There is a large amount of redundant data in a table with just eight rows! **Redundant data** can pose a huge problem in databases. First of all, someone has to enter the same data repeatedly. Second, if a change is made in one piece of the data, the change has to be made in many places. For example, if customer Starks changes his or her name to Starks-Johnson, you would go to the individual row in INVOICE and make that change. The redundancy may also lead to **anomalies.**

Anomalies

A ***deletion anomaly*** results when the deletion of information about one entity leads to the deletion of information about another entity. For example, in Figure 2-8, if an invoice for customer Garcia is removed, information about item number 4 is also deleted. An ***insertion anomaly*** occurs when the information about an entity cannot be inserted unless the information about another entity is known. For example, if the company buys a new item, this information cannot be entered unless an invoice

is created for a customer with that new item. An ***update anomaly*** can occur if the item price changes to a new price. The price change is valid after the change date, but not before the change date.

Unnecessary and unwanted redundancy and anomalies are not appropriate in databases. Such tables are in lower normal form. Normalization is a technique to reduce redundancy. It is a decomposition process to split tables. The splitting is performed carefully so that no information is lost. The higher the normal form is, the lower the redundancy. The table in Figure 2-8 is in first normal form (1NF).

First Normal Form (1NF)

A table is said to be in first normal form, or can be labeled 1NF, if the following conditions exist:

- The primary key is defined. This includes a composite key if a single column cannot be used as a primary key. In our INVOICE table, InvNo and ItemId are defined as the composite primary key components.
- All nonkey columns show functional dependency on the primary key components. If you know the invoice number and the item number, you can find out the invoice date, customer number and name, item name and price, and quantity ordered. For example, if InvNo = 1001 and ItemNo = 5 are known, then InvDate = 04/14/03, ItemName = Washer, ItemPrice = $1.99, CustNo = 212, and CustName = Starks.
- The table contains no multivalued columns. In a single-valued column, the intersection of a row and a column returns only one value. In a normalized table, the intersection of a row and a column is a single value. Some database packages, such as Unidata and Prime Information, allow multiple values in a column in a row, but Oracle does not. Figure 2-9 shows the INVOICE table of Figure 2-8 in unnormalized form. In Figure 2-9, the ItemNo, ItemName, ItemPrice, and Qty columns are multivalued.

INVOICE

InvNo	InvDate	CustNo	ItemNo	CustName	ItemName	ItemPrice	Qty
1001	04/14/03	212	1	Starks	Screw	$2.25	5
			3		Bolt	$3.99	5
			5		Washer	$1.99	9
1002	04/17/03	225	1	Connors	Screw	$2.25	2
			2		Nut	$5.00	3
1003	04/17/03	239	1	Kapur	Screw	$2.25	7
			2		Nut	$5.00	1
1004	04/18/03	211	4	Garcia	Hammer	$9.99	5

Figure 2-9 Unnormalized table with multivalued columns.

A table that is in 1NF may have redundant data. A table in 1NF does not show data consistency and integrity in the long run. The normalization technique is used to control and reduce redundancy and to bring the table to a higher normal form.

Second Normal Form (2NF)

A table is said to be in second normal form, or 2NF, if the following requirements are satisfied:

- All 1NF requirements are fulfilled.
- There is no partial dependency.

As you already know, partial dependency exists in a table in which nonkey columns are partially dependent on part of a composite key. Suppose a table is in 1NF and does not have a composite key. Is it in the second normal form also? Yes, it is in 2NF, because there is no partial dependency. Partial dependency only exists in a table with a composite key.

Third Normal Form (3NF)

A table is said to be in third normal form, or 3NF, if the following requirements are satisfied:

- All 2NF requirements are fulfilled.
- There is no transitive dependency.

A table that has transitive dependency is not in 3NF, but it needs to be decomposed further to achieve 3NF. However, a table in 2NF that does not contain any transitive dependency does not need any further decomposition and is automatically in 3NF.

Other, higher normal forms are defined in some database texts. Boyce–Codd normal form (BCNF), fourth normal form (4NF), fifth normal form (5NF), and domain key normal form (DKNF) are not covered in this text. In the following section, you will learn the normalization process by using dependency diagrams.

DEPENDENCY DIAGRAMS

A dependency diagram is used to show total (full), partial, and transitive dependencies in a table:

- The primary key components are highlighted. They are in bold letters and in boxes with a darker border. The primary key components are connected to each other using a bracket.
- The total and functional dependencies are shown with arrows drawn above the boxes.

- The partial and transitive dependencies are shown with arrows at the bottom of the diagram.

Conversion from 1NF to 2NF

We see in Figure 2-10 that a composite key is in the table and 1NF-to-2NF conversion is required. In this conversion, you remove all partial dependencies:

- First, write each primary key component on a separate line, because they will become primary keys in two new tables. (*Note:* If a primary key component does not have partial dependency on it, there is no need to write it on a separate line. In other words, you don't create a new table with that primary key.)
- Write the composite key on the third line. It will be the composite key in the third table.

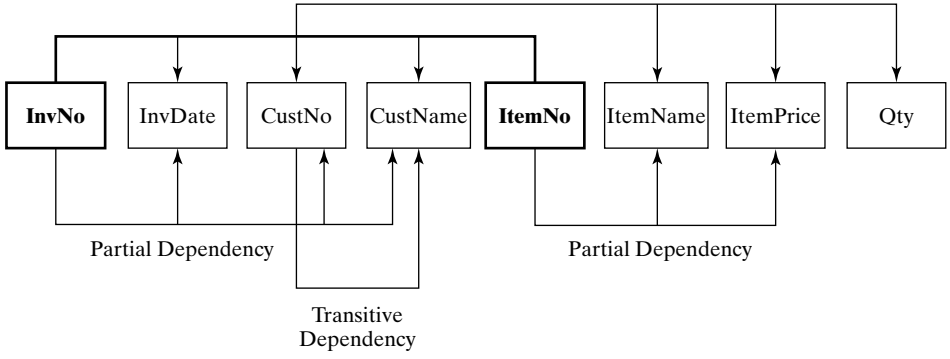


Figure 2-10 Dependency diagram.

Figure 2-11 shows the decomposition of one table in 1NF into three tables in 2NF. The reason behind the decomposition is moving columns with partial dependency to the new table along with the primary key. If only one of the two primary key columns has nonkey columns dependent on it, you will create only one new table to remove the partial dependency. The InvNo, CustNo, and CustName columns will move to the INVOICE table,

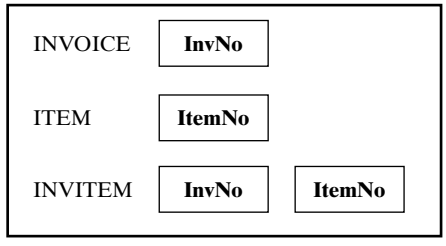


Figure 2-11 1NF-to-2NF decomposition.

because they are partially dependent on InvNo. ItemName and ItemPrice will move to the ITEM table, because they are partially dependent on ItemNo in Figure 2-10. The Qty column stays in INVITEM, because it is totally dependent on the composite key. The database will look like the one shown in Figure 2-12.

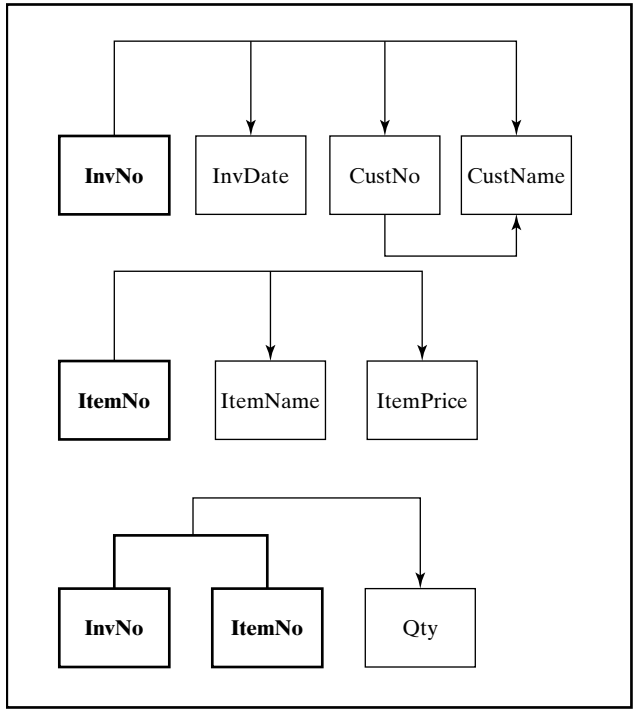


Figure 2-12 Tables in 2NF.

Conversion from 2NF to 3NF

The database tables in 2NF (see Fig. 2-12) have no partial dependency, but the INVOICE table still has transitive dependency:

- Move columns with the transitive dependency to a new table.
- Keep the primary key of the new table as a foreign key in the existing table.

In Figure 2-13, you see the decomposition from 2NF to 3NF to remove transitive dependency. A new CUSTOMER table is created with CustNo as its primary key. The CustNo column is kept in the INVOICE table as a foreign key to establish a relationship between INVOICE and CUSTOMER tables. The final database in 3NF looks like the one shown in Fig. 2-14.

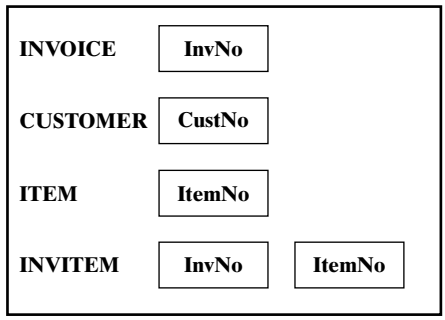


Figure 2-13 2NF-to-3NF decomposition.

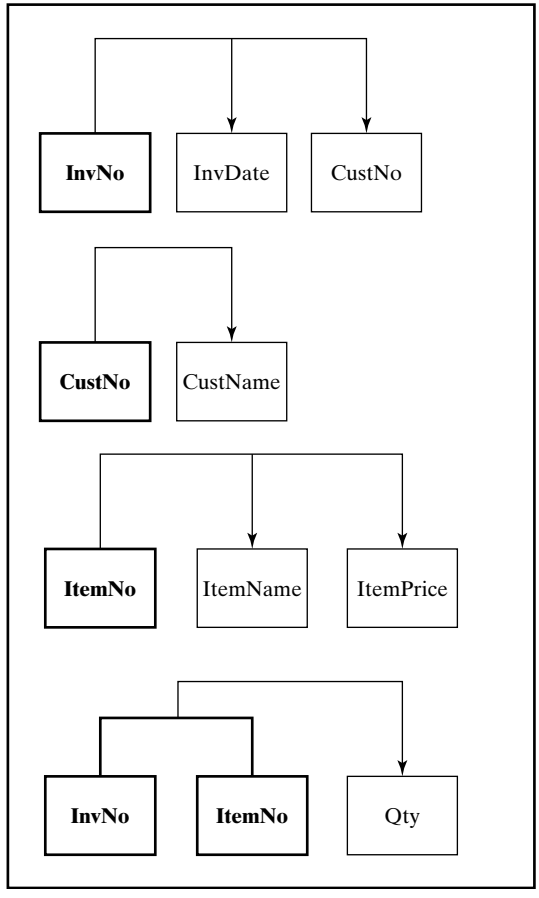


Figure 2-14 Tables in 3NF.

DENORMALIZATION

The normalization process splits tables into smaller tables. These tables are joined through common columns to retrieve information from different tables. The more tables you have in a database, the more joins are needed to get the desired information. In a multiuser environment, it is a costly overhead, and system performance is affected. Denormalization is the reverse process. It reduces the normal form, and it increases data redundancy. With denormalization, the information is stored with duplicate data, more storage is required, and anomalies and inconsistent data exist. The designer has to weigh this against performance to come up with a good design and performance.

ANOTHER EXAMPLE OF NORMALIZATION

In Figure 2-15, a table is shown in 1NF. The table contains a composite key that is composed of two columns, PlayerId and Year. This table contains each player's yearly statistics as well as team information. A player may belong to different teams during different years (it is assumed that a player belongs to one team during a year).

Looking at the table, the following dependencies exist:

- **Total dependency**—JerseyNum, PointsScoredInYear, GamesPlayed, TeamId, TeamName, and TeamLoc columns are dependent on primary key columns PlayerId and Year. A player may wear a different jersey number with same team or with a different team. Player may play for different team every year.
- **Partial dependency**—PlayerName and BirthDate columns are dependent on primary key column PlayerId only.
- **Transitive dependency**—TeamName and TeamLoc columns are dependent on non–key column TeamId. Fig 2-16.

1NF to 2NF (Removing Partial Dependencies)

A new table is created with a primary key column that has partial dependency on it. A new table is created with the PlayerId column as its primary key. The original table stays as it is with columns showing total dependency.

2NF to 3NF (Removing Transitive Dependencies)

A new table is created with the TeamId column as its primary key. TeamName and TeamLoc move to this new table. TeamId column also stays in the previous table as a foreign key to reference the new table.

Summary

The normalization process is not very easy to understand. In my classroom, some students find it very difficult. The process is based on common sense. A table is

Playerid	Playername	Year	JerseyNum	BirthDate	PointsScoredinYear	GamesPlayed	Teamid	TeamName	TeamLoc
1	JOHNSON	2001	32	4/15/1980	150	5	1	MUSTANGS	BRONX
1	JOHNSON	2002	32	4/15/1980	174	6	1	MUSTANGS	BRONX
1	JOHNSON	2003	32	4/15/1980	115	5	1	MUSTANGS	BRONX
2	NAMAN	2001	10	12/2/1985	100	3	1	MUSTANGS	BRONX
2	NAMAN	2002	10	12/2/1985	149	6	2	DEVILS	PRINCETON
2	NAMAN	2003	10	12/2/1985	185	6	5	EAGLES	BRUNSWICK
3	SHAW	2001	11	5/10/1986	99	5	4	BEARCATS	FORDS
3	SHAW	2002	11	5/10/1986	97	6	4	BEARCATS	FORDS
3	SHAW	2003	3	5/10/1986	115	6	6	KINGS	MANHATTAN
4	ALBERT	2003	33	5/19/1983	29	3	3	BULLDOGS	MONROE
5	ANTHONY	2001	21	1/19/1979	110	6	3	BULLDOGS	MONROE
5	ANTHONY	2002	21	1/19/1979	78	4	3	BULLDOGS	MONROE
5	ANTHONY	2003	33	1/19/1979	111	5	1	MUSTANGS	BRONX
6	RICHARDS	2003	33	7/10/1977	63	6	2	DEVILS	PRINCETON
7	ROBERTS	2003	55	6/6/1981	44	6	5	EAGLES	BRUNSWICK
8	JONES	2001	2	12/31/1981	123	6	6	KINGS	MANHATTAN
8	JONES	2002	2	12/31/1981	100	6	4	BEARCATS	FORDS
9	JORDAN	2003	23	2/17/1986	101	2	1	MUSTANGS	BRONX

Figure 2-15 Table in 1NF.

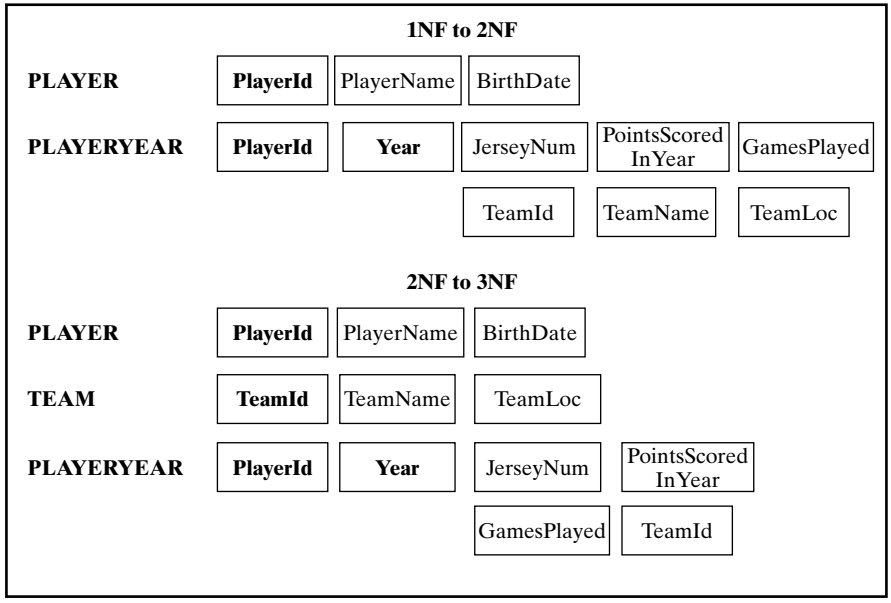


Figure 2-16 Tables in 2NF and 3NF.

supposed to describe one entity. If a table describes multiple entities, it needs to be decomposed. When tables are decomposed, there should be enough foreign keys to establish relationships among tables. You should not end up with a table that is not related to any other table in the database. A player's name has nothing to do with the year or the team he or she plays for, so it depends on the player's ID only. (In professional sports like the NBA, there are players who have changed their name! We have to treat those cases as exceptions, and we do not design something just to support such exceptions.) How many points did Jordan scored in 2002? To get that information, you need the player's ID as well as the year. We find new things to be complicated at first, but practice makes a man (or a woman) perfect. There is another problem on normalization in the exercise section. Go ahead and give it a try.

IN A NUTSHELL . . .

- A model is a simplified version of real-life, complex objects.
- The entity-relationship (E-R) diagram is an excellent communication tool that represents a database graphically.
- In an E-R diagram, an entity (set) is represented by a rectangle with the name of the entity set written as an uppercase, singular noun.
- An E-R diagram represents a relationship as an active verb inside a diamond-shaped box.

- The types of relationships (1:1, 1:M, and M:N) are called connectivity.
- The cardinality shows the lower and the upper limit of a relationship.
- All entities are not related to each other at all times. Such a relationship is known as an optional relationship. It can occur in 1:1, 1:M, and M:N relationships, and it can occur on one or both sides of the relationship.
- M:N relationships are complex to implement. Each M:N relationship is decomposed into two 1:M relationships using a third entity, known as a composite entity. A composite entity has a composite primary key, which is combination of primary keys from the other two entities.
- Simple attributes cannot be divided, but composite attributes can be subdivided.
- Attributes can be single valued or multivalued.
- All nonkey columns in a table are functionally dependent on the primary key columns of the table.
- In partial dependency, a nonkey column is dependent on part of the composite primary key.
- In transitive dependency, a nonkey column is dependent on another nonkey column.
- A database design involves both synthesis and analysis. Normalization is a process of analyzing a database created with synthesis.
- Normalization is a decomposition process to reduce data redundancy and data anomalies.
- A database in 1NF does not have any multivalued columns.
- A database in 2NF does not have any partial dependencies.
- A database in 3NF does not have any transitive dependencies.
- Higher normal forms are also possible, and the process of denormalization is performed on a database to weigh performance against redundancy.

EXERCISE QUESTIONS

True/False:

1. *Connectivity* is a term used for relationships in the E-R diagram.
2. Partial dependency can exist in a table with a simple primary key.
3. In transitive dependency, a column is dependent on the primary key.
4. Higher normal form means lower redundancy.
5. Normalization is a process of converting a database design from lower to higher normal form.
6. A 1NF table with simple primary key is already in 2NF.

Define the Following Terms:

1. Partial dependency.
2. Transitive dependency.
3. Normalization.
4. Data anomalies.
5. Cardinality.

E-R Diagram Exercise:

1. A student takes many courses, and many students take a course. Create an E-R diagram to represent the entities, connectivity, and cardinality. Decompose the E-R diagram with a composite entity to reduce each M:N relationship to two 1:M relationships. Also, draw the relational schema for the database.

Dependency Diagram Exercise:

EMP

EMPID	LAST	FIRST	DEPTID	DEPTNAME	DEPENDENTNO	DEPENDENTSSN	DEPENDENTDOB

1. Create a dependency diagram for the set of given columns for the EMP table:

EMPID	Employee's ID
LAST	Employee's last name
FIRST	Employee's first name
DEPTID	Employee's department number
DEPTNAME	Employee's department name
DEPENDENTNO	Employee's number of dependents
DEPENDENTSSN	Dependent's Social Security number
DEPENDENTDOB	Dependent's date of birth

The primary key columns are EMPID and DEPENDENTNO.

Normalization Exercise:

1. Using the dependency diagram of the EMP table from the previous exercise, normalize the table to 3NF. Use 1NF-to-2NF and then 2NF-to-3NF conversion.

PART 2
ORACLE SQL

3

Oracle9i: An Overview

IN THIS CHAPTER . . .

- You will learn the differences between a client/server database, such as Oracle, and PC-based database software.
- The Oracle client/server Database Management System (DBMS) and its utilities are introduced.
- The Oracle development environment SQL*Plus and its various types of commands are covered.
- An overview of primary language SQL (Structured Query Language) to communicate with the Oracle Server is given.
- SQL*Plus Worksheet and *i*SQL*Plus environments are introduced.
- Designs of two case study databases, a college's student registration database system and a company's employee database system, are discussed.

PERSONAL DATABASES

Personal database management systems, such as Microsoft Access and Visual Fox Pro, are usually stored on a user's desktop computer system or a **client computer.** These database packages are developed primarily for single-user applications. When such a package is used for a multiuser or a shared access environment, the database

applications and the data are stored on a file server, or a **server**, and data are transmitted to the client computers over the network (see Fig. 3-1). A server is a computer that accepts and services requests from other computers, such as client computers. A server also enables other computers to share resources. A server's **resources** could include the server's hard-disk drive space, application programs on a server's hard drive, data stored on the server's drive, or printers. A **network** is an infrastructure of hardware and software that enables computers to communicate with each other.

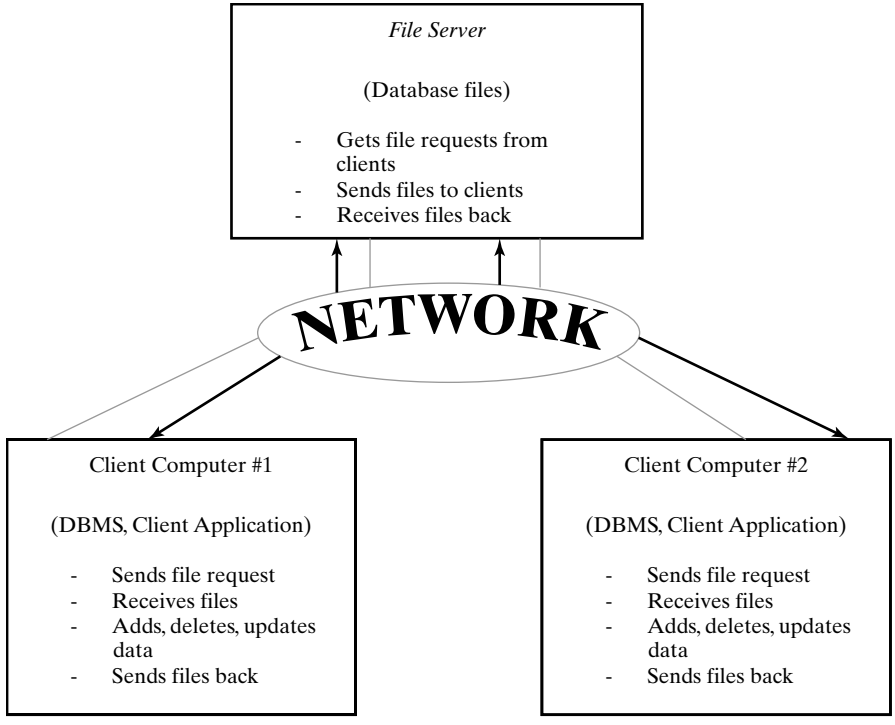


Figure 3-1 A personal database system in a multiuser environment.

Demand on Client and Network

In a network environment with a personal DBMS, the client computer must load the entire database application along with the client database application in its memory. If the client requires a small piece of data from the server's large database, the server has to transmit the entire database to the client over the network. In some database packages, only part of the database is transmitted. In any case, the client computer hardware must handle heavy demand, and the network must sustain heavy traffic in both directions. In the network environment, the system response to various client requests depends on the speed of the network and the amount of traffic over it.

Table Locking

The personal database system assumes that no two transactions will happen at the same time on one table, which is known as **optimistic locking**. In optimistic locking, the tables are not locked by the database system. If one agent sells a seat for a basketball game and another agent tries to sell the same seat at the same time, the database system will notify the second agent about the update on the table after his or her read—but it will go ahead and let the second agent sell the seat anyway. Application programmers can write code to avoid such a situation, but that requires added effort on programmer's part. Personal database software does not lock tables automatically.

Client Failure

When a client is performing record insertions, deletions, or updates, those records are locked by that client and are not available to the other clients. Now, if the client with all the record locks fails because of software or hardware malfunction or a power outage, the locked records stay locked. The transactions in progress at the time of failure are lost. The database can get corrupted and needs to be repaired. To repair the database, all users have to log off during the repair, which can take anywhere from a few minutes to a few hours! If the database is not repairable, data can be restored from the last backup, but the transactions since the last backup are lost and have to be reentered.

Transaction Processing

Personal databases, such as Microsoft Access, do not have file-based transaction logging. Instead, transactions are logged in the client's memory. If the client fails in the middle of a batch of transactions, some transactions are written to the database and some are not. The **transaction log** is lost, because it is not stored in a file. If a client writes a check to transfer money from a savings account to a checking account, the first transaction debits money from the savings account. Now, suppose the client fails right after that. The checking account never gets credited with the amount because the second transaction is lost!

CLIENT/SERVER DATABASES

Client/server databases, such as Oracle, run the DBMS as a process on the server and run a client database application on each client. The client application sends a request for data over the network to the server. When the server receives the client request, the DBMS retrieves data from the database, performs the required processing on the data, and sends only the requested data (or query result) back to the client over the network (see Fig. 3-2).

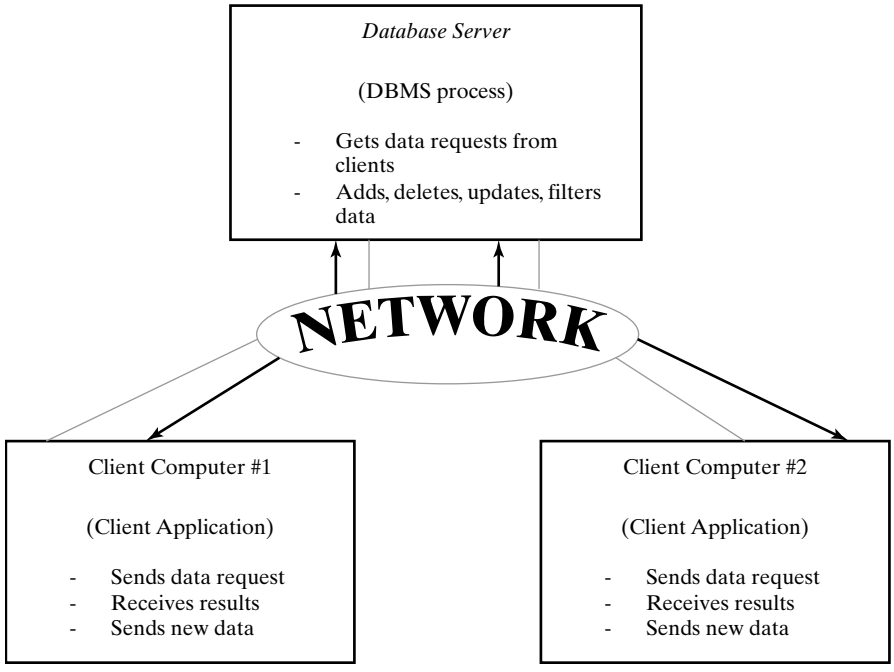


Figure 3-2 A client/server database system in a multiuser environment.

Demand on Client and Network

The client computer does not run the entire DBMS, only the client application that requests data from the server. The client does not store any database on its local drive; it receives only the requested data from the server. Data processing is performed on the server's side. The demand at the client's end is minimal. The clients request data from the server, and only the requested data are sent back via the network, which results in less network traffic.

Table Locking

In a client/server system, such as Oracle, when an agent reads a table to sell a seat for a basketball game, for example, it is locked totally or partly by the DBMS. The second agent cannot read the part of the table with available seats. Once the first agent sells the seat and it is marked as *sold,* the lock is released for the next agent. The DBMS takes care of the locking automatically, and it involves no extra effort on an application programmer's part.

Client Failure

In case of a client failure, the client/server database is not affected. The other clients are not affected either. Only the failed client's transactions in progress are lost. If

the server fails, a **central transaction log,** which keeps a log of all current database changes, allows the Database Administrator (DBA) or DBMS to complete or roll back unfinished transactions. The rolled-back transactions are not implemented in the database. The DBA (or DBMS) can notify clients to resubmit rolled-back transactions. Most client/server database packages have fast and powerful recovery utilities.

Transaction Processing

If a grouped transaction or batch transaction fails in the middle, all transactions are rolled back. The DBMS will enable the bank, for example, to make sure that both accounts' balances are changed if the batch transaction goes through. If the batch transaction fails, the balance in none of the accounts is changed.

ORACLE9i: AN INTRODUCTION

Oracle9i is a client/server DBMS that is based on the relational database model discussed in Chapter 1. Oracle9i is one of the most popular database-management software packages available today. The Oracle Corporation, incorporated in 1986, is the second-largest software company in the world. Its software product line includes Oracle9i Database, Oracle9i Application Server, Oracle9i Developer Suite, Oracle Collaboration Suite, and Oracle E-Business Suite. Oracle Corporation's revenue was $9.475 billion in the fiscal year ending May 2003, down 2 percent from the previous year. The net income for the fiscal year was $2.4 billion. Currently, the common stock trades at more than $14, with a company market capitalization of more than $75 billion. Oracle9i database is capable of supporting over 10,000 simultaneous users and a database size of up to 100 terabytes! It is preferred to the other PC-based RDBMS packages because its client/server database qualities, failure handling, recovery management, administrative tools to manage users and the database, object-oriented capabilities, graphical user interface (GUI) tools, and Web interface capabilities. It is widely used by corporations of all sizes to develop mission-critical applications. It is also used as a teaching tool by educational institutions to teach object-relational database technology, **Structured Query Language (SQL), PL/SQL** (Oracle's procedural language extension to SQL), and interfacing Web and Oracle databases. Oracle has an educational initiative program to form partnerships with educational institutions that enable these institutions to obtain Oracle database software at a nominal membership fee.

Oracle software is installed to work in three different environments. In a *stand-alone* environment, such as a laptop or desktop that is not on a network, Oracle Enterprise database software and SQL*Plus client software are installed on same machine. In a *Client/Server* environment, a two-tier architecture with a client communicating with a server, Oracle Enterprise database software resides on the server side, and SQL*Plus client software resides on the client machine. In *Three-Tier* architecture, the client communicates with the Oracle database server through a middle-tier iSQL*Plus, an interface through a Web browser.

Oracle9i components include Oracle9i Database, Oracle9i Application Server, and Oracle9i Developer Suite. Oracle9i Database introduces Oracle9i Real Application Cluster, which replaces Oracle Parallel Server, and features integrated system management, high availability, powerful disaster recovery, system fault recovery, planned downtime, and high security. Oracle9i Application Server is industry's preferred application server for database-driven Web sites, with an innovative and comprehensive set of middle-tier services. Oracle9i Developer Suite, an integrated product, provides a high-performance development environment with tools like Oracle Forms Developer, Oracle Designer, Oracle JDeveloper, Oracle Reports Developer, and Oracle Discoverer. Some of the Oracle9i tools include:

- ***SQL*Plus***—The SQL*Plus environment is for writing command-line SQL queries to work with database objects such as tables, views, synonyms, and sequences.
- ***PL/SQL***—PL/SQL is Oracle's extension to SQL for creating procedural code to manipulate data.
- ***Developer Suite***—This tool is used for developing database applications and includes:
  - ***JDeveloper***—a Java development tool.
  - ***Designer***—to model business processes and generate Enterprise applications.
  - ***Forms Developer***—a developement tool for Internet and client/server–based environments.
  - ***Oracle Reports***—a report generation tool.
- ***Enterprise Manager***—A tool for managing users and databases. Enterprise Manager uses the following tools:
  - ***Storage Manager***—to create and manage "tablespaces."
  - ***Instance Manager***—to start, stop, or tune databases.
  - ***Security Manager***—to create and manage users, profiles, and roles.
  - ***Warehouse Manager***—to manage data warehousing applications.
  - ***XML Database Manager***—to render traditional database data as XML for e-business support.
  - ***SQL Worksheet***—to enter, edit, and execute SQL*Plus code or to run client-side scripts.
  - i***SQL*Plus***—a Web-based environment to execute SQL*Plus code.
- ***Oracle Application Server (Oracle9iAS)***—A tool for creating a Web site that allows users to access Oracle databases through Web pages. It includes:
  - ***Web Server***.

THE SQL*PLUS ENVIRONMENT

When a user logs in to connect to the Oracle server, SQL*Plus provides the user with the **SQL>** prompt, where the user writes queries or commands. Features of SQL*Plus include:

- Accepts ad hoc entry of statements at the command line prompt (i.e., SQL>).
- Accepts SQL statements from files.
- Provides a line editor for modifying SQL queries.
- Provides environment, editor, format, execution, interaction, and file commands.
- Formats query results, and displays reports on the screen.
- Controls environmental settings.
- Accesses local and remote databases.

STRUCTURED QUERY LANGUAGE (SQL)

The standard query language for relational databases is SQL (Structured Query Language). It is standardized and accepted by ANSI (American National Standards Institute) and the ISO (International Organization for Standardization). Structured Query Language is a fourth-generation, high-level, nonprocedural language, unlike third-generation compiler languages such as C, COBOL, or Visual Basic, which are procedural. Using a nonprocedural language query, a user requests data from the RDBMS. The SQL language uses English-like commands such as CREATE, INSERT, DELETE, UPDATE, and DROP. The SQL language is standardized, and its syntax is the same across most RDBMS packages. The different packages have minor variations, however, and they do support some additional commands. Oracle's SQL is different from the ANSI SQL. Oracle's SQL is referred to as SQL*, but we will simply call it SQL throughout this text. Oracle9i also supports ANSI syntax for joining tables.

Oracle9i uses the following types of SQL statements for command-line queries to communicate with the Oracle server from any tool or application:

- ***Data retrieval***—retrieves data from the database (e.g., SELECT).
- ***Data Manipulation Language (DML)***—inserts new rows, changes existing rows, and removes unwanted rows (e.g., INSERT, UPDATE, and DELETE).
- ***Data Definition Language (DDL)***—creates, changes, and removes a table's structure (e.g., CREATE, ALTER, DROP, RENAME, and TRUNCATE).
- ***Transaction control***—manages and changes logical transactions. Transactions are changes made to the data by DML statements that are grouped together (e.g., COMMIT, SAVEPOINT, and ROLLBACK).
- ***Data Control Language (DCL)***—gives and removes rights to Oracle objects (e.g., GRANT and REVOKE).

In a few books, the SELECT statement is treated as a subset of DML language. I have chosen to separate the SELECT statement from other DML statements, because it does not perform any manipulation on data.

The SQL queries are typed at the SQL> prompt. If a query exceeds one line, the SQL*Plus environment displays the next line number on the line editor. An SQL query is sent to the server by ending a query with a semicolon (;). A query can also be sent to the server by using a forward slash (/) on a new line instead of ending the query with a semicolon.

LOGGING IN TO SQL*PLUS

In the Windows environment, click **Start | Programs | Oracle − Orahome92 | Application Development | SQL Plus** (see Fig. 3-3). A Log on window will pop up. Enter your **Username, Password,** and **Host String** as provided by your Database Administrator (see Fig 3-4).

In a command-line environment such as DOS, type **sqlplus [username [/password [@host/database]]]** to log in. If the entire command is typed, the password will be visible on the screen. If you enter your username only, a prompt will be displayed

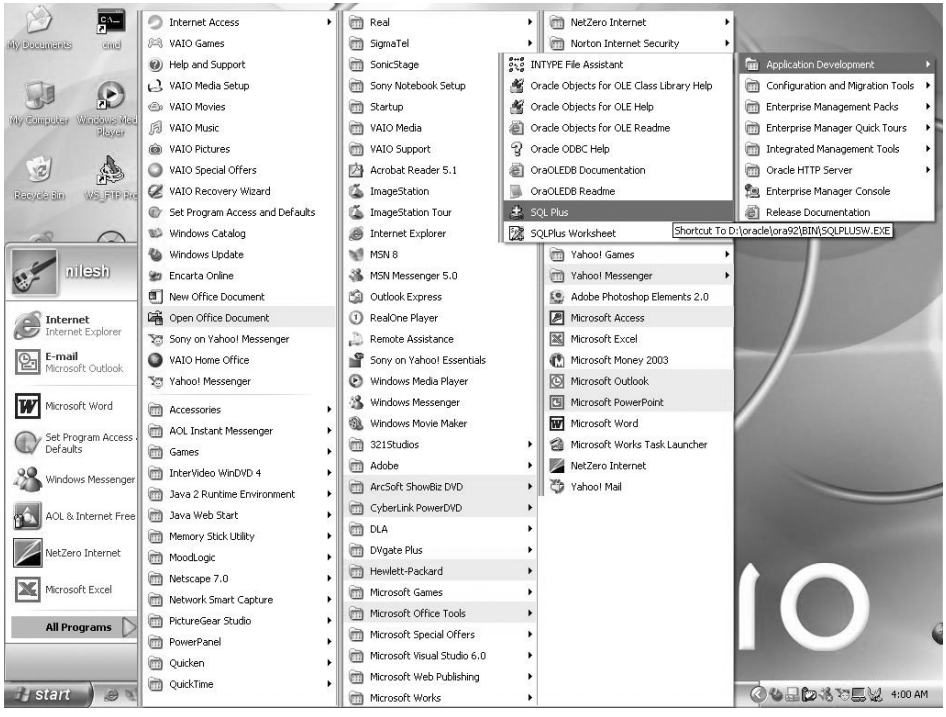


Figure 3-3 Running SQL*Plus.

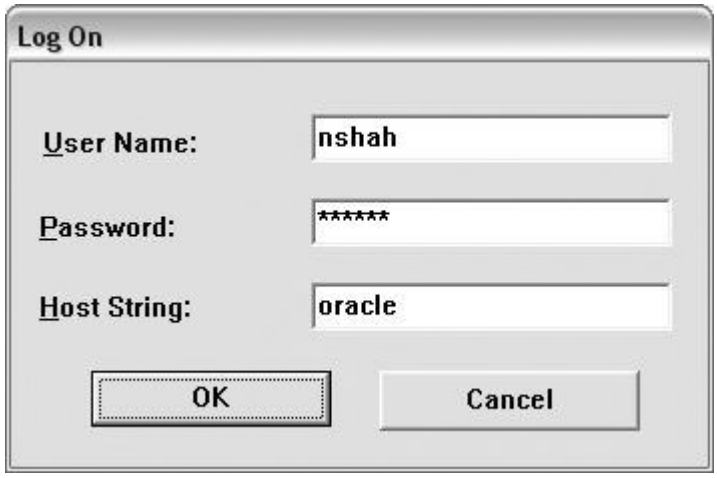


Figure 3-4 SQL*Plus Log On window.

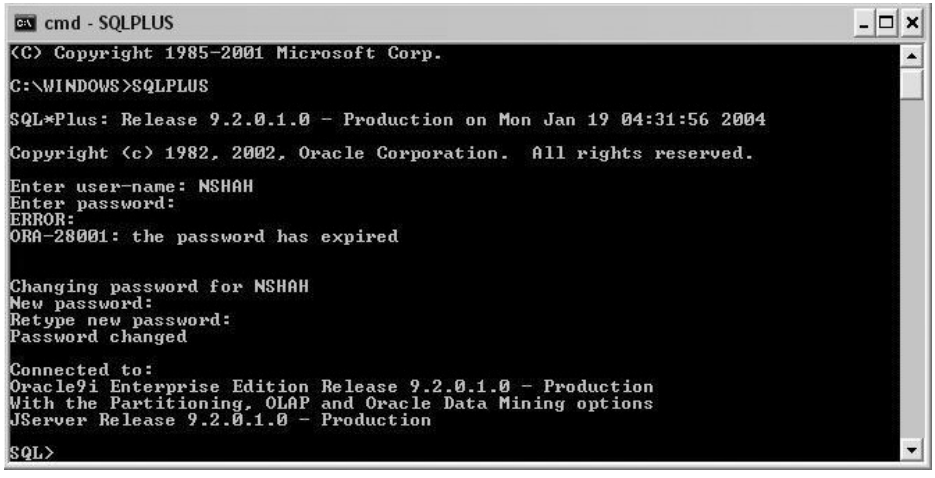


Figure 3-5 Running SQL*Plus from MS-DOS prompt.

for your password. The password typed at this prompt will be masked to maintain its integrity (see Fig 3-5).

There are a couple of common login problems. If you enter an incorrect username or password, you will receive the following error message from Oracle server:

ORA-01017: invalid username/password; logon denied

User should consult DBA (students should consult their instructor or academic computing personnel) to resolve username/password problems.

If there is a connectivity issue between your client PC and Oracle or the host string has an invalid entry, you will see the following error message:

ORA-12154: TNS: could not resolve service name

Oracle stores host string/service values in a file called TNSNAMES.ORA. If you receive a TNS error, call your DBA!

After a user logs in and the default SQL> prompt is displayed, the user can start a new Oracle session. A user can enter only one SQL*Plus command at the SQL> prompt at a time (see Fig. 3-6). SQL*Plus commands are not stored in the buffer. If a command is very long, you can continue the command on the next line by using a hyphen at the end of the current line.

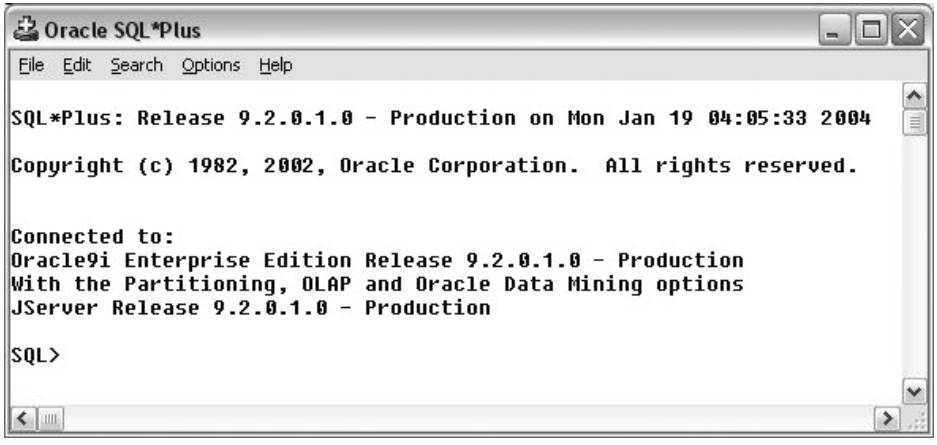


Figure 3-6 SQL*Plus environment—SQL prompt.

The SQL queries and SQL*Plus commands are typed at the SQL> prompt. The SQL*Plus commands do not have a terminator, but SQL queries are terminated using a semicolon (;) at the end or by typing a forward slash (/) on a new line. Figure 3-7 shows the differences between SQL statements and SQL*Plus commands.

A user may change his or her password by using SQL*Plus command PASSWORD at the SQL> prompt. SQL*Plus prompts user to enter the old password first, then the new password, and then to confirm new password by retyping it (see Fig. 3-8).

SQL*PLUS COMMANDS

In the tables of Figures 3-9 and 3-10, file-related (see Fig. 3-9) and editor-related (see Fig. 3-10) commands are described. The command words are in bold letters, and user-supplied filenames and extensions are in lowercase. The abbreviations allowed for the SQL*Plus commands are underlined. The optional parameters are enclosed within a pair of brackets ([]). Note that the filename in the file-related commands requires entire file path.

SQL	SQL*Plus
1. A nonprocedural language to communicate with the Oracle server.	1. An environment for executing SQL statements.
2. ANSI standard.	2. Oracle's proprietary environment.
3. Key words cannot be abbreviated.	3. Key words can be abbreviated.
4. Last statement is stored in the buffer.	4. Commands are not stored in the buffer.
5. Statements manipulate data and table structures in the database.	5. Commands do not allow manipulation of data in the database.
6. Uses a termination character to execute the command immediately.	6. Commands do not need a termination character.

Figure 3-7 SQL queries versus SQL*Plus commands.

Figure 3-8 PASSWORD command.

COMMAND	DESCRIPTION
GET *filename* [.*ext*]	Writes previously saved file to the buffer. The default extension is SQL. Writes SQL statements, not SQL*Plus commands.
START *filename* [.*ext*]	Runs a previously saved command from file.
@*filename*	Same as START.
EDIT	Invokes the default editor (e.g., Notepad), and saves buffer contents in a file called *afiedt.buf*.
EDIT [*filename* [.*ext*]]	Invokes editor with the command from a saved file.
SAVE *filename* [.*ext*] REPLACE |	Saves current buffer contents to a file with the option to replace or append.
APPEND	
SPOOL [*filename* [.*ext*] |OFF |OUT]	Stores query results in a file. OFF closes the file, and OUT sends the file to the system printer.
EXIT	Leaves SQL*Plus environment. Commits current transaction.

Figure 3-9 SQL*Plus file-related commands.

COMMAND	DESCRIPTION
APPEND *text*	Adds text to the end of the current line.
CHANGE / *old* / *new*	Changes old text to new text in the current line.
CHANGE / *text* /	Deletes text from the current line.
CLEAR BUFFER	Deletes all lines from the SQL buffer.
DEL	Deletes current line.
DEL *n*	Deletes line *n*.
DEL *m n*	Deletes lines *m* through *n*.
INPUT	Inserts an indefinite number of lines.
INPUT *text*	Inserts a line of text.
LIST	Lists all lines from the SQL buffer.
LIST *n*	Lists line *n*.
LIST *m n*	Lists lines *m* through *n*.
RUN	Displays and runs an SQL statement in the buffer.
N	Makes line *N* current.
n text	Replaces line *n* with text.
0 text	Inserts a line before line 1.
CLEAR SCREEN	Clears screen.

Figure 3-10 SQL*Plus editing commands.

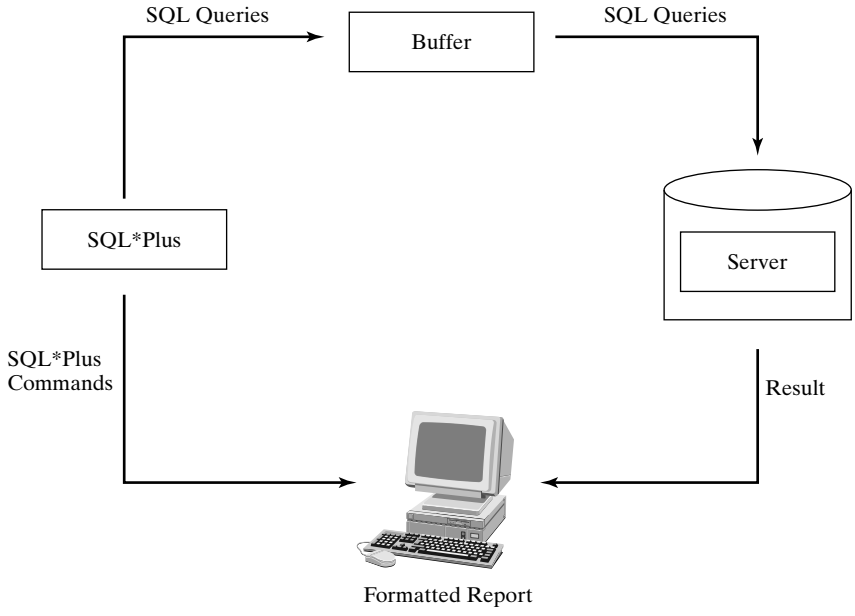


Figure 3-11 Interaction between SQL*Plus and SQL.

A query typed at the prompt is loaded in to the SQL*Plus buffer. When the query is sent to the server, the server processes data and sends back the result to the client computer, which can be formatted using SQL*Plus formatting commands (see Fig. 3-11).

ORACLE ERRORS AND ONLINE HELP

In the next chapter, you will learn to write and execute SQL statements and queries at the SQL*Plus prompt. If you make a syntax error, Oracle will display an error message showing the line number and the error location on that line. Oracle places an asterisk (*) at the location of the error and also displays an error code (e.g., ORA-00XXX), followed by a brief description. Just like any programming language compiler, some error messages are not user friendly. You will get used to some of the common error messages as you start experimenting. Some queries take up a few lines—and for a typist like me, typing mistakes are bound to happen! The online help screens are illustrated in the next chapter.

Some errors are easy to find; some are not. To fix an error, always start at the line where the error is shown, but keep in mind that the error might not be on that line. Check for common mistakes like misspelled keywords, missing commas, misplaced commas, missing parentheses, invalid user-defined names, or repeated user-defined identifiers. In the next chapter, we will actually go through the entire procedure by entering an erroneous query.

Each Oracle product has its own specific help file. The help application opens up a window, similar to Microsoft Windows help, where you can click on the Index tab and type the error code (ORA-00XXX) of interest. Oracle provides the user with an explanation of the cause of the error and the user action required to rectify it. If you don't see the Oracle Help option in the programs menu, search for the *ora.hlp* file and open it. You can also access Oracle9i Release 2 online help at the following URL:

http://download-west.oracle.com/docs/cd/B10501_01/mix.920/a96625/toc.htm

The URL is working at this time, but such URLs change frequently. Another way would be to go through Oracle's home page (www.oracle.com) and search for help on your Oracle product. You are required to register to Oracle Technology Network (otn.oracle.com) to access this page. Registration is free, and the membership benefits include free software downloads and access to online documentation. Figure 3-12 shows the initial Web page with Oracle9i Release 2's master index.

SQL*Plus also provides the user with Help on its various commands and SQL language. You may use the HELP INDEX command at the SQL> prompt to list the SQL*Plus commands.

To obtain help on one of the topics listed in the help index, type **INDEX [TOPIC]**. For example, type **HELP DESCRIBE** as shown in Figure 3-13, and SQL*Plus returns a brief description and syntax on the topic.

ALTERNATE TEXT EDITORS

The SQL*Plus editor is a line editor similar to EDLIN in MS-DOS. It is not fun working with line editors. The user does not have control over the screen. Line editors do not allow a user to move the cursor up and down, and clicking with a mouse is definitely out of the question. You can use an alternate text editor such as

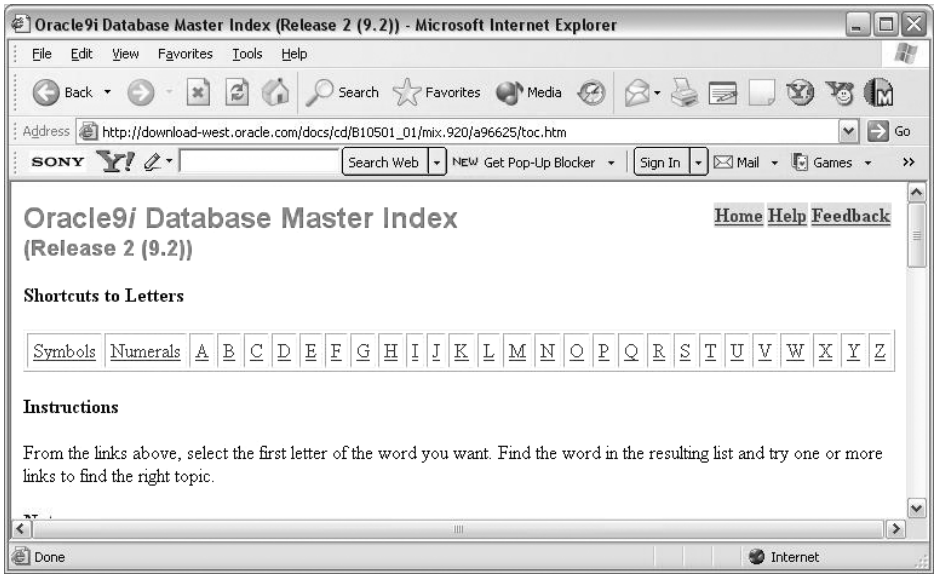


Figure 3-12 Oracle9i online documentation—index page.

```
SQL> HELP INDEX
Enter Help [topic] for help.
@               COPY            PAUSE                SHUTDOWN
@@              DEFINE          PRINT                SPOOL
/               DEL             PROMPT               SQLPLUS
ACCEPT          DESCRIBE        QUIT                 START
APPEND          DISCONNECT      RECOVER              STARTUP
ARCHIVE LOG     EDIT            REMARK               STORE
ATTRIBUTE       EXECUTE         REPFOOTER            TIMING
BREAK           EXIT            REPHEADER            TTITLE
BTITLE          GET             RESERVED WORDS (SQL) UNDEFINE
CHANGE          HELP            RESERVED WORDS (PL/SQL) VARIABLE
CLEAR           HOST            RUN                  WHENEVER OSERROR
COLUMN          INPUT           SAVE                 WHENEVER SQLERROR
COMPUTE         LIST            SET
CONNECT         PASSOWORD       SHOW
SQL>  HELP DESCRIBE
DESCRIBE

Lists the column definitions for a table, view, or synonym, or the specifications for
a function or procedure.
DESC[RIBE]  {[schema.] object [@connect_identifier]}
SQL>
```

Figure 3-13 SQL*Plus help.

Notepad or any other text editor in Windows to type your SQL queries. The query typed in a full-screen text editor has to be saved in a file with an *.sql* extension or copied to the clipboard. The query can be loaded from a file or pasted from the clipboard into SQL*Plus to execute it at the SQL>prompt.

The EDIT (or ED) command can be used to invoke an alternate text editor from the SQL*Plus command prompt. SQL*Plus allows the user to select an alternate editor, and in most cases, Windows' popular text editor, Notepad, is the default alternate editor. The EDIT command invokes the alternate editor with contents from the buffer or with an existing file. The user can make necessary changes to the query and transfer the contents back to SQL*Plus. We will see an illustration of this in Chapter 4. An alternate editor is defined or invoked with EDIT menu in SQL*Plus. The default alternate editor is Notepad for the Windows environment, but this can be changed to another text editor of user's choice with *Define Editor* . . . menu option (see Fig. 3-14).

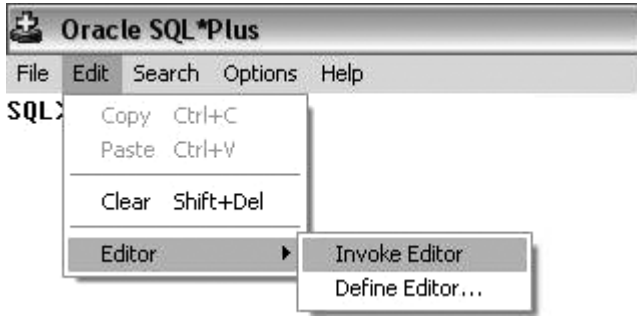


Figure 3-14 Alternate editor.

SQL*PLUS WORKSHEET

The SQL*Plus Worksheet is another environment available with Oracle's Enterprise Manager. The SQL*Plus Worksheet enables you to enter, edit, and execute SQL*Plus code. You can also run client-side scripts. The SQL*Plus Worksheet maintains a history of the commands you have issued, so you can easily retrieve and execute previous commands. You can execute SQL*Plus Worksheet by double-clicking on the SQL*Plus Worksheet icon in the Windows desktop or by selecting the following from the Windows START button:

START | [All] Programs | Oracle – OraHome92 | Application Development | SQLPlus Worksheet

An Enterprise Manager login screen is then displayed. The user logs in with username, password, and host string, just like logging into SQL*Plus. On a successful login, the user enters the SQL*Plus Worksheet with database connection (see Fig. 3-15). On the left side, a tool bar is displayed with connection, execute, command history, previous command, next command, and help icons from top to bottom, respectively. The user can select these same options from the File or Worksheet menu.

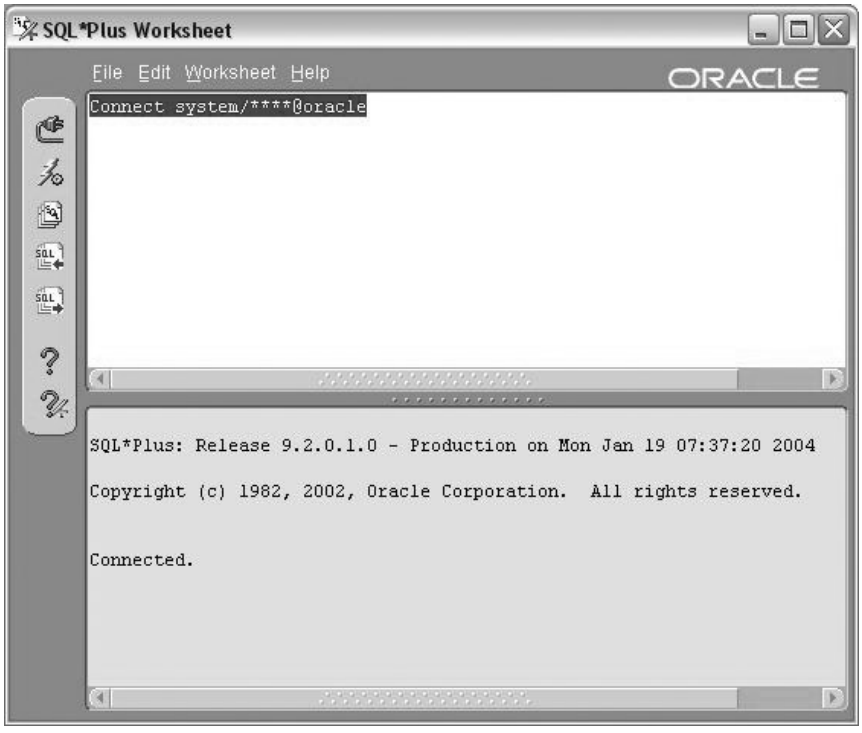


Figure 3-15 SQL*Plus worksheet.

The SQL*Plus Worksheet screen has two horizontal halves. The user issues an SQL query or SQL*Plus command in the upper half and then clicks on the lightening-bolt icon to execute. SQL*Plus Worksheet output is displayed in the lower half (see Fig. 3-16). During a session, the user issues many commands and statements. Unlike SQL*Plus, SQL*Plus Worksheet keeps all commands and statements in history. The user can click on the command history icon to view them in reverse order, with the most recent command at the top. The user can then select a command/statement and click GET to load it again and execute it.

The user may save input and output in separate files with the FILE menu and its options *Save Input As* . . . and *Save Output As* . . . , respectively. The input is stored in a file with the default extension *.sql*, and output is stored with the default extension *.txt*.

There are a few differences between SQL*Plus and SQL*Plus Worksheet. The following settings have been set up by Oracle Enterprise Manager SQL*Plus Worksheet. It is recommended that users do not change them:

- The sqlplus system variable SQLPROMPT is disabled by default.
- The sqlplus system variable SQLNUMBER is set to OFF by default.

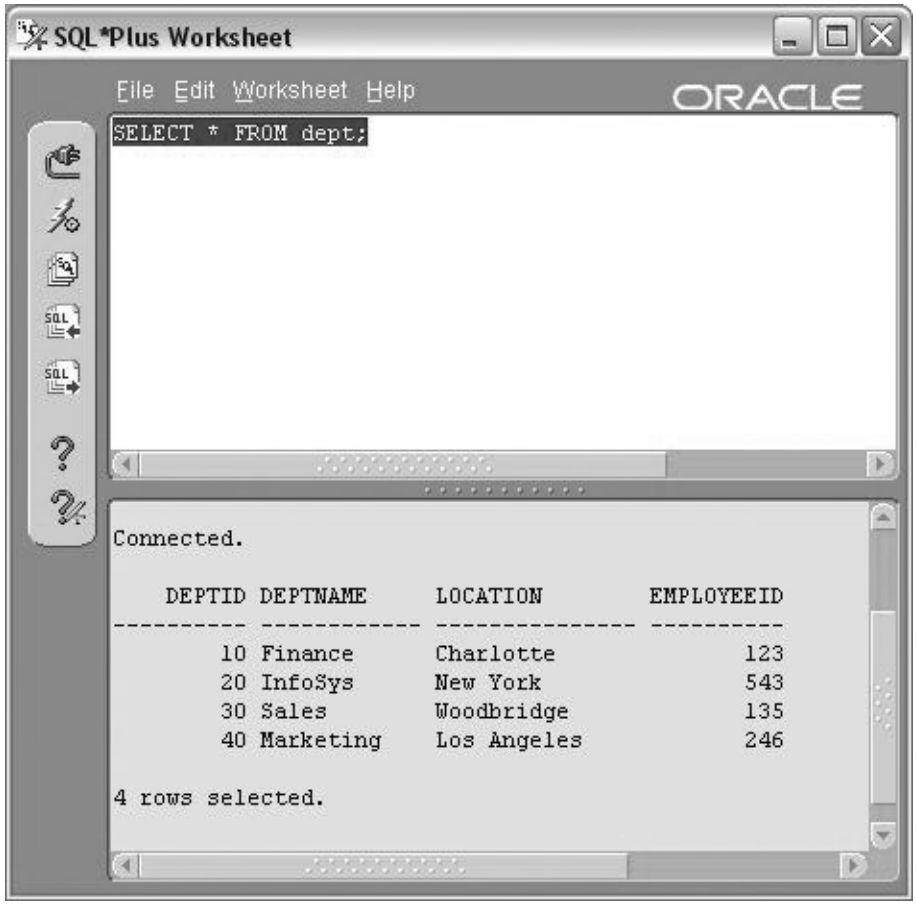


Figure 3-16 Input/output in SQL*Plus worksheet.

- The sqlplus system variable PAGESIZE is set to its maximum by default.
- The sqlplus system variable TAB is set to OFF by default.
- If there are any prompt characters "&" in a script, SQL*Plus treats it as a "define" prompt. (You will see more on DEFINE later.)
- The HOST command is not supported in SQL*Plus Worksheet.
- The SQL*Plus user variable _EDITOR is disabled by default. If you enable it by using "DEFINE_EDITOR = 'editorname'," the specified editor launches on the server when you type the EDIT command. As a result, you are unable to access normal SQL*Plus Worksheet functionalities.
- Remote execution is not supported.
- Remote load and save of script files is not supported.

*i*SQL*Plus

The third environment is Web based and is called *i*SQL*Plus. To access it through a Web browser, enter a URL as follows:

```
http://machinename.domainname:port/isqlplus
```

In this URL, *machinename* is your machine, but port number is not required in all versions.

In Figure 3-17, the following URL is used:

```
http://nshah-monroe/isqlplus
```

where *nshah-monroe* is the machine name. The domain name is not used, because *i*SQL*Plus is located in the local machine. In the login dialog, connection identifier (or host string) is optional if the URL points to the correct database instance. Consult your IT or lab personnel for information on server name, domain name, default port, and database name at your installation or college. Alternately, http://localhost:

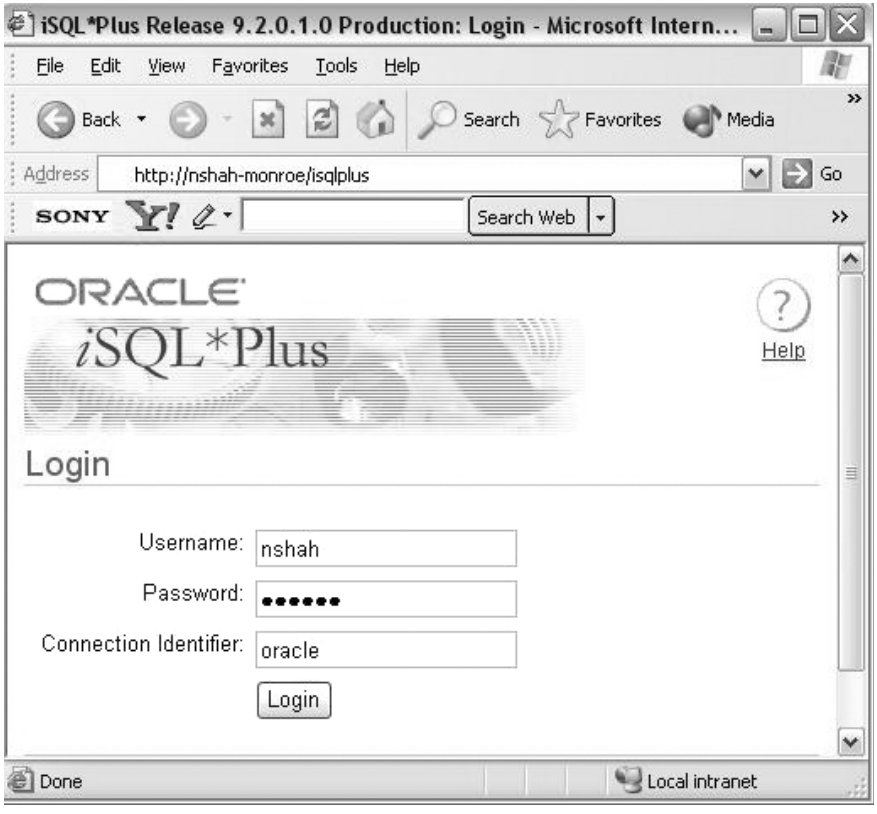


Figure 3-17 *i*SQL*Plus login screen.

port/isqlplus or https://localhost:port/isqlplus URL can be used. Oracle installer provides the port number to access HTTP (Apache) server at the installation time.

In Figure 3-18, a sample SQL query is executed in the *i*SQL*Plus work screen. The output is at the bottom. This environment also provides the user with options to execute-commands, save or load scripts, check command history, and more. Another benefit of this environment is the buttons on the top right, such as the Help buttton to connect to Oracle's online help, the History button to browse and select command history, and so on.

In this text, most of the screen shots are from the SQL*Plus environment. The SQL*Plus Worksheet and *i*SQL*Plus environments are migrations to graphical and

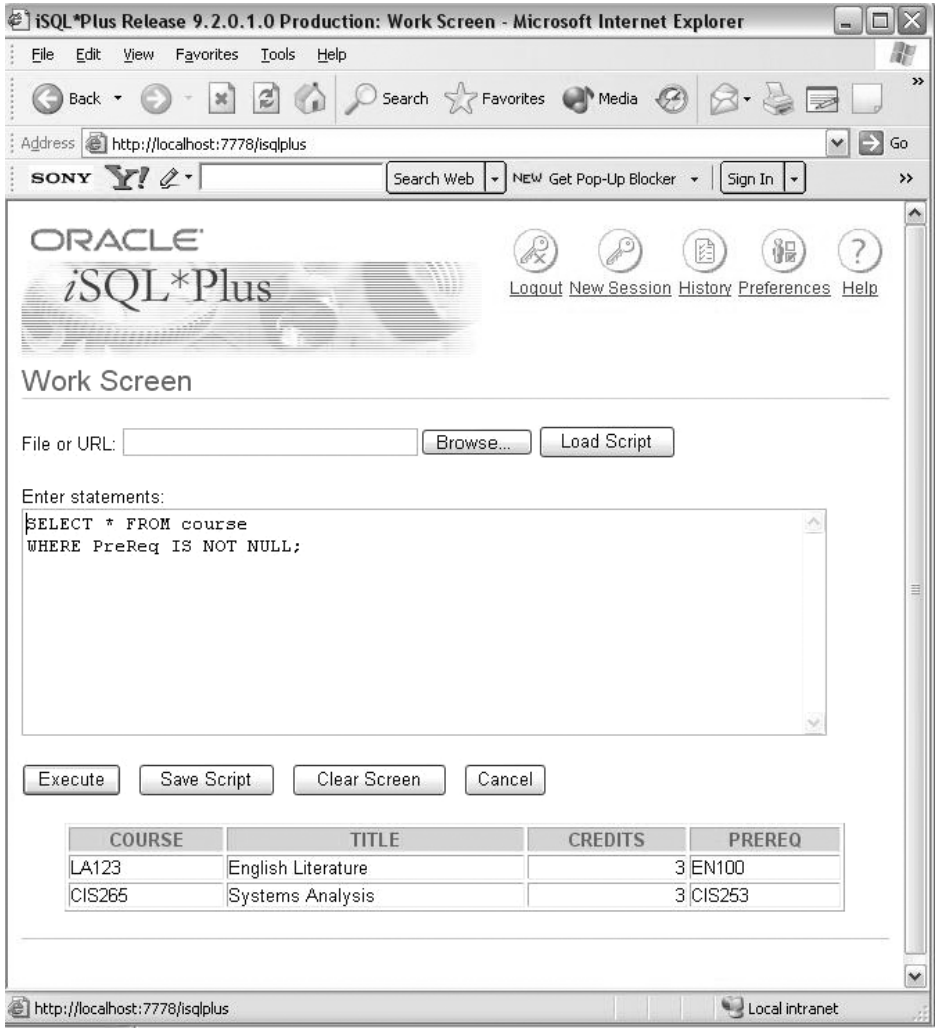


Figure 3-18 *i*SQL*Plus work screen with query and result.

Web-based environments, respectively, which are more popular environments than the command-based predecessors.

SAMPLE DATABASES

In this textbook, each chapter uses examples and lab activities to teach various Oracle query statements and utilities by using two fictitious databases. One database is developed for the Indo–US (IU) College, which keeps track of students, faculty, courses offered, and enrollment by semester. The other database is designed for the NamanNavan (N2) Corporation, which has employee, department, dependent, and employee-level information. These databases are designed using the normalization techniques covered in Chapter 2. Each database table contains appropriate data to explain the results obtained from different query statements in the later chapters.

The Indo–US (IU) College Student Database

The Indo–US (IU) College has a computerized database system in place. They have spent a large sum of money establishing a network infrastructure. The database-management system that resides on their icomputer system is a unnormalized database. Unnormalized databases are rare in the relational database software family. Their main feature is multivalued fields, which is a application programmer's nightmare. This database does support its own version of query language, but it is not common in the business world. It is not easy to find Information Systems personnel with experience using such a database system. The IU administration is fed up with the "holes" in the system because of redundant data, bad data, and lack of data integrity. Students do not have the ability to retrieve course information, register online, or retrieve their unofficial transcript records using computers in the laboratory or the library. Faculty members do not have online access to their own course information, rosters, or student information.

The college has decided to use a standard relational database management system to overcome deficiencies of the existing system. The tables for the IU College are illustrated in Figure 3-19, and the E-R diagram is illustrated in Figure 3-20.

The IU student database consists of 10 tables to store student master records, faculty master records, course master records, term-by-term course offerings, and student registration by each term. It also uses other supporting tables for lookup and additional information for basic entities in the database. With custom programming, the faculty and students are given online access to the system to view demographic information, course information, unofficial transcript records, availability of course sections by term, and so on. The faculty and student logins are created by the Information Systems department. Each student or faculty login name consists of the first letter of the first name, the first three letters of the last name, and the last four digits of the Social Security number. The Social Security number data is intentionally omitted from the STUDENT and FACULTY tables. Student and faculty members use their Social Security numbers as passwords to access the system.

STUDENT (StudentId, Last, First, Street, City, State, Zip, StartTerm, BirthDate, FacultyId, MajorId, Phone)

StudentId	Last	First	Street	City	State	Zip	Start Term	Birth Date	FacultyId	MajorId	Phone
00100	Diaz	Jose	1 Ford Avenue #7	Hill	NJ	08863	WN03	02/12/83	123	100	9735551111
00101	Tyler	Mickey	12 Morris Avenue	Bronx	NY	10468	SP03	03/18/84	555	500	7185552222
00102	Patel	Rajesh	25 River Road #3	Edison	NJ	08837	WN03	12/12/85	111	400	7325553333
00103	Rickles	Deborah	100 Main Street	Iselin	NJ	08838	FL02	10/20/70	555	500	7325554444
00104	Lee	Brian	2845 First Lane	Hope	NY	11373	WN03	11/28/85	345	600	2125555555
00105	Khan	Amir	213 Broadway	Clifton	NJ	07222	WN03	07/07/84	222	200	2017585555

FACULTY (FacultyId, Name, RoomId, Phone, DeptId)

FacultyId	Name	RoomId	Phone	DeptId
111	Jones	11	525	1
222	Williams	20	533	2
123	Mobley	11	529	1
235	Vajpayee	12	577	2
345	Sen	12	579	3
444	Rivera	21	544	4
555	Chang	17	587	5
333	Collins	17	599	3

COURSE (CourseId, Title, Credits)

CourseId	Title	Credits	PreReq
EN100	Basic English	0	
LA123	English Literature	3	EN100
CIS253	Database Systems	3	
CIS265	Systems Analysis	3	CIS253
MA150	College Algebra	3	
AC101	Accounting	3	

Figure 3-19 Sample tables for the Indo–US (IU) College database.

CRSSECTION (CsId, CourseId, Section, TermId, FacultyId, Day, StartTime, EndTime, RoomId, MaxCount)

CsId	CourseId	Section	TermId	FacultyId	Day	StartTime	EndTime	RoomId	MaxCount
1101	CIS265	01	WN03	111	MW	09:00	10:30	13	30
1102	CIS253	01	WN03	123	TR	09:00	10:30	18	40
1103	MA150	02	WN03	444	F	09:00	12:00	15	25
1104	AC101	10	WN03	345	MW	10:30	12:00	16	35
1205	CIS265	01	SP03		MW	09:00	10:30	14	35
1206	CIS265	02	SP03	111	TR	09:00	10:30	18	30
1207	LA123	05	SP03		MW	09:00	10:30	15	30
1208	CIS253	21	SP03	123	TR	09:00	10:30	14	40
1209	CIS253	11	SP03	111	MW	09:00	10:30	18	40
1210	CIS253	31	SP03	123	F	TBA	TBA	19	2

TERM (TermId, TermDesc, StartDate, EndDate)

TermId	TermDesc	StartDate	EndDate
SP02	Spring 2002	04/28/2002	08/16/2002
FL02	Fall 2002	09/08/2002	12/20/2002
WN03	Winter 2003	01/05/2003	04/18/2003
SP03	Spring 2003	04/27/2003	08/15/2003
FL03	Fall 2003	09/07/2003	12/19/2003

ROOM (RoomType, RoomDesc)

RoomType	RoomDesc
L	Lab
C	Classroom
O	Office

REGISTRATION (StudentId, CsId, Midterm, Final, RegStatus)

StudentId	CsId	Midterm	Final	Status
00100	1103	C	F	R
00100	1102	B	B	R
00100	1104	B	A	R
00102	1102	F	D	R
00102	1103	A	A	R
00103	1101	F	W	W
00103	1104	D	D	R
00100	1207			X
00103	1206			W
00104	1206			X
00104	1207			R
00104	1210			R
00105	1208			R
00105	1209			X
00101	1205			X
00102	1210			R
00102	1207			X
00102	1206			R

DEPARTMENT (DeptId, DeptName, FacultyId)

DeptId	DeptName	FacultyId
1	Computer Science	111
2	Telecommunications	222
3	Accounting	333
4	Math & Science	444
5	Liberal Arts	555

MAJOR (MajorId, MajorDesc)

MajorId	MajorDesc
100	AAS-Accounting
200	AAS-Computer Science
300	AAS-Telecommunications
400	BS-Accounting
500	BS-Computer Science
600	BS-Telecommunications

Figure 3-19 Sample tables for the Indo–US College database *(Continued)*.

LOCATION (RoomId, Building, RoomNo, Capacity, RoomType)

RoomId	Building	RoomNo	Capacity	RoomType
11	Gandhi	101	2	O
12	Gandhi	103	2	O
13	Kennedy	202	35	L
14	Kennedy	204	50	L
15	Nehru	301	50	C
16	Nehru	309	45	C
17	Gandhi	105	2	O
18	Kennedy	206	40	L
19	Kennedy	210	30	L
20	Gandhi	107	2	O
21	Gandhi	109	2	O

Figure 3-19 Sample tables for the Indo–US (IU) College database *(Continued).*

The STUDENT table contains demographic information. StudentId is used as a primary key field. Social Security numbers could have been used as the primary key, but a system-generated StudentId is used instead. Because of security issues, all student-related reports use StudentId instead of Social Security number. The table contains FacultyId as a foreign key, which references the FACULTY table to keep track of student advisors' information throughout the curriculum. Another foreign key, MajorId, uses the MAJOR table for a description of students' majors. A third foreign key, StartTerm, keeps track of each student's entry term into the college.

The FACULTY table contains location, phone extension, and department information for faculty members identified by a FacultyId, the table's primary key. To get further information, the DeptId foreign key field is used for departmental or chairperson's data from the DEPARTMENT table. Similarly, the RoomId field is used to get the building and room number of the faculty offices.

The COURSE table (primary key, CourseId) is the course master, with course title and credit information. The prerequisite information is kept in the PreReq column, which serves as a foreign key and references the primary key of the same COURSE table. It is not very common for a foreign key to reference the primary key of its own table. A self-join operation is used to join such a table to self.

Each term and its dates are entered in the TERM table. An abbreviated term and year are used together to create four-character primary key, such as WN03 for the Winter 2003 term. Normally, numeric columns are prefered for the primary key, but for our purposes, we are intentionally using a character column as a primary key. The LOCATION table with a unique RoomId serves multiple purposes by keeping all types of rooms: It helps academic departments in locating an available room for a new course section, and the room capacity helps the academic administration in scheduling classes with maximum allowable enrollment less than or equal to the room capacity.

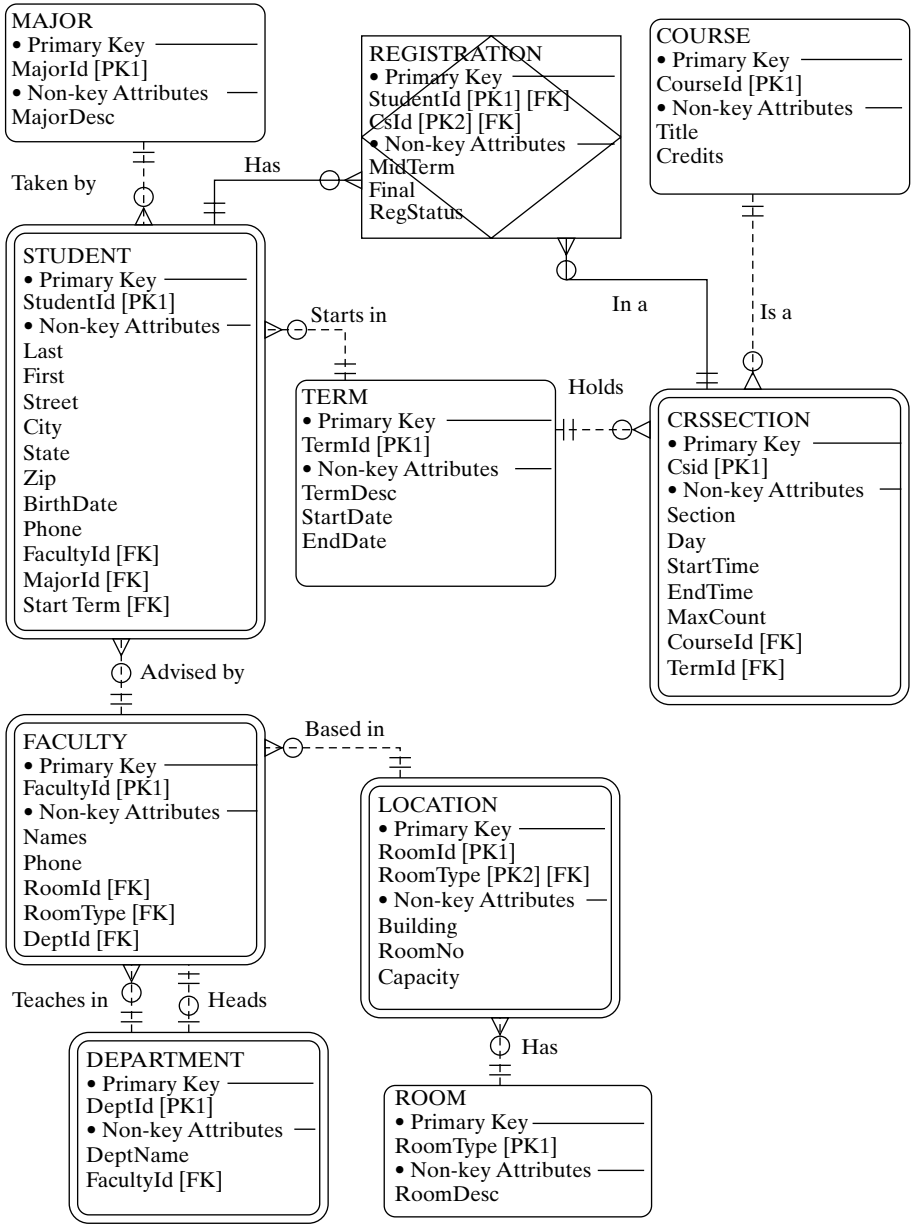


Figure 3-20 Indo–US (IU) College database E-R diagram.

Two of the most important tables are CRSSECTION and REGISTRATION. These tables are related to many other tables in the database, and they grow with each term. The CRSSECTION table contains courses offered during each term. It uses CsId as its primary key. The table references COURSE, TERM, FACULTY,

and LOCATION tables with the foreign keys CourseId, TermId, FacultyId, and RoomId, respectively. The table helps the college administration in flagging each section as *Open* or *Closed* based on the maximum enrollment allowed and the actual current enrollment.

The REGISTRATION table contains each student's schedule for every registration term. It can be used for printing class rosters based on student registration status, for obtaining midterm and final grades, and for grade point averages based on grades obtained. The database also contains three "lookup" tables, which are MAJOR, DEPARTMENT, and ROOM. A lookup table is the one that contains a numeric identification column and another column for its description.

The NamanNavan (N2) Corporation Employee Database

The NamanNavan (N2) Corporation is an up-and-coming name in the Information Systems field. They are distributors of computer hardware and software and providers of computer-related services. Recently, they have joined the Web marketing community with a broad product line. The management feels it is the right time for a changeover from an existing hard copy, paper-only system to a more sophisticated system to track their employees' basic information and the company's organizational structure, payroll, raises, and promotion-related issues. The tables shown in Figure 3-21 are created for the corporation's database, and Figure 3-22 illustrates the E-R diagram.

The N2 Corporation database contains six tables to describe all employee-related information. The EMPLOYEE table describes each employee in the company. Each employee is identified by a unique EmployeeId, which is the primary key for the table. The basic columns for the employee include the employee's last name, first name, immediate supervisor, date of hire for keeping track of anniversary and seniority, yearly salary, commission for the year, and other company-related information. This table includes four foreign keys. PositionId is a foreign key that references a lookup POSITION table to retrieve an employee's position/job title in the company. The DeptId is another foreign key to retrieve a department's name, location, and information about the manager. The third foreign key field, QualId, enables the

EMPLOYEE (EmployeeId, Lname, Fname, PositionId, Supervisor, HireDate, Salary, Commission, DeptId, QualId)

EmployeeId	Lname	Fname	PositionId	Supervisor	HireDate	Salary	Commission	DeptId	QualId
111	Smith	John	1		04/15/60	265000	35000	10	1
246	Houston	Larry	2	111	05/19/67	150000	10000	4	2
123	Roberts	Sandi	2	111	12/02/91	75000		10	2
433	McCall	Alex	3	543	05/10/97	66500		20	4
543	Dev	Derek	2	111	03/15/95	80000	20000	20	1
200	Shaw	Jinku	5	135	01/03/00	24500	3000	30	
135	Garner	Stanley	2	111	02/29/96	45000	5000	30	5
222	Chen	Sunny	4	123	08/15/99	35000		10	3

Figure 3-21 Sample tables for NamanNavan (N2) Corporation's employee database.

POSITION (PositionId, PosDesc)

PositionId	PosDesc
1	President
2	Manager
3	Programmer
4	Accountant
5	Salesman

DEPT (DeptId, DeptName, Location, EmployeeId)

DeptId	DeptName	Location	EmployeeId
10	Finance	Charlotte	123
20	InfoSys	New York	543
30	Sales	Woodbridge	135
40	Marketing	Los Angeles	246

QUALIFICATION (QualId, QualDesc)

QualId	QualDesc
1	Doctorate
2	Masters
3	Bachelors
4	Associates
5	High School

EMPLEVEL (LevelNo, LowSalary, HighSalary)

Level-No	LowSalary	HighSalary
1	1	25000
2	25001	50000
3	50001	100000
4	100001	500000

DEPENDENT (EmployeeId, DependentId, DepDOB, Relation)

EmployeeId	DependentId	DepDOB	Relation
543	1	09/28/58	Spouse
543	2	10/14/88	Son
200	1	06/10/76	Spouse
222	1	02/04/75	Spouse
222	2	08/23/97	Son
222	3	07/10/99	Daughter
111	1	12/12/45	Spouse

Figure 3-21 Sample tables for NamanNavan (N2) Corporation's employee database *(Continued).*

company to keep a record of an employee's highest qualification. The fourth foreign key, Supervisor, references the EmployeeId column in the same table, another candidate for the self-join.

The DEPT table, another important table in the database, includes demographic information about the department's name and primary location as well as the manager responsible for managing day-to-day operation. Each department is identified by a unique DeptId as a primary key. The DEPT table also contains an EmployeeId column as a foreign key to keep information about the department's manager.

The EMPLEVEL table has different grades based on the salary range as defined by the company. An employee belongs to a certain level or grade based on his or her salary. There is no direct relationship between the EMPLEVEL and EMPLOYEE tables, but a type of join called nonequijoin enables user to join these two tables.

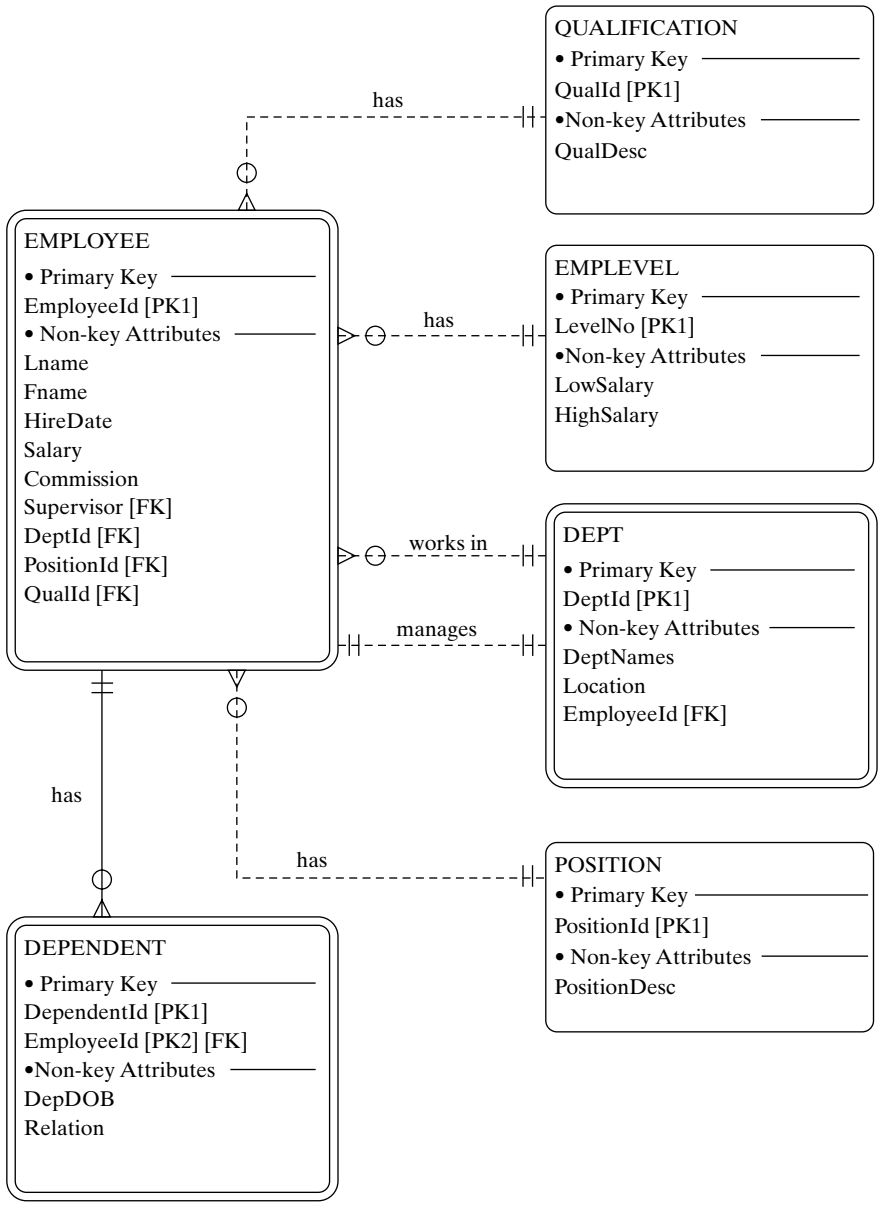


Figure 3-22 N2 Corporation database E-R diagram.

Each employee's dependents for purposes of the health plan and payroll taxes are included in the DEPENDENT table. This table does not have a single column that can be used as primary key. The EmployeeId and DependentId together make up the composite primary key for the table. Age is not used as an column, because it

does not remain the same. Age can be a very high maintenace column, changing every year and for different individuals on different days. The use of DepDOB column eliminates annual maintenance. Each dependent's age can be derived from DepDOB column or birth date by using available date-related functions in Oracle.

The POSITION and QUALIFICATION tables are basically lookup tables for the EMPLOYEE table to get descriptions based on foreign keys in the EMPLOYEE table that are primary keys in the POSITION and QUALIFICATION tables.

The EMPLOYEE, DEPENDENT, and DEPT tables may have more columns, but they are omitted for simplicity. Similarly, another lookup table could have been created to look up relations based on some RelationId column in the DEPENDENT table.

IN A NUTSHELL . . .

- Personal database management systems (DBMSs) are stored on a client computer and are meant for single users.
- A client/server DBMS runs on a server, and user applications run on the client computers.
- Personal databases are characterized by heavy hardware demands, heavy network traffic, database corruption, transaction losses, and poor recovery mechanisms.
- Client/server databases have better recovery mechanisms.
- Client/server databases provide automatic tables and record-level locking.
- Client/server databases provide file-based transaction logging for recovery of transactions in case of a failure.
- Oracle9i is a popular client/server database management system that is based on the relational model.
- The Oracle9i environment provides utilities to work with database tables; developing forms, reports, and graphs; managing users and databases; and interfacing the Web and databases.
- Structured Query Language (SQL) is a standard, nonprocedural language to work with relational database tables.
- SQL*Plus is Oracle's proprietary environment to enter SQL queries.
- SQL is a language with data retrieval, DML, DDL, DCL, and transaction control query statements.
- SQL*Plus is an environment that provides users with editing, file, formatting, execution, and interaction commands.
- The SQL*Plus Worksheet environment is an alternative to the line-based SQL*Plus environment, with full-page editing and GUI features.
- *i*SQL*Plus is a Web-based alternative to SQL*Plus that can be used through a Web browser.

- The IU College's student database system contains student, faculty, course, course section, and registration information. It includes the following tables:
  - STUDENT
  - FACULTY
  - COURSE
  - CRSSECTION
  - REGISTRATION
  - ROOM
  - TERM
  - LOCATION
  - MAJOR
  - DEPARTMENT
- The NamanNavan (N2) Corporation's employee database system contains employee, department, and dependent information in the following tables:
  - EMPLOYEE
  - DEPT
  - POSITION
  - EMPLEVEL
  - QUALIFICATION
  - DEPENDENT

EXERCISE QUESTIONS

1. How are client requests served by a server in a personal database and in a client/server database?
2. How are transactions handled by personal and client/server databases in case of a failure?
3. What are the advantages of client/server databases over personal databases in a multi-user environment?
4. Name various tools provided by the Oracle9i RDBMS.
5. What is SQL? What are the different types of statements a user can write using SQL in Oracle9i?
6. What are the functions of the SQL*Plus environment?
7. Give two examples of database applications appropriate for a personal database system.
8. Give two examples of database applications appropriate for a client/server database system.
9. What are the benefits of the SQL*Plus Worksheet and *i*SQL*Plus environments over the SQL*Plus environment?
10. What login problems may be encountered while logging in to SQL*Plus?
11. Name any three SQL*Plus editing commands.
12. Name any three SQL*Plus file-related commands.

LAB ACTIVITY

1. Log in to Oracle Server at your installation. Locate the *sqlplusw.exe* file on your client computer, and double-click on it or click on

   START | Programs | OraHome92 | Application Development | SQL*Plus

   Ask your DBA/lab personnel/professor for your login username, password, and database name/host string. Soon, you will get the SQL> prompt in the SQL*Plus environment.
2. You have not learned any query statements or SQL*Plus commands yet. Try the following statement at the prompt as given here:

```
SQL>  CREATE TABLE dept (DeptId NUMBER (2),
  2   DeptName VARCHAR2 (15) NOT NULL,
  3   Location VARCHAR2 (12),
  4   EmployeeId NUMBER (4),
  5   CONSTRAINT dept_deptid_pk PRIMARY KEY (deptid));
```

   The line numbers shown on the left are generated by SQL*Plus; they are not entered by the user. At this point, you do not know the CREATE TABLE statement, naming rules, data types, or contraints. Just copy the given statement exactly. If you type this statement without any errors, a "Table created" message will be displayed. If you make a mistake by misspelling a key word or by missing a punctuation mark, use SQL*Plus editing commands to debug the error.
3. Invoke the default full-page editor. What is the default editor?
4. What is displayed in the default editor when invoked?
5. Use the following SQL*Plus command at the SQL> prompt:

   DESCRIBE dept

   (*Note:* the SQL*Plus command does not end with a semicolon.) What is displayed by the command?
6. Use the following SQL*Plus command at the SQL> prompt:

   SHOW USER

   What is displayed by the command?
7. Exit from the SQL*Plus environment. How will you exit? Do you click on X to close window or use an SQL*Plus command?

4

Oracle Tables: Data Definition Language (DDL)

IN THIS CHAPTER . . .

- You will learn about Data Definition Language (DDL) statements to work with the structure of an Oracle database table.
- Various data types used in defining columns in a database table are discussed.
- Integrity and value constraints and their inclusion in a CREATE TABLE statement at the column and table level are outlined.
- Viewing, modifying, and removing a table's structure are also covered.

In the previous chapters, you learned about relational terminology, database modeling, normalization techniques, the SQL*Plus environment, and its commands. Now is the time to put everything together. In Oracle9i, database tables are objects stored under a user's account in an allocated **tablespace** (storage space) on the Oracle Server's disk. Each table under a user's account must have a unique table name. In the classroom environment, each student is a user with a unique login/username. Each object including a table created by a user is stored under that user's schema. In this and subsequent chapters, you will create and use tables for the Indo–US (IU) College and the NamanNavan (N2) Corporation. You will learn to create tables using SQL statements at the SQL*Plus prompt. You will also learn to use alternate editors for easier editing of erroneous statements.

NAMING RULES AND CONVENTIONS

A table is an object that can store data in an Oracle database. When you create a table, you must specify the table name, the name of each column, the data type of each column, and the size of each column. Oracle provides you with different constraints to specify a primary or a composite key for the table, to define a foreign key in a table that references a primary key in another table, to set data validation rules for each column, to specify whether a column allows NULL values, and to specify if a column should have unique values only.

The table and column names can be up to 30 characters long. It is possible to have a table name that is only one character long. In naming tables and columns, letters (A–Z, a–z), numbers (0–9) and special characters—$ (dollar sign), _(underscore), and # (pound or number sign)—are allowed. The table or column name, however, must begin with a letter. The names are not case sensitive, although Oracle stores all object names in uppercase in its data dictionary. Spaces and hyphens are not allowed in a table or a column name. An Oracle server–reserved word cannot be used as a table or a column name. Remember, the most common mistake is the use of a space in naming a table or a column. It is always a good practice to create short but meaningful names. Also, remember that a table name must be unique in a schema or user account; there must not be another Oracle object with same name in a schema. Figure 4-1 shows some valid and invalid table and column names. For invalid names, the reasons are in parentheses.

Valid Names	Invalid Names
STUDENT	STUDENT_COURSE_REGISTRATION_TABLE (more than 30 characters long)
MAJOR_CODE	MAJOR CODE (spaces not allowed)
X	CREATE (reserved word not allowed)
PROJECT2000	PROJECT***2000 (special character * not allowed)
STUDENT#REG#TABLE	#STUDENT (must start with a letter)

Figure 4-1 Valid/invalid table and column names.

DATA TYPES

When a table is created, each column in the table is assigned a data type. A data type specifies the type of data that will be stored in that column. By providing a data type for a column, the wrong kinds of data are prevented from being stored in the column. For example, a name such as Smith cannot be stored in a column with a NUMBER data type. Similarly, a job title such as Manager cannot be stored in a column

with a DATE data type. Data types also help to optimize storage space. Some of the Oracle data types are described below.

Varchar2

The VARCHAR2 type is a character data type to store variable-length alphanumeric data in a column. Currently, VARCHAR is synonymous with VARCHAR2, but it could be a separate data type with different semantics in the future. Users are advised to use VARCHAR2 only. A maximum size must be specified for this type. The default and minimum size is one character. The maximum allowable size is 4000 characters in Oracle9i. (The maximum size was 2000 characters in previous versions.) The size is specified within parentheses—for example, VARCHAR2(20). If the data are smaller than the specified size, only the data value is stored, and trailing spaces are not added to the value. For example, if a column NAME is assigned a data type VARCHAR2(25) and the name entered is Steve Jones, only 11 characters are stored. Fourteen spaces are not added to make its length equal to the size of the column. If a value longer than the specified size is entered, however, an error is generated. The longer values are not truncated. VARCHAR2 is the most appropriate type for a column whose values do not have a fixed length.

In Oracle9i, the VARCHAR2 data type can also take CHAR or BYTE parameters. For example, VARCHAR2(10 BYTE) is same as VARCHAR2(10) because byte is the default. If VARCHAR2(10 CHAR) is used, each CHAR may take up 1 to 4 bytes. In this text, you will see the default semantic only.

Char

The CHAR type is a character data type to store fixed-length alphanumeric data in a column. The default and minimum size is one character. The maximum allowable size is 2000 characters. (This was only 255 characters in previous versions.) If the value is smaller than the specified size is entered, trailing spaces are added to make its length equal to the specified length. If the value is longer than the specified size, an error occurs. The CHAR type is appropriate for fixed-length values. For example, PHONE, SOCIAL_SECURITY_NUMBER, or MIDDLE_INITIAL columns can use the CHAR type. The phone numbers and Social Security numbers have numeric values, but they also use special characters, such as hyphens and parentheses. Both use fixed-length values, however, so CHAR is the most appropriate type for them. The CHAR data type uses the storage more efficiently and processes data faster than the VARCHAR2 type.

In Oracle9i, the CHAR data type can also take CHAR or BYTE parameters. For example, CHAR(10 BYTE) is same as CHAR(10) because byte is the default. If CHAR(10 CHAR) is used, each CHAR may take up 1 to 4 bytes. In this text, you will see the default semantic only.

Number

The NUMBER data type is used to store negative, positive, integer, fixed-decimal, and floating-point numbers. The NUMBER data type is used for any column that is going to be employed in mathematical calculations—for example, SALARY, COMMISSION, or PRICE. When a number type is used for a column, its **precision** and **scale** can be specified. Precision is the total number of significant digits in the number, both to the left and to the right of the decimal point. The decimal point is not counted in specifying the precision. Scale is the total number of digits to the right of the decimal point. The precision can range from 1 to 38. The scale can range from −84 to 127.

An **integer** is a whole number without any decimal part. To define a column with integer values, only the scale size is provided. For example, EmployeeId in the EMPLOYEE table has values of 111, 246, 123, 433, and so on. The data type for it would be defined as NUMBER(3), where 3 represents the maximum number of digits. Remember to provide room for future growth when defining the size. If a corporation has up to 999 employees, a size of 3 will work for now. With future growth, the corporation's number of employees may rise to 1000 or higher. By using a size of 4, you provide room for up to 9999 employees.

A **fixed-point** decimal number has a specific number of digits to the right of the decimal point. The PRICE column has values in dollars and cents, which requires two decimal places—for example, values like 2.95, 3.99, 24.99, and so on. If it is defined as NUMBER(4,2), the first number specifies the precision and the second number the scale. Remember that the decimal place is not counted in the scale. The given definition will allow a maximum price of 99.99.

A **floating-point** decimal number has a variable number of decimal places. The decimal point may appear after any number of digits, and it may not appear at all. To define such a column, do not specify the scale or precision along with the NUMBER type. For example, TAXRATE, INTEREST_RATE, and STUDENT_GPA columns are likely to have variable numbers of decimal places. By defining a column as a floating-point number, a value can be stored in it with very high precision.

Date

The DATE data type is used for storing date and time values. The range of allowable dates is between January 1, 4712 B.C. and December 31, 9999 A.D. The day, month, century, hour, minute, and second are stored in the DATE-type column. There is no need to specify size for the DATE type. The default date format is DD-MON-YY, where DD indicates the day of the month, MON represents the month's first three letters (capitalized), and YY represents the last two digits of the year. These three values are separated by hyphens. The DD-MON-YYYY format also works as the default in Oracle9i. To use any other format to enter a date value, you are required to use the TO_DATE function. The default time format is HH:MM:SS A.M.,

representing hours, minutes and seconds in a 12-hour time format. If only a date is entered, the time defaults to 12:00:00 A.M. If only a time is entered, the date defaults to the first day of the current month. For example, HIREDATE for EmployeeId 111 in the EMPLOYEE table in the N2 Corporation database is stored as 15-APR-60 12:00:00 A.M.

In a table, it is not advisable to use columns like AGE, because age not only changes for all entities but also changes at different times. A column like AGE can become a very high-maintenance column. It is advisable to use BIRTHDATE as a column instead. Oracle9i provides users with quite a few built-in date functions for date manipulation. Just simple date arithmetic is enough to calculate age from the birth date! The birth date never changes, so no maintenance on it is necessary.

Other advanced data types used in Oracle are not used in the sample databases discussed in Chapter 3. These advanced data types are outlined here for your information only:

LONG. The LONG type is used for variable-length character data up to 2 gigabytes. There can be only one LONG-type column in a table. It is used to store a memo, invoice, or student transcript in the text format. When defining to LONG type, there is no need to specify its size.

NCHAR. The NCHAR type is similar to CHAR but uses 2-byte binary encoding for each character. The CHAR type uses 1-byte ASCII encoding for each character, giving it the capability to represent 256 different characters. The NCHAR type is useful for character sets such as Japanese Kanji, which has thousands of different characters.

CLOB. The Character Large Object data type is used to store single-byte character data up to 4 gigabytes.

BLOB. The Binary Large Object data type is used to store binary data up to 4 gigabytes.

NCLOB. The character Large Object type uses 2-byte character codes.

BFILE. The Binary File type stores references to a binary file that is external to the database and is maintained by the operating system's file system.

RAW(size) or **LONG_RAW.** These are used for raw binary data.

ROWID. For unique row address in hexadecimal format.

Many of the Large Object (LOB) data types are not supported by all versions of Oracle and its tools. These data types are used for storing digitized sounds, for images, or to reference binary files from Microsoft Excel spreadsheets or Microsoft Word documents. We will not use LOB data types in this book. Figure 4-2 shows a brief summary of Oracle data types and their use in storing different types of data.

Data Type	Use
VARCHAR2 (size)	Variable-length character data: 1 to 4000 characters
CHAR (size)	Fixed-length character data: 1 to 2000 characters
NUMBER (p)	Integer values
NUMBER (p, s)	Fixed-point decimal values
NUMBER	Floating-point decimal values
DATE	Date and time values
LONG	Variable-length character data up to gigabytes
NCHAR	Similar to CHAR; uses 2-byte encoding
BLOB	Binary data up to 4 gigabytes
CLOB	Single-byte character data up to 4 gigabytes
NCLOB	Similar to CLOB; supports 2-byte encoding
BFILE	Reference to an external binary file
RAW (size)	Raw binary data up to 2000 bytes
LONG_RAW	Same as RAW; stores up to 2 gigabytes
ROWID	Unique address of a row in a table

Figure 4-2 Data types and their use.

CONSTRAINTS

Constraints enforce rules on tables. An Oracle table can be created with the column names, data types, and column sizes, which are sufficient just to populate them with actual data. Without constraints, however, no rules are enforced. The constraints help you to make your database one with integrity. We learned the integrity rules in Chapter 1. The constraints are used in Oracle to implement integrity rules of a relational database and to implement data integrity at the individual-column level. Whenever a row/record is inserted, updated, or deleted from the table, a constraint must be satisfied for the operation to succeed. A table cannot be deleted if there are dependencies from other tables in the form of foreign keys.

Types of Constraints

There are two types of constraints:

1. ***Integrity constraints:*** define both the primary key and the foreign key with the table and primary key it references.
2. ***Value constraints:*** define if NULL values are disallowed, if UNIQUE values are required, and if only certain set of values are allowed in a column.

Naming a Constraint

Oracle identifies constraints with an internal or user-created name. For a user's account, each constraint name must be unique. A user cannot create constraints in two

different tables with the same name. The general convention used for naming constraints is

<table name>_<column name>_<constraint type>

Here, *table name* is the name of the table where the constraint is being defined, *column name* is the name of the column to which the constraint applies, and *constraint type* is an abbreviation used to identify the constraint's type. Figure 4-3 shows popular abbreviations used for the constraint type.

Constraint	Abbreviation
PRIMARY KEY	pk
FOREIGN KEY	fk
UNIQUE	uk
CHECK	ck or cc
NOT NULL	nn

Figure 4-3 Popular constraint abbreviations.

For example, a constraint name *emp_deptno_fk* refers to a constraint in table EMP on column DeptNo of type foreign key. A constraint name *dept_deptno_pk* is for a primary key constraint in table DEPT on column DeptNo.

If you do not name a constraint, the Oracle server will generate a name for it by using *SYS_Cn* format, where *n* is any unique number. For example, *SYS_C000010* is an Oracle server–named constraint. These names are not user friendly like user-named constraints.

Defining a Constraint

A constraint can be created at the same time the table is created, or it can be added to the table afterward. There are two levels where a constraint is defined:

1. ***Column level:*** A column-level constraint references a single column and is defined along with the definition of the column. Any constraint can be defined at the column level except for a FOREIGN KEY and composite primary key constraints. The general syntax is

   Column datatype [CONSTRAINT constraint_name] constraint_type,

   (In this book, you will see the following convention for syntax: Reserved words will be written in uppercase and user-defined identifiers in lower or mixed case. Optional parts will be within brackets ([ ]). The pipe symbol (|) will represent OR situations in statement syntax.)

2. ***Table level:*** A table-level constraint references one or more columns and is defined separately from the definitions of the columns. Normally, it is written after all columns are defined. All constraints can be defined at the table level except for the NOT NULL constraint. The general syntax is:

```
[CONSTRAINT constraint_name] constraint_type(Column, . . .),
```

The PRIMARY KEY Constraint. The PRIMARY KEY Constraint is also known as the entity integrity constraint. It creates a primary key for the table. A table can have only one primary key constraint. A column or combination of columns used as a primary key cannot have a null value, and it can only have unique values. For example, the DEPT table in the N2 Corporation database used the DeptId column as a primary key. At the column level, the constraint is defined by

```
DeptId NUMBER (2) CONSTRAINT dept_deptid_pk PRIMARY KEY,
```

At the table level; the constraint is defined by

```
CONSTRAINT dept_deptid_pk PRIMARY KEY(DeptId),
```

If a table uses more than one column as its primary key (i.e., a composite key), the key can only be declared at the table level. For example, the DEPENDENT table in the N2 database uses two columns for the composite primary key:

```
CONSTRAINT dependent_emp_dep_pk PRIMARY KEY(EmployeeId, DependentId),
```

The FOREIGN KEY Constraint. The FOREIGN KEY constraint is also known as the referential integrity constraint. It uses a column or columns as a foreign key, and it establishes a relationship with the primary key of the same or another table. For example, FacultyId in the STUDENT table in the IU College database references the primary key FacultyId in the FACULTY table. The STUDENT table is known as the dependent or child table, and the FACULTY table is known as the referenced or parent table.

To establish a foreign key in a table, the other referenced table and its primary key must already exist. Foreign key and referenced primary key columns need not have the same name, but a foreign key value must match the value in the parent table's primary key value or be NULL. For example, the foreign key FacultyId cannot have value 999 in the STUDENT table, because it does not exist in the FACULTY (parent) table's primary key FacultyId.

Oracle does not keep pointers for relationships, but they are based on constraints and data values within those columns. The relationship is purely logical and is not physical in Oracle. At the table level (in the STUDENT table),

```
CONSTRAINT student_facultyid_fk FOREIGN KEY(FacultyId) REFERENCES faculty(FacultyId),
```

Before ending a FOREIGN KEY constraint, ON DELETE CASCADE can be added to allow deletion of a record/row in the parent table and deletion of the

dependent rows/records in the child table. Without the ON DELETE CASCADE clause, the row/record in the parent table cannot be deleted if the child table references it. For example, the row for FacultyId 111 cannot be deleted from the FACULTY table, because it is referenced by a row in the STUDENT table.

The NOT NULL Constraint. The NOT NULL constraint ensures that the column has a value and the value is not a null (unknown or blank) value. A space or a numeric zero is not a null value. There is no need to use the not null constraint for the primary key column, because it automatically gets the not null constraint. The foreign key is permitted to have null values, but a foreign key is sometimes given the not null constraint. This constraint cannot be entered at the table level. For example, the name column in FACULTY table is not a key column, but you don't want to leave it blank. At the column level, the constraint is defined by:

```
Name VARCHAR2(15) CONSTRAINT faculty_name_nn NOT NULL,
```

or

```
Name VARCHAR2(15) NOT NULL,
```

In the second example, the user does not supply the constraint name, so Oracle will name it with SYS_C*n* format.

The UNIQUE Constraint. The UNIQUE constraint requires that every value in a column or set of columns be unique. If it is applied to a single column, the column has unique values only. If it is applied to a set of columns, the group of columns has a unique value together. The unique constraint allows null values unless NOT NULL is also applied to the column. For example, the DeptName column in the DEPT table should not have duplicate values. At the table level, the constraint is defined by:

```
CONSTRAINT dept_dname_uk UNIQUE(DeptName),
```

At the column level, the constraint is defined by

```
DeptName VARCHAR2(12) CONSTRAINT dept_dname_uk UNIQUE,
```

The composite unique key constraint can be defined only at the table level by specifying column names separated by a comma within parentheses. Oracle implicitly creates an index on the unique column to enforce the UNIQUE constraint.

The CHECK Constraint. The CHECK constraint defines a condition that every row must satisfy. There can be more than one CHECK constraint on a column, and the CHECK constraint can be defined at the column as well as the table level. At the column level, the constraint is defined by

```
DeptId NUMBER (2) CONSTRAINT dept_deptid_cc CHECK((DeptId >= 10) and (DeptId <= 99)),
```

At the table level, the constraint is defined by

```
CONSTRAINT dept_deptid_cc CHECK((DeptId >= 10) and (DeptId <= 99)),
```

The NOT NULL CHECK Constraint. A NOT NULL constraint can be declared as a CHECK constraint. Then, it can be defined at column or table level. For example,

```
Name VARCHAR2(15) CONSTRAINT faculty_name_ck CHECK(Name IS NOT NULL),
```

The DEFAULT Value (It's Not a Constraint). The DEFAULT value ensures that a particular column will always have a value when a new row is inserted. The default value gets overwritten if a user enters another value. The default value is used if a null value is inserted. For example, if most of the students live in New Jersey, "NJ" can be used as a default value for the State column in the STUDENT table. At the column level, the value is defined by:

```
State CHAR(2) DEFAULT 'NJ',
```

CREATING AN ORACLE TABLE

A user creates an Oracle table in the SQL*Plus environment. You will run the Oracle Client application from your PC as described in Chapter 3. An Oracle table is created from the SQL> prompt in the SQL*Plus environment. A Data Definition Language (DDL) SQL statement, CREATE TABLE, is used for table creation. A table is created as soon as the CREATE statement is successfully executed by the Oracle server. The general syntax of CREATE TABLE statement is:

```
CREATE TABLE [schema.] tablename
(column1 datatype [CONSTRAINT constraint_name] constraint_type. . .,
column2 datatype [CONSTRAINT constraint_name] constraint_type,
[CONSTRAINT constraint_name] constraint_type (column, . . . ),. . . );
```

In the syntax,

Schema is optional, and it is same as the user's login name.

Tablename is the name of the table given by the user.

Column is the name of a single column.

Datatype is the column's data type and size.

Constraint_name is the name of constraint provided by the user as per the conventions discussed earlier in this chapter.

Constraint_type is the integrity or value constraint.

Each column may have zero, one, or more constraints defined at the column level. The table level constraints are normally declared after all column definitions.

SQL is not case sensitive. In this textbook, the reserved words are written in capitalized letters and user-defined names in lower or mixed-case letters. The spaces,

tabs, and carriage returns are ignored. Let us create the LOCATION table in the IU College database using the CREATE TABLE statement. When the statement is executed and there are no syntax errors, a "Table Created" message will be displayed on the screen (see Fig. 4-4).

```
SQL>        CREATE TABLE location
  2         (RoomId NUMBER(2),
  3         Building VARCHAR2(7) CONSTRAINT location_building_nn NOT NULL,
  4         RoomNo CHAR(3) CONSTRAINT location_roomno_nn NOT NULL,
  5         Capacity NUMBER(2)
  6              CONSTRAINT location_capacity_ck CHECK(Capacity>0),
  7         RoomType CHAR,
  8         CONSTRAINT location_roomid_pk PRIMARY KEY(RoomId),
  9         CONSTRAINT location_roomno_uk UNIQUE(RoomNo);

Table created.

SQL>
```

Figure 4-4 CREATE TABLE statement.

If there are errors in the CREATE TABLE statement, the statement does not return the "Table Created" message when executed. Oracle displays an error message instead (see Fig. 4-5). The error messages are not very userfriendly. In the statement

```
SQL>          CREATE TABLE emplevel   (LevelNo NUMBER(1),
  2         LowSalaray      Number(6),
  3         HighSalary      Number(6)
  4         CONSRAINT   emplevel_levelno_pk PRIMARY KEY(LevelNo));
CREATE TABLE emplevel   (LevelNo NUMBER(1),

ERROR at line 1:
ORA-00922: missing or invalid option
SQL>  3
  3*        HighSalary      Number(6)
SQL>  A,
  3*        HighSalary      Number(6),
SQL> /
  CONSRAINT      emplevel_levelno_pk PRIMARY KEY(LevelNo))

ERROR at line 4:
ORA-00907: missing right parenthesis
SQL> C/CONSRAINT/CONSTRAINT/
  4*        CONSTRAINT    emplevel_levelno_pk PRIMARY KEY(LevelNo))
SQL> /

Table Created.

SQL>
```

Figure 4-5 CREATE TABLE statement with error.

shown, the column definition in line 3 is missing a comma—but the error message does not really tell us that! We will discuss error codes and messages later in this chapter.

We will debug the statement using SQL*Plus commands. The error is in line 3, and we will perform the following steps (see Fig. 4-5):

1. Go to line 3 (* is displayed next to the current line number).
2. Replace the character) in line 3 with), or append a comma (,) to the line.
3. Execute the debugged statement using a slash (/).

As you see in Figure 4-5, the statement has another error, this time in line 4. We use *C/CONSRAINT/CONSTRAINT* to change the incorrect spelling. Then, we execute the statement from buffer by entering a slash (/) again. There we go! Table is created. We can edit erroneous statement with the help of an alternate editor, such as Notepad. To load an erroneous statement in Notepad and modify it, we perform the following steps:

1. At the SQL> prompt, we type **ED** (or **EDIT**) to invoke Notepad.
2. We make required corrections to the script.
3. We save our statement on the disk using the Save option from the File menu in Notepad, and we name our statement A:\CREATE. Notepad adds the extension *.txt* to the filename. To suppress Notepad's default *.txt* extension, type the file name in a pair of double quotes, and use the extension .sql (e.g., "A:\CREATE.SQL").
4. We exit Notepad to go back to the SQL*Plus environment.
5. We can run the saved statement with @ or the RUN command.

The "Table Created" message is displayed when the statement is error-free. At this point, the table is created, and its structure is saved. We created the LOCATION and EMPLEVEL tables with PRIMARY KEY, UNIQUE, CHECK, and NOT NULL constraints. Once a table is created, more constraints can be added, more columns can be added, and existing columns' properties can be changed. We did not define any foreign key constraints with the LOCATION table, because the table referenced by the foreign key must already exist!

STORAGE Clause in CREATE TABLE

A CREATE TABLE statement may have an optional STORAGE clause. This clause is used to allocate initial disk space for the table at the time of creation with the INITIAL parameter and also to allocate additional space with the NEXT parameter in case the table runs out of allocated initial space. For example,

```
CREATE TABLE sample (Id NUMBER(3), Name VARCHAR2(25))
TABLESPACE CIS_DATA
STORAGE (INITIAL 1M NEXT 100K)
PCTFREE 20;
```

In the previous example, the TABLESPACE clause is used to specify the user's tablespace name. If it is not specified, Oracle uses the default permanent tablespace anyway. The STORAGE clause allocates 1 megabyte initially on tablespace CIS_DATA, and 100 kilobytes as additional space on the same tablespace. The INITIAL and NEXT parameters use values in K (kilobytes) or M (megabytes). The PCTFREE (percentage- free) clause is used to allow for future increment in row size. Oracle recommends the following formula in deciding initial extent size for a table:

$$AVG\_ROW\_LEN \cdot Number\ of\ rows \cdot (1 + 0.15) \cdot (1 + PCTFREE/100)$$

The AVG_ROW_LEN is a column in USER_TABLES Data Dictionary table. The 0.15 (or 15%) is recommended for overhead.

DISPLAYING TABLE INFORMATION

When a user creates a table or many tables in his or her database, Oracle tracks them all using its own Data Dictionary. Oracle has SQL statements and SQL*Plus commands for the user to view that information from Oracle's Data Dictionary tables.

Viewing a User's Table Names

A user types an SQL statement to retrieve his or her table names. Often, you use it to review information, and often, you want to find out what has already been created and what is to be created. To find out all tables owned by you, type the following statement:

```
SELECT TABLE_NAME FROM USER_TABLES;
```

Oracle creates system tables to store information about users and user objects. USER_TABLES is an Oracle system database table, and TABLE_NAME is one of its columns. The display will include all table names you have created and any other tables that belong to you. If you change USER_TABLES with ALL_TABLES, you can get listing of all tables you own as well as those you are granted privileges to by other users. The USER_ TABLES table has many other columns. To display all columns, type the following statement:

```
SELECT * FROM USER_TABLES;
```

(In this case, You will see more information than you need. The display rows will wrap many times to show all columns/information related to each table.)

You can get information about the STORAGE clauses' atributes by using the Data Dictionary view USER_SEGMENTS:

```
SELECT Segment_Name, Bytes, Blocks, Initial_Extent, Next_Extent
FROM USER_SEGMENTS;
```

Viewing a Table's Structure

You can display the structure entered by you in a CREATE TABLE statement. If you have made any changes to the table's structure, the changes will also show in the structure's display. Figure 4-6 shows the SQL*Plus command to view a table's structure. The command is DESCRIBE (or DESC), which does not need a semicolon at the end because it is not a SQL statement. Notice that the default display of column names, the NOT NULL constraint, and data type are in uppercase. You did not add a NOT NULL constraint for the primary key, but by default, Oracle adds it for all primary key columns.

```
SQL>  DESCRIBE student
 Name                                      Null?          Type
 ----------------------------------------- -------------- ---------------

 STUDENTID                                 NOT NULL       CHAR (5)
 LAST                                      NOT NULL       VARCHAR2 (15)
 FIRST                                     NOT NULL       VARCHAR2 (15)
 STREET                                                   VARCHAR2 (25)
 CITY                                                     VARCHAR2( 15)
 STATE                                                    CHAR (2)
 ZIP                                                      CHAR (5)
 STARTTERM                                                CHAR (4)
 BIRTHDATE                                                DATE
 FACULTYID                                                NUMBER (3)
 MAJORID                                                  NUMBER (3)
 PHONE                                                    CHAR (10)

SQL>
```

Figure 4-6 DESCRIBE command and table structure.

Viewing Constraint Information

Oracle's Data Dictionary table USER_CONSTRAINTS stores information about constraints you have entered for each column. Figure 4-7 shows the statement and the result, which include the constraint's name and type. When you type the statement, the table name must be typed in uppercase, because Oracle saves table names in uppercase. If you type the table name in lowercase, no constraint names will be displayed.

The constraints named by the user have more meaningful names than the ones named by Oracle. Constraints like NOT NULL are usually not named by the user. Oracle names them using the SYS_C*n* format, where *n* is any number. Constraint type C is displayed for NOT NULL and CHECK constraints. Constraint type P is for primary key and type R for foreign key constraints. You will type only the first two lines of the statement in Figure 4-7 to display all constraints in your account.

```
SQL>  SELECT CONSTRAINT_NAME,  CONSTRAINT_TYPE
  2   FROM USER_CONSTRAINTS
  3   WHERE TABLE_NAME = 'STUDENT';
CONSTRAINT_NAME                C
------------------------------
STUDENT_FIRST_NN               C
STUDENT_STUDENTID_PK           P
STUDENT_FACULTYID_FK           R
STUDENT_MAJORID_FK             R
STUDENT_STARTTERM_FK           R
STUDENT_LAST_NN                C

6 rows selected

SQL>
```

Figure 4-7 Constraint information.

The ORDER BY clause is added to sort the constraint display by table name. For example,

```
SELECT CONSTRAINT_NAME, CONSTRAINT_TYPE, TABLE_NAME
    FROM USER_CONSTRAINTS
    ORDER BY TABLE_NAME;
```

Another Data Dictionary table, USER_CONS_COLUMNS, stores column-related information about constraints. You can use the statement shown in Figure 4-8 to see the constraint names and associated column names. In this statement, only the table name within single quotes needs to be in uppercase, because Oracle stores table names in that case.

```
SQL>  COLUMN COLUMN_NAME FORMAT A15
SQL>  SELECT CONSTRAINT_NAME, COLUMN_NAME
   2  FROM USER_CONS_COLUMNS
   3  WHERE TABLE_NAME = 'STUDENT';
CONSTRAINT_NAME            COLUMN_NAME
-------------------------------------
STUDENT_FACULTYID_FK       FACULTYID
STUDENT_FIRST_NN           FIRST
STIDENT_LAST_NN            LAST
STUDENT_MAJORID_FK         MAJORID
STUDENT_STARTTERM_FK       STARTTERM
STUDENT_STUDENTID_PK       STUDENTID

6 rows selected.

SQL>
```

Figure 4-8 Constraint column information.

Viewing Tablespace Information

A tablespace consists of one or more physical data files. You can get information about all tablespaces available to you by using Data Dictionary view USER_TABLESPACES. We will discuss types of views and the Oracle Data Dictionary in later chapters. You can use the DESCRIBE command and the SELECT statement with a view the same way you use with tables. For example,

```
DESCRIBE USER_TABLESPACES
SELECT * FROM USER_TABLESPACES;
```

Similarly, another Data Dictionary view, USER_USERS, gives user information about his or her account as well as permanent and temporary tablespaces. For example,

```
SELECT * FROM USER_USERS;
```

COMMENT on Tables and Columns

When you create a table, you can add comments to the table and its columns. You can do it for documentation purpose with a COMMENT statement. For example,

```
COMMENT ON TABLE student IS 'Table holds students for INDO-US College'
COMMENT ON COLUMN employee.Lname IS 'Employee"s last name'
```

You can view information about all comments on tables and columns by using Data Dictionary views ALL_TAB_COMMENTS and ALL_COL_COMMENTS, respectively.

ALTERING AN EXISTING TABLE

In a perfect scenario, the table you create will not need any structural modifications. You must try to plan and design a database that is close to perfect in all respects. In reality, however, this is not the case. Even perfect tables need changes. There are certain modifications that you can make to a table's structure. There are other modifications that you cannot make to an existing table's structure.

Modifications allowed without any restrictions include:

- Adding a new column to the table.
- Deleting a foreign key constraint from a table.
- Deleting a primary key constraint from a table, which also removes any references to it from other tables in the database with CASCADE clause.
- Increasing the size of a column. For example, VARCHAR2(15) can be changed to VARCHAR2(20).
- Renaming columns (Oracle9i onward).
- Renaming constraints (Oracle9i onward).

Modifications allowed with restrictions include:

- Adding a foreign key constraint is allowed only if the current values are null or exist in the referenced table's primary key.
- Adding a primary key constraint is allowed if the current values are not null and are unique.
- Changing a column's data type and size is allowed only if there is no data in it (Oracle8i and earlier). In Oracle9i, column size may be decreased if existing data can be stored with the new column width.
- Adding a unique constraint is possible if the current data values are unique.
- Adding a check constraint is possible if the current data values comply with the new constraint.
- Adding a default value is possible if there is no data in the column.

Modifications not allowed include:

- Changing a column's name (Oracle8i and earlier).
- Changing a constraint's name (Oracle8i and earlier).
- Removing a column (Oracle8 and earlier).

In Oracle8i onward, you are allowed to remove/drop a column from a table or set it as unused. If you already have created a table and need to make a change that is not allowed, you may DROP the table and recreate it. You will also learn in a later chapter that a table can be created using another table with the use of a nested query.

Adding a New Column to an Existing Table

The general syntax to add a column to an existing table is

ALTER TABLE tablename

ADD columnname datatype;

For example, if the IU College decides to track a student's Social Security number along with the student's ID, a new column can be added to the STUDENT table, as shown in Figure 4-9. If the table already contained rows of data, you will have to use UPDATE statement (covered in the next chapter) for each row to add values in the newly added column.

```
SQL>   ALTER TABLE student
  2    ADD SocialSecurity   CHAR(9);

Table altered.

SQL>
```

Figure 4-9 ALTER TABLE—adding a column.

Modifying an Existing Column

The general syntax to modify an existing column is

```
ALTER TABLE tablename
    MODIFY columnname newdatatype;
```

where *newdatatype* is the new data type or the new size for the column. For example, say the IU College wants to allow data-entry personnel to enter values with or without dashes in the Social Security column. The data type can be changed from CHAR(9) to VARCHAR2(11) to accommodate this new format (see Fig. 4-10).

```
SQL>  ALTER TABLE student
  2   MODIFY SocialSecurity VARCHAR2(11);

Table altered.

SQL>
```

Figure 4-10 ALTER TABLE—modifying a column.

Adding a Constraint

In this section, we will try to add various constraints in a table using the ALTER TABLE statement. As introduced in Chapter 3, the EMPLOYEE table in the N2 corporation database has a PositionId column, which references the POSITION table's primary key PositionId. To add a constraint using ALTER TABLE, the syntax for table level constraint is used. The general syntax of ALTER TABLE is

```
ALTER TABLE tablename
    ADD [CONSTRAINT constraint_name] constraint_type (column, ...),
```

For example,

```
ALTER TABLE employee
ADD CONSTRAINT employee_positionid_fk  FOREIGN KEY (PositionId)
      REFERENCES position (PositionId);
```

Figure 4-11a shows the ALTER TABLE statement that adds a new constraint

```
SQL>  ALTER TABLE course
  2   ADD CONSTRAINT COURSE_PREREQ_FK FOREIGN KEY(PREREQ)
  3         REFERENCES COURSE(COURSE_ID);

Table altered.

SQL>
```

Figure 4-11a ALTER TABLE—adding a constraint.

to table COURSE. The foreign key column PreReq references primary key column CourseId of its own table. Such a reference is known as a **circular reference**.

The TERM table in the IU College database contains two columns, StartDate and EndDate. The start date for a term must fall before the end date for the same term. Use of a CHECK constraint will guarantee the necessary data integrity. The problem is that during creation of the TERM table, defining a constraint that compares values in two columns of the same table is not possible. The constraint can be defined with the ALTER TABLE statement, however, as shown in Figure 4-11b.

```
SQL>   ALTER TABLE term
  2    ADD CONSTRAINT term_startdate_ck
  3    CHECK(StartDate < EndDate);

Table altered.

SQL>
```

Figure 4-11b ALTER TABLE—adding a CHECK constraint.

Let us try to add another foreign key constraint in the STUDENT table as shown in Figure 4-12. As you see in Figure 4-12a, this error message is easier to understand. To create a foreign key constraint, the parent table, whose primary key column is referenced by the child table's foreign key column, must already exist in the database. Even the primary key column that is referenced must exist in the parent table defined as the primary key. Remember that the two columns, the foreign key and the

```
SQL>   ALTER TABLE student
  2    ADD CONSTRAINT student_facultyid_fk
  3         FOREIGN KEY (FacultyId) REFERENCES faculty(FacultyId);
REFERENCES faculty(FacultyId)
                *

ERROR at one 3:
ORA-00942: table or view does not exist
SQL>
```

Figure 4-12a ALTER TABLE—unsuccessful.

primary key that it references, need not have the same name. The best solution in this situation would be to create all tables without any foreign key constraints first, then create tables using a CREATE TABLE statement with FOREIGN KEY constraints to the reference tables already created. An alternate solution is to create all tables with their constraints except for the foreign key constraint. Once all the tables are created, use the ALTER TABLE statement to add the FOREIGN KEY constraint.

In Figure 4-12b, another try to create a foreign key has failed. The problem with the query is the creation of a foreign key in the wrong table. The FacultyId column is common in the STUDENT and FACULTY tables, but remember the rule! A

```
SQL>  ALTER TABLE faculty
  2   ADD CONSTRAINT faculty_facultyid_fk FOREIGN KEY(FacultyId)
  3   REFERENCES student(FacultyId);
 REFERENCES student(FacultyId)
                   *
ERROR at line 3:
ORA-00270; no matching unique or key for this column-list

SQL>
```

Figure 4-12b ALTER TABLE—unsuccessful.

foreign key must reference a primary key. In our query, primary key FacultyId in the FACULTY table is trying to reference a non–key column FacultyId in the STUDENT table. It will definitely won't work in Oracle!

Once the parent table FACULTY is created, the foreign key is successfully created in the STUDENT table (see Fig. 4-12c).

```
SQL>  ALTER TABLE student
  2   ADD CONSTRAINT student_facultyid_fk FOREIGN KEY(FacultyId)
  3   REFERENCES Faculty(FacultyID);

Table altered.

SQL>
```

Figure 4-12c ALTER TABLE—successful.

Now, let us add a NOT NULL constraint and a DEFAULT value to the StartTerm and State columns, respectively, in the STUDENT table. If a student does not have a start term, it is difficult for an academic department to track the student's class and projected date of graduation. If the college is located in the New Jersey area and most of the students are from in-state, it is a good idea to add a default value to minimize having to enter data. A user can always overwrite the default value, but if it is left blank or null, the default value is used by Oracle. To add such constraints, a MODIFY clause is used with an ALTER TABLE statement. For example,

```
ALTER TABLE student MODIFY StartTerm CHAR(4)
CONSTRAINT student_startterm_nn NOT NULL;
ALTER TABLE student MODIFY State CHAR(2)
DEFAULT 'NJ';
```

Dropping a Column (Oracle8i Onward)

As you already know, Oracle8 and earlier versions do not allow you to remove a column from a table, but with Oracle8i onward, you can. Even so, only one column can be dropped at a time. The column may or may not contain any data. When you drop

a column, there must be at least one column left in the table. In other words, you can't remove the last remaining column from a table. It is not possible to recover a dropped column and its data. The general syntax is

```
ALTER TABLE tablename DROP COLUMN columnname;
```

Oracle9i also allows a user to mark columns as unused by using

```
ALTER TABLE tablename SET UNUSED(columnname);
```

The unused columns are like dropped columns. This is not a very good feature, because the storage space used by unused columns is not released. They are not displayed with other columns or in the table's structure, and the user can drop all unused columns with the following statement. Setting a column to unused is quicker than dropping a column, however, and it requires fewer system resources. You can remove all unused columns when system resources are in less demand. The general syntax is

```
ALTER TABLE tablename DROP UNUSED COLUMNS;
```

If no columns are marked as unused, this statement does not return any error messages. Figure 4-13 shows setting a column as unused and then being dropped.

```
SQL>   ALTER TABLE student
  2    SET UNUSED(SocialSecurity);
Table altered.
SQL>   ALTER TABLE student
  2    DROP UNUSED COLUMNS;
Table altered.
SQL>
```

Figure 4-13 ALTER TABLE—dropping unused columns.

Dropping a Constraint

As you already know, you can view constraint information from the USER_CONSTRAINTS table or the USER_CONS_COLUMNS table. A dropped constraint is no longer enforced by Oracle, and it does not show up in the list of USER_CONSTRAINTS or USER_CONS_COLUMNS. The general syntax is

```
ALTER TABLE tablename
    DROP PRIMARY KEY|UNIQUE (columnname) |
    CONSTRAINT constraintname [CASCADE];
```

For example,

```
ALTER TABLE major
DROP PRIMARY KEY CASCADE;
```

This statement drops the primary key constraint from the MAJOR table. The CASCADE clause drops the dependent foreign key constraints, if any. You can drop a constraint by using its name, which is why it is important to name all constraints with a standard naming convention. For example,

```
ALTER TABLE employee
    DROP CONSTRAINT employee_deptid_fk;
```

Enabling/Disabling Constraints

You may enable or disable constraints as needed. A newly created constraint is enabled automatically. A constraint verifies table data as they are added or updated. This verification slows down the process, so you may want to disable a constraint if you are going to add or update large volume of data. When you reenable the constraint, Oracle checks the validity of the data and for any violations.

You may disable multiple constraints with one ALTER TABLE statement, but you may only enable one constraint at a time. The general syntax for enabling or disabling constraint is

```
ALTER TABLE tablename
ENABLE | DISABLE CONSTRAINT constraintname;
```

You may enable or disable a primary key constraint with the following syntax that does not use constraint name:

```
ALTER TABLE tablename ENABLE | DISABLE  PRIMARY KEY;
```

There is no CASCADE clause with ENABLE. The DISABLE and ENABLE clauses can also be used in a CREATE TABLE statement.

Renaming a Column (Oracle9i Version 9.2 Onward)

You can rename a column with the following statement:

```
ALTER TABLE tablename RENAME COLUMN oldname TO newname;
```

Renaming a Constraint (Oracle9i Version 9.2 Onward)

You can rename a constraint with the following statement:

```
ALTER TABLE tablename RENAME CONSTRAINT oldname TO newname;
```

Modifying Storage of a Table

You can change storage attributes of a table, such as NEXT, PCTFREE, and so, with the following statement:

```
ALTER TABLE tablename STORAGE (NEXT nK);
```

DROPPING A TABLE

When a table is not needed in the database, it can be dropped. Sometimes, the existing table structure has so many flaws it is advisable to drop it and recreate it. When a table is dropped, all data and the table structure are permanently deleted. The DROP operation cannot be reversed, and Oracle does not ask "Are You Sure?" You can drop a table only if you are the owner of the table or have the rights to do so. Many other objects based on the dropped table are affected. All associated indexes are removed. The table's views and synonyms become invalid. The general syntax is

```
DROP TABLE tablename [CASCADE CONSTRAINTS];
```

For example,

```
DROP TABLE sample;
```

Oracle displays a "Table dropped" message when a table is successfully dropped. If you add the optional CASCADE CONSTRAINTS clause, it removes foreign key references to the table as well.

RENAMING A TABLE

You can rename a table provided you are the owner of the table. The general syntax is

```
RENAME oldtablename TO newtablename;
```

For example,

```
RENAME dept TO department;
```

Oracle will display a "Table renamed" message when this statement is executed. (We will not change the DEPT table's name and will still refer to it by its original name later in this textbook.) The RENAME statement can be used to change name of other Oracle objects, such as a view, synonym, or sequence, which we will cover in a later chapter.

TRUNCATING A TABLE

Truncating a table is removing all records/rows from the table. The structure of the table, however, stays intact. You must be the owner of the table with the DELETE TABLE privilege to truncate a table. The SQL language has a DELETE statement that can be used to remove one or more (or all) rows from a table, and it is reversible as long as it is not committed. The TRUNCATE statement, on the other hand, is not reversible. Truncation releases storage space occupied by the table, but deletion does not. The syntax is

```
TRUNCATE TABLE tablename;
```

For example,

```
TRUNCATE TABLE employee;
```

Oracle displays a "Table truncated" message on this statement's execution. The EMPLOYEE table is an integral part of the N2 Corporation's database, and you do not want to truncate it unless you would like to enter all the employees' data again!

The truncate operation releases all table storage except for the initially allocated extent. You can "keep" all storage used by table with the REUSE STORAGE clause. For example,

```
TRUNCATE TABLE tablename REUSE STORAGE;
```

ORACLE'S VARIOUS TABLE TYPES

Oracle9i uses various types of tables—permanent tables, temporary tables, index-organized tables, and external tables. Permanent tables are used for storing data. Temporary tables are used during a session or a transaction. Temporary tables are like permanent tables, but they are not stored permanently. They store data during a session or a transaction. Temporary tables are created with the CREATE GLOBAL TEMPORARY TABLE statement. Index-organized tables are used for tables with primary key values that are looked up frequently. Index-organized tables are created with a CREATE TABLE *tablename* ... ORGANIZATION INDEX statement. External tables are stored "outside" the database with CREATE TABLE *tablename* ... ORGANIZATION EXTERNAL statement. These tables are based on flat files, such as comma-delimited, double quotes–delimited or fixed-length files, whose directory path is made known to Oracle with a CREATE DIRECTORY statement. In most cases, the end user works with permanent data tables only.

SPOOLING

Spooling is a very handy feature. During a session, a user can redirect all statements, queries, commands, and results to a file for later review or printout. The spooling method creates a text file of all actions and their results. Everything you see on your screen is redirected to the file, which is saved with an *.lst* extension by default.

To start spooling, go to the File menu in the SQL*Plus window. Then, click on Spool and Spool File in subsequent menus (see Fig. 4-14). You will be prompted to enter a file name, which will be created with an *.lst* extension.

To stop spooling at any point, use the same menu to click on Spool Off (see Fig. 4-14). When spooling is turned off, the file is saved to the disk and closed. The spooled file can be opened in any text editor, such as Notepad, for viewing or printing. In the classroom environment, I ask my students to spool all their work, which includes required queries and their results. The students can submit their disk or the printed hard copy.

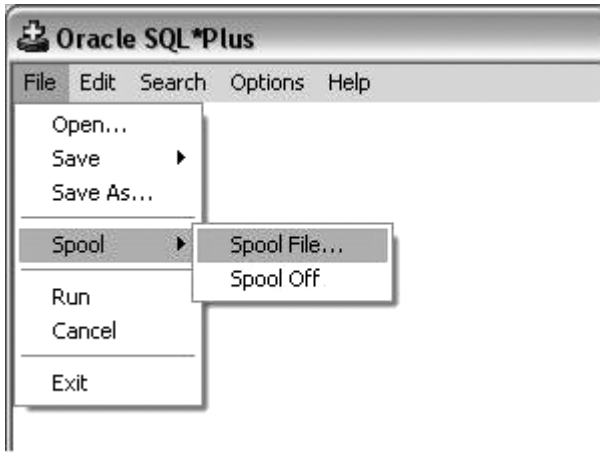


Figure 4-14 Spool menu.

You may start and stop spooling from the SQL> prompt with the

```
SQL> SPOOL filename
```

and

```
SQL> SPOOL OFF
```

commands, respectively.

ERROR CODES

If Oracle Error Help is installed on your system, you will be able to get to it by clicking on **START → Oracle → OraHome92.** Once the error help screen is displayed, click on the Index tab. Then, type the error code received from Oracle in the space provided. When you are done typing, click on the Display button to get an explanation of the error. The explanation of the error code is straightforward. The help function shows the cause of the error and gives hints for corrective actions.

You may use online help from Oracle's Web site by using the following URL:

```
http://otn.oracle.com/pls/db92/db92.error_search
```

To use the online help with error codes, follow three steps shown in Figure 4-15. In step 1, type the error code in the text box (see Fig. 4-15a). In step 2, select a result from Oracle's search results (see Fig. 4-15b). In step 3, view the cause of the error and the action required to rectify it (see Fig. 4-15c).

This online help utility requires you to sign up with The Oracle Technology Network (OTN). The free membership to OTN has many benefits, including free downloads of Oracle software products.

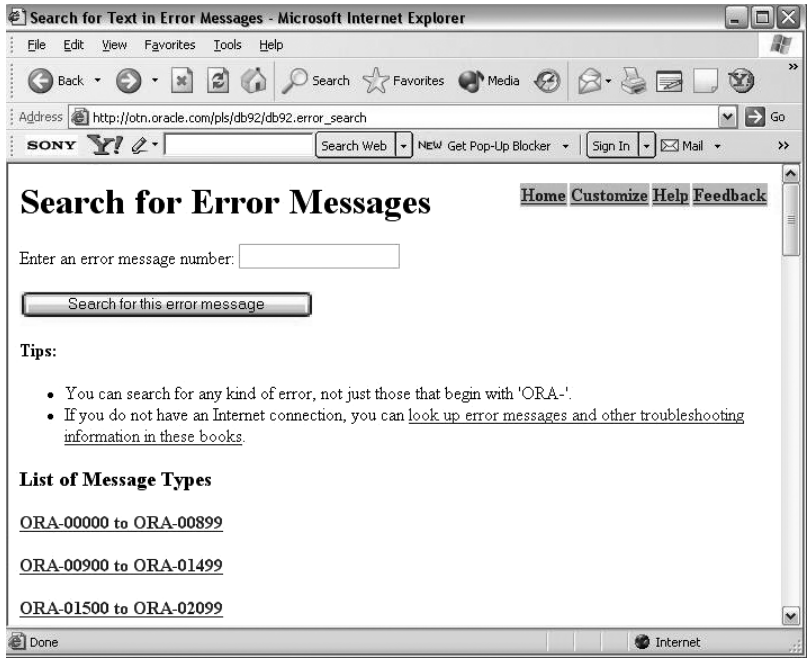


Figure 4-15a Online help—step 1.

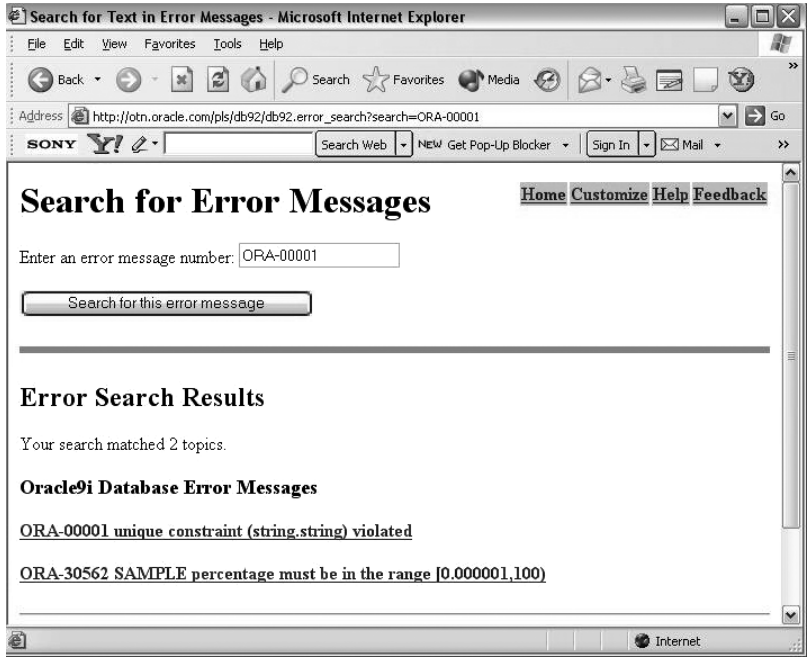


Figure 4-15b Online help—step 2.

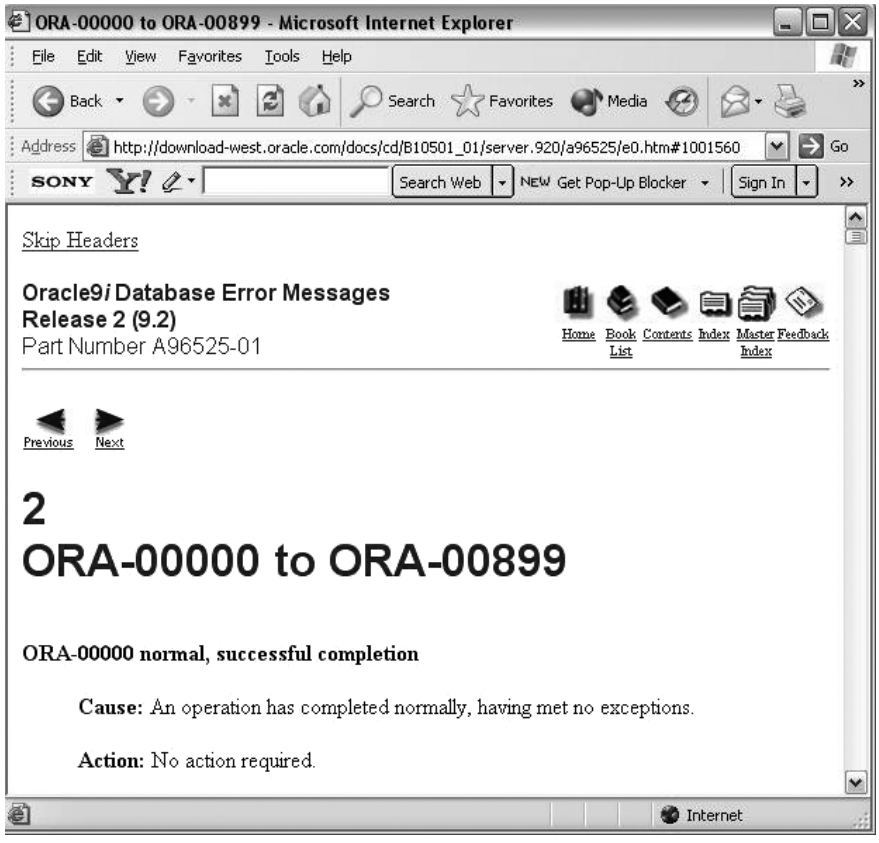


Figure 4-15c Online help—step 3.

In this chapter, you learned all the Data Definition Language (DDL) statements, which enable you to create and modify a table's structure. In the next chapter, you will learn about Data Manipulation Language (DML) statements to populate tables with the INSERT statement, to modify data using the UPDATE statement, and to remove data using the DELETE statement. We will also learn to retrieve a table's underlying data with the SELECT statement and its various clauses.

IN A NUTSHELL . . .

- Oracle database tables are stored under a user's account in an allocated tablespace on a server's disk.
- Oracle object names can be up to 30 characters long and can use letters, numbers, and the $, #, and _ characters only. All names must start with a letter.
- Each column in a table is assigned a data type to specify the type of data to be stored in it. Basic data types are CHAR (fixed-length character data),

VARCHAR2 (variable-length character data), NUMBER (integer, fixed-point decimal, and floating-point decimal values), and DATE (date and time values). The data-type column also includes the size.

- Additional data types include LONG, NCHAR, CLOB, BLOB, NCLOB, BFILE, and RAW.
- Constraints enforce rules for tables. Two types of constraints are integrity constraint (primary key and foreign key) and value constraint (check, not null, and unique). The constraints are named either by the user or by Oracle using a standard convention. A constraint is defined at the column or table level using slightly different syntax.
- A DDL statement, CREATE TABLE, is used for table creation. The creation of a table includes column names, data types, sizes of the columns, and constraint definitions.
- Oracle provides the user with SQL statements and SQL*Plus commands to view the user's tables, table structure, and constraint information.
- Oracle also provides the user with Data Dictionary tables and views for information about user account, tables, tablespaces, constraints, and objects.
- The ALTER TABLE statement is used to modify an existing table's structure. The modifications may include adding a new column, modifying an existing column, adding a constraint, or removing a constraint. Oracle8i and Oracle9i do allow a user to drop a column from a table. In Oracle9i, a column or constraint can also be renamed. There are restrictions imposed for certain modifications.
- A table can be dropped from the database or renamed. A table can be truncated to remove all its rows/records.
- Oracle error messages are displayed with error codes. A user can get more information about the causes of an error and the action necessary to correct it by using Oracle9i error messages from the Microsoft Windows environment or from Oracle's online help.
- The spooling method is used to spool all queries, statements, and commands along with their results to a text file.

EXERCISE QUESTIONS

True/False:

1. In Oracle9i, a table name cannot be one character long.
2. If a data value entered in a VARCHAR2 type field is longer than the actual size, the value is truncated.
3. The NUMBER data type can be used for integer, fixed-point decimal, and floating-point decimal values.

4. Two tables may have constraints with same constraint name under a user's database.
5. A foreign key must reference a primary key in another table, and both keys must have same column name.
6. If you try to enter value "Database" in a CHAR(4) column, only "Data" will be stored in it.
7. The value "Basketball" will be stored with five trailing spaces in a VARCHAR2(15) column.
8. A composite primary key can be defined at table level only.
9. A column with UNIQUE constraint may not contain a null value.
10. The NOT NULL constraint is defined at the table level only.
11. A CHECK constraint cannot be written to check for null values.
12. The value 9999.99 is the largest possible value for a NUMBER(6,2) column.

Find the Valid/Invalid Table/Column Names:

1. CRS-SECTION.
2. SALARY_LEVEL.
3. Employee'sId.
4. Employee Id.
5. $SALARY.
6. Proj2000.
7. Qualification_Code_For_Employees.

Write the Appropriate Column Name, Data Type, and Size for the Following Columns:

1. Student's date of birth.
2. Social Security number (without dashes).
3. Telephone number (with area code).
4. Employee's gender.
5. Employee's picture in a file.
6. Link to a Word document.
7. Customer's last name.

Write the Constraint Definitions for the Following Constraints (Use Case-Study Tables in Chapter 3):

1. Primary key in the DEPT table.
2. Foreign key DeptId in the EMPLOYEE table.
3. CHECK constraint for QualId in the EMPLOYEE table.
4. NOT NULL constraint for the MajorDesc column in the MAJOR table.
5. UNIQUE constraint for DeptName in the DEPT table.

Write answers for the following:

1. What is the use of data types? Name four basic data types, and state their use.
2. What are two types of constraints? Give two examples of each.
3. How are the constraints named?
4. Does Oracle allow a composite key? If so, how is it defined?

5. Can you change a column's name in an existing table in Oracle9i? Can you delete a column from a table?
6. Is it possible to add any type of constraint to an existing table? Are there any restrictions?
7. What are the differences between SQL and SQL*Plus?
8. How will you drop a table whose primary key is referenced by a foreign key in another table? Give two possible ways to accomplish the task.
9. How will you make sure that the value used in GENDER CHAR(1) column is either "M" or "F" only?
10. Name any three Oracle Data Dictionary tables, and give their use.
11. Can you reference a table that does not exist? Can you reference a table whose primary key is not defined? Can you reference part of a composite primary key?
12. What is the use of STORAGE clause with INITIAL and NEXT attributes?
13. How do you release all storage space with TRUNCATE statement?

LAB ACTIVITY

1. **a.** Use SQL statements to create STUDENT, FACULTY, COURSE, CRSSECTION, REGISTRATION, ROOM, TERM, LOCATION, MAJOR, and DEPARTMENT tables in the IU College database tables as given in Chapter 3. Use SQL*Plus commands or Notepad to debug your statements' errors, if any.
   - Define a primary key constraint for each table. (Do not specify foreign keys yet.)
   - Define NOT NULL, DEFAULT, UNIQUE, and CHECK constraints wherever appropriate.

   Before running your statements, start spooling to a file named CH4LAB1A.LST. When all tables are created, stop spooling, and print the spooled file.

   b. Now, add the required foreign key constraints for each table. Do not add any records yet. Spool your statements and results to the CH4LAB1B.LST file, and print it.

   c. Spool to the CH4LAB1C.LST file, and print all table names from your account, each table's structures, and constraint information for each table.
2. Use SQL statements to create all six tables from the N2 Corporation database in Chapter 3. If you have already created a DEPT table in Chapter 3's Lab Activity, you will skip it. Define the PRIMARY KEY, FOREIGN KEY, NOT NULL, DEFAULT, CHECK, and UNIQUE constraints in the CREATE TABLE statement. If not possible, use the ALTER TABLE statement to add a constraint. (*Remember:* The FOREIGN KEY constraint requires existence of the referenced table.) Spool your statements and results to the CH4LAB2.LST file, and print each table's structure and constraints as well.

5

Working with Tables: Data Management and Retrieval

IN THIS CHAPTER . . .

- You will learn how to populate tables using Data Manipulation Language (DML) statements.
- You will learn to change existing data and to remove unwanted rows/records.
- Data retrieval queries on single tables are shown.
- Various clauses are used with data retrieval queries for filtering and sorting of data.
- CASE structure is introduced.

DATA MANIPULATION LANGUAGE (DML)

SQL's Data Manipulation Language (DML) consists of three statements—INSERT, UPDATE, and DELETE. Data retrieval language, also known as a subset of DML, consists of the SELECT statement and its clauses. Some authors consider all four as DML statements. A new row is added to a table with the INSERT statement. Data in existing rows are changed with the UPDATE statement. The DELETE statement removes rows from a table. The SELECT statement retrieves data from tables, but it does not affect the data in any way. In other words, the SELECT statement does not

manipulate data; it only queries tables. The DML statements are not written permanently to the database unless they are committed. Many times, students do not exit properly from SQL*Plus, and they end up losing newly inserted rows or updated information. You can enter a COMMIT statement anytime to write DML statements to the disk. You can use ROLLBACK to undo the last set of DML statements. You will learn more about transactions in Chapter 9.

ADDING A NEW ROW/RECORD

The Data Manipulation Language (DML) statement INSERT is used to insert a new row/record into a table. A user can insert values for all columns or a selected list of columns in a record. The general syntax for the INSERT statement is

```
INSERT INTO tablename [(column1, column2, column3, . . . )]
VALUES (value1, value2, value3, . . . );
```

The column names are optional. If column names are omitted from the INSERT statement, you must enter a value for each column. If you know the correct order of column names, you can enter values in the same order following the VALUES keyword. (Use the SQL*Plus command DESCRIBE to display the table's structure to make sure.) If you insert values in the incorrect order and a numeric value is entered for a character - (CHAR)-type column, Oracle will not accept the new row and will generate an error message. If your statement is accepted, a "1 row created" message is displayed on the screen.

If you do enter column names, they do not have to be in the same order as they were defined in the table's structure at the time of creation. Once you enter column names, however, their respective values must be in the same order as the column names. For example, let us add a new record to the STUDENT table in the Indo–US (IU) College database:

```
INSERT INTO student (StudentId, Last, First, Street, City, State, Zip, StartTerm, BirthDate,
FacultyId, MajorId, Phone)
VALUES ('00100', 'Diaz', 'Jose', '1 Ford Avenue #7', 'Hill', 'NJ', '08863', 'WN03', '12-FEB-80',
123, 100, '9735551111');
```

When entering values, numeric data is not enclosed within quotes. The CHAR- and DATE-type values are enclosed within single quotes. How do you enter a character value that contains a single-quote character? For example, 'Daddy's Pizza Parlor' will result in an error. You must type two single quotes to enter a single-quote character. The solution is 'Daddy''s Pizza Parlor'. The first quotation mark acts as an escape character for the second one.

The default format to enter the DATE value is DD-MON-YY. In Oracle8i, if a two-digit year has a value greater than or equal to 50 (e.g., 60), it is stored as occurring in the twentieth century (e.g., 1960). If a two-digit year has a value less than 50 (e.g., 10), it is stored as occurring in the twenty-first century (e.g., 2010). The birth date of

15-APR-40 will be stored with the year as 2040. The student's calculated age will return a negative number! In Oracle9i, the format DD-MON-YY as well as DD-MON-YYYY are default formats. You are strongly advised to use a four-digit year. If you want to enter a date in any other format, the TO_DATE function is used for converting a character value to the date equivalent. For example,

```
TO_DATE('02/12/1980', 'MM/DD/YYYY')
TO_DATE('FEB 12, 1980', 'MON DD, YYYY')
```

A DATE-type column can store date as well as time values. If only the date value is entered in a DATE-type column, the time value is set to the midnight (12:00 A.M.). If only the time value is entered into a DATE-type column, the date is set to first of the current month. For example, a time value is entered in the HireDate column of the EMPLOYEE table with

```
TO_DATE('01:15 P.M.', 'HH:MI P.M.')
```

Then, the information is retrieved with the following format (try it after you learn the SELECT statement later in this chapter):

```
TO_CHAR(HireDate, 'DD-MM-YYYY HH:MI:SS P.M.')
```

The result will show the date as the first of the month in which time was entered along with the entered time.

Now, let us enter a new row into DEPT table in the NamanNavan (N2) Corporation's database without using the column names:

```
INSERT INTO dept
VALUES(10, 'Finance', 'Charlotte', 123);
```

The DEPT table contains four columns, and the values in the previous statement are in the correct order. While inserting values, you must remember that the foreign key columns in a table must either have a null value or must already exist as a primary key value in the table referenced by the foreign key.

For example, in the STUDENT table's INSERT statement, the value for FacultyId and MajorId columns are cross-referenced by Oracle in the FACULTY and MAJOR tables, respectively. If you have not populated those two parent tables, your new record in the STUDENT table will not be accepted. You must populate tables without foreign keys first; in other words, the parent tables must be populated before their child tables.

Rounding by INSERT

If you insert value 543.876 in a NUMBER(6,2) column, the precision is 4, and the scale is 2. The resulting value will be 543.88, rounded to two decimal places, or a scale of 2. The rounded value will be entered into the column.

Entering Null Values

Null values are allowed in non–primary key columns that do not have a NOT NULL constraint. Check the 'Null?' display from the DESCRIBE command before inserting a null value.

There are two methods for inserting a NULL value in a column:

1. ***Implicit method:*** In the implicit method, the column's name is omitted from the column list in an INSERT statement. For example,

```
INSERT INTO dept(DeptId, DeptName)
VALUES(50, 'Production');
```

   In this example, the Location and EmployeeId columns are not included. The new record will be inserted into the table with no values for those two columns. It is allowed only if the NOT NULL constraint is not used for them.

2. ***Explicit method:*** In the explicit method, the null value is used as a value for a numeric column, and an empty string ('') is used for date or character columns. For example,

```
INSERT INTO dept(DeptId, DeptName, Location, EmployeeId)
VALUES(60, 'Personnel', 'Chicago', NULL);
```

   You will insert null in EmployeeId if you do not know the manager's EmployeeId for the newly created Personnel Department in Chicago.

Often, you do not know the value of a column and decide to use a null value for it. If your table has records with null values, you have to update those records once the actual values are known. That is additional data entry. One way to avoid null values is by using a DEFAULT value on columns.

Entering Default Values

With Oracle9i, the INSERT statement has added syntax that lets you insert default values with the key word DEFAULT in place of a value for a column. If a default value is assigned to the column during the table's creation, that default value is inserted into the column. If no default value is assigned to the column, the key word DEFAULT will result into a null value for the column. Make sure there is no NOT NULL constraint on that column; otherwise, your new row will not be inserted.

Substitution Variables

Inserting rows into a table is a very tedious task. In real-life tables, we are talking about thousands of rows per table! There are screen designers, form creators, and so on. An SQL statement does not have those fancy boxes or buttons. The SQL language

does have substitution variables, which enable you to create an interactive SQL script. When you execute the script, Oracle prompts you to enter a value for the substitution variable. The ampersand (&) character is used before the substitution variable in the query. The substitution variables for CHAR- and DATE-type columns are enclosed within a pair of single quotation marks. Figure 5-1 shows the use of substitution variables and the interactive prompts displayed by Oracle.

```
SQL> INSERT INTO dept(DeptId, DeptName, Location, EmployeeId)
  2  VALUES(&dept_id, '&dept_name', '&location', &emp_id);
Enter value for dept_id: 70
Enter value for dept_name: Testing
Enter value for location: Miami
Enter value for emp_id: NULL
old   2:  VALUES(&dept_id, '&dept_name', '&location', &emp_id)
new   2:  VALUES(70, 'Testing', 'Miami', NULL)

1 row created.

SQL>
```

Figure 5-1 Substitution variables.

Question: You just ran the SQL statement in Figure 5-1. How will you insert the next record using the same statement?

Answer: The last SQL statement is in the buffer, so you will type a slash (/) to reexecute the statement from the buffer. If you stored the statement in a file, you can execute the same file again with the RUN or @ command.

If you execute an INSERT statement that contains a value containing the & character, such as the value R&D in Figure 5-2, Oracle treats it as a substitution variable. To avoid such a situation, you can disable the substitution-variable character (&) with the following SQL*Plus command:

```
SET DEFINE OFF
```

```
SQL> INSERT INTO dept
 2    VALUES (99, 'R&D', 'Windsor', 111);
Enter value for d:
```

Figure 5-2 Value with &.

Conversely, you can turn it on with the SET DEFINE ON command. You can also change the prefix with same command. For example, if you want to change the prefix for substitution variable to !, you will use the following command:

```
SET DEFINE !
```

CUSTOMIZED PROMPTS

The substitution-variable prompts are standard. Oracle displays "Enter the value for" followed by the name of the substitution variable. The SQL*Plus command ACCEPT is used for customized prompts. The ACCEPT command does not use an ampersand in front of the variable name. ACCEPT, in fact, accepts values for substitution variables that can be used later in other statements. If an ACCEPT statement is used for a variable, the value of that variable, once entered, is remembered during the session. You might not want to use the ACCEPT statement for a variable to be used later in more than one INSERT statement.The general syntax is

ACCEPT variablename PROMPT 'prompt message'

For an example, see Figure 5-3.

```
SQL> ACCEPT dept_id PROMPT 'Please enter department number(10 to 99): '
Please enter department number(10 to 99): 80
SQL> ACCEPT dept_name PROMPT 'Please enter department name(no nulls): '
Please enter department name(no nulls): Accounting
SQL> ACCEPT location PROMPT 'Please enter location city: '
Please enter location city: Monroe
SQL> ACCEPT manager PROMPT 'Please enter EmployeeId of Manager: '
Please enter EmployeeId of Manager: NULL
SQL> INSERT INTO dept
  2  VALUES(&dept_id, '&dept_name', '&location', &manager);
old  2: VALUES(&dept_id, '&dept_name', '&location', &manager)
new  2: VALUES(80, 'Accounting', 'Monroe', NULL)

1 row created.

SQL>
```

Figure 5-3 Custom prompt with ACCEPT.

Once a variable is defined with & (substitution variable) or ACCEPT, its value is known throughout that session. You can undefine such a variable with the SQL*Plus command UNDEFINE.

UPDATING EXISTING ROWS/RECORDS

Once data are added to the tables for various entities, they may not stay the same forever. A female employee gets married and changes her last name, a student changes his or her major, a customer/vendor moves to a new location, or an employee

gets a salary increment. These are real-life possibilities. When you create tables, you should use columns that are not very high maintenance. For example, you should not use a column called AGE. The age changes every year for an individual, and it also changes on different days for almost everybody.

In SQL, the UPDATE statement is used for such modifications to data. Only one table can be updated at a time, but it is possible to change more than one column at a time. The general syntax is

```
UPDATE tablename
SET column1 = newvalue [, column2 = newvalue, . . .]
[WHERE condition(s)];
```

The condition is optional, but in most cases, you would need to use it. If the condition is not used with UPDATE, all rows will be updated. The conditions are created using column names, relational operators, and values. You already know that Oracle is case sensitive as far as the values in single quotation marks are concerned. The relational operators are shown in Figure 5-4.

Relational Operator	Meaning
=	Equal to
<> or !=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Figure 5-4 Relational operators.

Suppose the student with ID 00103 in the IU College's database switches major from BS—Computer Science to BS—Telecommunications. We will write an update statement to change the student's MajorId in the STUDENT table. Figure 5-5 first shows an unsuccessful update operation. We are trying to change MajorId to 700, which did not work! There is no such value for MajorId in the MAJOR table, which is being referenced by the foreign key in the STUDENT table. See Oracle's error code and error message, which point out the integrity constraint was violated. Then, the figure shows that the value is changed to 600, which worked! In this figure, the value is changed back to the original value of 500. The same UPDATE statement without the WHERE clause would result in updating all students' MajorId to 500.

There are other operators for writing conditions like AND, OR, BETWEEN . . . AND, IN, and LIKE. We will learn more about them later in this chapter.

```
SQL> UPDATE student
  2  SET MajorId = 700
  3  WHERE  StudentId = '00103';
UPDATE student
*
ERROR at line 1:
ORA-02291: integrity constraint (SYSTEM.STUDENT_MAJORID_FK) violated - parent
key not found

SQL> 2
  2* SET MajorId = 700
SQL> c/700/600
  2* SET MajorId = 600
SQL> 1
  1  UPDATE student
  2  SET MajorId = 600
  3* WHERE  StudentId = '00103'
SQL> /

1 row updated.

SQL> 2
  2* SET MajorId = 600
SQL> c/600/500
  2* SET MajorId = 500
SQL> /

1 row updated.

SQL>
```

Figure 5-5 Unsuccessful and successful UPDATE statements.

DELETING EXISTING ROWS/RECORDS

Deletion is another data maintenance operation. When employees leave the company or students enroll but never start college, you might want to remove their information from your database. In Oracle, the SQL statement DELETE is used for deleting unwanted rows. Its general syntax is

```
DELETE [FROM] tablename
[WHERE condition(s)];
```

The keyword FROM is optional. The WHERE clause adds a condition to the DELETE statement. Once again, the condition is optional, but it is necessary. You normally would delete only those records that meet a criterion. The DELETE statement without a condition will result in a table with no rows. A DELETE statement without a WHERE clause has same effect as a TRUNCATE statement. The only difference is that the DELETE operation can be undone with ROLLBACK statement (see Chapter 9), but the TRUNCATE operation makes the change permanent.

```
SQL> DELETE FROM dept
  2  WHERE DeptId = 70;

1 row deleted.

SQL>
```

Figure 5-6 Successful DELETE statement.

Figure 5-6 shows successful execution of a DELETE statement. If a row with department number 70 exists, it is deleted.

If you try to delete a record from a table whose primary key value is used in another table's foreign key column, Oracle will display an "Integrity constraint . . . violated - child record found" error message. The parent record that is referenced by a child record cannot be removed. See Figure 5-7 for such an unsuccessful DELETE

```
SQL> DELETE FROM dept
  2    WHERE DeptId = 20;
DELETE FROM dept
*
ERROR at line 1:
ORA-02292: integrity constraint (SYSTEM.EMPLOYEE_DEPTID_FK) violated - child
record found

SQL>
```

Figure 5-7 Unsuccessful DELETE statement.

operation. In such cases, you may drop a constraint or temporarily disable it. A dropped constraint is removed permanently, whereas a disabled constraint can be enabled later.

RETRIEVING DATA FROM A TABLE

The main purpose of the SQL language is for querying the database. You have already learned to create, alter, insert, update, and delete by using SQL statements. The most important statement or query is the SELECT query. A user retrieves data from the underlying table or tables with a SELECT query. The output can be sorted and grouped, and information can be derived with the use of mathematical expressions and built-in functions. In Chapter 1, we covered nine relational operations. Now is the time to try those operations. The general syntax is

```
SELECT columnlist
FROM tablename;
```

The columns can be listed in any order. They do not have to be in the order

```
SQL> SELECT Last, First
  2  FROM student;
LAST            FIRST
--------------- ---------------
Diaz            Jose
Tyler           Mickey
Patel           Rajesh
Rickles         Deborah
Lee             Brian
Khan            Amir

6 rows selected.

SQL>
```

Figure 5-8 Output from a SELECT query.

given by the DESCRIBE command. For example, Figure 5-8 shows output from a SELECT query. This query displays the last name and first name of all students from the STUDENT table. The Last and First columns are vertical slices of the STUDENT table, a projection operation.

As you see, the column names are displayed in uppercase. In the STUDENT table, the column Last comes before the column First, but the output displays them in the order given in the SELECT query. By default, character data is displayed with left justification and numeric data with right justification.

Select (*)

If you want to see all columns in a table, you do not have to list them all. You can use an asterisk (*) in place of the column list, and all columns will be displayed in the same order as the underlying table structure. Figure 5-9 depicts use of the character *.

```
SQL> SELECT *
  2  FROM course;

COURSE  TITLE                CREDITS  PREREQ
------  -------------------- -------  ------
EN100   Basic English              0
LA123   English Literature         3  EN100
CIS253  Database Systems           3
CIS265  Systems Analysis           3  CIS253
MA150   College Algebra            3
AC101   Accounting                 3

6 rows selected.

SQL>
```

Figure 5-9 SELECT *.

In output from a SELECT statement, left justification is used for CHAR-, VARCHAR2-, and DATE-type columns, and right justification is used for NUMBER-type columns. The default column width for displaying a NUMBER column is 9. The display width for character columns is based on column's width.

Two problems with displaying all rows and all columns are the screen's default line size and page size. After 80 columns, the row display wraps to the next line. After displaying 11 rows under column headings, column headings are repeated for more rows. You can change these values from the Options menu in the SQL*Plus environment.

The environment variables can be changed by clicking on the *Options* menu in SQL*Plus, then by selecting *Environment* from it. Soon, an Environment window will pop up, and you can select *linesize* from the list of environment variables. Then, change the value from Default to Current, and type in the new value. The default value for line size is 80. Similarly, *pagesize* and other variables can be set (see Fig. 5-10). Alternately, you can type SET commands at the SQL> prompt. For example,

```
SET LINESIZE 150
```

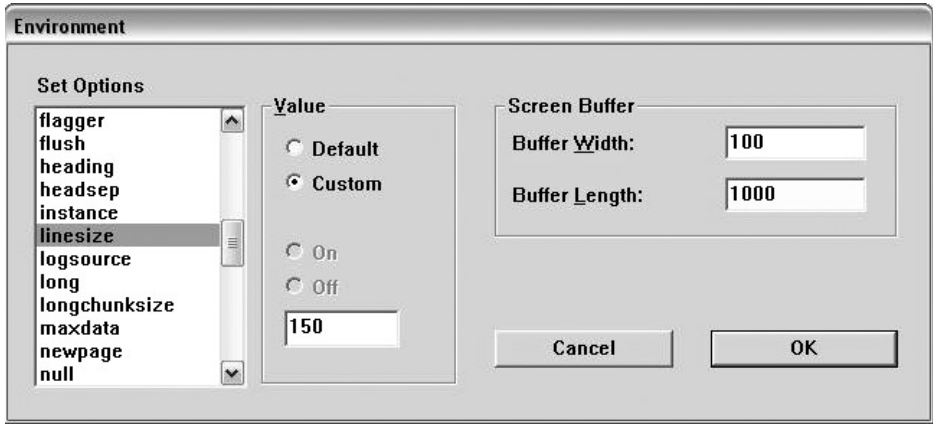


Figure 5-10 Setting environment variable line size.

Figures 5-11a and 5-11b show output from a SELECT query with a default linesize of 80 and output after changing the linesize to a higher value. You can get information about environment variables by using the SQL*Plus command SHOW ALL.

You will notice that the queries, which return less than six rows, do not get feedback from Oracle stating the number of rows returned. By default, Oracle sends a feedback message for queries returning six or more rows only. If you would like feedback for all queries irrespective of number of rows returned, use the following command in the beginning of your session:

```
SET FEEDBACK 1
```

```
SQL> SELECT * FROM student;

STUDE   LAST            FIRST                STREET                    CITY
-----   ----------      ----------------     -------------------       --------
ST  ZIP        STAR  BIRTHDATE   FACULTYID   MAJORID PHONE
--  -----      ----  ---------   ---------   --------- ------------

00100  Diaz                     Jose          1 Ford  Avenue #7         Hill
NJ   08863     WN03  12-FEB-83          123        100   9735551111

00101  Tyler                    Mickey        12 Morris Avenue          Bronx
NY   10468     SP03  18-MAR-84          555        500   7185552222

00102  Patel                    Rajesh        25 River Road #3          Edison
NJ   08837     WN03  12-DEC-85          111        400   7325553333

STUDE   LAST            FIRST                STREET                    CITY
-----   ----------      ----------------     -------------------       --------
ST  ZIP        STAR  BIRTHDATE   FACULTYID   MAJORID   PHONE
--  -----      ----  ---------   ---------   --------- ------------

00103  Rickles                  Deborah       100 Main Street           Iselin
NJ   08838     FL02       20-OCT-70          555        500   7325554444

00104  Lee                      Brian         2845 First Lane           Hope
NY   11373     WN03      28-NOV-85     345        600   2125555555

00105  Khan                     Amir          213 Broadway              Clifton
NJ    07222    WN03      07-JUL-84     222        200   2015556666

6 rows selected.

SQL>
```

Figure 5-11a Output before setting LINESIZE.

```
SQL>  SET LINESIZE 150
SQL>  SELECT * FROM student;

STUDE  LAST    FIRST   STREET            CITY     ST  ZIP    STAR  BIRTHDATE  FACULTYID  MAJORI
-----  ------  ------  ----------------  -------- --  -----  ----  ---------  ---------  ------
00100  Diaz    Jose    1 Ford Avenue #7  Hill     NJ  08863  WN03  12-FEB-83        123      10
00101  Tyler   Mickey  12 Morris Avenue  Bronx    NY  10468  SP03  18-MAR-84        555      50
00102  Patel   Rajesh  25 River Road #3  Edison   NJ  08837  WN03  12-DEC-85        111      40
00103  Rickles Deborah 100 Main Street   Iselin   NJ  08838  FL02  20-OCT-70        555      50
00104  Lee     Brian   2845 First Lane   Hope     NY  11373  WN03  28-NOV-85        345      60
00105  Khan    Amir    213 Broadway      Clifton  NJ  07222  WN03  07-JUL-84        222      20

6 rows selected.

SQL>
```

Figure 5-11b Output after setting LINESIZE.

Conversely, you can suppress feedback by using the following command:

```
SET FEEDBACK OFF
```

DISTINCT Function

The DISTINCT function is used to suppress duplicate values. The word *DISTINCT* is used right after the keyword SELECT and before the column name.

Let us see the difference in result from two SELECT queries, with and without the DISTINCT function. Figure 5-12a shows the SELECT statement without DISTINCT, which outputs 11 rows with some duplicate values. Figure 5-12b shows the SELECT statement with DISTINCT, which eliminates duplicate values and returns three unique values only.

```
SQL> SELECT Building
  2  FROM location;

BUILDIN
-------
Gandhi
Gandhi
Kennedy
Kennedy
Nehru
Nehru
Gandhi
Kennedy
Kennedy
Gandhi
Gandhi

11 rows selected.

SQL>
```

Figure 5-12a SELECT without DISTINCT.

```
SQL> SELECT DISTINCT Building
  2  FROM location;

BUILDIN
-------
Gandhi
Kennedy
Nehru

SQL>
```

Figure 5-12b SELECT with DISTINCT.

Column Alias

When a SELECT query is executed, SQL*Plus uses the column's name as the column heading. Normally, the user gives abbreviated names for columns, and they are not very descriptive. For example, in Figure 5-13, the column name Title is used for course name and PreReq for requirement.

```
SQL> SELECT CourseId AS "COURSE", Title "Course Name",
  2  PREREQ Requirement
  3  From course;

COURSE     Course Name          REQUIR
------     ---------------      ------
EN100      Basic English
LA123      English Literature   EN100
CIS253     Database Systems
CIS265     Systems Analysis     CIS253
MA150      College Algebra
AC101      Accounting

6 rows selected.

SQL>
```

Figure 5-13 Column aliases.

Column aliases are useful, because they let you change the column's heading. When a calculated value is displayed, the mathematical expression is not displayed as the column heading, but the column alias is displayed. The column alias is written right after the column name with the optional keyword AS in between. The alias heading appears in uppercase by default. If an alias includes spaces or special characters or if you want to preserve its case, you must enclose it in double quotation marks (" "). The general syntax is

SELECT columnname [AS] alias . . .

In the example in Figure 5-13, "Course Name" is an alias for Title, and Requirement is an alias for PreReq. The case is preserved for the first two aliases only, because they were enclosed within double quotes. Notice that the word AS is used for the first column but is omitted for the second and third columns. This is done for illustration purpose; you may use it in a query or omit it altogether.

COLUMN Command

In Figure 5-13, you may have noticed that the column alias REQUIREMENT appeared as REQUIR. Can you figure out the reason? The PreReq column has a data type of VARCHAR2(6), and SQL*Plus displayed only first six character of the column heading. SQL*Plus' COLUMN command allows you to specify format columns for

columns. The general syntax of the COLUMN command is

COLUMN columnname FORMAT formattype

For example,

```
COLUMN State FORMAT A5
COLUMN Salary FORMAT $999,999
```

In the examples above, the State column is given the alpha format with display width of five. In the STUDENT table, the State column has a data type of CHAR(2), and that displays only the first two characters of the column name in the output. The new format will display the entire column name as a heading and use five columns to display state values. The numeric format applied to the Salary column will display salary values in currency format (e.g., $265,000).

In Figure 5-14, the CourseId and Credits columns are displayed before using formatting columns with the COLUMN command and then after applying those columns. The SELECT statement is in the buffer, so we could execute it with a forward

```
SQL> SELECT CourseId, Credits, PreReq
  2  From course;

COURSE  CREDITS      PREREQ
------ ---------    ------
EN100           0
LA123           3    EN100
CIS253          3
CIS265          3    CIS253
MA150           3
AC101           3

6 rows selected.

SQL>  COLUMN CourseId FORMAT A8
SQL>  COLUMN Credits  FORMAT 9.99
SQL>  /

COURSEID    CREDITS  PREREQ
-------- --------- ------
EN100          .00
LA123         3.00   EN100
CIS253        3.00
CIS265        3.00   CIS253
MA150         3.00
AC101         3.00

6 rows selected.

SQL>
```

Figure 5-14 COLUMN command.

slash (/) at the SQL> prompt. Though the COLUMN command was used after initial execution of the SELECT statement, the COLUMN command, being an SQL*Plus command, did not replace it in the buffer. Notice the number of columns used in displaying CourseId and also its heading before and after use of the COLUMN command. For the Credits column, the change is in the way values are displayed after use of the COLUMN command.

Concatenation

The word *concatenation* is a common word in computer jargon, but in daily life, it is seldom used. Concatenation means joining or linking. In SQL, concatenation joins a column or a character string to another column. The result is a column that is a string or a sequence of characters. Two vertical bars or pipe symbols (||) are used as the concatenation operator. The symbol appears on your keyboard with the backslash (\) character. You will need to depress the SHIFT key to enter the character. Figure 5-15 shows the result of concatenating two columns in the EMPLOYEE table. The two columns' values are joined without any space separating them.

```
SQL> SELECT Lname || Fname
  2  FROM employee;

LNAME || FNAME
------------------------------
SmithJohn
HoustonLarry
RobertsSandi
McCallAlex
DevDerek
ShawJinku
GarnerStanley
ChenSunny
ZEESONIA

9 rows selected.

SQL>
```

Figure 5-15 Concatenation.

In Figure 5-16, we have altered the output so that the names are displayed with the last name and first name separated by a comma and a space. We need to use a character string to accomplish it.

Now, try this:

```
SELECT First || ' ' || Last || 'was born on ' || Birthdate
FROM student;
```

```
SQL> SELECT Lname  | | ', ' | | Fname | | ' makes $' | |
  2  Salary "Employee Salary Info"
  3  FROM employee;

Employee Salary Info
----------------------------------------------
Smith, John makes $265000
Houston, Larry makes $150000
Roberts, Sandi makes $75000
McCall, Alex makes $66500
Dev, Derek makes $80000
Shaw, Jinku makes $24500
Garner, Stanley makes $45000
Chen, Sunny makes $35000

8 rows selected.

SQL>
```

Figure 5-16 Concatenation with character strings and columns.

ARITHMETIC OPERATIONS

The arithmetic expressions are used to display mathematically calculated data. These expressions use columns, numeric values, and arithmetic operators (see Fig. 5-17). When arithmetic operators are used with columns in the SELECT query, the underlying data are not changed. The calculations are for output purposes only.

Operator	Use
*	Multiplication
/	Division
+	Addition
–	Subtraction

Figure 5-17 Arithmetic operators.

Order of Operation

The order of operation is as follows:

- Whatever is in parentheses is done first.
- Multiplication and division have higher precedence than addition and subtraction.
- If more than one operator of the same precedence is present, the operators are performed from left to right.

As you see in Figure 5-18, if the column alias is not used, the expression is displayed as the column heading. It is optional to leave a space on both sides of an arithmetic operator. One other peculiar thing is the total of Salary and Commission. When a Salary value is added to a null value in the Commission column, the total is a null value. To handle null values, the expression can be changed to salary + NVL (Commission, 0), where NVL is a function that replaces a NULL value with the second argument in parentheses—in this case, a zero—for arithmetic operation. We will revisit the NVL function shortly. Remember that any arithmetic operation with a null value returns a null value as result.

```
SQL> SELECT Lname, Fname, Salary+Commission
  2  FROM employee;

LNAME           FNAME           SALARY+COMMISSION
--------------- --------------- -----------------
Smith           John                       300000
Houston         Larry                      160000
Roberts         Sandi
McCall          Alex
Dev             Derek                      100000
Shaw            Jinku                       27500
Garner          Stanley                     50000
Chen            Sunny

8 rows selected.

SQL>
```

Figure 5-18 Arithmetic operation with null values.

RESTRICTING DATA WITH A WHERE CLAUSE

When we used the SELECT query in Fig. 5-8 earlier, we restricted the number of columns to only two. This was an example of a *projection* operation. Remember that the projection operation returns "vertical slices" or columns from a table! (This was covered in Chapter 1). In Figure 5-9, we basically displayed all columns. In both cases, all rows from the table were displayed. Many times, you don't want to see all the rows from a table, only those rows that meet a criteria. A WHERE clause is used with the SELECT query to restrict the rows that are picked. It is the implementation of a *selection* operation. The WHERE clause uses a simple condition or a compound condition. The rows that satisfy the supplied conditions are displayed in the output. The syntax of SELECT changes a little with an added WHERE clause. The general syntax of the WHERE clause is

```
SELECT columnlist
    FROM tablename
    [WHERE condition(s)];
```

The conditions are written using column names; relational (see Fig. 5-4), logical (see Fig. 5-19), and other comparison operators (see Fig. 5-21); literal values; mathematical expressions; and built-in functions.

You are familiar with the arithmetic operators and relational operators already. The logical operators AND and OR work with two conditions, whereas NOT works with only one condition. All three return a TRUE or a FALSE result. The truth table in Figure 5-20 shows the working of AND and OR operators.

Logical Operator	Meaning
AND	Returns TRUE only if both conditions are true.
OR	Returns TRUE if one or both conditions are true.
NOT	Returns TRUE if the condition is false.

Figure 5-19 Logical operators.

AND	OR
TRUE AND TRUE = TRUE	TRUE OR TRUE = TRUE
TRUE AND FALSE = FALSE	TRUE OR FALSE = TRUE
FALSE AND TRUE = FALSE	FALSE OR TRUE = TRUE
FALSE AND FALSE = FALSE	FALSE OR FALSE = FALSE
NULL AND TRUE = NULL	NULL OR TRUE = TRUE
NULL AND FALSE = FALSE	NULL OR FALSE = NULL
NULL AND NULL = NULL	NULL OR NULL = NULL

Figure 5-20 AND and OR.

Operator	Meaning
IS NULL	Is a null value.
BETWEEN . . . AND	Is between a range of values (both included).
IN	Match any value from a list (an alternate way to write OR).
LIKE	Match a value using wild cards.

Figure 5-21 Other comparison operators.

Other special comparison operators are given in Figure 5-21. The IS NULL operator checks for a null value. It returns TRUE for a null value and FALSE for a not null value. The BETWEEN . . . AND operator checks for a range of values using lower and upper limits. The IN operator is an alternate and shorter way of writing OR conditions. The LIKE operator is used with wild cards for pattern matching.

In this section, we will give many examples of restricted data retrieval. First, the IU College database has a few thousand students. The administration wants to identify students who started in the Winter 2003 term. We have used a few sample

records in each table for simplicity. Based on the rows entered, we will get output as given in Figure 5-22.

```
SQL> SELECT StudentId, Last, First
  2  FROM student
  3  WHERE StartTerm = 'WN03';

STUDE  LAST                 FIRST
-----  ---------------      ---------------
00100  Diaz                 Jose
00102  Patel                Rajesh
00104  Lee                  Brian
00105  Khan                 Amir

SQL>
```

Figure 5-22 Data retrieval with the WHERE clause.

The sample STUDENT table has six rows, but only four StartTerm values match WN03. When character values are tested in conditions, Oracle is case sensitive. Use of the value wn03 would have returned no rows because of the lowercase letters. The character and date values are enclosed within single quotation marks.

In Figure 5-23, no rows are selected from the query, even though one department in the N2 Corporation database is located there. The problem here is the all-uppercase value in the query. The actual data are in proper case, or in *initcap*. We will learn character functions soon to avoid these types of problems.

```
SQL> SELECT *
  2  FROM dept
  3  WHERE Location = 'MONROE';
no rows selected
SQL>
```

Figure 5-23 Case sensitivity.

Let us try another relational operator in the condition. The president of the N2 Corporation wants to find out the name and department number of all employees who make $50,000 or more in salary only. The president will be surprised if he does not see his name in the list in Figure 5-24!

Let us say it is time to schedule courses for the next term. The Accounting department wants to schedule a course in a classroom that accommodates 40 to 45 students. We can perform this query by using two conditions with the AND operator or by using the BETWEEN . . . AND operator. Figure 5-25 uses a BETWEEN . . . AND operator with 40 as the lower limit and 45 as the upper limit. The same condition can be written as a compound condition with the logical operator AND as given in Fig. 5-26.

```
SQL> SELECT Lname, Fname, Salary, DeptId
  2  FROM employee
  3  WHERE Salary >= 50000;
LNAME           FNAME           SALARY     DEPTID
--------------- --------------- ---------- ----------
Smith           John                265000         10
Houston         Larry               150000         40
Roberts         Sandi                75000         10
McCall          Alex                 66500         20
Dev             Derek                80000         20

5 rows selected.

SQL>
```

Figure 5-24 Relational operator >=.

```
SQL> COL RoomNo     FORMAT A6
SQL> COL RoomType   FORMAT A8
SQL> SELECT Building, RoomNo, Capacity, RoomType
  2  FROM location
  3  WHERE Capacity BETWEEN 40 AND 45;

BUILDIN ROOMNO     CAPACITY ROOMTYPE
------- ------ ---------- --------
Nehru   309            45 C
Kennedy 206            40 L

SQL>
```

Figure 5-25 BETWEEN . . . AND operator.

```
SQL> SELECT Building, RoomNo, Capacity, RoomType
  2  FROM location
  3  WHERE Capacity >= 40 AND Capacity <= 45;

BUILDIN ROOMNO     CAPACITY ROOMTYPE
------- ------ ---------- --------
Nehru   309            45 C
Kennedy 206            40 L

SQL>
```

Figure 5-26 Compound condition with the AND operator.

When the relational and logical operators are used together, the order of precedence is as follows if all operators exist:

- Whatever is in parentheses is performed first.
- Relational operators are performed second.

- The NOT operator is performed third.
- The AND operator is performed fourth.
- The OR operator is performed last.

We can write compound conditions with multiple operators and columns. Say we are looking for employees with a salary in the range of $50,001 to $100,000 and belonging to department 10. We can further restrict data by adding another condition to check for the Depart Id (see Fig. 5-27).

```
SQL> SELECT Lname, Fname, Salary, DeptId
  2  FROM employee
  3  WHERE Salary BETWEEN 50001 AND 100000
  4  AND DeptId = 10;

LNAME           FNAME           SALARY     DEPTID
--------------- --------------- ---------- ----------
Roberts         Sandi                88275         10

SQL>
```

Figure 5-27 Compound condition using two columns.

The BETWEEN . . . AND operator can also be applied to character values to find names starting with a range of characters. With character values, the operator checks for the first character of column values (see Fig. 5-28).

```
SQL> SELECT Lname, Fname
  2  FROM employee
  3  WHERE Lname BETWEEN 'M' AND 'Z';

LNAME           FNAME
--------------- ---------------
Smith           John
Roberts         Sandi
McCall          Alex
Shaw            Jinku

SQL>
```

Figure 5-28 BETWEEN . . . AND operator with character values.

The N2 Corporation is conducting a study regarding their employees' qualifications. The company wants to identify all employees with bachelors, masters, and doctorate degrees. The corresponding qualification codes are 3, 2, and 1. The most appropriate operator is OR, because the employee has to have one of the three codes (see Fig. 5-29). The same result can be obtained by using the IN operator in place of three conditions and two OR conjunctions. For example,

```
WHERE QualId IN(3, 2, 1);
```

```
SQL> SELECT Lname, Fname, Salary, DeptId, QualId
  2  FROM employee
  3  WHERE QualId=1 OR QualId=2 OR QualId=3;

LNAME           FNAME           SALARY     DEPTID     QUALID
--------------- --------------- ---------- ---------- ----------
Smith           John            265000         10          1
Houston         Larry           150000         40          2
Roberts         Sandi            75000         10          2
Dev             Derek            80000         20          1
Chen            Sunny            35000         10          3

5 rows selected.

SQL>
```

Figure 5-29 OR operator.

Similarly, if we want to find out the names of all students in the IU College database who are from New York and New Jersey, we can use the OR operator. What if we are looking for students from New York, New Jersey, Connecticut, Delaware, and Pennsylvania? With OR, we will need five conditions, so in this case, the IN operator is preferable. Figure 5-30 shows a different example using the of IN operator to find faculty members who belong to department 1, 2, or 3.

```
SQL> SELECT Name, Phone, DeptId
  2  FROM faculty
  3  WHERE DeptId IN(1,2, 3);

NAME            PHO        DEPTID
--------------- --- ----------
Jones           525             1
Williams        533             2
Mobley          529             1
Vajpayee        577             2
Sen             579             3
Collins         599             3

SQL>
```

Figure 5-30 IN operator.

Suppose we change the condition of Figure 5-30 to WHERE DeptId NOT IN (1, 2, 3). What will be the outcome? The NOT operator, when used with other operators, negates the result.

In the EMPLOYEE table, there is information about the employee's immediate supervisor to whom he or she reports. Is there any employee who does not have a supervisor? If so, then either the information is missing or the employee has the highest position in the company. Let us search for such employees in Figure 5-31.

```
SQL> SELECT Lname, Fname, Supervisor
  2    FROM employee
  3    WHERE Supervisor = NULL;

no rows selected

SQL> SELECT Lname, Fname, Supervisor
  2    FROM employee
  3    WHERE Supervisor = '';

no rows selected

SQL> SELECT Lname, Fname, Supervisor
  2    FROM employee
  3    WHERE Supervisor IS NULL;

LNAME           FNAME           SUPERVISOR
--------------- --------------- ----------
Smith           John

SQL>
```

Figure 5-31 IS NULL operator.

Fortunately, only one employee is without supervisor information, and this employee happens to hold the president's position. If more such records were found, data-entry personnel would have to UPDATE that information. Have another look at the figure. The first two times, no rows are selected with conditions Supervisor = NULL and Supervisor = ''. The ony way to check for null values is with the IS NULL operator.

Similarly, we can check for rows with no null value in a column. In other words, we would like to see rows with values in a column. For example, the EMPLOYEE table in the N2 Corporation database has rows with commission and also has rows with a null value for the commission. John Smith is the only employee in department 10 with a NOT NULL value in the commission column (see Fig. 5-32).

```
SQL> SELECT Lname, Fname, Salary, Commission
  2    FROM employee
  3    WHERE Commission IS NOT NULL
  4    AND DeptId = 10;

LNAME           FNAME           SALARY   COMMISSION
--------------- --------------- ---------- ----------
Smith           John                265000      35000

SQL>
```

Figure 5-32 IS NOT NULL operator.

Wild Cards

You have already seen examples of a search for a string value. There are times, however, when you do not know the exact string value. In these cases, you can select rows that match a pattern of characters. Such a search is known as a *wild-card search*. There are two wild cards for a pattern search. Figure 5-33 explains the use of these wild cards.

Wild Card	Use
%	Represents zero or more characters.
_(Underscore)	Represents any one character.

Figure 5-33 Wild cards.

A search with the wild cards requires you to use the LIKE operator. In the college's database, we want to see the information about faculty members whose names start with the letter *C*. All faculty names start with an uppercase letter. Oracle is case sensitive, so a *c* in place of a *C* will not return any faculty names. Figure 5-34 has a query that searches for such faculty names.

```
SQL> SELECT Name, Phone
  2    FROM faculty
  3    WHERE Name LIKE 'C%';

NAME                PHO
--------------- ---
Chang               587
Collins             599

2 rows selected.

SQL>
```

Figure 5-34 Wild-card %.

Similarly, if we want to find out the names of employees hired during the 1960s, we can look for hire dates that fall between January 1, 1960 and December 31, 1969. Would the BETWEEN . . . AND operator be the best choice for it? Not really! We can use a combination of both wild cards to achieve the same result. We have two employees with hire dates in the 1960s. Figure 5-35 uses '%6_', which means that the date starts with any characters, the second-to-last character is a 6, and the last character could be anything. In the two case-study databases, none of the tables uses a value that actually uses the character % or_.

```
SQL> SELECT Lname, Fname, HireDate
  2  FROM employee
  3  WHERE HireDate LIKE '%6_';

LNAME           FNAME           HIREDATE
--------------- --------------- ---------
Smith           John            15-APR-60
Houston         Larry           19-MAY-67

2 rows selected.

SQL>
```

Figure 5-35 Wild-cards % and _.

Question: How do you look for a value that has a wild-card character embedded in it?

Answer: You use an escape character. SQL does not provide any particular character as an escape character, but you can specify one for the query. The WHERE clause will look like this:

```
WHERE column LIKE '%/_%' ESCAPE '/';
```

The first % means the column value starts with any characters in the beginning. The second % means the value ends with any characters. The characters /_ mean there is a _ character in the value. The character / is used as the escape character, which changes the meaning of _ from a wild card to the underscore character. You may use any character as an escape character, which is defined following the key word ESCAPE.

SORTING

The order of rows in a table is arbitrary. You can insert rows in any order, and you do not have control over where rows will be inserted. When you type a SELECT query, the order of rows is not defined. You may want to see rows in a specific order, however, based on a column or columns. It is not necessary to display a sort column in the SELECT clause. For example, you may want to see employees in alphabetical order by their name, employees with the highest-paid employee first and the lowest-paid employee last, or students by their major in alphabetical order.

The ORDER BY clause is used with the SELECT query to sort rows in a table. The rows can be sorted in ascending or descending order. The rows can also be sorted based on one or more columns. The expanded syntax of SELECT given here uses an ORDER BY clause, which is always used last in the statement. The general syntax is

```
SELECT columnlist
    FROM tablename
    [WHERE condition(s)]
    [ORDER BY column|expression [ASC|DESC]];
```

In the syntax, ASC stands for ascending order. The default order is ascending, so there is no need to type ASC for ascending order. The keyword DESC stands for descending or reverse order.

In an ascending sort, numeric values are displayed from the smallest to the largest, character values are displayed in alphabetical order, and date values are displayed with the earliest date first (see Fig. 5-36). The null values are displayed last, in ascending order. In descending order, the effect is reversed for all type of values; the null values are displayed first in the descending order.

Type of Value	Ascending Sort Order
Numeric	Lowest to highest value
Character	Alphabetical order
Date	Earliest to latest date

Figure 5-36 Ascending sort order.

You can display null values first in a sort in ascending order by using the NULLS FIRST option. Try the following two statements, and see the difference in output:

```
SELECT CourseId, PreReq FROM course ORDER BY PreReq;
SELECT CourseId, PreReq FROM course ORDER BY PreReq NULLS FIRST;
```

In the next four examples, we will perform an ascending sort by one column (see Fig. 5-37), a descending sort by one column (see Fig. 5-38), a sort by column alias (see Fig. 5-39), and a sort by multiple columns (see Fig. 5-40). First, let us display all students in alphabetical order. The ORDER BY clause will use the Last

```
SQL> SELECT Last, First
  2  FROM student
  3  ORDER BY Last;

LAST                 FIRST
-------------------- --------------------
Diaz                 Jose
Khan                 Amir
Lee                  Brian
Patel                Rajesh
Rickles              Deborah
Tyler                Mickey

6 rows selected.

SQL>
```

Figure 5-37 Single-column sort.

```
SQL> SELECT Lname, Fname, Salary
  2    FROM employee
  3    WHERE DeptId   =   30
  4    ORDER BY Salary Desc;

LNAME                FNAME                SALARY
---------------      ---------------      ----------
Garner               Stanley                45000
Shaw                 Jinku                  24500

SQL>
```

Figure 5-38 Descending order sort.

```
SQL> SELECT Lname || ', ' || Fname AS fullname,
  2    Salary / 12 AS monthlysalary
  3    FROM employee
  4    WHERE DeptId   =   10
  5    ORDER BY monthlysalary
  6    /

FULLNAME                              MONTHLYSALARY
-------------------------------       -------------
Chen, Sunny                           2916.66667
Roberts, Sandi                            6250
Smith, John                           22083.3333

3 rows selected.

SQL>
```

Figure 5-39 Sort by an alias.

```
SQL> SELECT Lname, Fname, Salary, DeptId
  2  FROM employee
  3  ORDER BY DeptId, Salary Desc;

LNAME                FNAME                SALARY          DEPTID
---------------      ---------------      ----------      ----------
Smith                John                   265000              10
Roberts              Sandi                   75000              10
Chen                 Sunny                   35000              10
Dev                  Derek                   80000              20
McCall               Alex                    66500              20
Garner               Stanley                 45000              30
Shaw                 Jinku                   24500              30
Houston              Larry                  150000              40

8 rows selected.

SQL>
```

Figure 5-40 Multiple-column sort.

column as the sort field (see Fig. 5-37). In this example, if the optional word ASC is added to the ORDER BY clause, the sort clause will look like this:

```
ORDER BY Last ASC;
```

Now, let us find employees with their salaries in descending order (see Fig. 5-38). The employee with the highest salary will be at the top, and the employee with the lowest salary will be at the bottom. We will restrict it to employees belonging to department 30 only. There are only two employees in department 30, and the result shows the employee with the higher salary first.

Next, let us use an expression in the SELECT statement and give it a column alias. We will use the column alias as our sort column. The alias *monthlysalary* represents the monthly salary of each employee, and it is also used for sorting data (see Fig. 5-39).

In our next example, we will sort by two different columns, and each column will be sorted in a different order. In case of a sort by multiple columns, the first column is the primary sort column, and the second column is the secondary sort column. The rows are sorted based on the primary sort column first. Then, the rows with the same value in the primary sort columns are sorted within their group using the secondary sort column. For example, in sorting the EMPLOYEE table using DeptId as the primary sort column in ascending order and Salary as the secondary sort column in descending order, DeptId will be sorted first. Then, within each DeptId, rows will be sorted based on Salary in reverse order (see Fig. 5-40).

REVISITING SUBSTITUTION VARIABLES

The substitution variables can be used in statements other than the INSERT statement. They can substitute for column names, table names, expressions, or text. Their use is to generalize queries by inserting them as follows:

- In the SELECT statement in place of a column name.
- In the FROM clause in place of a table name.
- In the WHERE clause as a column expression or text.
- As an entire SELECT statement.

If a variable is to be reused within a query without getting a prompt again for the same variable, the double-ampersand (&&) substitution variable is used. The user gets only one prompt for the variable with &&, and the value of the variable is then used more than one time. In Fig. 5-41, the variable *columnname* is used twice, once in the SELECT statement in the column list and then again as the sort column in the ORDER BY clause. The user, however, gets only one prompt for the variable.

```
SQL> SELECT Last, First, &&columnname
  2    FROM student
  3    ORDER BY &columnname;
Enter  value for columnname: MajorId
old    1: SELECT  Last,  First,    &&columnname
new    1: SELECT  Last,  First,    MajorId
old    3: ORDER BY &columnname
new    3: ORDER BY MajorId

LAST                 FIRST                MAJORID
---------------      ---------------      ----------
Diaz                 Jose                      100
Khan                 Amir                      200
Patel                Rajesh                    400
Tyler                Mickey                    500
Rickles              Deborah                   500
Lee                  Brian                     600

6 rows selected.

SQL>
```

Figure 5-41 The && substitution variable.

DEFINE COMMAND

A variable can be defined at the SQL> prompt. The variable is assigned a value that is held until the user exits from SQL*Plus or undefines it. The general syntax is

```
DEFINE variable [= value]
```

For example,

```
DEFINE Last = Shaw
```

The variable *last* gets the value *Shaw*, which can be used as a substitution variable in a query. For example,

```
SELECT * FROM employee WHERE Lname = '&Last';
```

The DEFINE Last command will return the value of the variable if it already has a value; otherwise, Oracle will display an "UNDEFINED" message.

The variable's value can be erased with the UNDEFINE command. For example,

```
UNDEFINE last
```

The variable is valid during a session only. If you want to use a variable every time you log in, it can be defined in your login script file (login.sql).

```
SQL> DEFINE Temp = 999
SQL> DEFINE
DEFINE _CONNECT_IDENTIFIER = "oracle" (CHAR)
DEFINE _SQLPLUS_RELEASE = "902000100" (CHAR)
DEFINE _EDITOR = "Notepad" (CHAR)
DEFINE _O_VERSION = "Oracle9i Enterprise Edition Release
9. 2 . 0 . 1 . 0 - Production
With the Partitioning, OLAP and Oracle Data Mining options
JServer Release 9 . 2 . 0 . 1 . 0 - Production" (CHAR)
DEFINE _O_RELEASE = "902000100" (CHAR)
DEFINE DEPT_ID = "80" (CHAR)
DEFINE LOCATION = "Monroe" (CHAR)
DEFINE DEPT_NAME = "Accounting" (CHAR)
DEFINE MANAGER = "NULL" (CHAR)
DEFINE COLUMNNAME = "MajorId" (CHAR)
DEFINE TEMP = "999" (CHAR)
SQL>
```

Figure 5-42 The DEFINE command.

Figure 5-42 shows use of DEFINE command to define variable Temp with a value of 999. The DEFINE command then displays all defined variables for the session, including the just-defined variable Temp. The variables are stored with data-type CHAR.

CASE STRUCTURE

CASE structure is allowed anywhere expressions are allowed in SQL statements. SQL's CASE strcture is similar to the SELECT . . . CASE statement in Visual Basic language and the switch . . . case statement in C++ and Java. The general syntax of CASE is

```
CASE WHEN condition1 THEN
        Expression1
     WHEN condition2 THEN
        Expression2
     . . .
     [ELSE Expression]
END
```

In Figure 5-43, CASE structure is illustrated with an UPDATE statement. The Salary column is updated with different increments based on employees' department Id. Employees in Department 10 get a 10% raise, employees in Department 20 get a 5% raise, and others do not get any raise!

```
SQL> UPDATE employee
  2   SET Salary = CASE  WHEN  DeptId = 10 THEN
  3                          Salary * 1.10
  4                      WHEN  DeptId = 20 THEN
  5                          Salary * 1.05
  6                      ELSE  Salary
  7                      END
  8  /

8 rows updated.

SQL>
```

Figure 5-43 UPDATE with CASE structure.

IN A NUTSHELL . . .

- The Data Manipulation Language (DML) statements INSERT, UPDATE, and DELETE are used for data maintenance.
- Date and character values are enclosed in single quotation marks. Oracle is case sensitive about character values and format sensitive about date values. The default date format is DD-MON-YY. Number values are not enclosed in quotation marks.
- To use a character value with a single quotation mark, use the single quotation mark twice in succession.
- Null values can be entered implicitly or explicitly. A null value is an unknown and undefined value.
- A foreign key value in a child table is allowed only if the value exists in the parent table's primary key.
- A user can create interactive scripts by using substitution variables and the ACCEPT command.
- The UPDATE statement is used to change existing data.
- Conditions use column, value, relational, logical, and other special operators.
- Every INSERT, DELETE, and UPDATE command is successful provided the integrity constraints are not violated.
- A constraint can be dropped permanently or disabled temporarily using the ALTER TABLE statement. The disabled constraint can be enabled later.
- The SELECT query is used to retrieve data from existing tables.
- The wild-card* is used to display all columns with the SELECT query.
- The DISTINCT clause in front of a column name returns nonduplicate values only.
- A column alias is used for more descriptive column headings because the column names are abbreviated. By default, column names are displayed as column headings in data retrieval.

- A column alias can also be used for expressions. A column alias is enclosed in double quotation marks to preserve case or to use special characters.
- The concatenation characters (||) join a column or a character string to another column.
- The arithmetic operations are performed on number data types for data manipulation. Whatever is in parentheses is performed first. Operators * and / have higher precedence than + and − operators.
- Use of a null value in an arithmetic expression returns a null result.
- The logical operators AND and OR are used to evaluate multiple conditions.
- The special comparison operator LIKE is used when wild cards % and _ are used for pattern matching. The wild-card % represents zero or more characters. The wild card _ represents one character only.
- The order of rows in a table is undefined. The rows can be displayed in a sorted order with the ORDER BY clause in the SELECT statement. The default sort order is ascending. The DESC keyword is used for sorting in descending order.
- The && substitution variable is used to reuse a variable's value without getting prompted again.
- The DEFINE and UNDEFINE commands are used at the SQL*Plus prompt to assign or erase a variable's value.
- The CASE structure allows a user to perform different actions based on the supplied conditions' outcome. It is similar to Visual Basic's SELECT ... CASE and to C++ and Java's switch ... case statements. CASE structure can also be used with other SQL statements.

EXERCISE QUESTIONS

True/False:

1. In Oracle, character values are enclosed in single quotation marks, but date and number values are not.
2. In Oracle9i, the default display format for date is DD-MM-YYYY.
3. A null value is not defined or not known.
4. The UPDATE statement without the WHERE clause can update all rows in the table.
5. A column alias is enclosed in double quotation marks to display a column name in uppercase only.
6. The AND operator returns a TRUE result if one of the two conditions is true.
7. The substitution variable can be deleted with UNDEFINE command.
8. The WHERE clause restricts individual rows, but it does not filter columns.
9. There is no restriction in deleting a row from a parent table.
10. The DELETE statement without a WHERE clause has the same effect as TRUNCATE.

11. A null value may be inserted into a column by using NULL or '' (two single quotes).
12. The SELECT statement can be used to modify data in a table.
13. A search condition with wild card may not use equal (=) operator.
14. If the ORDER BY clause is used with the DESC option on a date column, the most current date is displayed first.
15. If the ORDER BY clause is used with the default sort order on a character column, the column values are displayed in alphabetical order.

List Output/Message from the Following Queries/Statements (Use Tables Created in the Chapter 4 Lab Activity):

1. SELECT First ‖' '‖ Last "Name", BirthDate FROM student;
2. SELECT DISTINCT (MajorId) FROM student;
3. SELECT* FROM location ORDER BY Building, Capacity DESC;
4. SELECT Lname, Fname, (Salary / 12) MONTHLYSALARY FROM employee ORDER BY MONTHLYSALARY;
5. DELETE FROM faculty WHERE DeptId = 2;

Indicate Which of the Following Queries/Statements Will Result in an Error Message and Why (Use Tables Created in the Chapter 4 Lab Activity):

1. SELECT LastName, FirstName FROM student;
2. SELECT DeptId,* FROM dept;
3. INSERT INTO dept VALUES (77, RESEARCH, NULL, NULL);
4. UPDATE employee
   SET DeptId = 88
   WHERE EmployeeId = 111;
5. DELETE FROM dept WHERE DeptId = 10;

LAB ACTIVITY

Write Queries/Statements for the Following (Use Tables Created in Chapter 4 Lab Activity):

1. Display all employee names (last name and first name separated by a comma and a space) and salary with appropriate column aliases.
2. Display all employees who do not get any commission.
3. Display unique building names from the LOCATION table.
4. Display all course sections offered in Winter 2003.
5. Display names of faculty members who work in Department 1 or 2. Use the IN operator in your query.
6. Find all New York and New Jersey students.
7. Give a 10% raise to employee number 111.
8. Delete Department 30 from the department table. If this is not successful, write down your suggestion to make it work.
9. For each CourseId, display the maximum count in descending order.
10. Insert a new term in the TERM table.

11. Create a custom prompt for the user to input any value between 50 and 99 into the DeptId column.
12. Find courses with no required prerequisite.
13. Display faculty names in descending order by their department but in alphabetical order by their name within each department.
14. Find faculty members whose name start with the letter *C*.
15. Find students who started in the year 2003. Use the start term column and wild card.
16. Write the SQL*Plus command to display a character column in 30 columns and a numeric column with 9,999.99 format.

6

Working with Tables: Functions and Grouping

IN THIS CHAPTER . . .

- Data retrieval statements are written using single-row functions.
- Numeric, character, conversion, and miscellaneous functions are introduced and used.
- Data are manipulated using aggregate or group functions.
- Various clauses are used with data retrieval queries for filtering and grouping of data.

BUILT-IN FUNCTIONS

The built-in functions provide a powerful tool for the enhancement of a basic query. A function takes zero or more arguments and returns a single value. Just like other software and programming languages, the functions covered in this section are specific to Oracle. Functions are used for performing calculations on data, converting data, modifying individual data, manipulating a group of rows, and formatting columns. In Oracle's SQL, there are two types of functions:

1. ***Single-row functions,*** which work on columns from each row and return one result per row.
2. ***Group functions*** or ***aggregate functions***, which manipulate data in a group of rows and return single result.

Single-Row Functions

The single-row functions take different types of arguments, work on a data item from each row, and return one value for each row. The arguments are in the form of a constant value, variable name, column, and/or expression. The value returned by a function may be of a different type than the argument(s) supplied. The general syntax is

Function(column | expression [, argument1, argument2, . . .])

where *function* is the name of the function, *column* is a column from a table, *expression* is a character string or a mathematical expression, and *argument* is any argument used by the function.

There are various types of single-row functions:

- ***Character functions*** take a character string or character-type column as an argument and return a character or numeric value.
- ***Number functions*** take a number or number-type column as an argument and return a numeric value.
- ***Date functions*** take a date value or date-type column as an argument and return date-type data. (*Exception:* The MONTHS_BETWEEN function returns a numeric value.)
- ***Conversion functions*** convert value from one data type to another.
- ***General functions*** perform different tasks.

Character Functions. The character functions perform case conversion or character manipulation. Figure 6-1 has a list of character functions and their use. The case-conversion character functions change a string or character-type column data's case. For example,

UPPER('Oracle')	→ 'ORACLE'
LOWER('DaTaBaSe SyStEmS')	→ 'database systems'
INITCAP('DaTaBaSe SyStEmS')	→ 'Database Systems'

For example, Figure 6-2 shows the use of character functions UPPER, LOWER, and INITCAP in the SELECT clause to display columns with different cases.

Often, more than one data-entry person will populate a table. One person enters names in all uppercase, and the other uses proper case. This could become a nightmare for data retrieval query writers if not for functions. Functions are very useful in the WHERE clause's conditions as well.

For example, in Figure 6-3, a query is issued with the condition *WHERE State* = '*ny*', and it resulted in a "no row selected" message. The table does contain students from New York state. The problem here is the case used in entering state values. The same query is rewritten with condition *WHERE UPPER*(*State*) = '*NY*', and it returned two student names. The use of the UPPER function converted the value in the

Character Function	Use
UPPER(*column* | *expr*)	Converts each letter to uppercase.
LOWER(*column* | *expr*)	Converts each letter to lowercase.
INITCAP(*column* | *expr*)	Converts character value to the proper case (i.e., first character of each word is converted to uppercase and the rest to lowercase).
CONCAT(*column* | *expr, column* | *expr*)	Joins the first value to the second value. Similar to the ‖ operator discussed earlier.
SUBSTR(*column*| *expr, x, y*)	Returns a substring, starting at character position *x*, and returns *y* number of characters.
SUBSTR(*column* | *expr, z*)	Returns a substring, starting at character position *z* and going to the end of string.
INSTR(*column* | *expr, c*)	Returns the position of the supplied character.
LTRIM(*column* | *expr, c*)	Removes the leading supplied character.
RTRIM(*column* | *expr, c*)	Removes the trailing supplied character.
TRIM('*c*' *FROM column* | *expr*)	Removes the leading and trailing characters.
TRIM(*column*)	Removes the leading and trailing spaces only.
LENGTH(*column* | *expr*)	Returns the number of characters.
LPAD(*column* | *expr, n,* '*str*')	Pads the value with 'str' to the left to a total width of *n*.
RPAD(*column* | *expr, n,* '*str*')	Pads the value with 'str' to the right to a total width of *n*.
REPLACE(*column* | *expr, c, r*)	Replaces substring *c*, if present in the column or expression, with string *r*.

Figure 6-1 Character functions.

```
SQL> SELECT  UPPER(Lname),   LOWER(Fname),
  2       INITCAP(Fname   ||    ' '    ||   Lname)
  3  FROM employee;

UPPER(LNAME)      LOWER(FNAME)      INITCAP(FNAME||''||LNAME)
---------------   ---------------   -----------------------------
SMITH             john              John Smith
HOUSTON           larry             Larry Houston
ROBERTS           sandi             Sandi Roberts
MCCALL            alex              Alex Mccall
DEV               derek             Derek Dev
SHAW              jinku             Jinku Shaw
GARNER            stanley           Stanley Garner
CHEN              sunny             Sunny Chen

8 rows selected.

SQL>
```

Figure 6-2 Character functions in SELECT.

```
SQL> SELECT Last, First FROM student
  2  WHERE State='ny';

no rows selected

SQL> SELECT Last, First FROM student
  2  WHERE UPPER(State) = 'NY';

LAST                 FIRST
-------------------- --------------------
Tyler                Mickey
Lee                  Brian

SQL>
```

Figure 6-3 Character function in WHERE.

column to uppercase, and it was then compared to the value NY, which has same case. The condition can be written as *WHERE LOWER*(*State*) = '*ny*' instead.

A character function is used in various SELECT clauses, including the ORDER BY clause. The LENGTH function returns the length of a character column or string literal. Suppose we want to see names in ascending order by length of names. The clause will use *ORDER BY LENGTH(Last)* instead of just *ORDER BY Last*, as shown in Figure 6-4.

```
SQL> SELECT Last, First FROM student
  2  ORDER BY LENGTH(Last);

LAST                 FIRST
-------------------- --------------------
Lee                  Brian
Diaz                 Jose
Khan                 Amir
Tyler                Mickey
Patel                Rajesh
Rickles              Deborah

6 rows selected.

SQL>
```

Figure 6-4 Character function in ORDER BY.

In Oracle9i, the LENGTH function is enhanced with other functions like LENGTHB (to get length in bytes instead of characters) and LENGTHC (to get length in unicode).

The character manipulation functions manipulate a character-type value to return another character- or numeric-type result. For example,

CONCAT('New', 'York') → 'NewYork'
SUBSTR('HEATER', 2, 3) → 'EAT'

INSTR('abcdefg', 'd')	→ 4
LTRIM('00022345', '0')	→ '22345'
RTRIM('0223455', '5')	→ '02234'
TRIM(' 'FROM' Monroe')	→ 'Monroe'
LENGTH('Oracle9i')	→ 8
LPAD(265000, 9, '$')	→ $$$265000
RPAD(265000, 9, '*')	→ 265000***
REPLACE('Basketball', 'ket', 'e')	→ 'Baseball'

In Oracle9i, the INSTR function is enhanced to take more arguments:

```
SELECT INSTR('CORPORATE FLOOR DOOR', 'OR' 1, 2) FROM dual;
```

where we are looking for string 'OR' and function is asked to start at the first character from the left to look for the second occurrence of the string. The result is 5, as the second 'OR' starts at position 5. If argument 1 is changed to −3, the function will start at the third character from right and search in the reverse direction.

Numeric Functions. The numeric functions take numeric value(s) and return a numeric value. The ROUND function rounds the value, expression, or column to *n* decimal places. If *n* is omitted, zero decimal place is assumed. If *n* is negative, rounding takes place to the left side of the decimal place. For example,

```
ROUND(25.465, 2) = 25.47
ROUND(25.465, 0) = 25
ROUND(25.465, –1) = 30
```

The TRUNC function truncates the value, expression, or column to *n* decimal places. If *n* is not supplied, zero decimal place is assumed. If *n* is negative, truncation takes place to the left side of the decimal place. For example,

```
TRUNC(25.465, 2) = 25.46
TRUNC(25.465, 0) = 25
TRUNC(25.465, –1) = 20
```

The POWER function finds the power of a number (n^p). For example,

```
POWER(2, 4) = 16
POWER(5, 3) = 125
```

The ABS function returns the absolute value of a column, expression, or value. For example,

```
ABS( –10) = 10
```

The MOD function finds the integer remainder of *x* divided by *y*. It ignores the quotient. For example,

```
MOD(5, 2) = 1
MOD(3, 5) = 3
MOD(8, 4) = 0
```

The SIGN function returns −1 for a negative number, 1 for a positive number, and 0 for a zero. For example,

```
SIGN(-50) = –1
SIGN(+43) = 1
SIGN(0) = 0
```

The FLOOR function is similar to the TRUNC function, and the CEIL function is similar to the ROUND function. However, both take one argument instead of two. For example,

```
FLOOR(54.7) = 54
CEIL(54.7) = 55
```

There is a difference in CEIL function, because it always returns the next higher integer value. For example,

```
ROUND (54.3) = 54
CEIL (54.3) = 55
```

Figure 6-5 shows the use of numeric functions, and Figure 6-6 explains their use. Did you notice the table name in Figure 6-5? The table is called DUAL, which is provided by Oracle. The DUAL table is owned by user SYS, and it is available to all users. The DUAL table is useful when you want to find the outcome of a function and the argument is not taken from any table. The DUAL table can also be used perform arithmetic expressions. For example,

```
SELECT 25000 * 0.25 FROM DUAL;
```

The DUAL table contains a single column called DUMMY and a single row with value X (see Fig. 6-7).

Date Functions. We already know that Oracle stores dates internally with day, month, year, century, hour, minute, and second information. The default date display format is DD-MON-YY. There is a very useful date function called SYSDATE that does not take any arguments. SYSDATE returns the system's current date. For example,

```
SELECT SYSDATE
FROM DUAL;
```

```
SQL> SELECT ROUND(5.55, 1), TRUNC(5.5), SIGN(-5.5), MOD(5,2),
  2  ABS(-5), POWER(3, 4), FLOOR(5.5), CEIL(5.5)
  3  FROM DUAL;

ROUND(5.55,1) TRUNC(5.5) SIGN(-5.5) MOD(5,2) ABS(-5) POWER(3,4) FLOOR(5.5) CEIL(5.5)
------------- ---------- ---------- -------- ------ ---------- ---------- ---------
          5.6          5         -1        1      5         81          5         6

SQL>
```

Figure 6-5 Using numeric functions.

Numeric Function	Use
ROUND(*column* | *expr, [n]*)	Rounds the column or expression to *n* decimal places.
TRUNC(*column* | *expr, [n]*)	Truncates the column or expression to *n* decimal places.
POWER(*n, p*)	Returns *n* raised to power $p(n^p)$.
ABS(*n*)	Returns the absolute value of *n*.
MOD(*x, y*)	Returns the integer remainder of *x*/*y*.
SIGN(*value*)	Returns 1 for positive, −1 for negative and 0 for a zero.
FLOOR(*value*)	Returns the largest integer less than or equal to value.
CEIL(*value*)	Returns the smallest integer greater than or equal to value.

Figure 6-6 Numeric functions.

```
SQL> DESCRIBE DUAL

Name                           Null?    Type
------------------------------ -------- ----------
DUMMY                                   VARCHAR2(1)

SQL> SELECT * FROM dual;

D
-
X

SQL>
```

Figure 6-7 DUAL table.

This query will display the current date. You can get more information about day, date and time by using a format mask with SYSDATE function. Try the following statement, which will make more sense after format masks are explained in Fig. 6-15:

```
SELECT TO_CHAR(SYSDATE, 'DY, MONTH DD, YYYY HH:MI:SS P.M.')
FROM DUAL;
```

Similarly, the DUAL table can be used to display the outcome of any character and number functions or an arithmetic expression.

The date-type column is very important. You can derive a lot of information from date columns by performing "date arithmetic." As you see in Figure 6-8, you can add or subtract a number of days to or from a date to get a new resulting date. You can also add a number of hours to a date. If you have two dates, you can find the gap in days between them.

Date Expression	Result
Date + number	Adds a number of days to a date.
Date − number	Substracts a number of days from a date.
Date + number/24	Adds a number of hours to a date.
Date1 − Date2	Gives the number of days between two dates.

Figure 6-8 Date arithmetic.

In Figure 6-9, we have an expression (SYSDATE − BirthDate) that finds the difference in days. Then, we divide the number of days by 365 to convert it to years.

```
SQL> SELECT Last, First, (SYSDATE – BirthDate) / 365 AGE
  2  FROM student;

LAST            FIRST           AGE
--------------- --------------- ----------
Diaz            Jose            20.8289038
Tyler           Mickey          19.7330134
Patel           Rajesh          17.9960271
Rickles         Deborah         33.1521915
Lee             Brian           18.0343832
Khan            Amir            19.4289038

6 rows selected.

SQL>
```

Figure 6-9 Age calculation from birth date.

To take leap years into consideration, we can divide by 365.25 days instead of by 365. The resulting age has a decimal value.

You can truncate (with the TRUNC function) the result to zero decimal places, and the age will be a whole number. The modified expression will look like this:

```
TRUNC((SYSDATE – BirthDate) / 365.25)
```

Similarly, we can find the number of months or number of weeks by dividing days by 30 or 7, respectively.

The common date functions and their use are given in Figure 6-10. The function MONTHS_BETWEEN returns a number. If *date1* is later than *date2*, the result is positive; otherwise, the result is negative. The decimal part in the result is because of the portion of the month or extra days of the month. It is useful in finding the delay between delivery date and payment date. For example,

MONTHS_BETWEEN('02-DEC-03', '04-APR-03') → 7.93548387

Date Function	Use
MONTHS_BETWEEN(date1, *date2*)	Finds the number of months between two dates.
ADD_MONTHS(date, *m*)	Adds calendar months to a date.
NEXT_DAY (date, 'day')	Finds the next occurrence of a day from the given date.
LAST_DAY(date)	Returns the last day of the month.
ROUND(date [, 'format'])	Rounds the date to the nearest day, month, or year.
TRUNC(date [, 'format'])	Truncates the date to the nearest day, month, or year.
EXTRACT(**YEAR | MONTH | DAY FROM date**)	Extracts the year, month, or day from a date value.
NEXT_TIME(*date, existing timezone, newtimezone*)	Returns the date in different time zone, such as EST or PST.

Figure 6-10 Date functions.

The function ADD_MONTHS adds the number of months supplied as a second argument. The number must be an integer value. It can be positive or negative. For example, if an item is shipped today and payment is due in three months, what is the payment date?

ADD_MONTHS('10-MAY-03', 3) → '10-AUG-03'

The function NEXT_DAY returns the next occurrence of a day of the week following the date supplied. The second argument could be a number in quotes or a day of the week. For example,

NEXT_DAY ('14-OCT-03', 'SUNDAY') → '19-OCT-03'
NEXT_DAY('14-OCT-03', 'TUESDAY') '21-OCT-03'

The function LAST_DAY finds the last date of the month for the date supplied as an argument. If something is due by the end of this month, what is that date? For example,

LAST_DAY('05-FEB-04') → '29-FEB-04'

The ROUND function rounds a date based on the format specified. If a format is missing, rounding is to the nearest day. For example,

```
ROUND(TO_DATE('07/20/03', 'MM/DD/YY'), 'MONTH') → '01-AUG-03'
```

Here, the date is nearer to August 1 than to July 1. In the next example, the date is nearest to the first of next year:

```
ROUND(TO_DATE('07/20/03', 'MM/DD/YY'), 'YEAR') → '01-JAN-04'
```

The TRUNC function truncates the date to the nearest format specified. Truncation to the nearest month returns the first day of the date's month, and truncation to the nearest year returns the January 1 of the date's year. For example,

```
TRUNC(TO_DATE('07/20/03', 'MM/DD/YY'), 'MONTH') → '01-JUL-03'
TRUNC(TO_DATE('07/20/03', 'MM/DD/YY'), 'YEAR') → '01-JAN-03'
```

The EXTRACT function extracts year, month, or day from a date value.For example,

```
SELECT EXTRACT(MONTH FROM sysdate),
     EXTRACT(DAY FROM sysdate),
EXTRACT(YEAR FROM sysdate) FROM dual;
```

The following is a list of a few more date- and time-related functions introduced in Oracle9i:

CURRENT_DATE—returns the current date the in session's time zone.

CURRENT_TIMESTAMP—returns the current date and time in the session's time zone.

DBTIMEZONE—returns the value of the database's time zone.

SESSIONTIMEZONE—returns the current session's time zone.

SYSTIMESTAMP—returns the date and time in the time zone of the database.

Other Functions. The NVL function converts a null value to an actual value supplied as an argument. The second argument is enclosed within the single quotation marks for columns with DATE, CHAR, or VARCHAR2 data types. The general syntax is

```
NVL (column, value)
```

If the value in a column is null, convert it to a specified value. For example,

```
NVL(Commission, 0)
NVL(HireDate, '01-JAN-03')
NVL(PreReq, 'None')
```

If the commission amount is null, convert it to zero. If HireDate is not entered, use '01-JAN-03' for it. If prerequisite is null, use 'None'.

Now, we will revisit our query in Fig. 5-18, where we tried to add Salary and Commission columns. The total was blank for employees without a value in the Commission column. Remember, any number plus a null value is equal to null. Let us rewrite the same query using the NVL function (see Fig. 6-11).

```
SQL> SELECT Lname, Fname,
  2         Salary + NVL(Commission, 0) "Total Salary"
  3  FROM employee;

LNAME           FNAME           Total Salary
--------------- --------------- ------------
Smith           John                  300000
Houston         Larry                 160000
Roberts         Sandi                  75000
McCall          Alex                   66500
Dev             Derek                 100000
Shaw            Jinku                  27500
Garner          Stanley                50000
Chen            Sunny                  35000

8 rows selected.

SQL>
```

Figure 6-11 Arithmetic with the NVL function.

An extension of the NVL function is the NVL2 function. It takes three parameters instead of the two parameters used by NVL function. The NVL2 function checks for null as well as not null values. If the column has a not null value, the second parameter is displayed. If the column has a null value, the third parameter is displayed. The general syntax is

NVL2(column, notnullvalue, nullvalue)

For example,

NVL2(PreReq, 'YES', 'NO')

If prerequisite has a not null value, YES is displayed. If prerequisite is null, NO is displayed.

Another similar function is COALESCE. It is also an extension to the NVL function. The NVL function specifies a single alternative for a null value, whereas the COALESCE function provides multiple alternatives. The general syntax is

COALESCE(column, alternative1, alternative2, . . .)

For example,

```
COALESCE(Commission, Salary, -1)
```

In other words, if the commission value is not null, then display it. If commission value is null, then display salary value. If salary value is null, then display −1.

The NULLIF function generates null values. First, it compares two expressions. Then, if their values are equal, it generates a null, or it returns the first expression. The general syntax is

```
NULLIF(exp1, exp2)
```

For example,

```
NULLIF(Supervisor, 111)
```

If Supervisor is equal to 111, then a null is displayed; otherwise, the supervisor's value is displayed.

The DECODE function is a conditional statement type of function. If you are familiar with any programming language like Visual Basic 6 (If ... ElseIf or Select ... Case structures) or C (if ... else if or switch ... case structures), you will understand the function with ease. The DECODE function tests a column or expression and for each of its matching value, provides an action. The general syntax is

```
DECODE(column | expr, value1, action1,
    [value2, action2, . . . ,]
    [, default]);
```

The default action is provided for any value that does not match the values checked within the function. If the default value is not used, a null value is returned for nonmatching values. For example, we are displaying new salary for all employees based on their PositionId. PositionId 1 gets a 20% raise, 2 gets 15%, 3 gets 10%, 4 gets 5%, and others get no increment at all. If the last default salary is not included in the statement, the new salary for employees with PositionId 5 is displayed as null.

```
SELECT Lname, Salary,
DECODE(PositionId,    1, Salary * 1.2,
                      2, Salary * 1.15,
                      3, Salary * 1.1,
                      4, Salary * 1.05,
                         Salary) "New Salary"
FROM employee;
```

The CASE structure is an easier alternative to the DECODE function. It also uses a conditional expression, with the key words WHEN and THEN. A CASE structure ends with the key word END. The CASE structure can be used in the SELECT,

FROM, WHERE, or ORDER BY clause. Let us rewrite the DECODE function (given above) with CASE structure:

```
SELECT Lname, Salary,
CASE    WHEN PositionId = 1 THEN Salary*1.2
        WHEN PositionId = 2 THEN Salary*1.15
        WHEN PositionId = 3 THEN Salary*1.1
        WHEN PositionId = 4 THEN Salary*1.05
        ELSE Salary
END "New Salary"
FROM employee;
```

Conversion Functions. The conversion functions convert data from one data type to another. The Oracle server follows some rules to convert data type implicitly. For example, if you enter a character string that includes a valid number, the Oracle server can successfully convert CHAR data to NUMBER data. If you enter a date as a string and use the default date format DD-MON-YY, the Oracle server can perform CHAR-to-DATE conversion successfully. It is advisable to use explicit data conversion functions for successful and reliable queries. The three conversion functions shown in Fig. 6-12 are used for explicit data-type conversion in queries.

Conversion Function	Use
TO_CHAR(*number* | *date [, format]*)	Converts a number or a date to a VARCHAR2 value based on the format provided.
TO_NUMBER(*char* [, *format*])	Converts a character value with valid digits to a number using the format provided.
TO_DATE(*char* [, *format*])	Converts a character value to date value based on the format provided. Default format is DD-MON-YY.

Figure 6-12 Conversion functions.

The TO_CHAR function converts a number or date value to its character equivalent. The format argument is enclosed in single quotation marks, and the format value is case sensitive. Figures 6-13 to 6-16 describe common formats for number and date with examples. In Figure 6-16, fill mode (fm) is used to remove unnecessary spaces or zeroes in the front or in the middle.

Number Format	Meaning
9	Number of 9s to determine length (e.g., 99999).
0	Displays leading zeroes (e.g., 099999).
$	Displays floating dollar sign (e.g., $99999).
.	Displays decimal point in specified location (e.g., 99999.99).
,	Displays comma in specified location (e.g., 99,999).
PR	Puts negative numbers in parenthesis (e.g., 99999PR).

Figure 6-13 Number formats.

```
SQL> SELECT Lname, Fname, TO_CHAR(Salary, '$999,999') SALARY
  2  FROM employee;

LNAME           FNAME           SALARY
--------------- --------------- ---------
Smith           John            $265,000
Houston         Larry           $150,000
Roberts         Sandi           $75,000
McCall          Alex            $66,500
Dev             Derek           $80,000
Shaw            Jinku           $24,500
Garner          Stanley         $45,000
Chen            Sunny           $35,000

8 rows selected.

SQL>
```

Figure 6-14 TO_CHAR with number format.

Date/Time Format	Meaning
YYYY	Four-digit year
Y, YY, or **YYY**	Last one, two, or three digits of the year
YEAR	Year spelled out
Q	Quarter of the year
MM	Two-digit month
MON	First three letters of the month
MONTH	Month name using nine characters; left characters padded with spaces
Month	Same as MONTH, but in InitCap format
RR	Two-digit year based on century (previous century for years 50 to 99 and current century for years 00 to 49)
RM	Month in Roman numerals
WW or **W**	Week number of year or month
DDD, DD, or **D**	Day of year, month, or week
DAY	Name of day using nine characters; left characters padded with blanks
DY	Three-letter abbreviated name of day
DDTH	Ordinal number (e.g., seventh)
DDSP	Spelled-out number
DDSPTH	Spelled-out ordinal number
HH, HH12, or **HH24**	Hour of day, or hour (0–12), or hour (0–23)
MI	Minute (0–59)
SS	Second (0–59)
SSSSS	Seconds from midnight (0–86399)
"of"	String in quotes displayed in the result
fm	Fill mode used with other format mask (e.g., DAY) to suppress blanks

Figure 6-15 Date/time formats.

```
SQL> SELECT Last, First,
  2  TO_CHAR(BirthDate, 'fmMonth DD, YYYY') DOB
  3  FROM student
  4 /

LAST            FIRST           DOB
--------------- --------------- ------------------
Diaz            Jose            February 12, 1983
Tyler           Mickey          March 18, 1984
Patel           Rajesh          December 12, 1985
Rickles         Deborah         October 20, 1970
Lee             Brian           November 28, 1985
Khan            Amir            July 7, 1984

6 rows selected.

SQL>
```

Figure 6-16 TO_CHAR with date format (fill mode).

As we have been saying, the default date format is set to DD-MON-YY in most Oracle installations. In Oracle-9i, DD-MON-YYYY also works as a default. If you enter a birth date as 15-APR-60, it will be stored with year as 2060. That will create undesired result. If the default format is DD-MON-RR, years 50 to 99 are interpreted as 1950 to 1999 and years 00 to 49 as 2000 to 2049. If you are not sure about your session settings, use a four-digit year with INSERT as well as other SQL statements. In Figure 6-17, the character date values are converted to date type with the TO_DATE function and format mask DD-MON-RR. Then, it is changed back to character format with the TO_CHAR function and format mask YYYY.

```
SQL> SELECT TO_CHAR(TO_DATE('15-APR-60', 'DD-MON-RR'), 'YYYY') "19TH",
  2           TO_CHAR(TO_DATE('15-APR-03', 'DD-MON-RR'), 'YYYY') "20TH"
  3  FROM DUAL;

19TH        20TH
----        ----
1960        2003

SQL>
```

Figure 6-17 TO_CHAR with RR date format.

Nested Functions. Single-row functions can be nested within each other. In nested functions, the innermost function is evaluated first, and then evaluation moves outward. The outermost function is evaluated last. There is no limit on layers of nesting for single-row functions. Evaluate the expression in Figure 6-18. First, the TO_DATE function is applied to the character date string, then the TRUNC function truncates it

```
SQL> SELECT NEXT_DAY(ADD_MONTHS(TRUNC
  2         (TO_DATE('01/13/03', 'MM/DD/YY'), 'MONTH'), 3), 'TUESDAY') +7
  3        AS "TAX DAY OR BIRTHDAY"
  4  FROM DUAL;

TAX DAY O
---------
15-APR-03

SQL>
```

Figure 6-18 Nested single-row functions.

to nearest month next, the ADD_MONTHS function adds three months to it, the NEXT_DAY function finds the date on next Tuesday, and finally, seven is added to it. The result is the tax day, which happens to be my birthday also. (People get gifts on their birthday, but I always have to send a gift to Uncle Sam on mine!)

Group Functions

The group functions perform an operation on a group of rows and return one result. Look at the EMPLOYEE and STUDENT tables:

- Who makes the lowest salary?
- Who got the maximum commission?
- What is the company's total payroll?
- How many students started in the Winter 2003 semester?

It is easy to look through small tables and find answers. In a real-life situation, however, most tables have thousands or even millions of records. It is efficient to look through them with simple queries and group functions.

While using the functions described in Figure 6-19, the key words DISTINCT or ALL can be used before listing the argument in parenthesis. The key word ALL,

Group Function	Use
SUM (*column*)	Finds the sum of all values in a column; ignores null values.
AVG (*column*)	Finds the average of all values in a column; ignores null values.
MAX(*column* | *expression*)	Finds the maximum value; ignores null values.
MIN(*column* | *expression*)	Finds the minimum value; ignores null values.
COUNT(* | *column* | *expression*)	Counts the number of rows, including nulls, for *; counts nonnull values if the column or expression is used as an argument.

Figure 6-19 Group functions.

which means use all values (including duplicate values), is the default. The key word DISTINCT tells the function to use nonduplicate values only.

Let us write a query to find the total, average, highest, and lowest salaries from the EMPLOYEE table. Figure 6-20 shows the use of group functions on a number column, Salary.

```
SQL> SELECT SUM(Salary), AVG(Salary), MAX(Salary), MIN(Salary)
  2  FROM EMPLOYEE;

SUM(SALARY)      AVG(SALARY)      MAX(SALARY)      MIN(SALARY)
-------------    -----------      -----------      -----------
      741000            92625           265000            24500

1 row selected.

SQL>
```

Figure 6-20 Group functions.

Now, we will try the MAX and MIN functions on a date field. Which student from the STUDENT table was born first, and which was born last? Check out Figure 6-21. The MAX of a date returns the latest date, and the MIN of a date returns the earliest date. If you use the function on a character column, MAX will return the last name alphabetically, and MIN will return the first name alphabetically.

```
SQL> SELECT MAX(BirthDate) YOUNGEST,
  2         MIN(BirthDate) OLDEST
  3  FROM student;

YOUNGEST  OLDEST
---------  ---------
12-DEC-85  20-OCT-70

SQL>
```

Figure 6-21 Group function on a date column.

In Figures 6-22 and 6-23, uses of the COUNT function on an entire row and a column are given. In Figure 6-22, when rows are counted in the EMPLOYEE table, all eight employees' rows are counted. In Figure 6-23, when EmployeeId column values are counted, it returns eight employees. When Commission column values are counted, the null values are ignored, giving us only five commissioned employees.

We can change that using the NVL function:

```
COUNT(NVL(Commission,0))
```

```
SQL> SELECT COUNT(*)
  2  FROM employee;

  COUNT(*)
----------
         8

SQL>
```

Figure 6-22 COUNT all rows.

```
SQL> SELECT COUNT(EmployeeId), COUNT(Commission)
  2  FROM employee;

COUNT(EMPLOYEEID)      COUNT(COMMISSION)
-----------------      -----------------
                8                      5

SQL>
```

Figure 6-23 COUNT columns with and without null values.

Null values in Commission columns are replaced with 0 in the query, and the value 8 is returned from the query.

Question: Which of the following queries will return a higher average from the Commission column?

```
SELECT AVG(Commission) FROM EMPLOYEE;
SELECT AVG(NVL (Commission, 0)) FROM EMPLOYEE;
```

Answer: The first query, because it adds five commission values and divides the total by five, whereas the second query divides the total by eight. The output would be 14600 from the first query but 9125 from the second query.

GROUPING DATA

The rows in a table can be divided into different groups to treat each group separately. The group functions can be applied to individual groups in the same fashion they are applied to all rows. The GROUP BY clause is used for grouping data. The general syntax is

```
SELECT column, groupfunction(column)
    FROM tablename
    [WHERE condition(s)]
    [GROUP BY column | expression]
    [ORDER BY column | expression [ASC | DESC]];
```

Important points to remember include:

- When you include a group function and the GROUP BY clause in your query, the individual column(s) appearing in SELECT must also appear in GROUP BY.
- The WHERE clause can still be used to restrict data before grouping.
- The WHERE clause cannot be used to restrict groups.
- A column alias cannot be used in a GROUP BY clause.
- The GROUP BY column does not have to appear in a SELECT query.
- When a column is used in the GROUP BY clause, the result is sorted in ascending order by that column by default. In other words, GROUP BY has an implied ORDER BY. You can still use an ORDER BY clause explicitly to change the implied sort order.
- In Oracle9i, the order of the WHERE and GROUP BY clauses in the SELECT query does not matter, but traditionally, the WHERE clause is written before the GROUP BY clause.

In the next few figures, you will see the effect of a GROUP BY clause on queries with group functions.

As you see in Figure 6-24, the DeptId column is automatically sorted, because it is used in the GROUP BY clause. The DeptId column is not necessary in the SELECT clause, but it is a good idea to include it so the counts make sense.

```
SQL> SELECT DeptId, COUNT(*) "# of Emp"
  2  FROM employee
  3  GROUP BY DeptId
  4 /

  DEPTID          # of Emp
-------          -------
      10                3
      20                2
      30                2
      40                1

SQL>
```

Figure 6-24 COUNT rows by group.

What will happen if the query in Figure 6-24 is typed without a GROUP BY clause? If the SELECT clause contains only a group function, it does not matter. The query in the figure, in fact, has the SELECT clause with a column and a group function. When a column appears in the SELECT clause along with the group function, the column must appear in the GROUP BY clause. Failure to do so results in error ORA-00937 (see Fig. 6-25).

```
SQL> SELECT DeptId, COUNT(*) "# of Emp"
  2  FROM employee
  3  /
SELECT DeptId, COUNT(*) "# of Emp"
       *

ERROR at line 1:
ORA-00937: not a single-group group function

SQL>
```

Figure 6-25 Missing GROUP BY clause.

Can we use a condition in the WHERE clause that contains a group function? The WHERE clause can be used to restrict rows, but it cannot be used to restrict groups, as you see in Figure 6-26. In this figure, we are trying to see buildings that have four or more rooms. The error ORA-00934 states that the group function is not allowed in the WHERE clause. We can fix this problem with a new, HAVING clause, which is used for restricting groups, as you will see shortly.

```
SQL> SELECT Building, COUNT(*)
  2  FROM location
  3  WHERE COUNT(*) >= 4
  4  GROUP BY Building;
WHERE COUNT(*) >= 4
       *

ERROR at line 3:
ORA-00934: group function is not allowed here

SQL>
```

Figure 6-26 Invalid WHERE clause.

HAVING Clause

The HAVING clause can restrict groups. The WHERE clause restricts rows, the GROUP BY clause groups remaining rows, the Group function works on each group, and the HAVING clause keeps the groups that match the group condition.

In the sample query (see Fig. 6-27), the WHERE clause filters out the building named Kennedy, the rest of the rows are grouped by the building names Gandhi and Nehru, the group function COUNT counts the number of rows in each group, and the HAVING clause keeps groups with four or more rows—that is, the Gandhi building with five rows/rooms.

The implied ascending sort with the GROUP BY clause can be overriden by adding an explicit ORDER BY clause to the query. Figure 6-28 shows outcome sorted in ascending order by Building column with the GROUP BY clause, then it is reversed to descending order by inserting the ORDER BY clause in line 5.

```
SQL> SELECT Building, COUNT(*) ROOMS
  2  FROM location
  3  WHERE UPPER(Building) <> 'KENNEDY'
  4  GROUP BY Building
  5  HAVING COUNT(*) >= 4;

BUILDIN         ROOMS
-------         ------
Gandhi               5

1 row selected.

SQL>
```

Figure 6-27 HAVING clause.

```
SQL> SELECT Building, COUNT(*)
  2  FROM location
  3  GROUP BY Building
  4  HAVING COUNT(*) > 2;

BUILDIN         COUNT(*)
-------         --------
Gandhi                 5
Kennedy                4

SQL> i
  5  ORDER BY Building DESC;

BUILDIN         COUNT(*)
-------         --------
Kennedy                4
Gandhi                 5

SQL>
```

Figure 6-28 GROUP BY sort order changed with ORDER BY.

Let us look at another example of GROUP BY and HAVING. Let us find employees who have more than two dependents. We will use the DEPENDENT table and GROUP BY EmployeeId column, find COUNT of DependentId, and also check for COUNT higher than 2:

```
SELECT EmployeeId, COUNT(DependentId)
    FROM dependent
    GROUP BY EmployeeId
    HAVING COUNT(DependentId) > 2;
```

Nesting Group Functions

The single-row functions can be nested to many levels, but the group functions can only be nested to two levels. For example,

```
SELECT SUM(MaxCount) FROM crssection GROUP BY CourseId;
```

will find the total available seats for each CourseId. If you use this output for an outer function in a nested scenario as follows, you will get a different answer:

```
SELECT MAX(SUM(MaxCount)) FROM crssection GROUP BY CourseId;
```

The answer returned by this query is 85, because the outer query takes totals by each CourseId and finds the one with the largest value.

IN A NUTSHELL . . .

- Single-row functions work on each row individually. They include character functions, number functions, date functions, data conversion functions, and other general functions. All functions take zero or more arguments and return one value back.
- The NVL function converts a null value to another specified value that is provided as its second argument.
- The DECODE function is similar to the if . . . else if or case structures in programming languages.
- In an expression with nested single-row functions, the innermost function is performed first, and the outermost function is performed last.
- The SYSDATE function is an Oracle function that returns the current date from the system. SYSDATE is very useful in date arithmetic.
- The group functions work on a group of rows to return one result per group. The rows can be grouped together by using the GROUP BY clause with a SELECT query.
- The WHERE clause is used to restrict rows; similarly, the HAVING clause is used to restrict groups.
- The group functions can be nested like single-row functions. The nesting is limited to two functions for group functions.

EXERCISE QUESTIONS

True/False:

1. A single-row function may be used in the SELECT clause, but it is not allowed in the WHERE clause.
2. The SYSDATE function can return the current date but not the current time.

3. Single-row functions can be nested to two levels only.
4. The MAX function on a date column will return the latest date.
5. The TO_CHAR function converts date and numeric values to VARCHAR2.
6. The AND operator returns a TRUE result if one of the two conditions is true.
7. There is no limit in nesting group functions to multiple levels.
8. The WHERE clause restricts individual rows, but the HAVING clause restricts groups.
9. The ORDER BY clause can be used in a query with the GROUP BY clause to override the implied sort order.
10. Null values in a column are not counted by the COUNT function.

List Output from the Following Queries (Use Tables Created in the Chapter 4 Lab Activity):

1. ```
   SELECT INITCAP(First) || ' ' || INITCAP(Last) "Student Name"
   FROM student;
   ```
2. ```
   SELECT COUNT(DISTINCT(MajorId)) FROM student;
   ```
3. ```
   SELECT Building, SUM(Capacity) TOTCAP
   FROM location
   GROUP BY Building
   ORDER BY TOTCAP DESC;
   ```
4. ```
   SELECT UPPER(Lname), UPPER(Fname), (SYSDATE – HIREDATE) DAYS
   FROM employee
   ORDER BY DAYS;
   ```
5. ```
   SELECT Building, AVG(Capacity)
   FROM location
   GROUP BY Building
   HAVING AVG(Capacity) > 25;
   ```

State Which of the Following Queries Will Result in an Error Message and Why (Use Tables Created in the Chapter 4 Lab Activity):

1. ```
   SELECT UPPER(FirstName || ' ' || LastName) FROM student;
   ```
2. ```
   SELECT DeptId, COUNT (*) FROM employee;
   ```
3. ```
   INSERT INTO DEPT VALUES(90, RESEARCH, NULL, NULL);
   ```
4. ```
   SELECT DeptId, SUM(Salary)
   FROM employee
   WHERE SUM(Salary) > 200000
   GROUP BY DeptId;
   ```
5. ```
   SELECT SUM(EmployeeId)
   FROM employee;
   ```

LAB ACTIVITY

Write Queries for the Following (Use Tables Created in Chapter 4 Lab Activity):

1. Display all employee names (last name and first name separated by a comma and a space) with proper case and salary with currency format.
2. Display all employees with their commission value. Display zero commission for employees who do not get any commission.

3. Count the total number of rooms in LOCATION.
4. Count the distinct building names in LOCATION.
5. Display all student names and birth dates (display birth dates with the format 20 OCTOBER, 1970).
6. Find the average, highest, and lowest age for employees.
7. Display the total number of dependents for each employee for employees who have at least two dependents.
8. Display only the year value from each employee's hire date.
9. Find average employee commission.
   (a) Ignore nulls.
   (b) Do not ignore nulls.
10. Find sum of maximum count by term by course (GROUP BY two columns).
11. Find 2 to the power of 10.
12. Display courses and prerequisites. If there is no prerequisite, display "none" or else display "one."
13. Count number of faculty members by each department.
14. Display average employee salary by department, but do no include departments with an average salary of less than $75,000.
15. Find the number of years employees have been working. Display the integer part of the value only.
16. Find students who are born in the month of May.
17. Display employee's last name and first name, followed by salary + commission if the commission is not null or else display salary only.
18. Display each employee's full name followed by a message based on salary. If the salary is above $100,000, display "HIGH." If the salary is between $50,000 and $100,000, display "MEDIUM." If the salary is below $50,000, display "LOW."

7

Multiple Tables: Joins and Set Operations

IN THIS CHAPTER . . .

- You will learn to design queries based on multiple tables.
- You will retrieve related data from various tables in a database.
- Various types of table joins are discussed.
- Set operations UNION, UNION ALL, INTERSECT and MINUS are used.

In Chapter 5, learned data retrieval techniques to obtain data from a single table with the SELECT query. In Chapter 6, learned the use of various clauses and functions used in SELECT statements. In this chapter, we will expand on what you have learned in the previous chapters. Here, you will learn to create queries in which data are retrieved from more than one table or with more than one query. For example, a student's demographic information is in the STUDENT table, and his or her faculty advisor's information is in the FACULTY table. An employee's name is in the EMPLOYEE table, but his or her department's information is in the DEPT table, any dependents are in the DEPENDENT table, and the salary grade is in the EMPLEVEL table. Sometimes, you can accomplish tasks by joining two or more tables—or by joining a table to itself.

JOIN

When the required data are in more than one table, related tables are joined using a join condition. The join condition combines a row in one table with a row in another table based on the same values in the common columns. In most cases (but not always), the common columns are the primary key in one table and a foreign key in another. In this section, you will be introduced to different types of joins based on the join condition used.

Cartesian Product

A Cartesian product results from a multitable query that does not have a WHERE clause. The product operation joins each row in the first table with each row in the second table. The product normally results in an output with a large number of rows and is not very useful. Whenever retrieving data from more than one table, you must use one or more valid join conditions to avoid a Cartesian product! You would perform a Cartesian product operation only if you were looking to find all possible combinations of rows from two tables.

In Figure 7-1, you will see an example of a product in which all students and faculty members are matched unconditionally. All resulting rows are not shown in the figure. The last and first names are selected from the STUDENT, table and a

```
SQL> SELECT Last, First, Name
  2  FROM student, faculty;

LAST            FIRST           NAME
--------------- --------------- ---------------
Diaz            Jose            Jones
Tyler           Mickey          Jones
Patel           Rajesh          Jones
Rickles         Deborah         Jones
Lee             Brian           Jones
Khan            Amir            Jones
Diaz            Jose            Williams
Tyler           Mickey          Williams
. . .
Patel           Rajesh          Collins
Rickles         Deborah         Collins
Lee             Brian           Collins
Khan            Amir            Collins

48 rows selected.

SQL>
```

Figure 7-1 Cartesian product.

name is selected from the FACULTY table. There is no join condition issued. The result is 48 rows, because the product of two tables with *m* and *n* rows, respectively, returns $m \cdot n$ rows. The STUDENT table has 6 rows, and the FACULTY table has 8 rows, hence the result ($6 \cdot 8$ rows). If you were looking for each student's last name and his or her faculty advisor's name, you would use a join condition using the STUDENT table's foreign key FacultyId and the FACULTY table's primary key FacultyId to find matching rows.

The Cartesian product is covered in this section, but it is not a join operation. There is no join without a join condition. In Oracle, you will perform a Cartesian product by not providing enough join conditions in the SELECT query. Remember that the number of join conditions is one less than the number of table names used in the FROM clause. There are four types of joins in Oracle:

1. Equijoin.
2. Nonequijoin.
3. Outer join.
4. Self-join.

Equijoin

The equijoin is a join with a join condition involving common columns from two tables. If you need to get information about a student from the STUDENT table and corresponding information about the faculty advisor from the FACULTY table, you would use the following syntax:

```
SELECT columnnames
  FROM tablenames
  WHERE join condition(s);
```

The column names include columns from both tables separated by commas, table names are all tables used separated by commas, and the join condition is a condition that includes common columns from each table. The join condition normally (but not always) includes a foreign key column from one table and the referenced primary key column from the other table. Suppose you want to get a student's last name, the student's first name, the faculty advisor's name, and the faculty advisor's phone number. You would get them from the STUDENT and FACULTY tables. The common column in both tables is FacultyId, which is the foreign key in the child STUDENT table and the primary key in the parent FACULTY table. The join condition will return the requested information from rows in two tables where the FacultyId value is same. The rows without a match are not selected by the query. Figure 7-2 shows the result from an equijoin.

In Figure 7-2, you see that all students are picked from the STUDENT table, but faculty members are picked based on the FacultyId in the student rows. The faculty member *Collins* (FacultyId = 333) is not selected because there is no match for it

```
SQL> SELECT student.Last  || ', ' ||  student.First STUDENT,
  2           faculty.Name FACULTY,  faculty.Phone TEL
  3  FROM student,  faculty
  4  WHERE student.FacultyId =  faculty.FacultyId
  5  /

STUDENT                        FACULTY      TEL
-----------------------------  -----------  ---
Diaz, Jose                     Mobley       529
Tyler, Mickey                  Chang        587
Patel, Rajesh                  Jones        525
Rickles, Deborah               Chang        587
Lee, Brian                     Sen          579
Khan, Amir                     Williams     533

6 rows selected.

SQL>
```

Figure 7-2 Equijoin.

in the STUDENT table. On the other hand, *Chang* (FacultyId = 555) is picked twice because it appears twice as a value in the foreign key column of the STUDENT table.

The Cartesian product, as mentioned earlier, is rarely useful, but equijoin is a very important operation in database querying. Another thing to be noted is the use of *tablename.columnname*. When columns are retrieved from more than one table, the use of a table name qualifier in front of the column name tells Oracle to retrieve that column from the specified table. Oracle is pretty smart about it. If a column name exists in only one of the two tables involved in the query, it is not necessary to use a table name as a qualifier. If a column exists in both tables, you must use the table name qualifier. The join condition in an equijoin will normally have the table name qualifier. Because the join condition usually has the same column names from two tables, the column names become ambiguous without a qualifier. The qualifier actually improves performance because you are telling the Oracle server where to go to find that column. Remember that the two join columns need not have the same name.

Sometimes, the required information is in more than two tables. In this case, the FROM clause will include all needed tables, and the WHERE clause will have more than one join condition. If you need to join n tables, you would need $n - 1$ join conditions. In our NamanNavan (N2) Corporation database, an employee's demographic information is in the EMPLOYEE table. The EMPLOYEE table has three foreign keys: PositionId, referencing the POSITION table; QualId, referencing the QUALIFICATION table; and DeptId, referencing the DEPT table. You would need to join four tables to retrieve information from all those tables. This means the query will have $4 - 1 = 3$ join conditions.

The query will look like the one shown in Figure 7-3. For simplicity, we will join three tables using two join conditions. There is no limit on the number of join conditions within a query.

```
SQL> SELECT employee.Lname || ', ' || employee.Fname EMPLOYEE,
  2         dept.DeptName DEPARTMENT, position.PosDesc POSITION
  3  FROM employee, dept, position
  4  WHERE employee.DeptId = dept.DeptId
  5  AND employee.PositionId = position.PositionId;

EMPLOYEE                  DEPARTMENT     POSITION
------------------------- -------------- ----------
Smith, John               Finance        President
Houston, Larry            Marketing      Manager
Roberts, Sandi            Finance        Manager
McCall, Alex              InfoSys        Programmer
Dev, Derek                InfoSys        Manager
Shaw, Jinku               Sales          Manager
Garner, Stanley           Sales          Manager
Chen, Sunny               Finance        Accountant

8 rows selected.

SQL>
```

Figure 7-3 Multiple joins.

The multiple-join example selects information from three tables using a query with two join conditions. If you look at the query, the table qualifiers are used quite a few times. There is a way to shorten and simplify this query.

Table Aliases

In Chapter 5, you learned about column aliases, which are used for renaming column headings in a query. Table aliases are used to avoid using lengthy table names over and over again in a query. A table alias can be from 1 to 30 characters long. Normally, very short alias names are used to shorten the query and save some keystrokes. The table alias appears in the FROM clause of the SELECT query. A table name is written, followed by a space, and then a table alias is supplied. Though they appear after the SELECT clause, alias names can also be used as qualifiers for column names in the SELECT clause. All table aliases are valid only in the SELECT query, where they are named and used.

In Figure 7-4, you will see the query from Figure 7-3 with table aliases. The results obtained from the queries in Figures 7-3 and 7-4 are similar, but the query in Figure 7-4 is shortened by the use of table aliases. In this example, we joined the

```
SQL> SELECT e.Lname || ', ' || e.Fname EMPLOYEE,
  2         d.DeptName DEPARTMENT, q.QualDesc QUALIFICATION
  3  FROM employee e, dept d, qualification q
  4  WHERE e.DeptId = d.DeptId
  5  AND e.QualId = q.QualId
  6  /

EMPLOYEE             DEPARTMENT     QUALIFICATI
-------------------- -------------- ----------
Smith, John          Finance        Doctorate
Houston, Larry       Marketing      Masters
Roberts, Sandi       Finance        Masters
McCall, Alex         InfoSys        Associates
Dev, Derek           InfoSys        Doctorate
Garner, Stanley      Sales          High School
Chen, Sunny          Finance        Bachelors

7 rows selected.

SQL>
```

Figure 7-4 Table aliases.

EMPLOYEE table with the QUALIFICATION tables instead of the POSITION table. The result contains seven rows in Figure 7-4 instead of eight rows, as in Figure 7-3, because one of the employees does not have a QualId.

Additional Conditions

In addition to join conditions, you may use additional conditions using the AND operator to restrict information. Suppose you want to see the information of Figure 7-4 for employees belonging to department number 10 only. Figure 7-5 shows the use of additional condition with the AND operator where the information is displayed for DeptId = 10 only. The three tables are joined for employees in Department 10, which results in three rows instead of all eight employee rows.

Nonequijoin

There is no matching column in the EMPLEVEL table for the Salary column in the EMPLOYEE table. The only possible relationship between the two tables is between the Salary column of the EMPLOYEE table and the LowSalary and HighSalary columns in the EMPLEVEL table. The join condition for these tables can be written using any operator other than the = operator. That is why it is called nonequijoin. Figure 7-6 is an example of a nonequijoin.

The nonequijoin condition of Figure 7-6 could have been written as

```
e.Salary >= l.LowSalary AND e.Salary <= l.HighSalary;
```

```
SQL> SELECT e.Lname || ', ' || e.Fname EMPLOYEE,
  2         d.DeptName DEPARTMENT, q.QualDesc QUALIFICATION,
  3         p.PosDesc POSITION
  4  FROM employee e, dept d, qualification q, position p
  5  WHERE e.DeptId = d.DeptId
  6  AND e.QualId = q.QualId
  7  AND e.PositionId = p.PositionId
  8  AND e.DeptId = 10
  9  /

EMPLOYEE                    DEPARTMENT     QUALIFICATI POSITION
--------------------------- -------------- ---------- ---------
Smith, John                 Finance        Doctorate  President
Roberts, Sandi              Finance        Masters    Manager
Chen, Sunny                 Finance        Bachelors  Accountant

SQL>
```

Figure 7-5 Additional condition with join.

```
SQL> SELECT e.Lname || ', ' || e.Fname EMPLOYEE, e.Salary, l.LevelNo
  2  FROM employee e, emplevel l
  3  WHERE e.Salary BETWEEN l.LowSalary AND l.HighSalary
  4  /

EMPLOYEE                       SALARY    LEVELNO
------------------------- ------------ ----------
Shaw, Jinku                     24500          1
Garner, Stanley                 45000          2
Chen, Sunny                     35000          2
Roberts, Sandi                  75000          3
McCall, Alex                    66500          3
Dev, Derek                      80000          3
Smith, John                    265000          4
Houston, Larry                 150000          4

8 rows selected.

SQL>
```

Figure 7-6 Nonequijoin.

If you look at the EMPLEVEL table, none of the salaries appears in more than one level. In other words, there is no overlapping. None of the employees makes a salary that is not included in the range of salaries. For these two reasons, each employee appears once in the result. Note that none of the columns are ambiguous, so table aliases are not necessary (though they are used here in this example).

Outer Join

You saw in the equijoin that the rows from two tables are selected only if the common column values are the same in both tables. If a row in one table does not have a matching value in the other table, it is not joined. Figure 7-2 displayed all students from the STUDENT table and their advisors from the FACULTY table. Some of the faculty members are not any student's advisor, so they did not get selected. Suppose you also want to see all those faculty advisor names. Then, you would change your query's join condition and create a join known as an outer join.

The table that does not contain the matching value is known as the deficient table. In our case, the deficient table is the STUDENT table, because it does not contain all faculty IDs. The outer join uses the (+) operator in the join condition on the deficient side. (You will see it soon in Figure 7-8, which compares equijoin and outer join on these tables.) The (+) operator can be used on any side of the join condition, but it cannot be used on both sides in one condition. The general syntax is

```
SELECT tablename1.columnname, tablename2.columnname
    FROM tablename1, tablename2
    WHERE tablename1.columnname (+) = tablename2.columnname;
```

The join condition will look different if the (+) operator is used on the right side. For example,

```
WHERE tablename1.columnname = tablename2.columnname (+);
```

Figure 7-7 shows an outer join using the EMPLOYEE and QUALIFICATION tables. The outer join operator (+) is used on the QUALIFICATION side, because

```
SQL> SELECT e.Fname || ', ' || e.Lname EMPLOYEE, q.QualDesc
  2  FROM employee e, qualification q
  3  WHERE e.QualId = q.QualId (+)
  4  /

EMPLOYEE                       QUALDESC
------------------------       --------------
John Smith                     Doctorate
Larry Houston                  Masters
Sandi Roberts                  Masters
Alex McCall                    Associates
Derek Dev                      Doctorate
Jinku Shaw
Stanley Garner                 High School
Sunny Chen                     Bachelors

8 rows selected.

SQL>
```

Figure 7-7 Outer join.

```
SQL> SELECT s.First || ' ' || s.Last STUDENT,
  2         f.Name ADVISOR
  3  FROM faculty f, student s
  4  WHERE s.FacultyId (+) = f.facultyId
  5  /

STUDENT                        ADVISOR
------------------------------ ----------------------
Rajesh Patel                   Jones
Jose Diaz                      Mobley
Amir Khan                      Williams
                               Vajpayee
                               Collins
Brian Lee                      Sen
                               Rivera
Mickey Tyler                   Chang
Deborah Rickles                Chang

9 rows selected.

SQL> SELECT s.First || ' ' || s.Last STUDENT,
  2         f.Name ADVISOR
  3  FROM faculty f, student s
  4  WHERE s.FacultyId = f.facultyId
  5  /
STUDENT                        ADVISOR
------------------------------ ----------------------
Jose Diaz                      Mobley
Mickey Tyler                   Chang
Rajesh Patel                   Jones
Deborah Rickles                Chang
Brian Lee                      Sen
Amir Khan                      Williams

6 rows selected.

SQL>
```

Figure 7-8 Comparing outputs from outer join and equijoin.

it is the deficient table or it generates a null value for the row(s) in the EMPLOYEE table without any qualification value. The equijoin would have returned seven employees, but the outer join also includes one extra employee without any qualification.

Figure 7-8 compares outputs from outer join and equijoin on the same tables. The equijoin returns six students with their faculty advisors' names, whereas the outer join returns three extra rows with faculty names. The outer join operator (+) is used on the STUDENT table's side, because it generates null values for faculty members with no match in the STUDENT table.

Self-Join

A self-join is joining a table to itself. It sounds meaningless, but think about it using the following scenario: In the EMPLOYEE table, EmployeeId is the primary key column that describes each entity. For example, EmployeeId 200 represents employee Shaw, Jinku. The table also has another column called Supervisor, which contains IDs of employee supervisors. How can you find name of the supervisor for an employee? You can look up the supervisor ID, go to the EmployeeId column to find its match, and then read the name. This is easier said than done, however. A self-join is one join that is not so easy to understand.

When a table is joined to itself, two copies of the same table are loaded or used. They are treated like any two different tables, and a join is produced from those two copies. Let us explain that by using the EMPLOYEE table. The following operations are performed in the self-join of Figure 7-9:

- Two copies of the EMPLOYEE table are created with the aliases *e* and *s*.
- An employee's last name is picked from the *e* table, and the corresponding Supervisor ID is retrieved.
- The matching EmployeeId is found from the *s* table. The first employee in the *e* table does not have a supervisor and so is not picked.
- The last name from the *s* table is retrieved based on the EmployeeId.

```
SQL> SELECT e.Lname || ', ' || e.Fname Employee,
  2          s.Lname || ', ' || s.Fname Manager
  3  FROM employee e, employee s
  4  WHERE e.Supervisor = s.EmployeeId;

EMPLOYEE                  MANAGER
------------------------- --------------------
Houston, Larry            Smith, John
Roberts, Sandi            Smith, John
McCall, Alex              Dev, Derek
Dev, Derek                Smith, John
Shaw, Jinku               Garner, Stanley
Garner, Stanley           Smith, John
Chen, Sunny               Roberts, Sandi

7 rows selected.

SQL>
```

Figure 7-9 Self-join.

In short, the table is looked at twice, once for the employee and once for the supervisor. The Indo–US (IU) College database also contains a table, which can be used in self-join. Table COURSE contains a PreReq column that references its own primary key, CourseId. You will perform this self-join in the chapter's lab activity.

SET OPERATORS

In Chapter 1, you learned about union, intersection, and difference operations. If you recall, these operations are possible on "union-compatible" tables. The implementation of these operations is through the use of set operators. The union compatibility is achieved or the set operations are performed on results from two independent queries. The output from both queries must return the same number of columns, and respective columns must have a similar domain. Figure 7-10 lists all set operators and their use.

The general syntax for any set operation is

SELECT-*Query1*
Set operator
SELECT-*Query2;*

where *Set operator* is one of the four set operators described in Figure 7-10a.

Set Operator	Use
UNION	It returns all rows from both queries, but duplicate rows are not repeated.
UNION ALL	It returns all rows from both queries, and it displays all duplicate rows.
INTERSECT	It returns all rows that appear in both queries' results.
MINUS	It returns rows that are returned by the first query minus rows that are returned by the second query.

Figure 7-10a Set operators.

In Figure 7-10b, set operators UNION, INTERSECT and MINUS are shown using Venn diagrams. The left circle represents the first table, and the right circle represents the second table. The shaded area is the area selected by each operation. The UNION operator selects all rows from the first table as well as the second table, so both circles are entirely shaded. The INTERSECT operator selects rows present in both tables, so the common area is shaded. The MINUS operator selects rows in the first table that are not present in the second table, so the area exclusively belonging to the first table is shaded.

We will use the STUDENT table from the IU College database, which has all student records. Now, we will use another table called WORKER (see Fig. 7-11), which contains staff members of the college and also student workers.

Union

The UNION operator takes output from two queries and returns all rows from both results. The duplicate rows are displayed only once. If you perform union on

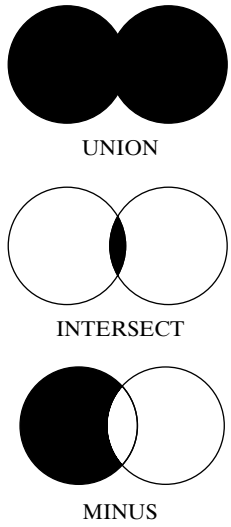


Figure 7-10b Set operators—illustration with Venn diagrams.

WORKER (WorkerId, Last, First)

WorkerId	Last	First
00110	Borges	Luz
00111	Bayer	Julia
00103	Rickles	Deborah
00113	Marte	Noemi
00105	Khan	Amir
00107	Feliciano	Sandi

Figure 7-11 WORKER table.

two very large tables, use a WHERE clause to filter rows. All six student's rows are selected from the first query, and four rows are selected from the second query. Two rows from the second query are duplicate rows (ID 00103 and ID 00105), and they are not repeated. Figure 7-12 lists all students and staff members in the result.

Union All

The UNION ALL operation is similar to the UNION operation. The only difference is that UNION ALL operation also displays duplicate rows. If you find UNION ALL of the STUDENT and WORKER tables, you will get six rows from the first query and six rows from the second query (see Fig. 7-13).

```
SQL> SELECT StudentId ID,  Last,  First FROM student
  2  UNION
  3  SELECT WorkerId, Last, First FROM worker
  4  /

ID    LAST            FIRST
----- --------------  ---------------
00100 Diaz            Jose
00101 Tyler           Mickey
00102 Patel           Rajesh
00103 Rickles         Deborah
00104 Lee             Brian
00105 Khan            Amir
00107 Feliciano       Sandi
00110 Borges          Luz
00111 Bayer           Julia
00113 Marte           Noemi

10 rows selected.

SQL>
```

Figure 7-12 UNION operation.

```
SQL> SELECT StudentId  ID,  Last, First FROM student
  2  UNION ALL
  3  SELECT WorkerId,  Last,  First FROM worker
  4  /

ID     LAST            FIRST
-----  --------------  -------------
00100  Diaz            Jose
00101  Tyler           Mickey
00102  Patel           Rajesh
00103  Rickles         Deborah
00104  Lee             Brian
00105  Khan            Amir
00110  Borges          Luz
00111  Bayer           Julia
00103  Rickles         Deborah
00113  Marte           Noemi
00105  Khan            Amir

ID     LAST            FIRST
-----  --------------  -------------
00107  Feliciano       Sandi

12 rows selected.

SQL>
```

Figure 7-13 UNION ALL operation.

Intersect

The INTERSECT operation works on output from two separate queries and returns rows that appear in both outputs. In the student and worker example, INTERSECT will return students who are also workers at the college. In Figure 7-14, you see only two student rows, which are the only students appearing in the WORKER table.

```
SQL> SELECT StudentId  ID,  Last,  First FROM student
  2    INTERSECT
  3    SELECT WorkerId,  Last, First FROM worker
  4    /

ID       LAST                FIRST
-----    -------------       ---------------
00103    Rickles             Deborah
00105    Khan                Amir

SQL>
```

Figure 7-14 INTERSECT operation.

Minus

The MINUS operation is same as the DIFFERENCE operation covered in Chapter 1. When MINUS is performed on outputs from two queries, the result is the rows in the first query's result that are not in the second query's result. Remember that the STUDENT table minus the WORKER table is not the same as the WORKER table minus the STUDENT table. Figure 7-15 is an example of a minus operation in which the result includes students who are not workers. Figure 7-16 shows workers who are not students.

Join operations are useful, but sometimes, they are achieved through a very complex query. Talking about complexity, Figure 7-17 gets information from six tables

```
SQL> SELECT StudentId  ID,  Last,  First FROM student
  2  MINUS
  3  SELECT WorkerId, Last, First FROM worker
  4  /

ID       LAST                FIRST
-----    -------------       -------------
00100    Diaz                Jose
00101    Tyler               Mickey
00102    Patel               Rajesh
00104    Lee                 Brian

4 rows selected.

SQL>
```

Figure 7-15 Student MINUS Worker.

```
SQL> SELECT WorkerId ID, Last, First FROM worker
  2  MINUS
  3  SELECT StudentId, Last, First FROM student
  4  /

ID    LAST            FIRST
----- --------------- ---------------
00107 Feliciano       Sandi
00110 Borges          Luz
00111 Bayer           Julia
00113 Marte           Noemi

4 rows selected.

SQL>
```

Figure 7-16 Worker MINUS Student.

```
SQL> SELECT e.Fname || ' ' || e.Lname "NAME",
  2         s.Fname || ' ' || s.Lname "SUPERVISOR",
  3         PosDesc, DeptName, QualDesc, LevelNo
  4  FROM employee e, position p, dept d,
  5       qualification q, employee s, emplevel l
  6  WHERE e.PositionId=p.PositionId
  7  AND e.DeptId =d.DeptId
  8  AND e.QualId = q.QualId (+)
  9  AND e.Supervisor=s.EmployeeId (+)
 10  AND e.Salary BETWEEN l.LowSalary AND l.HighSalary
 11  ORDER BY e.Lname, e.Fname;

NAME          SUPERVISOR      POSDESC    DEPTNAME    QUALDESC     LEVELNO
------------- --------------- ---------- ----------- ------------ ----------
Sunny Chen    Sandi Roberts   Accountant Finance     Bachelors             2
Derek Dev     John Smith      Manager    InfoSys     Doctorate             3
Stanley Garner John Smith     Manager    Sales       High School           2
Larry Houston John Smith      Manager    Marketing   Masters               4
Alex McCall   Derek Dev       Programmer InfoSys     Associates            3
Sandi Roberts John Smith      Manager    Finance     Masters               3
Jinku Shaw    Stanley Garner  Manager    Sales                             1
John Smith                    President  Finance     Doctorate             4

8 rows selected.

SQL>
```

Figure 7-17 Joining them all.

(actually five, because EMPLOYEE table is used twice) with two equijoins, two outer joins, one nonequijoin, and one self-join. The result contains everything we need to know about employees!

IN A NUTSHELL . . .

- Two tables can be joined with a common column. Usually, the common columns are a foreign key in one table and the primary key in the other table that is referenced.
- If a join condition is not used in a multitable query, it results in a Cartesian product.
- Four types of joins in Oracle are equijoin, nonequijoin, outer join, and self-join.
- It is possible to join more than two tables in a database. You need $n - 1$ conditions to join n tables.
- Table aliases are used in a query to avoid typing long table names. The table aliases are known only in the query where they are defined.
- An additional condition is used along with the join condition to filter out some rows.
- A nonequijoin has a join condition that does not use the equality (=) operator.
- An outer join is achieved by using the (+) operator on the deficient table's side in the join condition. In other words, the (+) operator is used on the side that generates null values. The outer join also selects rows without a matching row in another table involved in the join operation.
- A self-join joins a table with self. This operation uses two copies of same table.
- The set operators UNION, UNION ALL, INTERSECT, and MINUS are used to connect output from two individual SELECT queries. Both query outputs must return the same number of columns with similar domains.

EXERCISE QUESTIONS

True/False:

1. You always need at least two different tables for a join.
2. If a table has three rows and another table has four rows, their product will contain seven rows.
3. The common column in two tables must have same name to join them.
4. A table alias is known in the query in which it is created.
5. In an outer join, the (+) operator can be used on any one side of the equality (=) sign, but it cannot be on both sides of a join condition.
6. The set operator UNION does not repeat duplicate rows, but UNION ALL does.
7. TableA MINUS TableB is always the same as TableB MINUS TableA.
8. A self-join can be performed on any table, because all you need is one table.

9. An outer join usually returns more information than an equijoin on the same tables.
10. Two tables must have established foreign key–primary key relationship to perform a nonequijoin on them.

Define the Following Terms, and Give One Example of Each:

1. Equijoin.
2. Outer join.
3. Self-join.
4. Minus.
5. Cartesian product.

Answer the Following Questions:

1. What is the use of the set operator INTERSECT?
2. In which situations would you use a self-join?
3. When is it appropriate to use an outer join?
4. When would you use a MINUS operation?
5. How would you join five tables in a database?

LAB ACTIVITY

1. Use the N2 Corporation database tables to design the following queries.
   (Use the spooling method to capture all queries and results in the CHAP7SP1.LST file.)
   - **a.** Display all employee names and their department names.
   - **b.** Find name of the supervisor for employee number 433.
   - **c.** Find all employees' full names (lastname, firstname format) with salary and their supervisor's name with salary.
   - **d.** Find each employee's salary information and level based on the salary.
   - **e.** Display each employee's name, department name, position description, and qualification description. Which employee is missing? Why?
   - **f.** Find all employees in the sales department.
   - **g.** Display employee names and dependent information using an outer join.
   - **h.** Find the names of employees and number of years worked along with their department names in descending order by number of years worked.
   - **i.** Who works in the same department in which John Smith works?
2. Use the IU College database tables to design the following queries.
   (Use the spooling method to capture all queries and results in the CHAP7SP2.LST file)
   - **a.** Display a student's full name along with his or her major's description.
   - **b.** Get the names of students who received a final grade of F in Winter 2003.
   - **c.** Display student names, their faculty advisor names, and faculty office location.
   - **d.** Get Spring 2003 course sections with the faculty member assigned to teach the class. Include course sections without any faculty assigned to them.
   - **e.** Display course titles along with their prerequisite names. Display courses without prerequisite as well.

8

Subqueries: Nested Queries

IN THIS CHAPTER . . .

- Subqueries or nested queries are introduced for data retrieval.
- Various operators, such as ANY, SOME, and ALL, are introduced.
- Subqueries are used with other data management and data definition language statements.
- Multiple INSERT statements, such as INSERT ALL and INSERT FIRST, are used.
- A new MERGE statement is introduced.
- Correlated queries are introduced with EXISTS and NOT EXISTS operators.

In Chapter 5, you learned data retrieval techniques to obtain data from a single table with the SELECT query. You also learned the use of various clauses and functions used in SELECT statements in Chapter 6. In Chapter 7, you learned to create queries where data are retrieved from more than one table or with more than one query. Sometimes, you can accomplish tasks by joining two or more tables, by joining a table to itself, or by using the output from one query as data in another. The queries can be nested to create a table based on an existing table, to populate a table with rows from another table, or to perform updates and deletions.

SUBQUERY

Subqueries are also known as nested queries. A subquery is usually a SELECT query within one of the clauses in another SELECT query. Very powerful queries can be designed by using simple subqueries. A subquery is very useful when a query based on a table depends on the data in that subquery itself. The subquery can be used within a WHERE, HAVING, or FROM clause of another SELECT query. Subqueries are of two types:

1. **Single-row subquery:** a subquery that returns only one row of data; it is also known as a scalar subquery.
2. **Multiple-row subquery:** a subquery that returns more than one row of data.

Single-Row Subquery

The general syntax is

```
SELECT columnlist
FROM tablename
WHERE columnname operator
(SELECT columnnames
FROM tablename
WHERE condition);
```

There are certain rules you have to follow while creating a subquery:

- The subquery must be enclosed within a pair of parentheses.
- The subquery returns one column from one row in most cases. If no row is returned, the value is null. If more than one row is returned, an error occurs.
- The ORDER BY clause cannot be used in a subquery.
- The subquery is used on the right side of the condition.
- Relational operators (=, <> (or !=), >, >=, <, and <=) are used in the outer query's condition.

When a statement is written with a subquery, the inner query (subquery) is executed first. The inner query returns a value or a set of values to the outer query. Next, the outer query is executed with the result from the inner query.

In Figure 8-1, you see a subquery based on two tables. It returns a result similar to the one in Figure 7-5. The subquery example is substituted for a join condition and an additional condition of Figure 7-5. The inner query finds the DeptId 10 from the DEPT table based on DeptName = 'FINANCE'. The result is used in the condition of the outer query, which returns employees in DeptId = 10.

A subquery can also be based on only one table. Suppose you want to find the names of employees who make more salary than employee Dev (EmployeeId

```
SQL> SELECT Lname, Fname, Salary, DeptId
  2  FROM employee
  3  WHERE DeptId =
  4        (SELECT DeptId
  5        FROM dept
  6        WHERE UPPER (DeptName) = 'FINANCE');

LNAME           FNAME           SALARY     DEPTID
--------------- --------------- ---------- ----------
Smith           John                265000         10
Roberts         Sandi                75000         10
Chen            Sunny                35000         10

SQL>
```

Figure 8-1 Subquery using two tables.

= 543). You will find the salary for the employee in the inner query first and then use it in the outer query. For example, in Figure 8-2, the inner query finds the salary for employee Dev, and the outer query compares it with the salary received by other employees with the > = operator. Operator > = is used instead of > to include employee Dev in the output as well.

Similarly, we can write queries to tackle problems like finding employees who work in the same department in which John Smith works or employees who do not

```
SQL> SELECT Lname, Fname, Salary, DeptId
  2  FROM employee
  3  WHERE Salary >=
  4        (SELECT Salary
  5        FROM employee
  6        WHERE UPPER(Lname) = 'DEV');

LNAME           FNAME           SALARY     DEPTID
--------------- --------------- ---------- ----------
Smith           John                265000         10
Houston         Larry               150000         40
Dev             Derek                80000         20

SQL>
```

Figure 8-2 Subquery using one table.

work in the same department. Figure 8-3 shows all employees who do not work in employee John Smith's department. For simplicity, the inner query does not contain this multiple condition to check for the last name as well as the first name:

```
WHERE UPPER(Lname) = 'SMITH' AND UPPPER(Fname) = 'JOHN';
```

```
SQL> SELECT Lname, Fname, Salary, DeptId
  2  FROM employee
  3  WHERE DeptId !=
  4        (SELECT DeptId
  5        FROM employee
  6        WHERE UPPER(Lname) = 'SMITH')
  7  /

LNAME           FNAME           SALARY     DEPTID
--------------- --------------- ---------- ----------
Houston         Larry           150000     40
McCall          Alex            66500      20
Dev             Derek           80000      20
Shaw            Jinku           24500      30
Garner          Stanley         45000      30

5 rows selected.

SQL>
```

Figure 8-3 Subquery with ! = operator.

A subquery may be nested to more levels. In a multilevel subquery, the innermost query executes first, and then the execution propagates outward. In a three-level subquery, the output from the innermost query is used by the query nesting it, and the output of that query is used by the outmost query that encloses it.

In the Indo–US (IU) College database, each student has an assigned faculty advisor. The faculty advisor has an office in one of the college buildings. Suppose a student wants to find out location of his or her advisor's office. The result can be obtained by joining the STUDENT, FACULTY, and LOCATION tables. Alternately, a three-level subquery can render the same result as shown in Figure 8-4. Student Brian Lee wants to see his faculty advisor during office hours. First, the innermost query finds out the FacultyId of the advisor from the STUDENT table, which is used by enclosing a query to find the RoomId from the FACULTY table, which in turn is used by the outermost query to find the Building and RoomNo from the LOCATION table.

Creating a Table Using a Subquery You can create a table by using a nested SELECT query. The query will create a new table and populate it with the rows selected from the other table. The general syntax is

```
CREATE TABLE tablename
    AS
    SELECT-query;
```

When a new table is created with a subquery or a nested query, the primary key constraint is not transferred to the new table from the existing table. The NOT NULL is the only type of constraint that gets transferred to the new table. Other constraints can be added to the new table with the ALTER TABLE statement.

In Figure 8-5, new table TEMP is created based on the EMPLOYEE table. The SELECT query selects two employees belonging to Department 20 and adds them to

```
SQL> SELECT Building, RoomNo
  2 FROM location
  3 WHERE RoomId =
  4          (SELECT RoomId
  5          FROM faculty
  6          WHERE FacultyId =
  7     (SELECT FacultyId
  8     FROM student
  9     WHERE UPPER(Last) = 'LEE'
 10     AND UPPER(First) = 'BRIAN'));

BUILDIN     ROO
-------     ---
Gandhi      103

1 row selected.

SQL>
```

Figure 8-4 Subquery to three levels.

```
SQL> CREATE TABLE temp
 2 AS
 3 SELECT EmployeeId, Lname, Fname, Salary
 4 FROM employee
 5 WHERE DeptId = 20;

Table created.

SQL> DESCRIBE temp

Name                            Null?    Type
------------------------------- -------- ---------------
EMPLOYEEID                               NUMBER(3)
LNAME                           NOT NULL VARCHAR2(15)
FNAME                           NOT NULL VARCHAR2(15)
SALARY                                   NUMBER(6)

SQL> SELECT * FROM temp;

EMPLOYEEID LNAME           FNAME           SALARY
---------- --------------- --------------- ----------
       433 McCall          Alex                66500
       543 Dev             Derek               80000

2 rows selected.

SQL>
```

Figure 8-5 CREATE using a subquery.

the newly created table. The new table contains only four attributes, as selected by the inner query. The figure also shows use of the new TEMP table with the DESCRIBE and SELECT statements. A user may add more constraints to this newly created table with the ALTER TABLE statement.

INSERT Using a Subquery An existing table can be populated with a subquery. The table must already exist to insert rows into it. INSERT with a subquery does not create a new table. The general syntax is

```
INSERT INTO tablename [(column list)]
SELECT columnnames FROM tablename WHERE condition;
```

The INSERT statement does not use the VALUES clause. The subquery replaces the VALUES clause and provides values for the new rows. The column list in INSERT is optional. The column list can be used from the nested SELECT query.

In the example of Figure 8-6, all employees in Department 10 of the EMPLOYEE table are selected, and the TEMP table is populated with those employees only. The TEMP table already had two rows; three more rows are added with INSERT. Now, the table contains five rows in all. The TEMP table contains four columns, but values are inserted into three columns only. Employee salaries are not included for employees from Department 10.

```
SQL> INSERT INTO temp (EmployeeId, Lname, Fname)
  2  SELECT EmployeeId, Lname, Fname
  3  FROM employee
  4  WHERE DeptId = 10
  5  /

3 rows created.

SQL> SELECT *
  2  FROM temp;

EMPLOYEEID LNAME           FNAME           SALARY
---------- --------------- --------------- ----------
       433 McCall          Alex             66500
       543 Dev             Derek            80000
       111 Smith           John
       123 Roberts         Sandi
       222 Chen            Sunny

5 rows selected.

SQL>
```

Figure 8-6 INSERT using a subquery.

Inserting into Multiple Tables (Oracle9i Onward) In most cases, you use the INSERT statement to add a row into a table. Oracle9i has new feature that allows you to enter rows into multiple tables simultaneously. This feature is useful for transferring, archiving, and denormalizing data. The multiple INSERT statement is faster and more flexible than many one-row, simple INSERT statements. Two multiple INSERT statements are:

1. INSERT ALL (conditional and unconditional)
2. INSERT FIRST

Unconditional INSERT ALL In this example, rows are selected from the EMPLOYEE table and inserted into two existing tables, EMPLOYEE_SALARY and EMPLOYEE_DEPT. Notice that these two tables contain different columns. The inner SELECT statement retrieves rows from the EMPLOYEE table, and those rows are inserted into two tables. The INSERT statement does not have any conditions:

```
INSERT ALL
INTO employee_salary
            VALUES(EmployeeId, Lname, Fname, Salary, Commission)
INTO employee_dept
            VALUES(EmployeeId, Lname, Fname, DeptId, Supervisor)
SELECT EmployeeId, Lname, Fname, Salary, Commission, DeptId, Supervisor
      FROM employee WHERE Salary > 50000 OR DeptId <> 40;
```

Conditional INSERT ALL In this example, the rows are inserted into tables based on their individual conditions. The WHEN ... THEN clause is used with different conditions for inserting rows into different tables. Rows are inserted into the EMPLOYEE_SALARY table if salary is higher than 50000. Rows are inserted into the EMPLOYEE_DEPT table if DepartId is not equal to 40:

```
INSERT ALL
      WHEN Salary > 50000 THEN INTO employee_salary
            VALUES(EmployeeId, Lname, Fname, Salary, Commission)
      WHEN DeptId <> 40 THEN INTO employee_dept
            VALUES(EmployeeId, Lname, Fname, DeptId, Supervisor)
SELECT EmployeeId, Lname, Fname, Salary, Commission, DeptId, Supervisor
      FROM employee WHERE Salary > 50000 OR DeptId <> 40;
```

Conditional INSERT FIRST The conditional INSERT FIRST statement has the same syntax as the conditional INSERT ALL statement except for the key word FIRST in place of ALL. The working of the statement, however, is different. It tests conditions just like the INSERT ALL statement, but if a row satisfies both conditions Salary > 50000 and DeptId<>40, it will be inserted into the first table only, because the first condition of the WHEN clause is satisfied.

UPDATE Using a Subquery Another use of a subquery is in updating data. If an employee is leaving and his or her position, supervisor, and salary information are to be given to another existing employee, UPDATE can be performed with a subquery. The general syntax is

```
UPDATE tablename
SET columnname = value or expression
WHERE columnname operator
      (SELECT  subquery);
```

For example, the NamanNavan (N2) Corporation is very pleased with the performance of the entire FINANCE team and decides to raise their salary by 10% (see Fig. 8.7). The inner query supplies DeptId based on the department's name, which the outer query uses in its WHERE clause.

```
UPDATE employee
SET Salary  =  Salary * 1.10
WHERE DeptId =
          (SELECT DeptId
           FROM dept
           WHERE UPPER(DeptName)  =  'FINANCE');

3 rows updated.

SQL>
```

Figure 8-7 UPDATE using a subquery.

An alternate syntax is

```
UPDATE tablename
SET (columnnames) =
     (SELECT subquery)
[WHERE condition];
```

The optional WHERE clause is part of the outer UPDATE statement in the given syntax. The inner SELECT query may contain another WHERE clause.

In the example of Figure 8-8, Jinku Shaw (EmployeeId = 200) gets the position, supervisor, and DeptId of employee Stanley Garner (EmployeeId = 135). The inner query returns three values, which are assumed by three columns in the outer query. You must be careful with the order of columns in both the inner and outer queries. The order of columns must be same in both queries.

DELETE Using a Subquery A row or rows from a table can be deleted based on a value returned by a subquery. The general syntax is

```
DELETE FROM tablename
WHERE columnname =
          (SELECT subquery);
```

```
SQL> UPDATE employee
  2    SET (PositionId, Supervisor, DeptId) =
  3      (SELECT PositionId, Supervisor, DeptId
  4      FROM employee WHERE EmployeeId = 135)
  5  WHERE EmployeeId = 200;

1 row updated.

SQL>
```

Figure 8-8 UPDATE using a subquery.

For example, if a corporation decides to close the Accounting Department in Monroe and all employees in the department are to be removed from the database, you will use a DELETE statement with a subquery. You must remember that to remove a department from the DEPT table, you must remove all employees from that department first. Failure to do so will result in constraint violation because of the existence of child rows in the EMPLOYEE table. In Figure 8-9, we have tried to remove employees from the Accounting Department, but no rows have been deleted. There is no constraint violation here, but the EMPLOYEE table does not contain any employees in the Accounting Department (DeptId = 80).

```
SQL> DELETE FROM employee
  2  WHERE DeptId =
  3    (SELECT DeptId FROM dept
  4    WHERE UPPER (DeptName) = 'ACCOUNTING');

0 rows deleted.

SQL>
```

Figure 8-9 DELETE using a subquery.

Multiple-Row SubQuery

A multiple-row subquery returns more than one row. The operators used in single-row subqueries (=, <>, >, >=, < and <=) cannot be used with multiple-row subqueries. Figure 8-10 shows special operators used with multiple-row subqueries.

Operator	Use
IN	Equal to any of the values in a list.
ALL	Compare the given value to every value returned by the subquery.
ANY or **SOME**	Compare the given value to each value returned by the subquery.

Figure 8-10 Multiple-row subquery operators.

Let us look at examples of subqueries returning more than one row. You are already familiar with the IN operator, which is an alternative to multiple OR conditions. The IN operator looks for at least one match from the list of values provided. In Figure 8-11, the inner query returns two values (111, 123) for faculty members in department 1. You cannot use the = operator here, but the IN operator is more appropriate. Use of the = operator here will result in an error. The outer query returns students who have faculty 111 or 123 as their advisor.

```
SQL> SELECT StudentId, Last, First, FacultyId
  2  FROM student
  3  WHERE FacultyId IN
  4        (SELECT FacultyId
  5        FROM faculty WHERE DeptId = 1);

STUDE  LAST            FIRST           FACULTYID
-----  --------------- --------------- ----------
00100  Diaz            Jose                   123
00102  Patel           Rajesh                 111

SQL>
```

Figure 8-11 Subquery with IN operator.

The ANY operator can be used in combination with other relational operators. It serves like an OR operator, because it looks for a match to any one value! For example:

- <ANY means less than the maximum value in the list.
- =ANY means equal to any value in the list (similar to IN).
- >ANY means higher than the minimum value in the list.

The inner query in Figure 8-12 returns four values (150000, 75000, 80000, and 45000). The >ANY operator checks for values larger than the minimum salary of

```
SQL> SELECT EmployeeId, Lname, Fname, Salary
  2  FROM employee
  3  WHERE Salary >ANY
  4       (SELECT Salary FROM employee WHERE PositionId = 2)
  5  AND PositionId <> 2;

EMPLOYEEID  LNAME           FNAME           SALARY
----------  --------------- --------------- ----------
       111  Smith           John            265000
       433  McCall          Alex             66500

SQL>
```

Figure 8-12 Subquery with >ANY operator.

45000 in the list. The outer query also checks for PositionId not equal to 2. Rows with EmployeeId 111 and 433 are selected. You may use the operator SOME in place of ANY to achieve the same result.

The ALL operator can also be used with relational operators. It serves like an AND operator, because it looks for a match to all values! For example:

- >ALL means more than the maximum value.
- <ALL means less than the minimum value.
- = ALL is meaningless, because no value can be equal to all values in a list.

In Figure 8-13, the inner query returns four averages, one value per department (125000, 73250, 34750, and 150000). The <ALL operator means the employee with a salary less than the minimum in the list of values. From the original table in Chapter 3, only one row—and EmployeeId 200 and salary of 24500—is picked. If you change condition to =ALL, you will see the "no rows selected" message.

```
SQL> SELECT EmployeeId, Lname, Fname, Salary
  2  FROM employee
  3  WHERE Salary <ALL
  4       (SELECT AVG (Salary) FROM employee GROUP BY DeptId);

EMPLOYEEID  LNAME           FNAME           SALARY
----------  --------------- --------------- ----------
       200  Shaw            Jinku                24500
SQL>
```

Figure 8-13 Subquery with <ALL operator.

Try the query in Figure 8-12 with <ALL, >ALL, <ANY, and =ANY operators, and check out the results.

TOP-N ANALYSIS

Top-N queries are used to sort rows in a table and then to find the first-N largest or first-N smallest values. For example, you want to find the bottom five salaries in a company, the top three room capacities, or the last 10 employees hired by a company, you would use a Top-N query.

The Top-N query uses an ORDER BY clause to sort rows in ascending or descending order. The sorted rows are numbered with a pseudocolumn named ROWNUM. If the rows are sorted in ascending order by the Top-N column, the smallest value of the Top-N column is at the top of the list. The largest value of the Top-N column is at the top of the list if the rows are sorted in descending order by the Top-N column. You can display the required number of rows based on the ROWNUM with < or <= operators. The > and >= operators are not allowed with a ROWNUM pseudocolumn.

In Figure 8-14, Capacity is the Top-N column. The rows in the LOCATION table are sorted in descending order to get largest capacity at the top of the list. The condition ROWNUM <= 4 selects row numbers 1 through 4, the top-four capacities. The inner SELECT statement in the FROM clause is used as the data source for the outer SELECT statement. Such a subquery is known as an **inline view.** The inline view is a subquery that can be given an alias name for use in an SQL statement, just like a table alias. An inline view is not stored as an object like the other views created with a CREATE VIEW statement (see Chapter 9). A subquery may not use the ORDER BY clause, but an inline view may.

```
SQL> SELECT rownum, Building, RoomNo, Capacity
  2  FROM (SELECT Building, RoomNo, Capacity
  3    FROM location
  4    ORDER BY Capacity DESC)
  5  WHERE ROWNUM <= 4;

   ROWNUM   BUILDIN   ROO     CAPACITY
---------- -------- ---  ----------
         1  Kennedy  204           50
         2  Nehru    301           50
         3  Nehru    309           45
         4  Kennedy  206           40

SQL>
```

Figure 8-14 Top-4 capacities.

In Figure 8-15, SALARY is the Top-N column. The rows in the EMPLOYEE table are sorted in ascending order, bringing the lowest salaries to the top. The condition ROWNUM <= 3 retrieves the three lowest-salaried employees.

```
SQL> SELECT ROWNUM, Lname, Fname, Salary
  2  FROM (SELECT Lname, Fname, Salary
  3    FROM employee
  4    ORDER BY Salary)
  5  WHERE ROWNUM <= 3;

   ROWNUM   LNAME           FNAME           SALARY
---------- --------------- --------------- ----------
         1  Shaw            Jinku            24500
         2  Chen            Sunny            35000
         3  Garner          Stanley          45000

SQL>
```

Figure 8-15 Bottom-three salaries.

Important Note about Top-N Analysis

Top-N analysis is explained here in a way that is consistent with the Oracle online training documentation. You may not use the ORDER BY clause with an inner subquery. If you cannot perform this analysis with the inline view, which contains an ORDER BY clause in the inner query, create view with the GROUP BY clause in the SELECT query. The GROUP BY clause contains an implicit ORDER BY operation in the ascending order. You will be able to select the Top-N rows from the view based on the ROWNUM pseudocolumn. The limitation is that only < and <= operators are allowed with the pseudocolumn ROWNUM. As the implied sort is in the ascending order, the lowest values get moved to the top, and you will be able to get only the bottom-N values. The Top-N analysis has worked without any problems in Oracle9i, but I did face problems in Oracle8. The Oracle views are covered in the next chapter.

MERGE STATEMENT

You can use the MERGE statement to perform INSERT and UPDATE operations together. This operation is very useful in data warehousing. In Figure 8-16, WORKER is the target table. The STUDENT and WORKER tables are merged based on WorkerId and StudentId columns. If rows from two tables match, then the Last and First columns are updated in the WORKER table (WorkerId 00103 and 00105). If rows from the STUDENT table do not have a match in the WORKER table, those rows are inserted in the WORKER table (WorkerId 00102, 00100, 00104, and 00101).

CORRELATED SUBQUERY

Correlated subqueries are different from the other subqueries explained earlier in this chapter. In a correlated subquery, the inner (nested) query can reference columns from the outer query. The inner query is executed once for each row in the outer query. In other subqueries, the inner query was executed only once. It is a complex—but very powerful—feature. The example in Figure 8-17 performs a correlated subquery.

First, Oracle selects a row from the outer query. Then, it finds the value of the correlated column(s). Next, it executes the inner query for each row of the outer query, sends the result of the inner query to the outer query, and executes the outer query. If the row satisfies the condition, it outputs the row. Then, it selects the next row from outer query and repeats the same procedure.

In Figure 8-17, each employee's salary is matched with the maximum salary of his or her department (outer.DeptId). The result is employees with the maximum salary in their respective departments. With a simple GROUP BY clause, you can find the maximum salary for each department, but you cannot find employees making those salaries. A correlated subquery, however, enables you to find that information.

```
SQL> SELECT * FROM worker;

WORKE   LAST                 FIRST
-----   ---------------      ---------------
00110
00111
00103
00113
00105
00107

6 rows selected.

SQL> MERGE INTO worker w
  2  USING  (SELECT StudentId, Last, First
  3             FROM  student) s
  4  ON (s.StudentId = w.WorkerId)
  5  WHEN MATCHED THEN
  6             UPDATE SET w.Last = s.Last,
  7                        w.First = s.First
  8  WHEN NOT MATCHED THEN
  9         INSERT (w.WorkerId, w.Last, W.First)
 10  VALUES (s.StudentId, s.Last, s.First)
 11  /

6 rows merged.

SQL> SELECT * FROM worker;

WORKE   LAST                 FIRST
-----   ---------------      ---------------
00110
00111
00103   Rickles              Deborah
00113
00105   Khan                 Amir
00107
00102   Patel                Rajesh
00100   Diaz                 Jose
00104   Lee                  Brian
00101   Tyler                Mickey

10 rows selected.
```

Figure 8-16 MERGE statement in action.

EXISTS and NOT EXISTS Operators

The EXISTS and NOT EXISTS operators are used with correlated queries. The EXISTS operator checks if the inner query returns at least one row. It returns

```
SQL> SELECT EmployeeId, Salary, DeptId
  2  FROM employee outer
  3  WHERE Salary =
  4     (SELECT MAX(Salary)
  5     FROM employee
  6     WHERE DeptId = outer.DeptId
  7     GROUP BY DeptId);

EMPLOYEEID     SALARY     DEPTID
----------  ----------  ----------
       111      265000          10
       246      150000          40
       543       80000          20
       135       45000          30
SQL>
```

Figure 8-17 Correlated subquery.

TRUE or FALSE. The column names in the inner query have no significance. You can use any single character literal, such as *A, 1, z*, and so on.

In Figure 8-18, the EXISTS operator checks if a row from the FACULTY table (outer query) is returned at least once by the STUDENT table (inner query). The result includes all faculty members who are in the STUDENT table.

```
SQL> SELECT FacultyId, Name
  2  FROM faculty outer
  3  WHERE EXISTS
  4     (SELECT '1'
  5     FROM student
  6     WHERE FacultyId = outer.FacultyId);

FACULTYID   NAME
----------  ---------------
       111  Jones
       222  Williams
       123  Mobley
       345  Sen
       555  Chang
SQL>
```

Figure 8-18 EXISTS operator.

In Figure 8-19, the NOT EXISTS operator is the opposite of the EXISTS operator. It checks if the inner query does not return a row. In other words, it returns faculty members who are not in the STUDENT table.

```
SQL> SELECT FacultyId, Name
  2  FROM faculty outer
  3  WHERE NOT EXISTS
  4       (SELECT '1'
  5       FROM student
  6       WHERE FacultyId = outer.FacultyId);

 FACULTYID   NAME
----------  ---------------
       235   Vajpayee
       444   Rivera
       333   Collins

SQL>
```

Figure 8-19 NOT EXISTS operator.

IN A NUTSHELL . . .

- Subqueries are also known as nested queries. In a nested query, the inner query is executed first. The output from the inner query is then used by the outer query.
- A single-row subquery returns one row of data, whereas a multiple-row subquery returns more than one row of data.
- A single-row subquery may use relational operators, but multiple-row subquery may use IN, ANY (or SOME), and ALL operators in conjunction with relational operators.
- The inner query in the subquery is enclosed within parentheses, and it cannot use the ORDER BY clause.
- A subquery may be nested to multiple levels.
- A subquery can be used with the SELECT statement. It can also be used with CREATE to create a table and populate it with rows in another table.
- A subquery is also used with INSERT, DELETE, and UPDATE data manipulation queries.
- The multiple-row subqueries use the special operators IN, ALL, SOME, and ANY.
- Top-N analysis is used to sort rows in ascending or descending order and then to find the Top-N rows for the N highest or lowest values. The inline view is used for a Top-N analysis.
- A MERGE statement performs the INSERT and UPDATE operation together.
- In a correlated subquery, the inner query references a column in the outer query.
- EXISTS and NOT EXISTS are special operators used in correlated subqueries.

EXERCISE QUESTIONS

True/False:

1. An inner subquery may not use the ORDER BY clause.
2. The inner subquery does not have to be a SELECT statement.
3. A subquery may not be nested to more than three levels.
4. Operators ANY and SOME are the same.
5. The = ALL operator does not return any rows in most cases.
6. Top-N analysis uses ROWID attribute for row numbers.
7. An inline view may use the ORDER BY clause.
8. In correlated subqueries, the inner query can reference a column from the outer query.
9. = ANY means IN.
10. >ALL means greater than the minimum value.
11. A table can be created with a subquery and an INSERT statement.
12. A table can be created based on another table with a subquery.
13. When a table is created with a subquery, it inherits all constraints from the original table.

Define the Following Terms, and Give One Example of Each:

1. Single-row subquery.
2. Multiple-row subquery.
3. Inline view.
4. Top-N query.
5. Correlated subquery.

Answer the Following Questions:

1. State the various uses of a subquery.
2. In which situations would you use a three-level subquery?
3. What constraints are transferred to the newly created table with a subquery?
4. How does a Top-N query work?
5. What is the use of the INSERT ALL and INSERT FIRST statements?
6. Why is the MERGE statement useful?
7. A subquery returns three values: 35000, 45000, and 55000. The outer query has a condition that tests a value of 40000 against these values. If the =ANY, >ANY, <ALL, >ALL, or <>ALL operator is used, when will the value 40000 satisfy the condition? Use each operator separately.

LAB ACTIVITY

1. Use the N2 Corporation database tables to design the following subqueries. (Use the spooling method to capture all queries and results in the CHAP8SP1.LST file.)
   (a) Display employee Jinku Shaw's department name.
   (b) Find the name of the supervisor for employee number 433.

(c) Who has the same qualification as Stanley Garner?

(d) Which department has more employees than Department 20?

(e) Which employees have been working in the company longer than Larry Houston?

(f) Find all employees in the Sales Department by using a nested query.

(g) Create a new table, EMP30, and populate it with employees in Department 30 by using an existing table and a subquery. Use EmployeeId, Lname, Fname, HireDate, and Salary columns.

(h) Add more rows to the EMP30 table with employees in Department 40. Do not transfer the employee's salary.

(i) Update the salary of newly transferred employees from the EMPLOYEE table to the EMP30 table with a MERGE statement, and INSERT employees who are not in the EMP30 table.

(j) Find employees with the minimum salary in their own department with the use of a correlated subquery.

(k) Use a multiple-level subquery to display dependent information for employees who belong to the FINANCE department.

(l) Use set operator and subquery to find employees who do not have any dependents.

(m) Write a subquery that finds the average salary by each department. Check to find if employee 543's salary satisfies the =ANY, <ANY, >ANY, <ALL, or >ALL condition against those departmental average salaries.

2. Use the IU College database tables to design the following subqueries. (Use the spooling method to capture all queries and results in the CHAP8SP2.LST file)

(a) Display student Jose Diaz's faculty advisor's name and phone number.

(b) Find the rooms with the bottom-two capacities. Do not include office rooms.

(c) Find the Spring 2003 course sections with the top-three maximum count numbers.

(d) Find all information regarding classrooms (RoomType = 'C').

(e) Create a new table, SP03SECT, for Spring 2003 semester course sections by using a subquery. Include CourseId, Section, FacultyId, and RoomId columns only.

(f) Delete rows from the SP03SECT table for faculty member Mobley.

(g) Find faculty members who do not teach any course in the Spring 2003 semester. Use a correlated subquery with a NOT EXISTS operator on the SP03SECT table.

9

Advanced Features: Objects, Transactions, and Data Control

IN THIS CHAPTER . . .

- You will learn about various Oracle objects.
- You will use syntax to create, use, modify, and remove views, sequences, synonyms, and indexes.
- Advantages of transaction control are discussed.
- Users, roles, and privileges for data control are covered.

You have learned to create, modify, remove, use, and manipulate an Oracle object called a table. In this chapter, you will learn about other objects, such as the view, sequence, synonym, and index. Some of these objects are based on underlying Oracle tables, and some are independent objects. In this chapter, you will also learn about transactions and their advantages. You will be able to grant and revoke privileges regarding your own objects to other users.

VIEWS

A view is an Oracle object that gives the user a logical view of data from an underlying table or tables. You can restrict what users can view by allowing them to see only a few columns from a table. When a view is created from more than one table, the user can view data from the view without using join conditions and complex

conditions. The application programs can access data with data independence. The same data can be viewed differently with different views. Views also hide the names of the underlying tables, so the user does not know where the data came from. In short, a view is a logical representation of a subset of data from one or more tables. A view is stored as a SELECT statement in the Data Dictionary. There are two types of views: simple and complex.

Figure 9-1 shows the difference between simple and complex views. A simple view is based on one table. It does not contain group functions or grouped data, and data manipulation is always possible through it. On the other hand, a complex view is based on one or more tables. It may contain group functions and/or grouped data, and data manipulation is not always possible through it.

Simple View	Complex View
It is based on one table.	It is based on one or more tables.
It does not contain group functions.	It may contain group functions.
It does not contain grouped data.	It may contain grouped data.
Data manipulation is always possible.	Data manipulation is not always possible.

Figure 9-1 Types of views.

Creating a View

The general syntax is

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW viewname
[column aliases]
AS SELECT-subquery
[WITH CHECK OPTION [CONSTRAINT constraintname]]
[WITH READ ONLY];
```

A view is created with a SELECT subquery. The subquery cannot use an ORDER BY clause, but a view can.

In the syntax, OR REPLACE replaces an existing view with the same name, if it already exists. The FORCE option creates a view even if the underlying table does not yet exist. The default is NOFORCE, which does not create a view if the underlying table does not exist. The column aliases are used for the columns selected by the subquery. The number of aliases must match the number of columns selected by the subquery. The SELECT subquery can use all clauses except the ORDER BY clause. The WITH CHECK OPTION applies to the WHERE clause condition in the subquery. It allows insertion and updating of rows based on the condition that satisfies the view. The CHECK OPTION can also be given an optional constraint

name. The WITH READ ONLY option is to make sure that the data in the underlying table are not changed through the view.

Figure 9-2 shows creation of a simple view. View STU500_VU is based on the STUDENT table for students with MajorId = 500. The column aliases are given for all

```
SQL> CREATE VIEW stud500_vu
  2  (StuId, LastName, FirstName, Advisor, MajNum)
  3  AS SELECT StudentId, Last, First, FacultyId, MajorId
  4      FROM student
  5      WHERE MajorId = 500
  6  WITH CHECK OPTION;

View created.

SQL>
```

Figure 9-2 Creating a simple view.

four columns selected by the subquery. The user with access to the view can use it like any other table. The user does not even have to know the existence of the STUDENT table and other columns in it. The user only sees what you let him or her see! Now, Figure 9-3 shows use of the newly created view like a table with DESCRIBE and SELECT.

In Figure 9-4, you will see a complex view that is created from two tables, EMPLOYEE and DEPT. It uses group functions and derived data. It is not possible to

```
SQL> set linesize 50
SQL> DESCRIBE stu500_vu
Name                      Null?    Type
------------------------  -------- ------------
STUID                     NOT NULL CHAR(5)
LASTNAME                  NOT NULL VARCHAR2(15)
FIRSTNAME                 NOT NULL VARCHAR2(15)
ADVISOR                            NUMBER(3)
MAJNUM                             NUMBER(3)

SQL> set linesize 100
SQL> SELECT *
  2  FROM stu500_vu;

STUID LASTNAME        FIRSTNAME           ADVISOR    MAJNUM
----- --------------- --------------- ---------- ----------
00101 Tyler           Mickey                 555        500
00103 Rickles         Deborah                555        500

SQL>
```

Figure 9-3 Using a view like a table.

```
SQL> CREATE  OR  REPLACE FORCE VIEW dept_sal_vu
  2  AS
  3  SELECT d.DeptName DEPARTMENT, MIN(e.Salary) LOWEST,
  4         MAX(e.Salary) HIGHEST, AVG(e.Salary) AVERAGE
  5  FROM employee e, dept d
  6  WHERE e.DeptId = d. DeptId
  7  GROUP BY d.DeptName;

View created.

SQL> SELECT  *  FROM  dept_sal_vu;

DEPARTMENT        LOWEST     HIGHEST     AVERAGE
------------  ---------  ---------  ----------
Finance            35000     265000      125000
InfoSys            66500      80000       73250
Marketing         150000     150000      150000
Sales              24500      45000       34750

SQL>
```

Figure 9-4 Creating and using a complex view.

modify data through this view. Notice that the column aliases are given by the subquery, not by the outer CREATE VIEW statement. If a view by the name DEPT_SAL_VU already existed, this statement would have overwritten the previous view because of the CREATE OR REPLACE VIEW statement.

Other rules related to data manipulation on a view include:

- No data manipulation on derived columns, such as Salary/12 or Salary + NVL(Commission, 0).
- No data manipulation on the ROWNUM pseudocolumn.
- No insertion of a new row into the table if the base table contains columns with the NOT NULL constraint but are not selected by the view.
- No insertion of rows into the table if derived columns exist in the view.

Figure 9-2 has a simple view with a WITH CHECK OPTION clause. It applies to the WHERE condition in the subquery. The user cannot change the MajNum, because the view accesses rows with MajorId = 500 only. The UPDATE of MajNum to any other value will result in the following error message (see Fig. 9-5):

```
ORA-01402: view WITH CHECK OPTION where-clause violation.
```

Similarly, data manipulation on a view having a WITH READ ONLY clause results in an Oracle server error.

A user can list the names of views under his or her ownership by using

```
SELECT view_name FROM user_views;
```

```
SQL> update stu500_vu
  2 set MajNum = 600;
update stu500_vu
       *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation

SQL>
```

Figure 9-5 WITH CHECK OPTION where-clause violation.

Removing a View

A user who owns a view—or who has the privilege to remove it—can remove a view. The removal of a view does not affect data in the underlying table. When a view is removed, a "View dropped" message is displayed (see Fig. 9-6). The general syntax is

```
SQL> SELECT VIEW_NAME FROM USER_VIEWS;

VIEW_NAME
------------------------------
DEPTSALVU
DEPT_SAL_VU
STU500_VU

SQL> DROP VIEW deptsalvu;

View dropped.

SQL>
```

Figure 9-6 Listing and dropping view.

```
DROP VIEW viewname;
```

For example,

```
DROP VIEW stuvu500;
```

Altering a View

When you alter an underlying table, the view becomes invalid. You need to recompile that view to make it valid again. The ALTER VIEW statement is used for the recompilation of a view. For example,

```
ALTER VIEW deptsalvu COMPILE;
```

The same statement is used to check validity of a view in case the underlying table is dropped.

With Oracle9i, the ALTER VIEW statement lets you add constraints to a view. You can use clauses like MODIFY CONSTRAINT and DROP CONSTRAINT.

SEQUENCES

A sequence is an Oracle object that is used to generate a sequence of numbers. Many times, you create a table with a surrogate key column, such as StudentId, FacultyId, or EmployeeId. These columns have a numeric data type. Sequencing is a perfect solution for generating values for such numeric columns. These values can be unique or recycled again, depending on the column. If a sequence is used for a primary key column, however, the values must be unique! A sequence is not limited to the primary key columns but can be used on any numeric column. The general syntax is

```
CREATE SEQUENCE sequencename
  [INCREMENT BY n]
  [START WITH s]
  [MAXVALUE x | NOMAXVALUE]
  [MINVALUE m | NOMINVALUE]
  [CYCLE | NOCYCLE]
  [CACHE c | NOCACHE]
  [ORDER | NOORDER];
```

In this syntax, note the following:

INCREMENT BY *n*	The increment value for the number generation is *n*. The default increment is 1.
START WITH *s*	The starting value for the sequence is *s*. The default start value is 1.
MAXVALUE *x*	The maximum value for the generated number is *x*.
NOMAXVALUE	The sequence will keep generating until the maximum allowable value ($1 \cdot 10^{27}$) is generated in ascending order. NOMAXVALUE is -1 for a sequence in descending order (default).
MINVALUE *m*	The minimum value for the generated number is *m*.
NOMINVALUE	The minimum value is 1 for a sequence in ascending order and -10^{26} for a sequence in descending order (default).
CYCLE	The sequence continues generating after reaching the maximum or minimum value.
NOCYCLE	No more generation after maximum or minimum value is reached (default).
CACHE *c*	The Oracle server generates numbers in advance and

	stores them in a cache memory area for improved system performance. The default value for *c* is 20. If the user provides a number, the server will store that many numbers in the cache memory.
NOCACHE	The server does not store any sequence numbers in memory in advance.
ORDER	The numbers are generated in chronological order.
NOORDER	The numbers are not generated in chronological order.

Suppose you want to create a sequence to generate EmployeeId values in the EMPLOYEE table. The last EmployeeId in the table is 543. So, you want to start at 544, and the numbers will be incremented by 1 for every new employee (see Fig. 9-7).

```
SQL> CREATE SEQUENCE employee_employeeid_seq
  2  INCREMENT BY 1
  3  START WITH 544
  4   MAXVALUE 999
  5  NOCACHE;

Sequence created.

SQL>
```

Figure 9-7 Sequence.

```
SQL> CREATE SEQUENCE major_majorid_seq
  2  INCREMENT BY 10
  3  START WITH 610
  4  MAXVALUE 999
  5  NOCACHE;

Sequence created.

SQL> INSERT INTO major VALUES
  2  (major_majorid_seq.NEXTVAL, 'MS - Computer Science');

1 row created.

SQL>
```

Figure 9-8 Sequence—creation and use with NEXTVAL.

Let us create another sequence to generate MajorId numbers. If you view the MAJOR table, the last value used is 600, and each value is in steps of 100. In the future, you want to add numbers in steps of 10, starting with 610. The data type of the column is NUMBER(3), so the maximum allowable value should be 999 (see Fig. 9-8).

There are pseudocolumns named CURRVAL and NEXTVAL to reference sequence values. The NEXTVAL column returns the next available number in the sequence. The CURRVAL column gives the current sequence value. The NEXTVAL column is used with the sequence name (e.g. *sequencename.NEXTVAL*) to generate the next sequence number. When the new sequence number is generated, the current number is stored in the CURRVAL column. NEXTVAL must be used at least once to get the value from CURRVAL.

Because the sequence *major_majorid_seq* is used for the first time in Figure 9-8, the first value generated is the starting value defined for the sequence. The new major will get value 610. You can check it with the pseudocolumn CURRVAL with Oracle's dummy table DUAL. For example,

```
SELECT major_majorid_seq.CURRVAL FROM dual;
```

The number returned by CURRVAL is 610, which is the last value generated by NEXTVAL.

The *employee_employeeid_seq* sequence is created in Figure 9-7, but it is not used yet. If you try to find the current value, you will get an error message. Create a new sequence named *employee_employeeid_seq*, and then use the following statement to see the error message:

```
SELECT employee_EmployeeId_seq.CURRVAL FROM dual;
```

The query will result in an error message, because CURRVAL does not have a value yet and it is not defined.

Figure 9-9 uses the *dept_deptid_seq* sequence first to add a new department in the DEPT table with NEXTVAL. Then, a new row is added to the EMPLOYEE table, generating a new EmployeeId with employee_employeeid_seq.NEXTVAL

```
SQL> CREATE SEQUENCE dept_deptid_seq
  2  START WITH 90
  3  INCREMENT BY 1;

Sequence created.

SQL> INSERT INTO DEPT VALUES
  2   (dept_deptid_seq.NEXTVAL, 'IT', 'BRONX', 111);

1 row created.

SQL> INSERT INTO employee (EmployeeId,Lname,Fname,DeptId)
  2  VALUES (employee_employeeid_seq.NEXTVAL,
  3  'Viquez', 'Heillyn', dept_deptid_seq.CURRVAL);

1 row created.

SQL>
```

Figure 9-9 Sequence—NEXTVAL and CURRVAL.

and using the newly created department's value for the employee's DeptId with dept_deptid_seq.CURRVAL.

When you create a sequence and specify an increment value, you still may find gaps in the values generated. The gaps result from one of the following reasons:

- You generated sequence values in an INSERT statement, and the row was never written to the permanent database because of "rollback".
- You used the CACHE option, and the system crashed. So, the numbers brought to memory in advance are lost.
- A sequence is used in more than one table or on more than one column.
- Rows are deleted from a table.

```
SQL> SELECT sequence_name,   last_number,
  2       max_value, min_value, increment_by
  3  FROM user_sequences;

SEQUENCE_NAME                 LAST_NUMBER     MAX_VALUE   MIN_VALUE  INCREMENT_BY
----------------------------- ------------ ------------ ---------- ------------
DEPT_DEPTID_SEQ                        110 1.0000E+27            1            1
EMPLOYEE_EMPLOYEEID_SEQ                545          999          1            1
MAJOR_MAJORID_SEQ                      620          999          1           10

SQL>
```

Figure 9-10 USER_SEQUENCES.

You can get information about sequences you have created by using the query shown in Figure 9-10. The LAST_NUMBER column shows the next available number, and the other columns show values set at the time of creation of sequences.

Modifying a Sequence

You can modify a sequence if you own it or have the ALTER SEQUENCE privilege. The modified sequence does not affect past numbers, only numbers generated in the future. Modification of a sequence does not allow you to change the START WITH option. The sequence has to be removed and recreated to change the starting value. The maximum value cannot be set to a number less than the current number. You can change the INCREMENT BY, MAXVALUE|NOMAXVALUE, MINVALUE|NOMINVALUE, CYCLE|NOCYCLE, ORDER|NOORDER, and CACHE|NOCACHE options while modifying a sequence. The general syntax is

```
ALTER SEQUENCE sequencename
    [INCREMENT BY n]
    [MAXVALUE x | NOMAXVALUE]
    [MINVALUE m | NOMINVALUE]
    [CYCLE | NOCYCLE]
    [CACHE c | NOCACHE]
    [ORDER | NOORDER];
```

```
SQL> ALTER SEQUENCE major_majorid_seq
  2  INCREMENT BY 50
  3  MAXVALUE 999;

Sequence altered.

SQL>
```

Figure 9-11 Alter Sequence.

For example, look at the sequence modification in Figure 9-11. Only two options are changed for simplicity. A user may choose to modify more options.

Dropping a Sequence

You can drop a sequence with the DROP SEQUENCE statement. A removed sequence cannot be used anymore. For example,

```
DROP SEQUENCE major_MajorId_seq;
```

SYNONYMS

Sometimes, object names are very long. If a query uses the object's name more than once, the user has to type that long name many times. You already know the use of table aliases to shorten table names. Table aliases are, indeed, useful, but they are only known in the query where they are created. They are not stored as separate objects in your database. Synonyms are Oracle objects that are used to create alternative names for tables, views, sequences, and other objects. Even when you have the privilege to use another user's table, you have to qualify the table name with the user's name. You can create a synonym for *username.tablename.* The general syntax is

```
CREATE [PUBLIC] SYNONYM SynonymName
  For ObjectName;
```

```
SQL> CREATE SYNONYM esq
  2  FOR employee_employeeid_seq;

Synonym created.

SQL> CREATE SYNONYM emp
  2  FOR employee;

Synonym created.

SQL>
```

Figure 9-12 Creating synonym.

A synonym name must be different than all the other objects owned by the user. For example, Figure 9-12 shows the creation of a synonym for a sequence as well as a table.

If you have rights equivalent to those of a database administrator (DBA), you can create a PUBLIC synonym. A public synonym is available to all users. I create PUBLIC synonyms for all sample database tables, because my tables are made available to students for lab activities. For example,

```
CREATE PUBLIC SYNONYM reg
  FOR nshah.registration;
```

A short synonym *Reg* is used for table REGISTRATION owned by user *nshah.* With availability of a public synonym, the students do not have to qualify my table name with my user name. They can use

```
SELECT * FROM reg;
```

instead of

```
SELECT * FROM nshah.registration;
```

A synonym can be removed by using the DROP SYNONYM statement. Only the DBA privilege allows you to remove a public synonym. For example,

```
DROP SYNONYM esq;
```

```
SQL> SELECT SYNONYM_NAME,   TABLE_NAME,    TABLE_OWNER
  2  FROM USER_SYNONYMS;

SYNONYM_NAME                TABLE_NAME                     TABLE_OWNER
--------------------------  -----------------------------  -------------------
EMP                         EMPLOYEE                       NSHAH
ESQ                         EMPLOYEE_EMPLOYEEID_SEQ        NSHAH

SQL>
```

Figure 9-13 User_Synonyms.

A user can get information about synonyms and their table names by using Oracle's Data Dictionary table, USER_SYNONYMS (see Fig. 9-13).

INDEX

An index is another Oracle object that is used for faster retrieval of rows from a table. An index can be created explicitly by using the CREATE INDEX statement

or implicitly by Oracle. Once an index exists for a table, the user does not have to open or use the index with a command or a statement. The Oracle server uses the index to search for a row rather than scanning through the entire table. Indexing reduces both search time and disk input/output. All indexes are maintained separately from the table on which they are based. Creating and removing an index does not affect the table at all. When a table is dropped, all indexes based on that table are also removed.

Implicit indexes are created when the PRIMARY KEY or UNIQUE constraint is defined. Such indexes get the name of the constraint. A user can create explicit indexes based on non–primary key or nonunique columns or on any combination of columns for faster table access. An index based on a combination of columns is called a *composite index* or *concatenated index*. The general syntax is

```
CREATE INDEX indexname
   ON tablename (columnname1 [, columnname2], . . .);
```

The TABLESPACE and STORAGE clauses can be used with the CREATE INDEX statement. Indexes are stored in a different tablespace and, preferably, on a different physical device from the table data to optimize the index's performance. For example, Figure 9-14 creates an index based on student's last and first names to

```
SQL> CREATE INDEX stu_idx
  2  ON student(Last,  First);

Index created.

SQL> SELECT index_name, table_name FROM user_indexes
  2  WHERE table_name = 'STUDENT';

INDEX_NAME                     TABLE_NAME
------------------------------ ------------------------------
STUDENT_STUDENTID_PK           STUDENT
STU_IDX                        STUDENT

SQL>
```

Figure 9-14 Index.

speed searching of student information when the search involves the last name and first name as search key. The information from the USER_INDEXES table shows two indexes in the STUDENT table, because one index was created automatically for the primary key constraint.

You would create an index based on a column if a column is used often in querying or joining, has a big domain of values, or contains many Null values. When an index is created, it does not store Null values, so the searching would eliminate those rows. Do not create an index for a very small table, a column not used often in queries, or a table that often gets updated. Every insertion and deletion in a table

updates the index, which is added overhead on the system.

Oracle also supports *bitmapped* indexes, which are used in data warehousing but are not suitable for tables with a large number of updates. Each data manipulation statement updates the index file, thus slowing down the data manipulation operation.

Rebuilding an Index

When a table goes through many changes (insertions, deletions, and updates), it is advisable to rebuild indexes based on that table. You can drop an index and recreate it, but it is faster to just rebuild an existing index. Rebuilding compacts index data and improves performance. For example,

```
ALTER INDEX stu_idx REBUILD;
```

ROWID PSEUDOCOLUMN

Every row has a unique, system-generated address called the ROWID. This address contains the exact location of the row, and the index files store ROWID to retrieve rows. The row address or physical storage location of a row consists of the data object number, data block number, number of the row within the data block, and data file number. The ROWID can be used in a query for faster access of a row, but it cannot be changed. If a row is deleted and then inserted again, it gets a new ROWID, as shown in Figure 9-15.

```
SQL> SELECT ROWID, EmployeeId, Lname, Fname
  2  FROM employee
  3  WHERE EmployeeId = 200;

ROWID                  EMPLOYEEID  LNAME       FNAME
------------------  ----------  ----------  ----------
AAAHaCAABAAAMbaAAF         200  Shaw        Jinku

SQL> SELECT EmployeeId, Lname, Fname
 2   FROM employee
 3   WHERE ROWID = 'AAAHaCAABAAAMbaAAF';

EMPLOYEEID  LNAME           FNAME
----------  --------------  --------------
       200  Shaw            Jinku

SQL>
```

Figure 9-15 ROWID Pseudocolumn.

TRANSACTIONS

Oracle groups your Structured Query Language (SQL) statements into transactions. A transaction consists of a series of Data Manipulation Language (DML) statements, one Data Definition Language (DDL) statement, or one Data Control Language (DCL) statement. Because transaction control, Oracle guarantees data consistency. The transaction control statements give you flexibility to undo transactions or write transactions to the disk. Transactions provide consistency in case of a system failure.

In Oracle, your transactions start with the first executable SQL statement that you issue. The transaction ends when one of the following occurs:

- A COMMIT or ROLLBACK transaction control statement is used.
- A DDL (CREATE, ALTER, DROP, RENAME, or TRUNCATE) statement is executed (automatic commit).
- A DCL (GRANT or REVOKE) statement is executed (automatic commit).
- A user properly exits (commit).
- The system crashes (rollback).

Once a transaction ends, the new transaction starts with your next executable SQL statement.

There is also an environment variable AUTOCOMMIT. By default, it is set to OFF. A user can set it to ON or IMMEDIATE by typing

```
SET AUTOCOMMIT ON
SET AUTOCOMMIT IMMEDIATE
```

When AUTOCOMMIT is set to ON or IMMEDIATE, every DML statement is written to the disk as soon as it is executed, every DML statement is committed implicitly, and no rollback occurs with AUTOCOMMIT. AUTOCOMMIT can be toggled back to OFF for an explicit COMMIT. If the system crashes, any statements after the last COMMIT are rolled back, so partial changes to tables are not permanently written.

When a user is in the middle of a transaction, he or she can review the results of all DML statements by using SELECT queries. After reviewing the results, the user can decide to roll back or commit. The user is getting results from the database's temporary storage area. The other users with privileges on the same table cannot view the results of the DML queries until the user commits the changes. Until the user commits all the changes, the rows with DML statements are locked. Other users cannot change the data in the locked rows. By committing, the user changes become permanent, all users can view the changes, locks are released, save points are removed, and other users can manipulate affected rows.

In Figure 9-16, you see actions performed by three transaction control statements; COMMIT, SAVEPOINT, and ROLLBACK. In Figure 9-17, you see illustration

Transaction Control Statement	Action
COMMIT	Ends the current transaction, and writes all changes permanently to the disk.
SAVEPOINT *n*	Marks a point in the current transaction.
ROLLBACK [TO SAVEPOINT *n*]	Ends the current transaction by undoing all changes to the last commit if a TO SAVEPOINT clause is not used. It rolls back to the save point if the clause is used, removing the save point and any changes after the save point, but it does not end the transaction.

Figure 9-16 Transaction control statements.

```
SQL> INSERT INTO course VALUES('CIS395', 'ADV DATABASE', 3, 'CIS253');

1 row created.

SQL> INSERT INTO course VALUES('CIS340', 'OBJ ANALYSIS', 3, 'CIS265');

1 row created.

SQL> SAVEPOINT X;

Savepoint created.

SQL> UPDATE course SET credits=4 WHERE CourseId='CIS340';

1 row updated.

SQL> DELETE FROM course WHERE CourseId='CIS395';

1 row deleted.

SQL> SAVEPOINT Y;

Savepoint created.

SQL> INSERT INTO course VALUES('BL101', 'BUSINESS LAW', 3, '');

1 row created.

SQL> ROLLBACK TO SAVEPOINT Y;

Rollback complete.

SQL> ROLLBACK TO SAVEPOINT X;

Rollback complete.

SQL> ROLLBACK;

Rollback complete.
```

Figure 9-17 SAVEPOINT and ROLLBACK.

of a transaction with five DML statements. The transaction has started right after the some COMMIT statement and the user is at the end of the fifth DML statement. If user types COMMIT, the results of all five DML statements are written permanently to the disk. If the user types ROLLBACK TO SAVEPOINT Y, one INSERT after SAVEPOINT Y is rolled back, and save point Y is removed. If the user types ROLLBACK TO SAVEPOINT X, save point Y is removed along with save point X. Three DML statements—UPDATE, DELETE, and INSERT—are rolled back. If the user types ROLLBACK, the entire transaction is rolled back, all save points are removed, and a new transaction begins with the new statement. In Figure 9-17, one by one the save points are rolled back, and eventually, the entire transaction is rolled back. Back to square one!

- **Question:** If you typed a DML statement that failed during execution, is it still part of your transaction? Is it committed? Is it rolled back?
- **Answer:** The statement is rolled back, and it is no longer part of your transaction.

Read Consistency and Locking

Oracle implements read consistency. When any DML statement is issued for a database, the old copy of the database (before changes) is written into a rollback segment. The user who is making changes with DML statements can see the changes with a SELECT query from the database. All other users see the snapshot of the database before changes in the rollback segment. The data are consistent to all other users. When the changes are committed, the changes are available to all users, because the rollback segment copy is removed and the space is freed for another, later use. If the changes are rolled back, the old copy of the database is loaded into the rollback segment again.

A user does not have to write any explicit statements to lock tables. Oracle uses automatic locking with the least restrictions to provide sharing of data with integrity. Oracle has two lock classes; *exclusive* and *share lock*. An exclusive lock prevents sharing a resource until the lock is released. A share lock allows sharing for reading purposes. Oracle also allows a manual lock on data.

LOCKING ROWS FOR UPDATE

When a user issues a SELECT query, rows are not locked. Oracle does not lock rows for viewing. As you know, the rows with DML query changes are locked until changes are committed. Suppose you want to view rows and also change them. You would want to lock those rows for future changes. You would use the SELECT ... FOR UPDATE OF statement for such manual locks. The general syntax is

```
SELECT columnnames
  FROM tablenames
```

```
[WHERE condition]
FOR UPDATE OF columnnames
[NOWAIT];
```

The use of *columnnames* in FOR UPDATE OF does not mean that the locking is at the column level. The entire row is locked. The column names are used just for information. The NOWAIT clause, which tells the user instantaneously if another user has already locked the row, is optional. If you do not use NOWAIT in the statement, you will have to wait for any rows that have been locked by other applications to be released. You will not be able to do any processing on those rows until then.

Figure 9-18 shows the display of a row in the EMPLOYEE table and its locking. Once the row is locked, you can change the employee's salary and commission. You can actually change any column's value in this row. If another user tries to update this row, he or she will have to wait for you to release the lock by either committing or rolling back the transaction. It is a good practice to run COMMIT as soon as changes are done so that other users can access data.

```
SQL> SELECT Lname, Fname, Salary, Commission
  2  FROM employee
  3  WHERE EmployeeId = 544
  4  FOR UPDATE OF Salary, Commission
  5  NOWAIT;

LNAME           FNAME           SALARY    COMMISSION
--------------- --------------- --------- ----------
Viquez          Heillyn

SQL>
```

Figure 9-18 Locking a row with FOR UPDATE OF.

CONTROLLING ACCESS

A user's access needs to be controlled in a shared, multiuser Oracle environment. A user's access to the database can be restricted, and a user may or may not be allowed to use certain objects in the database. Security is classified into two types:

1. ***System security*** defines access to the database at the system level. It is implemented by assigning a username and password, allocating disk space, and providing a user with the ability to perform system operations.
2. ***Database security*** defines a user's access to various objects and the tasks a user can perform on them.

The Database Administrator (DBA) is the most trusted user, and a DBA has all the privileges. A DBA can create users, assign them privileges, and even drop users.

Users and Roles

The DBA can create a user with the CREATE USER statement. Once a user is created and a password assigned, the user needs privileges to do anything. The general syntax is

```
CREATE USER username[PROFILE profilename]
    IDENTIFIED BY password
    [DEFAULT TABLESPACE tablespacename]
    [TEMPORARY TABLESPACE tablespacename]
    [PASSWORD EXPIRE] [ACCOUNT UNLOCK];
```

The statement in Figure 9-19 creates a user XMAN, assigns a password as well as temporary and default tablespaces, and grants roles. A user needs, at minimum, two roles; CONNECT and RESOURCE.

```
SQL> CREATE USER "XMAN" PROFILE "DEFAULT"
  2      IDENTIFIED BY "CRICKET"
  3      DEFAULT TABLESPACE "CIS_DATA"
  4   TEMPORARY TABLESPACE "TEMP_DATA"
  5   ACCOUNT UNLOCK;

User created.

SQL> GRANT UNLIMITED TABLESPACE TO "XMAN";

Grant succeeded.

SQL> GRANT "CONNECT" TO "XMAN";

Grant succeeded.

SQL> GRANT "RESOURCE" TO "XMAN";

Grant succeeded.

SQL>
```

Figure 9-19 Creating a new user.

A user can change his/her own password with the PASSWORD command in SQL*Plus. A DBA can change any user's password with the ALTER USER statement. For example,

```
ALTER USER "XMAN" IDENTIFIED BY "PATRICK";
```

There is a pool of more than 100 system privileges available for the DBA to grant to users. The DBA assigns privileges based on the level of a user or on a user's needs.

In a company, there are many users of different levels. Many of them from the same level need the same privileges. It is easier to assign privileges to each level and then to assign users to that level. The levels are called **roles**. A role is similar to a group used by network operating systems. A DBA creates a role by using a CREATE ROLE statement. Privileges are then granted to the role, and the role is granted to the users. The general syntax is

```
GRANT privileges | role
    TO username1 [, username2  . . .];
```

Figure 9-19 shows the roles assigned to user XMAN. Once user XMAN gets privileges directly or through a role to go with the username and password, he or she can use those privileges immediately. Some users can be assigned more or fewer privileges than other users.

Object Privileges

An object privilege specifies what a user can do with a database object, such as a table, a sequence, or a view. There are 11 basic object privileges, and each object has a set of privileges out of the total of 11 privileges. The following is a list of all object privileges on various objects (objects are listed within parentheses):

- ALTER (table, sequence)
- INSERT (table, view)
- UPDATE (table, view)
- DELETE (table, view)
- SELECT (table, view, sequence)
- REFERENCES (table, view)
- INDEX (table)
- EXECUTE (procedure, function, package, library, user-defined type)
- UNDER (view, user-defined type)
- READ (directory)
- WRITE (directory)

UNDER is a new object privilege introduced with Oracle9i. It lets you create a subview under the current view. You can grant this object privilege only if you have the UNDER ANY VIEW privilege WITH GRANT OPTION on the immediate superview of this subview.

A user has his or her objects in his or her own schema. An owner has all possible privileges on the owner's objects. A user can grant privileges on objects from his or her own schema to other users or roles. The grantee can also be given further rights to grant the same privileges to other users on an object. The general syntax is

```
GRANT objectprivileges [(columnnames)] | ALL
    ON objectname
```

```
TO user|role|PUBLIC
[WITH GRANT OPTION];
```

where *objectprivileges* are some of the 11 privileges listed before. If all privileges are to be granted, ALL can be used instead of specifying each privilege separately. Columns on which privileges are granted are specified by *columnnames*. The key word PUBLIC grants privileges to all users. The WITH GRANT OPTION clause allows the grantee to grant privileges to other users and roles.

Figure 9-20 illustrates two GRANT statements. The first statement gives only SELECT privileges on table DEPT to user XMAN. The second statement gives SELECT, INSERT, and UPDATE privileges on table EMPLOYEE to two users, STUDENT and XMAN. The second statement also gives the two users privilege to pass those privileges to other users. If user NSHAH is granting these privileges, the grantee will have to qualify the table name with a username to use it (e.g., NSHAH.EMPLOYEE).

```
SQL> GRANT SELECT
  2  ON dept
  3  TO XMAN;

Grant succeeded.

SQL> GRANT SELECT, INSERT, UPDATE
  2    ON employee
  3    TO STUDENT, XMAN
  4    WITH GRANT OPTION;

Grant succeeded.

SQL>
```

Figure 9-20 Granting object privileges.

The user XMAN can use table EMPLOYEE with the following query:

```
SELECT * FROM nshah.employee;
```

Another approach is to create a synonym for the table, as you saw earlier in this chapter. For example,

```
CREATE SYNONYM emp FOR nshah.employee;
```

Privileges can be granted, and they can be taken away. If a user granted privileges by a WITH GRANT OPTION to another user and that second user passed on those privileges, the REVOKE statement takes privileges not only from the grantee but also from the users granted privileges by the grantee. The general syntax is

```
REVOKE privilege1 [, privilege2 . . . ] | ALL
   ON objectname
   FROM users|role|PUBLIC
   [CASCADE CONSTRAINTS];
```

The user with the REFERENCES privilege in the GRANT statement can reference the table. CASCADE CONSTRAINTS removes any foreign key or referential integrity constraints on the object.

In Figure 9-21, user XMAN loses ALL granted privileges on granted the EMPLOYEE table from user NSHAH. If user XMAN has passed privileges on the EMPLOYEE table to any other users, their privileges are also revoked.

```
SQL> REVOKE ALL
  2  ON employee
  3  FROM XMAN;

Revoke succeeded.

SQL>
```

Figure 9-21 Revoking object privileges.

Figure 9-22 shows a table with object privileges, which can be granted on various Oracle objects. Some objects, such as operator, indextype, and so on, are not included in this table, because they are beyond the scope of this text. You may grant the EXECUTE privilege to Java source, class, or resource created in Oracle with the CREATE JAVA statement. Oracle treats Java objects as procedures.

Object Privilege	Table	View	Sequence	Procedures, Functions, and Packages	Directory	Library	User-Defined Type
ALTER	X		X				
DELETE	X	X					
EXECUTE				X		X	X
INDEX	X						
INSERT	X	X					
ON COMMIT REFRESH	X						
QUERY REWRITE	X						
READ					X		
REFERENCES	X	X					
SELECT	X	X	X				
UNDER		X					X
UPDATE	X	X					
WRITE					X		

Figure 9-22 Object privileges and Oracle objects.

IN A NUTSHELL . . .

- The Oracle objects view, sequence, index, and synonym are stored in a user's schema.
- A view has two types: simple and complex.
- A simple view is based on a single table and does not contain function or grouped data. The manipulation of data is always possible from a simple view.
- A complex view is based on one or more tables and may contain functions and grouped data. Manipulation of data is not always possible from a complex view.
- A sequence is an Oracle object that is used to generate a sequence of numbers. The sequence number may start at any number and have any increment value, either in chronological order or unordered.
- A sequence object can be modified with some restrictions. A gap may occur in sequence values because of a system crash, a deletion of rows, an update operation, or the use of a sequence on more than one column.
- A synonym is used to provide a shortened name for an object.
- Oracle automatically creates an index object for all primary key and unique constraints. A user can create an index based on a set of columns for faster access.
- Transactions in Oracle provide the user with more flexibility and consistency in data. A user can employ COMMIT or ROLLBACK transactions.
- Oracle provides read consistency and automatic locking. A user can manually lock rows for updating with the FOR UPDATE OF clause.
- A database administrator (DBA) has all privileges. There are more than 100 system privileges and 11 object privileges. The DBA creates users and roles, and he or she grants them privileges.
- A user can grant privileges to other users on any object in the user's own schema and can also revoke those privileges.

EXERCISE QUESTIONS

True/False:

1. A simple view is based on a single table.
2. A complex view is always based on two or more tables.
3. In a sequence, to get the current value with CURRVAL, at least one number must be generated first by using NEXTVAL.
4. If a table is dropped, all indexes based on that table are automatically dropped.

5. When a system crashes, it results in an automatic COMMIT.
6. DML statements cannot be rolled back once committed.
7. Any user can create another user with the same privileges.
8. A user cannot pass on privileges to other users on objects from his or her own schema unless the WITH GRANT OPTION is used in a GRANT statement.
9. The SELECT statement generates an automatic lock on rows.
10. When DML statements are not committed, other users cannot modify the rows involved.
11. A sequence can be used only on the one table for which its created.
12. If a user owns a table named EMP, the user cannot have a synonym named EMP.
13. A view is created with WITH CHECK OPTION to prevent modifications to all columns used in the view.
14. CURRVAL returns the START WITH value of a sequence for a sequence that is created but is never used to generate a value.
15. When a view is dropped, the underlying table is also dropped.
16. When a table is dropped, the sequence used on that table also is dropped.

Answer the Following Questions:

1. Name any five Oracle objects.
2. What are the differences between simple and complex views?
3. When does an automatic commit occur?
4. What is the advantage of creating the following objects?
   a. Index.
   b. Sequence.
   c. View.
   d. Synonym.
5. How is transaction control important in a shared environment?
6. Name Oracle9i's object privileges. What is used to grant all object privileges? What is used to grant privileges to everybody?
7. When does an automatic rollback occur?
8. For what are the pseudocolumns ROWID, ROWNUM, NEXTVAL, and CURRVAL used?
9. How can you lock rows in a table with the SELECT query? If another user has already locked those rows, what will happen if you try to lock them as well? Is there any way to avoid the situation?
10. When a table is dropped, what happens to its indexes, synonyms, views, and sequences?

LAB ACTIVITY

Use the Case-Study Databases for the Following Queries:

1. Create a view to include all employee information, but hide salary and commission.
2. Create a view to include department name and average salary by department.

3. Create an index to search students faster based on their major ID.
4. Create a sequence to add room IDs, and then insert a new room into the LOCATION table using the newly created sequence. What is the CURRVAL after the new row is inserted?
5. GRANT only the SELECT privilege to another user on your TERM table.
6. INSERT a new Winter 2004 term in the TERM table. Use a SELECT query to see the result. Ask the user with the SELECT privilege (from activity 5) to view your TERM table. COMMIT your transaction, and ask the same user to view the table again.
7. Lock student ID 00101 for an update of major to 600. Update the row, and COMMIT.
8. Create a view that will display name, department number, and total income (salary + commission) of each employee in Department 10. Prevent a change of department through the view.
9. Create a view that will display department names and the sum of all employee income by department.
10. Create a sequence deptid_seq to generate department Id (in the DEPT table) and another sequence empid_seq to generate employee Id (in the EMPLOYEE table). Use deptid_seq to add a new department in the DEPT table. Now, add yourself as a new employee with empid_seq in the department you just added.

SQL Review: Supplementary Examples

In this section, a new database is introduced with four tables. This review section is added at this point in the text to review all the statements covered in Chapters 3 through 9. The scripts for creating all four tables and inserting rows into them are also provided after the illustrations of the tables. Each script is followed by various problems and their solutions. The four sample tables are:

1. ***CUSTOMER:*** A table with customer demographic information.
2. ***ITEM:*** A table with information about the items offered by the company.
3. ***INVOICE:*** A table with individual invoices produced for customers.
4. ***INVITEM:*** A composite entity with invoices and items ordered.

CUSTOMER (CustNo, CustName, State, Phone)

CustNo	CustName	State	Phone
211	Garcia	NJ	732-555-1000
212	Parikh	NY	212-555-2000
225	Elsenhauer	NJ	973-555-3333
239	Bayer	FL	407-555-7777

ITEM (ItemNo, ItemName, ItemPrice, QtyOnHand)

ItemNo	ItemName	ItemPrice	QtyOnHand
1	Screw	2.25	50
2	Nut	5.00	110
3	Bolt	3.99	75
4	Hammer	9.99	125
5	Washer	1.99	100
6	Nail	0.99	300

INVOICE (InvNo, InvDate, CustNo)

InvNo	InvDate	CustNo
1001	05-SEP-03	212
1002	17-SEP-03	225
1003	17-SEP-03	239
1004	18-SEP-03	211
1005	21-SEP-03	212

INVITEM (InvNo, ItemNo, Qty)

InvNo	ItemNo	Qty
1001	1	5
1001	3	5
1001	5	9
1002	1	2
1002	2	3
1003	1	7
1003	2	1
1004	4	5
1005	4	10

SCRIPT FOR CREATION OF TABLES

```
CREATE TABLE customer(CustNo NUMBER (3),
        CustName VARCHAR2(10) CONSTRAINT customer_custname_nn NOT NULL,
        State CHAR(2) DEFAULT 'NJ',
        Phone CHAR(12),
        CONSTRAINT customer_custno_pk PRIMARY KEY (CustNo));
CREATE TABLE item(ItemNo NUMBER (2),
        ItemName VARCHAR2(6),
        ItemPrice NUMBER(3,2),
        QtyOnHand NUMBER(3),
        CONSTRAINT item_itemno_pk PRIMARY KEY(ItemNo),
        CONSTRAINT item_qtyonhand_ck CHECK (QtyOnHand >= 0));
CREATE TABLE invoice(InvNo NUMBER(4),
        InvDate DATE,
        CustNo NUMBER(3) CONSTRAINT invoice_custno_nn NOT NULL,
        CONSTRAINT invoice_invoiceno_pk PRIMARY KEY(InvNo),
        CONSTRAINT invoice_custno_fk FOREIGN KEY(CustNo)
                REFERENCES customer(CustNo));
    CREATE TABLE invitem(InvNo NUMBER(4),
        ItemNo NUMBER(2),
```

```
    Qty NUMBER(2) NOT NULL,
    CONSTRAINT invitem_invno_itemno_pk PRIMARY KEY(InvNo, ItemNo),
    CONSTRAINT invitem_invno_fk FOREIGN KEY(InvNo)
            REFERENCES invoice(InvNo),
    CONSTRAINT invitem_itemno_fk FOREIGN KEY(ItemNo)
            REFERENCES item(ItemNo));
```

SCRIPT FOR INSERTION OF ROWS INTO TABLES

```
INSERT INTO customer VALUES(211, 'Garcia', 'NJ', '732-555-1000');
INSERT INTO customer VALUES(212, 'Parikh', 'NY', '212-555-2000');
INSERT INTO customer VALUES(225, 'Eisenhauer', 'NJ', '973-555-3333');
INSERT INTO customer VALUES(239, 'Bayer', 'FL', '407-555-7777');

INSERT INTO item VALUES(1, 'Screw', 2.25, 50);
INSERT INTO item VALUES(2, 'Nut', 5.00, 110);
INSERT INTO item VALUES(3, 'Bolt', 3.99, 75);
INSERT INTO item VALUES(4, 'Hammer', 9.99, 125);
INSERT INTO item VALUES(5, 'Washer', 1.99, 100);
INSERT INTO item VALUES(6, 'Nail', 0.99, 300);

INSERT INTO invoice VALUES(1001, TO_DATE('05-09-2003', 'dd-mm-yyyy'), 212);
INSERT INTO invoice VALUES(1002, TO_DATE('17-09-2003', 'dd-mm-yyyy'), 225);
INSERT INTO invoice VALUES(1003, TO_DATE('17-09-2003', 'dd-mm-yyyy'), 239);
INSERT INTO invoice VALUES(1004, TO_DATE('18-09-2003', 'dd-mm-yyyy'), 211);
INSERT INTO invoice VALUES(1005, TO_DATE('21-09-2003', 'dd-mm-yyyy'), 212);

INSERT INTO invitem VALUES(1001, 1, 5);
INSERT INTO invitem VALUES(1001, 3, 5);
INSERT INTO invitem VALUES(1001, 5, 9);
INSERT INTO invitem VALUES(1002, 1, 2);
INSERT INTO invitem VALUES(1002, 2, 3);
INSERT INTO invitem VALUES(1003, 1, 7);
INSERT INTO invitem VALUES(1003, 2, 1);
INSERT INTO invitem VALUES(1004, 4, 5);
INSERT INTO invitem VALUES(1005, 4, 10);
```

INSERTION OF ROWS WITH SUBSTITUTION VARIABLES

Alternate Method

```
INSERT INTO customer
  VALUES(&customer_no, '&customer_name', '&state', '&phone');
INSERT INTO item
  VALUES(&item_no, '&item_name', &price, &qty_on_hand);
```

```
INSERT INTO invoice
  VALUES(&inv_no, TO_DATE ('&inv_date', 'dd-mm-yyyy'), &cust_no);
INSERT INTO invitem
  VALUES(&invoice_no, &item_no, &qty);
```

Display all Customer Information

```
SELECT *
  FROM customer;
```

Display all Item Names and their Respective Unit Price

```
SELECT ItemName, ItemPrice
  FROM item;
```

Display Unique Invoice Numbers from the INVITEM Table

```
SELECT DISTINCT InvNo
  FROM invitem;
```

Display Item Information with Appropriate Column Aliases

```
SELECT ItemNo "Item Number", ItemName "Name of Item",
    ItemPrice "Unit Price"
    FROM item;
```

Display Item Name and Price Using Concatenation

```
SELECT ItemName || 'has a unit price of $' || ItemPrice FROM item;
```

Find the Total Value of Each Item Based on Quantity on Hand

```
SELECT ItemName, ItemPrice * QtyOnHand "Total Value"
    FROM item;
```

Find Customers from Florida

```
SELECT *
  FROM customer
  WHERE UPPER(State) = 'FL';
```

Display Items with a Unit Price of at Least $5

```
SELECT UPPER (ItemName), ItemPrice
  FROM item
  WHERE ItemPrice >= 5;
```

Find Items with a Unit Price Between $2 and $5

```
SELECT *
   FROM item
   WHERE ItemPrice BETWEEN 2 and 5;
```

or

```
SELECT *
   FROM item
   WHERE ItemPrice >= 2 AND ItemPrice <= 5;
```

Find Customers from the Tristate Area of New York, New Jersey, and Connecticut

```
SELECT *
   FROM customer
   WHERE State IN ('NJ', 'NY', 'CT');
```

Find all Customers Whose Names Start with the Letter *E*

```
SELECT *
   FROM customer
   WHERE UPPER(CustName) LIKE 'E%';
```

Find Items with the Letter *W* in their Name

```
SELECT *
   FROM item
   WHERE ItemName LIKE '%w%';
```

Sort all Customers Alphabetically

```
SELECT *
   FROM customer
   ORDER BY CustName;
```

Sort all Items in Descending Order by their Price

```
SELECT *
   FROM item
   ORDER BY ItemPrice DESC;
```

Sort all Customers by their State and also Alphabetically

```
SELECT *
   FROM customer
   ORDER BY State, CustName;
```

Display all Customers from New Jersey Alphabetically

```
SELECT *
   FROM customer
   WHERE UPPER(State) = 'NJ'
   ORDER BY CustName;
```

Display all Item Prices Rounded to the Nearest Dollar

```
SELECT ItemName, ROUND(ItemPrice, 0)
   FROM item;
```

Find the Payment Due Date if the Payment is Due in Two Months from the Invoice Date

```
SELECT InvNo, CustNo, InvDate, ADD_MONTHS(InvDate, 2) "Payment Due"
   FROM invoice;
```

or

```
SELECT InvNo, CustNo, InvDate, InvDate + 60 "Payment Due"
   FROM invoice;
```

Display Invoice Dates in "September 05, 2003" Format

```
SELECT InvNo, TO_CHAR (InvDate, 'fmMonth DD, YYYY') "Invoice Date"
   FROM invoice;
```

Find the Total, Average, Highest, and Lowest Unit Prices

```
SELECT SUM (ItemPrice), AVG (ItemPrice), MAX (ItemPrice), MIN (ItemPrice)
   FROM item;
```

Display How Many Different Items Are Available for Customers

```
SELECT COUNT (*)
   FROM item;
```

Count the Number of Items Ordered in Each Invoice

```
SELECT InvNo, COUNT(ItemNo) "Items Ordered"
   FROM invitem
   GROUP BY InvNo;
```

Find Invoices in which Three or More Items Are Ordered

```
SELECT InvNo, COUNT(ItemNo) "Items Ordered"
   FROM invitem
   GROUP BY InvNo
   HAVING COUNT(ItemNo) >= 3;
```

Find all Possible Combinations of Customers and Items (Cartesian Product)

```
SELECT c.*, t.*
    FROM customer c, item t;
```

Display all Item Quantities and Item Prices for Invoices

```
SELECT a.InvNo, a.ItemNo, b.ItemName, a.Qty, b.ItemPrice, a.Qty * b.ItemPrice "qty*price"
    FROM invitem a, item b
    WHERE a.ItemNo = b.ItemNo;
```

Find the Total Price for Each Invoice

```
SELECT a.InvNo, SUM(a.Qty * b.ItemPrice) "Total Amount"
    FROM invitem a, item b
    WHERE a.ItemNo = b.ItemNo
    GROUP BY a.InvNo;
```

Use an Outer Join to Display Items Ordered and Not Ordered

```
SELECT x.ItemNo, y.InvNo
    FROM item x, invitem y
    WHERE x.ItemNo = y.ItemNo(+);
```

Display Invoices, Customer Names, and Item Names Together (Multiple Joins)

```
SELECT a.InvNo, b.CustName, c.ItemName, d.Qty
    FROM invoice a, customer b, item c, invitem d
    WHERE a.CustNo = b.CustNo AND a.InvNo = d.InvNo
    AND c.ItemNo = d.ItemNo;
```

Find Invoices with *HAMMER* as an Item

```
SELECT v.InvNo, t.ItemName, v.Qty
    FROM invitem v, item t
    WHERE v.ItemNo = t.ItemNo AND UPPER(ItemName) = 'HAMMER';
```

Find Invoices with *HAMMER* as an Item by Using a Subquery

```
SELECT InvNo, Qty "Hammers ordered"
    FROM invitem
    WHERE ItemNo =
    (SELECT ItemNo FROM item WHERE Upper(t.ItemName) = 'HAMMER');
```

Display the Items Ordered in Invoice Number 1001 (Subquery)

```
SELECT ItemName
    FROM item
    WHERE ItemNo IN
    (SELECT ItemNo FROM invitem WHERE InvNo = 1001);
```

Find Items That Are Chapter than *NUT*

```
SELECT ItemName, ItemPrice
    FROM item
    WHERE ItemPrice <
    (SELECT ItemPrice FROM item WHERE UPPER(ItemName) = 'NUT');
```

Create a New Table for all New Jersey Customers Based on the Existing CUSTOMER Table

```
CREATE TABLE nj_customer
AS
SELECT CustNo, CustName, Phone
    FROM customer
    WHERE UPPER(State) = 'NJ';
```

Copy all New York Customers to the Newly Created NJ_CUSTOMER Table

```
INSERT INTO nj_customer
    SELECT CustNo, CustName, Phone
    FROM customer
    WHERE UPPER(State) = 'NY';
```

Rename NJ_CUSTOMER Table to NYNJ_CUSTOMER

```
RENAME nj_customer TO nynj_customer;
```

Find Customers Who Are Not from New York or New Jersey (Set Operator)

```
SELECT CustName, State
    FROM customer
MINUS
SELECT CustName, State
    FROM nynj_customer;
```

Delete Rows from the CUSTOMER Table that Are also in the NYNJ_CUSTOMER Table

```
DELETE FROM customer
    WHERE CustNo IN
    (SELECT CustNo FROM nynj_customer);
```

Find the Items with the Top-Three Prices

```
SELECT ROWNUM, ItemName, ItemPrice
    FROM (SELECT ItemName, ItemPrice FROM item ORDER BY ItemPrice DESC)
WHERE ROWNUM <=3;
```

Find the Two Items with the Lowest Quantity on Hand

```
SELECT ROWNUM, ItemName, ItemPrice, QtyOnHand
    FROM (SELECT ItemName, ItemPrice, QtyOnHand FROM item ORDER BY QtyOnHand)
WHERE ROWNUM <=2;
```

Create a Simple View with Item Names and Item Prices Only

```
CREATE OR REPLACE VIEW item_vu(Name, Price)
AS
SELECT ItemName, ItemPrice
    FROM item;
```

Create a View that Displays Invoice Number and Customer Names for New Jersey Customers

```
CREATE OR REPLACE VIEW nj_cust_vu
AS
SELECT InvNo, CustName
    FROM invoice, customer
    WHERE invoice.CustNo = customer.CustNo
    AND UPPER(State) = 'NJ'
WITH CHECK OPTION;
```

Create a Sequence that Can Be Used to Enter New Items into the ITEM Table

```
CREATE SEQUENCE itemnum_seq
    INCREMENT BY 1
    START WITH 7
    MAXVALUE 99
```

```
NOCYCLE
NOCACHE
ORDER;
```

Add a New Item into the ITEM Table with the ITEMNUM_SEQ Sequence

```
INSERT INTO item
    VALUES (itemnum_seq.NEXTVAL, 'Scissors', 7.95, 100);
```

Create a Synonym for the INVITEM Table

```
CREATE SYNONYM ii
    FOR invitem;
```

Create an Index File Based on Customer Name

```
CREATE INDEX customer_name_idx
    ON customer(CustName);
```

Lock Customer Bayer's Record to Update State and Phone Number

```
SELECT *
    FROM customer
    WHERE UPPER(CustName) = 'BAYER'
    FOR UPDATE OF State, Phone
    NOWAIT;
```

Give Everybody SELECT and INSERT Rights on Your ITEM Table

```
GRANT select, insert
    ON item
    TO public;
```

Revoke the INSERT Option on the ITEM Table from User BOND

```
REVOKE insert
ON item
FROM bond;
```

PART 3
PL/SQL

10

PL/SQL: A Programming Language

IN THIS CHAPTER . . .

- You will learn the basics of the PL/SQL programming language.
- The PL/SQL anonymous block is introduced.
- Variables, constants, data types, and declarations are discussed.
- The assignment statement and use of arithmetic operators are covered.
- The scope and use of various types of variables are shown in sample programs.
- You will be prepared to write simple PL/SQL blocks.

In Part 2, you learned Oracle's nonprocedural language SQL and its various statements to interface with the Oracle database. SQL is a great query language, but it has its limitations. So, Oracle Corporation has added a procedural language extension to SQL known as Programming Language Extensions to Structured Query Language (PL/SQL). It is Oracle's proprietary language for access of relational table data. PL/SQL is like any other high-level compiler language. If you are already familiar with another programming language, you will find PL/SQL constructs to be similar to those of Pascal, C, or Visual Basic. PL/SQL also possesses features of object-oriented languages, such as:

- Data encapsulation.
- Error handling.

- Information hiding.
- Object-oriented programming (OOP).

PL/SQL also allows embedding of SQL statements and data manipulation in its blocks. SQL statements are used to retrieve data, and PL/SQL control statements are used to process data in a PL/SQL program. The data can be inserted, deleted, or updated through a PL/SQL block, which makes it an efficient transaction-processing language.

The Oracle Server has an engine to execute SQL statements. The server also has a separate engine for PL/SQL. Oracle Developer tools have a separate engine to execute PL/SQL as well. The SQL statements are sent one at a time to the server for execution, which results in individual calls to the server for each SQL statement. It may also result in heavy network traffic. On the other hand, all SQL statements within a single PL/SQL block are sent in a single call to the server, which reduces overhead and improves performance.

A BRIEF HISTORY OF PL/SQL

Before PL/SQL was developed, users embedded SQL statements into hot languages like C++ and Java. PL/SQL version 1.0 was introduced with Oracle 6.0 in 1991. Version 1.0 had very limited capabilities, however, and was far from being a full-fledged programming language. It was merely used for batch processing.

With versions 2.0, 2.1, and 2.2, the following new features were introduced:

- The transaction control statements SAVEPOINT, ROLLBACK, and COMMIT.
- The DML statements INSERT, DELETE, and UPDATE.
- The extended data types Boolean, BINARY_INTEGER, PL/SQL records, and PL/SQL tables.
- Built-in functions—character, numeric, conversion, and date functions.
- Built-in packages.
- The control structures sequence, selection, and looping.
- Database access through work areas called cursors.
- Error handling.
- Modular programming with procedures and functions.
- Stored procedures, functions, and packages.
- Programmer-defined subtypes.
- DDL support through the DBMS_SQL package.
- The PL/SQL wrapper.
- The DBMS_JOB job scheduler.
- File I/O with the UTF_FILE package.

PL/SQL version 8.0, also known as PL/SQL8, came with Oracle8, the "object-relational" database software. Oracle allows creation of objects that can be accessed with Java, C++, Object COBOL, and other languages. It also allows objects and relational tables to coexist. The external procedures in Oracle allow you to compile procedures and store them in the shared library of the operating system—for example, an *.so* file in UNIX or a *.dll* (Dynamic Linked Library) file in Windows. Oracle's library is written in the C language. It also supports LOB (Large Object) data types.

FUNDAMENTALS OF PL/SQL

A PL/SQL program consists of statements. You may use upper- or lowercase letters in your program. In other words, PL/SQL is not case sensitive except for character string values enclosed in single quotes. Like any other programming language, PL/SQL statements consist of reserved words, identifiers, delimiters, literals, and comments.

Reserved Words

The reserved words, or key words, are words provided by the language that have a specific use in the language. For example, DECLARE, BEGIN, END, IF, WHILE, EXCEPTION, PROCEDURE, FUNCTION, PACKAGE, and TRIGGER are some of the reserved words in PL/SQL.

User-Defined Identifiers

User-defined identifiers are used to name variables, constants, procedures, functions, cursors, tables, records, and exceptions. A user must obey the following rules in naming these identifiers:

- The name can be from 1 to 30 characters in length.
- The name must start with a letter.
- Letters (A–Z, a–z), numbers, the dollar sign ($), the number sign (#) and the underscore (_) are allowed.
- Spaces are not allowed.
- Other special characters are not allowed.
- Key words cannot be used as user-defined identifiers.
- Names must be unique within a block.
- A name should not be the same as the name of a column used in the block.

It is a good practice to create short and meaningful names. Figure 10-1 shows a list of valid user-defined identifiers. Figure 10-2 shows a list of invalid user-defined identifiers along with reasons why they are invalid.

User-Defined Identifiers
Rate_of_pay
Num
A1234567890
Dollars$_and_cents
SS#

Figure 10-1 Valid user-defined identifiers.

Invalid User-Defined Identifiers	Reason
2Number	Starts with a number
Employee-name	Special character hyphen
END	Reserved word
Department number	Spaces
Largest_yearly_salary_paid_to_employees	Too long
Taxrate%	Special character %

Figure 10-2 Invalid user-defined identifiers.

Literals

Literals are values that are not represented by user-defined identifiers. Literals are of three types: numeric, character, and boolean. For example:

Numeric	**100, 3.14, −55, 5.25E7, or NULL**
Character	**'A', 'this is a string', '0001', '25-MAY-00', ' ', or NULL**
Boolean	**TRUE, FALSE, or NULL**

In this list of values, '25-MAY-00' looks like a date value, but it is a character string. It can be converted to date format by using the TO_DATE function. The value '' (two single quotes having nothing within) is another way of entering the NULL value.

PL/SQL is case sensitive regarding character values within single quotation marks. The values 'ORACLE', 'Oracle', and 'oracle' are three different values in PL/SQL. To embed a single quote in a string value, two single quote symbols are entered—for example, 'New Year''s Day'.

Numeric values can be entered in scientific notation with the letter *E* or *e*. Boolean values are not enclosed in quotation marks.

PL/SQL BLOCK STRUCTURE

PL/SQL is a block-structured language. A program can be divided into logical blocks. The block structure gives modularity to a PL/SQL program, and each object within a block has "scope." Blocks are of two types:

1. An ***anonymous block*** is a block of code without a name. It can be used anywhere in a program and is sent to the server engine for execution at runtime.

2. A ***named block*** is a block of code that is named. A *subprogram* is a named block that can be called and can take arguments. A *procedure* is a subprogram that can perform an action, whereas a *function* is a subprogram that returns a value. A *package* is formed from a group of procedures and functions. A *trigger* is a block that is called implicitly by a DML statement.

A PL/SQL block consists of three sections:

1. A **declaration section**.
2. An **executable section**.
3. An **exception-handling section**.

Figure 10-3 shows the use of three sections in a PL/SQL block. Of the three sections in a PL/SQL block, only the executable section is mandatory. The declaration and exception-handling sections are optional. The general syntax of an anonymous block is

```
[ DECLARE
  Declaration of constants, variables, cursors, and exceptions ]
BEGIN
  Executable PL/SQL and SQL statements
[EXCEPTION
  Actions for error conditions ]
END;
```

Section	Use
Declaration	An optional section to declare variables, constants, cursors, PL/SQL composite data types, and user-defined exceptions, which are referenced in executable and exception-handling sections.
Executable	A mandatory section that contains PL/SQL statements to manipulate data in the block and SQL statements to manipulate the database.
Exception handling	Specifies action statements to perform when an error condition exists in the executable section. It is also an optional section.

Figure 10-3 Sections in a PL/SQL block.

The DECLARE and EXCEPTION key words are optional, but the BEGIN and END key words are mandatory. The declarations made within a block are local to the block.

When a block ends, all objects declared within the block cease to exist. A block is the "scope" of objects declared in that block. When blocks are nested within each other, the declarations made in the outer block are global to the inner block. The object declarations made in the inner block, however, are local to it and cannot be referenced by the outer block. A basic PL/SQL block can be embedded in any other PL/SQL block, named or unnamed. Figure 10-4 shows all the blocks available in the Oracle server environment.

Block	Use
Anonymous block	An unnamed block, that is independent or embedded within an application.
Procedure/function	A named block that is stored on the Oracle server, can be called by its name, and can take arguments.
Package	A named PL/SQL module that is a group of functions, procedures, and identifiers.
Trigger	A block that is associated with a database table or a view. It is executed when automatically fired by a DML statement.

Figure 10-4 Programming constructs.

COMMENTS

Comments are used to document programs. They are written as part of a program, but they are not executed. In fact, comments are ignored by the PL/SQL engine. It is a good programming practice to add comments to a program, because this helps in readability and debugging of the program. There are two ways to write comments in PL/SQL:

1. To write a single-line comment, two dashes (--) are entered at the beginning of a new line. For example,

```
- -This is a single-line comment.
```

2. To write a multiline comment, comment text is placed between /* and */. A multiline comment can be written on a separate line by itself, or it can be used on a line of code as well. For example,

```
/*  This is a
    multiline comment
    that ends here.  */
```

A programmer can use a comment anywhere in the program.

DATA TYPES

Each constant and variable in the program needs a data type. The data type decides the type of value that can be stored in a variable. PL/SQL has four data types:

1. Scalar.
2. Composite.
3. Reference.
4. LOB.

A scalar data type is not made up of a group of elements. It is atomic in nature. The composite data types are made up of elements or components. PL/SQL supports three composite data types—***records, tables***, and ***varrays***, which are discussed in a later chapter. The reference data types deal with objects, which are briefly introduced in Appendix D.

There are four major categories of scalar data types:

1. Character.
2. Number.
3. Boolean.
4. Date.

Other scalar data types include raw, rowid, and trusted.

Character

Variables with a character data type can store text. The text may include letters, numbers, and special characters. The text in character-type variables can be manipulated with built-in character functions. Character data types include CHAR and VARCHAR2.

CHAR. The CHAR data type is used for fixed-length string values. The allowable string length is between 1 and 32,767. (If you remember, the allowable length in the Oracle database is only 2000.) If you do not specify a length for the variable, the default length is one. Get into the habit of specifying length along with the data type to avoid any errors.

If you are going to declare a variable of the CHAR type in PL/SQL code and that value is to be inserted into a table's column, the limitation on database size is only 2000 characters. You will have to find the substring of that character value to avoid the error message for inserting a character string longer than the length of the column.

If you specify a length of 10 and the assigned value is only five characters long, the value is padded with trailing spaces because of the fixed-length nature of this data type. The CHAR type is not storage efficient, but it is performance efficient.

VARCHAR2. The VARCHAR2 type is used for variable-length string values. Again, the allowable length is between 1 and 32,767. A column in an Oracle database with a VARCHAR2 type, however, can only take 4000 characters, which is smaller than the allowable length for the variable.

Suppose you have two variables with data type of CHAR(20) and VARCHAR2(20), and both are assigned the same value, 'Oracle9i PL/SQL'. The string value is only 15 characters long. The first variable, with CHAR(20), is assigned a value padded with five spaces; the variable with VARCHAR2(20) does not get a string value

padded with spaces. If the values of both variables are compared in a condition for equality, FALSE will be returned.

Other character data types are LONG (32,760 bytes, shorter than VARCHAR2), RAW (32,767 bytes), and LONG RAW (32,760 bytes, shorter than RAW). The RAW data values are neither interpreted nor converted by Oracle.

VARCHAR2 is the most recommended character data type.

Number

PL/SQL has a variety of numeric data types. Whole numbers or integer values can be handled by following data types:

BINARY_INTEGER (approximately $-2^{31} + 1$ to $2^{31} - 1$, or −2 billion to +2 billion)
INTEGER
INT
SMALLINT
POSITIVE (a subtype of BINARY_INTEGER—range, 0 to 2^{31})
NATURAL (a subtype of BINARY_INTEGER—range, 1 to 2^{31})

Similarly, there are various data types for decimal numbers:

NUMBER
DEC (fixed-point number)
DECIMAL (fixed-point number)
NUMERIC (fixed-point number)
FLOAT (floating-point number)
REAL (floating-point number)
DOUBLE PRECISION (floating-point number)

You are familiar with the NUMBER type from the Oracle table's column type. The NUMBER type can be used for fixed-point or floating-point decimal numbers. It provides an accuracy of up to 38 decimal places. When using the NUMBER type, the precision and scale values are provided. The precision of a number is the total number of significant digits in that number, and the scale is the number of significant decimal places. The precision and scale values must be whole-number integers. For example,

NUMBER(*p*, *s*)

where *p* is precision and *s* is scale.

If scale has a value that is negative, positive, or zero, it specifies rounding of the number to the left of the decimal place, to the right of the decimal place, or to the nearest whole number, respectively. If a scale value is not used, no rounding occurs.

Boolean

PL/SQL has a logical data type, Boolean, that is not available in SQL. It is used for Boolean data TRUE, FALSE, or NULL only. These values are not enclosed in single quotation marks like character and date values.

Date

The date type is a special data type that stores date and time information. The date values have a specific format. A user can enter a date in many different formats with the TO_DATE function, but a date is always stored in standard 7-byte format. A date stores the following information:

Century
Year
Month
Day
Hour
Minute
Second

The valid date range is from January 1, 4712 B.C., to December 31, 9999 A.D. The time is stored as the number of seconds past midnight. If the user leaves out the time portion of the data, it defaults to midnight (12:00:00 A.M.).

Various DATE functions are available for date calculations. For example, the SYSDATE function is used to return the system's current date.

OTHER DATA TYPES

NLS

The National Language Support (NLS) data type is for character sets in which multiple bytes are used for character representation. NCHAR and NVARCHAR2 are examples of NLS data types.

LOB

Like Oracle9i, PL/SQL also supports Large Object (LOB) data types to store large values of character, raw, or binary data. The LOB types allow up to 4 gigabytes of data. LOB variables can be given one of the following data types:

- The ***BLOB*** type contains a pointer to the large binary object inside the database.
- The ***CLOB*** type contains a pointer to a large block of single-byte character data of fixed width.

- The ***NCLOB*** type contains a pointer to a large block of multibyte character data of fixed width.
- The ***BFILE*** type contains a pointer to large binary objects in an external operating system file. It would contain the directory name and the filename.

Oracle provides users with a built-in package, DBMS_LOB, to manipulate the contents of LOBs.

VARIABLE DECLARATION

A scalar variable or a constant is declared with a data type and an initial value assignment. The declarations are done in the DECLARE section of the program block. The initial value assignment for a variable is optional unless it has a NOT NULL constraint. The constants and NOT NULL type variables must be initialized. The general syntax is

```
DECLARE
  identifiername [CONSTANT] datatype [NOT NULL] [:= | DEFAULT expression];
```

where *identifiername* is the name of a variable or constant. A CONSTANT is an identifier that must be initialized and the value of which cannot be changed in the program body. A NOT NULL constraint can be used for variables, and such variables must be initialized. The DEFAULT clause, or :=, can be used to initialize a constant or a variable to a value. An expression can be a literal, another variable, or an expression.

The identifiers are named based on rules given previously in this chapter. Different naming conventions can be used. You should declare one variable per line for good readability. For example,

```
DECLARE
  v_number        NUMBER(2);
  v_count         NUMBER(1) := 1;
  v_state         VARCHAR2(2) DEFAULT 'NJ';
  c_pi            CONSTANT NUMBER := 3.14;
  v_invoicedate   DATE DEFAULT SYSDATE;
```

In this example, you see a naming convention that uses *prefix v_* for variables and *prefix c_* for constants.

ANCHORED DECLARATION

PL/SQL uses %TYPE attribute to anchor a variable's data type. Another variable or a column in a table can be used for anchoring. In anchoring, you tell PL/SQL to

use a variable or a column's data type as the data type for another variable in the program. The general syntax is

variablename typeattribute%TYPE [value assignment];

where *typeattribute* is another variable's name or table's column with a table qualifier (e.g., *tablename.columnname*). It is very useful while retrieving a value of a column into a variable with a SELECT query in PL/SQL. For example,

```
DECLARE
  v_num1        NUMBER(3);
  v_num2        v_num1%TYPE;
```

in this example, *v_num1* is declared with a data type NUMBER(3). The next variable, *v_num2*, is declared using the anchoring method and the declaration attribute %TYPE. The variable *v_num2* gets the same data type as *v_num1*.

Two variables can also be declared and assigned data types to match the column's data type. The advantage is that you do not have to cross-reference the data type used in the table. Oracle does that for you. For example,

```
DECLARE
  v_empsal      employee.Salary%TYPE;
  v_deptname    dept.DeptName%TYPE;
```

Suppose you do not use the anchoring method to declare variables, which are assigned values directly from table columns. You can use the DESCRIBE command to list all the data types for columns. Then, you can declare variables in a program with the same types and lengths. This will work just fine. The problem will arise when the column lengths are increased to meet future demands. When you assign values from those columns to variables, VALUE_ERROR will occur—and you will have to go back to all the programs to change the variable's data length! Anchoring definitely is an advantage in such situations.

A %TYPE declaration anchors the data type of one variable based on another variable or column at the time of a PL/SQL block's compilation. If the source or original column's data type is changed, the PL/SQL block must be recompiled to re-anchor all anchored variables.

Nested Anchoring

The %TYPE attribute's use can be nested. For example,

```
DECLARE
  – – source variable v_commission
  v_commission          NUMBER(7, 2);
  – – anchored variable v_total_commission
  v_total_commission    v_commission%TYPE;
  – – nested anchoring variable v_net_commission
  v_net_commission      v_total_commission%TYPE;
```

In this example, the original variable *v_commission* anchors *v_total_commission*, which in turn is used to anchor *v_net_commission.* There is no limit on the number of layers of nesting in anchored declarations.

The source variable for a %TYPE declaration does not have to be in the same block. The variable could be a global variable declared at the SQL*Plus environment, or it could be declared in a block that contains the current block.

NOT NULL Constraint for %TYPE Declarations

If a source variable is declared with a NOT NULL constraint, the %TYPE declaration inherits the NOT NULL constraint from the source, its anchor. The anchored variable must be initialized with a value in the %TYPE declaration.

If the source for a %TYPE declaration is a table's column, the NOT NULL constraint is not inherited by the anchored variable. There is no need to initialize the anchored variable, and it can be assigned a NULL value.

ASSIGNMENT OPERATION

The assignment operation is one of the ways to assign a value to a variable. You have already learned that a variable can be initialized at the time of declaration by using the DEFAULT option or :=. The assignment operation is used in the executable section of the program block to assign a literal, another variable's value, or the result of an expression to a variable. The general syntax is

VariableName := Literal | VariableName | Expression;

For example,

```
v_num1      := 100;
v_num2      := v_num1;
v_sum       := v_num1 + v_num2;
```

In these examples, the assumption is made that three variables have already been declared. The first example assigns 100 to the variable *v_num1*. The second example assigns the value of the variable *v_num1* to the variable *v_num2.* The third example assigns the result of an addition operation on *v_num1* and *v_num2* to the variable *v_sum.*

The following statements are examples of invalid assignment operations and the reasons for their lack of validity:

```
v_count = 10;                 /* Wrong assignment operator, = sign */
v_count * 2 := v_double;      /* Expression cannot be on the left. */
v_num1 := v_num2 :=v_num3;    /* Cannot use two assignments in one statement. */
```

BIND VARIABLES

Bind variables are also known as host variables. These variables are declared in the host SQL*Plus environment and are accessed by a PL/SQL block. Anonymous blocks do not take any arguments, but they can access host variables with a colon prefix (:) and the host variable name. Host variables can be passed to procedures and functions as arguments. A host variable is declared at the SQL> prompt with the SQL*Plus VARIABLE statement. The syntax of a host variable declaration is

VARIABLE variablename datatype

For example,

```
SQL> VARIABLE double NUMBER
```

When a numeric variable is declared with VARIABLE command, precision and scale values are not used. If a VARCHAR2-type variable is declared, length is not used. A host variable's value can be printed in the SQL*Plus environment by using the PRINT command.

Let us put everything together in a program. The program contains a script that includes SQL*Plus statements and a PL/SQL block.

In Figure 10-5, two types of variables are used, a local variable *v_num* and a host variable *g_double*. The host variable *g_double* is declared in SQL*Plus with a VARIABLE statement, and the program block references it with a colon prefix (:). The local variable *v_num* is declared in the declaration section of a program block; there is no need to use the colon prefix with it. The program assigns the value 5 to

```
SQL> VARIABLE g_double NUMBER
SQL> DECLARE
  2       v_num   NUMBER(2);
  3  BEGIN
  4       v_num := 5;
  5       :g_double := v_num * 2;
  6  END;
  7  /

PL/SQL procedure successfully completed.

SQL> PRINT g_double

   G_DOUBLE
- - - - - - - - - - -
            10

SQL>
```

Figure 10-5 Using a host variable in a PL/SQL block.

the local variable *v_num*, doubles it, and stores the result in the host variable *g_double*. Finally, the resulting variable is printed in the host environment with a PRINT statement.

Question: How does a PL/SQL block end?
Answer: It ends with an END and a semicolon on the same line *and* a slash (/) on the next line.

The use of host variables should be minimized in a program block, because they affect performance. Every time a host variable is accessed within the block, the PL/SQL engine must stop to request the host environment for the value of the host variable. The variable's value can be assigned to a local variable to minimize calls to the host.

SUBSTITUTION VARIABLES IN PL/SQL

PL/SQL does not have any input capabilities in terms of having an input statement. There are no explicit input/output (I/O) statements, but substitution variables of SQL are available in PL/SQL. Substitution variables have limitations, which become apparent in a loop.

Let us rewrite the program code in Figure 10-5 to the one in Figure 10-6. When the code in Figure 10-6 is executed, a standard prompt for *p_num* appears on the screen for users to type in a value for it. As you see in the example, there is no need

```
SQL> VARIABLE g_double NUMBER
SQL> DECLARE
  2       v_num   NUMBER(2);
  3  BEGIN
  4       v_num  :=  &p_num;
  5       :g_double  :=  v_num  * 2;
  6  END;
  7  /
Enter value for p_num: 10

PL/SQL procedure successfully completed.

SQL> PRINT g_double

   G_DOUBLE
-----------
           20

SQL>
```

Figure 10-6 Local, host, and substitution variables.

to declare substitution variables in the program block. The value of the bind/host variable is printed with the PRINT command. The value of the local variable *v_num* cannot be printed with the PRINT command, however, because the scope of a local variable is within the block where it is declared.

When substitution variable is used in a program, the output contains lines showing how the substitution was done. You can suppress those lines by using the SET VERIFY OFF command before running the program.

PRINTING IN PL/SQL

There is no explicit output statement in PL/SQL. Oracle does have a built-in package called DBMS_OUTPUT with the procedure PUT_LINE to print. An environment variable named SERVEROUTPUT must be toggled ON to view output from it.

The DBMS_OUTPUT is the most frequently used package because of its capabilities to get lines from a file and to put lines into the buffer. The PUT_LINE procedure displays information passed to the buffer and puts a new-line marker at the end. For example,

```
DBMS_OUTPUT.PUT_LINE ('This line will be displayed');
```

The maximum size of the buffer is 1 megabyte. The following command enables you to view information from the buffer by using the DBMS_OUTPUT package and also sets the buffer size to the number of bytes specified:

```
SET SERVEROUTPUT ON [on size 10000];
```

The PL/SQL block in Figure 10-7 shows the use of DBMS_OUTPUT.PUT_LINE. Another procedure, DBMS_OUTPUT.PUT, also performs the same task of

```
SQL> VARIABLE NUM NUMBER
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2       DOUBLE   NUMBER;
  3  BEGIN
  4       :NUM  :=  5;
  5       DOUBLE  :=  :NUM * 2;
  6          DBMS_OUTPUT.PUT_LINE ((‘DOUBLE OF ’ ||
  7          TO_CHAR(:NUM)  ||  ’ IS ’  ||       TO_CHAR(DOUBLE));
  8  END;
  9  /
DOUBLE OF 5 IS 10

PL/SQL procedure successfully completed.

SQL>
```

Figure 10-7 DBMS_OUTPUT.PUT_LINE.

displaying information from the buffer, but it does not put a new-line marker at the end. If there is another DBMS_OUTPUT.PUT or DBMS_OUTPUT.PUT_LINE statement following that statement, its output will be displayed on the same line.

ARITHMETIC OPERATORS

Five standard arithmetic operators are available in PL/SQL for calculations. If more than one operator exists in an arithmetic expression, the following order of precedence is used:

- Exponentiation is performed first, multiplication and division are performed next, and addition and subtraction are performed last.
- If more than one operator of the same priority is present, they are performed from left to right.
- Whatever is in parentheses is performed first.

Figure 10-8 shows arithmetic operators and their use.

Arithmetic Operator	Use
+	Addition
–	Subtraction and negation
*	Multiplication
/	Division
**	Exponentiation

Figure 10-8 Arithmetic operators.

Question: What is the answer from the following expression?

```
–2 + 3 * (10 – 2 * 3)/6
```

Answer: 0 (The operations within the parentheses are performed first, with multiplication followed by subtraction. The result from within the parentheses is multiplied by 3, and that result is divided by 6. Finally, the result is added to −2)

In this chapter, you learned the basics of PL/SQL. In the next chapter, you will learn about three programming control structures: sequence, selection, and looping. (You already know one of them. All examples in this chapter were based on sequence

structure.) You will also learn to interface with the Oracle server by embedding SQL statements in PL/SQL program blocks.

IN A NUTSHELL . . .

- PL/SQL (Programming Language extension to Structured Query Language) is Oracle's proprietary language.
- PL/SQL contains features of modern languages, such as data encapsulation, error handling, information hiding, and object-oriented programming (OOP).
- PL/SQL is a block-structured language. A block is of two types: anonymous block and subprogram (procedures and functions).
- A program code contains reserved words, user-defined identifiers, delimiters, comments, and literals.
- A program block consists of three sections: declaration, executable, and exception handling.
- A program uses variables and constants to hold values. A variable's value can be changed, but a constant's value remains the same throughout the execution of the program.
- A variable is declared in the declarative section with a scalar data type. The standard data types are number, character, Boolean, and date. PL/SQL also supports other LOB (Large Object) data types.
- A declaration attribute %TYPE is used to anchor a variable with another variable's data type or with a table column's data type.
- A bind variable, or host variable, is global to a PL/SQL block. An anonymous block refers to it with a colon prefix (:). A bind variable is declared with the VARIABLE statement and is printed with the PRINT statement in the SQL*Plus environment.
- An assignment statement is used in the executable section to assign a literal, a variable's value, or the result of an expression to a variable. An assignment uses the := operator.
- PL/SQL does not have an input statement, but substitution variables are allowed in a block to assign a value to a variable.
- A built-in Oracle package and its procedure DBMS_OUTPUT.PUT_LINE are used to output information. An environment variable SERVEROUTPUT must be set to ON before using it.
- Arithmetic operators (+, −, *, /, and **) are used in mathematical expressions. The operations follow rules of precedence for evaluating expressions with more than one operator.

EXERCISE QUESTIONS

True/False:

1. PL/SQL is a nonprocedural language developed by Oracle Corporation.
2. The SQL language has built-in error checking and error handling, but PL/SQL does not.
3. Three types of variables used in a PL/SQL program are local, host, and substitution.
4. A variable with a NOT NULL constraint and a constant must be initialized with a value at the declaration time.
5. An assignment statement is used in the executable section to assign value to a variable or a constant.
6. In a declaration with the %TYPE attribute, the source is either a variable or a column in a table.
7. If the source variable is declared with a NOT NULL constraint, the anchored variable inherits the same constraint.
8. If the source column in a table has a NOT NULL constraint, the anchored variable also gets the NOT NULL constraint.
9. Exponentiation is performed before addition in an expression without any parentheses.
10. The declaration and executable sections are mandatory in a PL/SQL block.
11. A bind variable is declared with a VARIABLE command at the SQL> prompt.
12. A PL/SQL block must contain BEGIN and END key words.
13. A substitution variable is declared under the DECLARE section of a PL/SQL block.
14. A bind variable is used in a PL/SQL block with a colon prefix (:).
15. A local variable used in a PL/SQL block can be printed with the PRINT command.

Answer the Following Questions:

1. State differences between Oracle's SQL and PL/SQL languages.
2. What are the two types of blocks in PL/SQL? What are the differences between them?
3. Name four standard scalar data types used in PL/SQL. When is each type used for variables?
4. Name three types of variables used in PL/SQL. Where are they declared? Give a sample declaration of each.
5. Give any differences and similarities between := and DEFAULT.
6. What are advantages of the %TYPE attribute in a variable declaration?
7. State the rules of precedence used in arithmetic operations.
8. How will you declare a bind variable named *v_count*, use it in a PL/SQL block, and print its value?
9. Declare a variable *v_val*, which may not have a null value. Assign a value to *v_val* with a substitution variable, and print that value.
10. How do you run a PL/SQL block?

LAB ACTIVITY

1. Create a program script that uses a PL/SQL anonymous block to perform the following: Use a host variable AREA to store the result. Declare a local variable RADIUS with numeric data type. Declare a constant PI with value 3.14. Assign a value to the variable RADIUS by using a substitution variable. Calculate area of a circle by using the formula

   AREA = PI * RADIUS * RADIUS

   Then, print the result in SQL*Plus.
2. Write a PL/SQL block to find the square, cube, and double of a number inputted with a substitution variable, and print the results using the built-in package DBMS_OUTPUT.
3. Write a PL/SQL block to swap the values of two variables. Print the variables before and after swapping.
4. Write a PL/SQL program to input hours and rate. Find gross pay and net pay with a tax rate is 28%. Print your results. (No need to perform overtime calculations.)
5. Write a PL/SQL program with two variables, for the first name and the last name. Print the full name with the last name and first name separated by comma and a space.

11

More on PL/SQL: Control Structures and Embedded SQL

IN THIS CHAPTER . . .

- You will learn about various programming control structures in PL/SQL.
- Different decision-making statements based on various options are covered.
- Looping statements are introduced to perform a set of statements repetitively.
- SQL statements are embedded within a PL/SQL block to interact with the Oracle server.

In the previous chapter, you learned the basics of the PL/SQL programming language. You are now able to write simple programs using local, host, and substitution variables; can perform simple calculations by using assignment statements; and know how to use Oracle's built-in package DBMS_OUTPUT.PUT_LINE in program blocks to display results from the buffer. The sample programs and lab exercises have a series of statements that are executed from the beginning to the end in a linear fashion. When an anonymous block is executed, the code is sent to the PL/SQL engine for compilation. In this chapter, you will see the use of different control structures employed in a high-level programming language.

In the last chapter, you saw how to write PL/SQL programs independent of a database. We start with more independent PL/SQL programs, and then show the actual use of PL/SQL to interact with the Oracle database. A PL/SQL program block "talks" to the Oracle database by embedding SQL statements in its executable section.

CONTROL STRUCTURES

In a procedural language like PL/SQL, there are three basic programming control structures:

1. In a ***sequential structure***, a series of instructions are performed from the beginning to the end in a linear order. None of the instructions is skipped, and none of the instructions is repeated.
2. The ***selection structure*** is also known as a ***decision structure*** or an ***IF-structure***. It involves conditions with a TRUE or FALSE outcome. Based on the outcome, one of the options is performed, and the other option is skipped. Selection statements are also available for multiple options.
3. In a ***looping structure***, a series of instructions is performed repeatedly. There are different looping statements appropriate for a variety of situations. A programmer has to write a loop correctly to make it perform a specific number of times.

We have already covered sequential statements in the previous chapter. In this chapter, we will talk about the selection and the looping structures. In actuality, a program may utilize one or a combination of all control structures.

Selection Structure

There are three selection or conditional statements in PL/SQL. Relational operators, logical operators, and other special operators are used to create Boolean expressions or conditions. The tables in Figures 11-1, 11-2, and 11-3 are repeated from Chapter 5 for reading convenience. Figure 11-1 shows the use of relational operators, which constitute simple conditions. Figure 11-2 explains the use of logical operators in compound conditions. Figure 11-3 shows a truth table for the AND, OR, and NOT operators. The AND and OR operators are binary operators, because they work on two conditions. The NOT operator is a unary operator, because it works on a single condition.

Relational Operator	Meaning
=	Equal to
<> or !=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Figure 11-1 Relational operators.

Logical Operator	Meaning
AND	Returns TRUE only if both conditions are true.
OR	Returns TRUE if one or both conditions are true.
NOT	Returns TRUE if the condition is false.

Figure 11-2 Logical operators.

AND	OR	NOT
TRUE AND TRUE = TRUE	TRUE OR TRUE = TRUE	NOT TRUE = FALSE
TRUE AND FALSE = FALSE	TRUE OR FALSE = TRUE	NOT FALSE = TRUE
FALSE AND TRUE = FALSE	FALSE OR TRUE = TRUE	NOT NULL = NULL
FALSE AND FALSE = FALSE	FALSE OR FALSE = FALSE	
NULL AND TRUE = NULL	NULL OR TRUE = TRUE	
NULL AND FALSE = FALSE	NULL OR FALSE = NULL	
NULL AND NULL = NULL	NULL OR NULL = NULL	

Figure 11-3 Truth tables for AND, OR, and NOT operators.

Other special operators (IS NULL, IN, LIKE, and BETWEEN . . . AND) discussed in the SQL section are also available in PL/SQL. PL/SQL has five conditional or selection statements available for decision making:

1. IF . . .THEN . . . END IF.
2. IF . . . THEN . . . ELSE . . . END IF.
3. IF . . . THEN . . . ELSIF . . . END IF.
4. CASE . . . END CASE.
5. Searched CASE.

IF . . . THEN . . . END IF. The IF . . . THEN . . . END IF statement is also known as a simple IF statement. A simple IF statement performs action statements if the result of the condition is TRUE. If the condition is FALSE, no action is performed, and the program continues with the next statement in the block. The general syntax is

```
IF condition(s) THEN
   Action statements
END IF;
```

For example, Figure 11-4 shows a simple IF statement with an output statement, which will be performed if the day entered is 'SUNDAY'. If the condition is false, the statement is skipped. In this example, notice the use of the relational operator equals (=) in a Boolean condition and of the assignment operator (:=) in the action assignment statement.

```
SQL> DECLARE
  2       V_DAY    VARCHAR2(9)  :=  '&DAY';
  3  BEGIN
  4       IF (V_DAY = 'SUNDAY') THEN
  5         DBMS_OUTPUT.PUT_LINE('SUNDAY IS A HOLIDAY!');
  6       END IF;
  7  END;
  8  /
Enter value for day: SUNDAY

SUNDAY IS A HOLIDAY

PL/SQL procedure successfully completed.

SQL> /
Enter value for day: MONDAY

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-4 Simple IF statement.

You must have noticed the ***indentation*** in the program code. All program statements can start in the first column. In fact, you can write more than one statement on one line with the appropriate punctuation mark (;) separating them. Such programming practice, however, can make your program difficult to read. In turn, it is a good practice to indent statements within a block or a compound statement, because it makes your program more readable. A program will work just the same without indenting statements or without adding a ***comment*** to it, but good programming practices make everybody's life easier.

IF . . . THEN . . . ELSE . . . END IF. The IF . . . THEN . . . ELSE . . . END IF statement is an extension of the simple IF statement. It provides action statements for the TRUE outcome as well as for the FALSE outcome. The general syntax is

```
IF condition(s) THEN
   Action statements 1
ELSE
   Action statements 2
END IF;
```

If the condition's outcome is TRUE, *action statements 1* are performed. If the outcome is FALSE, *action statements 2* are performed. One set of statements is skipped in any case.

For an example, see Figure 11-5. If the entered age is 18 or older, age is displayed with string ADULT; otherwise, age is displayed with string MINOR.

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2       V_AGE   NUMBER(2) := '&AGE';
  3  BEGIN
  4       IF (V_AGE >=18) THEN
  5          DBMS_OUTPUT.PUT_LINE('AGE: ' || V_AGE || ' - ADULT');
  6       ELSE
  7          DBMS_OUTPUT.PUT_LINE('AGE: ' || V_AGE || ' - MINOR');
  8       END IF;
  9  END;
 10  /
Enter value for age: 21
AGE: 21 - ADULT

PL/SQL procedure successfully completed.

SQL> /
Enter value for age: 12
AGE: 12 - MINOR

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-5 IF. . . ELSE. . . END IF statement.

IF . . . THEN . . . ELSIF . . . END IF. The IF . . . THEN . . . ELSIF . . . END IF statement is an extension to the previous statement. When you have many alternatives/options, you can use previously explained statements, but the ELSIF alternative is more efficient than the other two. The DECODE function in SQL is not allowed in PL/SQL, and the IF . . . THEN . . . ELS . . . END IF statement is not allowed in SQL. The general syntax is

```
IF condition(s)1 THEN
   Action statements 1
ELSIF condition(s)2 THEN
   Action statements 2
. . .
ELSIF condition(s)N THEN
   Action statement N
[ELSE
   Else Action statements]
END IF;
```

Notice the word ELSIF, which does not have the last *E* in ELSE. ELSIF is a single word, but END IF uses two words. For example, let us revisit the DECODE function example of Chapter 6 (see Figure 11-6). Figure 11-7 shows the ELSIF equivalent of the DECODE function.

The same statement can be written with a simple IF statement, though less efficiently. You will need five simple IF statements to accomplish the same task as that performed by a single compound ELSIF statement. Let us take another example

```
SQL> SELECT LName, FName,
  2       DECODE (PositionId, 1, Salary*1.2,
  3                   2, Salary*1.15,
  4                   3, Salary*1.1,
  5                   4, Salary*1.05,
  6                      Salary) "New Salary"
  7  FROM employee;

LNAME                FNAME                New Salary
---------------      ---------------      ----------
Smith                John                     318000
Houston              Larry                    172500
Roberts              Sandi                     86250
McCall               Alex                      73150
Dev                  Derek                     92000
Shaw                 Jinku                     24500
Garner               Stanley                   51750
Chen                 Sunny                     36750
Viquez               Heillyn

9 rows selected.

SQL>
```

Figure 11-6 DECODE function.

```
SQL> DECLARE
  2       v_pos       NUMBER(1) := &Position;
  3  BEGIN
  4       IF v_pos=1 THEN
  5          DBMS_OUTPUT.PUT_LINE('20% increase');
  6       ELSIF v_pos=2 THEN
  7          DBMS_OUTPUT.PUT_LINE('15% increase');
  8       ELSIF v_pos=3 THEN
  9          DBMS_OUTPUT.PUT_LINE('10% increase');
 10       ELSIF v_pos=4 THEN
 11          DBMS_OUTPUT.PUT_LINE('5%  increase');
 12       ELSE
 13          DBMS_OUTPUT.PUT_LINE('No  increase');
 14       END IF;
 15  END;
 16  /
Enter value for position: 2
15% increase

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-7 ELSIF statement.

with compound conditions. First, we will use a simple IF statement (see Fig. 11-8), and then, we will rewrite it by using ELSIF (Figure 11-9).

The example here assigns a grade of A, B, C, D, or F based on *v_score*. We are assuming that the score is within the range of 0 to 100. You will need five simple IF statements with total of 10 conditions or two conditions per each statement. Now, suppose the value of *v_score* is 95. The first statement's condition is TRUE, so *v_grade* will be assigned 'A'. Because all simple IF statements are independent statements, the execution will continue with the next IF, and so on. There is no other TRUE condition for *v_score* equal to 95, so *v_grade* will be 'A' after execution of all five IF statements. This slows down your program's execution. The ELSIF, on the other hand, is one compound statement, and it stops as soon as a match is found. Let us see how the ELSIF statement looks.

```
SQL> DECLARE
  2      S    NUMBER(3) := &SCORE;
  3      GRADE    CHAR;
  4  BEGIN
  5      IF S >= 90 AND S <= 100 THEN
  6         GRADE := 'A';
  7      END IF;
  8      IF S >= 80 AND S <= 89 THEN
  9         GRADE := 'A';
 10      END IF;
 11      IF S >= 70 AND S <= 79 THEN
 12         GRADE := 'C';
 13      END IF;
 14      IF S >= 60 AND S <= 69 THEN
 15         GRADE := 'D';
 16      END IF;
 17      IF S >= 0 AND S <= 59 THEN
 18         GRADE := 'F';
 19      END IF;
 20      IF S < 0 AND S > 100 THEN
 21         GRADE := 'U';
 22      END IF;
 23      DBMS_OUTPUT.PUT_LINE('SCORE IS ' || TO_CHAR(S));
 24      DBMS_OUTPUT.PUT_LINE('GRADE IS ' || GRADE);
 25  END;
 26  /
Enter value for score: 93
SCORE IS 93
GRADE IS A

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-8 Simple IF with multiple conditions.

```
SQL> DECLARE
  2       S   NUMBER(3) := &SCORE;
  3       GRADE   CHAR;
  4  BEGIN
  5       IF S >= 90 AND S <= 100 THEN
  6          GRADE := 'A';
  7       ELSIF S >= 80 AND S <= 89 THEN
  8          GRADE := 'B';
  9       ELSIF S >= 70 THEN
 10          GRADE := 'C';
 11       ELSIF S >= 60 THEN
 12          GRADE := 'D';
 13       ELSIF S >= 0 THEN
 14          GRADE := 'F';
 15       ELSIF S < 0 AND S > 100 THEN
 16          GRADE := 'U';
 17       END IF;
 18       DBMS_OUTPUT.PUT_LINE('SCORE IS ' || TO_CHAR(S));
 19       DBMS_OUTPUT.PUT_LINE('GRADE IS ' || GRADE);
 20  END;
 21  /
Enter value for score: 77
SCORE IS 77
GRADE IS C

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-9 ELSIF statement.

The ELSIF statement reduces the number of conditions from 10 to 5 and the number of statements from five to one. Now, let us consider the same value, 95, as before. The condition is TRUE in the first IF clause, and *v_grade* is assigned value 'A'. The statement will not continue down anymore, because it will not enter the ELSIF part. The rest of the statement is ignored, thus making ELSIF more efficient than its counterpart (see Fig. 11-9). In this example, there is an added ELSIF statement to check for invalid scores (below 0 as well as above 100), which result in an undefined ('U') grade.

CASE. The CASE statement is an alternative to the IF . . . THEN . . . ELSIF . . . END IF statement. The CASE statement begins with key word CASE and ends with the key words END CASE. The body of the CASE statement contains WHEN clauses, with values or conditions, and action statements. When a WHEN clause's value/condition evaluates to TRUE, its action statements are executed. The general syntax is

```
CASE [variable_name]
  WHEN value1|condition1 THEN action_statement1;
  WHEN value2\condition2 THEN action_statement2;
  ...
```

```
    WHEN valueN|conditionN THEN action_statementN;
    ELSE action_statement;
END CASE;
```

Searched CASE. A statement with a value is known as a *CASE statement*, and a statement with conditions is known as a *searched CASE* statement. A CASE statement uses *variable_name* as a selector, but a searched CASE does not use *variable_name* as a selector. Figure 11-10 is an example of a CASE statement that evaluates if a number is odd or even. Figure 11-11 rewrites the same solution for a searched CASE statement.

```
SQL> DECLARE /* Example of Case */
  2      V_NUM NUMBER := &ANY_NUM;
  3      V_RES NUMBER;
  4  BEGIN
  5      V_RES := MOD(V_NUM, 2);
  6      CASE V_RES
  7          WHEN 0 THEN DBMS_OUTPUT.PUT_LINE(V_NUM || ' IS EVEN');
  8          ELSE DBMS_OUTPUT.PUT_LINE(V_NUM || ' IS ODD');
  9      END CASE;
 10  END;
 11  /
Enter value for any_num: 5
5 IS ODD

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-10 CASE statement.

```
SQL> DECLARE /* Example of Searched Case */
  2      V_NUM NUMBER := &ANY_NUM;
  3  BEGIN
  4      CASE
  5      WHEN MOD(V_NUM, 2)=0 THEN
  6        DBMS_OUTPUT.PUT_LINE(V_NUM || ' IS EVEN');
  7      ELSE
  8        DBMS_OUTPUT.PUT_LINE(V_NUM || ' IS ODD');
  9      END CASE;
 10  END;
 11  /
Enter value for any_num: 5
5 IS ODD

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-11 Searched CASE statement.

Nested IF. The nested IF statement contains an IF statement within another IF statement. If the condition in the outer IF statement is TRUE, the inner IF statement is performed. Any IF statement with a compound condition can be written as a nested IF statement. For example, the program segment in Figure 11-12 assigns an insurance surcharge based on an individual's gender and age. There are four categories:

1. Male 25 or over.
2. Male under 25.
3. Female 25 or over.
4. Female under 25.

```
SQL> DECLARE
  2      V_GENDER CHAR := '&SEX';
  3      V_AGE NUMBER(2) := '&AGE';
  4      V_CHARGE NUMBER(3,2);
  5  BEGIN
  6      IF (V_GENDER = 'M' AND V_AGE >= 25) THEN
  7         V_CHARGE := 0.05;
  8      END IF;
  9      IF (V_GENDER = 'M' AND V_AGE < 25) THEN
 10         V_CHARGE := 0.10;
 11      END IF;
 12      IF (V_GENDER = 'F' AND V_AGE >= 25) THEN
 13         V_CHARGE := 0.03;
 14      END IF;
 15      IF (V_GENDER = 'F' AND V_AGE < 25) THEN
 16         V_CHARGE := 0.06;
 17      END IF;
 18      DBMS_OUTPUT.PUT_LINE('GENDER: ' || V_GENDER);
 19      DBMS_OUTPUT.PUT_LINE('AGE: ' || TO_CHAR(V_AGE));
 20      DBMS_OUTPUT.PUT_LINE('SURCHARGE: ' || TO_CHAR(V_CHARGE));
 21  END;
 22  /
Enter value for sex: F
Enter value for age: 18
GENDER: F
AGE: 18
SURCHARGE: .06

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-12 Simple IF with multiple conditions.

Now, we will rewrite the code done with a simple IF in Figure 11-12 by using nested IF statements (see Fig. 11-13). Again, remember that the nested IF statement will make the code more efficient than the simple IF version. The THEN portion of the outer IF calculates the insurance surcharge for male individuals, and the ELSE portion calculates the same for the female individuals. The inner IF statements in each portion check for the age.

Looping Structure

Looping means iterations. A loop repeats a statement or a series of statements a specific number of times, as defined by the programmer. You would use a loop to repeat a series of statements many times rather than typing the same statements many

```
SQL> DECLARE
  2        V_GENDER  CHAR  := '&SEX';
  3        V_AGE NUMBER(2)  := '&AGE';
  4       V_CHARGE NUMBER(3,2);
  5  BEGIN
  6  IF (V_GENDER = 'M') THEN      /* MALE */
  7     IF   (V_AGE >= 25) THEN
  8          V_CHARGE  := 0.05;
  9     ELSE
 10          V_CHARGE := 0.10;
 11  END IF;
 12  ELSE      /* FEMALE */
 13     IF   (V_AGE >= 25) THEN
 14          V_CHARGE  := 0.03;
 15     ELSE
 16      V_CHARGE  := 0.06;
 17   END IF;
 18  END IF;
 19  DBMS_OUTPUT.PUT_LINE('GENDER: ' || V_GENDER);
 20  DBMS_OUTPUT.PUT_LINE('AGE: ' || TO_CHAR(V_AGE));
 21  DBMS_OUTPUT.PUT_LINE('SURCHARGE: '||TO_CHAR(V_CHARGE));
 22 END;
 23 /
Enter value for sex: F
Enter value for age: 18
GENDER: F
AGE: 18
SURCHARGE: .06

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-13 Nested IF statement.

times. Three types of looping statements are available in PL/SQL:

1. Basic loop.
2. WHILE loop.
3. FOR loop.

Each loop has different syntax, and each works somewhat differently.

Basic loop. A basic loop is a loop that is performed repeatedly. Once a loop is entered, all statements in the loop are performed. When the bottom of the loop is reached, control shifts back to the top of the loop. The loop will continue infinitely. An infinite loop, or a "never-ending loop," is a logical error in programming. The only way to terminate a basic loop is by adding an EXIT statement inside the loop. The general syntax is

```
LOOP
   Looping statement1;
   Looping statement2;
   . . .
   Looping statementN;
   EXIT [WHEN condition];
END LOOP;
```

The EXIT statement in a loop could be an independent statement, or it could be part of an IF statement. For example,

```
IF v_count > 10 THEN
   EXIT;
END IF;
```

You can also add a condition with the optional WHEN clause that will end the loop when the condition becomes true. For example,

```
EXIT WHEN v_count > 10;
```

The condition is not checked at the top of the loop, but it is checked inside the body of the loop. The basic loop is performed at least once, because the condition is tested after entering the body of the loop. Such a loop is also called a post-test loop.

The example shown in Figure 11-14 uses a ***counter*** to control the number of loop executions. There are three necessary statements in a counter-controlled loop. The counter must be initialized, the value of the counter must change within the loop (increment or decrement), and a proper condition must exist in the loop. If value of the counter is not changed inside the loop, it will result in an infinite loop. The initial value, the increment/decrement, and the condition control the total number of loop executions.

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2       V_COUNT       NUMBER(2);
  3       V_SUM         NUMBER(2) := 0;
  4       V_AVG         NUMBER(3,1);
  5  BEGIN
  6  V_COUNT := 1;      /* COUNTER INITIALIZED */
  7  LOOP
  8      V_SUM := V_SUM + V_COUNT;
  9      V_COUNT := V_COUNT + 1;      /* COUNTER INCREMENTED */
 10      EXIT WHEN V_COUNT > 10;      /* CONDITION   */
 11  END LOOP;
 12  V_AVG := V_SUM / (V_COUNT -1);
 13  DBMS_OUTPUT.PUT_LINE('AVERAGE OF 1 TO 10 IS '
 14                                  || TO_CHAR(V_AVG));
 15 END;
 16 /
AVERAGE OF 1 TO 10 IS 5.5

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-14 Counter-controlled basic loop.

Question: In a basic loop, if the counter is initialized to one and is incremented within the loop by two, and if the condition at the bottom of the loop body is EXIT WHEN the counter is less than 10, how many times the loop is performed?

Answer: One time (the loop is performed once and the first check of condition returns true, so the loop ends). Tricky, isn't it?

Question: In a basic loop, the counter is initialized to zero and is incremented within the loop by one. How many times will the loop be performed if the condition at the bottom of the loop body is EXIT WHEN the counter equals five?

Answer: Five times (for counter values equal to 0, 1, 2, 3, and 4).

Question: In a basic loop, the counter is initialized to 10 and is incremented within the loop by one. How many times will the loop be performed if the condition at the bottom of the loop body is EXIT WHEN the counter equals 10?

Answer: The loop is infinite (the condition will never become true).

WHILE loop. The WHILE loop is an alternative to the basic loop and is performed as long as the condition is true. It terminates when the condition becomes false. If the condition is false at the beginning of the loop, the loop is not performed at all. The WHILE loop does not need an EXIT statement to terminate. The general syntax is

```
WHILE condition LOOP
  Looping statement1;
  Looping statement2;
  . . .
  Looping statement n;
END LOOP;
```

In Figure 11-15, you see the same average program of Figure 11-14 rewritten with the WHILE loop. There are obvious differences between the basic loop and the WHILE loop. Figure 11-16 explains the differences between them.

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2       V_COUNT      NUMBER(2);
  3       V_SUM        NUMBER(2) := 0;
  4       V_AVG        NUMBER(3,1);
  5  BEGIN
  6    V_COUNT := 1;   /* COUNTER INITIALIZED */
  7    WHILE V_COUNT <= 10 LOOP      /* CONDITION */
  8       V_SUM := V_SUM + V_COUNT;
  9       V_COUNT := V_COUNT + 1;     /* COUNTER INCREMENTED */
 10    END LOOP;
 11    V_AVG := V_SUM / (V_COUNT -1);
 12    DBMS_OUTPUT.PUT_LINE
 13            ('AVERAGE OF 1 TO 10 IS ' || TO_CHAR(V_AVG));
 14 END;
 15 /
AVERAGE OF 1 TO 10 IS 5.5

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-15 Counter-controlled WHILE loop.

Basic Loop	WHILE Loop
It is performed as long as the condition is false.	It is performed as long as the condition is true.
It tests the condition inside the loop (post-test loop).	It checks condition before entering the loop (pretest loop).
It is performed at least one time.	It is performed zero or more times.
It needs the EXIT statement to terminate.	There is no need for an EXIT statement.

Figure 11-16 Differences between a basic loop and a WHILE loop.

FOR loop. The FOR loop is the simplest loop you can write. Unlike the basic and WHILE loops, you do not have to initialize, test, and increment/decrement the loop control variable separately. You do it in the loop's header. There is no need to use an EXIT statement, and the counter need not be declared. The counter used in the loop is implicitly declared as an integer, and it is destroyed on the loop's termination. The counter may not be used within the loop body in an assignment statement as a target variable. The general syntax is

```
FOR counter IN [REVERSE] lower..upper LOOP
  Looping statement1
  Looping statement2
  . . .
  Looping statementN
END LOOP;
```

The counter varies from the lower value to the upper value, incrementing by one with every loop execution. The loop can also be used with the counter starting at a higher value and decrementing by one with every loop execution. The key word REVERSE is used for varying the counter in the reverse order, or from a higher to a lower value.

The program in Figure 11-17 does not declare *v_count*, and there is no condition or explicit statement to change the counter's value. The same program with the counter's value in reverse order will only change by one line. The FOR statement will look like

```
FOR v_count IN REVERSE 1..10 LOOP
```

```
SQL> DECLARE
  2       V_COUNT       NUMBER(2);
  3       V_SUM         NUMBER(2) := 0;
  4       V_AVG         NUMBER(3,1);
  5  BEGIN
  6  FOR V_COUNT IN 1..10 LOOP
  7     V_SUM := V_SUM + V_COUNT;
  8  END LOOP;
  9  V_AVG := V_SUM / 10;
 10  DBMS_OUTPUT.PUT_LINE
 11     ('AVERAGE OF 1 TO 10 IS ' || TO_CHAR(V_AVG));
 12 END;
 13 /
AVERAGE OF 1 TO 10 IS 5.5

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-17 FOR loop.

One important point about the loop-control variable is that it is declared, incremented/decremented, and destroyed implicitly, but its value cannot be changed explicitly within the FOR loop's body. It cannot be a target variable in an assignment statement.

Nested loops. You can use a loop within another loop. Loops can be nested to many levels. When the inner loop ends, it does not automatically end the outer loop enclosing it. You can end an outer loop from within the inner loop by labeling each loop and then using the EXIT statement. *EXIT WHEN* exits the current loop, but *EXIT out_loop WHEN* also exits the outer loop. The loop labels use the same naming rules as those used for identifiers. The loops are labeled before the key word LOOP on the same line or on a separate line. The loop label is enclosed within two pairs of angle brackets (≪ and ≫). For example,

```
<<out_loop>>
LOOP
  . . .
    EXIT WHEN condition;
  <<in_loop>>
    LOOP
      . . .
      EXIT out_loop WHEN condition;   /* exits out_loop */
      EXIT WHEN condition;            /*exits in_loops */
      . . .
    END LOOP in_loop;                 /* label optional here */
    . . .
END LOOP out_loop;                    /*label optional here */
```

NESTED BLOCKS

PL/SQL block may contain another PL/SQL block; in other words, PL/SQL blocks can be nested. The execution starts with the outer block and continues with the inner block. The variables declared in the outer block are global to the inner block, and they are accessible in the inner block. The variables declared in the inner block, however, are not accessible in the outer block. For example,

```
DECLARE                   /* Outer block starts here.*/
  Var1 NUMBER;            /* known to outer and inner*/
BEGIN
    . . .                 /* can use Var1 here */
  DECLARE                 /*Inner block starts here.*/
      Var2 NUMBER;        /* known to inner block */
  BEGIN
      . . .               /* can use Var1 and Var2 here */
    END;                  /* Inner block ends here.*/
    . . .                 /* can use Var1 here */
END;                      /* Outer block ends here.*/
```

SQL IN PL/SQL

The PL/SQL statements have control structures for calculations, decision making, and iterations. You need to use SQL to interface with the Oracle database. When changes are necessary in the database, SQL can be used to retrieve and change information. PL/SQL supports all Data Manipulation Language (DML) statements, such as INSERT, UPDATE, and DELETE. It also supports the Transaction Control Language statements ROLLBACK, COMMIT, and SAVEPOINT. You can retrieve data using the data retrieval statement SELECT. A row of data can be used to assign values to variables. More than one row can be retrieved and processed using cursors (covered in the next chapter). PL/SQL statements can use single-row functions, but group functions are not available for PL/SQL statements. SQL statements in the PL/SQL block, however, can still utilize those group functions.

PL/SQL does not support Data Definition Language (DDL) statements, such as CREATE, ALTER, and DROP. The Data Control Language (DCL) statements GRANT and REVOKE also are not available in PL/SQL.

SELECT Statement in PL/SQL

The SELECT statement retrieves data from Oracle tables. The syntax of SELECT is different in PL/SQL, however, because it is used to retrieve values from a row into a list of variables or into a PL/SQL record. The general syntax is

```
SELECT columnnames
INTO variablenames / RecordName
FROM tablename
WHERE condition;
```

where *columnnames* must contain at least one column and may include arithmetic or string expressions, single-row functions, and group functions. *Variablenames* must contain a list of local or host variables to hold values retrieved by the SELECT clause. The variables are declared either at the SQL* Plus prompt or locally under the DECLARE section (see Fig. 11-18). The *recordname* is a PL/SQL record (covered in the next chapter). All the features of SELECT in SQL are available with an added mandatory INTO clause.

The INTO clause must contain one variable for each value retrieved from the table. The order and data type of the columns and variables must correspond. The SELECT . . . INTO statement must return one and only one row. It is your responsibility to code a statement that returns one row of data. If no rows are returned, the standard exception (error condition) NO_DATA_FOUND occurs. If more than one row are retrieved, the TOO_MANY_ROWS exception occurs. You will learn more about exceptions and exception handling in the next chapter.

In Figure 11-18, a few columns from a row of the EMPLOYEE table are retrieved into a series of variables. The variables can be declared with data types, but more appropriately, attribute %TYPE is used to avoid any data-type mismatches.

The SQL statement in PL/SQL ends with a semicolon (;). The INTO clause is mandatory in a SELECT statement when used in a PL/SQL block. Figure 11-19

```
SQL>  DECLARE
   2        V_LAST          EMPLOYEE.LNAME%TYPE;
   3        V_FIRST         EMPLOYEE.FNAME%TYPE;
   4        V_SAL           EMPLOYEE.SALARY%TYPE;
   5   BEGIN
   6   SELECT LNAME,  FNAME,  SALARY
   7      INTO  V_LAST,  V_FIRST,  V_SAL
   8      FROM EMPLOYEE
   9   WHERE EMPLOYEEID = 200;
  10   DBMS_OUTPUT.PUT_LINE
  11       ('EMPLOYEE NAME: ' || V_FIRST || ' ' || V_LAST );
  12   DBMS_OUTPUT.PUT_LINE
  13       ('SALARY:       ' || TO_CHAR(V_SAL));
  14  END;
  15  /
EMPLOYEE NAME: Jinku Shaw
SALARY:              24500

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-18 SELECT-INTO in PL/SQL.

```
SQL> DECLARE
   2        V_ID          EMPLOYEE.EMPLOYEEID%TYPE;
   3        V_DEPT        EMPLOYEE.DEPTID%TYPE  :=  &DEPT_NUM;
   4   BEGIN
   5    SELECT EMPLOYEEID INTO V_ID
   6    FROM EMPLOYEE WHERE DEPTID  =  V_DEPT;
   7   END;
   8  /
Enter value for dept_num: 10
DECLARE
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 5

SQL> /
Enter value for dept_num: 50
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 5

SQL>
```

Figure 11-19 SELECT . . . INTO with Error.

shows another example that results in exceptions because the SELECT…INTO statement returns either too many rows or no data.

DATA MANIPULATION IN PL/SQL

You can use all DML statements in PL/SQL with the same syntax you used in SQL. The three DML statements to manipulate data are:

1. The **INSERT** statement to add a new row in a table.
2. The **DELETE** statement to remove a row or rows.
3. The **UPDATE** statement to change values in a row or rows.

INSERT Statement

We will use an INSERT statement to add a new employee in the EMPLOYEE table. The statement will use sequences created earlier. For simplicity, only a few columns are used in the statement in Figure 11-20. NEXTVAL uses the next value from the sequence as the new EmployeeId, and CURRVAL uses the current value of the department from that sequence. If you also decide to insert today's date as the hire date, you could use the SYSDATE function for the value.

```
SQL> BEGIN
  2      INSERT INTO EMPLOYEE
  3         (EMPLOYEEID, LNAME, FNAME, SALARY, DEPTID)
  4         VALUES
  5              (EMPLOYEE_EMPLOYEEID_SEQ.NEXTVAL, 'RAI',
  6               'AISH', 90000, DEPT_DEPTID_SEQ.CURRVAL);
  7      COMMIT;
  8  END;
  9  /

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-20 INSERT in PL/SQL.

DELETE Statement

We will show the use of the DELETE statement in the PL/SQL block to remove some rows. Suppose the NamanNavan (N2) Corporation decides to remove the IT Department. All the employees belonging to that department must be removed

```
SQL> DECLARE
  2       V_DEPTID      DEPT.DEPTID%TYPE;
  3  BEGIN
  4       SELECT DEPTID
  5            INTO V_DEPTID
  6       FROM DEPT
  7       WHERE UPPER(DEPTNAME) = '&DEPT_NAME'
  8       DELETE FROM EMPLOYEE
  9            WHERE DEPTID = V_DEPTID;
 10       COMMIT;
 11 END;
 12 /
Enter value for dept_name: IT

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-21 DELETE in PL/SQL.

from the EMPLOYEE table. Figure 11-21 shows the DELETE statement in PL/SQL.

UPDATE Statement

The UPDATE statement can be used in a PL/SQL block for modification of data. The company decides to give a bonus commission to all the employees who are entitled to commission. The bonus is 10% of the commission received. Figure 11-22 shows an example of an UPDATE statement in PL/SQL block to modify commission.

```
SQL> DECLARE
  2      V_INCREASE   NUMBER := &DECIMAL_INCREASE;
  3  BEGIN
  4      UPDATE EMPLOYEE
  5           SET SALARY = SALARY * (1 + V_INCREASE)
  6      WHERE EMPLOYEEID = &EMP_ID;
  7      COMMIT;
  8 END;
  9 /
Enter value for decimal_increase: 0.15
Enter value for emp_id: 545

PL/SQL procedure successfully completed.

SQL>
```

Figure 11-22 UPDATE in PL/SQL.

TRANSACTION CONTROL STATEMENTS

You know what a transaction is. You also know the transaction control capabilities in Oracle. (If you don't remember, review Chapter 9). In Figures 11-18, 11-19, and 11-20, after performing a DML statement, the sample blocks have used a COMMIT statement. You do not have to commit within a PL/SQL block. If you do decide to use it, your data will be written to the disk right away, and the locks from those rows will be released. All transaction control statements are allowed in PL/SQL, and they are as follows:

- The COMMIT statement to commit the current transaction.
- The SAVEPOINT statement to mark a point in your transaction.
- The ROLLBACK [TO SAVEPOINT *n*] statement to discard all or part of the transaction.

IN A NUTSHELL . . .

- The three control structures in PL/SQL are sequence, selection, and looping.
- In a sequence structure, the instructions are performed in a linear order.
- The selection structure involves decision making based on the outcome of a Boolean expression.
- The looping structure contains a series of instructions that are performed repeatedly.
- Four selection structure statements in PL/SQL are IF . . . THEN . . . END IF, IF . . . THEN . . . ELSE . . . END IF, IF . . . THEN . . . ELSIF . . . END IF, and CASE statements.
- A CASE statement uses a variable as a selector and checks its value in WHEN clauses.
- A searched CASE statement does not use a variable as a selector, but it does use conditions in WHEN clauses.
- Boolean expressions or conditions use relational operators (=, <> or !=, >, >=, <, and <=), logical operators (AND, OR, and NOT), and other operators (BETWEEN. . .AND, IS NULL, and LIKE).
- It is good practice to add comments to a program and indent statements within a programming block.
- The three types of loops in PL/SQL are the basic loop, WHILE loop, and FOR loop.
- The basic loop is performed at least once and for as long as the condition is false. It is known as a post-test loop. The basic loop needs the EXIT statement to terminate.

- The WHILE loop is performed zero or more times and for as long as the condition is true. It is a pretest loop. The WHILE loop does not need the EXIT statement to terminate.
- The FOR loop is the simplest loop to write. It declares a loop-control variable implicitly, and it does not need the EXIT statement to end.
- The nested loop has a loop within a loop. Termination of the inner loop does not automatically terminate the outer loop. The loops can be labeled and can be terminated with the EXIT statement.
- PL/SQL programming blocks can also be nested. The variables declared in the outer block are available in the inner block, but the variables declared in the inner block are not accessible to the outer block.
- PL/SQL does not interact directly with the Oracle database. SQL is embedded in a PL/SQL block to work with tables.
- DML, data retrieval, and transaction control SQL statements are allowed in PL/SQL. DDL and DCL statements are not allowed in PL/SQL.
- The group functions are not supported in PL/SQL statements, but they can be used in SQL statements within a PL/SQL block.
- The SELECT-INTO statement retrieves data from a table into a set of variables. It must retrieve only one row at a time.
- The INSERT, DELETE, and UPDATE statements in a PL/SQL block are used to manipulate data in tables.

EXERCISE QUESTIONS

True/False:

1. IF statements are used to repeat a series of instructions.
2. The WHILE loop is always performed at least once.
3. A basic loop is performed as long as the condition is false.
4. The WHILE loop is performed as long as the condition is true.
5. The FOR loop exits when the EXIT statement is encountered.
6. The counter (loop-control variable) used in a FOR loop need not be declared explicitly.
7. The SELECT statement must have a mandatory INTO clause when used in a PL/SQL block.
8. The group functions are not supported in PL/SQL statements, but single-row functions are.
9. The GRANT statement is allowed in a PL/SQL block.
10. When two PL/SQL blocks are nested, the variables declared in the outer block are accessible in the inner block.

Answer the Following Questions:

1. Name and explain the control structures used in PL/SQL programming.
2. What is the difference between IF . . . THEN . . . ELSE . . . END IF and IF . . . THEN . . . ELSIF . . . END IF statements?

3. What is the difference between CASE and searched CASE statements?
4. Give four differences between the basic loop and the WHILE loop.
5. List the SQL statements allowed and not allowed in a PL/SQL block.
6. What is the purpose of a SELECT statement in the PL/SQL block? Explain it with an example.

Complete the Table for a Counter-Controlled WHILE Loop

Initial Value of Counter	Condition	Increment/Decrement	Number of Loop Executions
1	<10	1	
0	<= 100	5	
10	>= 5	−1	
1	>= 7	1	
0	<= 10	2	

LAB ACTIVITY

1. Write a PL/SQL block to find out if a year is a leap year. A leap year is divisible by 4 but not by 100, or it is divisible by 400. For example, 2000 and 2004 are leap years, but 1900 and 2001 are not leap years. (*Hint:* The function MOD(*n, d*) divides *n* by *d* and returns the integer remainder from the operation)
2. Write a PL/SQL block to print all odd numbers between 1 and 10 using a basic loop.
3. Using a FOR loop, print the values 10 to 1 in reverse order.
4. Create a table called ITEM with one column called ItemNum with the NUMBER type. Write a PL/SQL program to insert values of one to five for ItemNum.
5. Input a number with a substitution variable, and then print its multiplication table using a WHILE loop.
6. Input a month number between 1 and 12 and a four-digit year, and print the number of days in that month. For February (month = 2), check for a leap year to display the number of days as equal to 28 or 29.
7. Use a PL/SQL block to delete item number 4 from the ITEM table created in lab activity 4.
8. Write a PL/SQL block to ask a user to input a valid employee Id. Retrieve the employee's name, qualification description, salary, and commission. Print the name, qualification, and sum of the salary and commission.
9. You went to a video store and rented a DVD that is due in 3 days from the rental date. Input the rental date, rental month, and rental year. Caclculate and print the return date, return month, and return year. For example,

Rental Date	Rental Month	Rental Year	Return Date	Return Month	Return Year
2	12	2003	5	12	2003
27	2	2000	1	3	2000
30	12	2003	2	1	2004

12

PL/SQL Cursors and Exceptions

IN THIS CHAPTER . . .

- You will learn about a private work area for an SQL statement and its active set, called a cursor.
- You will be introduced to implicit and explicit cursor types.
- You will perform open, fetch, and close actions on explicit cursors.
- Use of cursor FOR loops and its implied statements are explained.
- Cursors with parameters and variable cursors are introduced.
- PL/SQL errors, known as exceptions, and their types are discussed.
- The process of declaring, raising, and handling different types of exceptions is covered.

In previous chapters, you learned about different control structures: sequence, selection, and looping. All structured programming languages support these structures. Other statements are also available in most of the languages. One of these additional statements is the GOTO statement, which allows you to branch unconditionally. All you have to code is GOTO ≪*labelname*≫, and the control shifts to the statement after the label. The GOTO statement, though available, is not preferred, however, because of its nonstructured nature. You also know how to use an SQL statement within a PL/SQL block for data retrieval, data manipulation, and transaction control.

In this chapter, you will learn about some advanced features of PL/SQL, such as retrieving more than one row from a database into a work area called a *cursor*. One of the strongest benefits of PL/SQL is its error-handling capabilities. The error conditions, known as *exceptions*, in PL/SQL, are also covered in this chapter.

CURSORS

When you execute an SQL statement from a PL/SQL block, Oracle assigns a private work area for that statement. The work area, called a cursor, stores the statement and the results returned by execution of that statement. A cursor is created either implicitly or explicitly by you.

Types of Cursors

The cursor in PL/SQL is of two types:

1. In a ***static cursor***, the contents are known at compile time. The cursor object for such an SQL statement is always based on one SQL statement.
2. In a ***dynamic cursor***, a cursor variable that can change its value is used. The variable can refer to different SQL statements at different times.

This chapter covers static cursors in detail. It also introduces you to the new concept of dynamic cursors using a cursor variable. The static cursors are of two types as well:

1. You do not declare an ***implicit cursor***. PL/SQL declares, manages, and closes it for every Data Manipulation Language (DML) statement, such as INSERT, UPDATE, or DELETE.
2. You declare an explicit cursor when you have an SQL statement in a PL/SQL block that returns more than one row from an underlying table. The rows retrieved by such a statement into an explicit cursor make up the active set. When opened, the cursor points to the first row in the active set. You can retrieve and work with one row at a time from the active set. With every fetch of a row, the pointer moves to the next row. The cursor returns the current row to which it is pointing.

IMPLICIT CURSORS

PL/SQL creates an implicit cursor when an SQL statement is executed from within the program block. The implicit cursor is created only if an explicit cursor is not attached to that SQL statement. Oracle opens an implicit cursor, and the pointer is set to the first (and the only) row in the cursor. Then, the SQL statement is fetched and executed by the SQL engine on the Oracle server. The PL/SQL engine closes the

implicit cursor automatically. A programmer cannot perform on an implicit cursor all the operations that are possible on explicit cursor statements. PL/SQL creates an implicit cursor for each DML statement in PL/SQL code. You cannot use an explicit cursor for DML statements. You can choose to declare an explicit cursor for a SELECT statement that returns only one row of data, but if you don't declare an explicit cursor for a SELECT statement returning one row of data, an implicit cursor is created for it.

You have no control over an implicit cursor. The implied queries perform operations on implicit cursors. PL/SQL actually tries to fetch twice to make sure that a TOO_MANY_ROWS exception does not exist. The explicit cursor is more efficient, because it does not try that extra fetch. It is possible to use an explicit cursor for a SELECT statement that returns just one row, because you have control over it. For example,

```
CURSOR deptname_cur IS
SELECT DeptName, Location FROM dept WHERE DeptId = 10;
```

Here, only one row is retrieved by the cursor with two column values, Finance and Charlotte, which later can be assigned to variables by fetching that row.

EXPLICIT CURSORS

An explicit cursor is declared as a SELECT statement in the PL/SQL block. It is given a name, and you can use explicit statements to work with it. You have total control of when to open the cursor, when to fetch a row from it, and when to close it. There are cursor attributes in PL/SQL to get the status information on explicit cursors. Remember, you can declare an explicit cursor for a SELECT statement that returns one or more rows, but you cannot use an explicit cursor for a DML statement.

Four actions can be performed on an explicit cursor:

1. Declare it.
2. Open it.
3. Fetch row(s) from it.
4. Close it.

Declaring an Explicit Cursor

A cursor is declared as a SELECT statement. The SELECT statement must not have an INTO clause in a cursor's declaration. If you want to retrieve rows in a specific order into a cursor, an ORDER BY clause can be used in the SELECT statement. The general syntax is

```
DECLARE
    CURSOR cursorname IS
      SELECT statement;
```

where *cursorname* is the name of the cursor that follows identifier-naming rules. The SELECT statement is any valid data-retrieval statement. The cursor declaration is done in the DECLARE section of the PL/SQL block, but a cursor cannot be used in programming statements or expressions, as with other variables.

For example, Figures 12-1 and 12-2 show declarations of two cursors. In Figure 12-1, the cursor is based on a SELECT query that will retrieve all rows from the DEPT table in the work area. In the Figure 12-2, two columns, EmployeeId and Salary, are selected into the cursor with DeptId equal to 20.

```
SQL> DECLARE
  2      CURSOR DEPT_CUR
  3      IS
  4          SELECT *
  5          FROM DEPT;
  6  BEGIN
  7     . . .
  8  END;
```

Figure 12-1 Explicit cursor.

```
SQL> DECLARE
  2      CURSOR EMPLOYEE_CUR
  3      IS
  4          SELECT EMPLOYEEID, SALARY
  5          FROM EMPLOYEE
  6          WHERE DEPTID = 20;
  7  BEGIN
  8   . . .
  9  END;
```

Figure 12-2 Explicit cursor.

A cursor is based on a SELECT statement, so it is linked to at least one table from the database. The list that follows can contain the names of columns, local variables, constants, functions, and bind variables. It is possible for a variable to have the same name as a column in a table. If you try to use both of them together in a SELECT statement, the column gets higher precedence. Though permitted, it is not advisable to use the same name for a variable that exists in a column retrieved by the SELECT statement.

In the next section, we will talk about the actions performed on an explicit cursor.

Actions on Explicit Cursors

Actions are performed on cursors declared in the DECLARE section of the block. Before rows can be retrieved from a cursor, you must open the cursor.

Opening a Cursor. When a cursor is opened, its SELECT query is executed. The active set is created using all tables in the query and then restricting to rows that meet the criteria. The data retrieved by the SELECT query is brought into the cursor or the work area. The cursor points to the first row in the active set. PL/SQL uses an OPEN statement to open a cursor. The general syntax is

```
OPEN cursorname;
```

For example,

```
OPEN employee_cur;
```

You must open a cursor that has not been opened in the program block or is closed to retrieve data into it. If you try to open a cursor that is already open, the following Oracle error message appears:

```
PLS-06511:PL/SQL: cursor already open
```

Notice the change in the error message prefix PLS! The errors with prefix ORA are Oracle database errors, and the errors with prefix PLS are PL/SQL errors. You will see later how this can be avoided using the cursor attribute %ISOPEN.

Fetching Data from a Cursor. The SELECT statement creates an active set based on tables in the FROM clause, column names in the SELECT clause, and rows based on conditions in the WHERE clause. The cursor is a virtual table that you can work with. You can retrieve a row that the cursor is pointing to, and the values from that row are retrieved into variables or into a PL/SQL record to perform processing. After reading values from a row into variables, the cursor pointer moves to the next row in the active set. The number of variables must match the number of columns in the row. In PL/SQL, a FETCH statement is used for this action. The general syntax is

```
FETCH cursorname INTO variablelist / recordname;
```

where *variablelist* may include a local variable, a table, or a bind variable and *recordname* is the name of a record structure. For example,

```
FETCH employee_cur INTO v_empnum, v_sal;
```

or

```
FETCH employee_cur INTO emp_rec;
```

where *emp_rec* is declared with %ROWTYPE declaration attribute:

```
emp_rec employee_cur%ROWTYPE;
```

In the first example, two columns, EmployeeID and Salary, are retrieved into *v_empnum* and *v_sal*, respectively. The number of items matches the number of variables in the SELECT statement. The order of items and variables must also match. The variables should be declared with a %TYPE declaration variable to ensure the correct data type. If the number of items in SELECT does not match the number of variables, it results in the following compiler error:

```
PLS-00394:wrong number of values in the INTO list of a FETCH statement
```

The second example of FETCH uses a record. A composite data type can be used for the record instead of the CursorName%ROWTYPE declaration. You will learn about the record data type in the next chapter.

Suppose you opened a cursor in a PL/SQL block to retrieve data from a table, and then inserts, deletes, and updates are performed on that table after the OPEN statement is executed. Oracle enforces **read consistency,** and the data manipulation statements are ignored. You will have the same data from the point of execution of OPEN to the point of execution of CLOSE statements. Changing data in the underlying table does not change data in the work area.

Closing a cursor. When you are done with a cursor, you should close it. A closed cursor can be reopened again. If you terminate your PL/SQL program without closing an open cursor, it will not result in an exception. In fact, the local cursor declared in a PL/SQL block is closed automatically when the block terminates. It is a good habit, however, to close an open cursor before terminating the block. There is a limit to the number of cursors a user may open simultaneously. The default value is in a parameter called OPEN_CURSORS, which has default value of 50. A user releases memory by closing a cursor. PL/SQL uses the CLOSE statement to close a cursor. The general syntax is

```
CLOSE cursorname;
```

For example,

```
CLOSE employee_cur;
```

EXPLICIT CURSOR ATTRIBUTES

Actions can be performed on cursors with OPEN, FETCH, and CLOSE statements. You can get information about the current status of a cursor or the result of the last fetch by using cursor attributes. The four explicit cursor attributes are:

%ISOPEN	It returns TRUE if the cursor is open; otherwise, it returns FALSE.
%FOUND	It returns TRUE if the last fetch returned a row; otherwise, it returns FALSE.
%NOTFOUND	It returns TRUE if the last fetch did not return a row; otherwise, it returns FALSE. It complements the %FOUND attribute.
%ROWCOUNT	It returns total number of rows returned.

%ISOPEN

The %ISOPEN attribute is useful in making sure that you do not open a cursor that is already open. It is also appropriate for making sure that a cursor is open before

trying to fetch from it. For example, Figure 12-3 tests to see if a cursor is open. If it is not open, already, the cursor is opened. Then, execution continues with a loop and a fetch in it.

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2      V_LAST EMPLOYEE.LNAME%TYPE;
  3      V_FIRST EMPLOYEE.FNAME%TYPE;
  4      V_SAL EMPLOYEE.SALARY%TYPE;
  5      CURSOR EMPLOYEE_CUR IS
  6      SELECT LNAME, FNAME, SALARY
  7      FROM EMPLOYEE
  8      WHERE DEPTID = 20;
  9  BEGIN
 10       IF NOT EMPLOYEE_CUR%ISOPEN THEN
 11            OPEN EMPLOYEE_CUR;
 12       END IF;
 13       LOOP
 14            FETCH EMPLOYEE_CUR
 15                 INTO V_LAST, V_FIRST, V_SAL;
 16            EXIT WHEN NOT EMPLOYEE_CUR%FOUND;
 17            DBMS_OUTPUT.PUT_LINE
 18            (V_FIRST || ' ' || V_LAST || ' ' || V_SAL);
 19       END LOOP;
 20       DBMS_OUTPUT.PUT_LINE
 21          (EMPLOYEE_CUR%ROWCOUNT || ' EMPLOYEE(S) FOUND');
 22  END;
 23  /
Alex McCall 66500
Derek Dev 80000
2 EMPLOYEE(S) FOUND

PL/SQL procedure successfully completed.

SQL>
```

Figure 12-3 Cursor attributes.

%FOUND

The %FOUND attribute returns a TRUE if the last FETCH returned a row; otherwise, it returns a FALSE. For example, Figure 12-3 shows a block segment that exits the loop if a row is not found. The loop continues as long as a row is fetched.

%NOTFOUND

The %NOTFOUND attribute returns a TRUE if the last FETCH did not return a row; otherwise, it returns a FALSE. It is the opposite of the %FOUND attribute.

The statement

```
EXIT WHEN NOT employee_cur%FOUND;
```

can be written as

```
EXIT WHEN employee_cur%NOTFOUND;
```

%ROWCOUNT

When a cursor is opened and no fetch is done from it, %ROWCOUNT is equal to zero. With every fetch, %ROWCOUNT is incremented by one. The cursor must be open before using %ROWCOUNT on it. For example, the code in Figure 12-3 goes through the loop as long as a row is fetched. A count of the number of rows fetched is kept by the PL/SQL engine. In the code, we are printing the total number of rows fetched by the cursor at the end.

IMPLICIT CURSOR ATTRIBUTES

An implicit cursor cannot be opened, fetched from, or closed with a statement. You do not name implicit cursors. The cursor attributes are available for an implicit cursor with the name SQL as a prefix. The four attributes for a implicit cursor are:

1. SQL%ISOPEN.
2. SQL%ROWCOUNT.
3. SQL%NOTFOUND.
4. SQL%FOUND.

If an implicit cursor is not open, SQL%ROWCOUNT will return NULL. Similarly, the other three attributes will return FALSE. You will never get an INVALID_CURSOR error for an implicit cursor. The %ISOPEN attribute will always return FALSE, because it is open only during the statement's execution. It is opened and closed implicitly. When a SELECT statement returns either no or more than one row, the NO_DATA_FOUND or TOO_MANY_ROWS exception, respectively, is raised. The cursor attribute SQL applies to the last SQL statement executed in the block.

CURSOR FOR LOOPS

The cursor FOR loop is the easiest way to write a loop for explicit cursors. The cursor is opened implicitly when the loop starts. A row is then fetched into the record from the cursor with every iteration of the loop. The cursor is closed automatically when the loop ends, and the loop ends when there are no more rows. The cursor

FOR loop automates all the cursor actions. The general syntax is

```
FOR recordname IN cursorname LOOP
        Loop statements;
        . . .
END LOOP;
```

where *recordname* is the name of the record that is declared implicitly in the loop and is destroyed when the loop ends and *cursorname* is the name of declared explicit cursor.

Figure 12-4 uses a Cursor FOR loop with a record. When the loop starts, the cursor is opened implicitly. During the loop execution, an implicit fetch retrieves a row into the record for processing with each loop iteration. When an implicit fetch cannot retrieve a row, the cursor is closed, and the loop terminates. The OPEN, FETCH, and CLOSE statements are missing, because these operations are performed implicitly. The record's columns are addressed with *recordname.columnname* notation. If the record is accessed after the END LOOP statement, it will throw an exception, because the record's scope is only within the loop body.

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2      CURSOR EMPLOYEE_CUR IS
  3      SELECT LNAME, FNAME, SALARY
  4      FROM EMPLOYEE;
  5  BEGIN
  6     FOR EMP_REC IN EMPLOYEE_CUR LOOP
  7        IF EMP_REC.SALARY > 75000 THEN
  8        DBMS_OUTPUT.PUT(EMP_REC.FNAME || ' ');
  9        DBMS_OUTPUT.PUT(EMP_REC.LNAME || ' ');
 10        DBMS_OUTPUT.PUT_LINE(EMP_REC.SALARY || ' ');
 11       END IF;
 12     END LOOP;
 13  END;
 14  /
John Smith 265000
Larry Houston 150000
Derek Dev 80000

PL/SQL procedure successfully completed.

SQL>
```

Figure 12-4 Cursor FOR Loop.

Cursor FOR Loop Using a Subquery

Use of a subquery in the cursor FOR loop eliminates declaration of an explicit cursor. The cursor is created by a subquery in the FOR loop statement itself. In Figure 12-5, an explicit cursor is used with implicit actions. One thing that is missing is the cursor name. The cursor declaration is not necessary, because it is created through the subquery. This subquery is similar to the inline view covered in the SQL section of this text.

```
SQL> BEGIN
  2     FOR EMP_REC IN
  3        (SELECT FNAME, LNAME, SALARY, COMMISSION
  4         FROM EMPLOYEE
  5         WHERE DEPTID = 10) LOOP
  6         DBMS_OUTPUT.PUT_LINE
  7           (EMP_REC.FNAME || ' ' || EMP_REC.LNAME || ' $' ||
  8           TO_CHAR(EMP_REC.SALARY + NVL(EMP_REC.COMMISSION, 0)));
  9     END LOOP;
 10  END;
 11  /
John Smith $300000
Sandi Roberts $75000
Sunny Chen $35000

PL/SQL procedure successfully completed.

SQL>
```

Figure 12-5 Cursor FOR loop with a subquery.

SELECT . . . FOR UPDATE CURSOR

When you type a SELECT query, the result is returned to you without locking any rows in the table. Row locking is kept to a minimum. You can explicitly lock rows for update before changing them in the program. The FOR UPDATE clause is used with the SELECT query for row locking. The locked rows are not available to other users for DML statements until you release them with COMMIT or ROLLBACK commands. Rows that are locked for update do not have to be updated. The general syntax is

```
CURSOR cursorname IS
  SELECT columnnames
  FROM tablename(s)
  [WHERE condition]
  FOR UPDATE [OF columnnames] [NOWAIT];
```

The optional part of a FOR UPDATE clause is *OF columnnames*, which enables you to specify columns to be updated. You can actually update any column in a locked row. The optional word *NOWAIT* tells you right away if another user has already locked the table and lets you continue with other tasks. If you do not use NOWAIT and one or more rows are already locked by another user, you will have to wait until the lock is released. If you have granted UPDATE privilege to another user on your table, that user can prevent you from performing DML operations on your own table by locking them indefinitely!

WHERE CURRENT OF CLAUSE

In a cursor, data manipulation in the form of UPDATE or DELETE is performed on rows fetched. The WHERE CURRENT OF clause allows you to perform data manipulation only on a recently fetched row. The general syntax is

```
UPDATE tablename
 SET clause
 WHERE CURRENT OF cursorname;
-----------------------------
DELETE FROM tablename
 WHERE CURRENT OF cursorname;
```

You do not have to use a separate WHERE condition. The WHERE CURRENT OF clause references the cursor, and changes apply to only the last fetched row.

CURSOR WITH PARAMETERS

A cursor can be declared with parameters, which allow you to pass values to the cursor. These values are passed to the cursor when it is opened, and they are used in the query when it is executed. With the use of parameters, you can open and close a cursor many times with different values. The cursor with different values will then return different active sets each time it is opened. When parameters are passed, you need not worry about the scope of variables. The general syntax is

```
CURSOR cursorname
   [(parameter1 datatype, parameter2 datatype, . . .)]
IS
SELECT query;
```

where *parameter1*, *parameter2*, and so on are formal parameters passed to the cursor and *datatype* is any scalar data type assigned to the parameter. The parameters are assigned only data types; they are not assigned size.

When a cursor is opened, values are passed to the cursor. Each value must match the positional order of the parameters in a cursor's declaration. The values

can be passed through literals, PL/SQL variables, or bind variables. The parameters in a cursor are passed in to the cursor, but you cannot pass any value out of the cursor through parameters.

For example, in the PL/SQL program of Figure 12-6, the cursor *employee_cur* is declared with a parameter *dept_num,* which is also used in the cursor SELECT statement's WHERE clause. When the program executes, it asks to input a value for department number with substitution variable DEPARTMENT_ID, which is assigned to variable D_ID. The cursor is opened with parameter D_ID, which has value of 10 as entered by the user. The format parameter DEPT_NUM gets value of parameter D_ID. The active set is created based on DEPTID = DEPT_NUM. Then, the cursor loop prints all employees for department number 10. The parameter can be passed a value with a literal (as done in here), a bind variable, or an expression.

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2      V_FIRST EMPLOYEE.FNAME%TYPE;
  3      V_LAST EMPLOYEE.LNAME%TYPE;
  4      D_ID  NUMBER(2) := &DEPARTMENT_ID;
  5      CURSOR EMPLOYEE_CUR (DEPT_NUM EMPLOYEE.DEPTID%TYPE) IS
  6      SELECT LNAME, FNAME
  7      FROM EMPLOYEE
  8      WHERE DEPTID = DEPT_NUM;
  9
 10  BEGIN
 11      OPEN EMPLOYEE_CUR(D_ID);
 12      DBMS_OUTPUT.PUT_LINE
 13      ('EMPLOYEES IN DEPARTMENT ' || TO_CHAR(D_ID));
 14      LOOP
 15          FETCH EMPLOYEE_CUR INTO V_LAST, V_FIRST;
 16            EXIT WHEN EMPLOYEE_CUR%NOTFOUND;
 17          DBMS_OUTPUT.PUT_LINE(V_LAST || ', ' || V_FIRST);
 18      END LOOP;
 19      CLOSE EMPLOYEE_CUR;
 20  END;
 21  /
Enter value for department_id: 10
EMPLOYEES IN DEPARTMENT 10
Smith, John
Roberts, Sandi
Chen, Sunny

PL/SQL procedure successfully completed.

SQL>
```

Figure 12-6 Cursor with Parameter.

A cursor with a parameter can be opened multiple times with a different parameter value to get a different active set.

When you declare a cursor with one or more parameters, you can initialize it to a default value as follows:

```
CURSOR employee_cur (dept_id employee.DeptId%TYPE := 99) IS
```

CURSOR VARIABLES: AN INTRODUCTION

An explicit cursor is the name of the work area for an active set. A cursor variable is a reference to the work area. A cursor is based on one specific query, whereas a cursor variable can be opened with different queries within a program. A static cursor is like a constant, and a cursor variable is like a pointer to that cursor. You can also use the action statements OPEN, FETCH, and CLOSE with cursor variables. The cursor attributes %ISOPEN, %FOUND, %NOTFOUND, and %ROWCOUNT are available for cursor variables. Cursor variables have many similarities with static cursors.

The cursor variable has other capabilities in addition to the features of a static cursor. It is a variable, so it can be used in an assignment statement. A cursor variable can also be assigned to another cursor variable.

REF CURSOR Type

Two steps are involved in creating a cursor variable. First, you have to create a referenced cursor type. Second, you have to declare an actual cursor variable with the referenced cursor type. The general syntax is

```
TYPE cursortypename IS REF CURSOR [RETURN returntype];
cursorvarname cursortypename;
```

where *cursortypename* is the name of the type of cursor. The RETURN clause is optional. The *returntype* is the RETURN data type and can be any valid data structure, such as a record or structure defined with %ROWTYPE. For example,

```
TYPE any_cursor_type IS REF CURSOR;
any_cursor_var              any_cursor_type;
TYPE employee_cursor_type IS REF CURSOR
  RETURN employee%ROWTYPE;
employee_cursor_var         employee_cursor_type;
```

In this example, the first cursor type, *any_cursor_type*, is called the ***weak*** type, because its RETURN clause is missing. This type of cursor type can be used with any query. The cursor type declared with the RETURN clause is called the ***strong*** type, because it links a row type to the cursor type at the declaration time.

Opening a Cursor Variable

You assign a cursor to the cursor variable when you OPEN it. The general syntax is

```
OPEN cursorname / cursorvarname FOR SELECT query;
```

If the cursor type is declared with the RETURN clause, the structure from the SELECT query must match the structure specified in the REF CURSOR declaration. For example,

```
OPEN employee_cursor_var FOR SELECT * FROM employee;
```

The structure returned by the SELECT query matches the RETURN type employee%ROWTYPE.

The other cursor type, *any_cursor_type*, is declared without the RETURN clause. It can be opened without any worry about matching the query's result to anything. The weak type is more flexible than the strong type, but there is no error checking. Let us look at some OPEN statements for the weak cursor variable:

```
OPEN any_cursor_var FOR SELECT * FROM dept;
OPEN any_cursor_var FOR SELECT * FROM employee;
OPEN any_cursor_var FOR SELECT DeptId FROM dept;
```

It is possible to have all three statements in one program block. The cursor variable assumes different structures with each OPEN.

Fetching from a Cursor Variable

The fetching action is same as that of a cursor. The compiler checks the data structure type after the INTO clause to see if it matches the query linked to the cursor. The general syntax is

```
FETCH cursorvarname INTO recordname / variablelist;
```

(***Note:*** At the end of this chapter, under ***More Sample Programs***, are three more coding examples on cursors.)

EXCEPTIONS

In PL/SQL, errors are known as exceptions. An exception occurs when an unwanted situation arises during the execution of a program. Exceptions can result from a system error, a user error, or an application error. When an exception occurs, control of the current program block shifts to another section of the program, known as the exception section, to handle exceptions. If the exception handler exists, it is performed. If the exception handler does not exist in the current block, control propagates to the outer blocks. If the handler is not in any of the blocks, PL/SQL returns an error, and the script stops.

A programmer writes a program to perform certain tasks, keeping only the positive things in mind. Programming is more than just writing statements to perform a

task, however. A programmer must think of all the negative situations that may arise while the program is executed. For example, the system might run out of memory, the database might not be accessible, or the user might type in the wrong value or press the wrong key. The programmer must put extra effort into program design to remove bugs and make the program error-proof with additional code to perform in case of exceptions. PL/SQL provides ways to trap and handle errors, and it is possible to create PL/SQL programs with full protection against errors. When exception-handling code is written for an exception, that exception can occur anywhere in the block, and the same handler can deal with it.

The syntax of an anonymous block is given below. Control transfers from the execution section to the exception section. PL/SQL browses through the section to look for the handler. If the handler is present, it is executed. The program may have more than one exception handler, written with WHEN . . . THEN statements like an ELSIF or CASE structure (as supported by Oracle9i). For example,

```
DECLARE
 Declaration of constants, variables, cursors, and exceptions
BEGIN
 /* Exception is raised here.*/
EXCEPTION
 /* Exception is trapped here.*/
END;
```

The general syntax of an exception section is:

```
EXCEPTION
  WHEN exceptionname1 [OR exceptionname2, . . .] THEN
    Executable statements
  [WHEN exceptionname3 [OR exceptionname4, . . .] THEN
    Executable statements]
  [WHEN OTHERS THEN
    Executable statements]
```

An exception is handled when the exception name matches the name of the raised exception. The exceptions are trapped by name. If an exception is raised but no handler for it is present, the WHEN OTHERS clause is performed (if present). If there is no handler for an exception and no WHEN OTHERS clause, the error number and associated message are displayed to the user.

TYPES OF EXCEPTIONS

There are three types of exceptions in PL/SQL:

1. ***Predefined Oracle server exceptions*** are exceptions that are named by PL/SQL and are raised implicitly when a PL/SQL or DBMS error occurs.

There are approximately 20 such exceptions. Each has a name and associated error number.

2. ***Nonpredefined Oracle server exceptions*** are standard Oracle server errors that are not named by the system. They can be declared in the declaration section but are raised implicitly by the server. These exceptions do not have a name, but they do have an associated error number.
3. ***User-defined exceptions*** are exceptions that are declared in the declaration section and are raised by the user explicitly. The user decides which abnormal condition is an exception. The Oracle server does not consider these conditions to be errors.

Predefined Oracle Server Exceptions

Exceptions that are given names by PL/SQL are declared in a PL/SQL package called STANDARD. The exception-handling routine is also defined there. The user does not have to declare or raise predefined server exceptions. Figure 12-7 provides the exception name, the error code returned by the built-in function SQLCODE, and a brief description of the exception.

Exception Name	Error Number	Brief Description
NO_DATA_FOUND	ORA-01403	Single-row SELECT returned no data.
TOO_MANY_ROWS	ORA-01422	Single-row SELECT returned more than one row.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero.
VALUE_ERROR	ORA-06502	Arithmetic, conversion, truncation, or size constraint error occurred.
STORAGE_ERROR	ORA-06500	PL/SQL ran out of memory, or memory is corrupted.
LOGIN_DENIED	ORA-01017	Logging on to Oracle with an invalid username or password.
NOT_LOGGED_ON	ORA-01012	PL/SQL program issues a database call without being connected to Oracle.
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal problem.
ACCESS_INTO_NULL	ORA-06530	Attempted to assign values to the attributes of an uninitialized object.
CURSOR_ALREADY_OPEN	ORA-06511	Attempted to open an already-open cursor.
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value.
INVALID_CURSOR	ORA-01001	Illegal cursor operation occurred.
INVALID_NUMBER	ORA-01722	Conversion of a character string to number failed.
ROWTYPE_MISMATCH	ORA-06504	Host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types.
TIMEOUT_ON_RESOURCE	ORA-00051	Time-out occurred while Oracle is waiting for a resource.

Figure 12-7 Predefined/named system exceptions.

Suppose a program block generates an error message for exception error number ORA-01403 that is not handled by the exception section. The error has occurred because of a SELECT statement that did not return any data. You can write an exception handler as shown in Figure 12-8.

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
  2      V_FIRST EMPLOYEE.FNAME%TYPE;
  3      V_LAST EMPLOYEE.LNAME%TYPE;
  4      D_ID  NUMBER(2) := &DEPARTMENT_ID;
  5  BEGIN
  6      SELECT LNAME, FNAME
  7      INTO V_LAST, V_FIRST
  8      FROM EMPLOYEE
  9      WHERE DEPTID = D_ID;
 10      DBMS_OUTPUT.PUT_LINE(' ');
 11      DBMS_OUTPUT.PUT_LINE(V_LAST || ', ' || V_FIRST);
 12  EXCEPTION
 13      WHEN NO_DATA_FOUND THEN
 14          DBMS_OUTPUT.PUT_LINE
 15          ('NO SUCH DEPARTMENT WITH EMPLOYEES');
 16      WHEN TOO_MANY_ROWS THEN
 17          DBMS_OUTPUT.PUT_LINE
 18          ('MORE THAN ONE EMPLOYEE IN DEPT ' || D_ID);
 19  END;
 20  /
```

Figure 12-8 Handling named exceptions (source).

In Figure 12-8, two named exceptions, NO_DATA_FOUND and TOO_MANY_ROWS, are handled. The NO_DATA_FOUND exception occurs when a SELECT . . . INTO statement does not retrieve a row. The TOO_MANY_ROWS exception occurs when a SELECT . . . INTO statement retrieves more than one row. Figure 12-9 shows execution of the code in Figure 12-8 to demonstrate handling of both exceptions. Input value 10 returned more than one row, resulting in the TOO_MANY_ROWS exception, which is raised implicitly and handled. Input value 50 returned 0 rows, resulting in the NO_DATA_FOUND exception, which is also raised implicitly and handled by the block. Input value 40 returned one employee, so no exception was thrown.

Nonpredefined Oracle Server Exceptions

A nonpredefined Oracle server exception has an attached Oracle error code, but it is not named by Oracle. You can trap such exceptions with a WHEN OTHERS clause or by declaring them with names in the DECLARE section. The declared exception

```
Enter value for department_id: 10
MORE THAN ONE EMPLOYEE IN DEPT 10

PL/SQL procedure successfully completed.

SQL> /
Enter value for department_id: 50
NO SUCH DEPARTMENT WITH EMPLOYEES

PL/SQL procedure successfully completed.

SQL> /
Enter value for department_id: 40
Houston, Larry

PL/SQL procedure successfully completed.

SQL>
```

Figure 12-9 Handling named exceptions (output).

is raised implicitly by Oracle, or you can raise it explicitly. You can write exception-handler code for it.

Pragma Exception_Init. PRAGMA is a compiler directive that associates an exception name with an internal Oracle error code. The PRAGMA directive is not processed with the execution of a PL/SQL block, but it directs the PL/SQL compiler to associate a name with the error code. You can use more than one PRAGMA EXCEPTION_INIT directive in your DECLARE section to assign names to different error codes. You may even assign more than one name to the same error number. Naming an internal error code makes your program more readable.

Naming and associating are two separate statements in the declaration section. First, an exception name is declared as an EXCEPTION. Second, the declared name is associated with an internal error code returned by SQLCODE with the PRAGMA directive. The general syntax is

```
exceptionname EXCEPTION;
PRAGMA EXCEPTION_INIT (exceptionname, errornumber);
```

where *exceptionname* is user supplied and errornumber is Oracle's internal error code. The error code is a numeric literal with a negative sign (−).

Suppose you tried to remove a department from the DEPT table but the child rows still exist in the EMPLOYEE table, because there are employees with that DeptId. You will get Oracle error ORA-02292. You can declare an exception and associate it with the server error code number -2292. Figure 12-10 shows a PL/SQL block with a declaration and exception trapping of a nonpredefined Oracle exception. There is no explicit RAISE statement.

```
SQL> DECLARE
  2      emp_remain EXCEPTION;
  3      PRAGMA EXCEPTION_INIT (emp_remain, -2292);
  4      v_deptid dept.DeptId%TYPE := &p_deptnum;
  5  BEGIN
  6      DELETE FROM dept
  7      WHERE DeptId = v_deptid;
  8      COMMIT;
  9  EXCEPTION
 10      WHEN emp_remain THEN
 11      DBMS_OUTPUT.PUT('DEPARTMENT ' || TO_CHAR(v_deptid));
 12      DBMS_OUTPUT.PUT(' cannot be removed - ');
 13      DBMS_OUTPUT.PUT_LINE('Employees in department');
 14  END;
 15  /
Enter value for p_deptnum: 10
DEPARTMENT 10 cannot be removed - Employees in department

PL/SQL procedure successfully completed.

SQL> /
Enter value for p_deptnum: 60

PL/SQL procedure successfully completed.

SQL>
```

Figure 12-10 Nonpredefined Oracle exception.

Exception-Trapping functions. When an exception occurs in your program, you don't know the error code for the error and its associated message unless you take specific action to identify them. Once you know the error code and the message, you can modify your program to take action based on the error. The two functions to identify the error code and error message are:

1. **SQLCODE**. The SQLCODE function returns a negative error code number. The number can be assigned to a variable of NUMBER type.
2. **SQLERRM**. The SQLERRM function returns the error message associated with the error code. The maximum length of error message is 512 bytes. It can be assigned to a VARCHAR2-type variable.

Figure 12-11 shows the use of SQLCODE and SQLERRM to identify the error code and message for further modifications of the exception section of a program based on information displayed.

```
SQL> DECLARE
  2      V_FIRST      EMPLOYEE.FNAME%TYPE;
  3      V_LAST       EMPLOYEE.LNAME%TYPE;
  4      D_ID         NUMBER(2) := &DEPARTMENT_ID;
  5      V_CODE       NUMBER;
  6      V_MSG        VARCHAR2(255);
  7  BEGIN
  8      SELECT LNAME, FNAME
  9      INTO V_LAST, V_FIRST
 10      FROM EMPLOYEE
 11      WHERE DEPTID = D_ID;
 12      DBMS_OUTPUT.PUT_LINE(' ');
 13      DBMS_OUTPUT.PUT_LINE(V_LAST || ', ' || V_FIRST);
 14  EXCEPTION
 15      WHEN OTHERS THEN
 16          V_CODE := SQLCODE;
 17          V_MSG := SQLERRM;
 18          DBMS_OUTPUT.PUT_LINE('ERROR CODE: ' || SQLCODE);
 19          DBMS_OUTPUT.PUT_LINE(SQLERRM);
 20  END;
 21  /
Enter value for department_id: 10
ERROR CODE: -1422
ORA-01422: exact fetch returns more than requested number of rows

PL/SQL procedure successfully completed.

SQL>
```

Figure 12-11 SQLCODE and SQLERRM.

User-Defined Exceptions

The standard errors covered under the previous two types are in the STANDARD package with an error code and an accompanying message. Often, however, you will encounter situations that are specific to a given program. For example, a birth date falls in the future, a quantity in an invoice is negative, a student registers for course without satisfying prerequisite, and so on.

You are allowed to define your exceptions in PL/SQL. You must perform three steps for exceptions you want to define:

1. You must **declare** the exception in the DECLARE section. There is no need to use a PRAGMA directive, because there is no standard error number to associate.
2. You must **raise** the exception in the execution section of the program with an explicit RAISE statement.
3. You must **write** the handler for the exception.

Figures 12-12 and 12-13 are examples of the user-defined exceptions *invalid_commission* and *no_commission*. The *invalid_commission* exception is raised when the commission value is negative. The *no_commission* exception is raised when the commission value is NULL. Figure 12-12 shows the source code, and Figure 12-13 shows exception handling based on the EmployeeId entered by the user.

```
SQL> DECLARE
  2      invalid_commission EXCEPTION;
  3      no_commission EXCEPTION;
  4      v_comm employee.Commission%TYPE;
  5  BEGIN
  6      SELECT Commission
  7      INTO v_comm
  8      FROM employee
  9      WHERE EmployeeId = &emp_id;
 10      IF v_comm < 0 then
 11         RAISE invalid_commission;
 12      ELSIF v_comm IS NULL THEN
 13         RAISE no_commission;
 14      ELSE
 15         DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_comm));
 16      END IF;
 17  EXCEPTION
 18      WHEN invalid_commission THEN
 19          DBMS_OUTPUT.PUT_LINE('Commission is negative.');
 20      WHEN no_commission THEN
 21          DBMS_OUTPUT.PUT_LINE('No commission value');
 22      WHEN OTHERS THEN
 23          DBMS_OUTPUT.PUT_LINE('No such ID');
 24  END;
 25  /
```

Figure 12-12 User-defined exception (source).

RAISE_APPLICATION_ERROR Procedure

The RAISE_APPLICATION_ERROR procedure allows you to display nonstandard error codes and user-defined error messages from a stored subprogram. The general syntax is

RAISE_APPLICATION_ERROR (error_code, error_message [, TRUE/FALSE];

where the *error_code* is a user-specified number between −20,000 and −20,999 and *error_message* is a user-supplied message that can be up to 512 bytes long. The third Boolean parameter, TRUE/FALSE, is optional. TRUE means "place the error on

```
Enter value for emp_id: 111
35000

PL/SQL procedure successfully completed.

SQL> /
Enter value for emp_id: 123
No commission value

PL/SQL procedure successfully completed.

SQL> /
Enter value for emp_id: 546
Commission is negative.

PL/SQL procedure successfully completed.

SQL> /
Enter value for emp_id: 321
No such ID

PL/SQL procedure successfully completed.


SQL>
```

Figure 12-13 User-defined exception (output).

stack of other errors." FALSE is the default value, and it replaces all previous errors. For example,

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
  RAISE_APPLICATION_ERROR
        (–20001, 'Department does not exist');
```

You can use a RAISE_APPLICATION_ERROR procedure in the execution and exception sections of the program. It is very useful in communicating errors between the client and the server.

In a PL/SQL program with an anonymous block that has nested blocks as well as procedures and functions, the outermost block nests other blocks and calls the procedures and functions. Each block can have its own exception-handling section, and some blocks may not have an exception-handling section. An exception declared in the inner block cannot be raised in the enclosing outer block. If an exception is declared in the outer block, it can be raised in the block itself or in its inner subblock. When the exception is raised implicitly or explicitly in an inner block without the exception-handling section, control shifts to the adjacent outer block and then propagates outward until its handler is found or it ends up being an unhandled exception.

The RAISE statement is very much like the GOTO statement. They both branch to another part of the program. The difference is that the RAISE statement branches to the exception section, whereas the GOTO statement branches to another statement in an executable block.

MORE SAMPLE PROGRAMS

In this section, you will see the PL/SQL blocks based on the topics covered in this chapter, such as an explicit cursor, a cursor FOR loop, a cursor with parameters, and exception handling. The code in Figure 12-14 uses an explicit cursor *emp_cur.* The active set contains the employee's last name, first name, salary, and commission. The WHILE loop is used to work with one row at a time. Within the loop, an employee's salary and commission are added together to find the total income. Also, note the use of a single-row function NVL in case the commission value is NULL. Finally, total company wages (the total of all employee salaries and commissions) are printed.

When the program in Figure 12-15 is executed, you will be prompted to enter the date for the substitution variable *p_date.* When the cursor is opened with *v_date* as a parameter, it will retrieve rows that have HireDate after the inputted date. The information for those employees will be printed. The program also will display the total number of employees selected.

The program in Figure 12-16 selects employees with PositionId of 2 who are managers. It locks those rows for future update. Using a cursor FOR loop, each manager's salary is modified to give a 7% raise. The WHERE CURRENT OF clause is used to modify the current row fetched. The rows are released with the COMMIT command.

The program in Figure 12-17 displays two customized prompts for an employee's ID and the percentage increment/raise. First, rows are locked with the FOR UPDATE clause. The UPDATE statement changes the salary if the ID is correct. If the employee ID does not exist, a standard exception is raised implicitly. The exception is handled by displaying an appropriate message.

IN A NUTSHELL . . .

- A cursor is a private work area to store a statement and its active set.
- A static cursor's contents are known at compile time, and a dynamic cursor uses a cursor variable, which can refer to different SQL statements at different times.
- An implicit cursor is declared, managed, and closed by PL/SQL.
- The programmer declares an explicit cursor for a PL/SQL block that returns more than one row from the table.

```
SQL> SET SERVEROUTPUT ON
SQL> /* program uses a cursor to get employee information.
DOC>Then prints total wages for each employee.
DOC>Program also prints total company wages. */
SQL> DECLARE
  2      CURSOR EMP_CUR IS
  3          SELECT LNAME, FNAME, SALARY, COMMISSION
  4          FROM EMPLOYEE WHERE SALARY IS NOT NULL;
  5      V_FIRST EMPLOYEE.FNAME%TYPE;
  6      V_LAST EMPLOYEE.LNAME%TYPE;
  7      V_SAL EMPLOYEE.SALARY%TYPE;
  8      V_COMM EMPLOYEE.COMMISSION%TYPE;
  9      V_TOTSAL EMPLOYEE.SALARY%TYPE;
 10      V_SUM EMPLOYEE.SALARY%TYPE := 0;
 11  BEGIN
 12      OPEN EMP_CUR;
 13      FETCH EMP_CUR INTO V_LAST, V_FIRST, V_SAL, V_COMM;
 14      WHILE EMP_CUR%FOUND LOOP
 15          V_TOTSAL := V_SAL + NvL(V_COMM, 0);
 16          V_SUM := V_SUM + V_TOTSAL;
 17          DBMS_OUTPUT.PUT(V_LAST || ', ' || V_FIRST);
 18          DBMS_OUTPUT.PUT_LINE
 19          (' makes ' || TO_CHAR(V_TOTSAL, '$999,999'));
 20          FETCH EMP_CUR INTO V_LAST,V_FIRST,V_SAL,V_COMM;
 21      END LOOP;
 22      DBMS_OUTPUT.PUT_LINE
 23      ('COMPANY WAGES: ' || TO_CHAR(V_SUM, '$999,999'));
 24  END;
 25  /
Smith, John makes  $300,000
Houston, Larry makes  $160,000
Roberts, Sandi makes  $75,000
McCall, Alex makes  $66,500
Dev, Derek makes  $100,000
Shaw, Jinku makes  $27,500
Garner, Stanley makes  $50,000
Chen, Sunny makes  $35,000
COMPANY WAGES:  $814,000

PL/SQL procedure successfully completed.

SQL>
```

Figure 12-14 Sample program—explicit cursor.

```
SQL> SET SERVEROUTPUT ON
SQL> /* Program uses a cursor with a date parameter.
DOC>User inputs a value for the parameter.
DOC>Program displays employees hired after inputted date */
SQL> DECLARE
  2      CURSOR EMP_CUR(START_DATE DATE) IS
  3          SELECT LNAME, FNAME, HIREDATE, DEPTNAME
  4          FROM EMPLOYEE E, DEPT D
  5          WHERE HIREDATE > START_DATE AND D.DEPTID=E.DEPTID;
  6      V_FIRST EMPLOYEE.FNAME%TYPE;
  7      V_LAST EMPLOYEE.LNAME%TYPE;
  8      V_HIRE EMPLOYEE.HIREDATE%TYPE;
  9      V_DEPT DEPT.DEPTNAME%TYPE;
 10      V_DATE EMPLOYEE.HIREDATE%TYPE := '&P_DATE';
 11  BEGIN
 12      OPEN EMP_CUR(V_DATE);
 13      FETCH EMP_CUR INTO V_LAST, V_FIRST, V_HIRE, V_DEPT;
 14      WHILE EMP_CUR%FOUND LOOP
 15          DBMS_OUTPUT.PUT(V_LAST || ', ' || V_FIRST);
 16          DBMS_OUTPUT.PUT
 17          (' was hired on ' || TO_CHAR(V_HIRE, 'MM/DD/YYYY'));
 18          DBMS_OUTPUT.PUT_LINE(' in ' || V_DEPT ||' department');
 19          FETCH EMP_CUR INTO V_LAST, V_FIRST, V_HIRE, V_DEPT;
 20      END LOOP;
 21      DBMS_OUTPUT.PUT_LINE
 22      ('TOTAL EMPLOYEES: ' || EMP_CUR%ROWCOUNT);
 23      CLOSE EMP_CUR;
 24  END;
 25  /
Enter value for p_date: 31-DEC-1995
McCall, Alex was hired on 05/10/1997 in InfoSys department
Shaw, Jinku was hired on 01/03/2000 in Sales department
Garner, Stanley was hired on 02/29/1996 in Sales department
Chen, Sunny was hired on 08/15/1999 in Finance department
TOTAL EMPLOYEES: 4

PL/SQL procedure successfully completed.

SQL>
```

Figure 12-15 Sample program—cursor with parameter.

```
SQL> /* Program uses a cursor to update salaries for any one
DOC>position. It locks rows with FOR UPDATE clause.
DOC>It updates row fetched with WHERE CURRENT OF clause */
SQL> DECLARE
  2      CURSOR SAL_CUR IS
  3          SELECT LNAME, SALARY
  4          FROM EMPLOYEE
  5          WHERE POSITIONID =
  6               (SELECT POSITIONID FROM POSITION
  7               WHERE UPPER(POSDESC) = '&POSITION')
  8          FOR UPDATE;
  9  BEGIN
 10      FOR SAL_REC IN SAL_CUR LOOP
 11          UPDATE EMPLOYEE
 12              SET SALARY = SALARY * 1.07
 13          WHERE CURRENT OF SAL_CUR;
 14      END LOOP;
 15      -- COMMIT;  /* can be uncommented for immediate update */
 16  END;
 17  /
Enter value for position: MANAGER

PL/SQL procedure successfully completed.

SQL>
```

Figure 12-16 Sample program—cursor FOR loop and WHERE CURRENT OF.

- Four actions are performed on explicit cursors: DECLARE, OPEN, FETCH, and CLOSE.
- The cursor attributes %ISOPEN, %FOUND, %NOTFOUND, and %ROWCOUNT give the status of the cursor.
- The cursor attributes that are used with an implicit cursor have *SQL* as a qualifier or a prefix (e.g., SQL%ISOPEN).
- A cursor FOR loop implicitly opens, fetches, and closes a cursor.
- An explicit cursor does not need to be declared if the cursor FOR loop uses a subquery to create a cursor.
- The SELECT . . . FOR UPDATE statement is used with a cursor to lock rows for future updates. These rows are released with a COMMIT or ROLLBACK statement.
- The WHERE CURRENT OF clause allows you to perform data manipulation on a recently fetched row.
- A cursor with parameters enables you to pass values to the cursor.

```
SQL> /* Program prompts user for employeeid and percentage rise.
DOC>If employeeid does not exist, exception is raised and handled */
SQL> DECLARE
  2      V_EMPID EMPLOYEE.EMPLOYEEID%TYPE;
  3      V_SAL EMPLOYEE.SALARY%TYPE;
  4      V_RAISE NUMBER(3,2) := &P_RAISE;
  5  BEGIN
  6      SELECT EMPLOYEEID, SALARY
  7      INTO V_EMPID, V_SAL FROM EMPLOYEE
  8      WHERE EMPLOYEEID = &P_EMPID FOR UPDATE NOWAIT;
  9      UPDATE EMPLOYEE
 10          SET SALARY = SALARY + SALARY * V_RAISE
 11      WHERE EMPLOYEEID = V_EMPID;
 12      DBMS_OUTPUT.PUT_LINE('SALARY UPDATED FOR EMPLOYEE ' || V_EMPID);
 13  EXCEPTION
 14      WHEN NO_DATA_FOUND THEN
 15          DBMS_OUTPUT.PUT_LINE('NO SUCH EMPLOYEEID IN TABLE');
 16  END;
 17  /
Enter value for p_raise: 0.05
Enter value for p_empid: 999
NO SUCH EMPLOYEEID IN TABLE

PL/SQL procedure successfully completed.

SQL> /
Enter value for p_raise: 0.10
Enter value for p_empid: 111
SALARY UPDATED FOR EMPLOYEE 111

PL/SQL procedure successfully completed.

SQL>
```

Figure 12-17 Sample program—exception handling.

- A cursor variable can be opened with different queries within the same program. All action statements and cursor attributes can be used with a cursor variable.
- PL/SQL errors are called exceptions, and they are handled in the exception section of the PL/SQL block.
- Three types of exceptions are predefined Oracle server exceptions, nonpredefined Oracle server exceptions, and user-defined exceptions.
- Predefined Oracle server exceptions are declared in the Oracle package called STANDARD. There are approximately 20 such exceptions.

- Nonpredefined Oracle server exceptions are declared with a PRAGMA EXCEPTION_INIT directive to associate an exception name with a standard error code.
- User-defined exceptions are declared, raised, and handled explicitly.
- The exception-trapping functions SQLCODE and SQLERRM return an error code and the associated error message, respectively.

EXERCISE QUESTIONS

True/False:

1. A cursor variable is a dynamic cursor that can refer to different SQL statements at different times.
2. An implicit cursor is used when an SQL statement in the PL/SQL block returns more than one row from the table.
3. The ORDER BY clause is not allowed in the SELECT statement of an explicit cursor's declaration.
4. If a cursor FOR loop uses a subquery with an IN clause, there is no need to declare that cursor.
5. A nonpredefined Oracle server error is declared with a PRAGMA EXCEPTION_INIT directive.
6. A user-defined exception is declared with a PRAGMA EXCEPTION_INIT directive.
7. The RAISE statement is used to raise a predefined Oracle server exception.
8. A cursor is based on a SELECT query, which is executed when the cursor is opened.
9. A record used in a cursor FOR loop must be declared in the DECLARE section.
10. A cursor FOR loop is opened, fetched from, and closed automatically.

State Differences Between the Following Terms:

1. Static cursor and dynamic cursor.
2. Implicit cursor and explicit cursor.
3. Predefined Oracle server exception and user-defined exception.
4. Nonpredefined Oracle server exception and user-defined exception.

Answer the Following Questions:

1. What actions can be performed on an explicit cursor? Give an example of each statement's use.
2. What are four cursor attributes? State their use.
3. Can you use cursor attributes with implicit cursors? If yes, how?
4. What is a cursor FOR loop? What are its benefits?
5. What are exceptions? Where are they handled?
6. Name the error-trapping functions. How are they useful?
7. How are the three types of exceptions declared, raised, and handled?

LAB ACTIVITY

1. Create a PL/SQL block to declare a cursor to select last name, first name, salary, and hire date from the EMPLOYEE table. Retrieve each row from the cursor, and print the employee's information if the employee's salary is greater than $50,000 and the hire date is before 31-DEC-1997 (explicit cursor problem).
2. Create a PL/SQL block that declares a cursor. Pass a parameter of the same type as the Salary column in the EMPLOYEE table to the cursor. Open the cursor with a value for the parameter. Retrieve information into the cursor for a salary higher than the parameter value. Use a loop to print each employee's information from the cursor (cursor with parameter problem).
3. Create a PL/SQL block to increase the salary of employees in department 10. The salary increase is 15% for employees making less than $100,000 and 10% for employees making $100,000 or more. Use a cursor with a FOR UPDATE clause. Update the salary with a WHERE CURRENT OF clause in a cursor FOR loop (cursor FOR loop problem).
4. Write a PL/SQL block to retrieve employees from the EMPLOYEE table based on a qualification ID. If the qualification ID returns more than one row, handle the exception with the appropriate handler, and print the message "More than one employee with such qualification." If the qualification ID returns no employee, handle the exception with the appropriate handler, and display the message "No employees with such qualification." If the qualification ID returns one employee, print that employee's name, qualification, and salary (predefined server exception problem).
5. Write a PL/SQL block that retrieves the entire COURSE table into a cursor. Then, ask the user to input a course ID to search. If the course exists, print its information. If the course does not exist, throw a user-defined exception, and display the message "Course does not exist" in the COURSE table when you execute the block with a course ID such as CIS555. (user-defined exception problem).
6. Write a PL/SQL block that asks the user to input first number, second number, and an arithmetic operator (+, −, *, or /). If the operator is invalid, throw and handle a user-defined exception. If the second number is zero and the operator is /, handle the ZERO_DIVIDE predefined server exception.

13

PL/SQL Composite Data Types: Records, Tables, and Varrays

IN THIS CHAPTER . . .

- You will learn about composite data types in PL/SQL.
- The basics of a PL/SQL record structure and its declaration, assignment of a value, and use in a program are covered.
- The PL/SQL composite data type of table is discussed, together with its declaration, referencing, and types of assignments.
- Built-in methods to obtain table information are outlined.
- A complex structure, a table of records, is covered.
- Variable-sized arrays, or varrays, are introduced.

COMPOSITE DATA TYPES

Composite data types are like scalar data types. Scalar data types are atomic, because they do not consist of a group. Composite data types, on the other hand, are groups, or "collections." Examples of composite data types are RECORD, TABLE, nested TABLE, and VARRAY. In this chapter, we will talk about all four composite data types.

PL/SQL RECORDS

PL/SQL records are similar in structure to a row in a database table. A record consists of components of any scalar, PL/SQL record, or PL/SQL table type. These components are known as fields, and they have their own values. PL/SQL records are similar in structure to "struct" in the C language. The record does not have a value as a whole; instead, it enables you to access these components as a group. It makes your life easier by transferring the entire row into a record rather than each column into a variable separately.

A PL/SQL record is based on a cursor, a table's row, or a user-defined record type. You learned about a record in a cursor FOR loop in the previous chapter. A record can be explicitly declared based on a cursor or a table:

```
CURSOR cursorname IS
        SELECT query;
Recordname CursorName%ROWTYPE;
```

A record can also be based on another composite data type called TABLE. We will examine user-defined records in the next section.

Creating a PL/SQL Record

In this section, you will learn to create a user-defined record. You create a RECORD type first, and then you declare a record with that RECORD type. The general syntax is

```
TYPE recordtypename IS RECORD
    (fieldname1 datatype | variable%TYPE | table.column%TYPE |
    table%ROWTYPE [[NOT NULL] := | DEFAULT Expression]
    [, fieldname2 . . .
    , FieldName3 . . . );
recordname recordtypename;
```

For example,

```
TYPE employee_rectype IS RECORD
    (e_last      VARCHAR2(15),
    e_first      VARCHAR2(15),
    e_sal        NUMBER(8,2));
employee_rec employee_rectype;
```

In this declaration, *employee_rectype* is the user-defined RECORD type. Three fields are included in its structure; *e_last, e_first*, and *e_sal*. The record *employee_rec* is a record declared with the user-defined record type *employee_rectype*. Each field declaration is similar to a scalar variable declaration.

Now, you will look at another declaration with the %TYPE attribute. For example,

```
TYPE employee_rectype IS RECORD
    (e_id          NUMBER(3) NOT NULL := 111,
    e_last         employee.Lname%TYPE,
    e_first        employee.Fname%TYPE,
    e_sal          employee.Salary%TYPE);
employee_rec       employee_rectype;
```

The NOT NULL constraint can be used for any field to prevent Null values, but that field must be initialized with a value.

Referencing Fields in a Record

A field in a record has a name that is given in the RECORD-type definition. You cannot reference a field by its name only; you must use the record name as a qualifier:

recordname.fieldname

The record name and field name are joined by a dot (.). For example, you can reference the *e_sal* field from the previous declaration as

```
employee_rec.e_sal
```

You can use a field in an assignment statement to assign a value to it. For example,

```
employee_rec.e_sal := 100000;
employee_rec.e_last := 'Jordan';
```

Working with Records

A record is known in the block where it is declared. When the block ends, the record no longer exists. You can assign values to a record from columns in a row by using the SELECT statement or the FETCH statement. The order of fields in a record must match the order of columns in the row. A record can be assigned to another record if both records have the same structure.

A record can be set to NULL, and all fields will be set to NULL. However, do not try to assign a NULL to a record that has fields with the NOT NULL constraint. For example,

```
Employee_rec := NULL;
```

In the previous chapter, you saw the use of the %ROWTYPE attribute. The record declared with %ROWTYPE has the same structure as the table's row. For example,

```
emp_rec      employee%ROWTYPE;
```

Here, *emp_rec* assumes the structure of the EMPLOYEE table. The fields in *emp_rec* take their column names and their data types from the table. It is advantageous to use %ROWTYPE, because it does not require you to know the column names and their data types in the underlying table. If you change the data type and/or size of a column, the record is created at execution time and is defined with the updated table structure. The fields in the record declared with %ROWTYPE are referenced with the qualified name *recordname.fieldname*.

The program in Figure 13-1 declares a record with a record type. The SELECT query retrieves a row into the record based on the student ID entered at the prompt. The fields in the record are printed using the *recordname.fieldname* notation.

```
SQL> DECLARE
  2       TYPE STUDENT_RECORD_TYPE IS RECORD
  3       (S_LAST    VARCHAR2(15),
  4       S_FIRST    VARCHAR2(15),
  5       S_PHONE  VARCHAR2(10));
  6       STUDENT_REC STUDENT_RECORD_TYPE;
  7  BEGIN
  8       SELECT LAST, FIRST, PHONE INTO STUDENT_REC
  9       FROM STUDENT WHERE STUDENTID = '&STUD_ID';
 10       DBMS_OUTPUT.PUT_LINE(STUDENT_REC.S_LAST || ', '
 11          || STUDENT_REC.S_FIRST || ' --> ' ||
 12          STUDENT_REC.S_PHONE);
 13  END;
 14  /
Enter value for stud_id: 00100
Diaz, Jose --> 9735551111

PL/SQL procedure successfully completed.

SQL>
```

Figure 13-1 PL/SQL record.

Nested Records

You can create a nested record by including a record into another record as a field. The record that contains another record as a field is called the **enclosing record**. For example,

```
DECLARE
  TYPE address_rectype IS RECORD
    (first      VARCHAR2(15),
    last        VARCHAR2(15),
```

```
  street      VARCHAR2(25),
  city        VARCHAR2(15),
  state       CHAR (2),
  zip         CHAR (5));
TYPE all_address_rectype IS RECORD
  (home_address       address_rectype,
  bus_address         address_rectype,
  vacation_address    address_rectype);
address_rec           all_address_rectype;
```

In this example, *all_address_rectype* nests *address_rectype* as a field type. If you decide to use an unnested simple record, the record becomes cumbersome. There are six fields in *address_rectype*. You will have to use six fields each for each of the three record fields *home_address, bus_address*, and *vacation_address*, which will result in a total of 18 fields.

Nesting records makes code more readable and easier to maintain. You can nest records to multiple levels. Dot notation is also used to reference fields in the nested situation. For example, *address_rec.home_address.city* references a field called *city* in the nested record *home_address*, which is enclosed by the record *address_rec*.

PL/SQL TABLES

A table, like a record, is a composite data structure in PL/SQL. A PL/SQL table is a single-dimensional structure with a collection of elements that store the same type of value. In other words, it is like an array in other programming languages. If you know COBOL, arrays are called tables in COBOL terminology, although there are dissimilarities between a traditional array and a PL/SQL table. A table is a dynamic structure that is not constrained, whereas an array is not dynamic in most computer languages.

Declaring a PL/SQL Table

A PL/SQL TABLE declaration is done in two steps, like a record declaration:

1. Declare a PL/SQL table type with a TYPE statement. The structure could use any of the scalar data types.
2. Declare an actual table based on the type declared in the previous step.

The general syntax is

```
TYPE tabletypename IS TABLE OF
    datatype | variablename%TYPE | tablename.columnname%TYPE
    [NOT NULL] INDEX BY BINARY_INTEGER;
```

For example,

```
TYPE deptname_table_type IS TABLE OF dept.DeptName%TYPE
    INDEX BY BINARY_INTEGER;
TYPE major_table_type IS TABLE OF VARCHAR2(50)
    INDEX BY BINARY_INTEGER;
```

You can declare a table type with a scalar data type (VARCHAR2, DATE, BOOLEAN, or POSITIVE) or with the declaration attribute %TYPE. Optionally, you can use NOT NULL in a declaration, which means that none of the elements in the table may have a Null value. You must, however, add an INDEX BY BINARY_INTEGER clause to the declaration. This is the only available clause for indexing a table at present. Indexing speeds up the search process from the table. The primary key is stored internally in the table along with the data column. The table consists of two columns, the index/primary key column and the data column.

You define the actual table based on the table type declared earlier. The general syntax is

tablename tabletypename;

For example,

```
deptname_table deptname_table_type;
major_table major_table_type;
```

Figure 13-2 illustrates a table's structure. It contains a primary key column and a data column. You cannot name these columns. The primary key has the type BINARY_INTEGER, and the data column is of any valid type. There is no limit on the number of elements, but you cannot initialize elements of a table at declaration time.

Primary Key Column	Data Column
. . .	. . .
1	Sales
2	Marketing
3	Information Systems
4	Finance
5	Production
. . .	. . .

Figure 13-2 PL/SQL table structure.

Referencing Table Elements/Rows

The rows in a table are referenced in the same way that an element in an array is referenced. You cannot reference a table by its name only. You must use the primary key value in a pair of parentheses as its subscript or index:

tablename (primarykeyvalue)

The following are valid assignments for the table's rows:

```
deptname_table(5) := 'Human Resources';
major_table(100) := v_major;
```

You can use an expression or a value other than a BINARY_INTEGER value, and PL/SQL will convert it. For example,

```
/* 25.7 is rounded to 26. */
deptname_table(25.7) := 'Training';
/* '5' || '00' is converted to 500. */
deptname_table ('5' || '00') := 'Research';
/* v_num + 7 is evaluated. */
deptname_table(v_num + 7) := 'Development';
```

In other programming languages, such as C or Visual Basic, you specify the number of elements in an array when you declare the array. The memory locations are reserved for elements in an array at the declaration time. In a PL/SQL table, the primary key values are not preassigned. A row is created when you assign a value to it. If a row does not exist and you try to access it, the PL/SQL predefined server exception NO_DATA_FOUND is raised. You can keep track of rows' primary key values if you use them in a sequence and keep track of the minimum and the maximum value.

Assigning Values to Rows in a PL/SQL Table

You can assign values to the rows in a table in three ways:

1. Direct assignment.
2. Assignment in a loop.
3. Aggregate assignment.

Direct Assignment. You can assign a value to a row with an assignment statement, as you already learned in the previous topic. This is preferable if only a few assignments are to be made. If an entire database table's values are to be assigned to a table, however, a looping method is preferable.

Assignment in a Loop. You can use any of the three PL/SQL loops to assign values to rows in a table. The program block in Figure 13-3 assigns all Sunday dates for the year 2004 to a table. The primary key index value will vary from 1 to 52. The table column will contain dates for 52 Sundays. If you are innovative, you can create great applications with loops and tables.

```
SQL> DECLARE
  2       TYPE DATE_TABLE_TYPE IS TABLE OF DATE
  3     INDEX BY BINARY_INTEGER;
  4       SUNDAY_TABLE DATE_TABLE_TYPE;
  5       V_DAY BINARY_INTEGER := 1;
  6       V_DATE DATE;
  7       V_COUNT NUMBER(3) := 1;
  8   BEGIN
  9       V_DATE := '01-JAN-2004';
 10       WHILE V_COUNT <= 365 LOOP
 11       IF UPPER(TO_CHAR(V_DATE, 'DAY')) LIKE '%SUNDAY%' THEN
 12          SUNDAY_TABLE(V_DAY) := V_DATE;
 13             DBMS_OUTPUT.PUT_LINE
 14             (TO_CHAR(SUNDAY_TABLE(V_DAY), 'MONTH DD, YYYY'));
 15          V_DAY := V_DAY +1;
 16       END IF;
 17       V_COUNT := V_COUNT +1;
 18       V_DATE := V_DATE +1;
 19       END LOOP;
 20   END;
 21   /
JANUARY   04, 2004
JANUARY   11, 2004
JANUARY   18, 2004
JANUARY   25, 2004
FEBRUARY  01, 2004
FEBRUARY  08, 2004
FEBRUARY  15, 2004
FEBRUARY  22, 2004
FEBRUARY  29, 2004
. . .
DECEMBER  19, 2004
DECEMBER  26, 2004

PL/SQL procedure successfully completed.

SQL>
```

Figure 13-3 Table row assignment in a loop.

Aggregate Assignment. You can assign a table's values to another table. The data types of both tables must be compatible. When you assign a table's values to another table, the table receiving those values loses all its previous primary key values as well as its data column values. If you assign an empty table with no rows to another table with rows, the recipient table is cleared. In other words, it loses all its rows. Both tables must have the same type for such an assignment.

Built-In Table Methods

The built-in table methods are procedures or functions that provide information about a PL/SQL table. Figure 13-4 lists the built-in table methods and their use. The general syntax is

tablename.methodname [(index1 [, index2])]

where *methodname* is one of the methods described in Figure 13-4. For example,

```
total_rows := deptname_table.COUNT;          /* counts elements */
deptname_table.DELETE(5);                    /* deletes element 5 */
deptname_table.DELETE(7, 10);                /* deletes element 7 to 10 */
deptname_table.DELETE;                       /* deletes all elements */
next_row := deptname_table.NEXT(25);         /* index after 25 */
previous_row := deptname_table.PRIOR(7);     /* index before 7 */
first_row := deptname_table.FIRST;           /* smallest index */
last_row := deptname_table.LAST;             /* largest index */
IF deptname_table.EXISTS(11) THEN . . .      /* true, if  index 11 exists */
```

Built-in Method	Use
FIRST	Returns the smallest index number in a PL/SQL table.
LAST	Returns the largest index number in a PL/SQL table.
COUNT	Returns the total number of elements in a PL/SQL table.
PRIOR(*n*)	Returns the index number that is before index number *n*.
NEXT(*n*)	Returns the index number that is after index number *n*.
EXISTS(*n*)	Returns TRUE if index *n* exists in the table.
TRIM	Removes one element from end of the table.
TRIM (*n*)	Removes *n* elements from end of the table.
DELETE	Removes all elements from a PL/SQL table.
DELETE (*n*)	Removes the *n*-th element from the table.
DELETE (*m*, *n*)	Removes all elements in the range *m* . . . *n* from a table.
EXTEND	Appends a null element to a table.
EXTEND (*n*)	Appends *n* null elements to a table.
EXTEND (*n*, *x*)	Appends *n* copies of the *x*-th element to a table.

Figure 13-4 PL/SQL built-in table methods.

Now, look at another example of TABLE, in Figure 13-5, where the program declares two table types. The two tables based on these TABLE types are parallel tables. The corresponding values in the two tables are related. The program populates these two tables using a cursor FOR loop. Then, another simple FOR loop is used to print information from the two tables.

```
SQL> DECLARE
  2       TYPE LNAME_TABLE_TYPE IS TABLE OF EMPLOYEE.LNAME%TYPE
  3               INDEX BY BINARY_INTEGER;
  4       NAME_TABLE LNAME_TABLE_TYPE;
  5       TYPE SALARY_TABLE_TYPE IS TABLE OF EMPLOYEE.SALARY%TYPE
  6               INDEX BY BINARY_INTEGER;
  7       SALARY_TABLE SALARY_TABLE_TYPE;
  8       C BINARY_INTEGER := 0;
  9       CURSOR C_LASTSAL IS
 10          SELECT LNAME, SALARY FROM EMPLOYEE;
 11       V_TOT NUMBER(2);
 12  BEGIN
 13       SELECT COUNT(*) INTO V_TOT FROM EMPLOYEE;
 14       /*  TABLE ASSIGNMENT IN LOOP */
 15       FOR LASTSAL_REC IN C_LASTSAL LOOP
 16          C := C+1;
 17          NAME_TABLE(C) := LASTSAL_REC.LNAME;
 18          SALARY_TABLE(C) := LASTSAL_REC.SALARY;
 19    END LOOP;
 20       /*    PRINTING TABLE IN LOOP */
 21       FOR C IN 1..V_TOT LOOP
 22          DBMS_OUTPUT.PUT_LINE(NAME_TABLE(C) || ' ' ||
 23          TO_CHAR(SALARY_TABLE(C), '$999,999.99'));
 24    END LOOP;
 25  END;
 26  /
Smith    $291,500.00
Houston    $160,500.00
Roberts    $80,250.00
McCall    $66,500.00
Dev    $85,600.00
Shaw    $24,500.00
Garner    $48,150.00
Chen    $35,000.00

PL/SQL procedure successfully completed.

SQL>
```

Figure 13-5 PL/SQL table.

Table of Records

The PL/SQL table type is declared with a data type. You may use a record type as a table's data type. The %ROWTYPE declaration attribute can be used to define the record type. When a table is based on a record, the record must consist of fields with

scalar data types. The record must not contain a nested record. The following examples show different ways to declare table types based on records:

- *A PL/SQL table type based on a programmer-defined record:*

```
TYPE student_record_type IS
    RECORD (stu_id NUMBER(3), stu_name VARCHAR2(30));
TYPE student_table_type IS TABLE OF student_record_type
    INDEX BY BINARY_INTEGER;
Student_table student_table_type;
```

- *A PL/SQL table type based on a database table:*

```
TYPE employee_table_type IS TABLE OF employee%ROWTYPE
   INDEX BY BINARY_INTEGER;
Employee_table employee_table_type;
```

- *A PL/SQL table type based on a row returned by a cursor:*

```
CURSOR employee_cur IS SELECT * FROM employee;
TYPE employee_cur_table_type IS employee_cur%ROWTYPE
    INDEX BY BINARY_INTEGER;
Employee_cur_table employee_cur_table_type;
```

The %ROWTYPE attribute is not used when the table is based on a user-defined record. You use the %ROWTYPE attribute when the table is based on a database table or a cursor.

The fields of a PL/SQL table based on a record are referenced with the following syntax:

```
tablename (index).fieldname
```

For example,

```
Student_table(10).stu_name := 'Ephrem';
Employee_table(13).Salary := 50000;
```

PL/SQL VARRAYS

A **varray** is another composite data type or collection type in PL/SQL. Varray stands for variable-size array. They are single-dimensional, bounded collections of elements with the same data type. They retain their ordering and subscripts when stored in and retrieved from a database table. They are similar to a PL/SQL table, and each element is assigned a subscript/index starting with 1.

A PL/SQL VARRAY declaration is done in two steps, like a table declaration:

1. Declare a PL/SQL VARRAY type with a TYPE statement. The TYPE declaration includes a size to set the upper bound of a varray. The lower bound is always one.
2. Declare an actual varray based on the type declared in the previous step.

The general syntax is

```
DECLARE
    TYPE varraytypename IS VARRAY (size) OF ElementType [NOT NULL];
    varrayname varraytypename;
```

For example,

```
DECLARE
    TYPE Lname_varray_type IS VARRAY(5) OF employee.LName%TYPE;
    Lname_varray  Lname_varray_type := Lname_varray_type( );
```

When a varray is declared, it is NULL. It must be initialized before referencing its elements. In the second step of a varray's declaration, the assignment part initializes it. The EXTEND method is used before adding a new element to a varray. In the example above, the upper bound would be five, which limits number of elements to five.

In Figure 13-6, COURSEID_VARRAY_TYPE is declared with upper bound of 10. Next, the COURSEID_VARRAY is declared with the varray type and then initialized. A cursor FOR loop then adds elements to the varray. Notice the use of the EXTEND method before assigning a value to the new element. The COUNT method returns the number of elements, the LIMIT method the upper bound, the FIRST method the first subscript, and the LAST method the last subscript.

In Oracle9i, it is possible to create a collection of a collection (multilevel collection) like a varray of varrays. For example,

```
DECLARE
   TYPE varray_type1 IS VARRAY(3) OF NUMBER;
   TYPE varray_type2 IS VARRAY(2) of varray_type1;
```

In Figure 13-7, V1 is a varray, and V2 is a varray of varray V1. Varray V1 contains three elements, and varray V2 contains six elements $(2 \cdot 3 = 6)$. Elements of varray V1 are referenced with one subscript, but elements of varray V2 are referenced with two subscripts.

There is one more type of collection in PL/SQL. ***Nested tables*** use a column that has a table type as its data type and are single-dimensional, unbounded collections of

elements with the same data type. A nested table can be used in PL/SQL as well as a database table. You can use a column that has a table type as its data type in a database table.

```
SQL> DECLARE
  2    CURSOR COURSE_CUR IS
  3        SELECT COURSEID
  4        FROM COURSE
  5        WHERE ROWNUM <= 5;
  6    TYPE COURSEID_VARRAY_TYPE IS VARRAY(10) OF COURSE.COURSEID%TYPE;
  7    COURSEID_VARRAY COURSEID_VARRAY_TYPE := COURSEID_VARRAY_TYPE();
  8    V_COUNT NUMBER(1) := 1;
  9  BEGIN
 10      FOR COURSE_REC IN COURSE_CUR LOOP
 11        COURSEID_VARRAY.EXTEND;
 12        COURSEID_VARRAY(V_COUNT) := COURSE_REC.COURSEID;
 13        DBMS_OUTPUT.PUT_LINE
 14           ('Courseid(' || V_COUNT || '): '
 15                  || COURSEID_VARRAY(V_COUNT));
 16        V_COUNT := V_COUNT + 1;
 17      END LOOP;
 18      DBMS_OUTPUT.PUT_LINE
 19            ('NUMBER OF ELEMENTS: ' || COURSEID_VARRAY.COUNT);
 20      DBMS_OUTPUT.PUT_LINE
 21            ('LIMIT ON ELEMENTS: ' || COURSEID_VARRAY.LIMIT);
 22      DBMS_OUTPUT.PUT_LINE
 23            ('FIRST ELEMENTS: ' || COURSEID_VARRAY.FIRST);
 24      DBMS_OUTPUT.PUT_LINE
 25            ('LAST ELEMENTS: ' || COURSEID_VARRAY.LAST);
 26  END;
 27  /
Courseid(1): EN100
Courseid(2): LA123
Courseid(3): CIS253
Courseid(4): CIS265
Courseid(5): MA150
NUMBER OF ELEMENTS: 5
LIMIT ON ELEMENTS: 10
FIRST ELEMENTS: 1
LAST ELEMENTS: 5

PL/SQL procedure successfully completed.

SQL>
```

Figure 13-6 PL/SQL Varray.

```
SQL> DECLARE
  2       TYPE V_TYPE1 IS VARRAY(3) OF NUMBER;
  3       TYPE V_TYPE2 IS VARRAY(2) OF V_TYPE1;
  4       V1 V_TYPE1 := V_TYPE1(10, 20, 30);
  5       V2 V_TYPE2 := V_TYPE2(V1);
  6  BEGIN
  7       DBMS_OUTPUT.PUT_LINE('VARRAY:');
  8       FOR I IN 1..3 LOOP
  9         DBMS_OUTPUT.PUT_LINE('V1(' || I || ')=' || V1(I));
 10       END LOOP;
 11       V2.EXTEND;
 12       V2(2) := V_TYPE1(100, 200, 300);
 13       DBMS_OUTPUT.PUT_LINE('VARRAY OF VARRAY:');
 14    FOR I IN 1..2 LOOP
 15       FOR J IN 1..3 LOOP
 16       DBMS_OUTPUT.PUT_LINE
 17       ('V2(' || I || ')(' || J || ')=' || V2(I)(J));
 18        END LOOP;
 19    END LOOP;
 20  END;
 21  /
VARRAY:
V1(1) = 10
V1(2) = 20
V1(3) = 30
VARRAY OF VARRAY:
V2(1)(1) = 10
V2(1)(2) = 20
V2(1)(3) = 30
V2(2)(1) = 100
V2(2)(2) = 200
V2(2)(3) = 300

PL/SQL procedure successfully completed.

SQL>
```

Figure 13-7 Varray of varrays.

IN A NUTSHELL . . .

- PL/SQL has composite data types, which are data types like scalar data types. The composite data types consist of groups or collections.
- PL/SQL composite data types include records, tables, nested tables, and varrays.

- PL/SQL records are similar in structure to rows in the database table. They consist of components of any scalar type, PL/SQL record type, or PL/SQL table type.
- Components in a PL/SQL record are called fields.
- A record declaration is performed in two steps. First, a record type is declared. Then, a record is declared with the declared record type.
- The fields in a record are referenced with the record name as a qualifier (e.g., *recordname.columnname*).
- A record's fields can be assigned values using a simple assignment statement with SELECT or FETCH.
- A nested record is a record used as a field in another record. The record containing another record is called the enclosing record.
- A PL/SQL table is another composite data type. It is a single-dimensional structure with a collection of elements that store the same type of values.
- A table is like an array, but a table is unbounded.
- A table declaration is performed in two steps. First, a table type is declared. Then, a table is declared with that table type.
- A table consists of two columns, a primary key column and a data column. The primary key is of type BINARY_INTEGER.
- A table element is referenced by the table name with its index number parentheses.
- There are three ways to assign values to a table's elements: direct assignment, assignment in a loop, and aggregate assignment.
- The table's information is obtained with built-in methods, which are PL/SQL functions and procedures. The methods are used with a table name as a qualifier (e.g., *tablename.method*).
- A table of records is declared with a record type as the table's data type. It can be declared with a record type as its type, with a database table and %ROWTYPE, or with a row returned by a cursor.
- Fields in a table of records are referenced with *tablename(index).fieldname*.
- Nested tables use a column that has a table type as its data type. They are single-dimensional, unbounded collections of elements.
- Varrays are single-dimensional, bounded collections of elements. They retain their ordering and subscripts when stored and retrieved from a database table.
- A varray must be initialized before referencing its elements. The EXTEND method is used before adding a new element to a varray.
- The COUNT, LIMIT, FIRST, and LAST varray methods return count, upper bound, first subscript, and last subscript, respectively.
- A varray of varrays is a multilevel collection, with elements that can be referenced using two subscripts.

EXERCISE QUESTIONS

True/False:

1. A database table is an example of a PL/SQL composite data type.
2. A PL/SQL record can be set to a Null value with an assignment, in which case all its fields are set to NULL.
3. A record that contains another record as a field is called the enclosing record.
4. A PL/SQL table has a specified number of rows, which cannot be changed, at declaration time.
5. A PL/SQL table has two columns, a primary key column and a data column.
6. A PL/SQL table's row is referenced by the table name and a numeric index.
7. When a PL/SQL table is assigned to another PL/SQL table of the same type, the recipient table loses its previous rows and indexes.
8. A varray's COUNT must be higher than its LIMIT.
9. Once declared, a varray's size limit cannot be changed.
10. A varray must be initialized before adding an element to it.

Answer the Following Questions:

1. How do you declare a PL/SQL record? Explain with an example.
2. Give examples of the assignment of values to fields in a PL/SQL record.
3. What is a PL/SQL table? What are its similarities and differences between a PL/SQL table and an array?
4. How do you declare a PL/SQL table? Explain with an example.
5. Give three methods used in assigning values to the rows in a PL/SQL table.
6. What is a table of records?
7. What is a varray? What are the similarities and differences between a varray and a PL/SQL table?
8. How is a varray of varrays declared? How are its elements referenced?

LAB ACTIVITY

1. Create a PL/SQL block to retrieve last name, first name, and salary from the EMPLOYEE table into a cursor. Populate three tables with the values retrieved into the cursor: one to store last names, one to store first names, and one to store salaries. Use a loop to retrieve this information from the three tables, and print it to the screen using the package DBMS_OUTPUT.PUT_LINE.
2. Declare a PL/SQL record based on the structure of the DEPT table. Use a substitution variable to retrieve information about a specific department, and store it in the PL/SQL record. Print the record information using DBMS_OUTPUT.PUT_LINE.

3. Use a PL/SQL table of records to retrieve all information from the DEPT table, and print the information to the screen. You will declare a table to store names and locations of the departments. Remember that the department number is a multiple of 10. Retrieve all department information from the DEPT table to the PL/SQL table using a loop, and then use another loop to print the information.
4. Use a PL/SQL cursor to retrieve all CourseId and Title information from the COURSE table. Create two varrays to hold CourseId and Title values, respectively. Add elements to both varrays, and assign values retrieved into the cursor. Display the values from both varrays.

14

PL/SQL Named Blocks: Procedure, Function, Package, and Trigger

IN THIS CHAPTER . . .

- You will learn about PL/SQL modules, also known as named blocks.
- The basics of a named module called a procedure (its call, its body, and its parameter types) are explained.
- The PL/SQL module called a function is covered, along with its call, body, and RETURN types.
- Package structure, specification, body, benefits, and call to its blocks are discussed.
- Triggers, their types, and their functioning are introduced.

In previous chapters dealing with PL/SQL, an anonymous block was used for all programming examples. The anonymous block does not have a name, and it cannot be called by another block in the program. It cannot take arguments from another block, either. An anonymous block can call other types of PL/SQL blocks called **procedures** and **functions**. The procedures and functions are **named blocks**, and they can be called with parameters. An anonymous block can be nested within a procedure, function, or another anonymous block. The purpose of a procedure or function call is to modularize a PL/SQL program. A named PL/SQL block is compiled when it is created or when it is altered. The compilation process consists of three steps: syntax error checking, binding, and p-code creation. A syntactically

error-free program's variables are assigned storage in the binding stage. Then, the list of instructions, called p-code, is generated for the PL/SQL engine. P-code is stored in the database for all named blocks.

PROCEDURES

A procedure is a named PL/SQL program block that can perform one or more tasks. A procedure is the building block of modular programming. The general syntax of a procedure is

```
CREATE [OR REPLACE] PROCEDURE procedurename
   [ (parameter1 [, parameter2 . . .]) ]
IS
   [ constant/variable declarations ]
BEGIN
   executable statements
[ EXCEPTION
   exception handling statements ]
END [ procedurename ];
```

where *procedurename* is a user-supplied name that follows the rules used in naming identifiers. The parameter list has the names of parameters passed to the procedure by the calling program as well as the information passed from the procedure to the calling program. The local constants and variables are declared after the reserved word IS. If there are no local identifiers to declare, there is nothing between the reserved words IS and BEGIN. The executable statements are written after BEGIN and before EXCEPTION or END. There must be at least one executable statement in the body. The reserved word EXCEPTION and the exception-handling statements are optional.

Calling a Procedure

A call to the procedure is made through an executable PL/SQL statement. The procedure is called by specifying its name along with the list of parameters (if any) in parentheses. The call statement ends with a semicolon (;). The general syntax is

```
procedurename [ (parameter1, . . . ) ];
```

For example,

```
monthly_salary(v_salary);
calculate_net(v_monthly_salary, 0.28);
display_messages;
```

In these examples of procedure calls, parameters are enclosed in parentheses. You can use a variable, constant, expression, or literal value as a parameter. If you are not passing any parameters to a procedure, parentheses are not needed.

Procedure Header

The procedure definition that comes before the reserved word IS is called the procedure header. The procedure header contains the name of the procedure and the parameter list with data types (if any). For example,

```
CREATE OR REPLACE PROCEDURE monthly_salary
 (v_salary_in IN employee.Salary%TYPE)
CREATE OR REPLACE PROCEDURE calculate_net
 (v_monthly_salary_in IN employee.Salary%TYPE,
 v_taxrate_in IN NUMBER)
CREATE OR REPLACE PROCEDURE display_messages
```

The procedure headers in the examples are based on the procedure calls shown previously. The parameter list in the header contains the name of a parameter along with its type. The parameter names used in the procedure header do not have to be the same as the names used in the call. The number of parameters in the call and in the header must match, and the parameters must be in the same order.

Procedure Body

The procedure body contains declaration, executable, and exception-handling sections. The declaration and exception-handling sections are optional. The executable section contains action statements, and it must contain at least one.

The procedure body starts after the reserved word IS. If there is no local declaration, IS is followed by the reserved word BEGIN. The body ends with the reserved word END. There can be more than one END statement in the program, so it is a good idea to use the procedure name as the optional label after END.

Parameters

Parameters are used to pass values back and forth from the calling environment to the Oracle server. The values passed are processed and/or returned with a procedure execution. There are three types of parameters: IN, OUT, and IN OUT. Figure 14-1 shows the uses of these parameters.

Parameter Type	Use
IN	Passes a value into the program; read-only type of value; it cannot be changed; default parameter type. For example, constants, literal, and expressions can be used as IN parameters.
OUT	Passes a value back from the program; write-only type of value; cannot assign a default value. If a program is successful, value is assigned. For example, a variable can be used as OUT parameter.
IN OUT	Passes a value in and returns a value back; value is read from and then written to. For example, a variable can be used as a IN OUT parameter.

Figure 14-1 Types of parameters.

Actual and Formal Parameters

The parameters passed in a call statement are called the **actual parameters**. The parameter names in the header of a module are called the **formal parameters**. The actual parameters and their matching formal parameters must have the same data types. In a procedure call, the parameters are passed without data types. The procedure header contains formal parameters with data types, but the size of the data type is not required. Figure 14-2 shows the relationship between actual and formal parameters.

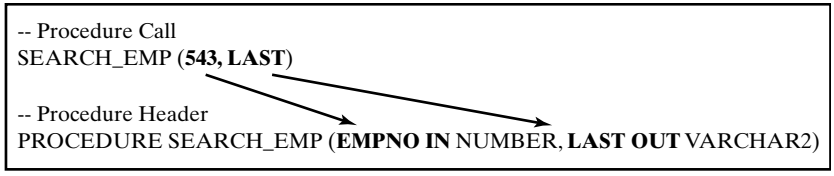


Figure 14-2 Actual and formal parameters.

Matching Actual and Formal Parameters

There are two different ways in PL/SQL to link formal and actual parameters:

1. In ***positional notation***, the formal parameter is linked with an actual parameter implicitly by position (see Fig. 14-2). Positional notation is more commonly used for parameter matching.
2. In ***named notation***, the formal parameter is linked with an actual parameter explicitly by name. The formal parameter and actual parameters (the values of the parameters) are linked in the call statement with the symbol =>.

The general syntax is

```
formalparametername => argumentvalue
```

For example,

```
EMPNO => 543
```

In Figure 14-3, a procedure code is shown. If a procedure with the same name already exists, it is replaced. You can type it in any editor such as Notepad. When you run it, a "Procedure created" message is displayed. The procedure named *dependent_info* is compiled into p-code and then stored in the database for future execution.

You can execute this procedure from the SQL*Plus environment (SQL> prompt) with the EXECUTE command. For example,

```
SQL> EXECUTE dependent_info
```

```
SQL> CREATE OR REPLACE PROCEDURE DEPENDENT_INFO
  2  IS
  3      CURSOR DEP_CUR IS
  4          SELECT LNAME, FNAME, COUNT(DEPENDENTID) CNT
  5          FROM EMPLOYEE E, DEPENDENT D
  6          WHERE E.EMPLOYEEID = D.EMPLOYEEID
  7          GROUP BY LNAME, FNAME;
  8  BEGIN
  9      FOR DEP_REC IN DEP_CUR LOOP
 10          IF DEP_REC.CNT >= 2 THEN
 11              DBMS_OUTPUT.PUT_LINE(DEP_REC.LNAME || ', ' ||
 12              DEP_REC.FNAME||' has '||DEP_REC.CNT||' dependents');
 13          END IF;
 14      END LOOP;
 15  END;
 16  /

Procedure created.

SQL> EXECUTE DEPENDENT_INFO
Dev, Derek has 2 dependents
Chen, Sunny has 3 dependents

PL/SQL procedure successfully completed.

SQL>
```

Figure 14-3 Procedure without parameters.

If you receive an error, use the following command:

```
SHOW ERROR
```

The procedure becomes invalid if the table on which it is based is deleted or altered. The compiled version of the procedure is stored, and this version should be re-compiled in case of alterations to the table(s). You can recompile that procedure with the following command:

```
ALTER PROCEDURE procedurename COMPILE;
```

In Figure 14-4, the procedure *search_emp* receives three parameters—*i_empid, o_last,* and *o_first*—as IN, OUT, and OUT types, respectively. Parameter *i_empid* is used for input/reading, and parameters *o_last* and *o_first* are used for writing. In Figure 14-5, you will see an anonymous block that calls the procedure in Figure 14-4 with three parameters. The procedure searches for the employee's name based on the V_ID that is passed. If the employee is not found, the exception is handled in the procedure. If the employee is found, the procedure sends out the last name and first name

```
SQL> CREATE OR REPLACE PROCEDURE SEARCH_EMP
  2       (I_EMPID IN NUMBER,
  3        O_LAST OUT VARCHAR2,
  4        O_FIRST OUT VARCHAR2)
  5  IS
  6  BEGIN
  7       SELECT LNAME, FNAME
  8          INTO O_LAST, O_FIRST
  9       FROM EMPLOYEE
 10       WHERE EMPLOYEEID = I_EMPID;
 11  EXCEPTION
 12    WHEN OTHERS THEN
 13    DBMS_OUTPUT.PUT_LINE('EmployeeId ' ||
 14         TO_CHAR(I_EMPID) || ' does not exist');
 15  END SEARCH_EMP;
 16  /

Procedure created.

SQL>
```

Figure 14-4 Procedure with parameters.

```
SQL> DECLARE
  2       V_LAST     EMPLOYEE.LNAME%TYPE;
  3       V_FIRST    EMPLOYEE.FNAME%TYPE;
  4       V_ID       EMPLOYEE.EMPLOYEEID%TYPE := &EMP_ID;
  5  BEGIN
  6       SEARCH_EMP(V_ID, V_LAST, V_FIRST);
  7       IF V_LAST IS NOT NULL THEN
  8            DBMS_OUTPUT.PUT_LINE('Employee: ' || V_ID);
  9            DBMS_OUTPUT.PUT_LINE('Name: ' || V_LAST || ', ' || V_FIRST);
 10       END IF;
 11  END;
 12  /
Enter value for emp_id: 100
EmployeeId 100 does not exist

PL/SQL procedure successfully completed.

SQL> /
Enter value for emp_id: 200
Employee: 200
Name: Shaw, Jinku

PL/SQL procedure successfully completed.

SQL>
```

Figure 14-5 Anonymous block with call to procedure in Figure 14-4.

of the employee to the calling anonymous block. The anonymous block then prints the employee's information.

FUNCTIONS

A function, like a procedure, is a named PL/SQL block. Like a procedure, it is also a stored block. The main difference between a function and a procedure is that a function always returns a value to the calling block. A function is characterized as follows:

- A function can be passed zero or more parameters.
- A function must have an explicit RETURN statement in the executable section to return a value.
- The data type of the return value must be declared in the function's header.
- A function cannot be executed as a stand-alone program.

A function may have parameters of the IN, OUT, and IN OUT types, but the primary use of a function is to return a value with an explicit RETURN statement. The use of OUT and IN OUT parameter types in functions is rare—and considered to be a bad practice. The general syntax is

```
CREATE [ OR REPLACE ] FUNCTION functionname
        [ (parameter1 [, parameter2 . . . ]) ]
        RETURN DataType
IS
        [ constant | variable declarations ]
BEGIN
        executable statements
        RETURN returnvalue
[ EXCEPTION
        exception-handling statements
        RETURN returnvalue ]
END [ functionname ];
```

The RETURN statement does not have to be the last statement in the body of a function. The body may contain more than one RETURN statement, but only one is executed with each function call. If you have RETURN statements in the exception section, you need one return for each exception.

Function Header

The function header comes before the reserved word IS. The header contains the name of the function, the list of parameters (if any), and the RETURN data type.

Function Body

The body of a function must contain at least one executable statement. If there is no declaration, the reserved word BEGIN follows IS. If there is no exception handler, you can omit the word EXCEPTION. The function name label next to END is optional. There can be more than one return statement, but only one RETURN is executed in a function call.

RETURN Data Types

A function can return a value with a scalar data type, such as VARCHAR2, NUMBER, BINARY_INTEGER, or BOOLEAN. It can also return a composite or complex data type, such as a PL/SQL table, a PL/SQL record, a nested table, VARRAY, or LOB.

Calling a Function

A function call is similar to a procedure call. You call a function by mentioning its name along with its parameters (if any). The parameter list is enclosed within parentheses. A procedure does not have an explicit RETURN statement, so a procedure call can be an independent statement on a separate line. A function does return a value, so the function call is made via an executable statement, such as an assignment, selection, or output statement. For example,

```
v_salary := get_salary(&emp_id);
IF emp_exists(v_empid) . . .
```

In the first example of a function call, the function *get_salary* is called from an assignment statement with the substitution variable *emp_id* as its actual parameter. The function returns the employee's salary, which is assigned to the variable *v_salary*.

In the second example, the function call to the function *emp_exists* is made from an IF statement. The function searches for the employee and returns a Boolean TRUE or FALSE to the statement.

An anonymous block calls *get_deptname* function of Figure 14-6 in Figure 14-7, with an employee's department number as a parameter. The function returns the department name back to the calling block. The calling block then prints the employee's information along with the department name.

In this example, the code of function *get_deptname* is executed in SQL*Plus, which returns a "Function created" message if the function code has no syntactical errors. Then, the calling anonymous block is executed, which calls the compiled function *get_deptname* with the *v_deptid* parameter of the NUMBER type. The function searches through the DEPT table, retrieves the corresponding department name, and returns *v_deptname*, which is assigned to *v_dept_name* in the anonymous block.

```
SQL> CREATE OR REPLACE FUNCTION GET_DEPTNAME
  2       (I_DEPTID IN NUMBER)
  3        RETURN VARCHAR2
  4  IS
  5       V_DEPTNAME VARCHAR2(12);
  6  BEGIN
  7       SELECT DEPTNAME
  8          INTO V_DEPTNAME
  9       FROM DEPT
 10       WHERE DEPTID = I_DEPTID;
 11       RETURN V_DEPTNAME;
 12  END GET_DEPTNAME;
 13  /

Function created.

SQL>
```

Figure 14-6 Function with parameters.

```
SQL> DECLARE
  2       V_DEPTID          EMPLOYEE.DEPTID%TYPE;
  3       V_DEPT_NAME       VARCHAR2(12);
  4       V_EMPID           EMPLOYEE.EMPLOYEEID%TYPE := &EMP_ID;
  5  BEGIN
  6       SELECT DEPTID
  7       INTO V_DEPTID FROM EMPLOYEE
  8       WHERE EMPLOYEEID = V_EMPID;
  9       V_DEPT_NAME := GET_DEPTNAME(V_DEPTID);
 10            DBMS_OUTPUT.PUT_LINE('Employee: ' || V_EMPID);
 11            DBMS_OUTPUT.PUT_LINE
 12               ('Department Name: ' || V_DEPT_NAME);
 13  EXCEPTION
 14       WHEN OTHERS THEN
 15             DBMS_OUTPUT.PUT_LINE(V_EMPID || ' not found.');
 16  END;
 17  /
Enter value for emp_id: 200
Employee: 200
Department Name: Sales

PL/SQL procedure successfully completed.

SQL>
```

Figure 14-7 Function call.

In the next example, reusability of a function is explained. When a function is compiled and stored, it can be called many times. You write it once, but you can use it many times. Figure 14-8 has a WHILE loop in the anonymous block that calls the function *do_total* of Figure 14-9 eight times, for values of the counter (or DeptId)

```
SQL> DECLARE
  2       V_SAL  EMPLOYEE.SALARY%TYPE;
  3       V_COMM EMPLOYEE.COMMISSION%TYPE;
  4       V_TOTSAL NUMBER(8) := 0;
  5       V_TOTCOMM NUMBER(8) := 0;
  6       v_TOTAL NUMBER(8);
  7       COUNTER NUMBER(2) := 10;
  8    BEGIN  /* MAIN BLOCK  */
  9       DBMS_OUTPUT.PUT_LINE('DEPTID      SALARY        COMMISSION');
 10       DBMS_OUTPUT.PUT_LINE('----------  ------------  --------------------');
 11       WHILE COUNTER <= 40 LOOP
 12          SELECT SUM(NVL(SALARY, 0)), SUM(NVL(COMMISSION, 0))
 13             INTO V_SAL, V_COMM FROM EMPLOYEE
 14             WHERE DEPTID = COUNTER;
 15          DBMS_OUTPUT.PUT_LINE(COUNTER || '      ' ||
 16             TO_CHAR(V_SAL, '$999,999') || '  '
 17                          || TO_CHAR(V_COMM, '$999,999'));
 18          V_TOTSAL := DO_TOTAL(V_TOTSAL, V_SAL);
 19          V_TOTCOMM := DO_TOTAL(V_TOTCOMM, V_COMM);
 20          COUNTER := COUNTER + 10;
 21       END LOOP;
 22       V_TOTAL := V_TOTSAL + V_TOTCOMM;
 23       DBMS_OUTPUT.PUT_LINE('SUBTOTAL ' ||
 24         TO_CHAR(V_TOTSAL, '$999,999') || '  ' ||
 25               TO_CHAR(V_TOTCOMM, '$999,999'));
 26       DBMS_OUTPUT.PUT_LINE('TOTAL OF SALARY AND COMMISSION: ' ||
 27     TO_CHAR(V_TOTAL, '$999,999'));
 28    END;  /*  MAIN BLOCK  */
 29    /
DEPTID      SALARY        COMMISSION
----------  ------------  --------------------
10          $375,000      $35,000
20          $146,500      $20,000
30           $69,500       $8,000
40          $150,000      $10,000
SUBTOTAL    $741,000      $73,000
TOTAL OF SALARY AND COMMISSION:  $814,000

PL/SQL procedure successfully completed.

SQL>
```

Figure 14-8 Function call from a loop.

```
SQL> CREATE OR REPLACE FUNCTION DO_TOTAL
  2       (I_AMOUNT IN NUMBER, I_TOTAL IN NUMBER)
  3        RETURN NUMBER
  4  IS
  5       V_RESULT   NUMBER(8) := 0;
  6  BEGIN
  7       V_RESULT := I_TOTAL + I_AMOUNT;
  8       RETURN V_RESULT;
  9  END DO_TOTAL;
 10  /
Function created.
SQL>
```

Figure 14-9 This function (see Fig. 14-8) is called multiple times.

equal to 10, 20, 30, and 40. For each value of the counter, the function *do_total* is called twice, once with two salary parameters and once with two commission parameters. Each time, *do_total* adds *v_sal* to *v_totsal* or *v_comm* to *v_totcomm* and returns totals to *v_totsal* and *v_totcomm*, respectively. Then, variable *v_total* is used to find the grand total of all salaries and commissions in the anonymous block. The block then prints the total payroll for the company.

Calling a Function from an SQL Statement

A stored function block can be called from an SQL statement, such as SELECT. For example,

```
SELECT get_deptname(10) FROM dual;
```

This function call with parameter 10 will return the department name Finance in the N2 example tables. You can also use a substitution variable as parameter.

PACKAGES

A package is a collection of PL/SQL objects. The objects in a package are grouped within BEGIN and END blocks. A package may contain objects from the following list:

- Cursors.
- Scalar variables.
- Composite variables.
- Constants.
- Exception names.
- TYPE declarations for records and tables.

- Procedures.
- Functions.

Packages are modular in nature, and Oracle has many built-in packages. If you remember, DBMS_OUTPUT is a built-in package. You also know that a package called STANDARD contains definitions of many operators used in Oracle. There are many benefits to using a package.

The objects in a package can be declared as public objects, which can be referenced from outside, or as private objects, which are known only to the package. You can restrict access to a package to its specification only and hide the actual programming aspect. A package follows some rules of object-oriented programming, and it gives programmers some object-oriented capabilities. A package compiles successfully even without a body if the specification compiles. When an object in the package is referenced for the first time, the entire package is loaded into memory. All package elements are available from that point on, because the entire package stays in memory. This one-time loading improves performance and is very useful when the functions and procedures in it are accessed frequently. The package also follows top-down design.

Structure of a Package

A package provides an extra layer to a module. A module has a header and a body, whereas a package has a specification and a body. A module's header specifies the name and the parameters, which tell us how to call that module. Similarly, the package specification tells us how to call different modules within a package.

Package Specification

A package specification does not contain any code, but it does contain information about the elements of the package. It contains definitions of functions and procedures, declarations of global or public variables, and anything else that can be declared in a PL/SQL block's declaration section. The objects in the specification section of a package are called **public objects**. The general syntax is

```
CREATE [OR REPLACE] PACKAGE packagename
IS
    [ constant, variable and type declarations ]
    [ exception declarations ]
    [ cursor specifications ]
    [ function specifications ]
    [ procedure specifications ]
END [ packagename ];
```

For example,

```
PACKAGE bb_team
IS  total_players CONSTANT INTEGER := 12;
```

```
    player_on_dl EXCEPTION;
  FUNCTION team_average(points IN NUMBER, players IN NUMBER)
                                              RETURN NUMBER;
  END bb_team;
```

The package specification for the course_info package in Figure 14-10 contains the specification of a procedure called FIND_TITLE and functions HAS_PREREQ and FIND_PREREQ. The COURSE_INFO package contains three modules in all.

```
SQL> CREATE OR REPLACE PACKAGE COURSE_INFO
  2  AS
  3  PROCEDURE FIND_TITLE
  4        (I_ID IN COURSE.COURSEID%TYPE,
  5         O_TITLE OUT COURSE.TITLE%TYPE);
  6  FUNCTION HAS_PREREQ
  7        (I_ID IN COURSE.COURSEID%TYPE)
  8         RETURN BOOLEAN;
  9  FUNCTION FIND_PREREQ
 10        (I_ID IN COURSE.COURSEID%TYPE)
 11         RETURN VARCHAR2;
 12  END COURSE_INFO;
 13  /

Package created.

SQL>
```

Figure 14-10 Package specification.

Package Body

A package body contains actual programming code for the modules described in the specification section. It also contains code for the modules not described in the specification section. The module code in the body without a description in the specification is called a **private module**, or a **hidden module**, and it is not visible outside the body of the package.

The general syntax of a package body is

```
PACKAGE BODY packagename
IS
        [ variable and type declarations ]
        [ cursor specifications and SELECT queries ]
        [ header and body of functions ]
        [ header and body of procedures ]
[ BEGIN
        executable statements ]
```

```
[ EXCEPTION
        exception handlers ]
END [ packagename ];
```

As a field is to a record, so an object is to a package. When you reference an object in a package, you must qualify it with the name of that package using dot notation. If you do not use dot notation to reference an object, the compilation will fail. Within the body of a package, you do not have to use dot notation for that package's objects, but you definitely have to use dot notation to reference an object from another package. For example,

```
IF bb_team.total_players < 10 THEN
```

For example,

```
EXCEPTION
    WHEN bb_team.player_on_dl THEN
```

where *total_players* and *player_on_dl* are modules/objects in the *bb_team* package.

There is a set of rules that you must follow in writing a package's body:

- The variables, constants, exceptions, and so on declared in the specification must not be declared again in the package body.
- The number of cursor and module definitions in the specification must match the number of cursor and module headers in the body.
- Any element declared in the specification can be referenced in the body.

In Figures 14-10 and 14-11, package specification and body, respectively, are shown for the *course_info* package. The calls to a procedure and a function in the *course_info* package are shown in Figures 14-12 and 14-13, respectively.

In Figure 14-12, a call is made to the procedure FIND_TITLE of the COURSE_INFO package in Figure 14-11. The procedure is passed V_COURSEID as an IN parameter. If CourseId is invalid, the procedure throws an exception and then handles that exception with an appropriate message. If CourseId is valid, the OUT parameter V_TITLE is assigned course title.

In Figure 14-13, a call is made to the function HAS_PREREQ of the COURSE_INFO package with one parameter, V_COURSEID. If course does not have a prerequisite, the function displays the appropriate message. If course does not exist, an exception is thrown in the function body, and the message is displayed. The function returns FALSE in both cases. The function returns TRUE if the prerequisite exists. If TRUE is returned, another function, FIND_PREREQ, is called with the V_COURSEID parameter. The function returns concatenated prerequisite ID and title. Figure 14-13 shows output from all three situations mentioned above.

You can also use the EXECUTE command to run a package's procedure:

```
EXECUTE packagename.procedurename
```

```
SQL> CREATE OR REPLACE PACKAGE BODY COURSE_INFO AS
  2  PROCEDURE FIND_TITLE
  3       (I_ID IN COURSE.COURSEID%TYPE, O_TITLE OUT COURSE.TITLE%TYPE) IS
  4  BEGIN
  5       SELECT TITLE INTO O_TITLE FROM COURSE WHERE COURSEID = I_ID;
  6  EXCEPTION
  7       WHEN OTHERS THEN
  8          DBMS_OUTPUT.PUT_LINE(I_ID || ' not found.');
  9  END FIND_TITLE;
 10  ------------------------------------------------------------------------------------------
 11  FUNCTION HAS_PREREQ
 12       (I_ID IN COURSE.COURSEID%TYPE) RETURN BOOLEAN IS
 13        V_PREREQ VARCHAR2(6);
 14  BEGIN
 15       SELECT NVL(PREREQ, 'NONE' ) INTO V_PREREQ
 16            FROM COURSE WHERE COURSEID = I_ID;
 17       IF V_PREREQ = 'NONE' THEN
 18             DBMS_OUTPUT.PUT_LINE('No prerequisite');
 19             RETURN FALSE;
 20       ELSE RETURN TRUE;
 21       END IF;
 22  EXCEPTION
 23       WHEN NO_DATA_FOUND THEN
 24       DBMS_OUTPUT.PUT_LINE('Course: ' || I_ID || ' does not exist');
 25       RETURN FALSE;
 26  END HAS_PREREQ;
 27  ------------------------------------------------------------------------------------------
 28  FUNCTION FIND_PREREQ
 29       (I_ID IN COURSE.COURSEID%TYPE) RETURN VARCHAR2 IS
 30        V_ID     VARCHAR2(6);
 31        V_TITLE  VARCHAR2(25);
 32        V_PRE    VARCHAR2(30);
 33  BEGIN
 34       SELECT NVL(P.COURSEID, 'NONE'), NVL(P.TITLE, 'NONE')
 35        INTO V_ID, V_TITLE FROM COURSE C, COURSE P
 36        WHERE C.PREREQ = P.COURSEID AND C.COURSEID = I_ID;
 37             V_PRE := V_ID || '--' || V_TITLE;
 38             RETURN V_PRE;
 39  EXCEPTION
 40       WHEN OTHERS THEN RETURN 'NONE';
 41  END;
 42  END COURSE_INFO;
 43  /

Package body created.
```

Figure 14-11 Package body.

```
SQL> /*  Anonymous block calls
DOC>    procedure FIND_TITLE  in package COURSE_INFO  */
SQL> DECLARE
  2       V_COURSEID COURSE.COURSEID%TYPE := '&P_COURSEID';
  3       V_TITLE COURSE.TITLE%TYPE;
  4  BEGIN
  5       COURSE_INFO.FIND_TITLE(V_COURSEID, V_TITLE);
  6       IF V_TITLE IS NOT NULL THEN
  7            DBMS_OUTPUT.PUT_LINE(V_COURSEID || ': ' || V_TITLE);
  8       END IF;
  9  END;
 10  /
Enter value for p_courseid: CIS265
CIS265: Systems Analysis

PL/SQL procedure successfully completed.

SQL> /
Enter value for p_courseid: CIS100
CIS100 not found.

PL/SQL procedure successfully completed.

SQL>
```

Figure 14-12 Call to procedure in the package of Figure 14-11.

TRIGGERS

A **database trigger**, known simply as a **trigger**, is a PL/SQL block. It is stored in the database and is called automatically when a triggering event occurs. A user cannot call a trigger explicitly. The triggering event is based on a Data Manipulation Language (DML) statement, such as INSERT, UPDATE, or DELETE. A trigger can be created to fire before or after the triggering event. For example, if you design a trigger to execute after you INSERT a new employee in the EMPLOYEE table, the trigger executes after the INSERT statement. The execution of a trigger is also known as **firing the trigger**. The general syntax is

```
CREATE [ OR REPLACE ] TRIGGER triggername
BEFORE | AFTER | INSTEAD OF triggeringevent ON table|view
        [ FOR EACH ROW ]
        [ WHEN condition ]
DECLARE
        Declaration statements
BEGIN
        Executable statements
EXCEPTION
        Exception-handling statements
END;
```

```
SQL> /*  Anonymous block calls function HAS_PREREQ
DOC>    and function FIND_PREREQ in package COURSE_INFO  */
SQL> DECLARE
  2       V_FLAG BOOLEAN;
  3       V_COURSEID COURSE.COURSEID%TYPE := '&P_COURSEID';
  4       V_TITLE VARCHAR2(30);
  5  BEGIN
  6       V_COURSEID := UPPER(V_COURSEID);
  7       V_FLAG := COURSE_INFO.HAS_PREREQ(V_COURSEID);
  8       IF V_FLAG = TRUE THEN
  9        V_TITLE := COURSE_INFO.FIND_PREREQ(V_COURSEID);
 10        DBMS_OUTPUT.PUT_LINE('Course: ' || V_COURSEID);
 11        DBMS_OUTPUT.PUT_LINE('Pre-Requisite - ' || V_COURSEID);
 12       END IF;
 13  END;
 14  /
Enter value for p_courseid: CIS265
Course: CIS265
Pre-Requisite - CI5253

PL/SQL procedure successfully completed.

SQL> /
Enter value for p_courseid: CIS253
No prerequisite

PL/SQL procedure successfully completed.

SQL> /
Enter value for p_courseid: CIS999
Course: CIS999 does not exist

PL/SQL procedure successfully completed.

SQL>
```

Figure 14-13 Call to functions in the package of Figure 14-11.

where CREATE means you are creating a new trigger and REPLACE means you are replacing an existing trigger. The key word REPLACE is optional, and you should only use it to modify a trigger. If you use REPLACE and a procedure, function, or package exists with the same name, the trigger replaces it. If you create a trigger for a table and then decide to modify it and associate it with another table, you will get an error. If a trigger already exists in one table, you cannot replace it and associate it with another table.

A trigger is very useful in generating values for derived columns, keeping track of table access, preventing invalid entries, performing validity checks, or

maintaining security. There are some restrictions, however, involving creation of a trigger:

- A trigger cannot use a Transaction Control Language statement, such as COMMIT, ROLLBACK, or SAVEPOINT. All operations performed by a trigger become part of the transaction. The trigger operations get committed or rolled back with the transaction.
- A procedure or function called by a trigger cannot perform Transaction Control Language statements.
- A variable in a trigger cannot be declared with LONG or LONG RAW data types.

BEFORE Triggers

The BEFORE trigger is fired before execution of a DML statement. The BEFORE trigger is useful when you want to plug into some values in a new row, insert a calculated column into a new row, or validate a value in the INSERT query with a lookup in another table.

Figure 14-14 is an example of a trigger that fires before a new row is inserted into a table. If a new employee is being added to the EMPLOYEE table, you can use a trigger to get the next employee number from the sequence, use SYSDATE as the employee's hire date, and so on. The trigger in Figure 14-14 fires before the INSERT statement. The naming convention used in the example uses the table name the trigger is for, followed by *bi* for "before insert," and then the word *trigger*. A trigger uses a pseudorecord called :NEW, which allows you to access the currently processed row. The type of record :NEW is *tablename*%TYPE. In this example, the type of :NEW is employee%TYPE. The columns in this :NEW record are referenced with dot notation (e.g., :NEW.EmployeeId).

```
SQL> CREATE OR REPLACE TRIGGER EMPLOYEE_BI_TRIGGER
  2  BEFORE INSERT ON EMPLOYEE
  3  FOR EACH ROW
  4  DECLARE
  5       V_EMPID   EMPLOYEE.EMPLOYEEID%TYPE;
  6  BEGIN
  7       SELECT EMPLOYEE_EMPLOYEEID_SEQ.NEXTVAL
  8       INTO V_EMPID FROM DUAL;
  9       :NEW.EMPLOYEEID := V_EMPID;
 10       :NEW.HIREDATE := SYSDATE;
 11  END;
 12  /

Trigger created.

SQL>
```

Figure 14-14 BEFORE trigger.

The trigger *employee_bi_trigger* provides values of EmployeeId and HireDate, so you need not include those values in your INSERT statement. If you have many columns that can be assigned values via a trigger, your INSERT statement will be shortened considerably. In Figure 14-15, a row is inserted in the EMPLOYEE table without values for EmployeeId and HireDate columns. Those columns are given value with firing of the BEFORE trigger of Figure 14-14.

```
SQL> INSERT INTO EMPLOYEE
  2  (LNAME, FNAME, POSITIONID, SUPERVISOR, SALARY, DEPTID, QUALID)
  3  VALUES
  4  ('ZEE', 'SONIA', 3, 543, 100000, 20, 2);

1 row created.

SQL> SET LINESIZE 200
SQL> SELECT * FROM EMPLOYEE WHERE LNAME='ZEE';

EMPLOYEEID LNAME           FNAME           POSITIONID SUPERVISOR
---------- --------------- --------------- ---------- ----------
       546 ZEE             SONIA                    3        543

SQL>
```

Figure 14-15 BEFORE trigger—row inserted.

AFTER Triggers

An AFTER trigger fires after a DML statement is executed. It utilizes the built-in Boolean functions INSERTING, UPDATING, and DELETING. If the triggering event is one of the three DML statements, the function related to the DML statement returns TRUE and the other two return FALSE. For example, if the current DML statement is INSERT, then INSERTING returns TRUE, but DELETING and UPDATING return FALSE.

The example in Figure 14-16 uses an existing table named TRANSHISTORY, which keeps track of transactions performed on a table. It keeps track of updates and deletions, the user who performs them, and the dates on which they are performed. The trigger is named *employee_adu_trigger,* where *adu* stands for "after delete or update." The trigger uses the transaction type based on the last DML statement. It also plugs in the user name and today's date. The information is then inserted in the TRANSHISTORY table. Figure 14-17 shows rows inserted in the TRANSHISTORY table on use of DELETE and UPDATE statements by trigger.

For the example in Figure 14-14, we used FOR EACH ROW. Such a trigger is known as a **row trigger.** If it is based on INSERT, the trigger fires once for every newly inserted row. If it is based on UPDATE statement and the UPDATE affects five rows, the trigger is fired five times, once for each affected row. In Figure 14-16, we did not use a FOR EACH ROW, clause. Such a trigger is known as a **statement trigger.** A statement trigger is fired only once for the statement, irrespective of the number of rows affected by the DML statement.

```
SQL> CREATE OR REPLACE TRIGGER EMPLOYEE_ADU_TRIGGER
  2  AFTER DELETE OR UPDATE ON EMPLOYEE
  3  DECLARE
  4       V_TRANSTYPE VARCHAR2(6);
  5  BEGIN
  6       IF DELETING THEN
  7            V_TRANSTYPE := 'DELETE';
  8       ELSIF UPDATING THEN
  9            V_TRANSTYPE := 'UPDATE';
 10       END IF;
 11       INSERT INTO TRANSHISTORY
 12       VALUES ('EMPLOYEE', V_TRANSTYPE, USER, SYSDATE);
 13  END;
 14  /

Trigger created.

SQL>
```

Figure 14-16 AFTER trigger.

```
SQL> DELETE FROM EMPLOYEE
  2  WHERE UPPER(LNAME) = 'VIQUEZ';

1 row deleted.

SQL> SELECT * FROM TRANSHISTORY;

TABLENAME    TRANSTYPE    USER_NAME    TRAN_DATE
-----------  -----------  -----------  -----------
EMPLOYEE     DELETE       NSHAH        05-DEC-03

SQL> UPDATE EMPLOYEE
  2  SET COMMISSION = SALARY * 0.10
  3  WHERE EMPLOYEEID = 547;

1 rows updated.

SQL> SELECT * FROM TRANSHISTORY;

TABLENAME    TRANSTYPE    USER_NAME    TRAN_DATE
-----------  -----------  -----------  -----------
EMPLOYEE     DELETE       NSHAH        05-DEC-03
EMPLOYEE     UPDATE       NSHAH        05-DEC-03

SQL>
```

Figure 14-17 AFTER trigger—in action.

In Figure 14-17, you see the workings of the AFTER TRIGGER of Figure 14-16. A row is deleted from the EMPLOYEE table, and the trigger inserts a row in the TRANSHISTORY table. Then, a row is updated in the EMPLOYEE table, and the trigger inserts another row in the TRANSHISTORY table.

INSTEAD OF Trigger

The BEFORE and AFTER triggers are based on database tables. From version 8i onward, Oracle provides another type of trigger called an INSTEAD OF trigger, which is not based on a table but is based on a view (covered in Chapter 9). The INSTEAD OF trigger is a row trigger. If a view is based on a SELECT query that contains set operators, group functions, GROUP BY and HAVING clauses, DISTINCT function, join, and/or a ROWNUM pseudocolumn, data manipulation is not possible through it.

An INSTEAD OF trigger is used to modify a table that cannot be modified through a view. This trigger fires "instead of" triggering DML statements, such as DELETE, UPDATE, or INSERT.

In Figure 14-18, a complex view is created with a SELECT query and an outer join. FacultyId 235, 333, and 444 are not used in the STUDENT table; in other words, there is no "child" row in STUDENT table for those faculty members. FacultyId 235 and 333 are not used in the CRSSECTION table either. The DELETE statement to delete Faculty Id 235 in Figure 14-18 returned an error message. We will accomplish deletion of row by creating an INSTEAD OF trigger.

```
SQL> CREATE OR REPLACE VIEW STUDENT_FACULTY
  2  AS
  3  SELECT S.STUDENTID, S.LAST, S.FIRST, F.FACULTYID, F.NAME
  4  FROM STUDENT S, FACULTY F
  5  WHERE S.FACULTYID(+) = F.FACULTYID;

View created.

SQL> DELETE FROM student_faculty WHERE FacultyId = 235;
DELETE FROM student_faculty WHERE FacultyId = 235
            *
ERROR at line 1:
ORA-01752: cannot delete from view without exactly one key-preserved table

SQL>
```

Figure 14-18 No data manipulation through complex view.

In Figure 14-19, an INSTEAD OF DELETE trigger is created on the STUDENT_FACULTY view. Now, when the DELETE statement is issued to delete a faculty member with the complex view, the trigger is fired, and the faculty

```
SQL> CREATE OR REPLACE TRIGGER faculty_delete_iod
  2  INSTEAD OF DELETE ON student_faculty
  3  FOR EACH ROW
  4  BEGIN
  5        DELETE FROM faculty
  6        WHERE FacultyId = :OLD.FacultyId;
  7  END;
  8  /

Trigger created.

SQL> DELETE FROM student_faculty WHERE FacultyId = 235;

1 row deleted.

SQL>
```

Figure 14-19 Data manipulation and the INSTEAD OF Trigger.

member is deleted without any error message. Notice the use of pseudorow called :OLD in this trigger, which gets the value of FacultyId 235 from the DELETE statement that the user had issued.

DATA DICTIONARY VIEWS

Oracle maintains a very informative Data Dictionary. A few Data Dictionary views are useful for getting information about stored PL/SQL blocks. The following are examples of queries to USER_PROCEDURES (for named blocks), USER_TRIGGERS (for triggers only), USER_SOURCE (for all source codes), USER_OBJECTS(for any object), and USER_ERRORS (for current errors) views:

```
SELECT Object_Name, Procedure_Name FROM USER_PROCEDURES;
SELECT Trigger_Name, Trigger_Type, Triggering_Event, Table_Name, Trigger_Body
      FROM USER_TRIGGERS;
SELECT Name, Type, Line, Text FROM USER_SOURCE;
SELECT Object_Name, Object_Type FROM USER_OBJECTS;
SELECT Name, Type, Sequence, Line, Position FROM USER_ERRORS;
```

These views can provide information ranging from the name of an object to the entire source code. Use the DESCRIBE command to find out the names of columns in each Data Dictionary view, and issue SELECT queries according to the information desired.

IN A NUTSHELL . . .

- A procedure is a named PL/SQL module that can perform one or more tasks.
- A procedure call contains the name of the procedure along with the list of parameters enclosed within parentheses.
- A procedure body, like an anonymous block, consists of declaration, executable, and exception sections.
- There are three types of parameters: The IN type passes a value into a subprogram, the OUT type passes a value to the calling program, and the IN OUT type passes a value into a subprogram and returns a value.
- The formal parameters in a module's header must match the actual parameters in the call to the module, with positional or named notation.
- A function is a PL/SQL named module that always returns a value to the calling program.
- A function can return a scalar data type, such as VARCHAR2, NUMBER, BINARY_INTEGER, and BOOLEAN, or a complex or composite data type, such as a table, a record, a nested table, VARRAY, or LOB.
- A function call is made via an executable PL/SQL statement, such as an assignment or an IF statement.
- A function or a procedure is stored in memory by executing it first. It is then called by another module.
- A package is a collection of PL/SQL objects, which are either public (can be called by an outside module) or private (known only to the package module).
- The structure of a package includes a specification and a body. The members in a package are referenced with the package name as a qualifier and dot notation.
- A database trigger, or simply a trigger, is stored in the database and is called implicitly when a triggering event occurs.
- A trigger is based on a DML statement, such as INSERT, DELETE, or UPDATE.
- A BEFORE trigger is fired before execution of a DML statement, and an AFTER trigger is fired after execution of a DML statement.
- A row trigger is fired once for each affected row, whereas a statement trigger is fired only once, irrespective of the number of rows affected by the DML statement.
- An INSTEAD OF trigger is based on a view instead of a database table.
- Oracle provides the user with many Data Dictionary views for named blocks, such as USER_PROCEDURES, USER_TRIGGERS, USER_SOURCE, USER_OBJECTS, and USER_ERRORS.

EXERCISE QUESTIONS

True/False:

1. A parameter of type IN passes a read-only value to a module.
2. A parameter of type OUT is assigned a value only if the called module is performed successfully.
3. If a procedure has an IN parameter, it must have an OUT parameter.
4. A function always has an OUT parameter to return a value.
5. V_EMPNAME IN VARCHAR2(25) is a valid formal parameter definition in the header of a module.
6. A procedure does not require a RETURN type, but a function does.
7. Control of execution shifts from a function to the calling program with the RETURN statement.
8. All public procedures and functions in a package body must be declared in the package specification.
9. A trigger is fired either before or after a triggering event.
10. A trigger based on a SELECT statement always fires automatically after the statement's execution.

Answer the Following Questions:

1. Name three types of PL/SQL modules. Describe each module with one characteristic specific to the module.
2. What are the differences between a procedure and a function?
3. Name three types of parameters. List the characteristics of each type.
4. How are actual parameters and formal parameters associated? Explain with an example.
5. What are the benefits of using a package? Describe two parts of a package and their contents.
6. What is a trigger? Explain the working of BEFORE and AFTER triggers.
7. How are INSTEAD OF triggers different from BEFORE and AFTER triggers?
8. What Data Dictionary tables/views are available to get information about named blocks? Give information stored by those views/tables.

LAB ACTIVITY

1. Write a procedure that is passed a student's identification number and returns the student's full name and phone number from the STUDENT table to the calling program. Also, write an anonymous block with the procedure call.
2. Write a function, and pass a department number to it. If the DEPT table does not contain that department number, return a FALSE value; otherwise, return a TRUE value. Print the appropriate message in the calling program based on the result.
3. Write a package that contains a procedure and a function. The procedure is passed a room number. If the room number exists, the procedure gets the capacity of the room

and the building name from the LOCATION table. If the room number does not exist, the procedure performs the appropriate exception-handling routine. The function is passed a *csid* and returns the maximum number of seats available in the course section.

4. Write a trigger that is fired before the DML statement's execution on the EMPLOYEE table. The trigger checks the day based on SYSDATE. If the day is Sunday, the trigger does not allow the DML statement's execution and raises an exception. Write the appropriate message in the exception-handling section.
5. Write a trigger that is fired after an INSERT statement is executed for the STUDENT table. The trigger writes the new student's ID, users name, and system date in a table called TRACKING. (*Note:* You must create the TRACKING table first.)
6. Create a complex view EMP_DEP_VIEW using an outer join between the EMPLOYEE and DEPENDENT tables with employee names and dependent birthdates and relations. The outer join will also return employees without any dependents. Now, create an INSTEAD OF trigger based on EMP_DEP_VIEW to enable you to delete employee 433 through view.

PART 4
MISCELLANEOUS TOPICS

15

Oracle with Java: A Tutorial on JDBC and SQLj

IN THIS CHAPTER . . .

- A background overview of Java is given.
- Connection to the Oracle database through Java is covered.
- Steps to connect to Oracle with Sun's JDBC driver are shown.
- Set up of OracleDriver for Java and use of Oracle's thin client are outlined.
- Oracle's SQLj is introduced.
- Use of SQLj iterators is shown.
- PL/SQL blocks are accessed through SQLj.

The purpose of this chapter is not to teach the Java language but to illustrate Java's ability to connect to the Oracle database with drivers provided by Sun and Oracle. Java is a language for Web-based applications, and it is an integral part of the Oracle9i environment. With Oracle8i, the SQL statement CREATE JAVA was added for creating a Java source, class, or resource. Familiarity with Java is required for understanding terminology and code samples given here.

JAVA: A PROGRAMMING LANGUAGE

The Java language was developed by James Gosling for Sun Microsystems. The language was intended for the consumer electronics market but later became a general-purpose business language and a language for Web-based Internet programs. Java is a portable programming language with a broad set of predefined classes and methods that handle most of the fundamental requirements of a programmer. Some of the features of Java include platform independence, object orientation, multithreading, security, and the ability to connect to various database servers. With Java, you can create stand-alone applications, applets, servlets, Java Server Pages (JSPs), and Enterprise Java Beans. Database connectivity is a very important ingredient for server-side programming. If you know C/C++ language, the syntax of Java is very similar—but the similarity ends there! I have found my experience with Java to be very positive and interesting. Whether you use the command-line Java Development Kit (JDK) or a GUI-based Integrated Development Environment (IDE), working with Java is fun. In this chapter, Java code is created with JBuilder8, an IDE from Borland Corporation. Oracle Corporation markets a similar product called JDeveloper, which was originally licensed from Borland Corporation.

JDBC

JDBC is an Application Programming Interface (API) for database access in Java. Using the JDBC 3.0 API, you can access virtually any data source, from relational databases to spreadsheets to flat files. JDBC technology also provides a common base on which tools and alternate interfaces can be built.

The JDBC 3.0 API is comprised of two packages:

1. The java.sql package.
2. The javax.sql package.

Java contains a rich library of classes with which to send SQL statements to the database server, such as Oracle Server, for data retrieval or manipulation. Sun Microsystems provides JDBC drivers with Java. Many database vendors (Oracle, Microsoft, Sybase, and others) and some third-party vendors provide JDBC drivers, which implement JDBC API for various database engines. The Java applications with JDBC are portable and are independent of the database server.

The JDBC–ODBC Bridge allows applications written in the Java programming language to use the JDBC API with many existing ODBC drivers. The Bridge is itself a driver based on JDBC technology (“JDBC driver”) that is defined in the class sun.jdbc.odbc.JdbcOdbcDriver. The Bridge defines the JDBC subprotocol of ODBC. In this chapter, two examples of JDBC applications are given, one with

Sun's sun.jdbc.odbc.JdbcOdbcDriver and another with Oracle's oracle.jdbc.driver. OracleDriver. There are five steps in creating a JDBC application:

1. Import JDBC classes or packages with JDBC classes.
2. Load JDBC drivers.
3. Establish connection with the database.
4. Execute SQL statements/interact with the database.
5. Close connection.

Importing Package or JDBC classes

In Java, all classes from the *java.lang* package are readily available. All other packages and their classes must be imported to make them available to the program. For example, to import the *JOptionPane* class from the *javax.swing* package, you would issue the following statement:

```
import javax.swing.JOptionPane;
```

To import the entire *java.sql* package and all its classes, you would type

```
import java.sql.*;
```

Loading JDBC Drivers

A Java program may load many JDBC drivers to connect with different database servers. The syntax for loading a JDBC driver is

```
Class.forName("drivername");
```

For example,

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

or

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Connecting to the Oracle Database

The *DriverManager* class manages JDBC drivers. After a JDBC driver is loaded, the *getConnection( )* classwide method of the DriverManager class is used for establishing a connection to the database. The general syntax is

```
Connection conn = DriverManager.getConnection(url, user, password)
```

For example, using Sun's driver,

```
Connection conn =
DriverManager.getConnection("jdbc:odbc:shah_ora", userName, passWord);
```

where *"jdbc:odbc:shah_ora"* is the url suited for Sun's jdbc.odbc.JdbcOdbcDriver. The url contains data source name *shah_ora*. Data source is created with Windows operating system's Administrative Tool Data Sources (ODBC) in the control panel. ODBC (Open DataBase Connectivity) is a programming interface that enables applications to access data in database-management systems that use SQL as a data access standard. Username and password can be passed as string objects or hard-coded literals.

Another example, using Oracle's driver, would be

```
Connection conn = DriverManager.getConnection
("jdbc:oracle:thin:@nshah-monroe:1521:oracle","nshah","india_usa");
```

where url is suited for Oracle's oracle.jdbc.driver.OracleDriver and it contains driver type *thin*, server name *nshah-monroe*, default port number *1521*, and database sid *oracle*. The username and password are hard coded, however, which you might not want to do. The hard-coded fictitious password 'india_usa' is used in this chapter's examples, but it is just an example. The server name *nshah-monroe* in this example is the name of the local machine, where Oracle9i resides. If you want to connect to the Oracle server from your PC, you will need the server name as well as the domain name. For example, if the server name is *oracleadmin* at domain *indo-us.edu*, you will use *oracleadmin.indo-us.edu.*

When getConnection( ) method is called with a url, DriverManager locates a suitable driver. If it does not find a suitable driver, it throws an exception. If it does find a suitable driver, the Connection object (referenced by *conn*) is returned, and a connection is established. The interaction with the database is possible through this connection object. One Java application may have multiple connections with the same database, and it may have connections with different databases.

Interacting with the Oracle Database

The Connection object *conn* is used to interact with the database using SQL statements. There are three classes for sending SQL statements to the server:

1. Statement class for SQL statements without parameters.
2. PreparedStatement class for SQL statements with different parameters.
3. CallableStatement class for executing stored procedures.

Statement Class. This class is used for SQL statements without parameters., such as SELECT, INSERT, DELETE, UPDATE, or CREATE TABLE. First,

a statement object is created with the Connection class' instance method *create-Statement( )*. For example,

```
Statement stmt = conn.createStatement( );
```

Then, executeQuery( ) method is used with Statement object *stmt* for a SELECT query, and executeUpdate( ) method is used for an INSERT, DELETE, UPDATE or CREATE TABLE statement. For example,

```
int n = stmt.executeUpdate(query);
```

You can also use conn.commit( ), conn.setAutoCommit (true|false), and conn.rollback( ) methods after using Data Manipulation Language (DML) statements with stmt.executeUpdate( ). These methods have same effect that you have seen with COMMIT, SET AUTOCOMMIT ON|OFF, and ROLLBACK statements in SQL.

ResultSet Class. When *stmt.executeQuery(SELECT-query)* is executed, the data are retrieved into a ResultSet object. For example,

```
ResultSet rset = stmt.executeQuery(select_query);
```

where ResultSet class object *rset* contains data retrieved by the SELECT-query. The rows in the data are retrieved in sequence, and the columns in each row are positioned. The pointer is positioned before the first row of data. With every call to the next( ) method, the pointer moves to the next row. The ResultSet class contains a few useful methods:

getString(*n*) Numeric position *n* is used to get data from a column.

wasNull( ) Checks last column for a null value, and returns true or false accordingly.

findColumn(column) Returns the position of a column.

ResultSetMetaData Class. The ResultSetMetaData class is used to get metadata information about ResultSet object *rset*. The ResultSetMetaData object *rsmeta* is created with the following statement:

```
ResultSetMetaData rsmeta = rset.getMetaData( );
```

This class provides the user with many methods to get metadata information. For example,

getColumnCount( ) Returns the number of columns retrieved into result set.

getColumnLabel(*n*) Returns the column title for dispay based on column position parameter *n*.

getColumnName(*n*) Returns the column name based on column position parameter *n*.

getColumnType(*n*) Returns the column's data type based on position parameter *n*.

getColumnDisplaySize(*n*) Returns column's size based on position parameter *n*.

getTableName(*n*) Returns name of table based on column position parameter *n*.

getPrecision(*n*) Returns the number of digits to the left of decimal point.

getScale(*n*) Returns the number of digits to the right of decimal point.

PreparedStatement Class. The PreparedStatement class is a derived class (or subclass) of the Statement class. It allows execution of the same SQL statement many times with different parameters. An object of the PreparedStatement class is created with the prepareStatement(*sql_statement*) method, where *sql_statement* may contain parameters. The statement is already compiled for faster execution. The parameters can be set with method like set*XYZ*(*n, val*), where *n* is the position of parameter, *val* is a variable or literal, and *XYZ* is a data type like String, Int, Float, and so on. For example,

```
PreparedStatement pstmt =
conn.prepareStatement("INSERT INTO dept VALUES(?,  ?)");
pstmt.setInt(1, 99);      // first parameter set to 99
pstmt.setString(2, "Athletics");      // second parameter set to 'Athletics'
pstmt.executeUpdate( );
```

After pstmt is built with set*XYZ*( ) methods, the executeUpdate( ) method is used to execute the SQL statement.

CallableStatement Class. The CallableStatement class is a subclass of the PreparedStatement class. It is used to execute PL/SQL anonymous or named blocks. Its object is created with prepareCall( ) method. The IN parameters are set with set*XYZ*( ) method that was used earlier with the superclass PreparedStatement. The OUT parameters are set with registerOutParameter(*n*, *sTypeCode*), where *n* is the position and *sTypeCode* is the SQL type code defined in java.sql.Types class. Once a callable statement is built with all parameter values or type, it is executed. For example,

```
CallableStatement cstmt =
conn.prepareCall("{? = call do_total(?, ?)}"); // function call
cstmt.registerOutParameter(1, Types.NUMBER);   // OUT parameter
cstmt.setInt(2, v_sal);   // IN parameter
cstmt.setInt(3, v_comm);   // IN parameter
cstmt.execute( );
```

Closing Connection

The JDBC session ends with closing the database connection. Before disconnecting from the database, the stmt.close( ) method is used to close the ResultSet generated by that statement. At last, the conn.close( ) method is used to close the connection and release the JDBC resources.

SUN'S JDBC DRIVER AND THE ORACLE DATA SOURCE

Creating a Data Source in the Windows Control Panel

To make the program work with *sun.jdbc.odbc.JdbcOdbcDriver*, you must have the Oracle driver in Oracle's home directory, such as OraHome92. The typical installation of Oracle installs this driver.

First, from My Computer, run the Windows Control Panel. In Control Panel, double-click on the Administrative Tools icon, and then on Data Sources (ODBC) to bring up the ODBC Data Source Administrator (see Fig. 15-1). Visual FoxPro

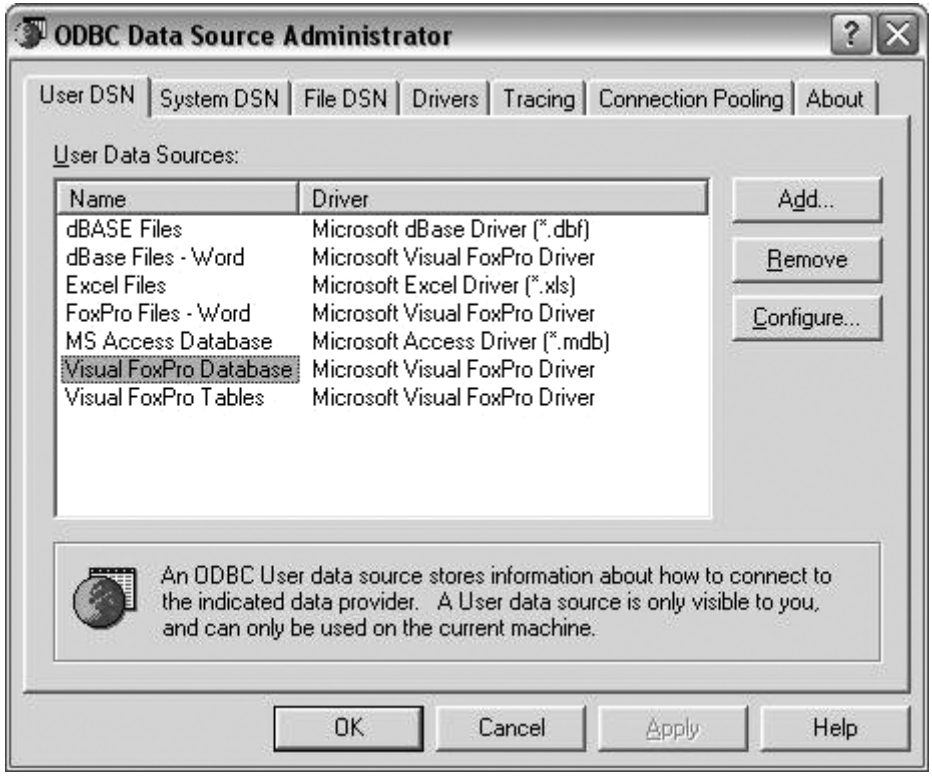


Figure 15-1 ODBC Data Source Administrator.

database is highlighted in the figure by default, but you are going to use a different driver, which is not listed in Figure 15-1.

Second, click on the Add button to bring up the Create New Data Source dialog box (see Fig. 15-2).

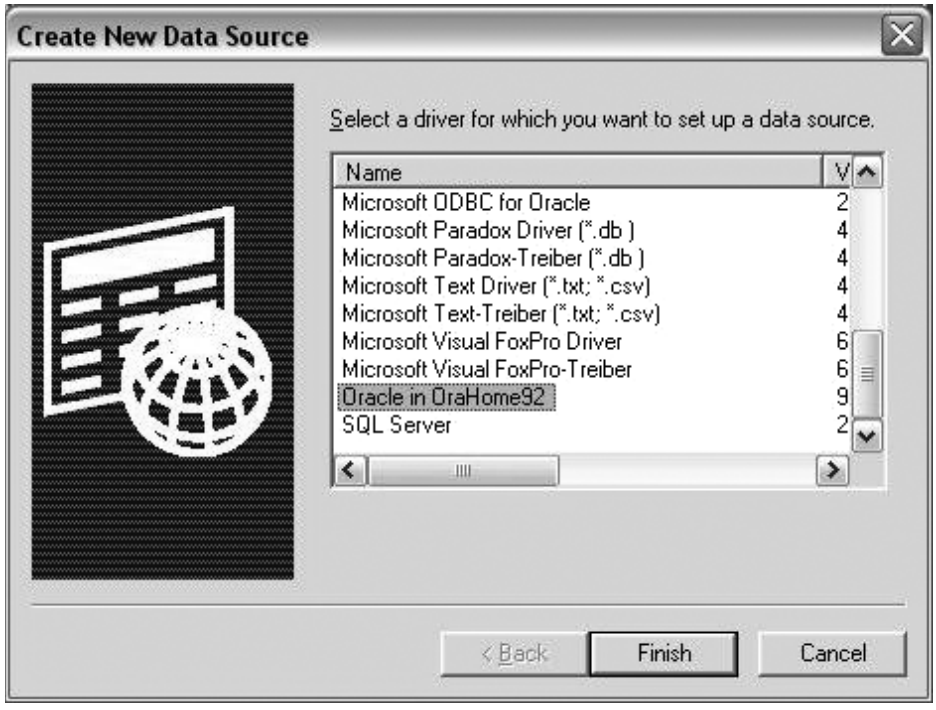


Figure 15-2 Create New Data Source.

Third, select the Oracle in OraHome92 driver, and click on the Finish button to bring up the Oracle ODBC Driver Configuration dialog window. Type the values for Data Source Name, Description, TNS Service Name, and User ID as shown in Figure 15-3. Click on the Test Connection button to log in with a password, and test the connection to the Oracle database. If the connection is successful, click on OK.

Your data source is created as shown in Figure 15-4. This data source name is used later in the Java program for connectivity.

Sample Java Code

Figure 15-5 shows code for database connectivity to the Oracle database with Sun Microsystem's JDBC driver. The program loads then driver, then prompts the user for a usename and password. The connection is established with url, username, and password. The url includes the data source created in the previous section. The rows are retrieved from the PHONE table with the executeQuery( ) method. The rows

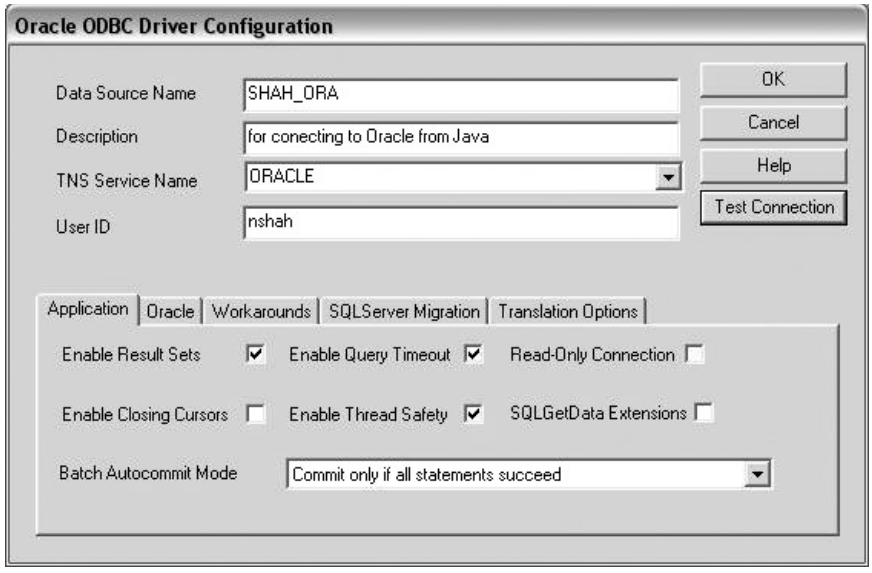


Figure 15-3 Oracle ODBC Driver Configuration.

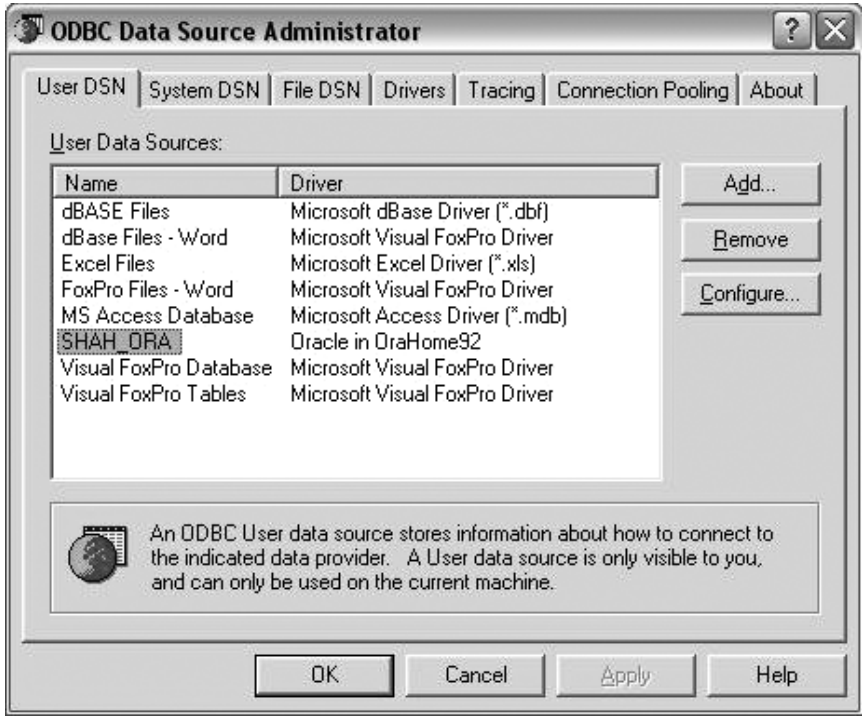


Figure 15-4 ODBC Data Source Administrator.

```
package oracle;
import  java.sql.*;
import  java.io.*;
import  javax.swing.*;
public  class Connect {
    public static void main(String[] args)
            throws SQLException, IOException{
        try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        catch (ClassNotFoundException e) {
          System.out.println("Could not load driver");
        }
        String userName, passWord;
         userName = JOptionPane.showInputDialog("Oracle username: ").trim();
         passWord = JOptionPane.showInputDialog("Oracle password: ").trim();
         Connection conn = DriverManager.getConnection
                    ("jdbc:odbc:shah_ora", userName, passWord);
         Statement stmt = conn.createStatement();
         ResultSet rset = stmt.executeQuery
              ("SELECT LAST, FIRST, PHONE, RELATION FROM PHONE");
         while (rset.next()) {
            System.out.println(rset.getString(1)+"\t"+rset.getString(2)+
                "\t"+rset.getString(3)+"\t"+rset.getString(4));
         }
          stmt.close();
         conn.close();
    }
}
```

Figure 15-5 Java source with data source.

are read from the result set with the next( ) method and subsequently displayed within the while loop.

Figure 15-6 shows rows from the PHONE table with an SQL query. Figure 15-7 shows rows from the same table with an embedded SQL statement in the Java program of Figure 15-5.

```
SQL> SELECT * FROM phone;

LAST            FIRST          PHONE          RELATION
--------------- -------------- -------------- ------------
SHAH            NAMAN          609.555.1111   SON
JOHNSON         KIM            732.555.2222   FRIEND
SHAH            DHIRAJ         222.386.3260   FATHER
MATTHEWS        MICKEY         407.555.3333   FRIEND

SQL>
```

Figure 15-6 Oracle table output.

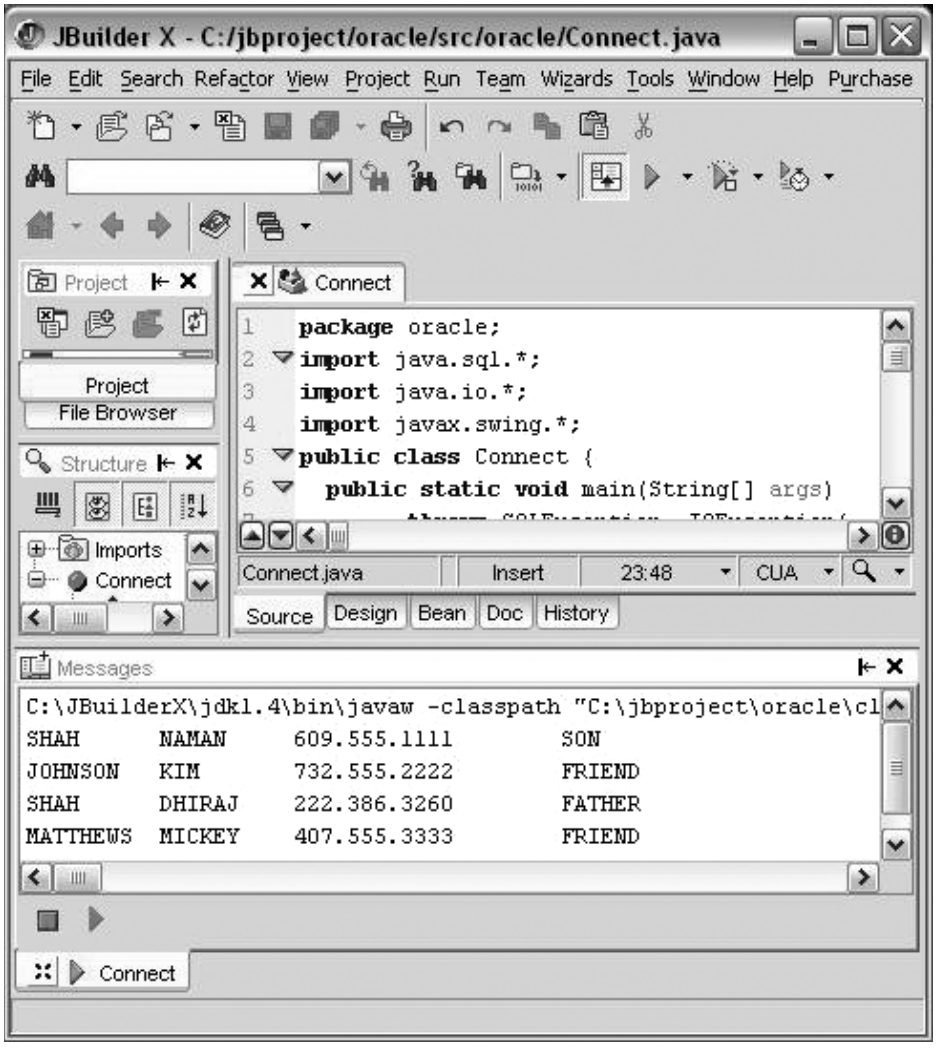


Figure 15-7 Table output from Java.

ORACLEDRIVER AND ORACLE *thin* DRIVER

Setting Up oracle.jdbc.driver.OracleDriver for SDK1.4 or JBuilder8

The oracle.jdbc.driver.OracleDriver is set up as follows:

1. After successful installation of Oracle8i or 9i, find the home directory (folder) for ORACLE_HOME. It will be C:\ORACLE\ORA81 for Oracle8i and

C:\ORACLE\ORA90 or C:\ORACLE\ORA92 for Oracle9i. For example, it is **C:\oracle\ora92** on my computer.

2. Search for the *CLASSES12.ZIP* file under your Oracle home directory in the JDBC\LIB directory. For example, the path on my computer is **c:\oracle\ora92\jdbc\lib\classes12.zip**. You may use the Windows file name search for file path.
3. Now, go to the LIB directory under Java's home directory on your computer. Java's home directory will be C:\J2SDK1.4.1_01 for Sun's command-line Java Development Kit or C:\JBUILDER8\JDK1.4 for Borland's JBuilder IDE version 8. For example, the path to LIB folder on my computer is **c:\jbuilder8\jdk1.4\lib**.
4. Copy the CLASSES12.ZIP file from the Oracle home to the Java home directory. For example, for JBuilder8,

```
C:\> copy c:\oracle\ora92\jdbc\lib\classes12.zip c:\jbuilder8\jdk1.4\lib
```

   For command-line Java,

```
C:\> copy c:\oracle\ora92\jdbc\lib\classes12.zip c:\j2sdk1.4.1_01\lib
```

5. If you are using command-line Java, set the following classpath to use OracleDriver from the CLASSES12.ZIP file:

```
Set classpath = c:\j2sdk1.4.1_01\lib\classes12.zip;
```

   If you are using JBuilder, follow the directions given here:
   - In JBuilder8 IDE, create your project, application, or applet.
   - Before executing your program, right-click on your project name (*.jpx* file) in the Project Pane on the top left.
   - Select **Add files/packages . . .** from the pop-up menu.
   - Select **OracleDriver** from the following folder path:

```
C:\jbuilder8\jdk1.4\lib\classes12.zip\oracle\jdbc\driver\
```

6. Now, follow the steps given below to add a new library (if you did not copy the CLASSES12.ZIP file in the default library directory):
   - Go to the **Projects** menu, and select **Project Properties**.
   - Click on the **Required Library** tab.
   - Click on the **Add** button.
   - Click on the **New** button.
   - Change the name text-box value to **shahLib** (or any other name).
   - Click on the **Add** button.
   - Click on the **OK** button.
   - Select the path where the CLASSES12.ZIP file is saved.

- Click on the **OK** button (you will see a new library path).
- Click on the **OK** button.
- Click on the **OK** button (you will see shahLib in the required library listing).

The versions of the products mentioned in this section—Oracle9i, JBuilder8 and JDK1.4—can be downloaded for free from Oracle.com, Borland.com, and Sun.com, respectively, for personal use or educational purposes only. Software companies introduce new software releases frequently. When you follow the instructions given here, substitute folder or directory names acoording to the software version installed on your system. (*Note*: You must add the OracleDriver and library to each project in JBuilder as given in Steps 5 and 6. You need not create the library again.)

Sample Java Code

Figure 15-8 shows Java code (similar to that in Figure 15-5) for database connectivity with the Oracle database with Oracle's JDBC driver. The program loads the driver, then establishes a connection with the url and hard-coded username and password.

```
package oracle;
import java.sql.*;
import java.io.*;
public class Connect {
   public static void main(String[] args)
         throws SQLException, IOException{
      try {
       Class.forName("oracle.jdbc.driver.OracleDriver");
      }
      catch (ClassNotFoundException e) {
         System.out.println("Could not load driver");
      }
      String user, pass;
      Connection conn = DriverManager.getConnection
          ("jdbc:oracle:thin:@nshah-monroe:1521:oracle","nshah","india_usa");
      Statement stmt = conn.createStatement();
      ResultSet rset = stmt.executeQuery
          ("SELECT LAST, FIRST, PHONE, RELATION FROM PHONE");
      while (rset.next()) {
         System.out.println(rset.getString(1)+"\t"+rset.getString(2)+
            "\t"+rset.getString(3)+"\t"+rset.getString(4));
      }
      stmt.close();
      conn.close();
   }
}
```

Figure 15-8 Java source with OracleDriver.

The url contains the Oracle *thin* driver. The rows are retrieved from the PHONE table with the executeQuery( ) method. The rows are read from the result set and are subsequently displayed.

JAVA APPLET: PUTTING IT ALL TOGETHER*

In the following example, a Java applet is created with some of the classes described in previous sections. The applet connects to a database to perform INSERT, SELECT, and a search based on last name.

In Figure 15-9, LastNameSearch is used with a last name entered in a JTextField object. The result is displayed in the jTextArea object. The interface uses four JTextField objects for inputting Last, First, Phone, and Relation values. A JTextField object is used for entering the username, and a JPasswordField object is used for entering

Figure 15-9 Java Applet in AppletViewer.

**Important:* A good knowledge of Java language and swing components is essential to understand this code.

the password for connecting to Oracle9i. These two objects are given value through the *text* property, and they are disabled to prevent any changes to these values. Every connection to the database uses these values for username and password. There are four JButton objects. The AddRow button gets inputs from four text boxes and adds a new row to the PHONE table. If any one of the four boxes is left blank, an error pops up, and the insert operation is not performed. The LastNameSearch button gets the last name input and searches through the table for matching rows. It displays matching rows in the JTextArea object. If the last name field is left blank or no matching rows are found, an appropriate message is displayed in the JTextArea object. The ClearTextOnly button clears the four JTextField objects. Finally, the DisplayPhoneList button is used to display the entire PHONE table. The username and password can be hard-coded in the url, or the user can be asked to input values for the same. The input portion is present in the source given below, but it is commented out:

```
package  dataapplet;
import  java.awt.*;
import  java.awt.event.*;
import  java.applet.*;
import  javax.swing.*;
import  java.sql.*;
import  java.io.*;
public class AppletData extends Applet {
   private boolean isStandalone = false;
   JTextField jTextField1 = new JTextField();
   JTextField jTextField2 = new JTextField();
   JTextField jTextField3 = new JTextField();
   JTextField jTextField4 = new JTextField();
   JButton jButton1 = new JButton();
   JLabel jLabel1 = new JLabel();
   JLabel jLabel2 = new JLabel();
   JLabel jLabel3 = new JLabel();
   JLabel jLabel4 = new JLabel();
   JButton jButton2 = new JButton();
   JButton jButton3 = new JButton();
   JLabel jLabel5 = new JLabel();
   JTextArea jTextArea1 = new JTextArea();
   JButton jButton4 = new JButton();
   JButton jButton5 = new JButton();
   JPasswordField jPasswordField1 = new JPasswordField();
   JTextField jTextField5 = new JTextField();
   JLabel jLabel6 = new JLabel();
   JLabel jLabel7 = new JLabel();
    //Get a parameter value
   public String getParameter(String key, String def) {
      return isStandalone ? System.getProperty(key, def) :
        (getParameter(key) != null ? getParameter(key) : def);
   }
```

```
//Construct the applet
public AppletData() {
}
//Initialize the applet
public void init() {
    try  {
      jbInit();
    }
    catch(Exception e) {
       e.printStackTrace();
    }
}
//Component initialization
private void jbInit() throws Exception {
  jTextField1.setBackground(Color.white);
  jTextField1.setText("");
  jTextField1.setBounds(new Rectangle(27, 42, 85, 20));
  this.setLayout(null);
  jTextField2.setBounds(new Rectangle(135, 41, 85, 20));
  jTextField2.setText("");
  jTextField3.setBounds(new Rectangle(28, 91, 85, 20));
  jTextField3.setText("");
  jTextField4.setBounds(new Rectangle(135, 90, 85, 20));
  jTextField4.setText("");
  jButton1.setBackground(SystemColor.activeCaption);
  jButton1.setBounds(new Rectangle(238, 39, 136, 21));
  jButton1.setFont(new java.awt.Font("Dialog", 1, 11));
  jButton1.setText("AddRow");
  jButton1.addActionListener(
            new AppletData_jButton1_actionAdapter(this));
  jLabel1.setFont(new java.awt.Font("Dialog", 1, 11));
  jLabel1.setText("Last");
  jLabel1.setBounds(new Rectangle(54, 23, 32, 17));
  jLabel2.setBounds(new Rectangle(162, 22, 33, 17));
  jLabel2.setFont(new java.awt.Font("Dialog", 1, 11));
  jLabel2.setText("First");
  jLabel3.setBounds(new Rectangle(48, 71, 41, 17));
  jLabel3.setFont(new java.awt.Font("Dialog", 1, 11));
  jLabel3.setText("Phone");
  jLabel4.setBounds(new Rectangle(157, 71, 56, 17));
  jLabel4.setFont(new java.awt.Font("Dialog", 1, 11));
  jLabel4.setText("Relation");
  jButton2.setBackground(SystemColor.activeCaption);
  jButton2.setBounds(new Rectangle(238, 89, 136, 21));
  jButton2.setFont(new java.awt.Font("Dialog", 1, 11));
  jButton2.setText("ClearTextOnly");
  jButton2.addActionListener(
            new AppletData_jButton2_actionAdapter(this));
```

```
jButton3.setBackground(SystemColor.activeCaption);
jButton3.setBounds(new Rectangle(28, 262, 132, 29));
jButton3.setFont(new java.awt.Font("Dialog", 1, 11));
jButton3.setText("DisplayPhoneList");
jButton3.addActionListener(
        new AppletData_jButton3_actionAdapter(this));
jLabel5.setFont(new java.awt.Font("Dialog", 1, 24));
jLabel5.setText("Phone List");
jLabel5.setBounds(new Rectangle(244, 8, 135, 27));
this.setBackground(Color.orange);
jTextArea1.setRequestFocusEnabled(false);
jTextArea1.setColumns(50);
jTextArea1.setTabSize(8);
jTextArea1.setBounds(new Rectangle(28, 117, 347, 134));
jButton5.addActionListener(
        new AppletData_jButton5_actionAdapter(this));
jButton5.setText("LastNameSearch");
jButton5.addActionListener(
        new AppletData_jButton5_actionAdapter(this));
jButton5.setFont(new java.awt.Font("Dialog", 1, 11));
jButton5.setBounds(new Rectangle(238, 64, 136, 21));
jButton5.setBackground(SystemColor.activeCaption);
jPasswordField1.setEnabled(false);
jPasswordField1.setText("india_usa");
jPasswordField1.setBounds(new Rectangle(291, 272, 70, 23));
jTextField5.setEnabled(false);
jTextField5.setText("nshah");
jTextField5.setBounds(new Rectangle(207, 273, 70, 22));
jLabel6.setFont(new java.awt.Font("Dialog", 1, 14));
jLabel6.setText("UserName");
jLabel6.setBounds(new Rectangle(206, 254, 77, 18));
jLabel7.setFont(new java.awt.Font("Dialog", 1, 14));
jLabel7.setText("PassWord");
jLabel7.setBounds(new Rectangle(291, 255, 74, 15));
this.add(jTextField1, null);
this.add(jTextField2, null);
this.add(jTextField3, null);
this.add(jTextField4, null);
this.add(jButton1, null);
this.add(jTextArea1, null);
this.add(jButton2, null);
this.add(jButton4, null);
this.add(jButton5, null);
this.add(jLabel5, null);
this.add(jLabel1, null);
this.add(jLabel3, null);
this.add(jLabel2, null);
this.add(jLabel4, null);
```

```
    this.add(jLabel6, null);
    this.add(jTextField5, null);
    this.add(jLabel7, null);
    this.add(jPasswordField1, null);
    this.add(jButton3, null);
}
//Get Applet information
public String getAppletInfo() {
    return "Applet Information";
}
//Get parameter info
public String[][] getParameterInfo() {
    return null;
}
/********************* ADDS a ROW **********************/
void jButton1_actionPerformed(ActionEvent e) throws Exception {
// user can be prompted to input username and password,
// but commented out here
/*     String userName, passWord;
    userName =
      JOptionPane.showInputDialog("Oracle username: ").trim();
    passWord =
      JOptionPane.showInputDialog("Oracle password: ").trim(); */
    try {
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch (ClassNotFoundException ex) {
    System.out.println("Could not load driver");
}
Connection conn = DriverManager.getConnection
                ("jdbc:odbc:shah_ora",jTextField5.getText(),
                            jPasswordField1.getText());
    PreparedStatement pstmt =
     conn.prepareStatement(
     "INSERT INTO phone VALUES(?,?,?,?)"); //statement with 4 parameters
    String one = jTextField1.getText();
    String two = jTextField2.getText();
    String three = jTextField3.getText();
    String four = jTextField4.getText();
    if (one.equals("")||two.equals("")
      ||three.equals("")|| four.equals(""))
{
      JOptionPane.showMessageDialog
            (null, "One or more fields are left blank","Error", JOptionPane.ERROR_MESSAGE);
}
else {                                          // 4 parameters set here
    pstmt.setString(1, one);
    pstmt.setString(2, two);
```

```
    pstmt.setString(3, three);
    pstmt.setString(4, four);
    pstmt.executeUpdate();                                    // INSERT executed
}
conn.commit();  // commits insert operation
pstmt.close();
conn.close();
}
/***************** CLEARS ALL TEXT FIELDS *****************/
void jButton2_actionPerformed(ActionEvent e) {
    jTextField1.setText("");
    jTextField2.setText("");
    jTextField3.setText("");
    jTextField4.setText("");
}
/******************** DISPLAYS ALL ROWS ******************/
void jButton3_actionPerformed(ActionEvent e) throws SQLException{
    jTextArea1.setText("");
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }
    catch (ClassNotFoundException ex) {
      System.out.println("Could not load driver");
    }
    Connection conn =
          DriverManager.getConnection("jdbc:odbc:shah_ora",
          jTextField5.getText(), jPassordField1.getText());
    Statement stmt = conn.createStatement();
    ResultSet rset = stmt.executeQuery
    ("SELECT Last, First, Phone, relation
        FROM phone ORDER BY Last, first ");          // all rows retrieved
    String s="";
    while (rset.next()) {
       s+=(rset.getString(1)+"\t\t"+rset.getString(2)+
      "\t"+rset.getString(3)+"\t"+rset.getString(4)+"\n");
    }
    jTextArea1.setText(s);                 // rows displayed in JTextArea object
    stmt.close();
    conn.close();
}
/************** SEARCHES TABLE BASED ON LAST NAME **************/
void jButton5_actionPerformed(ActionEvent e) throws SQLException{
    jTextArea1.setText("");
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }
    catch (ClassNotFoundException ex) {
       System.out.println("Could not load driver");
    }
```

```
Connection conn =
     DriverManager.getConnection("jdbc:odbc:shah_ora", jTextField5.getText(),
      jPasswordField1.getText());
      Statement stmt = conn.createStatement();
      String l = jTextField1.getText().trim().toUpperCase();
      String query =
         "SELECT Last, first, Phone, Relation " +
         "FROM phone WHERE Upper(Last)= '"+ l + "'";
      ResultSet rset = stmt.executeQuery(query);
      String s="";
      while (rset.next()) {
          s+=(rset.getString(1)+"\t\t"+rset.getString(2)+
          "\t"+rset.getString(3)+"\t"+rset.getString(4)+"\n");
      }
      if (s.equals(""))
          jTextArea1.setText
          ("Last name field is left blank\nOR\ndoes not exist in table");
      else
          jTextArea1.setText(s);
          stmt.close();
          conn.close();
      }
}
class AppletData_jButton1_actionAdapter implements java.awt.event.ActionListener
         {
  AppletData adaptee;
  AppletData_jButton1_actionAdapter(AppletData adaptee) throws Exception{
     this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
     try{
         adaptee.jButton1_actionPerformed(e);
     }
     catch (Exception ex){
          System.out.println("Error");
     }
 }
}
class AppletData_jButton2_actionAdapter implements java.awt.event.ActionListener
         {
   AppletData_jButton2_actionAdapter(AppletData adaptee) {
       this.adaptee = adaptee;
    }
    public void actionPerformed(ActionEvent e) {
        adaptee.jButton2_actionPerformed(e);
    }
}
```

```
class AppletData_jButton3_actionAdapter implements java.awt.event.ActionListener
        {
  AppletData adaptee;

  AppletData_jButton3_actionAdapter(AppletData adaptee) throws Exception {
     this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
     try{
      adaptee.jButton3_actionPerformed(e);
    }
     catch (Exception ex){
          System.out.println("Error");
     }
  }
}

class AppletData_jButton5_actionAdapter implements java.awt.event.ActionListener
        {
  AppletData adaptee;
  AppletData_jButton5_actionAdapter(AppletData adaptee) throws Exception{
     this.adaptee = adaptee;
  }
  public void actionPerformed(ActionEvent e) {
     try{
       adaptee.jButton5_actionPerformed(e);
     }
     catch (Exception ex){
     System.out.println("Error");
  }
}
}
```

SQLj

An alternative to JDBC, SQLj is Oracle's implementation of the ANSI SQL-1999 Part-0 "Embedded SQL in Java," which specifies the integration of SQL statements in Java programs. Oracle SQLj is more concise and easier to write than JDBC, and it provides compile-time schema validation and syntax checking for easier debugging.

With SQLj, SQL statements can be embedded directly in a Java program. You need to configure your Java environment by adding JARs; *runtime12.jar, translator.jar*, and *classes12.jar*. If you remember, classes12.jar (from the CLASSES12.ZIP file) was used during the set up of JDBC. You also need the SQLj executable, *sqlj.exe*. SQLj is an attractive alternative, because it can perform syntax checking of SQL statements at translation time. The SQLj translator translates a Java program (*.sqlj*) with SQL statements into Java code (*.java*), which can be executed through the JDBC driver. SQLj code is more compact than its JDBC counterpart.

Configuring Oracle SQLj in JBuilder8

To configure Oracle SQLj in JBuilder 8, do the following:

1. In JBuilder8, choose Tools | configure Libraries |New . . . | Type "SQLj" for 'Name:' field | keep entry "User Home" for 'Location' field |Add . . . | select two files, **runtime12.jar** and **translator.jar**, from the Oracle directory C:\ORACLE\ORA92\SQLJ\LIB.
2. In JBuilder8, choose Tools | configure Libraries |New . . . | Type "Oracle Driver" for 'Name:' field | keep entry "User Home" for 'Location' field |Add . . . | select **classes12.jar** from the Oracle directory C:\ORACLE \ORA92\JDBC\LIB.
3. In JBuilder8, choose Tools | Enterprise Setup | SQLj | Oracle | click on the . . . (ellipses) button for SQLj executable | select **sqlj.exe** file from C:\ORACLE\ORA92\BIN\.
4. In Jbuilder8, choose Tools | Enterprise Setup | SQLj | Oracle | Add SQLj library from Step 1 to the 'Library' list box, and click on OK.

Creating an SQLj Project

To create an SQLj project, do the following:

1. In JBuilder8, File | New Project | name your project "sqljone" | Finish.
2. Right click on the *sqljone.jpx* file in project pane, and select 'Add files/Packages'. Type "TestSQLj.sqlj" for file name, and select file type as SQLJ (*.sqlj*) file. You will see a dialog box with message "The selected file does not exist. Do you wish to create it?" Click on OK to create it.
3. In the project pane, double-click on the *TestSQLj.sqlj* file to open it up in the editor. Type the code from Figure 15-10 in the editor. You can substitute names for username, password, and so on.
4. File | Save All.
5. Select Project | Project Properties | Paths | Required Libraries | Add "SQLj" library.
6. Select Project | Project Properties | Paths | Required Libraries | Add "Oracle Driver" library.
7. Select Project | Project Properties | Build | General | SQLj Traslator | select "Oracle" from the combo box.
8. In the project pane, right-click on *TestSQLj.sqlj*, and select Make. The process will produce a Java (*.java*) file and invoke the Java compiler to produce a bytecode (*.class*) file.
9. Right click on the *TestSQLj.java* file, and select Run Using Defaults. You will see the output.

```
import oracle.sqlj.runtime.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.io.*;
import java.sql.*;
public class TestSQLj2 {
    public static void main(String [] args) {
        try {
          Oracle.connect
          ("jdbc:Oracle:thin:@localhost:1521:oracle",
                                        "nshah", "india_usa", true);
          String dname = "";
          int num = 10;
          #sql {SELECT DeptName into :dname
                FROM dept WHERE DeptId = :num};
          System.out.println("Deptname is " + dname);
        }
        catch (SQLException e) {
          e.printStackTrace();
        }
        finally {
          try { Oracle.close(); }
          catch(SQLException e) { e.printStackTrace(); }
        }
    }
}
```

Figure 15-10 SQLj program.

The program in Fig. 15-10 illustrates statements involved in writing an SQLj program:

- **Import classes.** The *java.sql* package is imported for JDBC classes, *sqlj.runtime* and *sqlj.runtime.ref* packages for SQLj runtime classes, and the *oracle.sqlj.runtime* package for Oracle class.
- **Connect to the database.** Connection to the Oracle database is achieved with the Oracle.connect( ) method. The method takes four parameters; url, username, password, and autocommit mode. The autocommit mode could be true or false. If false is used, the programmer has to commit SQL statements. If you are using any database other than Oracle, you would use DriverManager.registerDriver( ), but this is not needed for Oracle database.
- **Embed SQL statements.** The SQL statements are embedded with #*sql* {*sql_statement*}, where #*sql* tells the SQLj translator that the SQL statement follows. The statements may include host variables with a colon (:) prefix (as seen in PL/SQL section), which are declared in the Java program.

For example, string variable *dname* is declared in Java and used as *:dname* in an SQL statement as a host variable, and Java variable *num* is used as *:num* as shown Figure 15-10.

HOST VARIABLES

A host variable is used with a colon (:) prefix followed by IN, OUT or INOUT depending on if it is for input, output, or both, respectively. Remember them from PL/SQL? The default is IN for host variables, except for the INTO clause of SELECT statement. We can rewrite the SQL statement in Figure 15-10 as

```
#sql {SELECT deptName INTO :OUT dname FROM phone WHERE DeptId = :IN num};
```

where *dname* and *num* are declared in the Java program.

You need to choose data types carefully while using Java variables in SQL. Oracle's CHAR and VARCHAR2 types correspond to string in Java. Oracle's DATE type corresponds to java.sql.Date class. Oracle's NUMBER type corresponds to Java's int, long, float, and double. A CURSOR in Oracle is ResultSet in Java. Any valid Java expression can be used as a host expression in an SQL statement.

SQLj ITERATORS

When an SQL query returns more than one row in a Java program, an SQLj iterator is used. An iterator is similar to a JDBC result set, but columns are given data types in the former. An iterator is based on cursor in the SQL query. SQLj constructs an *iterator* class for iterator declaration. The instance variables and methods from the iterator class are available in the program. There are two types iterator declarations:

1. Named Iterator: Data types and column names are specified.
2. Positional Iterator: Only data types are specified.

Named Iterator

The general syntax for declaring a named iterator is

```
#sql iterator iteratorName (type1 column1, typ2 column2, . . . , typeN columnN);
```

In Figure 15-11, a named iterator is used. The following statement declares a named iterator *DeptIter:*

```
#sql iterator DeptIter(int i, String n, String l);
```

Iterator contains three columns with their data type. SQLj creates a class with the same name, DeptIter. Then, a null reference named *di* to DeptIter class is declared,

```
import oracle.sqlj.runtime.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.io.*;
import java.sql.*;
public class TestSQLj3 {
    public static void main(String [ ] args) {
        #sql iterator DeptIter(int i, String n, String l);
        try  {
          Oracle.connect
           ("jdbc:Oracle:thin:@localhost:1521:oracle",
                          "nshah", "india_usa", true);
         DeptIter di = null;
         #sql di =
              {SELECT deptid i, deptname n, location l FROM dept};
         while (di.next( )) {
               System.out.println(di.i( )+"\t"+di.n( )+"\t\t"+di.l( ));
          }
         di.close( );
        }
        catch (SQLException e) {
           System.out.println(e.getMessage( ));
        }
        finally {
           try { Oracle.close( ); }
           catch(SQLException e) { e.printStackTrace( ); }
        }
    }
}
```

Figure 15-11 SQLj named iterator.

which is populated with a SELECT query. It is important to use column aliases for columns retrieved, and the column aliases must be same as the column names in the iterator:

```
DeptIter di = null;
#sql di={SELECT deptid i,deptname n,location l FROM dept};
```

The next( ) method of the DeptIter class is used to retrieve one row from the iterator. SQLj also creates get or accessor methods with the same name as the column names to get the values of columns, such as i( ), n( ) and l( ):

```
while (di.next( )) {
      System.out.println(di.i()+"\t"+di.n()+"\t\t"+di.l());
}
```

Finally, the iterator is closed with the close( ) method after it has been processed:

```
di.close( );
```

Positional Iterator

A positional iterator is similar to a named iterator, but it is declared with data types that positionally match the data types of columns retrieved with the SELECT query. The general syntax of declaration is

```
#sql iterator IteratorName (type1, type2, . . . , typeN);
```

For example, the positional iterator is declared only with data types in Figure 15-12 as follows:

```
#sql iterator DeptIter(int, String, String);
```

```
import oracle.sqlj.runtime.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.io.*;
import java.sql.*;
public class TestSQLj4 {
    public static void main(String [] args) {
        #sql iterator DeptIter(int, String, String);
        int i=0;
        String n=null;
        String loc=null;
        try  {
          Oracle.connect
          ("jdbc:Oracle:thin:@localhost:1521:oracle",
                    "nshah", "india_usa", true);
          DeptIter di = null;
          #sql di = {SELECT deptid, deptname, location FROM dept};
          #sql {fetch :di into :i, :n, :loc};
          while (!di.endFetch()) {
               System.out.println(i+"\t"+n+"\t\t"+loc);
               #sql {fetch :di into :i, :n, :loc};
          }
          di.close();
    }
    catch (SQLException e) {
       System.out.println(e.getMessage());
    }
    finally {
       try { Oracle.close(); }
       catch(SQLException e) { e.printStackTrace(); }
    }
    }
}
```

Figure 15-12 SQLj positional iterator.

The iterator is instantiated and populated with an SQL query as given in the following statements:

```
DeptIter di = null;
#sql di = {SELECT deptid, deptname, location FROM dept};
```

There is no need to use column aliases with an SQL query, because a fetch statement is used to retrieve columns into host variables declared as Java variables with appropriate data types:

```
int i=0;
String n=null;
String loc=null;
#sql {fetch :di into :i, :n, :loc};
```

The endFetch( ) method checks for row fetched, and while there is no end of fetch, more rows are fetched in the loop with each iteration:

```
while (!di.endFetch()) {
        System.out.println(i+"\t"+n+"\t\t"+loc);
        #sql {fetch :di into :i, :n, :loc};
}
```

Finally, the iterator is closed when the processing is done:

```
di.close( );
```

SQLj is basically used for static SQL statements, but it is possible to use dynamic SQL with SQLj through JDBC. An SQLj program may contain SQLj as well as JDBC code. JDBC's result set and SQLj's iterators can be assigned to each other.

PL/SQL FROM SQLj

SQLj can embed SQL statements with #sql. SQLj can also embed an entire anonymous block with #sql. SQLj can call PL/SQL stored procedures and functions as well.

The general syntax of anonymous block in SQLj is

```
#sql { anonymous block statements };
```

The general syntax of PL/SQL procedure call is

```
#sql { CALL procedurename [ (parameterlist) ] };
```

The general syntax of PL/SQL function call is

```
variablename =  #sql { VALUES (functionname (parameterlist)) };
```

In Figure 15-13, two calls are made to PL/SQL named blocks, one to procedure FIND_TITLE in package COURSE_INFO and one to function FIND_PREREQ in the same package. If a PL/SQL block already exists, there is no need to reinvent the wheel in Java! The named blocks are passed IN, OUT, and INOUT parameters in the same way that PL/SQL uses them. The procedure is passed course ID as an IN parameter and title as an OUT parameter. The function is passed course ID as an IN parameter, and it returns the prerequisite of the course.

```
import oracle.sqlj.runtime.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
import java.io.*;
import java.sql.*;
import javax.swing.*;
public class TestSQLj4 {
    public static void main(String [] args) {
        try {
          Oracle.connect
          ("jdbc:Oracle:thin:@localhost:1521:oracle",
                        "nshah", "india_usa", true);
          String cid=JOptionPane.showInputDialog("Enter Course Id:");
           String title;
           String preReq;
           #sql {CALL course_info.find_title(:in cid, :out title)};
           #sql preReq = {VALUES(course_info.find_prereq(:in cid))};
           System.out.println(cid+"-"+title+" —> PreReq: "+preReq);
        }
        catch (SQLException e) {
           System.out.println(e.getMessage());
        }
        finally {
          try { Oracle.close(); }
          catch(SQLException e) { e.printStackTrace(); }
        }
    }
}
```

Figure 15-13 SQLj calls to PL/SQL named blocks.

IN A NUTSHELL . . .

- Java is a portable programming language with a broad set of predefined classes and methods that handle most fundamental requirements of programmer.
- JDBC is an API (Application Programming Interface) for database access in Java.

- There are five steps in creating a JDBC application: Import JDBC classes or package with JDBC classes, load the JDBC drivers, establish the connection with the database, execute SQL statements/interact with the database, and close the connection.
- JDBC uses a variety of classes for interaction with the Oracle database: Statement, ResultSet, ResultSetMetaData, PreparedStatement, and CallableStatement.
- To make program work with *sun.jdbc.odbc.JdbcOdbcDriver*, you must have Oracle driver in OraHome92 (Oracle Home).
- An Oracle-supplied JDBC driver is used with Oracle thin driver for connecting to the Oracle database with Java.
- SQLj is an alternative to JDBC. With SQLj, SQL statements can be embedded directly in a Java program.
- SQLj translator translates a Java program (*.sqlj*) with SQL statements into Java code (*.java*), which can be executed through the JDBC driver.
- An SQLj program includes importing of classes, connecting to the database, and embedding of SQL statements.
- In an SQLj program, a host variable is used with a colon (:) prefix followed by IN, OUT, or INOUT depending on if it is for input, output, or both, respectively.
- When an SQL query returns more than one row in Java program, an SQLj iterator is used. An interator is similar to a JDBC result set, but columns are given data types in an iterator.
- There are two types iterator declarations: named iterator, in which data types and column names are specified; and positional iterator, in which only data types are specified.
- SQLj can embed SQL statements with #sql. SQLj can also embed an entire anonymous block with #sql. SQLj can call PL/SQL stored procedures and functions as well.

EXERCISE QUESTIONS

True/False:

1. Java is a language for Web-based Internet programming.
2. To connect to the Oracle database with Java, you can only use the JDBC driver provided by Sun Microsystems.
3. Java establishes a connection with the Oracle driver, then loads the JDBC driver.
4. When a SELECT-query is executed with executeQuery( ) method, data are retrieved into a ResultSet object.
5. ResultSetMetaData class is used to get metadata information about a ResultSet object.
6. PreparedStatement class allows execution of SQL statements with parameters.

7. CallableStatement class allows a call to PL/SQL blocks.
8. SQL statements are embedded directly into the Java program with SQLj.
9. Host variables are used in an SQLj program as IN parameters only.
10. Two types of SQLj iterators are named and anonymous iterators.

Answer the Following Questions:

1. Compare the features of JDBC and SQLj.
2. State and briefly explain the steps in creating a JDBC application.
3. What is the difference in use of the JDBC drivers provided by Sun and Oracle?
4. Which Oracle-provided files are necessary for creating an SQLj project with any Java environment?
5. Describe the use of iterators in SQLj projects.

LAB ACTIVITY

1. Create a JDBC project with an Oracle data source to retrieve and display rows from the COURSE table.
2. Create a JDBC project to add a new employee in the EMPLOYEE table. Pass parameters to your SQL statement.
3. Create a JDBC project to call the procedure created in lab activity 1 of Chapter 14.
4. Create an SQLj project to retrieve student name and faculty name from the STUDENT and FACULTY tables, respectively. Use host variables declared in a Java program, and display the retrieved rows (iterator problem).
5. Create an SQLj project to call the function created in lab activity 2 of Chapter 14.

16

Oracle9i: Architecture and Administration

IN THIS CHAPTER . . .

- The functions of a Database Administrator (DBA) are listed.
- Different aspects of Oracle's architecture are explained.
- You will learn about Oracle database administration with Oracle Enterprise Manager (OEM).
- Oracle security, users, roles, and system privileges are covered.
- Oracle Data Dictionary views and their types are listed.

Oracle is a very complex product, and its capabilities are increasing with every new release of the software. The Database Administrator (DBA) is the most critical position in the daily operations of the database environment. The successful implementation of a database depends on the DBA.

DATABASE ADMINISTRATOR (DBA)

The DBA is responsible for installing the Oracle database, managing day-to-day needs of the complex database, and running the system at peak performance. A DBA performs software maintenance, resource management, data administration,

database tuning, troubleshooting, data security, and backup and recovery. Some of the duties performed by the DBA are:

- Install and upgrade Oracle and its tools.
- Configure the Oracle instance.
- Create a database.
- Create, alter, and remove database users and roles.
- Grant and restrict access rights.
- Allocate and manage physical and logical storage structures.
- Develop security strategies.
- Develop backup and recovery procedures.
- Monitor system performance.
- Analyze database performance, and implement solutions to problems.
- Communicate with Oracle support service personnel.
- Troubleshoot locking problems.

In short, the DBA is the most trusted user in the database environment. The DBA must possess a thorough knowledge of the operating system on which Oracle is installed, hardware specifications needed for the server and the clients, memory structures and Oracle processes, PL/SQL modules and their behavior in the system, client–server architecture, and networking-related issues.

ORACLE ARCHITECTURE: AN OVERVIEW

Figure 16-1 presents an overview of Oracle architecture. Oracle architecture contains three major areas:

1. System Global Area (SGA) or memory structure.
2. Background processes.
3. Physical storage structure.

The **System Global Area** (SGA) is a memory area used, for example, to store information that is shared by database and user processes. The SGA consists of a database buffer cache, a shared pool, a Java pool, and redo log buffers. In Oracle8i, the Java pool was used only if Java was used, but it is required in Oracle9i. The database buffer cache contains actual data from the database. The user transactions are first stored in the buffer and then written to the disk. The shared pool contains the executed SQL and PL/SQL statements for reuse of the information. The shared pool keeps parsed statements based on a "least recently used" algorithm. The redo log buffers store the redo entries for the online redo logs before writing them to the disk.

The **background processes** run simultaneously and independently of each other. These processes work on databases, and there can be a number of such processes

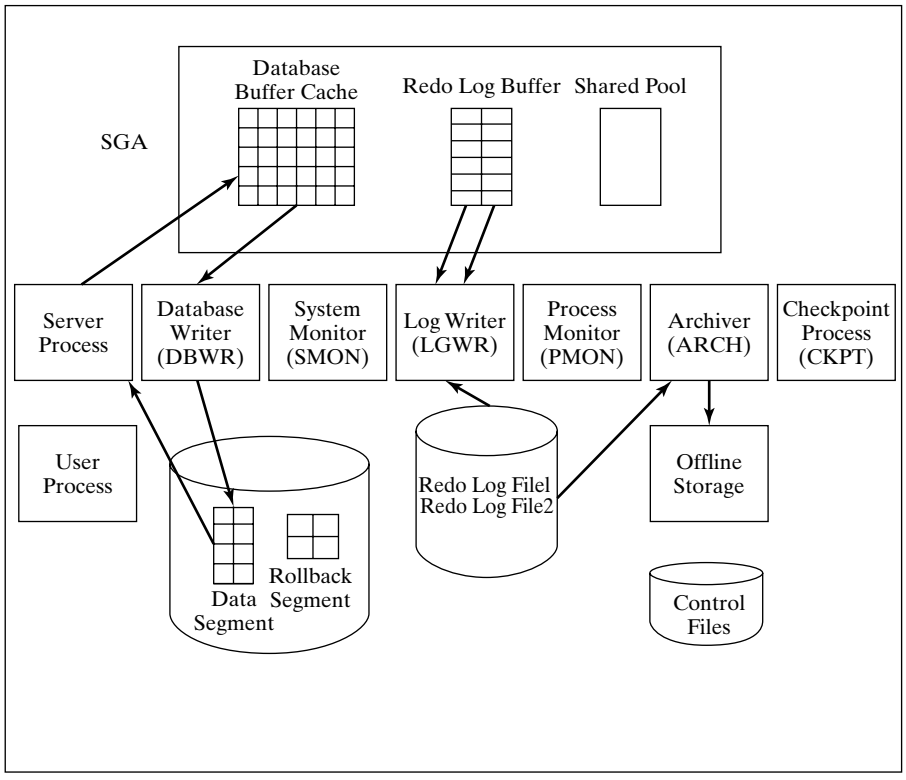


Figure 16-1 Oracle Architecture.

depending on the configuration of the Oracle initialization file (INIT.ORA). Some of the background processes are dedicated server processes, database writer (DBWR), system monitor (SMON), process monitor (PMON), log writer (LGWR), archiver (ARCH), and checkpoint process (CKPT).

SMON process performs all housekeeping tasks. A system process is created for each user process to handle its requests. It performs instance recovery, rolls forward or back as needed for recovery, clears temporary segments that are not needed, coelesces free spaces into a large area of free space, and "listens" to user processes. The user writes to the database buffer cache, and DBWR takes data from the buffer cache in SGA and writes to the database. DBWR is the only process that can write to the database. All updates are performed in the buffer cache, and not on the database files. PMON monitors all processes, checks for failed processes, and terminates them properly. It also releases resources used by failed processes. LGWR writes the SGA redo log buffer's contents to the redo log files. Each instance contains its own LGWR. The redo log buffer holds data images before and after changes. Redo log files contain data as well as rollback information. LGWR writes the redo log buffer's contents to redo log files when the COMMIT command is issued,

the buffer is one-third full, or a 3-second timeout occurs. ARCH archives redo log files to offline or online storage files. CKPT maintains checkpoints.

In Figure 16-2, Data Dictionary view V$BGPROCESS lists process address, process name, and description. Another column named ERROR can also be displayed to show error information. This statement was the first statement executed on a startup of a new user session, and it returned 69 background processes. The processes are listed in order of their startup, and their order is

PMON → DBWR → ARCH → LGWR → SMON

```
SQL> SELECT PADDR, NAME, DESCRIPTION
  2    FROM V$BGPROCESS;

PADDR    NAME   DESCRIPTION
-------- ------ ------------------------------------------------
681E46BC PMON   process cleanup
00       DIAG   diagnosibility process
00       FMON   File Mapping Monitor Process
00       LMON   global enqueue service monitor
00       LMD0   global enqueue service daemon 0
00       LMS0   global cache service process 0
00       LMS1   global cache service process 1
00       LMS2   global cache service process 2
00       LMS3   global cache service process 3
00       LMS4   global cache service process 4
00       LMS5   global cache service process 5
...
00       DMON   DG Broker Monitor Process
00       RSM0   DR Resource Manager Process
00       NSV0   DR Server NetSlave Process 0

69 rows selected.

SQL>
```

Figure 16-2 Oracle background processes.

The Oracle **database** consists of data files, at least two online redo log files, and at least two control files. Data files contain data physically stored on the disk under an operating system's directory structure. The Oracle **instance** is the System Global Area (SGA) memory and the background (shadow) processes. Each SGA is exclusively used by an instance. When an instance starts, memory is allocated for an SGA, and memory is deallocated when the instance shuts down. The Oracle Enterprise Manager (OEM) starts the instance. The database is mounted on the instance and is then opened. The users connect to the instance to access the database. The database is mounted on a single instance in most cases, except for the Oracle Parallel Server (OPS) environment, where a database can be mounted on many instances.

An Oracle instance is defined as ORACLE_SID or ORA_SID. In most cases, database and instance have one-to-one relationship, but in Oracle9i, the database can be mounted by one or more instances. The data are available only when the database is open.

You also must select the database name and its instance name. The initialization file has a name with INIT as a prefix to the instance name with extension ORA. For example, if the instance name is ORCL, then the initialization file is INITORCL.ORA. The database name is specified in the INIT.ORA file in the DB_NAME parameter. The INIT.ORA file is in the DBS directory of the ORACLE_HOME.

The instance contains four types of files:

1. The **parameter file** INIT.ORA is read when an instance is opened.
2. The **control files** are read when the database is mounted.
3. The **data files** are read when the database is opened.
4. The changes to a database are logged in the **redo log files.**

Oracle uses many different logical database structures. A **tablespace** is the basic storage allocation to a database. During Oracle's installation, many tablespaces with minimal capacity are created. Every database has a SYSTEM tablespace, and it also has other tablespaces, such as a TEMPORARY tablespace. The tablespaces are operating system files with the .ORA extension. The DBA creates tablespaces for databases according to the need, and a user-supplied name may contain other extensions, such as .DAT or .DBF.

A user account is called a **schema.** Each object is stored under its owner's schema, which is referenced with a username. An Oracle database is created with two schemas: SYS to store the Data Dictionary, and SYSTEM to store more Data Dictionary information and tables for Oracle tools.

Each object is stored as one or more **segments**. A segment resides in only one tablespace. A tablespace may contain many segments, however, and a segment may contain many **extents**. An extent may contain many **blocks**. The block size is defined in the DB_BLOCK_SIZE parameter. Whenever a user updates a table, the old value is written to the **rollback segment** for read consistency. This also allows a user to roll back updates without committing them. Oracle uses **temporary segments** during table creation and joins.

Other database structures are **tables** to store user data and the Data Dictionary and the **index** for fast search operation from the table.

The Oracle Relational DataBase Management System (RDBMS) is sold as a base product in the Enterprise version. It also comes with other options, data cartridges, development tools, and Enterprise applications. The available licensing options are concurrent user license, named user license, and site license.

The **version** number normally has four numbers (e.g., 9.2.0.1). The first number is the major release number. The second number is the minor release number. The third number is the code release number, and the fourth number is the patch number for the code number.

Performance can be measured in terms of time taken in the execution of a complex query, number of users online concurrently, or time taken by a batch job. The performance depends on the amount of memory, disk space, CPU, bus speed, and network speed. All database packages are input/output (I/O) bound. The speed of I/O affects the performance of such a system. The available resources must be configured properly for optimum performance.

The database must be readily available to the users at all times. Safety measures are considered at the planning phase, configuration phase, and implementation phase for the availability issue. The three configurations are replication, hot standby database, and parallel server. The **replication** method uses separate databases by duplicating the entire implementation of the database on multiple computer systems, where all updates are performed on all database implementations. The operations can still continue if one of the databases crashes. The **hot standby database** method uses only one database at a time. The other standby copy is in recover mode at all times. The redo log files are used to recover the standby copy. If the primary copy fails, the standby copy is recovered completely and is then brought up as the primary database. In an Oracle **parallel server** configuration, multiple computer systems are used with parallel processing capability to share a common database. In the event of a computer system failure, the operations still continue as long as the shared database is available.

Even with an implementation having redundant hardware and a redundant database, you still need a good backup mechanism. Oracle has utilities to perform the logical backup or physical backup at the data level. The logical backup utility **export**, or **EXP**, copies all SQL statements to recreate all database objects and to insert data as well. The export can be at the database, schema, table, or user level. The backed-up data with EXP can be recovered on different platforms with different operating systems and different versions of Oracle. The backup format with EXP is proprietary to Oracle. The **import**, or **IMP**, utility copies the logical backup of data back to the database. A **cold backup** is performed on a database when it is down, or "cold." When a database is running in archive log mode, Oracle saves the redo log into **archive log files.** These files can be used to reconstruct transactions after the last backup. The database can also be backed up while it is running, or "hot"; such a backup is called a **hot backup.** The archive log files can be used with cold as well as hot backups. The recovery manager (RMAN) manages cold backup, hot backup, and the archive log files. The RMAN also enables you to perform incremental hot backups, but it does not support export. Many third-party tools are available for backup and recovery of a database.

The connections to Oracle databases are through **services**, which are processes on an Oracle server or host. The service name is also known as a database alias, which refers to an instance on a host. The relation between a service and an instance is stored in a file called **TNSNAMES.ORA**, which is in Oracle home's *admin* folder in the Windows environment. If there is a change to TNSNAMES.ORA, the change has to be Enterprise-wide. The changes to TNSNAMES.ORA are a problem during the implementation. Oracle solves this problem with an **Oracle Names server**, which

performs name resolution without using TNSNAMES.ORA. An Oracle Names server can integrate with other name-resolution services, such as the Novell Netware Directory service (NDS).

Oracle can connect to databases created under different vendors' software, ranging from Microsoft's PC-database Access to Microsoft's SQL-Server. Microsoft's **Open Database Connectivity** (ODBC) driver running on a client enables the client to connect to a server running SQL*Net. The client need not run SQL*Net, because the ODBC driver emulates it. Oracle also utilizes **gateway** products to connect to non-Oracle database hosts. The gateway translates Oracle SQL queries to a non-Oracle host's native SQL and returns data to the Oracle server. An Oracle gateway is available from the server with a database link, but it is not available from a client.

INSTALLATION

Oracle9i installer software is called **Universal Installer.** The installer looks different based on the platform for which it is bought. The installer is available for character mode, Windows mode, or Motif mode. In spite of the different look, the installer performs the same task on all platforms using the following steps:

- Installs Oracle software components.
- Creates a starter database.
- Executes operating system functions to run Oracle.

You have to select components based on your installation needs and licenses bought from Oracle Corporation. If an installer installs a component you don't need, you can remove it with the installer. If you select a component that is dependent on another component, the installer automatically selects it the other component. Many decisions are made before the installation process. You decide to create or not to create a starter database; select a "home" location for the Oracle software; plan the directory structure for the Oracle data files; define the database block size; specify the number, size, and location of log files; and specify the maximum number of data files allowed. (*Important for students:* If you are installing downloaded trial version of Oracle, use default values wherever possible to avoid any future problems.)

When the installer creates a starter database, the SYSTEM and USERS tablespaces are not allocated enough space, and the block size is very small. The block size for a database cannot be changed. You should find out the block size used by the operating system and the hardware, and then select a block size that is a multiple of that value. The block size should not be larger than the amount of data your operating system can transfer in a single operation. You should select a small block size for a transaction-based system, which has queries involving single rows. For a large system with bulk data retrievals and transfers, you should select a large block

size. The block size is specified in the INIT.ORA file with the DB_BLOCK_SIZE parameter.

CONNECTING TO THE ORACLE9i DATABASE

Oracle9i provides user with different ways to connect to the database:

- SQL*Plus (may go away in the future).
- SQL*Plus Worksheet.
- iSQL*Plus (Web-based).
- Form.
- Reports.

Oracle Enterprise version has OEM bundled with it. OEM's Console is a Windows-based tool to administer Oracle resources. A DBA can connect to the Enterprise Manager with normal, SYSOPER, or SYSDBA privileges. The console allows the DBA to create, start, or shut down databases; create, monitor, or lock users; create and manage tablespaces; or execute SQL statements. You can run OEM Console in a Windows environment by using

Start | Programs | Oracle – OraHome92 | Enterprise Manager Console

Then, launch it stand-alone (as seen in Figure 16-3).

Figure 16-3 Oracle Enterprise manager console—launching stand-alone.

From SQL*Plus or SQL*Plus Worksheet, you can use the following command:

CONNECT *username/password* AS SYSDBA

Figure 16-4 is the login screen for connecting to OEM. The login screen pops up when you select database from the OEM screen. Oracle creates two default users with DBA privileges on installation, SYS and SYSTEM. SYS owns Oracle's Data Dictionary tables and views. SYSTEM owns tables required for Oracle's development tools. Before version 9i, Oracle provided default passwords for SYS and SYSTEM accounts. In version 9i, however, you have the ability to provide a password at installation time. The DBA needs to connect as SYSDBA or SYSOPER to start and shut down databases. In addition, the SYSDBA privilege gives full access to all database objects.

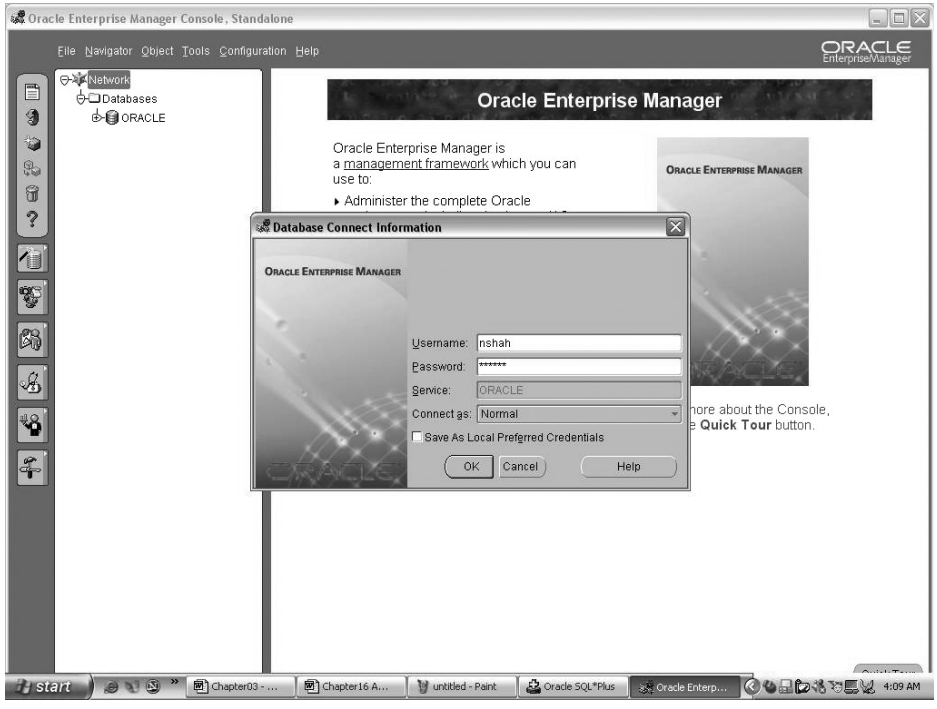


Figure 16-4 Oracle Enterprise manager login screen.

On successful login to the Console, the initial OEM screen is displayed. Here, the DBA can manage Oracle modules, such as Instance, Schema, Security, Storage, and so on (as shown in Figure 16-5).

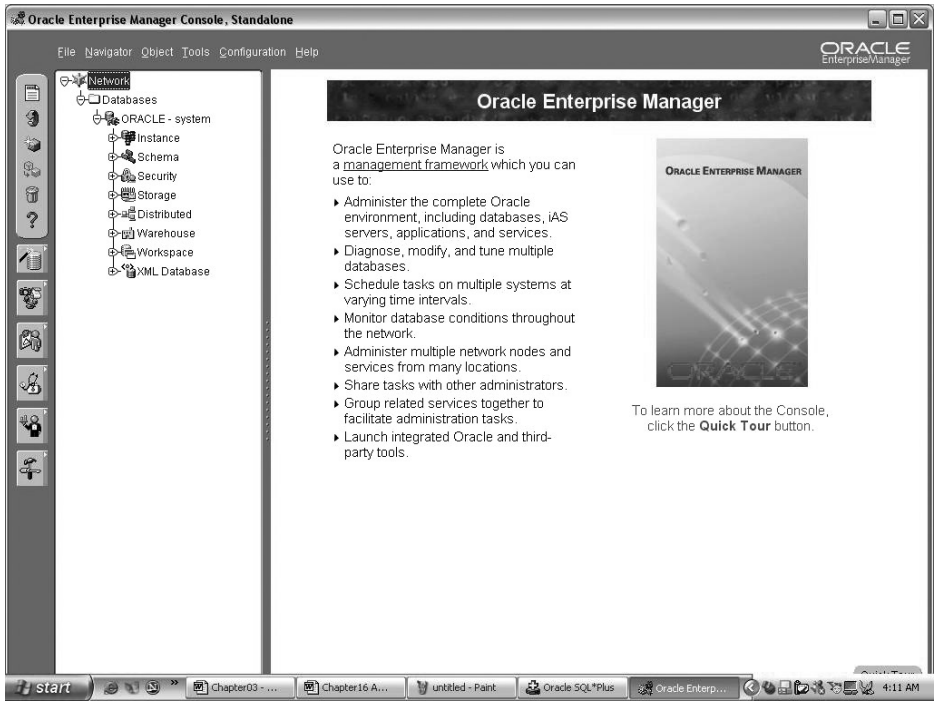


Figure 16-5 Oracle Enterprise Manager initial screen.

INSTANCE AND DATABASE

An instance is used to access the database. You can perform two operations on an instance: STARTUP, and SHUTDOWN. When an instance starts, memory is allocated for SGA, and background processes are started. When an instance shuts down, the database is closed, and memory is released. Database is another entity, and you can perform three operations on it: OPEN, MOUNT, and CLOSED. A database state is changed in one direction only, from MOUNT to OPEN, when an instance is started. The operations can be performed with OEM's Instance Manager, SQL*Plus, or SQL*Plus Worksheet. The general syntax for startup is

STARTUP [FORCE] [NOMOUNT\|MOUNT\|OPEN] [Oracle_sid] [PFILE=name] [RESTRICT]

where FORCE shuts down instance before starting it up, NOMOUNT starts an instance without mounting a database, MOUNT mounts a database for DBA operations only, OPEN makes a database available to users, and RESTRICT restricts access to users with SESSION-related privileges. PFILE contains parameters for startup. If it is not specified, parameters are taken from INIT<Oracle_sid>.ORA file.

When an instance shuts down, the database is closed and dismounted. The general syntax for shutdown is

SHUTDOWN [NORMAL|IMMEDIATE| TRANSACTIONAL|ABORT]

where the NORMAL option shuts down after all users are logged out and all transactions are committed or rolled back, the IMMEDIATE option disconnects all users and rolls back all transactions, the TRANSACTIONAL option finishes all transactions and disallows new transactions, and the ABORT option is like a system failure that requires recovery.

You can use Data Dictionary views *v$session, v$database*, and *v$instance* to get information about user/processes, databases, and instances, respectively.

WORKING WITH ORACLE ENTERPRISE MANAGER (OEM)

Tablespace with Storage Manager

Tablespace is a logical unit of storage, which consists of physical files under an operating system. It may be made up of more than one file, and each file could be physically located on a separate disk. A tablespace can be online or offline. It can be UNDO (Oracle9i onward), PERMANENT, or TEMPORARY. A user with the DBA role or with the CREATE TABLESPACE system privilege may create a tablespace.

Figure 16-6 illustrates use of Oracle Storage Manager in creating a new tablespace, CIS_DATA.ora, for students in a course and its actual location on the Oracle server. The default tablespaces are allocated inadequate space, so the tablespace created in this figure was allocated 100 MB. The SHOW SQL button shows the generated SQL code. Figure 16-7 shows successful creation of tablespace.

You can create a tablespace and specify the operating system file that makes up the tablespace with a CREATE TABLESPACE statement at the command prompt using the following general syntax:

```
CREATE [UNDO | TEMPORARY | PERMANENT] TABLESPACE tablespacename
DATAFILE 'filespecs' SIZE [size K | M]
AUTOEXTEND [ON | OFF NEXT n [K | M] MAXSIZE m [K | M] | UNLIMITED]
DEFAULT STORAGE (storage clause)
BLOCKSIZE 2K | 4K | 8K | 16K | 32K
EXTENT MANAGEMENT DICTIONARY | LOCAL
ONLINE | OFFLINE
PERMANENT | TEMPORARY;
```

A temporary tablespace is for keeping temporary sort data, which is created in the same way but with type Temporary selected (radio button shown in Fig. 16-6). The storage clause is similar to the one used with tables. If the storage clause is not used with a table, then the storage clause from tablespace is used for it.

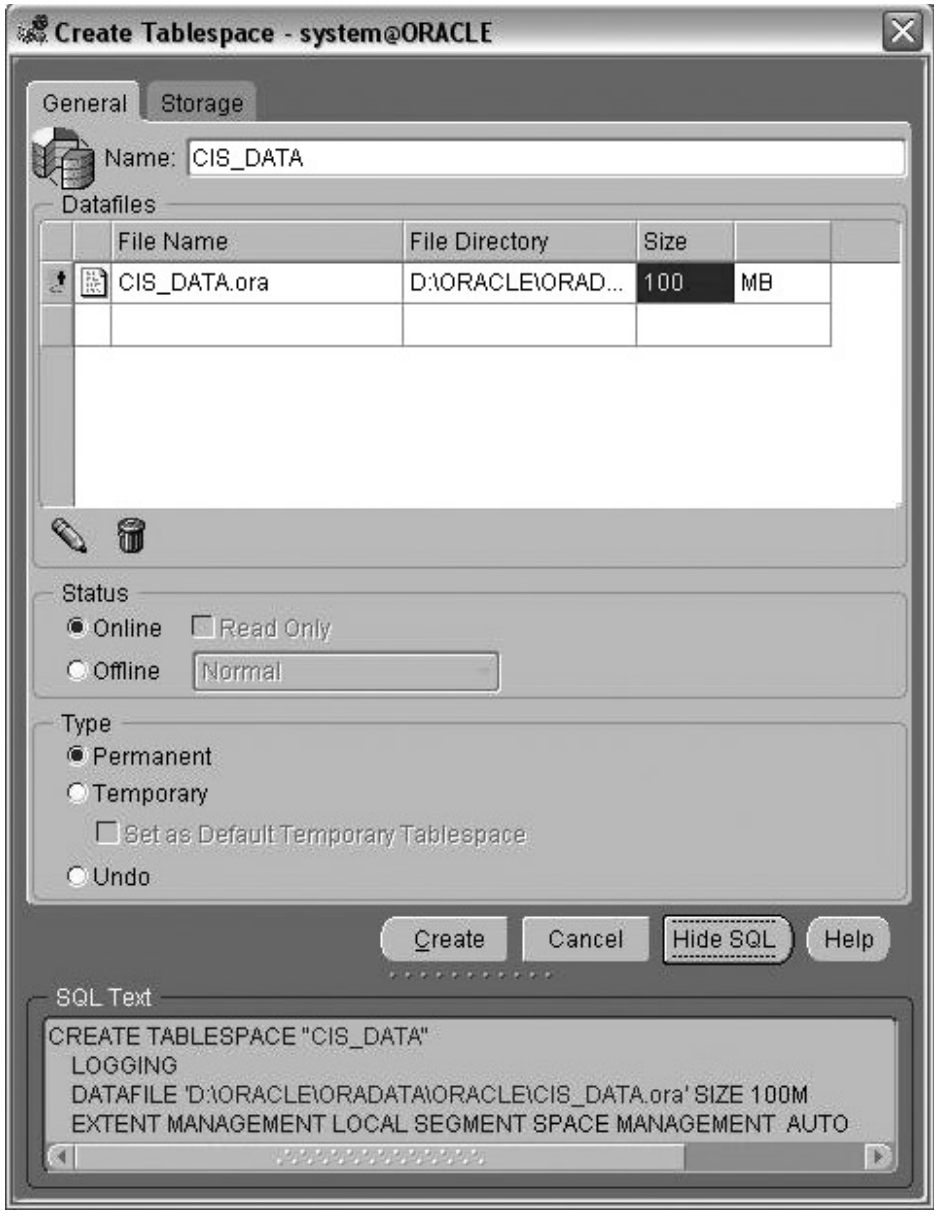


Figure 16-6 Creating a tablespace.

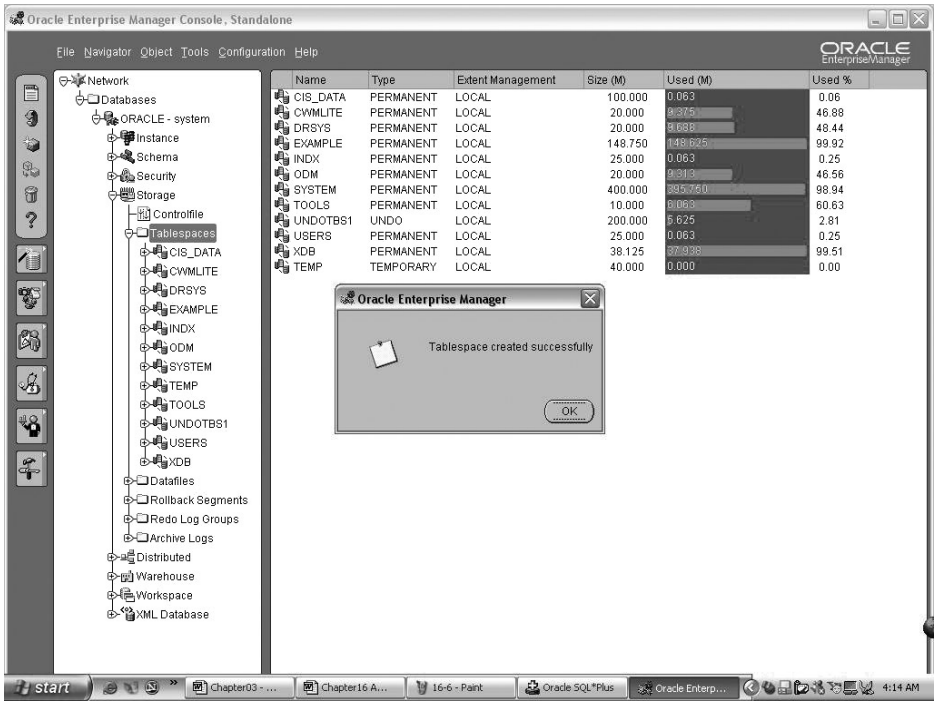


Figure 16-7 Tablespace created with information on all tablespaces.

Storage manager can be used for altering a tablespace or datafile also. You can see information about all tablespaces by using USER_TABLESPACES data dictionary view (see Fig. 16-8).

User and Role with Security Manager

The security of a database is a very important issue for the DBA. Database security prevents unauthorized access and use of objects.

The DBA assigns a unique user ID to each authorized user. The general syntax to create a **user** is

```
CREATE USER username IDENTIFIED BY passwordname
 [DEFAULT TABLESPACE defaulttablespacename]
 [TEMPORARY TABLESPACE temporarytablespacename]
 [QUOTA storagespace ON defaulttablespacename]
 [PROFILE profilename];
```

A DBA or anyone with the CREATE USER system privilege can create a user. If the default tablespace is not specified, DEFAULT TABLESPACE is used.

```
SQL> SELECT TABLESPACE_NAME,   INITIAL_EXTENT,   NEXT_EXTENT,
  2           BLOCK_SIZE FROM USER_TABLESPACES;

TABLESPACE_NAME                INITIAL_EXTENT  NEXT_EXTENT  BLOCK_SIZE
------------------------------ --------------  -----------  ----------
SYSTEM                                  65536                      8192
UNDOTBS1                                65536                      8192
TEMP                                  1048576      1048576         8192
CWMLITE                                 65536                      8192
DRSYS                                   65536                      8192
EXAMPLE                                 65536                      8192
INDX                                    65536                      8192
ODM                                     65536                      8192
TOOLS                                   65536                      8192
USERS                                   65536                      8192
XDB                                     65536                      8192

TABLESPACE_NAME                INITIAL_EXTENT  NEXT_EXTENT  BLOCK_SIZE
------------------------------ --------------  -----------  ----------
CIS_DATA                                65536                      8192
TEMP_DATA                             1048576                      8192

13 rows selected.

SQL>
```

Figure 16-8 Tablespaces storage information.

You must define the quota in K or M to enable user to create tables and indexes. If a profile is not specified, the DEFAULT profile is used.

You can create a user with OEM's Security Manager tool, as shown in Figure 16-9. In the OEM Console, expand the Security tree by clicking on +. You will see Users, Roles, and Profiles folders under Security. Right-click on the Users folder to get a pop-up menu. Then, you will select Create . . . from the menu to get the screen shown in Figure 16-9. User creation involves Name, Profile, Authentication, Password, Confirm Password, Default Tablespace, and Temporary Tablespace entries. You can lock a user's account and expire a user's password from the same screen. If you click on the Show SQL button, you will see the SQL code generated by your entries in user creation screen.

If a user account is locked, the user cannot log in. The locked user gets the following message from the Oracle server:

```
ERROR:
ORA-28000: the account is locked
```

Once a user is created, you can use that user as a template and create another user based on the first user. You need to supply a new username and password only,

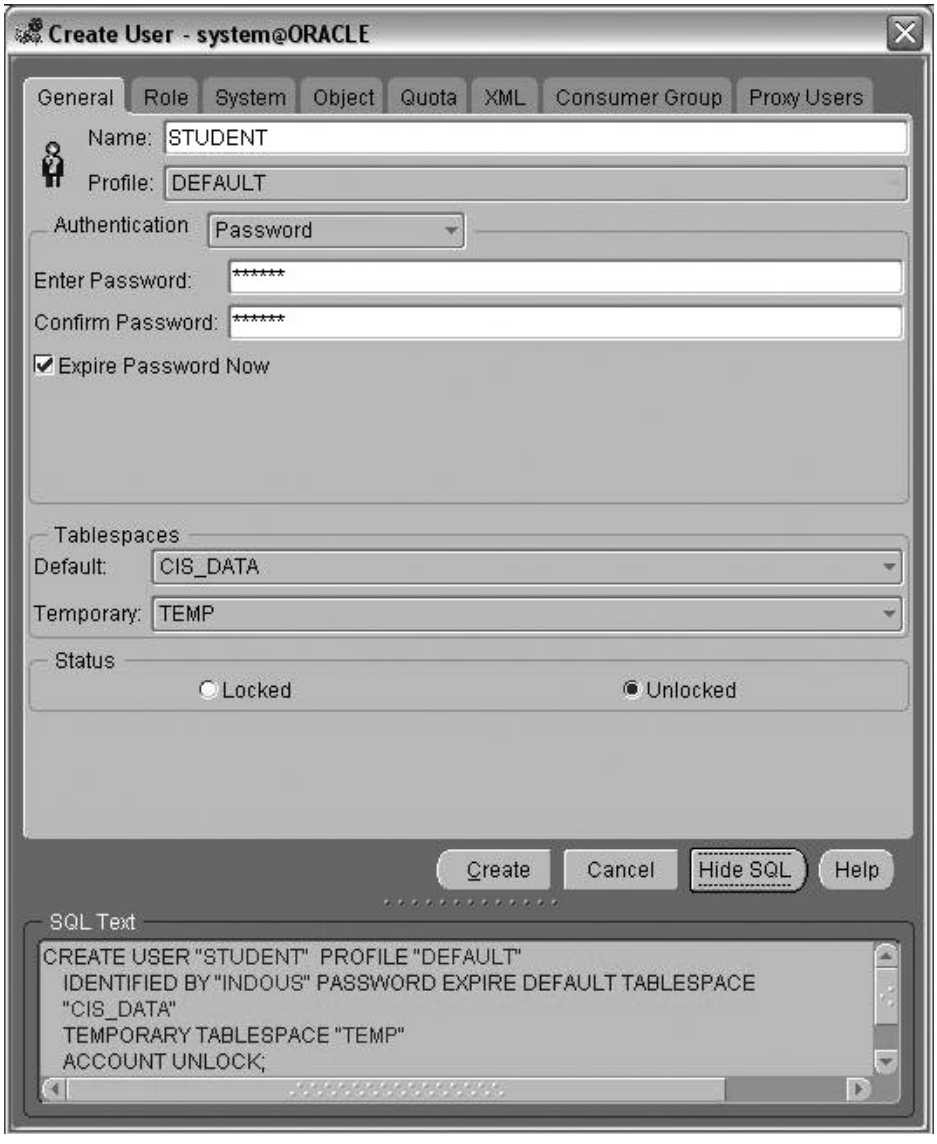


Figure 16-9 Creating a user.

because the new user inherits the profile, tablespaces, and system privileges from the template user. Figure 16-10 shows the process of creating a user based on another user. Right-click on a username to be used as a template, and select Create Like . . .

A user can be removed with the following statement:

DROP USER username [CASCADE];

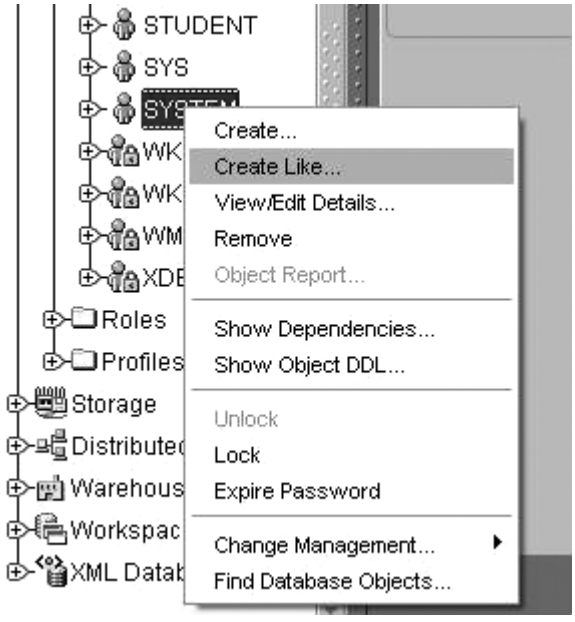


Figure 16-10 Creating a user with Create Like ...

where the CASCADE option removes all objects owned by the user and also removes foreign key constraints.

The Oracle database and initialization files of various components create the initial users at the time of installation. These users have different levels of privileges and are granted different roles. Some of the initial users are:

- **SYS**—This user is granted the DBA role and owns the Data Dictionary. The SYS user is granted all roles.
- **SYSTEM**—This user can manage the database and can also manage packages and tables for additional features within the database. The SYSTEM user is also granted the DBA role.
- **SCOTT**—This user has only the CONNECT and RESOURCE roles. The SCOTT user is a user with the basic end-user privileges.

The other initial users are RMAN, CTXSYS, ORDSYS, MDSYS, and DBSNMP. Because the SYS and SYSTEM users are granted the DBA roles, they inherit all system privileges through the DBA role. The users CTXSYS and MDSYS are also granted all system privileges, but not through the DBA role.

Roles are the same as groups in operating system terminology. Oracle uses roles to grant system and object privileges to users. A DBA or anyone with the

CREATE ROLE privilege can create a role. The role is granted system and object privileges. The role is then granted to a user by the DBA or anyone with the GRANT ANY ROLE system privilege. There are approximately 30 Oracle roles. A user is granted roles according to the need and level of use. When you grant a role to a user, the user inherits all privileges from the role. A user needs at least the CONNECT and RESOURCE roles to create a table in the allocated tablespace. Most end users and students are granted these two roles to work with their own objects.

Figure 16-11 shows the creation of a role called USER with two basic roles, CONNECT and RESOURCE. You can grant the USER role to a user called STUDENT, and the user STUDENT will get the same CONNECT and RESOURCE roles through role USER.

You will follow the same procedure to create a new role in Console. First, right-click on the Roles folder, and then select Create . . . from the pop-up menu. In the General tab (as shown in Fig. 16-11), role and authentication are entered. In the Role tab, various roles are granted to the newly created role. Once a role is created, a "Role created successfully" message is displayed.

Some default roles in the database also allow the user to take certain actions. When a new database is created with the CREATE DATABASE command, six default roles are automatically created. The DBA can run different scripts and create more roles. The six default roles are:

1. **CONNECT:** A user with this role can connect to the database and create any object other than a segment.
2. **RESOURCE:** This role is an extension to the CONNECT role. A user with this role can create types, procedures, triggers, and snapshots.
3. **DBA:** This role has all system privileges except for UNLIMITED TABLESPACE, because that is not granted to a role.
4. **DELETE_CATALOG_ROLE:** This role, which allows deletion of any object owned by SYS, is granted to the DBA and the SYS schema explicitly.
5. **EXECUTE_CATALOG_ROLE:** This role, which allows execution of any object owned by SYS, is granted to the DBA, SYS schema, EXP_FULL_DATABASE role, and IMP_FULL_DATABASE role.
6. **SELECT_CATALOG_ROLE:** This role, which allows selection from any object owned by SYS, is granted to the DBA, SYS schema, EXP_FULL_DATABASE role, and IMP_FULL_DATABASE role.

You can use the Data Dictionary view DBA_ROLES to find roles and their password. Oracle9i creates 30 roles on installation, whereas Oracle6 contained only three roles—CONNECT, RESOURCE, and DBA!

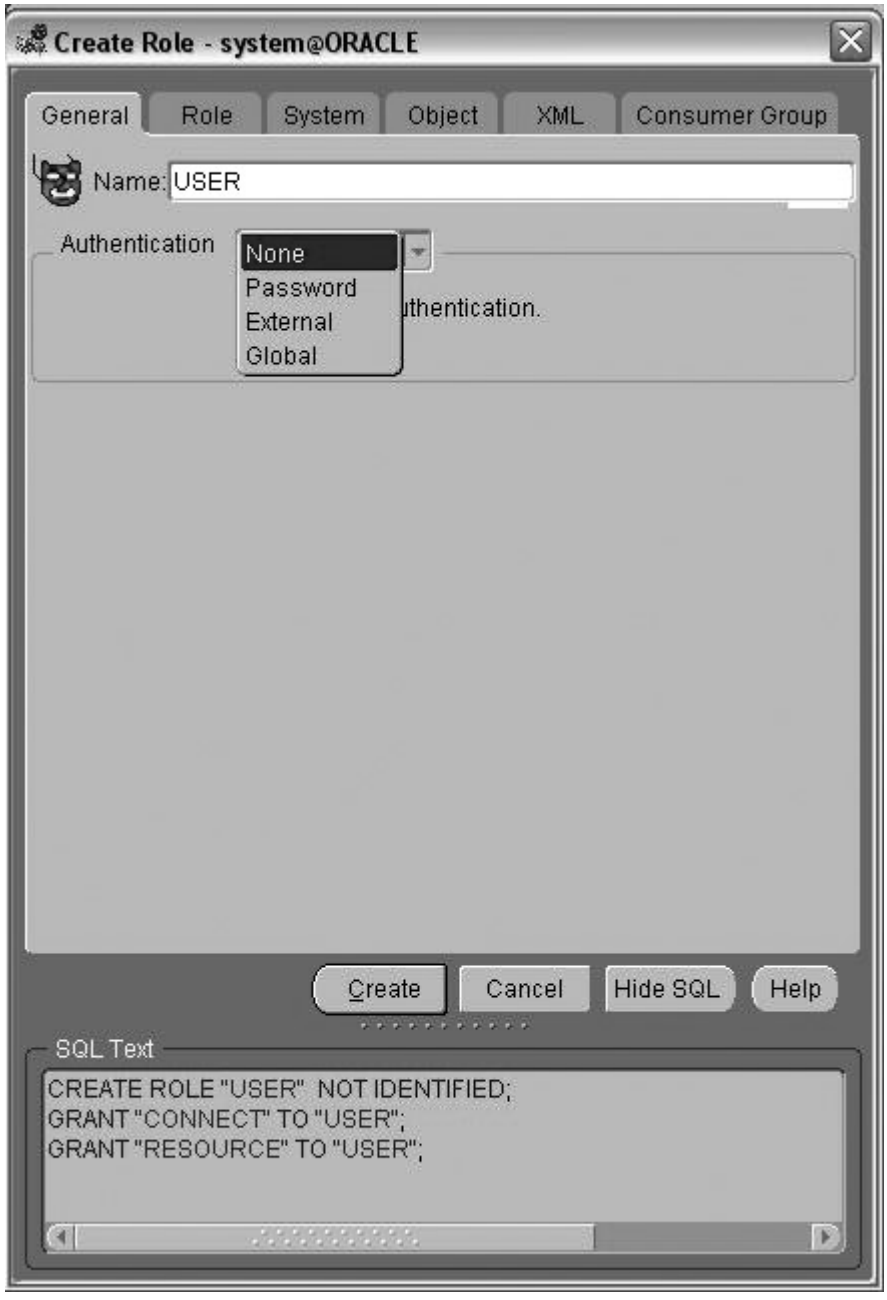


Figure 16-11 Creating a role.

SYSTEM PRIVILEGES

System privileges allow a user to take certain actions within the database. Figure 16-12 shows a few system privileges needed for certain types of actions within the database. As you know, Oracle9i has more than 100 defined system privileges. They can be categorized into three types based on their effect:

System Privilege	Type of Action
CREATE	Create an object in a user's own schema.
CREATE ANY	Create an object in another user's schema.
CREATE SESSION	Connect to database.
DROP	Drop an object in a user's own schema.
DROP ANY	Drop an object in another user's schema.
ALTER SYSTEM	Manipulate an instance.
ALTER DATABASE	Manipulate database.
ALTER USER	Change a user's role or password.
ALTER|CREATE|DROP|MANAGE TABLESPACE	Manipulate a tablespace.

Figure 16-12 System privileges and actions.

1. **Privileges that affect the entire database:** The DBA or user who is granted the DBA role has such privileges. These privileges allow the DBA to alter the database, create users, create roles, grant roles, manage tablespace, remove users, and so on.
2. **Privileges that allow a user to create objects in the user's own schema:** These privileges allow user to create tables, views, sequences, synonyms, procedures, triggers, and so on. These privileges are granted to CONNECT and RESOURCE roles. These roles are then granted to a user rather than granting individual privileges.
3. **Privileges that allow a user to manipulate objects in any schema:** These privileges allow you to manipulate objects in other users' schemas. The DBA and users with the DBA role are granted these privileges—for example, CREATE ANY, DROP ANY, ALTER ANY, SELECT ANY, INSERT ANY, EXECUTE ANY, and so on.

System privileges are granted carefully to users after intelligent planning. Only DBAs and users with GRANT ANY PRIVILEGE system privilege can grant system privileges to other users. The privileges are granted through a role or granted individually with the following statement:

GRANT systemprivilege TO username [WITH ADMIN OPTION];

If WITH ADMIN OPTION is used, the user can pass on that privilege to another user. Data Dictionary view DBA_SYS_PRIVS can show privileges granted to a user/role (see Fig. 16-13). The query in Figure 16-13 returned 139 privileges granted to the DBA role. You can also find out the same information from OEM Security Manager.

```
SQL> SELECT grantee, privilege, admin_option
  2  FROM dba_sys_privs
  3  WHERE grantee = 'DBA';

GRANTEE                        PRIVILEGE                      ADM
------------------------------ ------------------------------ ---
DBA                            AUDIT ANY                      YES
DBA                            DROP USER                      YES
DBA                            RESUMABLE                      YES
DBA                            ALTER USER                     YES
DBA                            ANALYZE ANY                    YES
DBA                            BECOME USER                    YES
DBA                            CREATE ROLE                    YES
DBA                            CREATE RULE                    YES
DBA                            CREATE TYPE                    YES
DBA                            CREATE USER                    YES
DBA                            CREATE VIEW                    YES
.  .  .
139 rows selected.

SQL>
```

Figure 16-13 DBA_SYS_PRIVS.

ORACLE DATA DICTIONARY

The Data Dictionary in Oracle consists of tables and related views. The Data Dictionary gives the structure and inside view of the Oracle database, and it has grown with each Oracle release. You can use SQL statements with Data Dictionary tables/views just like you would with user tables/views. You can get information about various Oracle objects and users of the database. The Data Dictionary contains **static Data Dictionary views**, which are owned by user SYS. The static Data Dictionary views are based on tables that are updated with Oracle DDL statements only. It should not be updated with Data Manipulation Language (DML) statements. The SYS tables, views, and synonyms are created with the CATALOG.SQL script file. The procedural objects are created with the CATAPROC.SQL file. These scripts files are provided by Oracle and are copied into the ORACLE\ORA92\RDBMS\ADMIN directory along with other script files.

The script files also create public synonyms for Data Dictionary views (e.g., synonym *tabs* for user_tables and *seq* for user_sequences). Before displaying information from a view, DESCRIBE its structure first to avoid unnecessary information.

There are additional views known as **dynamic performance Data Dictionary views**, or simply **V$ views**. The V$ views are based on internal memory structures, or virtual tables, which begin with the X$ prefix. The V$ views and X$ tables have information about the instance. The information in the two Data Dictionary views is:

- *Static Data Dictionary views*—These views are for information on database objects, database data files, and database users. The views begin with USER_<objects_you_own>, ALL_<objects_you_have_access_to>, DBA_<all_objects>—for example, DBA_CONSTRAINTS, DBA_CONS_COLUMNS, DICTIONARY, DBA_INDEXES, USER_TABLES, ALL_TRIGGERS, DBA_ROLES, DBA_PROFILES, DBA_SYS_PRIVS, DBA_USERS, DBA_TABLESPACES, DBA_VIEWS, and so on.
- *Dynamic performance Data Dictionary views*—These views are for information on instance objects, archive log files, and currently connected users—for example, V$SESSION, V$PROCESS, V$TABLESPACE, V$SQL, V$SGA, and so on.

IN A NUTSHELL . . .

- The DBA is responsible for installing the Oracle database, managing daily operations, and running the database at peak performance.
- The Oracle database is the data stored on disk, and the Oracle instance is the System Global Area (SGA) memory and background processes.
- An instance contains four types of files: the parameter file INST.ORA, control files, data files, and redo log files.
- A tablespace is the basic storage allocation to a database.
- A user account or username is called a schema. An Oracle database is created with two schemas, SYS and SYSTEM.
- Oracle uses three configurations for availability to users: replication, hot standby, and Oracle parallel server.
- Oracle provides good backup mechanisms in the forms of EXP/IMP, cold backup, archive log files, and hot backup.
- The installation process installs Oracle components, creates a starter database, and executes operating system functions to run Oracle.
- Oracle networking connects clients to a database and a database to another database. Connections in Oracle are through services. The TNSNAMES.ORA file contains relations between a service and an instance.
- Oracle can connect to other vendor-supplied databases by using ODBC drivers and gateway products.

- Security is a very important issue for the DBA. The DBA creates users and grants them privileges and roles.
- System privileges are categorized into those that affect the entire database, those that allow users to create objects in their own schema, and those that allow users to manipulate objects in any schema.
- Oracle creates default roles and initial users at the time of installation.
- Oracle provides a powerful set of tools, such as SQL*Plus, SQL*Plus Worksheet, EXP/IMP, and Enterprise Manager.
- SQL*Plus is an environment to interface with the database. It provides users with editing, file-related, variable-related, formatting, and environment variable commands.
- Oracle's Data Dictionary contains static and dynamic tables and views.

EXERCISE QUESTIONS

True/False:

1. The file TNSNAMES.ORA contains the names of default roles and initial users.
2. Three major areas of Oracle architecture are SGA, background processes, and physical storage structures.
3. A user needs CONNECT and RESOURCE roles to create a table in his or her own schema.
4. The replication method uses separate databases by duplicating the entire implementation of a database on multiple computer systems.
5. The hot standby database method uses only one database at a time, and the other standby copy is in recover mode at all times.
6. An Oracle instance is the SGA and physical storage structures.
7. SYS user is granted the DBA role and owns the Data Dictionary.
8. An initial SCOTT user, created by Oracle, has all system privileges.
9. SQL*Plus Worksheet is a Web-based environment to connect to the Oracle database.
10. An instance is first opened and then mounted.

Answer the Following Questions:

1. What is the difference among users, roles, and system privileges?
2. What are the duties of a DBA?
3. How does Oracle9i make sure that the database is available to users at all times?
4. Explain the backup mechanisms used by Oracle9i.
5. Discuss three types of Oracle system privileges.
6. Describe Oracle9i architecture.
7. Describe Oracle's Data Dictionary.

Appendix A

Sample Databases: Table Definitions

This textbook utilizes two sample databases thoughout its chapters. In this section, the table structures are described. The primary key columns are underlined. It is always a good idea to have primary key columns with the NUMBER data type, but other data types are also used here for primary key columns (e.g., the primary key column in the TERM table. All columns, their appropriate data types, and constraints are given. However, some of the columns, though not used for any mathematical operations, are assigned the NUMBER data type for simplicity. They can also be assigned one of the character data types, CHAR or VARCHAR2. Students may modify the table structures as desired.

THE INDO–US (IU) COLLEGE STUDENT DATABASE

STUDENT

Column Name	Data Type	Constraints
StudentId	CHAR(5)	PRIMARY KEY
Last	VARCHAR2(15)	NOT NULL
First	VARCHAR2(15)	NOT NULL
Street	VARCHAR2(25)	
City	VARCHAR2(15)	

Column Name	Data Type	Constraints
State	CHAR(2)	
Zip	CHAR(5)	
StartTerm	CHAR(4)	FOREIGN KEY
BirthDate	DATE	
FacultyId	NUMBER(3)	FOREIGN KEY
MajorId	NUMBER(3)	FOREIGN KEY
Phone	CHAR(10)	

FACULTY

Column Name	Data Type	Constraints
FacultyId	NUMBER(3)	PRIMARY KEY
Name	VARCHAR2(15)	NOT NULL
RoomId	NUMBER(2)	FOREIGN KEY
Phone	CHAR(3)	UNIQUE
DeptId	NUMBER(1)	FOREIGN KEY

CRSSECTION

Column Name	Data Type	Constraints
CsId	NUMBER(4)	PRIMARY KEY
CourseId	VARCHAR2(6)	FOREIGN KEY, NOT NULL
Section	CHAR(2)	NOT NULL
TermId	CHAR(4)	FOREIGN KEY, NOT NULL
FacultyId	NUMBER(3)	FOREIGN KEY
Day	VARCHAR2(2)	
StartTime	VARCHAR2(5)	
EndTime	VARCHAR2(5)	
RoomId	NUMBER(2)	FOREIGN KEY
MaxCount	NUMBER(2)	CHECK

COURSE

Column Name	Data Type	Constraints
CourseId	VARCHAR2(6)	PRIMARY KEY
Title	VARCHAR2(20)	UNIQUE
Credits	NUMBER(1)	CHECK
PreReq	VARCHAR2(6)	FOREIGN KEY

REGISTRATION

Column Name	Data Type	Constraints
StudentId	CHAR(5)	COMPOSITE PRIMARY KEY, FOREIGN KEY
CsId	NUMBER(4)	COMPOSITE PRIMARY KEY, FOREIGN KEY
Midterm	CHAR	CHECK
Final	CHAR	CHECK
RegStatus	CHAR	CHECK

ROOM

Column Name	Data Type	Constraints
Room Type	CHAR	PRIMARY KEY
RoomDesc	VARCHAR2(9)	

LOCATION

Column Name	Data Type	Constraints
RoomId	NUMBER(2)	PRIMARY KEY
Building	VARCHAR2(7)	NOT NULL
RoomNo	CHAR(3)	NOT NULL, UNIQUE
Capacity	NUMBER(2)	CHECK
RoomType	CHAR	FOREIGN KEY

TERM

Column Name	Data Type	Constraints
TermId	CHAR(4)	PRIMARY KEY
TermDesc	VARCHAR2(11)	
StartDate	DATE	
EndDate	DATE	

DEPARTMENT

Column Name	Data Type	Constraints
DeptId	NUMBER(1)	PRIMARY KEY
DeptName	VARCHAR2(20)	
FacultyId	NUMBER(3)	FOREIGN KEY

MAJOR

Column Name	Data Type	Constraints
MajorId	NUMBER(3)	PRIMARY KEY
MajorDesc	VARCHAR2(25)	

THE NAMANNAVAN (N2) CORPORATION EMPLOYEE DATABASE

EMPLOYEE

Column Name	Data Type	Constraints
EmployeeId	NUMBER(3)	PRIMARY KEY
Lname	VARCHAR2(15)	NOT NULL
Fname	VARCHAR2(15)	NOT NULL
PositionId	NUMBER(1)	FOREIGN KEY
Supervisor	NUMBER(3)	FOREIGN KEY
HireDate	DATE	
Salary	NUMBER(6)	
Commission	NUMBER(5)	
DeptId	NUMBER(2)	FOREIGN KEY
QualId	NUMBER(1)	FOREIGN KEY

Dept

Column Name	Data Type	Constraints
DeptId	NUMBER(2)	PRIMARY KEY
DeptName	VARCHAR2(12)	
Location	VARCHAR2(15)	
EmployeeId	NUMBER(3)	FOREIGN KEY

EMPLEVEL

Column Name	Data Type	Constraints
LevelNo	NUMBER(1)	PRIMARY KEY
LowSalary	NUMBER(6)	
HighSalary	NUMBER(6)	

POSITION

Column Name	Data Type	Constraints
PositionId	NUMBER(1)	PRIMARY KEY
PosDesc	VARCHAR2(10)	

DEPENDENT

Column Name	Data Type	Constraints
EmployeeId	NUMBER(3)	COMPOSITE PRIMARY KEY, FOREIGN KEY
DependentId	NUMBER(1)	COMPOSITE PRIMARY KEY
DepDOB	DATE	
Relation	VARCHAR2(8)	

QUALIFICATION

Column Name	Data Type	Constraints
QualId	NUMBER(1)	PRIMARY KEY
QualDesc	VARCHAR2(11)	

Appendix B

Quick Reference to SQL and PL/SQL Syntax

SQL KEY WORDS

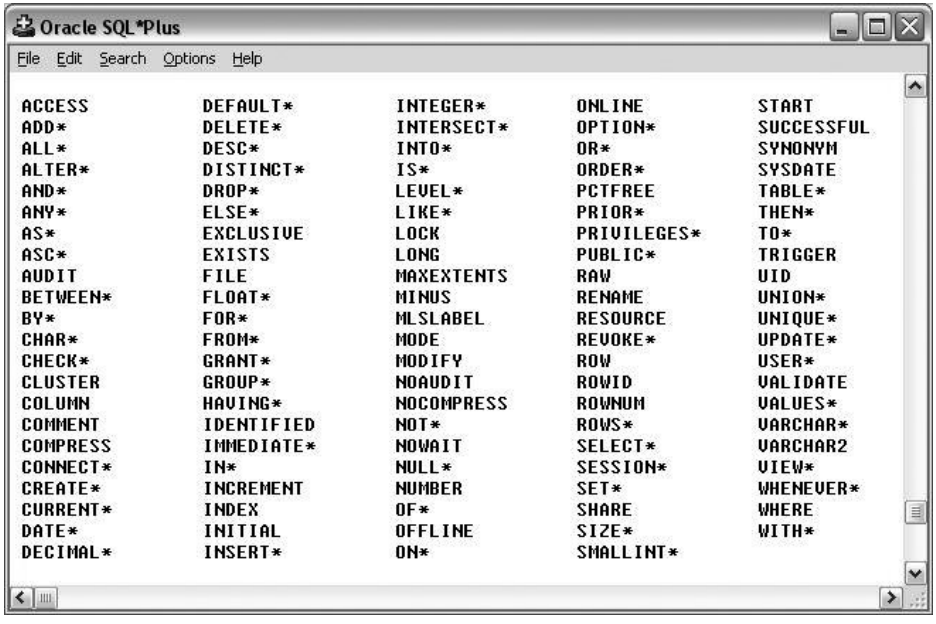


Figure B-1 SQL keywords.

PL/SQL KEY WORDS

Figure B-2 PL/SQL keywords.

SQL AND PL/SQL SYNTAX

In the syntax for various SQL statements and PL/SQL blocks, the following convention is used:

- The key words are in uppercase letters.
- The user-defined names are in lowercase or mixed case.
- The optional items are enclosed in brackets ([]).
- The pipe symbol (|) is used to denote OR.

Creating a Table

```
CREATE TABLE [schema.] tablename
(column1 datatype [ CONSTRAINT constraint_name ] constraint_type . . .,
(column2 datatype [ CONSTRAINT constraint_name ] constraint_type . . .,
 . . .
[ CONSTRAINT constraint_name ] constraint_type (column, . . . ), . . .)
[TABLESPACE tablespacename]
[STORAGE (INITIAL n K|M NEXT n K|M)]
[PCTFREE p];
```

Column-Level Constraint

```
Column datatype [ CONSTRAINT constraint_name ] constraint_type,
```

Table-Level Constraint

```
[ CONSTRAINT constraint_name ] constraint_type ( column, . . . ),
```

Adding a Column to an Existing Table

```
ALTER TABLE tablename
    ADD columnname datatype;
```

Modifying an Existing Column

```
ALTER TABLE tablename
    MODIFY columnname newdatatype;
```

Adding a Constraint to a Table

```
ALTER TABLE tablename
    ADD [ CONSTRAINT constraint_name ] constraint_type (columnname|
    expression) [ References tablename (columnname) ]
```

Dropping a Column (Oracle8 Onward)

```
ALTER TABLE tablename DROP COLUMN columnname;
```

Setting a Column as Unused (Oracle8 Onward)

```
ALTER TABLE tablename SET UNUSED (columnname);
```

Dropping an Unused Column (Oracle8 Onward)

```
ALTER TABLE tablename DROP UNUSED COLUMNS;
```

Renaming a Column (Oracle9i Onward)

```
ALTER TABLE tablename RENAME COLUMN oldname TO newname;
```

Renaming a Constraint (Oracle9i Onward)

```
ALTER TABLE tablename
    RENAME CONSTRAINT oldname TO newname;
```

Dropping a Table

```
DROP TABLE tablename;
```

Renaming a Table

```
RENAME oldtablename TO newtablename;
```

Truncating a Table

```
TRUNCATE TABLE tablename [REUSE STORAGE];
```

Inserting a New Row into a Table

```
INSERT INTO tablename [ (column1, column2, column3, . . .) ]
    VALUES (value1, value2, value3, . . .);
```

Customized Prompts

```
ACCEPT variablename PROMPT 'prompt message'
```

Updating Rows

```
UPDATE tablename SET column1 = newvalue
    [, column2 = newvalue, . . .]
    [ WHERE condition ];
```

Deleting Rows

```
DELETE [ FROM ] tablename
    [ WHERE condition ];
```

Dropping a Constraint

```
ALTER TABLE tablename
    DROP PRIMARY KEY | UNIQUE (columnname) |
    CONSTRAINT constraintname [ CASCADE ];
```

Enabling|Disabling a Constraint

```
ALTER TABLE tablename
    DISABLE CONSTRAINT constraintname | PRIMARY KEY [ CASCADE ];
ALTER TABLE tablename
    ENABLE CONSTRAINT constraintname;
```

Retrieving Data from a Table

```
SELECT column, groupfunction (column)
    FROM tablename [WHERE condition(s)]
    [GROUP BY column|expression]
    [HAVING groupcondition]
    [ORDER BY column|expression [ASC|DESC]];
```

Define Command

```
DEFINE variable [= value]
SET DEFINE ON | character
```

Decode Function

```
DECODE (column | expr, value1, action1,
    [value2, action2, . . .,]
    [, default]);
```

Case Structure

```
CASE WHEN condition1 THEN
        expression1
    WHEN condition2 THEN
        expression2
    . . .
    [ELSE expression]
END
```

Joining Tables: Equijoin or Outer Join

```
SELECT tablename1.columnname, tablename2.columnname
  FROM tablename1, tablename2
  WHERE tablename1.columnname [(+)] = tablename2.columnname [(+)];
```

Set Operation

```
SELECT-query1
UNION | UNION ALL | MINUS | INTERSECT
SELECT-query2;
```

Select Subquery

```
SELECT columnlist
    FROM tablename
    WHERE columnname operator
                (SELECT columnname
                FROM tablename
                WHERE condition);
```

Creating a Table Using a Subquery

```
CREATE TABLE tablename
AS
SELECT-query;
```

Inserting a Row Using a Subquery

```
INSERT INTO tablename [(column aliases)]
SELECT columnnames FROM tablename WHERE condition;
```

Inserting into Multiple Tables

```
INSERT ALL
[WHEN condition1 THEN] INTO table1
    VALUES (columnlist)
[WHEN condition2 THEN] INTO table2
    VALUES (columnlist)
. . .
SELECT columnlist
FROM tablename [WHERE condition(s)];
```

Updating Using a Subquery

```
UPDATE tablename
SET (columnnames) operator
    (SELECT-FROM-WHERE SUBQUERY)
WHERE condition;
```

Deleting Using a Subquery

```
DELETE FROM tablename
    WHERE columnname operator
        (SELECT-FROM-WHERE SUBQUERY);
```

Top-N Query

```
SELECT ROWNUM, columnlist
FROM (SELECT-query with ORDER BY clause)
WHERE ROWNUM < | <= n;
```

Merge Statement

```
MERGE INTO tablename tablealias
USING ((SELECT-query) tablealias
ON joincondition
WHEN MATCHED THEN
    UPDATE-statement
WHEN UNMATCHED THEN
    INSERT-statement;
```

Creating a View

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW viewname
    [column aliases]
AS SELECT-subquery
[WITH CHECK OPTION [CONSTRAINT constraintname]]
[WITH READ ONLY];
```

Altering a View

```
ALTER VIEW viewname COMPILE;
```

Dropping a View

```
DROP VIEW viewname;
```

Creating a Sequence

```
CREATE SEQUENCE sequencename
    [INCREMENT BY n]
    [START WITH s]
    [MAXVALUE x | NOMAXVALUE]
    [MINVALUE m | NOMINVALUE]
    [CYCLE | NOCYCLE]
    [CACHE c | NOCACHE]
    [ORDER | NOORDER];
```

Modifying a Sequence

```
ALTER SEQUENCE sequencename
    [INCREMENT BY n]
    [MAXVALUE x | NOMAXVALUE]
    [MINVALUE m | NOMINVALUE]
    [CYCLE | NOCYCLE]
    [CACHE c | NOCHACHE]
    [ORDER | NOORDER];
```

Creating a Synonym

```
CREATE [PUBLIC] SYNONYM synonymname
FOR [schema.]objectname;
```

Dropping a Synonym

```
DROP VIEW synonymname;
```

Creating an Index

```
CREATE INDEX indexname
    ON tablename (columnname1 [, columnname2] . . .);
```

Rebuilding an Index

```
ALTER INDEX indexname REBUILD;
```

Locking Rows for Update

```
SELECT columnnames
FROM tablenames
WHERE condition
FOR UPDATE OF columnnames
[NOWAIT];
```

Creating a User

```
CREATE USER username[PROFILE profilename]
IDENTIFIED BY password
    [DEFAULT TABLESPACE tablespacename]
     [TEMPORARY TABLESPACE tablespacename]
    [PASSWORD EXPIRE] [ACCOUNT UNLOCK];
```

Changing a User's Password

```
ALTER USER username
    IDENTIFIED BY password;
```

Granting System Privileges

```
GRANT privilege1 [, privilege2 . . .]
    TO username1 [, username2 . . .];
```

Granting Object Privileges

```
GRANT objectprivileges [(columnnames)] | ALL
    ON objectname
    TO user|role|PUBLIC
    [WITH GRANT OPTION];
```

Revoking Privileges

```
REVOKE privilege1 [, privilege2 . . .] | ALL
ON objectname
FROM users|role|PUBLIC
[CASCADE CONSTRAINTS];
```

PL/SQL Anonymous Block

```
DECLARE
    declaration of constants, variables, cursors, and exceptions
BEGIN
    PL/SQL and SQL statements
EXCEPTION
    actions for error conditions
END;
```

PL/SQL Variable/Constant Declaration

```
DECLARE
    identifiername [CONSTANT] datatype
        [NOT NULL] [:= |DEFAULT expression];
```

Anchored Variable Declaration

```
variablename typeattribute%TYPE [value assignment];
```

Assignment Operation

```
variablename:=literal|variablename|Expression;
```

IF-THEN-END IF

```
IF condition(s) THEN
    action statements
END IF
```

If-Then-Else-End If

```
IF condition(s) THEN
    action statements 1
ELSE
    action statements 2
END IF
```

If-Then-Elsif-End If

```
IF condition(s)1 THEN
    action statements 1
ELSIF condition(s)2 THEN
    action statements 2
. . .
ELSIF condition(s)N THEN
    action statement N
[ELSE
    else action statements]
END IF
```

Case Statement

```
CASE [variablename]
  WHEN value1 | condition1 THEN
      action_statements1
  WHEN value2 | condition2 THEN
      action_statements2
  . . .
  [ELSE action_statements]
END CASE;
```

Basic Loop

```
LOOP
    looping statement1;
    looping statement2;
    . . .
    looping statementN;
    EXIT [WHEN condition];
END LOOP;
```

While Loop

```
WHILE condition LOOP
    looping statement 1;
    looping statement 2;
    . . .
    looping statement n;
END LOOP;
```

For Loop

```
FOR counter IN [REVERSE] lower..upper LOOP
    looping statement 1
    looping statement 2
    . . .
    looping statement n
END LOOP;
```

Bind/Host Variable

```
VARIABLE variablename datatype
```

Select-Into in PL/SQL

```
SELECT columnnames
INTO variablenames | recordname
FROM tablename
WHERE condition;
```

Explicit Cursor Declaration

```
CURSOR cursorname IS
    SELECT statement;
```

Opening an Explicit Cursor

```
OPEN cursorname;
```

Fetching a Row from an Explicit Cursor

```
FETCH cursorname INTO variablelist | recordname;
```

Closing an Explicit Cursor

```
CLOSE cursorname;
```

Cursor For Loop

```
FOR recordname IN cursorname LOOP
    loop statements;
    . . .
END LOOP;
```

Cursor For Loop with a Subquery

```
FOR recordname IN (SELECT-query) LOOP
    loop statements;
    . . .
END LOOP;
```

Where Current Of Clause

```
UPDATE tablename
    SET clause
WHERE CURRENT OF cursorname;
```

Cursor with Select–For Update

```
CURSOR cursorname IS
    SELECT columnnames
    FROM tablename
    [WHERE condition]
    FOR UPDATE [OF columnnames] [NOWAIT];
```

Cursor with Parameters

```
CURSOR cursorname
    [(parameter1 datatype, parameter2 datatype, . . . )]
IS
SELECT query;
```

Ref Cursor Type

```
TYPE cursortypename IS REF CURSOR [RETURN returntype];
cursorvarname cursortypename;
```

Opening a Cursor Variable

```
OPEN cursorname | cursorvarname FOR SELECT query;
```

Fetching from a Cursor Variable

```
FETCH cursorvarname INTO recordname | variablelist;
```

Exception Section

```
EXCEPTION
    WHEN exceptionname1 [OR exceptionname2, . . .] THEN
        executable statements
    [WHEN exceptionname3 [OR exceptionname4, . . .] THEN
        executable statements]
    [WHEN OTHERS THEN
        executable statements]
```

Pragma Exception_Init Directive

```
exceptionname EXCEPTION;
    PRAGMA EXCEPTION_INIT (exceptionname, errornumber);
```

Raise_Application_Error Procedure

```
RAISE_APPLICATION_ERROR(error_code, error_message [, TRUE|FALSE];
```

Creating a PL/SQL Record

```
TYPE recordtypename IS RECORD
    (fieldname1 datatype | variable%TYPE | table.column%TYPE |
        table%ROWTYPE [[NOT NULL] := | DEFAULT expression]
    [, fieldname2 . . .
    , fieldname3 . . .);
recordname recordtypename;
```

Declaring a PL/SQL Table

```
TYPE tabletypename IS TABLE OF
    datatype | variablename%TYPE | tablename.columnname%TYPE
    [NOT NULL] INDEX BY BINARY_INTEGER;
tablename tabletypename;
```

Declaring a PL/SQL Varray

```
DECLARE
    TYPE varraytypename IS VARRAY (size) OF elementtype [NOT NULL];
    varrayname varraytypename;
```

PL/SQL Procedure

```
CREATE [ OR REPLACE ] PROCEDURE procedurename
    [ (parameter1 [, parameter2 . . .] ) ]
IS
    [ constant | variable declarations ]
BEGIN
    executable statements
[ EXCEPTION
    exception handling statements ]
END [ procedurename ];
```

Calling a Procedure

```
procedurename [ (parameter1, . . .) ];
```

Recompiling a Procedure

```
ALTER PROCEDURE procedurename COMPILE;
```

PL/SQL Function

```
CREATE [ OR REPLACE ] FUNCTION functionname
    [ (parameter1 [, parameter2 . . .] ) ]
    RETURN datatype
IS
     [ constant | variable declarations ]
BEGIN
     executable statements
     RETURN returnvalue
[ EXCEPTION
     exception handling statements
     RETURN returnvalue ]
END [ functionname ];
```

PL/SQL Package Specification

```
CREATE [ OR REPLACE ] PACKAGE packagename
IS
     [ constant, variable, and type declarations ]
     [ exception declarations ]
     [ cursor specifications ]
     [ function specifications ]
     [ procedure specifications ]
END [ packagename ];
```

PL/SQL Package Body

```
PACKAGE BODY packagename
IS
     [ variable and type declarations ]
     [ cursor specifications and SELECT queries ]
     [ header and body of functions ]
     [ header and body of procedures ]
[ BEGIN
     executable statements ]
[ EXCEPTION
     exception handlers ]
END [ packagename ];
```

PL/SQL Trigger

```
CREATE [ OR REPLACE ] TRIGGER triggername
INSTEAD OF|BEFORE|AFTER triggeringevent ON tablename|viewname
```

```
[ FOR EACH ROW ]
[ WHEN condition ]
DECLARE
    declaration statements
BEGIN
    executable statements
EXCEPTION
exception handling statements
END;
```

Creating a Tablespace

```
CREATE[UNDO|TEMPORARY|PERMANENT] TABLESPACE tablespacename
    DATAFILE 'filespecs' SIZE [size K | M]
    AUTOEXTEND [ON | OFF NEXT n[K | M] MAXSIZE m[K | M] | UNLIMITED]
    DEFAULT STORAGE (storage clause)
    BLOCKSIZE 2K | 4K | 8K | 16K | 32K
EXTENT MANAGEMENT DICTIONARY | LOCAL
ONLINE | OFFLINE
PERMANENT | TEMPORARY;
```

Starting Up an Instance

```
STARTUP [FORCE] [NOMOUNT|MOUNT|OPEN]
        [Oracle_sid] [PFILE=name] [RESTRICT]
```

Shutting Down an Instance

```
SHUTDOWN [NORMAL|IMMEDIATE|TRANSACTIONAL|ABORT]
```

Creating a User from the Command Line with Various Clauses

```
CREATE USER username IDENTIFIED BY passwordname
    [DEFAULT TABLESPACE defaulttablespacename]
    [TEMPORARY TABLESPACE temporarytablespacename]
    [QUOTA storagespace ON defaulttablespacename]
    [PROFILE profilename];
```

Dropping a User

```
DROP USER username [CASCADE];
```

Logging into SQL*PLUS from the Command Line

```
SQLPLUS [username[|password]] [@hostname] [@script] [parameter list]
```

Appendix C

Reference to SQL*Plus Commands

Allowable command abbreviations are underlined.

ARRAYSIZE. The ARRAYSIZE command sets the number of rows fetched from the database. The valid values are 1 to 5000. The general syntax is

SET ARRAYSIZE n

AUTOCOMMIT. The AUTOCOMMIT command sets the automatic commit of a DML statement to On or OFF. The general syntax is

SET AUTOCOMMIT ON | OFF | IMMEDIATE

BREAK. The BREAK command specifies action based on a change of a value of a column or an expression, or when a row is returned. The general syntax is

BREAK [ON column | expression | ROW | REPORT [action list]] . . .

The actions are SKIP *n* lines, SKIP PAGE, NODUPLICATES, or DUPLICATES.

BTITLE. The BTITLE command specifies the format of the title at the bottom of each page. The general syntax is

BTITLE [ON | OFF] [printspec [text|variable] . . .]

where ON|OFF turns the title on or off. The printspecs are COL *n*, SKIP *n*, LEFT, CENTER, RIGHT, and BOLD. The text is a character string to be printed, and the variable is a user-defined or system variable.

CLEAR BUFFER. The CLEAR BUFFER command clears an entire SQL statement from the buffer. The general syntax is

CLEAR BUFFER

CLEAR COLUMNS. The CLEAR COLUMNS command clears all column formatting set during the current session. The general syntax is

CLEAR COLUMNS

CLEAR SCREEN. The CLEAR SCREEN command clears the entire screen. The general syntax is

CLEAR SCREEN

COLSEP. The COLSEP command sets the text printed between columns retrieved by a SELECT statement. The default text is a single space. The general syntax is

SET COLSEP text

COLUMN. The COLUMN command is used to display, suppress, reset, or set column attributes as well as to set column headings. The general syntax/use is

COLUMN	**(shows display attributes for all columns)**
COLUMN columnname	**(shows display attributes for one column)**
COLUMN columnname CLEAR	**(resets display attributes of column)**
COLUMN columnname FORMAT formattype	**(sets display attribute of column)**
COLUMN columnname HEADING columnheading	**(sets column heading)**

COMPUTE. The COMPUTE command performs calculations using standard mathematical functions and displays the summary lines. The general syntax is

COMPUTE [function . . . OF column|alias|expression . . .
ON column|alias|expression|REPORT|ROW]

where the functions are SUM, AVG, COUNT, MAXIMUM, MINIMUM, NUMBER, STD, and VARIANCE. The ON clause must match the BREAK statement.

CONNECT. The CONNECT command connects a user to the database. The general syntax is

```
CONNECT username [|password [@hostname]]
```

COPY. The COPY command copies a result from table to table or allows you to append or to create rows or to create new tables. The general syntax is

```
COPY FROM <DB> TO <DB> <OPT> <TABLE> { (<COLS>) } USING <SEL>
```

where

```
<db>    = database string (e.g., scott/tiger@d:nshah-monroe)
<opt>   = key word APPEND, CREATE, INSERT or REPLACE
<table> = name of the target table
<cols>  = a list of target column aliases separated by commas
<sel>   = any valid SELECT statement
```

DEFINE. The DEFINE command defines a variable and stores it with CHAR data type. If a variable name is not supplied, it shows all previously defined variables. The general syntax is

```
DEFINE[variablename = literal]
```

To turn use of a substitution character on or off, use

```
SET DEFINE ON | OFF
```

DESCRIBE. The DESCRIBE command describes the structure of a table or view with column names, data types, and lengths. The general syntax is

```
DESCRIBE table_or_viewname
```

DISCONNECT. The DISCONNECT command commits the current transaction and disconnects a user from the database, but it does not exit from SQL*Plus. The general syntax is

```
DISCONNECT
```

ECHO. The ECHO command controls the display of a command as it is executed. ON lists the display; OFF suppresses the display. The general syntax is

SET ECHO ON | OFF

EXECUTE. The EXECUTE command executes a PL/SQL block/statement. The general syntax is

EXECUTE statement | blockname

EXIT. The EXIT command commits the current DML transaction, disconnects the user from the database, and closes the SQL*Plus session. The general syntax is

EXIT

FEEDBACK. The FEEDBACK command displays the number of rows returned by a query when the query returns at least *n* rows. It can also suppress the display with the OFF switch. The default value for *n* is 6. The general syntax is

SET FEEDBACK n | OFF | ON

HELP. The HELP command starts the SQL*Plus help function and displays help on a specified topic; otherwise, it displays a list of topics. The general syntax is

HELP [topic]

HOST. The HOST command executes an operating system command from SQL*Plus. If the command is not specified, the system prompt is displayed. You can return back to SQL*Plus by typing EXIT. The general syntax is

HOST [command]

LINESIZE. The LINESIZE command is used to set the total number of characters displayed by SQL*Plus per line before wrapping. The default line size is 80. The general syntax is

SET LINESIZE n

NUMWIDTH. The NUMWIDTH command sets the width for displaying numbers. The default is nine. The general syntax is

SET NUMWIDTH n

PAGESIZE. The PAGESIZE command sets the number of lines per page. The default is 24. You can change it to zero to suppress all titles, headings, and page

breaks. The general syntax is

SET PAGESIZE n

PAUSE. The PAUSE command displays a blank line followed by a line with specified text, then waits for the user to press the Enter key. The general syntax is

SET PAUSE OFF | ON | text

REMARK. The characters followed by the REMARK keyword on the same line are treated as a comment that is ignored by SQL*Plus. The general syntax is

REMARK [text]

SET. The SET command shows and sets system and environment variables. The general syntax is

SET variablename option | value

SHOW. The SHOW command shows the values of all system variables or the current user's name. The general syntax is

SHOW ALL | USER

SHOWMODE. The SHOWMODE command controls the display of old and new settings when a variable setting is changed with SET. The general syntax is

***SET SHOWMODE ON* | OFF**

SHUTDOWN. The SHUTDOWN command shuts down the Oracle database instance. The general syntax is

SHUTDOWN

SPOOL. The SPOOL command with a filename starts the spooling of statements and results into that file. The default file extension is *lst*. The OFF switch stops writing to the file and closes it. The general syntax is

SPOOL filename[.ext] | OFF | OUT

SQLCASE. The SQLCASE command converts the case of SQL statements, PL/SQL statements, and text, including text in quotation marks. The general syntax is

SET SQLCASE UPPER | LOWER | MIXED

SQLPROMPT. The SQLPROMPT command can be used to set the SQL*Plus prompt. The default prompt is SQL >. The general syntax is

SET SQLPROMPT text

TIME. The TIME command, when set to ON, shows current time before the SQL> prompt. The general syntax is

SET TIME ON | OFF

TIMING. The TIMING command, when turned ON, shows timing statistics. The general syntax is

SET TIMING ON | OFF

TTITLE. The TTITLE command specifies the format of the title, which is displayed at the top of each page. The general syntax is

TTITLE [ON|OFF] [printspec [text|variable] . . .]

UNDEFINE. The UNDEFINE command deletes a variable defined with DEFINE or & or &&. The general syntax is

UNDEFINE variablename

VERIFY. The VERIFY command, when turned ON, shows an SQL statement before and after SQL*Plus replaces substitution variables with values. The general syntax is

SET VERIFY ON | OFF

WRAP. The WRAP command controls the truncation of a row if it is longer than the line width. The general syntax is

SET WRAP ON | OFF

SQL*PLUS EDITING COMMANDS

Command	Description
APPEND *text*	Adds text to the end of the current line.
CHANGE / *old* / *new*	Changes old text to new text in the current line.
CHANGE / *text* /	Deletes text from the current line.
CLEAR BUFFER	Deletes all lines from the SQL buffer.
DEL	Deletes the current line.
DEL *n*	Deletes line number *n*.
DEL *m n*	Deletes lines *m* through *n*.
INPUT	Inserts an indefinite number of lines.
INPUT *text*	Inserts a line of text.
LIST	Lists all lines from the SQL buffer.
LIST *n*	Lists line number *n*.
LIST *m n*	Lists lines from *m* through *n*.
RUN	Displays and runs an SQL statement from buffer.
N	Makes line *N* current.
n text	Replaces line *n* with text.
0 *text*	Inserts a line before line 1.
CLEAR SCREEN	Clears the screen.

SQL*PLUS FILE-RELATED COMMANDS

Command	Description
GET *filename* [.ext]	Writes a previously saved file to the buffer. The default extension is *sql*. Writes SQL statements, not SQL*Plus commands.
START *filename* [.ext]	Runs a previously saved command from the file.
@*filename*	Same as START (default extension must be *.sql*).
EDIT	Invokes the default editor (e.g., Notepad), and saves the buffer contents in a file called *afiedt.buf*.
EDIT [*filename* [.ext]]	Invokes the editor with the command from a saved file.
SAVE *filename* [.ext] REPLACE | APPEND	Saves the current buffer contents to a file with the option to replace or append.
SPOOL [*filename* [.ext] | OFF | OUT]	Stores query results in a file. OFF closes the file, and OUT sends file to the system printer.
EXIT	Leaves the SQL*Plus environment.

Note: The filename in the file-related commands also requires a file path.

Appendix D

Object Orientation

AN OBJECT

An object in Oracle9i is a reusable component that represents real-word things. An object is defined with a user-defined data type called an object type. Object types are used as data types to define columns (known as object columns) in a table. Object types are also used in place of a list of columns for an object table. An object type can be used as an element in another object type as well.

An object contains a name, attribute(s), and methods. An object contains data and information about what can be done with the data. An object may also contain another object. The methods are procedures and functions written in an Oracle9i-supported languages. Each method has a name as well as the name of the object that contains that method. The method can be passed data through parameters from a calling program.

Oracle9i is a relational database as well as an object-oriented database. It provides several ways to connect relational tables and objects:

- ***Object view***—An object view is like a relational view. It does not contain any data, but it is based on underlying tables. It allows users to view a relational table with an object orientation. You can modify data in the underlying table with SQL statements, with object methods, or by using an object view.

- ***Object table***—An object table is a table that is described by object type, not by attribute names. The elements in an object type define the data for the object table. The object table can also contain object methods as part of a table's definition. These methods are used to perform data manipulation on the object table. You can define a primary key for an object table and create an index for it as well.
- ***Relational table with object column***—A relational table may contain one or more columns with the object type as their data type. Such a table is also known as a hybrid table, because it contains columns with scalar data types as well as object columns.

An object reference (key word REF) is a special data type in a table that facilitates a foreign key in the table. It establishes a relationship between two objects—for example, an object table called SOFTWARE_COMPANY_OBJ with a primary key and a second object table, SOFTWARE_OBJ, that has a foreign key column that connects software to the company that makes it. This foreign key column is defined with the data type REF and references the object named SOFTWARE_COMPANY_OBJ.

Let us take examples of object types, object tables, REF columns, and tables with object type and look at SQL statements on these objects (see Fig. D-1).

Name	Item Type	Attribute Name	Data Type
NAME_TYPE	Object type	LAST_NAME	VARCHAR2
		FIRST_NAME	VARCHAR2
FULLNAME_TYPE	Object type	NAME_REF	REF to *NAME_TYPE*
		MID_INITIAL	VARCHAR2
NAME_TABLE	Object table		Row of *FULLNAME_TYPE*
STUDENT_TABLE	Hybrid table	STUDENT_ID	NUMBER
		FULL_NAME	*FULLNAME_TYPE*
		PHONE_NUM	VARCHAR2

Figure D-1 Object types, REF column, object column, and hybrid table.

SQL QUERIES FOR OBJECTS

Retrieving Data from an Object Table

In Figure D-1, the object table NAME_TABLE contains rows with the object type FULLNAME_TYPE, which in turn has two attributes of type REF (NAME_REF) and VARCHAR2 (MID_INITIAL). The REF to NAME_TYPE in turn contains two attributes, LAST_NAME and FIRST_NAME, of type VARCHAR2. You can write a standard SQL query to display the contents of the object table NAME_TABLE. For example,

```
SELECT * FROM NAME_TABLE;
```

The retrieved rows are displayed with column headings as follows:

```
LAST_NAME FIRST_NAME MID_INITIAL
---------- ---------- ----------
```

You can use WHERE, ORDER BY, and GROUP BY clauses with the SELECT statement, just as in standard relational SQL.

Now, let us display the last name, first name, and phone number of students from STUDENT_TABLE. The attributes belonging to the object type are referenced using dot notation:

tablealias.objecttype.attribute

For example,

```
SELECT S.FULL_NAME.LAST_NAME, S.FULL_NAME.FIRST_NAME,
PHONE_NUM FROM STUDENT_TABLE S;
```

When you use a column related to an object, you must use a table alias. In the example above, STUDENT_TABLE has the alias S. If you do not use a table alias, you get the following error message:

```
ORA-00904: invalid column name
```

Inserting a Row into an Object Table

The value for an object is inserted by entering the name of the object and then enclosing all values for the object's attributes in parentheses. For example,

```
INSERT INTO STUDENT_TABLE VALUES
(543, FULL_NAME('Spencer', 'Karen', 'A'), '732-555-6789');
```

where FULL_NAME('Spencer', 'Karen', 'A') contains values for three attributes in the object FULL_NAME, which are LAST_NAME, FIRST_NAME, and MID_INITIAL.

Updating an Object

Suppose you want to change a student's last name. Use the UPDATE statement with an alias for the table name, and qualify the attribute with the table name and the object name. For example,

```
UPDATE STUDENT_TABLE S
SET S.FULL_NAME.LAST_NAME = 'Martinez'
WHERE STUDENT_ID = 543;
```

Deleting Rows from an Object Table

```
DELETE FROM NAME_TABLE N
WHERE LAST_NAME = 'Smith';
```

Appendix E

What's New in Oracle9i SQL and PL/SQL?

This appendix describes new features of SQL and PL/SQL in Oracle9i (release 1).

NEW FEATURES IN SQL

1. **New or modified built-in data types:**

   CHAR—can take CHAR or BYTE parameter.
   VARCHAR2—can take CHAR or BYTE parameter.
   TIMESTAMP—for additional datetime functionality.
   INTERVAL YEAR TO MONTH—for additional datetime functionality.
   INTERVAL DAY TO SECOND—for additional datetime functionality.

2. **New or enhanced expressions:**

   CASE expressions—enhanced with Searched Case expression.
   CURSOR expressions—enhanced to pass as REF CURSOR arguments to function.
   DATETIME expressions—new.
   INTERVAL expressions—new.
   Scalar subquery expressions—new.

3. **New built-in functions:**

ASCIISTR	BIN_TO_NUM	COALESCE
COMPOSE	CURRENT_DATE	CURRENT_TIMESTAMP
DBTIMEZONE	DECOMPOSE	EXISTSNODE
EXTRACT(*datetime*)	EXTRACT (XML)	FIRST
FROM_TZ	GROUP_ID	GROUPING_ID
GROUPING_ID	LAST	LOCAL_TIMESTAMP
NULLIF	ROWTONHEX	ROWIDTONCHAR
SESSIONTIMEZONE	SYS_CONNECT_BY_PATH	SYSTIMESTAMP
TO_CHAR(*character*)	TO_CLOB	TO_DSINTERVAL
TO_NCHAR(*character*)	TO_NCHAR(*datetime*)	TO_NCHAR(*number*)
TO_NCLOB	TO_TIMESTAMP	TO_TIMESTAMP_TZ
TO_YMINTERVAL	TREAT	TZ_OFFSET
UNISTR	WIDTH_BUCKET	

4. **Enhanced functions:**

   INSTR.
   LENGTH.
   SUBSTR.

5. **New system and object privileges:**

   EXEMPT ACCESS POLICY.
   RESUMABLE.
   SELECT ANY DICTIONARY.
   UNDER ANY TYPE.
   UNDER ANY NEW.
   UNDER(*object privilege*).

6. **New SQL statements:**

   CREATE PFILE.
   CREATE SPFILE.
   MERGE.

7. **SQL statements with new syntax:**

   ALTER DATABASE—new syntax to end hot backup while database is mounted; also for standby databases.
   ALTER INDEX—to get statistics on index usage.
   ALTER OUTLINE—for modifications to public and private outlines.

ALTER ROLE—to identify role using application-specified package.

ALTER SESSION—to specify if statements issued during a session can be suspended.

ALTER SYSTEM—extended SET clause.

ALTER TABLE—allows partitioning by specified values.

ALTER TYPE—to change attribute or method definition of an object type.

ALTER VIEW—to add constraints to views.

ANALYZE—new ONLINE and OFFLINE clauses, and selection of standard or user-defined (or both) statistics.

CONSTRAINT_CLAUSE—for index handling when dropping or disabling constraints.

CREATE CONTEXT—to initialize the context from the LDAP directory or an OCI interface and to make context accessible throughout an instance.

CREATE CONTROLFILE—for creation of Oracle-managed files.

CREATE DATABASE—to create default temporary tablespaces and to undo tablespaces.

CREATE FUNCTION—to create pipelined and parallel table functions and user-defined aggregate functions.

CREATE OUTLINE—for creation of private and public outlines.

CREATE ROLE—to identify a role using an application-specified package.

CREATE TABLE—allows creation of external tables, creation of Oracle-managed files, and partitioning by a list of values.

CREATE TABLESPACE—for segment space management, creation of Oracle-managed files, and creation of undo tablespaces.

CREATE TEMPORARY TABLESPACE—for creation of Oracle-managed files.

CREATE TYPE—allows creation of subtypes.

CREATE VIEW—lets you create subviews of object views and define constraints on views.

DROP TABLESPACE—lets you drop operating system files when contents are dropped from a tablespace.

FILESPEC—for creation of Oracle-managed files.

INSERT—lets you insert default column values.

SELECT—lets you specify multiple grouping in GROUP BY clause, assign names to subquery blocks, and support ANSI-compliant join syntax.

SET TRANSACTION—to specify a name for a transaction.

UPDATE—to update default column values.

NEW FEATURES IN PL/SQL

1. **Integration of SQL and PL/SQL:**

   PL/SQL supports the complete range of syntax for SQL statements.
   No error messages for valid SQL syntax as in previous versions.
   Compile time error-checking.

2. **CASE expressions:**

   New CASE statements as alternatives for IF statements.

3. **New date/time types:**

   TIMESTAMP data type that records time in fractional seconds.
   TIMESTAMP WITH TIME ZONE and TIMESTAMP WITH LOCAL TIME ZONE for adjusting account for time zone differences (with daylight savings).
   INTERVAL DAY TO SECOND and INTERVAL YEAR TO MONTH for representing differences between two date and time values.

4. **Type evolution:**

   Attributes and methods can be dropped from an object type without recreating the type and corresponding data.

5. **Native compilation of PL/SQL code:**

   Improved performance by compiling Oracle-supplied and user-created stored procedures into native executables with C-language development tools.

6. **Table functions and cursor expressions:**

   You can query a set of returned rows like a table.
   Result sets can be passed from one function to another.

7. **Multilevel collections:**

   Nesting of collection types—for example, VARRAY of VARRAYS and PL/SQL table of PL/SQL table.

8. **LOB data types:**

   You can operate on LOB types like other similar types.
   Character functions for CLOB and NCLOB.
   Treat BLOB as RAWS.
   Conversion from LOB to other types like LONG made simple.

9. **MERGE statement:**

   A statement that combines INSERT and UPDATE into a single operation.

10. **Bulk operations:**

    You can perform bulk SQL operations using native dynamic SQL—for example, EXECUTE IMMEDIATE statement.

    Perform bulk INSERT and UPDATE operations and, in case of errors on some rows, continue and examine errors after the operation is complete.

Appendix F

Additional References

In this book, an attempt is made to provide an in-depth understanding of relational database concepts, Oracle's nonprocedural language SQL, and the procedural language PL/SQL. Both language features apply to Oracle8, Oracle8i, and Oracle9i, although Oracle9i has some added features. The author has examined various resources to provide an adequate amount of knowledge to the readers while staying within the scope of this book. This appendix lists additional sources of information available for further reference of the topics covered in this book.

WEB SITES

1. **Oracle Corporation—*http://www.oracle.com***

   Home page of Oracle Corporation.

2. **Oracle Magazine—*http://www.oramag.com***

   For a free subscription to the bimonthly *Oracle Magazine*.

3. **Oracle Technology Network (OTN)—*http://otn.oracle.com***

   Oracle's network to reach users—low-cost offers, free downloads, and so on.

4. **Oracle University—*http://education.oracle.com***

Information about Oracle training tracks, schedules, online registration, and certifications.

5. **Oracle User's Group—*http://www.oug.com***

   The brainstorm of a number of dedicated user-group officers.

6. **OracleZone—*http://www.oraclezone.com***

   A wide variety of Oracle information, real-life problems, discussions, and troubleshooting tips posted by Oracle users.

7. **Orapub—*http://orapub.com***

   A site founded by a former Oracle employee, Craig Shallahamer, devoted to all Oracle-related issues.

8. **My Oracle—*http://my.oracle.com***

   A portal for creating a personalized home page for U.S. news, BBC news, CNET news, stock market quotes, and other goodies.

BOOKS AND OTHER PUBLISHED MATERIAL

1. **Relational database concepts:**

   Kroenke, David, *Database Processing: Fundamentals, Design, and Implementation*, Prentice-Hall, January 2002.

   Rob, Peter, and Treyton Williams, *Database Design and Application Development*, McGraw-Hill, Primis Custom Publishing.

2. **Oracle SQL:**

   Mishra, Sanjay, and Alan Beaulieu, *Mastering Oracle SQL*, O'Reilly and Associates, April 2002.

   Rischert, Alice, *Oracle SQL Interactive Workbook 2/E*, Prentice-Hall, December 2002.

3. **Oracle PL/SQL:**

   Feuerstein, Steven, *Oracle PL/SQL Programming 3/E*, O'Reilly and Associates, September 2002.

   Rosenzweig, Benjamin and Elena Silvestrova, *Oracle PL/SQL Interactive Workbook 2/E*, Prentice-Hall, October 2002.

   Urman, Scott, *Oracle9i PL/SQL Programming*, McGraw-Hill Osborne, November 2001.

4. **Miscellaneous topics:**

   Loney, Kevin, and Marlene Theriault, *Oracle9i DBA Handbook*, McGraw-Hill Osborne, November 2001.

   Price, Jason, *Oracle9i JDBC Programming*, McGraw-Hill Osborne, May 2002.

5. **General reference:**

   Staron, Richard J., *Guerrilla Oracle*, Addison Wesley, April 2003.

Index