Oracle9*i*: SQL for End Users

Student Guide

40060GC10
Production 1.0
November 2001
D34060

ORACLE®

Author

Priya Nathan

Technical Contributors and Reviewers

Anthony Nicoletti
Christopher Lawless
Christina Yetman
Helen Robertson
Ligia Robayo
Claire Bennett
Vasily Strelnikov
Andrew Brannigan
Jonathan Grove
Mozhe Jalali
Janet Stern
Chris Nguyen
Mehrdad Soltani

Publisher

Shane Mattimoe

Contents

Preface

Profile

Before You Begin This Course

Before you begin this course, you should be able to use a graphical user interface (GUI). Required prerequisites are familiarity with data processing concepts and techniques.

How This Course Is Organized

Oracle9i: SQL For End Users is an instructor-led course featuring lectures and hands-on exercises. Online demonstrations and written practice sessions reinforce the concepts and skills introduced.

Related Publications

Oracle Publications

Title	Part Number
Oracle9i Reference, Release 1 (9.0.1)	A90190-02
Oracle9i SQL Reference, Release 1 (9.0.1)	A90125-01
Oracle9i Concepts, Release 1 (9.0.0)	A88856-02
Oracle9i Server Application Developer's Guide Fundamentals	
Release 1 (9.0.1)	A88876-02
*iSQL*Plus User's Guide and Reference, Release 9.0.0*	A88826-01
*SQL*Plus User's Guide and Reference, Release 9.0.1*	A88827-02

Additional Publications

- System release bulletins
- Installation and user's guides
- *read.me* files
- International Oracle User's Group (IOUG) articles
- *Oracle Magazine*

Typographic Conventions

What follows are two lists of typographical conventions used specifically within text or within code.

Typographic Conventions Within Text

Convention	Object or Term	Example
Uppercase	Commands, functions, column names, table names, PL/SQL objects, schemas	Use the `SELECT` command to view information stored in the `LAST_NAME` column of the `EMPLOYEES` table.
Lowercase, italic	Filenames, syntax variables, usernames, passwords	**where:** *role* is the name of the role to be created.
Initial cap	Trigger and button names	Assign a When-Validate-Item trigger to the ORD block. Choose Cancel.
Italic	Books, names of courses and manuals, and emphasized words or phrases	For more information on the subject see *Oracle Server SQL Language Reference Manual* Do *not* save changes to the database.
Quotation marks	Lesson module titles referenced within a course	This subject is covered in Lesson 3, “Working with Objects.”

Typographic Conventions (continued)

Typographic Conventions Within Code

Convention	Object or Term	Example
Uppercase	Commands, functions	**SELECT** **employee_id** **FROM employees;**
Lowercase, italic	Syntax variables	**CREATE ROLE *role*;**
Initial cap	Forms triggers	**Form module: ORD** **Trigger level: S_ITEM.QUANTITY item** **Trigger name: When-Validate-Item** **. . .**
Lowercase	Column names, table names, filenames, PL/SQL objects	**. . .** **OG_ACTIVATE_LAYER** **(OG_GET_LAYER ('prod_pie_layer'))** **. . .** **SELECT last_name** **FROM employees;**
Bold	Text that must be entered by a user	**CREATE USER scott** **IDENTIFIED BY tiger;**

I

Relational Principles and Oracle Concepts

Objectives

After completing this lesson, you should be able to do the following:

- Describe the system life cycle development phases
- Discuss the theoretical and physical aspects of a relational database
- Describe the Oracle implementation of the RDBMS and ORDBMS
- Describe the Internet Platform Architecture
- Describe new features of Oracle9*i*
- Describe how SQL is used in the Oracle product set

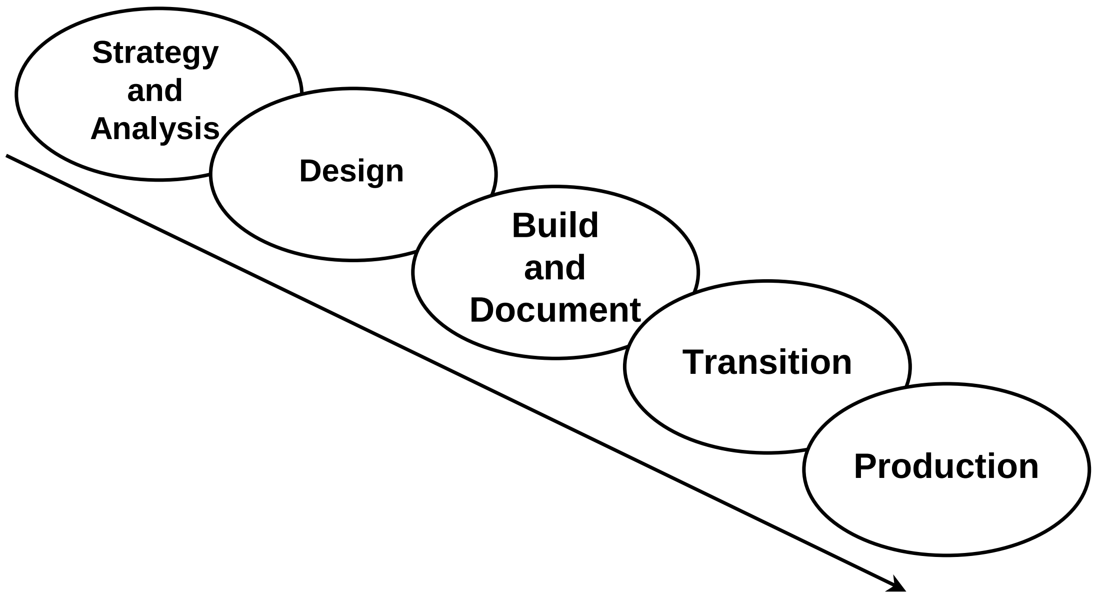
ORACLE

I-2

Lesson Aim

In this lesson, you will gain an understanding of the relational database management system (RDBMS), the object relational database management system (ORDBMS) and the new features of Oracle9*i*.

System Development Life Cycle



I-3

System Development Life Cycle

From concept to production, you can develop a database by following the system development life cycle, which has multiple stages of development. This top-down, systematic approach to database development transforms business information requirements into an operational database.

Strategy and Analysis

- Study and analyze the business requirements. Interview users and managers to identify the information requirements. Incorporate the enterprise and application mission statements as well as any future system specifications.
- Build models of the system. Transfer the business narrative into a graphical representation of business information needs and rules. Confirm and refine the model with the analysts and experts.

Design

Design the database based on the model developed in the strategy and analysis phase.

Build and Document

- Build the prototype system. Write and execute the commands to create the tables and supporting objects for the database.
- Develop user documentation, Help text, and operation manuals to support the use and operation of the system.

System Development Life Cycle (continued)

Transition

Refine the prototype. Move an application into production with user-acceptance testing, conversion of existing data, and parallel operations. Make required modifications.

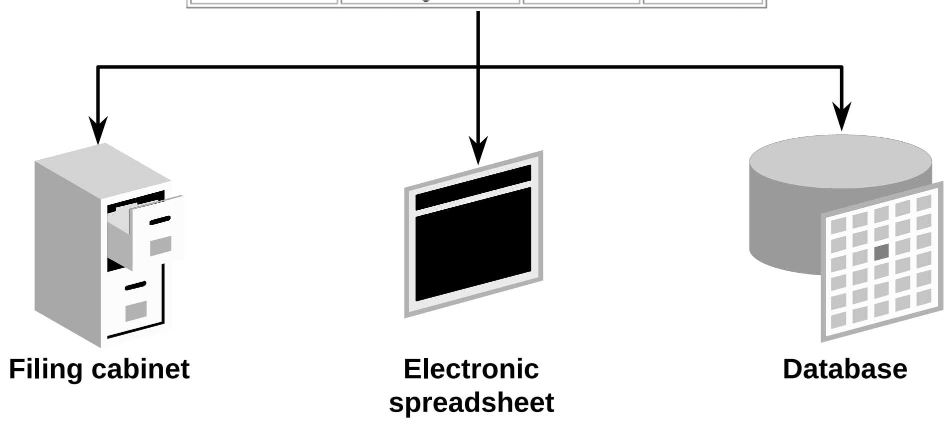
Production

Roll out the system to the users. Operate the production system. Monitor its performance, and enhance and refine the system.

Note: The various phases of the system development life cycle can be carried out iteratively. This course focuses on the build phase of the system development life cycle.

Data Storage on Different Media

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700



I-5

Storing Information

Every organization has some information needs. A library keeps a list of members, books, due dates, and fines. A company needs to save information about employees, departments, and salaries. These pieces of information are called data.

Organizations can store data on various media and in different formats: for example, a hard-copy document in a filing cabinet or data stored in electronic spreadsheets or databases.

A database is an organized collection of information.

To manage databases, you need database management systems (DBMS). A DBMS is a program that stores, retrieves, and modifies data in the database on request. There are four main types of databases: hierarchical, network, relational, and object-oriented. Object relational databases are a hybrid of object-oriented databases.

Note: Oracle7 is a relational database management system whereas Oracle8, 8*i*, and 9*i* are object relational database management systems.

Relational Database Concepts

- **Dr. E. F. Codd proposed the relational model for database systems in 1970.**
- **The relational model consists of the following:**
  - **Collection of objects or relations**
  - **Set of operators to act on the relations**
  - **Data integrity for accuracy and consistency**

ORACLE

I-6

Relational Database Concepts

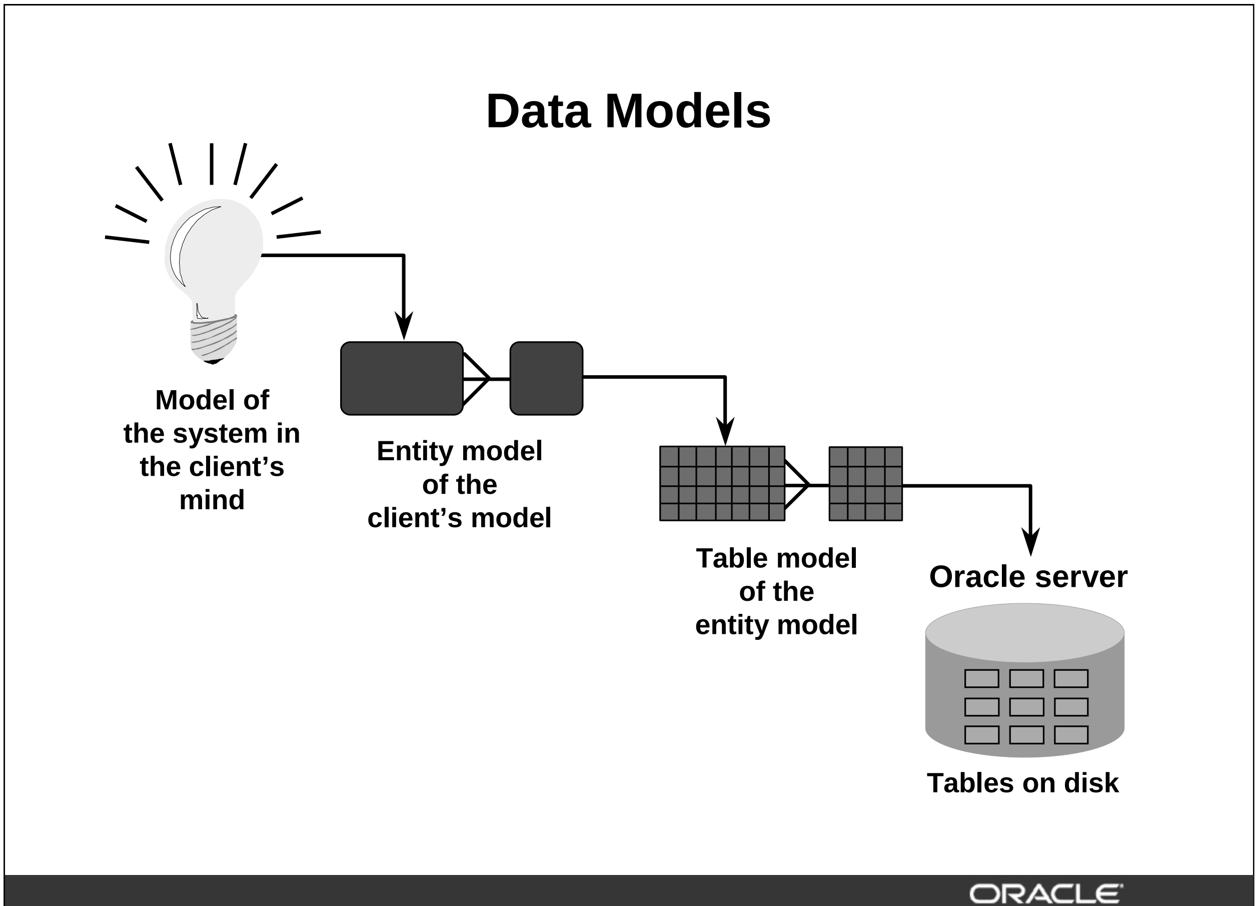
The principles of the relational model were first outlined by Dr. E. F. Codd in a June 1970 paper called "A Relational Model of Data for Large Shared Data Banks." In this paper, Dr. Codd proposed the relational model for database systems.

The more popular models used at that time were hierarchical and network, through simple flat file data structures. Relational database management systems (RDBMS) soon became very popular, especially for their ease of use and flexibility in structure. In addition, innovative vendors, such as Oracle, supplemented the RDBMS with a suite of powerful application development and user products, providing a total solution.

Components of the Relational Model

- Collections of objects or relations that store the data
- A set of operators that can act on the relations to produce other relations
- Data integrity for accuracy and consistency

For more information, see E. F. Codd, *The Relational Model for Database Management.*



I-7

Data Models

Models are a cornerstone of design. Engineers build a model of a car to work out any details before putting it into production. In the same manner, system designers develop models to explore ideas and improve the understanding of the database design.

Purpose of Models

Models help communicate the concepts in people's minds. They can be used to do the following:

- Communicate
- Categorize
- Describe
- Specify
- Investigate
- Evolve
- Analyze
- Imitate

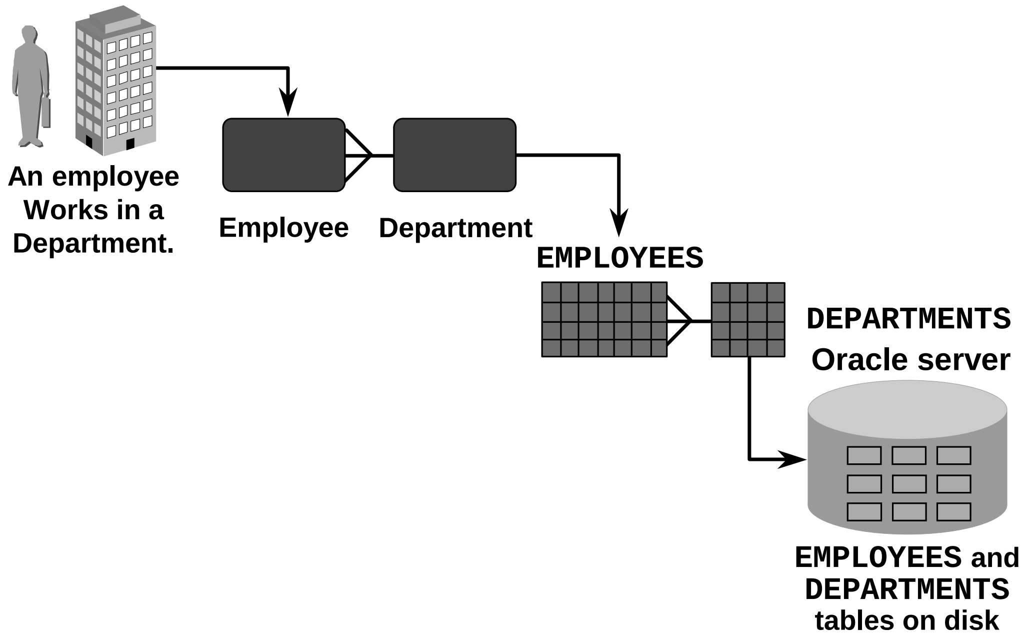
The objective is to produce a model that fits a multitude of these uses, can be understood by an end user, and contains sufficient detail for a developer to build a database system.

Data Models (Continued)

Using some portion of the system as a basis, the developer draws up a blueprint of the system. This blueprint is a major deliverable that will validate the design standards and analysis. Through blueprints, users can evaluate the ability of systems to meet their needs. The blueprint is the entity model of the system in the client’s mind. Every detail should be laid out. A lot of time is spent in this phase so that the developer has a clear picture of the system before moving on to the next phase.

The next phase is the translation of the entity model to the table model. This phase involves the design of the tables and columns along with the detailed specification of domains and check constraints on the columns. The above database design translates to the actual tables and other database objects on the Oracle server.

The Employee and Department Data Model



Creating the Employee and Department Data Model

Consider a system that involves employees and departments. An organization has a number of employees and a number of departments that these employees work in. An employee works for a single department, while a department can have many employees working in it. The steps to create the employee and department data model is given below:

1. Create a mental model of the system. In this case, each employee works in a department, so the system must store information about employees and departments entities.
2. Using the system model arrived at in step 1, create models of the employee and department. Also, create a model of the "each employee works in a department" relationship that exists between the employee and department entities.
3. Using the employee and department table models arrived at in step 2, design the `EMPLOYEES` and `DEPARTMENTS` tables.
4. Using the table models arrived at in step 3, create the `EMPLOYEES` and `DEPARTMENTS` tables on the Oracle Server and create the relationship between the two tables.

The Relationship Between the `EMPLOYEES` and `DEPARTMENTS` Tables

All employees must be assigned to a department. This means that every row in the `EMPLOYEES` table must reference a row in the `DEPARTMENTS` table. However, there may be departments that do not have employees assigned to them yet.

Definition of a Relational Database

A relational database is a collection of relations or two-dimensional tables.

Oracle server

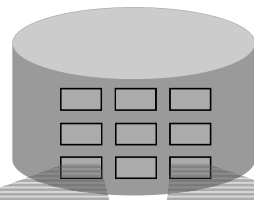


Table Name: **`EMPLOYEES`**

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL
100	Steven	King	SKING
101	Neena	Kochhar	NKOCHHAR
102	Lex	De Haan	LDEHAAN
103	Alexander	Hunold	AHUNOLD
104	Bruce	Ernst	BERNST

...

Table Name: **`DEPARTMENTS`**

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500

...

I-10

Definition of a Relational Database

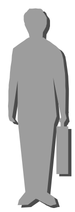
A relational database uses relations or two-dimensional tables to store information.

For example, you might want to store information about all the employees in your company. In a relational database, you store different pieces of information about your employees, such as employee information, a department information, and a salary information.

.

The EMPLOYEES Table

The EMPLOYEES table stores employee information



EMPLOYEE_ID: 102

LAST_NAME: De Haan

EMAIL: LDEHAAN

Table Name: EMPLOYEES

EMPLOYEE_ID	LAST_NAME	EMAIL
100	King	SKING
101	Kochhar	NKOCHHAR
102	De Haan	LDEHAAN
103	Hunold	AHUNOLD

ORACLE

I-11

The EMPLOYEES Table

Each row in the EMPLOYEES table stores information about one employee. Each column stores a particular piece of information about that employee.

For example, the third row in the EMPLOYEES table gives the following information:

Employee’s employee number:	EMPLOYEE_ID = 102
Employee’s last name:	LAST_NAME = De Haan
Employee’s email id:	EMAIL = LDEHAAN

Relational Database Terminology

EMPLOYEE_ID	LAST_NAME	SALARY	COMMISSION_PCT	DEPARTMENT_ID
100	King	24000		90
101	Kochhar	17000		90
102	De Haan	17000		90
103	Hunold	9000		60
104	Ernst	6000		60
107	Lorentz	4200		60
124	Mourgos	5800		50
141	Rajs	3500		50
142	Davies	3100		50
143	Matos	2600		50
144	Vargas	2500		50
149	Zlotkey	10500	.2	80
174	Abel	11000	.3	80
176	Taylor	8600	.2	80
178	Grant	7000	.15	
200	Whalen	4400		10
201	Hartstein	13000		20
202	Fay	6000		20
205	Higgins	12000		110
206	Gietz	8300		110

ORACLE

I-12

Terminology Used in a Relational Database

A relational database can contain one or more tables. A table is the basic storage structure of an RDBMS. A table holds all the data necessary about something in the real world—for example, employees, invoices, or customers.

The slide shows the contents of the `EMPLOYEES` table or relation. The numbers indicate the following:

1. A single row or *tuple* representing all data required for a particular employee. Each row in a table should be identified by a primary key, which allows no duplicate rows. The order of rows is insignificant; specify the row order when the data is retrieved.
2. A column or attribute containing the employee number. The employee ID identifies a unique employee in the `EMPLOYEES` table. In this example, the employee ID column is designated as the primary key. A primary key must contain a value and the value must be unique.
3. A column that is not a key value. A column represents one kind of data in a table; in the example, the salary of all the employees. Column order is insignificant when storing data; specify the column order when the data is to be retrieved.
4. A field can be found at the intersection of a row and a column. There can be only one value in it.

Terminology Used in a Relational Database (Continued)

5. A column containing the department number, which is also a foreign key. A foreign key is a column (or collection of columns) that defines how tables relate to each other. A foreign key refers to a primary key or a unique key in the same table or in another table. In the example, `DEPARTMENT_ID` uniquely identifies a department in the `DEPARTMENTS` table.
6. A field may have no value in it. This is called a null value. In the `EMPLOYEES` table, the `DEPARTMENT_ID` for the employee Grant is `NULL`.

Relating Multiple Tables

- **Each row of data in a table is uniquely identified by a primary key (PK).**
- **You can logically relate data from multiple tables using foreign keys (FK).**

Table Name: **EMPLOYEES**

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
102	Lex	De Haan	90
104	Bruce	Ernst	60
142	Curtis	Davies	50
174	Ellen	Abel	80
202	Pat	Fay	20
206	William	Gietz	110

...

Primary key (EMPLOYEE_ID) **Foreign key** (DEPARTMENT_ID)

Table Name: **DEPARTMENTS**

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

Primary key

I-14

Relating Multiple Tables

Each table contains data that describes exactly one entity. For example, the EMPLOYEES table contains information about employees. Categories of data are listed across the top of each table, and individual cases are listed below. Using a table format, you can readily visualize, understand, and use information.

Because data about different entities is stored in different tables, you may need to combine two or more tables to answer a particular question. For example, you may want to know the location of the department where an employee works. In this scenario, you need information from the EMPLOYEES table (which contains data about employees) and the DEPARTMENTS table (which contains information about departments). An RDBMS enables you to relate the data in one table to the data in another by using foreign keys. A foreign key is a column or a set of columns that refer to a primary key in the same table or another table.

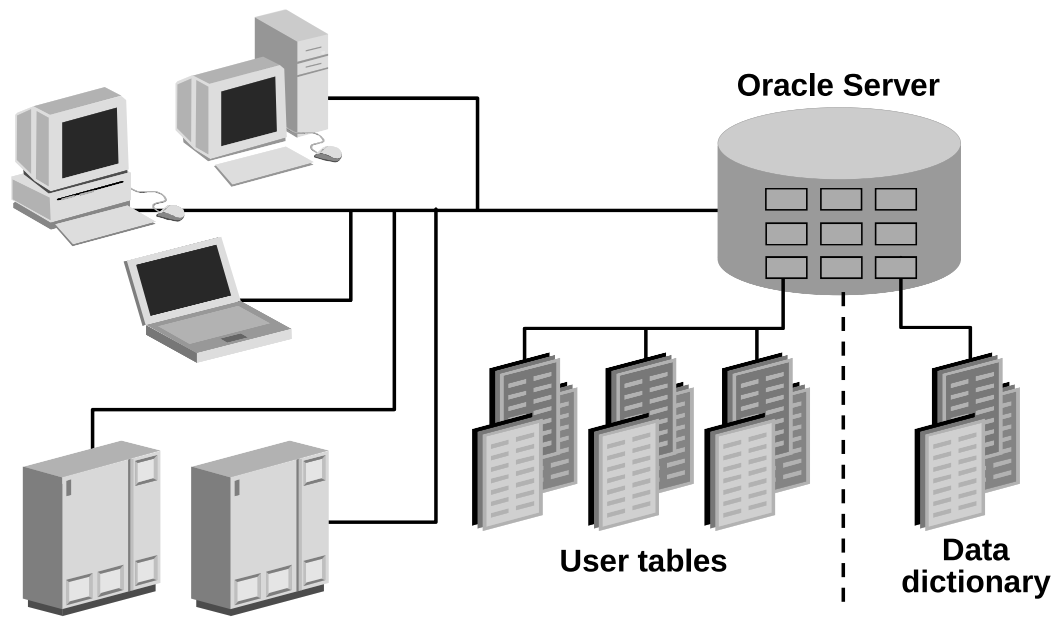
The ability to relate data in one table to data in another enables you to organize information in separate, manageable units. Employee data can be kept logically distinct from department data by storing it in a separate table.

Relating Multiple Tables (Continued)

Guidelines for Primary Keys and Foreign Keys

- No duplicate values are allowed in a primary key.
- It is extremely unlikely that a primary key will be changed. A Primary key can be changed if no foreign key is referencing it
- Foreign keys are based on data values and are purely logical, not physical, pointers.
- A foreign key value must match an existing primary key value or unique key value, or else be null.
- A foreign key must reference either a primary key or unique key column.

Oracle8: Object Relational Database Management System



ORACLE

I-16

Evolution of the Oracle Server

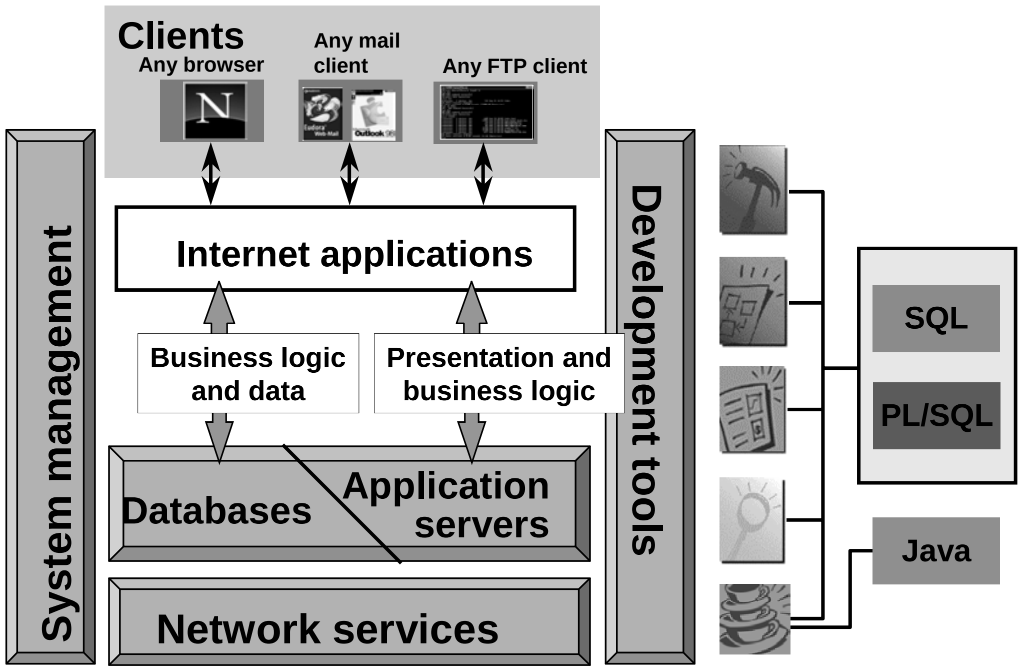
The Oracle server has evolved from an RDBMS to an ORDBMS and is now designed to optimize traditional, Internet and intranet applications, and to stimulate the emerging hosted application market on the Internet.

Object-Relational Database

The Oracle8 Enterprise Edition has made a major leap in data management technology with the introduction of an object-relational paradigm. Database schemas and applications today are becoming increasingly complex. Often, several separate applications with similar data, such as customer information, billing, and shipping, exist in different database schemas and an MIS department must manage the interoperation. Corporate management of the information becomes a difficult task involving the integration of different relational objects and different applications, possibly from different vendors, into a more coherent end-user data model. By enhancing the relational database with object extensions, Oracle addresses the need to simplify data modeling and extend the database with new datatypes.

Oracle applications may run on the same computer as the Oracle8 Server. Alternatively, you can run applications on a local system and run the Oracle8 Server on another system (client-server architecture). This client-server environment provides a wide range of computing resources. For example, a form-based airline reservation application can run on a client personal computer while accessing flight data that is conveniently managed by an Oracle8 Server on a central computer.

Oracle8*i*: Internet Platform



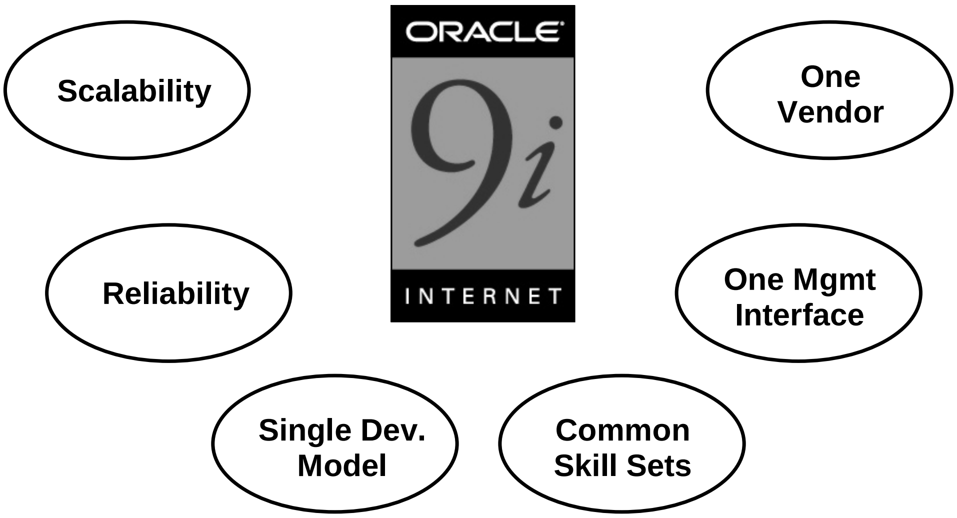
Oracle Internet Platform

Oracle8*i* offers a comprehensive high-performance Internet platform for e-commerce and data warehousing. This integrated platform includes everything needed to develop, deploy, and manage Internet applications. The Oracle Internet platform is built on three core pieces:

- Browser-based clients to process presentation
- Application servers to execute business logic and serve presentation logic to browser-based clients
- Databases to execute database-intensive business logic and serve data

Oracle offers a wide variety of the most advanced graphical user interface (GUI) driven development tools to build business applications, as well as a large suite of software applications for many areas of business and industry. Stored procedures, functions, and packages can be written by using SQL, PL/SQL, or Java.

Oracle9*i*



I-18

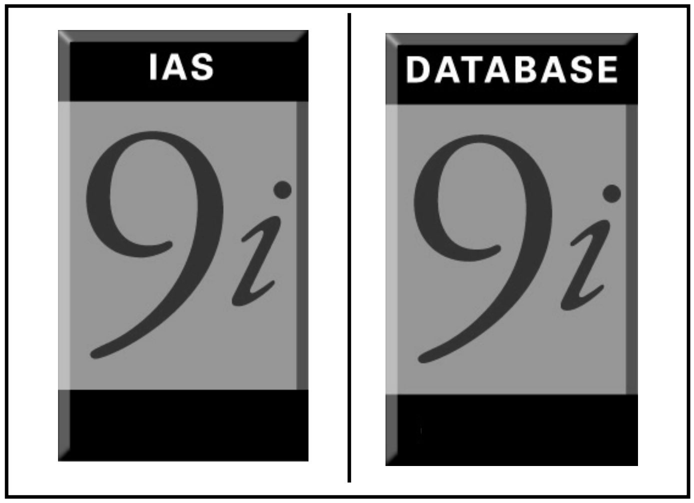
Oracle9*i* Features

Oracle offers a comprehensive high-performance infrastructure for e-business. It is called Oracle9*i*. Oracle9*i* includes everything needed to develop, deploy, and manage Internet applications.

Benefits include:

- Scalability from departments to enterprise e-business sites
- Robust, reliable, available, secure architecture
- One development model, easy deployment options
- Leverage an organization’s current skillset throughout the Oracle platform (including SQL, PL/SQL, Java, and XML)
- One management interface for all applications
- Industry standard technologies, no proprietary lock-in

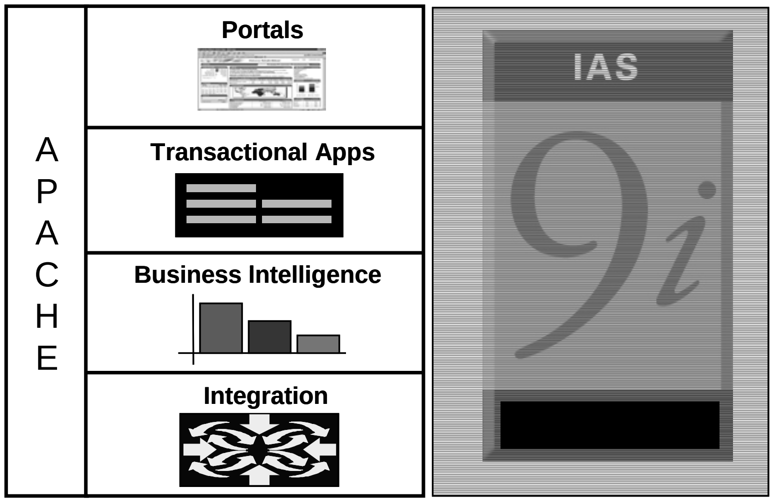
Oracle9*i*



Oracle9*i*

There are two products, Oracle9*i* Application Server and Oracle9*i* Database, that provide a complete and simple infrastructure for Internet applications.

Oracle9*i* Application Server



I-20

Oracle9*i* Application Server

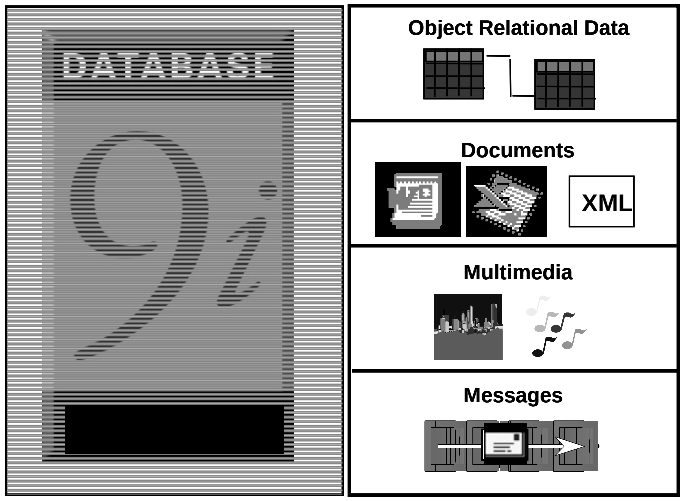
The Oracle9*i* Application Server (Oracle9*i*AS) runs all your applications. The Oracle9*i* Database stores all your data.

Oracle9*i* Application Server is the only application server to include services for all the different server applications you will want to run. Oracle9*i*AS can run your:

- Portals or Web sites
- Java transactional applications
- Business intelligence applications

It also provides integration between users, applications, and data throughout your organization.

Oracle9*i* Database



ORACLE

I-21

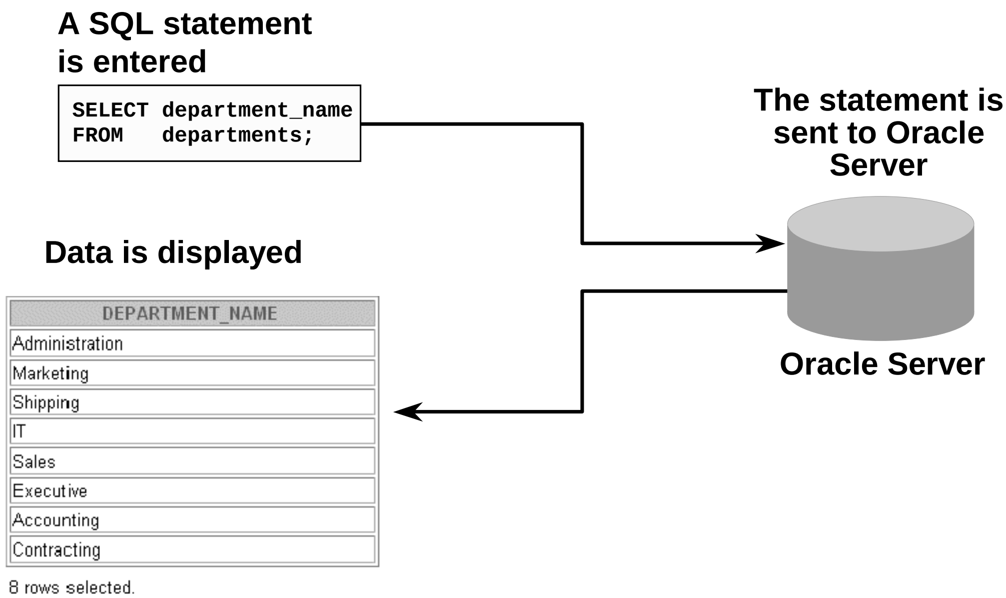
Oracle9*i* Database

The roles of the two products are very straightforward. Oracle9*i* Database manages all your data. This is not just the object relational data that you expect an enterprise database to manage. It can also be unstructured data like:

- Spreadsheets
- Word documents
- PowerPoint presentations
- XML
- Multimedia data types like MP3, graphics, video, and more

The data does not even have to be in the database. Oracle9*i* Database has services through which you can store metadata about information stored in file systems. You can use the database server to manage and serve information wherever it is located.

Communicating with the RDBMS Using SQL



I-22

Structured Query Language

Structured Query Language (SQL) is the set of statements using which all programs and users access data in an Oracle database. In some application programs, you may access the database without directly writing SQL or PL/SQL commands. But these applications in turn must use SQL when executing the user’s request.

Dr. E. F. Codd published the paper, “A Relational Model of Data for Large Shared Data Banks,” in June 1970 in the Association of Computer Machinery (ACM) journal, Communications of the ACM. Codd's model is now accepted as the definitive model for relational database management systems (RDBMS). The language, Structured English Query Language ("SEQUEL") was developed by IBM Corporation, Inc., to use Codd's model. SEQUEL later became SQL (still pronounced “sequel”). In 1979, Relational Software, Inc. (now Oracle Corporation) introduced the first commercially available implementation of SQL. Today, SQL is accepted as the standard RDBMS language.

Structured Query Language (Continued)

How Does SQL Work?

The strengths of SQL provide benefits for all types of users, including application programmers, database administrators, managers, and end users. Technically speaking, SQL is a data sublanguage. The purpose of SQL is to provide an interface to a relational database, such as Oracle, and all SQL statements are instructions to the database. In this, SQL differs from general-purpose programming languages like C and BASIC. The features of SQL are listed below:

- It processes sets of data as groups rather than as individual units.
- It provides automatic navigation to the data.
- It uses statements that are complex and powerful individually, which therefore stand alone.

Data Types Supported in SQL

- `CHAR(size)`
- `VARCHAR2(size)`
- `LONG`
- `NUMBER(p,s)`
- `DATE`
- `TIMESTAMP`
- `RAW(size)`
- `LONG RAW`
- `CLOB`
- `BLOB`

I-24

Data Types Supported in SQL

Each literal or column value manipulated by Oracle has a data type. A value's data type associates a fixed set of properties with it. These properties cause Oracle to treat values of one data type differently from the values of another.

The table on the following page summarizes the Oracle internal data types.

Data Types Supported in SQL (continued)

Data Types	Description
`CHAR(size)`	Used to store fixed length character data of length *size*. Maximum size is 2000 bytes. Default size is 1 byte.
`VARCHAR2(size)`	Used to store a variable-length character string having maximum length *size* bytes. Maximum *size* is 4000 bytes.
`LONG`	Used to store variable-length character data up to 2 GB.
`NUMBER(p,s)`	Used to store a number having a precision *p* and scale *s*. *p* is the number of significant digits and *s* is the scale. *p* is a positive number up to 38 and *s* can vary from -84 to 127. *p* is the total length of numbers excluding the decimal and *s* is the maximum number of digits after the decimal. The decimal does not take up a space.
`DATE`	Used to store dates. Valid dates range from 01/01/4712 BC to 31/12/9999 AD. Both date and time are stored.
`TIMESTAMP (fractional_seconds _precision)`	Year, month, and day values of date, as well as hour, minute, and second values of time, where `fractional_seconds_precision` is the number of digits in the fractional part of the `SECOND` datetime field. Accepted values of `fractional_seconds_precision` are 0 to 9. The default is 6.
`RAW(size)`	Used to store raw binary data of length *size* bytes. Maximum size is 2000 bytes.
`LONG RAW`	Used to store raw binary data of variable length up to 2 GB.
`CLOB`	Character data up to 4 gigabytes
`BLOB`	Binary data up to 4 gigabytes

SQL Statements

`SELECT`	Data retrieval

`INSERT`
`UPDATE`
`DELETE`
`MERGE`
	Data manipulation language (DML)

`CREATE`
`ALTER`
`DROP`
`RENAME`
`TRUNCATE`
	Data definition language (DDL)

`COMMIT`
`ROLLBACK`
`SAVEPOINT`
	Transaction control

`GRANT` `REVOKE`	Data control language (DCL)

ORACLE

I-26

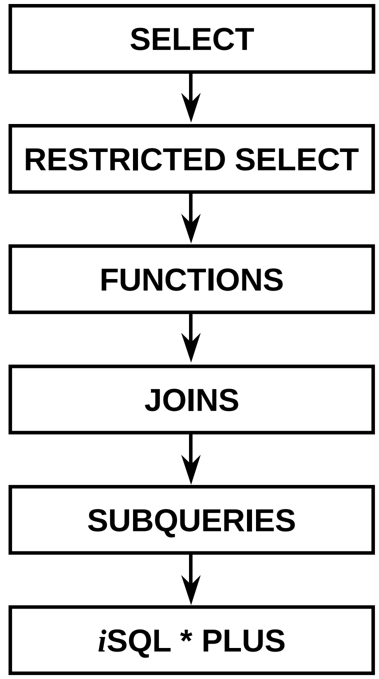
SQL Statements

Oracle SQL complies with industry-accepted standards. Oracle Corporation ensures future compliance with evolving standards by actively involving key personnel in SQL standards committees. Industry-accepted committees are the American National Standards Institute (ANSI) and the International Standards Organization (ISO). Both ANSI and ISO have accepted SQL as the standard language for relational databases.

SQL Statements (continued)

Statement	Description
SELECT	Retrieves data from the database
INSERT UPDATE DELETE MERGE	Enters new rows, changes existing rows, and removes unwanted rows from tables in the database, respectively. Collectively known as *data manipulation language* (DML). Use the MERGE statement to select rows from one table for update or insertion into another table. The decision whether to update or insert into the target table is based on a condition in the ON clause. MERGE is covered in detail in appendix A
CREATE ALTER DROP RENAME TRUNCATE	Sets up, changes, and removes data structures from tables. Collectively known as *data definition language* (DDL).
COMMIT ROLLBACK SAVEPOINT	Manages the changes made by DML statements. Changes to the data can be grouped together into logical transactions.
GRANT REVOKE	Gives or removes access rights to both the Oracle database and the structures within it. Collectively known as *data control language* (DCL).

Overview of Course Material



Course Material Overview

This course has an Introduction and a total of eight lessons that cover the following subjects:

- `SELECT` statements
- Restricted `SELECT` statements
- Functions (single row functions, data conversion functions, and group functions)
- Joins
- Subqueries
- *i*SQL*Plus

Each lesson begins with a statement of the objectives, and Lessons 1 through 8 end with practice exercises.

This course focuses on the relational aspects of Oracle database management systems. In particular, it focuses on the data retrieval language statements. It does not cover user-defined datatypes, or objects. *i*SQL*Plus will be used in the practices to enter and execute SQL statements.

Tables Used in the Course

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALA
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	240
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	170
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	170
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	90
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	42
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	58
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	35
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	31
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	26
				21.2004	09-JUL-98	ST_CLERK	25
				4.1344.429018	29-JAN-00	SA_MAN	105
				4.16			
				4.16			

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

DEPARTMENTS

GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

JOB_GRADES

Tables Used in the Course

The following main tables will be used in this course:

- `EMPLOYEES` table, which gives details of all the employees
- `DEPARTMENTS` table, which gives details of all the departments
- `JOB_GRADES` table, which gives details of salaries for various grades

Note: The structure and data for all the tables is provided in Appendix D.

Summary

- **Relational databases are:**
  - **Composed of relations**
  - **Managed by relational operations**
  - **Governed by data integrity constraints**
- **Oracle8 is based on the object relational database management system.**
- **The Oracle8*i* server introduced a comprehensive high-performance Internet platform for e-commerce and data warehousing.**
- **The Oracle9*i* Server includes two components that provide a complete and simple infrastructure for Internet applications.**
- **You store and manage information in an Oracle server by using the SQL language.**

I-30

Summary

Relational database management systems are composed of objects and relations. They are managed by operations and governed by data integrity constraints.

Oracle8*i* offers a comprehensive high-performance Internet platform for e-commerce and data warehousing.

Oracle9*i* is designed to optimize traditional, internet and intranet applications, and to stimulate the emerging hosted application market on the internet. Oracle9*i* components include the following:

- Oracle9*i* Database
- Oracle9*i* Application Server

SQL is the language you use to communicate with the server to access, manipulate, and control data.

1

Writing Basic SELECT SQL Statements

Objectives

After completing this lesson, you should be able to do the following:

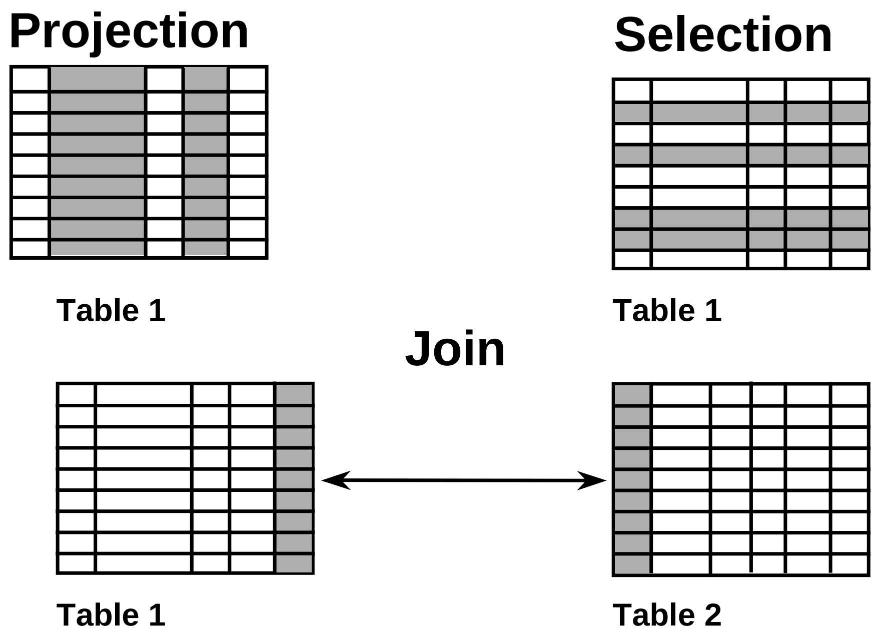
- List the capabilities of SQL `SELECT` statements
- Execute a basic `SELECT` statement
- Differentiate between SQL statements and *i*SQL*Plus commands

1-2

Lesson Aim

To extract data from the database you need to use the structured query language (SQL) `SELECT` statement. You may need to restrict the columns that are displayed. This lesson describes all the SQL statements that you need to perform these actions. This lesson also covers the use of *i*SQL*Plus commands to execute SQL statements.

Capabilities of SQL SELECT Statements



Using SQL SELECT Statements

A SELECT statement retrieves information from the database. You can use a SELECT statement to do the following:

- Projection: Choose the columns in a table that you want the query to return. You can choose as few or as many columns of the table as you require.
- Selection: Choose the rows in a table that you want the query to return. You can use various criteria to restrict the rows that you see.
- Join: Bring together data stored in different tables by creating a link through a column that appears in both tables.

You will learn more about selections and joins in a later lesson.

Basic SELECT Statement

```
SELECT        [DISTINCT] {*|column|expression [alias],...}
FROM          table
[WHERE        condition(s)]
[GROUP BY     group_by_expression]
[ORDER BY     column];
```

- **SELECT identifies the columns to be displayed.**
- **FROM identifies the table that contains the columns.**

Contents of a Basic SELECT Statement

In its simplest form, a SELECT statement must include the following:

- A SELECT clause, which specifies the columns to be displayed
- A FROM clause, which specifies the table that contains the columns listed in the SELECT clause

In the syntax:

SELECT	Displays a list of one or more columns
DISTINCT	Suppresses duplicate rows and lists columns in ascending order
*	Selects all columns
column	Selects the named column
alias	Gives selected column a different heading
FROM *table*	Specifies the table that contains the columns

Contents of a Basic SELECT Statement (Continued)

`WHERE`	Restricts the query result to rows that meet a condition
`conditions`	Composed of column names, expressions, constants, and a comparison operator
`GROUP BY`	Divides the rows in a table into groups
`group_by_expression`	Specifies columns whose values determine the basis for grouping rows
`ORDER BY`	Sorts the rows in the output

Note: The words *keyword*, *clause*, and *statement* are used throughout this course:

- A *keyword* refers to an individual SQL element.
  For example, `SELECT` and `FROM` are keywords.
- A *clause* is a part of a SQL statement.
  For example, `SELECT employee_id, last_name` is a clause.
- A *statement* is a combination of two or more clauses.
  For example, `SELECT * FROM EMPLOYEES` is a SQL statement.

Writing SQL Statements

- **SQL statements are not case sensitive.**
- **SQL statements can be on one or more lines.**
- **Keywords cannot be abbreviated or split across lines.**
- **Clauses are usually placed on separate lines.**
- **Tabs and indents are used to enhance readability.**

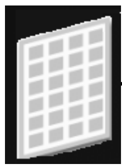
Writing SQL Statements

To construct valid SQL statements that are both easy to read and edit, follow these simple rules and guidelines:

- SQL statements are not case sensitive.
- You can enter SQL statements on one or more lines.
- You cannot abbreviate or split keywords across lines.
- Place clauses on separate lines for readability and ease of editing.
- Use tabs and indents to make code more readable.
- Enter keywords in uppercase. Enter all other words, such as table names and columns, in lowercase. This enhances readability.

Retrieving All Columns from a Table

DEPARTMENTS TABLE



Retrieve all columns from the DEPARTMENTS table

DEPARTMENTS TABLE

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

8 rows selected.

1-7

Retrieving All Columns

Assume that you want to display all of the columns of information stored in the `DEPARTMENTS` table. This is the simplest use of the `SELECT` statement in SQL. All rows are retrieved from the table and all columns are displayed. This is the equivalent of retrieving the entire contents of a table. In the case of the `DEPARTMENTS` table, the "entire contents of the table" translates to all details of all departments.

Selecting All Columns

```
SELECT   *
FROM     departments;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

8 rows selected.

Selecting All Columns, All Rows

You can display all columns of data in a table by following the SELECT keyword with an asterisk (*). In the example on the slide, the DEPARTMENTS table contains four columns: DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION_ID. The table contains eight rows, one for each department.

You can also display all columns in the table by listing all the columns after the SELECT keyword.

To display all columns in the JOB_GRADES table, enter the following command.

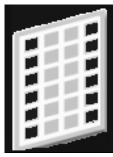
```
SELECT grade_level, lowest_sal, highest_sal
FROM   job_grades;
```

You can also use the * character to display all columns in the JOB_GRADES table.

```
SELECT *
FROM job_grades;
```

Creating a Projection on a Table

DEPARTMENTS TABLE



Retrieve DEPARTMENT_ID and DEPARTMENT_NAME columns from the DEPARTMENTS table

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
50	Shipping
60	IT
80	Sales
90	Executive
110	Accounting
190	Contracting

8 rows selected.

Only two columns are displayed

DEPARTMENTS TABLE

ORACLE

1-9

Retrieving Specific Columns

Assume that you want to display only two columns of data stored in the DEPARTMENTS table. This "projection" is a typical use of the SELECT statement in SQL. You can use the projection on the DEPARTMENTS table to select only certain details about each employee, in this case, the department number and the location of each department.

To display the DEPARTMENT_ID and LOCATION_ID columns from the DEPARTMENTS table enter the following command:

```
SELECT department_id, location_id
FROM   departments;
```

Basic Projection Rules

- Use an asterisk (*) to display all columns.
- You can select as many columns as you want.
- Use a comma to separate the column names.
- The columns appear in the order selected.

Selecting Specific Columns

```
SELECT department_id, department_name
FROM   departments;
```

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
50	Shipping
60	IT
80	Sales
90	Executive
110	Accounting
190	Contracting

8 rows selected.

ORACLE

Selecting Specific Columns, All Rows

You can use the SELECT statement to display specific columns of the table by specifying the column names, separated by commas. The example in the slide displays all the department numbers and department names from the DEPARTMENTS table.

In the SELECT clause, specify the columns in the order in which you want them to appear in the output. For example, to display department name before department ID, use the following statement:

```
SELECT department_name,department_id
FROM   employees;
```

Selecting Specific Columns, All Rows (Continued)

DEPARTMENT_NAME	DEPARTMENT_ID
Administration	10
Marketing	20
Shipping	50
IT	60
Sales	80
Executive	90
Accounting	110
Contracting	190

8 rows selected.

Default Data Justification

Character left justified

Date left justified

Number right justified

`EMPLOYEES` table

LAST_NAME	HIRE_DATE	SALARY
King	17-JUN-87	24000
Kochhar	21-SEP-89	17000
De Haan	13-JAN-93	17000
Hunold	03-JAN-90	9000
Ernst	21-MAY-91	6000
Lorentz	07-FEB-99	4200
Mourgos	16-NOV-99	5800
Rajs	17-OCT-95	3500
Davies	29-JAN-97	3100
Matos	15-MAR-98	2600
Vargas	09-JUL-98	2500

1-12

Default Data Justification

- Character and date data are left justified.
- Number data are right justified.
- By default, the results of queries display column headings in uppercase.
- You can override the column heading display with an alias. Column aliases are covered later in this lesson.
- Use the `SELECT` statement given below to display the last name, hire date and salary of the employees. The results are as displayed in the slide.

```
SELECT last_name,hire_date,salary
FROM employees;
```

Arithmetic Expressions

Create expressions on `NUMBER` and `DATE` data types by using arithmetic operators.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

1-13

Using Arithmetic Expressions

You may need to modify the way in which data is displayed, perform calculations, or look at what-if scenarios. You can do so by using arithmetic expressions. An arithmetic expression may contain column names, constant numeric values, and the arithmetic operators.

The table in the slide lists the arithmetic operators available in SQL. You can use arithmetic operators in any clause of a SQL statement except the `FROM` clause.

Using Arithmetic Operators

```
SELECT last_name, salary, salary+300
FROM employees;
```

LAST_NAME	SALARY	SALARY+300
King	24000	24300
Kochhar	17000	17300
De Haan	17000	17300
Hunold	9000	9300
Ernst	6000	6300
Lorentz	4200	4500
Mourgos	5800	6100
Rajs	3500	3800
Davies	3100	3400
Matos	2600	2900
Vargas	2500	2800
Zlotkey	10500	10800
Abel	11000	11300

...

20 rows selected.

1-14

Using Arithmetic Operators

The example in the slide uses the addition operator to calculate a salary increase of $300 for all employees and displays a new column, SALARY+300 in the output.

Note that the calculated column, SALARY+300, is not a new column in the EMPLOYEES table; it is for display only. By default, the name of the new column comes from the calculation that generated it: in this case, SALARY+300.

Note: SQL ignores blank spaces before and after the arithmetic operator.

Using Arithmetic Operators on Multiple Columns

```
SELECT last_name,department_id, salary,
       salary * commission_pct
FROM employees;
```

LAST_NAME	DEPARTMENT_ID	SALARY	SALARY*COMMISSION_PCT
King	90	24000	
Kochhar	90	17000	
De Haan	90	17000	
Hunold	60	9000	
...			
Zlotkey	80	10500	2100
Abel	80	11000	3300
Taylor	80	8600	1720
...			
Fay	20	6000	
Higgins	110	12000	
Gietz	110	8300	

20 rows selected.

1-15

Using Arithmetic Operators on Multiple Columns

The example in the slide multiplies the value in the SALARY column with the value in the COMMISSION_PCT column for each row in the EMPLOYEES table.

Operator Precedence

* / + –

- Multiplication and division take priority over addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to force prioritized evaluation and to clarify statements.

1-16

Operator Precedence

If an arithmetic expression contains more than one operator, multiplication and division are evaluated first. If operators within an expression have the same priority, evaluation is done from left to right.

Expressions within parentheses are evaluated first, so you can use parentheses to change precedence.

Operator Precedence

```
SELECT last_name, salary, 100+salary*12
FROM   employees;
```

LAST_NAME	SALARY	100+SALARY*12
King	24000	288100
Kochhar	17000	204100
De Haan	17000	204100
Hunold	9000	108100
Ernst	6000	72100
Lorentz	4200	50500
Mourgos	5800	69700
Rajs	3500	42100
Davies	3100	37300
Matos	2600	31300
Vargas	2500	30100
Zlotkey	10500	126100
Abel	11000	132100

...

20 rows selected.

ORACLE

1-17

Operator Precedence (Continued)

The example in the slide displays the name, salary, and annual compensation of employees.

The example calculates the annual compensation as 12 multiplied by the monthly salary, plus a one-time bonus of $100 because multiplication has a higher order of precedence than addition. Observe the output that shows multiplication was done before the addition and not from left to right.

Note: Use parentheses to reinforce the standard order of precedence and improve clarity. For example, the expression in the slide can be written as `100+(12*SALARY)` with no change in the result.

Using Parentheses

```
SELECT last_name, salary, (100+salary)*12
FROM   employees;
```

LAST_NAME	SALARY	(100+SALARY)*12
King	24000	289200
Kochhar	17000	205200
De Haan	17000	205200
Hunold	9000	109200
Ernst	6000	73200
Lorentz	4200	51600
Mourgos	5800	70800
Rajs	3500	43200
Davies	3100	38400
Matos	2600	32400
Vargas	2500	31200
Zlotkey	10500	127200
Abel	11000	133200

...

20 rows selected.

ORACLE

1-18

Using Parentheses to Override Operator Precedence

You can override the rules of precedence by using parentheses to specify the order in which operators are executed.

The example in the slide displays the name, salary, and annual compensation of employees. It calculates the annual compensation as monthly salary plus a monthly bonus of $100, multiplied by 12. Expressions in parentheses are evaluated first; therefore the addition takes priority over the multiplication.

Defining a Column Alias

- **Renames a column heading**
- **Is useful with calculations**
- **Immediately follows the column name or expression**
- **Include the optional AS keyword between the column name and the alias**
- **Requires double quotation marks if it is case sensitive or contains spaces or special characters**

ORACLE

1-19

Column Aliases

When displaying the result of a query, *i*SQL*Plus normally uses the name of the selected column as the column heading. In many cases, this heading is not descriptive and thus is difficult to understand. You can change a column heading by using a column alias.

Specify the alias after the column or expression in the `SELECT` list using a space as the separator or following the keyword AS. By default, alias headings appear in uppercase. If the alias is case sensitive or if it contains spaces or special characters such as # or $, enclose it in double quotation marks (" "). Column aliases can contain spaces and special characters such as # and $.

Using Column Aliases

```
SELECT last_name "Name",
       salary*12 "Annual Salary"
FROM   employees;
```

Name	Annual Salary
King	288000
Kochhar	204000
De Haan	204000
Hunold	108000
Ernst	72000
Lorentz	50400
Mourgos	69600
Rajs	42000
Davies	37200
Matos	31200
Vargas	30000
Zlotkey	126000
Abel	132000

...

20 rows selected.

ORACLE

1-20

Using Column Aliases (Continued)

The example displays the name and annual salary of all the employees. Because "`Annual Salary`" contains spaces, it has been enclosed in double quotation marks. Notice that the column heading in the output are exactly the same as the column alias.

The Concatenation Operator

- Concatenates columns or character strings to other columns
- Is represented by two vertical bars ||
- Creates a result column that is a character expression

1-21

The Concatenation Operator

You can link columns to other columns, arithmetic expressions, or constant values to create a character expression, by using the concatenation operator ||. Columns on either side of the operator are combined to make a single output column.

Using the Concatenation Operator

```
SELECT first_name || last_name AS "Names"
FROM employees;
```

Names
StevenKing
NeenaKochhar
LexDe Haan
AlexanderHunold
BruceErnst
DianaLorentz
KevinMourgos
TrennaRajs
CurtisDavies
RandallMatos
PeterVargas
EleniZlotkey
EllenAbel

...

20 rows selected.

ORACLE

Using the Concatenation Operator

In the example, FIRST_NAME and LAST_NAME are concatenated and given the alias “Names.” Notice that the first name and last name are combined to make a single output column.

Literals

- **A literal is a constant value of character, expression, or number that can be included in the `SELECT` list.**
- **Date and character literal values must be enclosed in single quotation marks.**
- **Each character string is displayed once for each row returned.**

ORACLE

1-23

Literals

The terms literal and constant value are synonymous and refer to a fixed data value. For example, 'JACK', 'BLUE ISLAND', and '101' are all character literals; 5001 is a numeric literal. Note that character literals are enclosed in single quotation marks (' '), which enable Oracle to distinguish them from schema object names. Number literals should not be enclosed in single quotation marks.

Many SQL statements and functions require you to specify character and numeric literal values. You can also specify literals as part of expressions and conditions.

Using Literal Character Strings

```
SELECT last_name ||' is a '|| job_id
       AS "Employee Details"
FROM   employees;
```

Employee Details
King is a AD_PRES
Kochhar is a AD_VP
De Haan is a AD_VP
Hunold is a IT_PROG
Ernst is a IT_PROG
Lorentz is a IT_PROG
Mourgos is a ST_MAN
Rajs is a ST_CLERK
Davies is a ST_CLERK
Matos is a ST_CLERK
Vargas is a ST_CLERK
Zlotkey is a SA_MAN
Abel is a SA_REP

...

20 rows selected.

1-24

Using Literal Character Strings

The example in the slide displays the names and jobs of all employees. The column has the heading “Employee Details”. Notice the space between the single quotation marks in the `SELECT` statement. The spaces improve the readability of the output.

In the following example the name and salary of each employee is concatenated with a literal to give the returned rows more meaning:

```
SELECT last_name ||': 1 Month salary = '||salary MONTHLY
FROM   employees;
```

MONTHLY
King: 1 Month salary = 24000
Kochhar: 1 Month salary = 17000
De Haan: 1 Month salary = 17000
Hunold: 1 Month salary = 9000
Ernst: 1 Month salary = 6000
Lorentz: 1 Month salary = 4200

...

20 rows selected.

Duplicate Rows

The default display of queries is all rows, including duplicate rows.

```
SELECT department_id
FROM   employees;
```

DEPARTMENT_ID
90
90
90
60
60
60
50
50
50
50
50
80
80

...

20 rows selected.

ORACLE

Duplicate Rows

Unless you indicate otherwise, *i*SQL*Plus displays the results of a query without eliminating duplicate rows. The example in the slide displays all the department numbers from the `EMPLOYEES` table. Notice that some department numbers are repeated.

Eliminating Duplicate Rows

Eliminate duplicate rows by using the `DISTINCT` keyword in the `SELECT` clause.

```
SELECT DISTINCT department_id
FROM   employees;
```

DEPARTMENT_ID
10
20
50
60
80
90
110

8 rows selected.

ORACLE

Eliminating Duplicate Rows

To eliminate duplicate rows in the result, include the `DISTINCT` keyword in the `SELECT` clause immediately after the `SELECT` keyword. In the example in the slide, the `EMPLOYEES` table actually contains 20 rows but there are only eight distinct department numbers in the table. You can specify multiple columns after the `DISTINCT` qualifier. The `DISTINCT` qualifier affects all the selected columns, and the result represents a distinct combination of the columns.

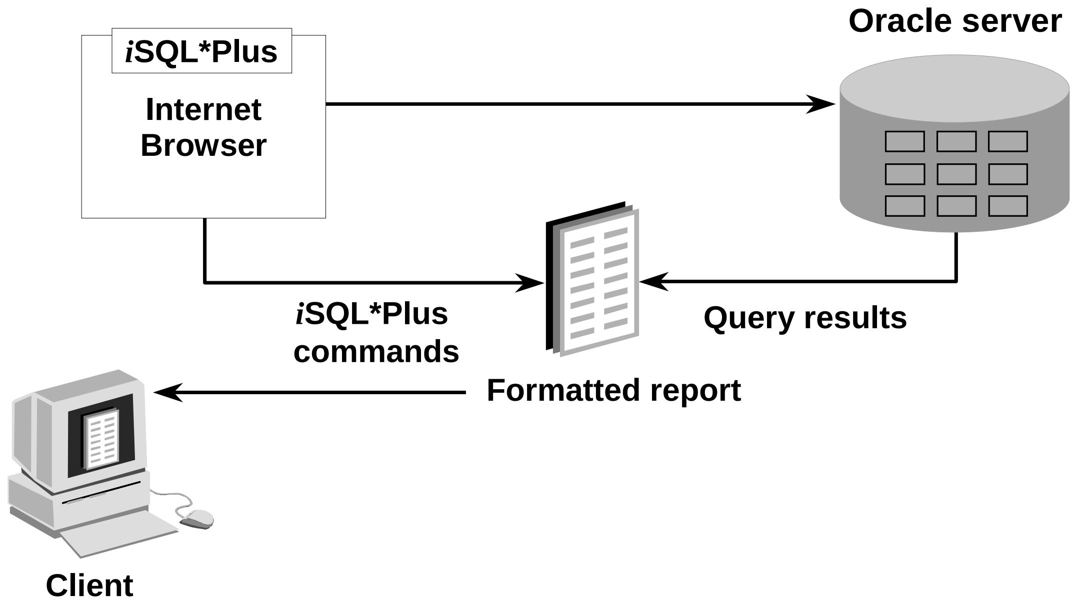
```
SELECT DISTINCT department_id, job_id
 FROM employees;
```

DEPARTMENT_ID	JOB_ID
10	AD_ASST
20	MK_MAN
20	MK_REP
50	ST_CLERK
50	ST_MAN
60	IT_PROG

...

13 rows selected.

SQL and *i*SQL*Plus Interaction



1-27

SQL and *i*SQL*Plus

SQL is a command language for communication with the Oracle Server from any tool or application.

*iSQL*Plus* is an Oracle tool that recognizes and submits SQL statements to the Oracle Server for execution and contains its own command language.

Features of SQL

- Can be used by a range of users, including those with little or no programming experience
- Is a nonprocedural language
- Reduces the amount of time required for creating and maintaining systems
- Is an English-like language

Features of *i*SQL*Plus

- Can be accessed from a browser
- Accepts ad hoc entry of statements
- Provides online editing for modifying SQL statements
- Controls environmental settings
- Formats query results into a basic report
- Accesses local and remote databases

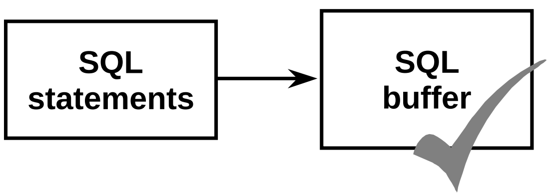
SQL Statements Versus *i*SQL*Plus Commands

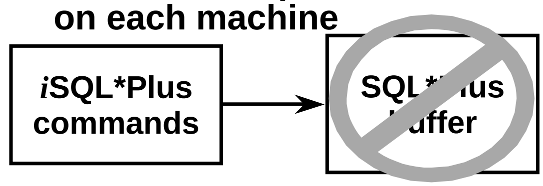
SQL

- A language
- ANSI standard
- Keywords cannot be abbreviated
- Statements manipulate data and table definitions in the database

***i*SQL*Plus**

- An environment
- An Oracle proprietary tool
- Keywords can be abbreviated
- Commands do not allow manipulation of values in the database
- Runs on a browser
- Centrally loaded, does not have to be implemented on each machine





ORACLE

1-28

SQL and *i*SQL*Plus (Continued)

The following table compares SQL and *i*SQL*Plus:

SQL	***i*SQL*Plus**
Is a language for communicating with the Oracle server to access data	Recognizes SQL statements and sends them to the server
Is based on American National Standards Institute (ANSI) standard SQL	Is the Oracle proprietary interface for executing SQL statements
Manipulates data and table definitions in the database	Does not allow manipulation of values in the database
Does not have a continuation character	Has a dash (-) as a continuation character if the command is longer than one line
Cannot be abbreviated	Can be abbreviated
Uses a termination character (`;`) to execute commands immediately	Does not require termination characters; executes commands immediately **Note**: If more than one SQL statements are being executed simultaneously, then each SQL statement must be terminated by a ; symbol.
Uses functions to perform some formatting	Uses commands to format data

Overview of *i*SQL*Plus

After you log in to *i*SQL*Plus, you can:

- **Describe the table structure**
- **Edit your SQL statement**
- **Execute SQL from *i*SQL*Plus**
- **Save SQL statements to files and append SQL statements to files**
- **Execute statements stored in saved files**
- **Load commands from a text file into the *i*SQL*Plus window**

ORACLE

1-29

*i*SQL*Plus

*i*SQL*Plus is an environment in which you can do the following:

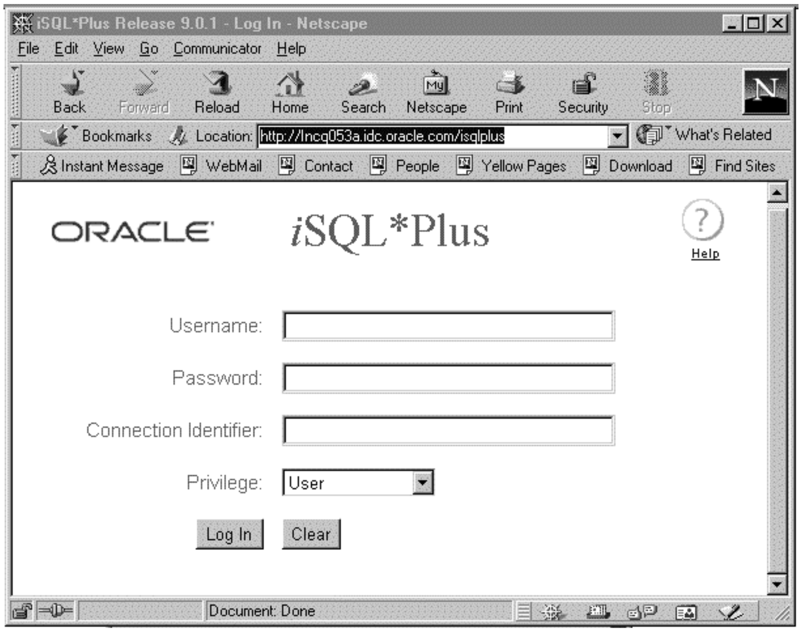
- Execute SQL statements to retrieve, modify, add, and remove data from the database
- Format, perform calculations on, store, and print query results in the form of reports
- Create script files to store SQL statements for repetitive use.

*i*SQL*Plus commands can be divided into the following main categories:

Category	**Purpose**
Environment	Affects the general behavior of SQL statements for the session
Format	Formats query results
File manipulation	Saves statements into text script files, and run statements from text script files
Execution	Sends SQL statements from the browser to the Oracle server
Edit	Modifies SQL statements in the Edit window
Interaction	Allows to create and pass variables to SQL statements, print variable values, and print messages to the screen
Miscellaneous	Has various commands to connect to the database, manipulate the *i*SQL*Plus environment, and display column definitions

Logging On to *i*SQL*Plus

From your Windows browser environment



1-30

Logging On to *i*SQL*Plus

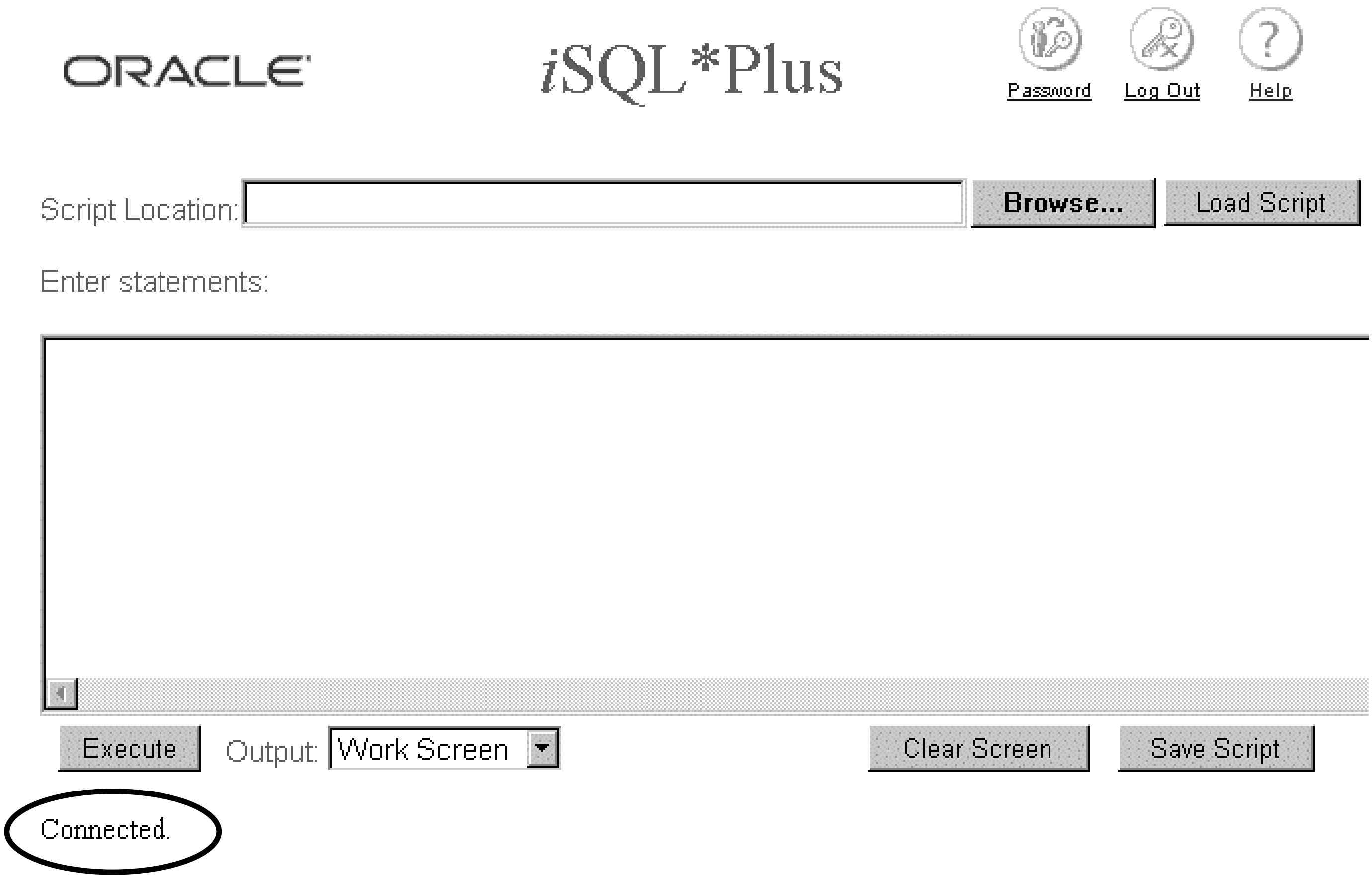
To log on through a browser environment:

1. Start the browser.
2. Enter the URL address of the *i*SQL*Plus environment.
3. Fill in the username, password, and Oracle Connection Identifier fields.
4. The Privilege dropdown list has three options:
   - User--is the default connection. *i*SQL*Plus connects to the specified database with no administrator privileges.
   - AS SYSDBA--connects to the specified database with SYSDBA privileges.
   - AS SYSOPER--connects to the specified database with SYSOPER privileges.

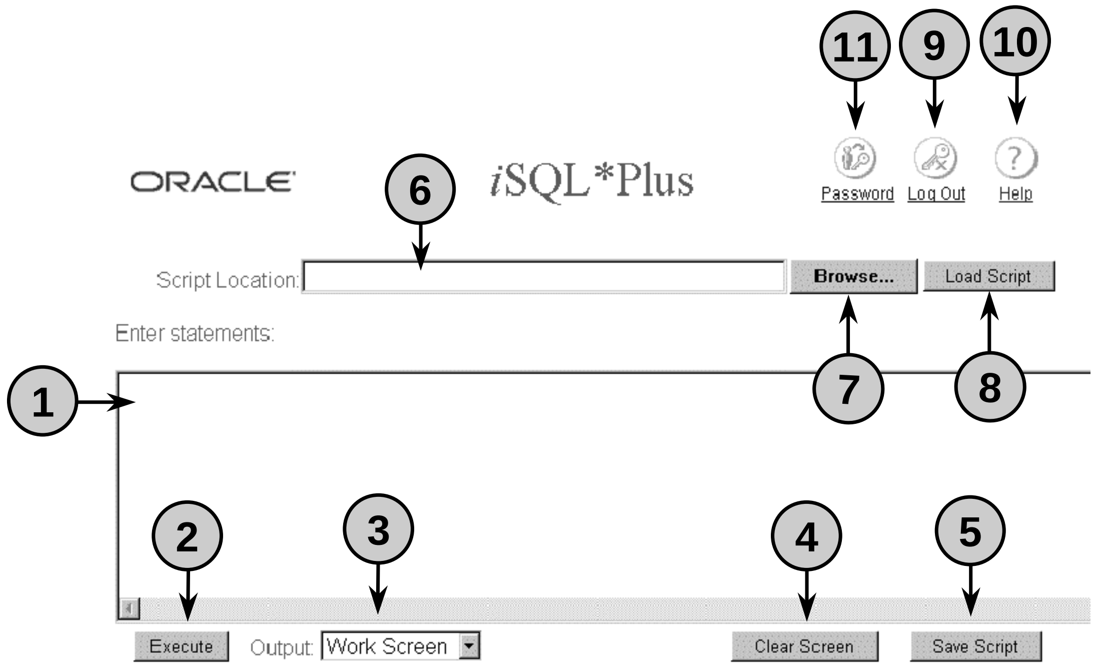
To connect with either SYSDBA or SYSOPER privileges, your username and password must be added to the Oracle HTTP Server authentication file.

After you have successfully logged on to *i*SQL*Plus, you see the following screen:

Logging On to *i*SQL*Plus (Continued)



The *i*SQL*Plus Environment

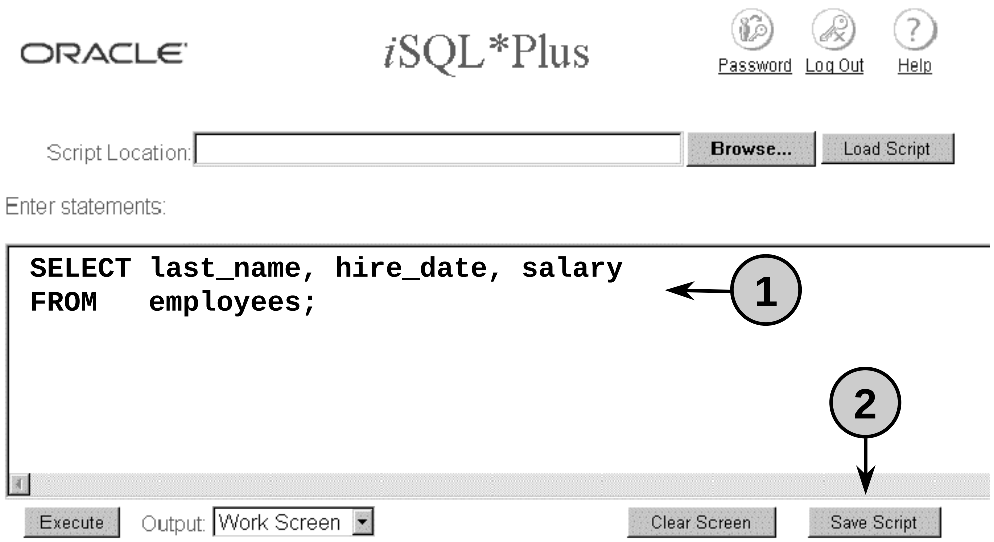


The *i*SQL*Plus Environment

Within the Windows browser, the *i*SQL*Plus window has several key areas:

1. Edit window: The area where you type the SQL statements and *i*SQL*Plus commands.
2. Execute button: Click to execute the statements and commands in the Edit window.
3. Output option: Defaults to Work Screen, which displays the results of the SQL statement beneath the edit window. The other options are File or Window. File saves the contents to a specified file. Window places the output to the screen, but in a separate window.
4. Clear Screen button: Click to clear text from the edit window.
5. Save Script button: Saves the contents of the edit window to a file.
6. Script Location: Identifies the name and location of the script file that you want to execute.
7. Browse button: Allows you to search for a script file using the Windows File upload dialog box.
8. Load script: Click the Load Script button to load the script specified in the Script location: field into the *i*SQL*Plus input area for editing or execution.
9. Log out icon: Click to end the *i*SQL*Plus session and return to the *i*SQL*Plus log in screen.
10. Help icon: Provides access to *i*SQL*Plus Help documentation.
11. Password icon: Allows you to change your password.

Interacting with Script Files



ORACLE

1-33

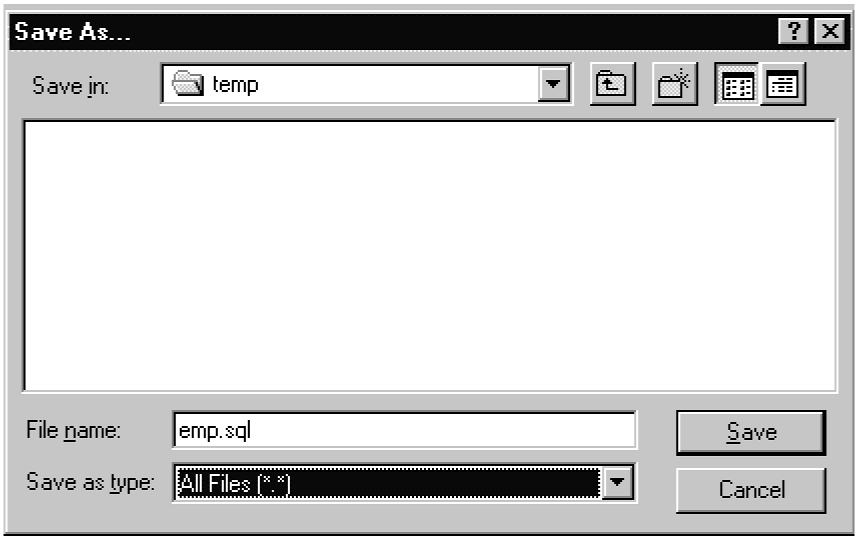
Interacting with Script Files

Placing Statements and Commands into a Text Script File

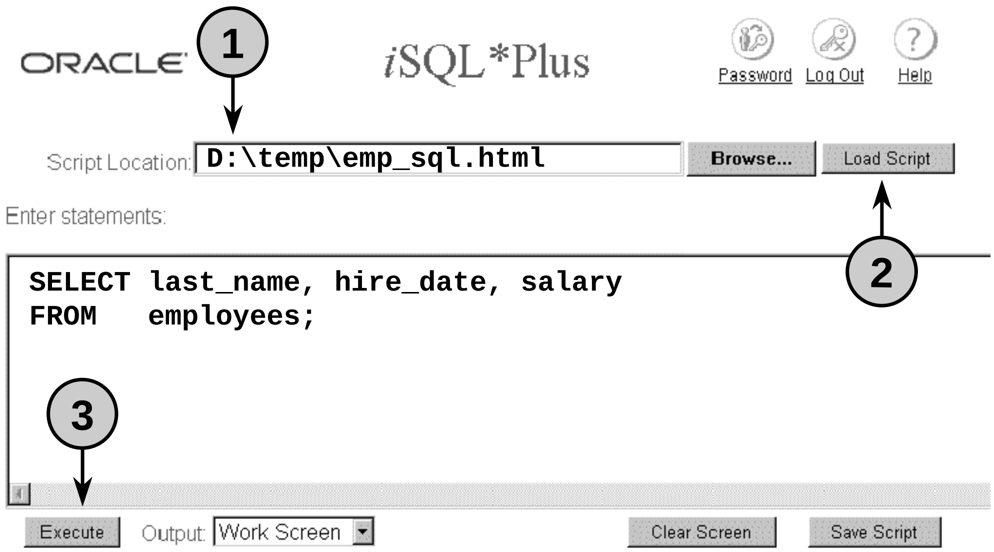
You can save commands and statements from the window in *i*SQL*Plus to a text script file as follows:

1. Type the SQL statement(s) into the Edit window in *i*SQL*Plus.
2. Click the Save Script button. This brings up the Windows File Save As dialog box. Identify the name of the file. It defaults to a `.html` extension. You can change the file type to a text file or save it as a `.sql` file. The Windows File Save As dialog box is shown in the next page.

Interacting with Script Files (Continued)



Interacting with Script Files



ORACLE

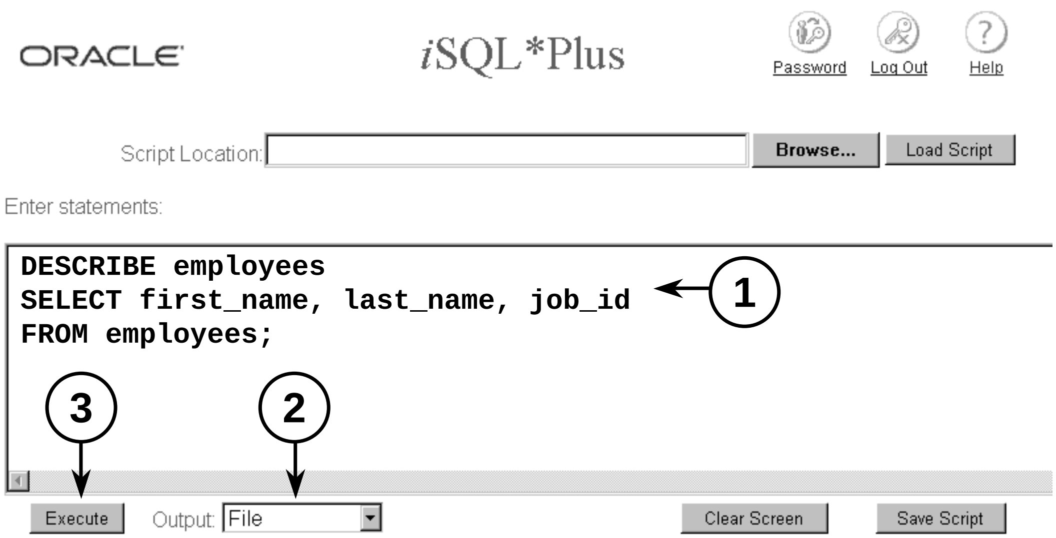
Interacting with Script Files

Using Statements and Commands from a Script File in *i*SQL*Plus

You can use previously saved commands and statements from a script file in *i*SQL*Plus as follows:

1. Type in the script name and location. Or, you can click the Browse button to find the script name and location.
2. Click the Load Script button. The file contents are loaded into the *i*SQL*Plus edit window.
3. Click the Execute button to run the contents of the *i*SQL*Plus edit window.

Interacting with Script Files



1-36

Interacting with Script Files

Saving Output to a File

You can save the results generated from a SQL statement or *i*SQL*Plus command to a file:

- Type the SQL statement(s) and *i*SQL*Plus command(s) into the edit window in *i*SQL*Plus .
- Change the Output option to File.
- Click the Execute button to run the contents of the *i*SQL*Plus edit window. This brings up the File Save As dialog box. Specify the desired file name. It defaults to a `.html` extension.

You can change the file type. The results are sent to a file with the specified name.

Displaying Table Structure

Use the *i*SQL*Plus `DESCRIBE` command to display the structure of a table.

```
DESC[RIBE] tablename
```

1-37

Displaying the Table Structure

In *i*SQL*Plus, you can display the structure of a table using the `DESCRIBE` command. The command shows the column names and data types, as well as whether a column *must* necessarily contain data.

In the syntax:

`tablename` is the name of any existing table, view, or synonym accessible to the user

Note: As mentioned before, *i*SQL*Plus command words can be abbreviated, but must contain at least the first four characters. This is why the characters `'RIBE'` of `DESCRIBE` are shown as optional.

Displaying Table Structure

```
DESCRIBE employees
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

ORACLE

1-38

Displaying the Table Structure (Continued)

The example on the slide displays the information about the structure of the `EMPLOYEES` table.

In the result:

Null? indicates whether a column *must* contain data; `NOT NULL` indicates that a column must contain data.

Type displays the data type for a column.

The data types are described in the following table:

Data Type	Description
NUMBER(*p*,*s*)	Number value having a maximum number of digits *p*, with *s* digits to the right of the decimal point
VARCHAR2(*s*)	Variable-length character value of maximum size *s*
DATE	Date and time value between January 1, 4712 B.C., and December 31, 9999 A.D.

Summary

Use *i*SQL*Plus as an environment to:

- Retrieve data from a database table with the SELECT statement
- Execute SQL statements
- Edit SQL statements

```
SELECT       [DISTINCT] {*, column|expression [alias],...}
FROM         table
[WHERE       condition(s)]
[GROUP BY    group_by_expression]
[ORDER BY    column];
```


SELECT Statement

In this lesson, you have learned about retrieving data from a database table with the SELECT statement.

```
SELECT  [DISTINCT] {*,column[alias],...}
FROM    table;
```

SELECT	Displays a list of at least one column
DISTINCT	Suppresses duplicates
*	Selects all columns
column	Selects the named column
alias	Gives selected columns different headings
FROM *table*	Specifies that the table contains the columns

*i*SQL*Plus

*i*SQL*Plus is an execution environment that you can use to send SQL commands to the database server and to edit and save SQL commands. You can execute commands from the SQL prompt or from a script file.

Practice 1 Overview

This practice covers the following topics:

- Selecting all data from different tables
- Describing the structure of tables
- Performing arithmetic calculations
- Specifying column names

1-40

Practice 1 Overview

This is the first of many practices. The solutions (if you require them) can be found in Appendix C, "Practice Solutions." Practices are intended to introduce all topics covered in the lesson. Questions 2 through 4 are paper-based.

Some practices contain "if you want extra challenge" questions. Do these only if you have completed all of the other questions within the allocated time and would like a further challenge to your skills.

Take the practices slowly and precisely. You can experiment with saving and running command files. If you have any questions ask your instructor.

Practice 1

1. Initiate an *i*SQL*Plus session by using the user ID and password provided by the instructor.
2. SQL commands are always held in the buffer.

   True/False
3. *i*SQL*Plus commands are used to query data.

   True/False
4. Show the structure of the `DEPARTMENTS` table.

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

5. Select all information from the `DEPARTMENTS` table.

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

8 rows selected.

Practice 1 (Continued)

6. Show the structure of the `EMPLOYEES` table.

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

Practice 1 (Continued)

7. Display the last name and hire date for each employee.

LAST_NAME	HIRE_DATE
King	17-JUN-87
Kochhar	21-SEP-89
De Haan	13-JAN-93
Hunold	03-JAN-90
Ernst	21-MAY-91
Lorentz	07-FEB-99
Mourgos	16-NOV-99
Rajs	17-OCT-95
Davies	29-JAN-97
Matos	15-MAR-98
Vargas	09-JUL-98
Zlotkey	29-JAN-00
Abel	11-MAY-96
Taylor	24-MAR-98
LAST_NAME	HIRE_DATE
Grant	24-MAY-99
Whalen	17-SEP-87
Hartstein	17-FEB-96
Fay	17-AUG-97
Higgins	07-JUN-94
Gietz	07-JUN-94

20 rows selected.

Practice 1 (Continued)

8. Display the hire date and last name for each employee, with the hire date appearing first.

HIRE_DATE	LAST_NAME
17-JUN-87	King
21-SEP-89	Kochhar
13-JAN-93	De Haan
03-JAN-90	Hunold
21-MAY-91	Ernst
07-FEB-99	Lorentz
16-NOV-99	Mourgos
17-OCT-95	Rajs
29-JAN-97	Davies
15-MAR-98	Matos
09-JUL-98	Vargas
29-JAN-00	Zlotkey
11-MAY-96	Abel
24-MAR-98	Taylor
HIRE_DATE	LAST_NAME
24-MAY-99	Grant
17-SEP-87	Whalen
17-FEB-96	Hartstein
17-AUG-97	Fay
07-JUN-94	Higgins
07-JUN-94	Gietz

20 rows selected.

Practice 1 (Continued)

9. Display the last name, hire date, and annual salary, excluding commission, for each employee. Label the annual salary column as ANNUAL.

LAST_NAME	HIRE_DATE	ANNUAL
King	17-JUN-87	288000
Kochhar	21-SEP-89	204000
De Haan	13-JAN-93	204000
Hunold	03-JAN-90	108000
Ernst	21-MAY-91	72000
Lorentz	07-FEB-99	50400
Mourgos	16-NOV-99	69600
Rajs	17-OCT-95	42000
Davies	29-JAN-97	37200
Matos	15-MAR-98	31200
Vargas	09-JUL-98	30000
Zlotkey	29-JAN-00	126000
Abel	11-MAY-96	132000
Taylor	24-MAR-98	103200
LAST_NAME	HIRE_DATE	ANNUAL
Grant	24-MAY-99	84000
Whalen	17-SEP-87	52800
Hartstein	17-FEB-96	156000
Fay	17-AUG-97	72000
Higgins	07-JUN-94	144000
Gietz	07-JUN-94	99600

20 rows selected.

Practice 1 (Continued)

If you want an extra challenge, try the following exercises:

10. List all the specific job ids that exist in the organization.

JOB_ID
AC_ACCOUNT
AC_MGR
AD_ASST
AD_PRES
AD_VP
IT_PROG
MK_MAN
MK_REP
SA_MAN
SA_REP
ST_CLERK
ST_MAN

12 rows selected.

Practice 1 (Continued)

11. Select the last name, department id, and hire date for all employees. Display the data as shown:

WHO, WHERE, AND WHEN
King has worked in department 90 since 17-JUN-87
Kochhar has worked in department 90 since 21-SEP-89
De Haan has worked in department 90 since 13-JAN-93
Hunold has worked in department 60 since 03-JAN-90
Ernst has worked in department 60 since 21-MAY-91
Lorentz has worked in department 60 since 07-FEB-99
Mourgos has worked in department 50 since 16-NOV-99
Rajs has worked in department 50 since 17-OCT-95
Davies has worked in department 50 since 29-JAN-97
Matos has worked in department 50 since 15-MAR-98
Vargas has worked in department 50 since 09-JUL-98
Zlotkey has worked in department 80 since 29-JAN-00
Abel has worked in department 80 since 11-MAY-96
Taylor has worked in department 80 since 24-MAR-98
WHO, WHERE, AND WHEN
Grant has worked in department since 24-MAY-99
Whalen has worked in department 10 since 17-SEP-87
Hartstein has worked in department 20 since 17-FEB-96
Fay has worked in department 20 since 17-AUG-97
Higgins has worked in department 110 since 07-JUN-94
Gietz has worked in department 110 since 07-JUN-94

20 rows selected.

2

Restricting and Sorting Data

Objectives

After completing this lesson, you should be able to do the following:

- **Limit the rows retrieved by a query**
- **Sort the rows retrieved by a query**

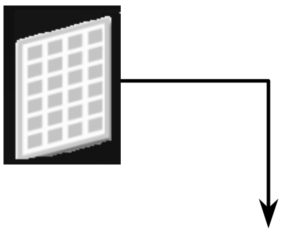
2-2

Lesson Aim

While retrieving data from the database, you may need to restrict the rows of data that are displayed or specify the order in which the rows are displayed. This lesson explains the SQL statements that you use to perform these actions.

Limiting Rows by Using a Restriction

EMPLOYEES TABLE



Retrieve all employees in department 90

EMPLOYEES TABLE

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_	MANAGER_ID	DEPARTMENT_ID
100	Steven	King	SKING	515.123.4567	17-JU		90
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SE	100	90
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JA	100	90

Limiting Rows

The example in the slide displays all the employees in department 90. The set of rows with a value of 90 in the DEPARTMENT_ID column are the only ones returned. This method of restriction is the basis of the WHERE clause in SQL.

Limiting the Rows Selected by a Query

- **Restrict the rows returned by using the `WHERE` clause.**

```
SELECT      [DISTINCT] {*, column|expression [alias],...}
FROM        table
WHERE       condition(s)
[GROUP BY   group_by_expression]
[ORDER BY   column];
```

- **The `WHERE` clause follows the `FROM` clause.**

2-4

Limiting the Rows Selected by a Query

You use a `WHERE` clause to restrict the rows returned by a query. A `WHERE` clause contains a condition that must be met, and it directly follows the `FROM` clause.

In the syntax:

`WHERE`	Restricts the query to rows that meet a condition
`condition`	Is composed of a comparison operator placed between column names, expressions or constants

The `WHERE` clause can compare values in columns, literal values, arithmetic expressions, or functions. The `WHERE` clause consists of three elements:

- Column name
- Comparison operator
- Column name, constant, or list of values

Using the WHERE Clause

```
SELECT last_name, job_id, department_id
FROM   employees
WHERE  department_id = 90 ;
```

LAST_NAME	JOB_ID	DEPARTMENT_ID
King	AD_PRES	90
Kochhar	AD_VP	90
De Haan	AD_VP	90

Using the WHERE Clause

In the example, the SELECT statement retrieves the last name, job ID, and department ID of all employees who work in department 90. Both the DEPARTMENT_ID column and the number 90 are of numeric data type. Data types must match when you are using comparison operators.

You can also restrict restrict the rows returned by a query by columns that are not included in the SELECT clause. The example below restricts the output by the DEPARTMENT_ID column. Observe that this column is not included in the SELECT statement.

```
SELECT last_name, job_id
FROM   employees
WHERE  department_id=90;
```

LAST_NAME	JOB_ID
King	AD_PRES
Kochhar	AD_VP
De Haan	AD_VP

Character Strings and Dates

- Character strings and date values are enclosed in single quotation marks.
- Character values are case sensitive and date values are format sensitive.
- The default date format is `DD-MON-RR`.
  - Allows you to store 21st century dates in the 20th century by specifying only the last two digits of the year.
  - Allows you to store 20th century dates in the 21st century in the same way.

ORACLE

2-6

Character Strings and Dates

Character strings and dates in the `WHERE` clause must be enclosed in single quotation marks (' '). Number constants, however, should not be enclosed in single quotation marks. All character and date searches are case sensitive.

```
SELECT last_name, job_id, department_id, hire_date
FROM employees
WHERE last_name='Abel';
```

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
Abel	SA_REP	80	11-MAY-96

Oracle stores dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds. The default date display is `DD-MON-RR`.

Note: Changing the default date format is covered later in the course.

Comparison Operators

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

ORACLE

Comparison Operators

Use comparison operators in conditions to compare one expression with another. Comparison operators are used in the `WHERE` clause in the following format.

Syntax

… WHERE *expr operator value*

Examples

```
… WHERE department_id = 90
… WHERE salary >= 1500
… WHERE first_name = 'Lex'
```

Using the Comparison Operators with Characters

```
SELECT last_name, manager_id
FROM   employees
WHERE  last_name='Ernst';
```

LAST_NAME	MANAGER_ID
Ernst	103

Using the Comparison Operators with Characters

In the slide, the SELECT statement retrieves the last name and manager ID from the EMPLOYEES table where the employee name is ‘Ernst ‘. Because the character comparison is case sensitive, the value in the WHERE clause must match the case of the employee name exactly.

```
SELECT last_name, department_id
FROM employees
WHERE last_name ='ernst';
no rows selected
```

Column aliases cannot be used with comparison operators.

```
SELECT last_name EMPNAME, department_id
FROM   employees
WHERE  EMPNAME ='Ernst';
```

```
WHERE EMPNAME ='Ernst'
      *
ERROR at line 3:
ORA-00904: invalid column name
```

Other SQL Comparison Operators

Operator	Meaning
`BETWEEN ...AND...`	Between two values (inclusive)
`IN (set)`	Match any of a list of values
`LIKE`	Match a character pattern
`IS NULL`	Is a null value

ORACLE

More Comparison Operators

You can also use these comparison operators in conditions that compare one expression with another. The examples below show sample `WHERE` clauses that use comparison operators.

Examples

```
… WHERE salary BETWEEN 300 and 500
… WHERE salary IN (1500,1300)
… WHERE first_name LIKE 'Nee%'
… WHERE manager_id IS NULL
```

Using the BETWEEN Operator

Use the BETWEEN operator to display rows based on a range of values.

```
SELECT   last_name, salary
FROM     employees
WHERE    salary BETWEEN 9000 AND 17000;
```

Lower limit   Higher limit

LAST_NAME	SALARY
Kochhar	17000
De Haan	17000
Hunold	9000
Zlotkey	10500
Abel	11000
Hartstein	13000
Higgins	12000

7 rows selected.

ORACLE

Using the BETWEEN Operator

Use the BETWEEN operator to display rows based on a range of values. The range that you specify contains a lower limit and an upper limit.

The SELECT statement in the slide returns rows from the EMPLOYEES table for any employee whose salary is between $9000 and $17000.

Note: Values specified with the BETWEEN operator are inclusive. You must specify the lower limit first. Observe in the output that the records with the salary values of 9000 and 17000 are included in the output.

Using the `IN` Operator

Use the `IN` operator to test for values in a set.

```
SELECT employee_id, last_name,
       salary,      manager_id
FROM   employees
WHERE  manager_id IN (100, 102, 103);
```

EMPLOYEE_ID	LAST_NAME	SALARY	MANAGER_ID
104	Ernst	6000	103
107	Lorentz	4200	103
103	Hunold	9000	102
101	Kochhar	17000	100
102	De Haan	17000	100
124	Mourgos	5800	100
149	Zlotkey	10500	100
201	Hartstein	13000	100

8 rows selected.

ORACLE

Using the `IN` Operator

Use the `IN` operator to test for values in a specified set.

The example in the slide displays the employee ID, name, salary, and manager ID of all the employees whose manager ID is 100, 102, or 103.

The `IN` operator can be used with any data type. The following example returns a row from the `EMPLOYEES` table for an employee whose department number is included in the set of department numbers in the `WHERE` clause.

```
SELECT  employee_id,  last_name, manager_id, department_id
FROM    employees
WHERE   department_id IN (60,50);
```

Note: If characters or dates are used in the set, they must be enclosed in single quotation marks (' ').

Using the IN Operator with Strings

Use the IN operator to test for values in a set of strings.

```
SELECT last_name, department_id, hire_date
FROM   employees
WHERE  last_name IN ('De Haan','Kochhar') ;
```

LAST_NAME	DEPARTMENT_ID	HIRE_DATE
Kochhar	90	21-SEP-89
De Haan	90	13-JAN-93

ORACLE

Using the IN Operator (continued)

The example in the slide retrieves details for employees whose name matches 'De Haan ' or 'Kochhar '.

The WHERE clause uses the IN operator to check for the occurrence of the employee last name in a set of names: De Haan, Kochhar. Note that the set of names is in mixed case.

The following example below retrieves the last names of all employees who are either salesmen or marketing managers.

```
SELECT last_name
FROM   employees
WHERE  job_id IN ('SA_MAN','MK_MAN');
```

LAST_NAME
Hartstein
Zlotkey

.

Using the LIKE Operator

- **Use the LIKE operator to perform wildcard searches of valid search string values.**
- **Search conditions can contain either literal characters or numbers.**
  - **The % symbol denotes zero or many characters**
  - **The _ symbol denotes one character**

```
SELECT last_name
FROM  employees
WHERE last_name LIKE 'H%';
```

LAST_NAME
Hartstein
Higgins
Hunold

Using the LIKE Operator

You may not always know the exact search condition. You can select rows that match a character pattern by using the LIKE operator. The character pattern matching operation is called a wildcard search. You can use two symbols to construct the search string: the percentage sign and the underscore.

Symbol	Description
%	Represents any sequence of zero or more characters
_	Represents any single character

The SELECT statement in the slide returns the employee name from the EMPLOYEES table for an employee whose last name begins with an uppercase H. Names that begin with a lowercase h are not returned.

The following example displays the last names, salaries, and jobs of all employees whose job ID begins with uppercase M.

```
SELECT  last_name, salary, job_id
FROM employees
WHERE job_id  LIKE 'M%';
```

Using the LIKE Operator

- **You can combine pattern matching characters.**

```
SELECT last_name
FROM employees
WHERE first_name LIKE '_a%';
```

LAST_NAME
Matos
Fay

- **Use the ESCAPE identifier to search for % and _.**

Combining Wildcard Characters

You can use the % and _ symbols in any combination with literal characters. The example in the slide displays the names of all employees whose name has an 'a' as the second character.

The ESCAPE Option

When you need to have an exact match for the actual % and _ characters, use the ESCAPE option. You specify the ESCAPE character using this option. If you have 'K_' appearing as part of a job ID, you may search for it using the following SQL statement:

```
SELECT last_name, job_id
FROM employees
WHERE job_id LIKE '%K\_%' ESCAPE'\';
```

LAST_NAME	JOB_ID
Hartstein	MK_MAN
Fay	MK_REP

The ESCAPE option identifies the backslash as the escape character. In the pattern, the escape character precedes the underscore. This causes the Oracle server to interpret the underscore literally. If escape character is not specified, there is no default escape character.

Using the IS NULL Operator

Test for null values with the `IS NULL` operator.

```
SELECT  last_name, manager_id
FROM    employees
WHERE   manager_id IS NULL;
```

LAST_NAME	MANAGER_ID
King	

Using the IS NULL Operator

The `IS NULL` operator tests for values that are null. A null value means that the value is unavailable, unassigned, unknown, or inapplicable. You cannot test with (=) because a null value cannot be equal or unequal to any value. The example in the slide retrieves the names of all employees who do not have a manager.

To display the last name, job ID, and commission for all employees who are not entitled to get a commission, use the following statement:

```
SELECT last_name,  job_id, commission_pct
FROM  employees
WHERE  commission_pct IS  NULL;
```

LAST_NAME	JOB_ID	COMMISSION_PCT
King	AD_PRES	
Kochhar	AD_VP	
De Haan	AD_VP	
Hunold	IT_PROG	

■ ■ ■

16 rows selected.

Logical Operators

Operator	Meaning
`AND`	Returns `TRUE` if *both* component conditions are `TRUE`
`OR`	Returns `TRUE` if *either* component condition is `TRUE`
`NOT`	Returns `TRUE` if the following condition is `FALSE`

ORACLE

2-16

Logical Operators

A logical operator combines the result of two or more component conditions to produce additional or alternative conditions or to invert the result of a single condition. Three logical operators are available in SQL:

- `AND`
- `OR`
- `NOT`

All the examples so far have specified only one condition in the `WHERE` clause. You can use the `AND` and `OR` operators to specify several conditions in one `WHERE` clause.

Using the AND Operator

AND requires both conditions to be TRUE.

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 1100  AND job_id='ST_CLERK' ;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

Using the AND Operator

With the AND operator, both conditions must be true for the row to be selected. In the example in the slide, the details of any employee who has a job ID of ST_CLERK and earns $1100 or more is retrieved.

Note: All character searches are case sensitive. No rows are returned if ST_CLERK is not in all uppercase letters. You must enclose character and date strings in single quotation marks.

AND Truth Table

The following table shows the results of combining two expressions with AND:

AND	**TRUE**	**FALSE**	**UNKNOWN**
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

Using the AND Operator

AND requires both conditions to be TRUE.

```
SELECT last_name, manager_id,
       salary,    department_id
FROM   employees
WHERE  salary > 5000
AND    department_id = 60;
```

LAST_NAME	MANAGER_ID	SALARY	DEPARTMENT_ID
Hunold	102	9000	60
Ernst	103	6000	60

ORACLE

2-18

Using the AND Operator (continued)

You may combine any two (or more) conditions with the AND operator.

The following example uses the IN and = operators to retrieve all employees whose job id is AD_VP and who work in department number 90 or 60.

```
SELECT last_name, department_id,job_id
FROM   employees
WHERE  department_id in (90,60)
AND    job_id = 'AD_VP';
```

LAST_NAME	DEPARTMENT_ID	JOB_ID
Kochhar	90	AD_VP
De Haan	90	AD_VP

Using the OR Operator

OR requires either condition to be TRUE.

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 12000
OR     job_id = 'ST_CLERK';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500
201	Hartstein	MK_MAN	13000
205	Higgins	AC_MGR	12000

9 rows selected.

ORACLE

2-19

Using the OR Operator

The OR operator selects a row for which either condition is true. Therefore an employee who has a job title of ST_CLERK or earns $12,000 or more is selected. Observe that the value 12000 is included in the results set.

OR Truth Table

The following table shows the results of combining two expressions with OR:

OR	**TRUE**	**FALSE**	**UNKNOWN**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Using the OR Operator

OR requires either condition to be TRUE.

```
SELECT   last_name, department_id,manager_id
FROM     employees
WHERE    department_id = 60
OR       manager_id    = 124;
```

LAST_NAME	DEPARTMENT_ID	MANAGER_ID
Rajs	50	124
Davies	50	124
Matos	50	124
Vargas	50	124
Hunold	60	102
Ernst	60	103
Lorentz	60	103

7 rows selected.

ORACLE

2-20

Using the OR Operator (continued)

The example in the slide selects any employee who works in department ID 60 or whose manager ID is 124.

Using the NOT Operator

```
SELECT first_name, job_id
FROM   employees
WHERE  job_id NOT IN
               ('ST_CLERK','SA_REP','IT_PROG');
```

FIRST_NAME	JOB_ID
Steven	AD_PRES
Neena	AD_VP
Lex	AD_VP
Kevin	ST_MAN
Eleni	SA_MAN
Jennifer	AD_ASST
Michael	MK_MAN
Pat	MK_REP
Shelley	AC_MGR
William	AC_ACCOUNT

10 rows selected.

ORACLE

2-21

Using the NOT Operator

The example in the slide displays the last name and job ID of all employees whose job ID is not ST_CLERK, SA_REP, or IT_PROG.

The following table shows the result of applying the NOT operator to a condition:

NOT Truth Table

NOT	TRUE	FALSE	UNKNOWN
	FALSE	TRUE	UNKNOWN

Note: The NOT operator can also be used with other SQL operators such as BETWEEN, LIKE, and NULL:

```
... WHERE   salary   NOT BETWEEN 1000 AND 1500
... WHERE   first_name NOT LIKE '%A%'
... WHERE   commission_pct IS  NOT NULL
```

Using the NOT Operator

```
SELECT employee_id, last_name,department_id,
       manager_id
FROM   employees
WHERE  manager_id NOT LIKE '10%' ;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	MANAGER_ID
141	Rajs	50	124
142	Davies	50	124
143	Matos	50	124
144	Vargas	50	124
174	Abel	80	149
176	Taylor	80	149
178	Grant		149
202	Fay	20	201
206	Gietz	110	205

9 rows selected.

ORACLE

2-22

Using the NOT Operator with the LIKE Operator

The expression in the slide returns the employee ID, last name, department ID, and manager ID of the employees whose manager ID does not begin with 10.

Using the NOT Operator

```
SELECT employee_id, salary, manager_id
FROM employees
WHERE salary NOT BETWEEN 4000 AND 15000;
```

EMPLOYEE_ID	SALARY	MANAGER_ID
100	24000	
101	17000	100
102	17000	100
141	3500	124
142	3100	124
143	2600	124
144	2500	124

7 rows selected.

Using the NOT Operator with the BETWEEN Operator

The expression in the slide returns the employee ID, salary, and manager ID of all employees whose salary is not between $4000 and $15000. In other words, the expression is true if an employee earns less than $4000 or more than $15000.

Using the NOT Operator

```
SELECT last_name, salary AS
       "Salary Before Commission",commission_pct
FROM   employees
WHERE  commission_pct IS NOT NULL ;
```

LAST_NAME	Salary Before Commission	COMMISSION_PCT
Zlotkey	10500	.2
Abel	11000	.3
Taylor	8600	.2
Grant	7000	.15

ORACLE

2-24

Using the NOT Operator with the IS NULL Operator

The expression in the slide evaluates to true if the value in the COMMISSION_PCT column is not a null value. In other words, the expression is true if an employee earns a commission.

Rules of Precedence

Order Evaluated	Operator
1	All comparison operators
2	`NOT`
3	`AND`
4	`OR`

Use parentheses to override rules of precedence.

2-25

Rules of Precedence

Precedence is the order in which Oracle evaluates different operators in the same expression. When evaluating an expression containing multiple operators, Oracle evaluates operators with higher precedence before evaluating those with lower precedence. Oracle evaluates operators with equal precedence from left to right within an expression.

Rules of Precedence

```
SELECT   last_name,manager_id, job_id
FROM     employees
WHERE    manager_id = 100
OR       manager_id = 124
AND      job_id = 'ST_CLERK';
```

LAST_NAME	MANAGER_ID	JOB_ID
Rajs	124	ST_CLERK
Davies	124	ST_CLERK
Matos	124	ST_CLERK
Vargas	124	ST_CLERK
Kochhar	100	AD_VP
De Haan	100	AD_VP
Mourgos	100	ST_MAN
Zlotkey	100	SA_MAN
Hartstein	100	MK_MAN

9 rows selected.

2-26

Example of Precedence of the `AND` Operator

In the example in the slide, there are effectively two conditions, either of which can be met:

- The first condition is that `MANAGER_ID` is 100
- The second condition is that `MANAGER_ID` is 124 and `JOB_ID` is ST_CLERK.

Observe that the results set contains nine records.

Rules of Precedence

Use parentheses to force priority.

```
SELECT   last_name,manager_id, job_id
FROM     employees
WHERE    (manager_id = 100
OR       manager_id = 124)
AND      job_id = 'ST_CLERK';
```

LAST_NAME	MANAGER_ID	JOB_ID
Rajs	124	ST_CLERK
Davies	124	ST_CLERK
Matos	124	ST_CLERK
Vargas	124	ST_CLERK

2-27

Using Parentheses

In the example in the slide, there are two conditions both having to be met:

- The first condition is that `MANAGER_ID` is 100 or 124
- The second condition is that `JOB_ID` is ST_CLERK.

Observe that the results set contains only four records as against the example in the previous page that retrieved nine records.

ORDER BY Clause

- **Sort rows with the ORDER BY clause**
  - **ASC: ascending order, default**
  - **DESC: descending order**
- **The ORDER BY clause comes last in the SELECT statement.**

```
SELECT   last_name, job_id, department_id
FROM     employees
ORDER BY department_id ;
```

LAST_NAME	JOB_ID	DEPARTMENT_ID
Whalen	AD_ASST	10
Hartstein	MK_MAN	20
Fay	MK_REP	20
Mourgos	ST_MAN	50
Rajs	ST_CLERK	50
Davies	ST_CLERK	50

...

20 rows selected.

2-28

The ORDER BY Clause

The order of rows returned in a query result is undefined. You can use the ORDER BY clause to sort the rows. You must place the ORDER BY clause last. You can specify a column, an expression or an alias to sort by.

Syntax

```
SELECT     expr
FROM       table
[WHERE     condition (s)]
[ORDER BY  {column, expr, alias} [ASC|DESC]];
```

In the syntax:

ORDER BY	Specifies the order in which the retrieved rows are displayed
ASC	Orders the rows in ascending order. This is the default order
DESC	Orders the rows in descending order

Note: If you don't use an ORDER BY clause, the sort order is undefined, and the Oracle Server may not always fetch rows in the same order for the same query. Use the ORDER BY clause to display the rows in a specific order.

Sorting in Descending Order

```
SELECT   last_name, job_id, department_id
FROM     employees
ORDER BY department_id DESC ;
```

LAST_NAME	JOB_ID	DEPARTMENT_ID
Grant	SA_REP	
Higgins	AC_MGR	110
Gietz	AC_ACCOUNT	110
King	AD_PRES	90
Kochhar	AD_VP	90
De Haan	AD_VP	90
Zlotkey	SA_MAN	80
Taylor	SA_REP	80
Abel	SA_REP	80

...

20 rows selected.

ORACLE

2-29

Default Ordering of Data

The default sort order is ascending:

- Numeric values are displayed with the lowest values first: for example, 1 to 999.
- Date values are displayed with the earliest value first: for example, 01-JAN-1992 before 01-JAN-1995.
- Character values are displayed in alphabetical order: for example, A first and Z last.
- Null values are displayed last for ascending sequences and first for descending sequences.

Reversing the Default Order

To reverse the order in which rows are displayed, use the keyword `DESC` after the column name in the `ORDER BY` clause. The example in the slide sorts the result beginning with the highest department ID.

Sorting by Column Alias

```
SELECT   employee_id, last_name,salary*12 annsal
FROM     employees
ORDER BY annsal ;
```

EMPLOYEE_ID	LAST_NAME	ANNSAL
144	Vargas	30000
143	Matos	31200
142	Davies	37200
141	Rajs	42000
107	Lorentz	50400
200	Whalen	52800
124	Mourgos	69600
104	Ernst	72000
202	Fay	72000
178	Grant	84000
206	Gietz	99600

...

20 rows selected.

ORACLE

Sorting by Column Aliases

You can use a column alias in the `ORDER BY` clause. The example in the slide sorts the data by annual salary.

Sorting by Multiple Columns

The order of an ORDER BY list is the order of the sort.

```
SELECT  last_name, department_id, salary
FROM   employees
ORDER BY department_id, salary DESC ;
```

LAST_NAME	DEPARTMENT_ID	SALARY
Whalen	10	4400
Hartstein	20	13000
Fay	20	6000
Mourgos	50	5800
Rajs	50	3500
Davies	50	3100
Matos	50	2600
Vargas	50	2500
Hunold	60	9000
Ernst	60	6000
Lorentz	60	4200
Abel	80	11000

...

20 rows selected.

ORACLE

2-31

Sorting by Multiple Columns

You can sort query results by more than one column.

In the `ORDER BY` clause, specify the columns and separate the column names with commas. If you want to reverse the order of a column, specify `DESC` after its name.

Example:

Display last name and salary of all employees. Order the result by department ID in ascending and then salary in descending order.

```
SELECT last_name,  salary
FROM  employees
ORDER BY department_id, salary DESC;
```

Sorting by a Column Not in the SELECT List

```
SELECT   last_name, department_id
FROM     employees
ORDER BY salary ;
```

LAST_NAME	DEPARTMENT_ID
Vargas	50
Matos	50
Davies	50
Rajs	50
Lorentz	60
Whalen	10
Mourgos	50
Ernst	60
Fay	20
Grant	
Gietz	110
Taylor	80

...

20 rows selected.

Sorting by a Column Not in the SELECT List

You can sort by columns that are not included in the SELECT clause. The example in the slide lists the output in ascending order of salary even though the SALARY column does not appear in the SELECT statement.

The statement is repeated below with the SALARY column included in the SELECT list. Comparison verifies that the order of both results is the same.

```
SELECT   last_name, department_id, salary
FROM     employees
ORDER BY salary;
```

LAST_NAME	DEPARTMENT_ID	SALARY
Vargas	50	2500
Matos	50	2600
Davies	50	3100
Rajs	50	3500

...

20 rows selected.

Summary

```
SELECT       [DISTINCT] {*, column|expression [alias],...}
FROM         table
WHERE        condition(s)
[GROUP BY    group_by_expression]
[ORDER BY    {column, expr, alias} [ASC|DESC]] ;
```


Summary

In this lesson, you learned about restricting and sorting rows returned by the `SELECT` statement. You also learned how to use various operators.

Practice 2 Overview

This practice covers the following topics:

- Selecting data and changing the order of rows displayed
- Restricting rows by using the `WHERE` clause

Practice 2 Overview

This practice contains a variety of exercises using the `WHERE` clause and the `ORDER BY` clause.

Practice 2

1. You can order by a column that you have not selected.

   True/False

2. The following statement will execute successfully.

   True/False

```
SELECT *
FROM   employees
WHERE  salary*12=9600;
```

3. Display the last name of the employee with the employee ID 104.

LAST_NAME
Ernst

4. Display the last name, manager ID, and salary for all employees in department 20.

LAST_NAME	MANAGER_ID	SALARY
Hartstein	100	13000
Fay	201	6000

5. Display the last name and hire date of all employees whose last name begins with the letter H.

LAST_NAME	HIRE_DATE
Hartstein	17-FEB-96
Higgins	07-JUN-94
Hunold	03-JAN-90

Practice 2 (continued)

6. Display the last name, manager ID, and salary for all employees whose salary is in the range of $6000 through $8000.

LAST_NAME	MANAGER_ID	SALARY
Ernst	103	6000
Grant	149	7000
Fay	201	6000

7. Display the employee ID and last name for all clerks (`JOB_ID = ST_CLERK`) and who work for manager 100 or 124.

EMPLOYEE_ID	LAST_NAME
141	Rajs
142	Davies
143	Matos
144	Vargas

8. Display the employee ID, last name, and manager ID for all employees whose salary is greater than $2500 and who work in department 50.

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
124	Mourgos	100
141	Rajs	124
142	Davies	124
143	Matos	124

Practice 2 (continued)

9. Display the last names and salary for all employees who work for the manager with the manager ID 124, starting with the employee with the highest salary and ending with the employee with the lowest salary.

LAST_NAME	SALARY
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500

10. Display the last name, job ID, and salary for all non sales employees who are earning less than $2000 or more than $15000.

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000
Kochhar	AD_VP	17000
De Haan	AD_VP	17000

3 Single-Row Number and Character Functions

Objectives

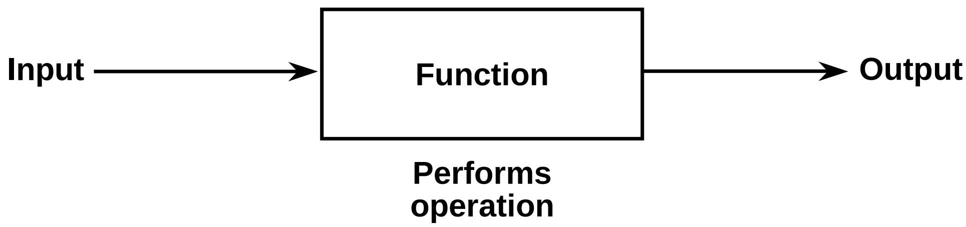
After completing this lesson, you should be able to do the following:

- **Describe the various types of functions available in SQL**
- **Use single-row character and number functions in `SELECT` statements**
- **Handle null values in arithmetic expressions**

Lesson Aim

You use functions to manipulate data values. Functions make the basic query block more powerful. This is the first of two lessons that explore functions. This lesson focuses on single-row character and number functions.

How does a Function Work?

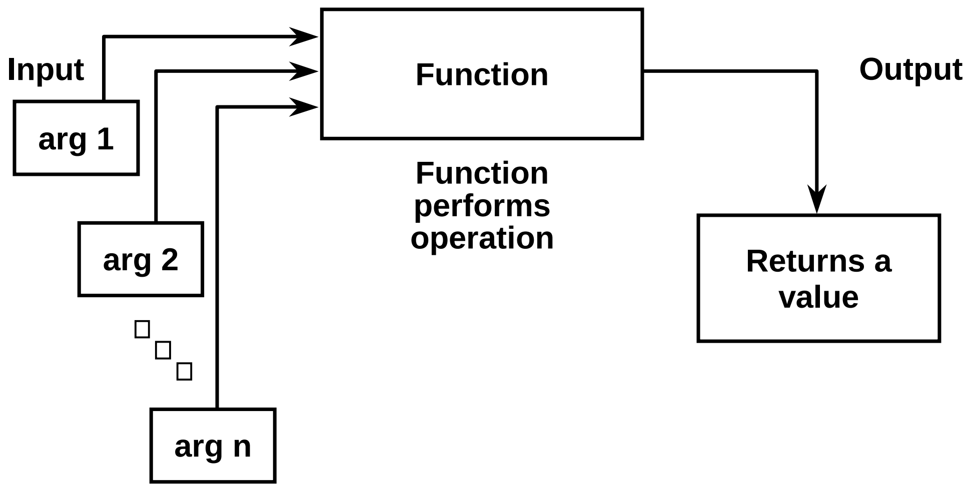


3-3

How does a Function Work?

A function performs an operation on some input that it receives and returns the result of the operation.

How SQL Functions Work



SQL Functions

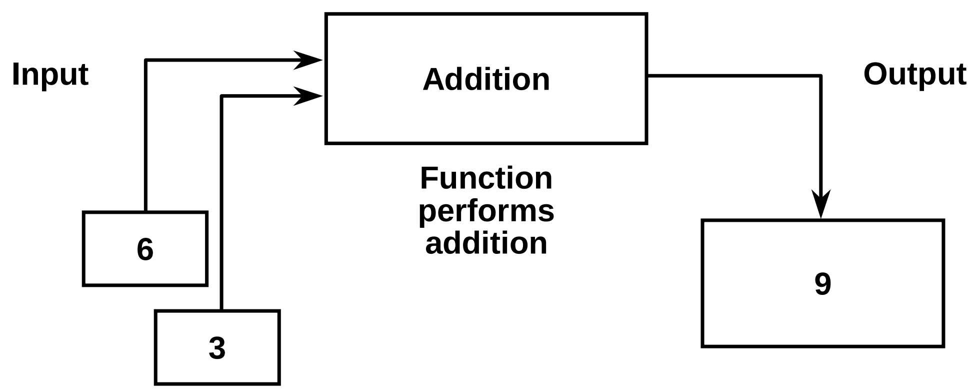
Functions are a very powerful feature of SQL. You use them to:

- Perform calculations on data
- Modify individual data items
- Manipulate output for groups of rows
- Format dates and numbers for display
- Convert column data types

SQL functions can accept arguments and always return a single value.

Note: Most of the functions described in this lesson are specific to Oracle's version of SQL.

Example of a Function

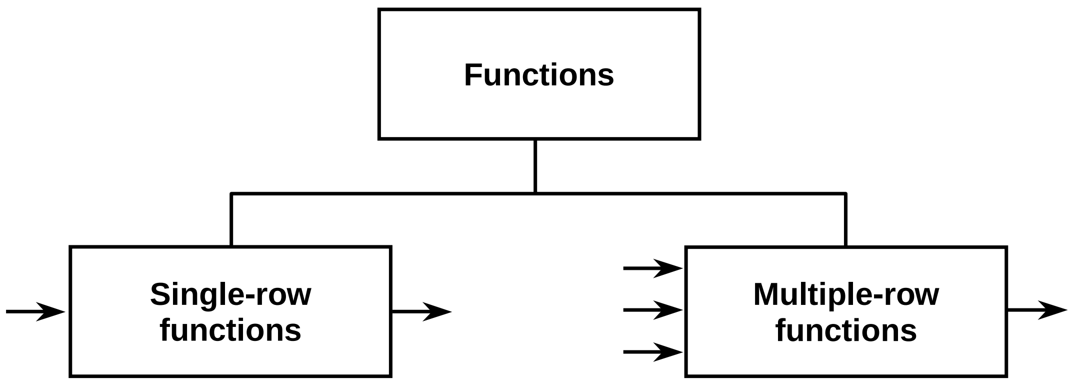


Example of a Function

The `ADDITION` function is a simple example of a function. The `ADDITION` function takes several numbers, adds them all together, and gives a result. In this example:

- The function is the addition operator.
- The input to the function is a list of numbers.
- The output is the sum of the numbers.

Two Types of SQL Functions



3-6

Types of SQL Functions

There are two distinct types of functions:

- Single-row functions
- Multiple-row functions

Single-Row Functions

Single-row functions operate on single rows only and return one result per row. There are different types of single-row functions. This lesson covers character and number functions.

Multiple-Row Functions

Multiple-row functions manipulate groups of rows to give one result per group of rows. Multiple-row functions are covered later in this course.

For a complete list of available functions and syntax, see *Oracle Server SQL Reference*.

Single-Row Functions

- Manipulate data items
- Accept arguments and return one value
- Act on each row returned
- Return one result per row
- Can modify the data type
- Can be nested

3-7

How Single-Row Functions Work?

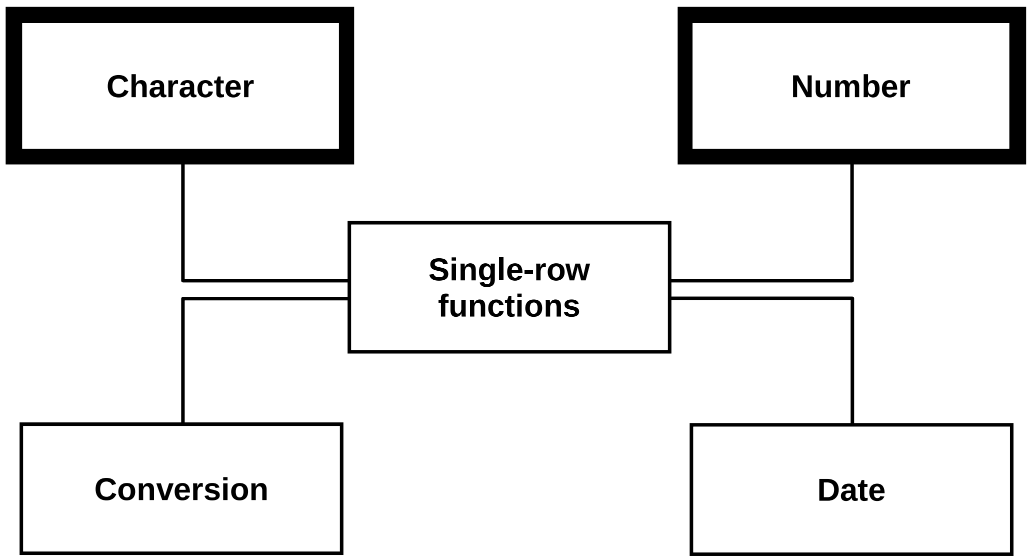
Single-row functions are used to manipulate data items. They accept one or more arguments and return one value for each row returned by the query. An argument can be:

- A user-supplied constant
- A variable value
- A column name
- An expression

Features of Single-Row Functions

- Single-row functions act on each row returned in the query.
- Single-row functions return one result per row.
- Single-row functions can return a data value of a different type than that referenced.
- Single-row functions can accept one or more arguments.
- Single-row functions can be used in `SELECT`, `WHERE`, and `ORDER BY` clauses.
- Single-row functions can be nested.

Single-Row Functions



Single-Row Functions

This lesson covers the following single-row functions:

- Character functions: Accept character input and can return both character and number values
- Number functions: Accept numeric input and return numeric values

The remaining single-row functions are covered in the next lesson.

Calling a Function in SQL

```
function_name ({col, expr}[ , arg1[ , arg2])
```

- `function_name` **name of the function**
- `column` **any named database column**
- `expression` **any character string or calculated** `expression`
- `Arg1,arg2` **any argument to be used by the function**

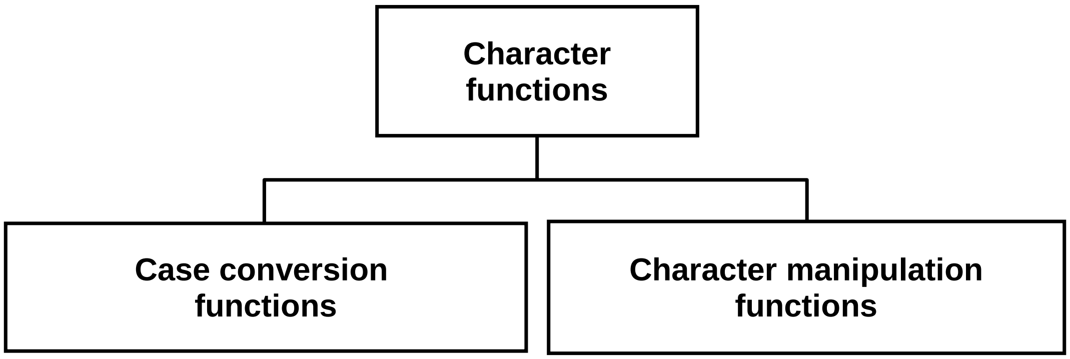
3-9

Calling a Function in SQL

SQL functions are built into Oracle and are available for use in various SQL statements. If you call a SQL function with an argument of a datatype other than the datatype expected by it, Oracle implicitly converts the argument to the expected datatype before performing the function.

In the syntax diagrams for SQL functions, arguments are indicated by their datatypes. The slide displays a generic format for the SQL functions.

Character Functions



LOWER

UPPER

INITCAP

3-10

Character Functions

Single-row character functions accept character data as input and can return both character and number values. Character functions can be divided into:

- Case conversion functions: Convert the case of character strings
- Character manipulation functions: Perform operations on strings such as creating a substring, instring and so on.

This lesson covers the case conversion functions. Character manipulation functions are not covered in this course.

Note: This list is a subset of the available character functions.

For more information, see *Oracle Server SQL Reference, "Character Functions."*

Case Conversion Functions

Convert the case for character strings

Function	Result
`LOWER('SQL Course')`	sql course
`UPPER('SQL Course')`	SQL COURSE
`INITCAP('SQL Course')`	Sql Course

Case Conversion Functions

`LOWER`, `UPPER`, and `INITCAP` are the three case conversion functions.

`LOWER:`	Converts a mixed case or uppercase character string to lowercase
`UPPER:`	Converts a mixed case or lowercase character string to uppercase
`INITCAP:`	Converts the first letter of each word to uppercase and the remaining letters to lowercase

Using Case Conversion Functions

Display the last names of all employees in uppercase.

```
SELECT UPPER(last_name) as "LAST NAME"
FROM   employees;
```

LAST NAME
KING
KOCHHAR
DE HAAN
HUNOLD
ERNST
LORENTZ
MOURGOS
RAJS
DAVIES
MATOS
VARGAS
ZLOTKEY
ABEL

...

20 rows selected.

3-12

Using Case Conversion Functions

The example in the slide displays the last names of all employees in uppercase letters. Observe the usage of the double quotes in the alias, `LAST NAME`. Usage of the double quotes preserves the case of the alias and helps include a space in the alias name.

Using Case Conversion Functions

Display the employee ID, last name, and department ID for employee Taylor.

```
SELECT employee_id, last_name, department_id
FROM employees
WHERE last_name = 'taylor';
```

```
no rows selected
```

```
SELECT employee_id, last_name, department_id
FROM employees
WHERE  LOWER(last_name)= 'taylor';
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
176	Taylor	80

3-13

Using Case Conversion Functions (continued)

The example in the slide displays the employee ID, last name, and department ID of the employee Taylor.

The `WHERE` clause in the first SQL statement specifies the last name as 'taylor'. Because all of the data in the `EMPLOYEES` table is stored in initcap case, the name 'taylor' does not find a match in the `EMPLOYEES` table and no rows are selected.

The `WHERE` clause in the second SQL statement specifies that the last name column in the `EMPLOYEES` table is converted to lowercase and compared to 'taylor'. Because both the names are in lower case now, a match is found and one row is selected. You can produce the same result by rewriting the `WHERE` clause in the following manner:

```
… WHERE last_name = 'Taylor'
```

Note that the name in the output appears as it was stored in the database.

Number Functions

- **ROUND: Rounds value to the specified number of decimal places**

```
ROUND(45.926, 2) ⟶ 45.93
```

- **TRUNC: Truncates value to the specified number of decimal places**

```
TRUNC(45.926, 2) ⟶ 45.92
```

- **MOD: Returns remainder of division**

```
MOD(1600, 300) ⟶ 100
```


Number Functions

Number functions accept numeric input and return numeric values.

Function	Purpose
ROUND(column|expression, *n*)	Rounds the column, expression, or value to *n* decimal places. If *n* is omitted, the column, expression or value is rounded to 0 decimal places. If *n* is negative, numbers to the left of the decimal point are rounded.
TRUNC(column|expression,*n*)	Truncates the column, expression, or value to *n* decimal places or if *n* is omitted, no decimal places. If *n* is negative, numbers to the left of the decimal point are truncated to zero.
MOD(*m*,*n*)	*m* is divided by *n* a whole number of times.

Note: This list is a subset of the available number functions.

For more information, see *Oracle Server SQL Reference, "Number Functions."*

Using the ROUND Function

Display the monthly commission of salesmen rounded to hundredths and to no decimal places.

```
SELECT ROUND(commission_pct/12,2),
       ROUND(commission_pct/12,0)
FROM   employees
WHERE  job_id = 'SA_MAN';
```

ROUND(COMMISSION_PCT/12,2)	ROUND(COMMISSION_PCT/12,0)
.02	0

3-15

The ROUND Function

The ROUND function rounds the column, expression, or value to n decimal places. If the second argument is 0 or is missing, the value is rounded to zero decimal places. If the second argument is 2, the value is rounded to two decimal places, or to hundredths and to a whole number. If the second argument is -1, the value is rounded to one decimal place to the left, or to the nearest multiple of ten.

More Examples of ROUND

```
SELECT ROUND(16.746), ROUND(16.746,1), ROUND(16.746,2),
       ROUND(16.746,-1)
FROM dual;
```

ROUND(16.746)	ROUND(16.746,1)	ROUND(16.746,2)	ROUND(16.746,-1)
17	16.7	16.75	20

Using the TRUNC Function

Display the monthly commission of salesmen truncated to hundredths, and to no decimal places.

```
SELECT TRUNC(commission_pct/12,2),
       TRUNC(commission_pct/12,0)
FROM   employees
WHERE  job_id='SA_MAN';
```

TRUNC(COMMISSION_PCT/12,2)	TRUNC(COMMISSION_PCT/12,0)
.01	0

ORACLE

The TRUNC Function

The TRUNC function truncates the column, expression, or value to n decimal places.

The TRUNC function and ROUND function work with similar arguments. If the second argument is 0 or is missing, the value is truncated to zero decimal places. If the second argument is 2, the value is truncated to two decimal places, or to hundredths and to a whole number. If the second argument is -1, the value is truncated to one decimal place to the left, or to the preceding multiple of ten.

More Examples of TRUNC

```
SELECT TRUNC(16.746), TRUNC(16.746,1), TRUNC(16.746,2),
       TRUNC(16.746,-1)FROM dual;
```

TRUNC(16.746)	TRUNC(16.746,1)	TRUNC(16.746,2)	TRUNC(16.746,-1)
16	16.7	16.74	10

The ROUND and TRUNC functions can also be used with date functions. This subject is covered later in the course.

Note: DUAL is a one-column, one-row table that is used as a dummy table.The DUAL table is covered later in this course.

Defining a Null Value

- **A null is a value that is unavailable, unassigned, unknown, or inapplicable.**
- **A null is not the same as zero or a blank space.**

```
SELECT  last_name, job_id, commission_pct
FROM employees;
```

LAST_NAME	JOB_ID	COMMISSION_PCT
King	AD_PRES	
Kochhar	AD_VP	
De Haan	AD_VP	
Hunold	IT_PROG	
Ernst	IT_PROG	
Lorentz	IT_PROG	
Mourgos	ST_MAN	
Rajs	ST_CLERK	
Davies	ST_CLERK	
Matos	ST_CLERK	
Vargas	ST_CLERK	
Zlotkey	SA_MAN	.2
Abel	SA_REP	.3

...

20 rows selected.

ORACLE

Null Values

If a row lacks the data value for a particular column, the value is said to be null, or to contain null.

A null value is a value that is unavailable, unassigned, unknown, or inapplicable. It is not the same as zero or a space. Zero is a number, and a space is a character.

Columns of any data type can contain null values, unless the creator of the column defined it as `NOT NULL` or as `PRIMARY KEY.`

In the `COMMISSION_PCT` column in the `EMPLOYEES` table, notice that only a salesman can earn commission. Other employees are not entitled to earn commission. A null value represents this fact.

Null Values in Arithmetic Expressions

Arithmetic expressions that contain a null value evaluate to null.

```
SELECT last_name NAME,job_id,
       12*salary*(1+commission_pct)
FROM   employees;
```

NAME	JOB_ID	12*SALARY*(1+COMMISSION_PCT)
King	AD_PRES	
Kochhar	AD_VP	
De Haan	AD_VP	
Hunold	IT_PROG	
Ernst	IT_PROG	
Lorentz	IT_PROG	
Mourgos	ST_MAN	
Rajs	ST_CLERK	
Davies	ST_CLERK	
Matos	ST_CLERK	
Vargas	ST_CLERK	
Zlotkey	SA_MAN	151200
Abel	SA_REP	171600

...

20 rows selected.

ORACLE

3-18

Null Values (continued)

If any column value in an arithmetic expression is null, the result is null. If you attempt to perform division with zero, you get an error. However, if you divide a number by null, the result is a null or unknown.

In the example in the slide, the `12*salary*(1+commission_pct)`expression is intended to calculate the annual remuneration for each employee. However, several employees (for example, King) show no value in the `12*salary*(1+commission_pct)` column. This is because King is not a salesman and does not get any commission. Because the `COMMISSION_PCT` column in the arithmetic expression is null, the result is null.

Note that for employee Abel, who is a salesman, the expression gives a valid annual remuneration amount.

Note: For more information, see *Oracle Server SQL Reference, "Elements of SQL*."

The NVL Function

```
NVL (expr1, expr2)
```

- **Use the `NVL` function to force a value where a null would otherwise appear**
- **`NVL` can be used with date, character, and number data types.**
- **Data types must match. For example:**
  - `NVL(commission+pct,0)`
  - `NVL(hire_date,'01-JAN-97')`
  - `NVL(job_id,'no job yet')`

The NVL function

The `NVL` function provides a mechanism to deal with null values.

- The `NVL` function requires two arguments:
  - An expression
  - A non null value
- You can use the `NVL` function to convert a null number, date, or character string to another number, date, or character string as long as the data types match.

In the syntax shown in the slide:

`expr1` is the source value or expression that may contain null
`expr2` is the target value for converting null

Using the NVL Function to Handle Null Values

```
SELECT last_name, job_id,
       12*salary*(1+NVL(commission_pct,0))
FROM   employees;
```

LAST_NAME	JOB_ID	12*SALARY*(1+NVL(COMMISSION_PCT,0))
King	AD_PRES	288000
Kochhar	AD_VP	204000
De Haan	AD_VP	204000
Hunold	IT_PROG	108000
Ernst	IT_PROG	72000
Lorentz	IT_PROG	50400
Mourgos	ST_MAN	69600
Rajs	ST_CLERK	42000
Davies	ST_CLERK	37200
Matos	ST_CLERK	31200
Vargas	ST_CLERK	30000
Zlotkey	SA_MAN	151200
Abel	SA_REP	171600

...

20 rows selected.

ORACLE

Using the NVL Function to Handle Null Values

Only employees with job titles of SA_MAN or SA_REP show a value in the COMMISSION_PCT column of the EMPLOYEES table. In other words, only salesmen earn commission. All other employees have a null value in the COMMISSION_PCT column.

To achieve the correct result for King (and all other employees who do not earn commission), you must convert the null value to a number before applying the arithmetic operator. In the example in the slide, the NVL function is used to convert a COMMISSION_PCT value of NULL to zero.

Summary

Use functions to:

- **Perform calculations on data by using number functions or using character functions**
- **Modify individual data items**
- **Use single-row functions to manipulate:**
  - **Character data: `LOWER`, `UPPER`, `INITCAP`**
  - **Number data: `ROUND`, `TRUNC`, `MOD`**
- **Handle null values**
  - **`NVL`**

ORACLE

3-21

Summary

Single-row functions can manipulate:

- Character data: `LOWER`, `UPPER`, `INITCAP`
- Number data: `ROUND`, `TRUNC`, `MOD`
- `NULL` data: `NVL`

Practice 3 Overview

This practice covers the following topics:

- Using number functions to alter the display of numeric data
- Using character functions to alter the display of character data
- Using the `NVL` function to handle `NULL` values

3-22

Practice 3 Overview

This practice gives you a chance to use character, number and `NVL` functions in the `SELECT` statement.

Practice 3

1. Single-row functions work on many rows to produce a single result.

   True/False

2. Display the last name and salary plus $600 for all employees in department 20. The name should be displayed in upper case.

NAME	SALARY+600
HARTSTEIN	13600
FAY	6600

3. Display the employee ID, last name, and salary increased by 15% and expressed as a whole number, for all employees in department 20. Round up any cents in the new salary amounts to the nearest dollar. Give the column the heading, SAL+15%, as shown:

EMPLOYEE_ID	LAST_NAME	SAL+15%
201	Hartstein	14950
202	Fay	6900

Practice 3 (continued)

4. Produce the following list of employees and their jobs.

Employees and Jobs
King works as a ad_pres
Kochhar works as a ad_vp
De Haan works as a ad_vp
Hunold works as a it_prog
Ernst works as a it_prog
Lorentz works as a it_prog
Mourgos works as a st_man
Rajs works as a st_clerk
Davies works as a st_clerk
Matos works as a st_clerk
Vargas works as a st_clerk
Zlotkey works as a sa_man
Abel works as a sa_rep
Taylor works as a sa_rep
Employees and Jobs
Grant works as a sa_rep
Whalen works as a ad_asst
Hartstein works as a mk_man
Fay works as a mk_rep
Higgins works as a ac_mgr
Gietz works as a ac_account

20 rows selected.

5. Display the employee ID, last name, monthly commission percentage, and monthly commission pct rounded to two decimal places for all salesmen. (`JOB_ID = 'SA_MAN' or JOB_ID ='SA_REP'`)

   Note: `COMMISSION_PCT` is an annual figure.

EMPLOYEE_ID	LAST_NAME	COMMISSION_PCT/12	COMM_ROUNDED
174	Abel	.025	.03
176	Taylor	.016666667	.02
178	Grant	.0125	.01
149	Zlotkey	.016666667	.02

Practice 3 (continued)

6. Produce a one - column report showing the first name and last name of each employee separated by a dash (-). Give the column the heading Employee Details, as shown:

Employees Details
Steven- King
Neena- Kochhar
Lex- De Haan
Alexander- Hunold
Bruce- Ernst
Diana- Lorentz
Kevin- Mourgos
Trenna- Rajs
Curtis- Davies
Randall- Matos
Peter- Vargas
Eleni- Zlotkey
Ellen- Abel
Jonathon- Taylor
Employees Details
Kimberely- Grant
Jennifer- Whalen
Michael- Hartstein
Pat- Fay
Shelley- Higgins
William- Gietz

20 rows selected.

Practice 3 (continued)

7. Display the last name, job ID, and total annual income (including commission where applicable) for all employees.

LAST_NAME	JOB_ID	ANNUAL_SAL
King	AD_PRES	288000
Kochhar	AD_VP	204000
De Haan	AD_VP	204000
Hunold	IT_PROG	108000
Ernst	IT_PROG	72000
Lorentz	IT_PROG	50400
Mourgos	ST_MAN	69600
Rajs	ST_CLERK	42000
Davies	ST_CLERK	37200
Matos	ST_CLERK	31200
Vargas	ST_CLERK	30000
Zlotkey	SA_MAN	151200
Abel	SA_REP	171600
Taylor	SA_REP	123840
Grant	SA_REP	96600
Whalen	AD_ASST	52800
Hartstein	MK_MAN	156000
Fay	MK_REP	72000
Higgins	AC_MGR	144000
Gietz	AC_ACCOUNT	99600

20 rows selected.

Practice 3 (continued)

8. Display the employee ID, last name, and salary plus the commission amount increased by 20% for all employees.

EMPLOYEE_ID	LAST_NAME	NEW SALARY
100	King	24000
101	Kochhar	17000
102	De Haan	17000
103	Hunold	9000
104	Ernst	6000
107	Lorentz	4200
124	Mourgos	5800
141	Rajs	3500
142	Davies	3100
143	Matos	2600
144	Vargas	2500
149	Zlotkey	10920
174	Abel	11660
176	Taylor	8944
EMPLOYEE_ID	LAST_NAME	NEW SALARY
178	Grant	7210
200	Whalen	4400
201	Hartstein	13000
202	Fay	6000
205	Higgins	12000
206	Gietz	8300

20 rows selected.

4

Single-Row Date and Conversion Functions

Objectives

After completing this lesson, you should be able to do the following:

- Use `SYSDATE` in conjunction with `SELECT` statements
- Describe the use of conversion functions
- Use date functions in `SELECT` statements
- Nest functions within a `SELECT` statement

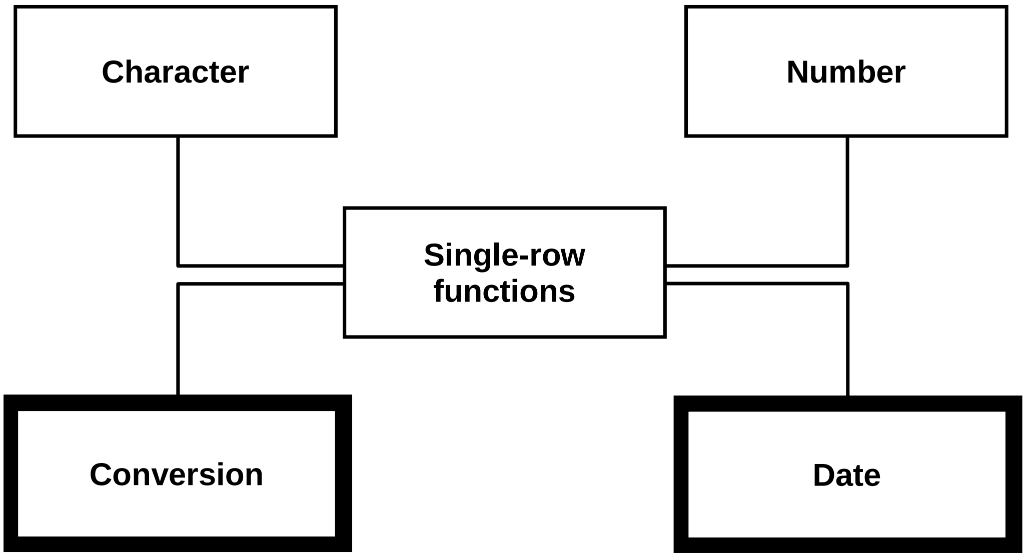
4-2

Lesson Aim

The previous lesson discussed the number and character single-row functions.

This lesson focuses on single-row functions that operate on dates and functions that convert data from one type to another: for example, from character data to numeric. The lesson also covers nested functions.

Single-Row Functions



Single-Row Functions

This lesson covers the following single-row functions:

- Date functions: Operate on values of the date data type. All date functions return a value of date data type except for the `MONTHS_BETWEEN` function, which returns a number.
- Conversion functions: Convert a value from one data type to another.

Working with Dates

- **Oracle database stores dates in an internal numeric format: century, year, month, day, hours, minutes, seconds.**
- **The default display date format is `DD-MON-RR`.**

```
SELECT last_name, hire_date
FROM   employees
WHERE  last_name like 'G%';
```

LAST_NAME	HIRE_DATE
Gietz	07-JUN-94
Grant	24-MAY-99

Oracle Date Storage

In the example in the slide, the `HIRE_DATE` for the employee is displayed in the default format `DD-MON-RR`. However, dates are not stored in the database in this format. All the components of the date and time are stored. So, although a `HIRE_DATE` like `07-JUN-94` is displayed as day, month and year there is also time and century information associated with it. The complete data might be `June 07, 1994 5:10:43 p.m`.

This data is stored internally as follows:

CENTURY	YEAR	MONTH	DAY	HOUR	MINUTE	SECOND
19	94	06	07	5	10	43

Alternatively, if the `HIRE_DATE` is in the 21st century, say `07-Jun-2001`, the complete data might be June 07, 2001 5:10:43 p.m.

This data is stored internally as follows:

CENTURY	YEAR	MONTH	DAY	HOUR	MINUTE	SECOND
20	01	06	07	5	10	43

Oracle Date Storage (Continued)

Centuries and the Year 2000

The Oracle server is Year 2000 compliant. When a record with a date column is inserted into a table, the century information is picked up from the `SYSDATE`. However, when the date column is displayed on the screen, the century component is not displayed by default. The `DATE` datatype always stores year information as a four-digit number internally, two digits for the century and two digits for the year. For example, the Oracle database stores the year as 1996 or 2001, and not just as 96 or 01.

RR Date Format

Current Year	Specified Date	RR Format	YY Format
1995	27-OCT-95	1995	1995
1995	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017
2001	27-OCT-95	1995	2095

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="2">If the specified two-digit year is:</td></tr>
<tr><td>0–49</td><td>50–99</td></tr>
<tr><td rowspan="2">If two digits of the current year are:</td><td>0–49</td><td>The return date is in the current century</td><td>The return date is in the century before the current one</td></tr>
<tr><td>50–99</td><td>The return date is in the century after the current one</td><td>The return date is in the current century</td></tr>
</table>

The RR Date Format Element

The RR date format is similar to the YY element, but it allows you to specify different centuries. You can use the RR date format element instead of YY, so that the century of the return value varies according to the specified two-digit year and the last two digits of the current year. The table on the slide summarizes the behavior of the RR element.

Current Year	Given Date	RR Format	YY Format
1994	27-OCT-95	1995	1995
1994	27-OCT-17	2017	1917
2001	27-OCT-17	2017	2017

SYSDATE

- Use SYSDATE to display the current date and time.
- SYSDATE can be displayed using the DUAL table.
- DUAL is a one-column, one-row table that is used as a dummy table.

```
SELECT SYSDATE
FROM   DUAL;
```

SYSDATE
16-OCT-01

SYSDATE

SYSDATE is a date function that returns the current date and time. You can use SYSDATE just as you would use any other column name. For example, you can display the current date by selecting SYSDATE from a table. It is customary to select SYSDATE from a dummy table called DUAL.

DUAL

The DUAL table is automatically created by the Oracle server and can be accessed by all users. It has one column, DUMMY, defined to be VARCHAR2(1), and contains one row with a value 'X'. The DUAL table is useful for computing a constant expression with the SELECT statement. Because DUAL has only one row, the constant is returned only once. Alternatively, you can select a constant, pseudocolumn, or expression from any table, but the value will be returned as many times as there are rows in the table.

Example

Display the current date by using the DUAL table:

```
SELECT     SYSDATE
 FROM      DUAL;
```

Note: SYSDATE is a SQL function that returns the current date and time. Your results may differ from the example in the slide.

Arithmetic with Dates

- **Add or subtract a number to or from a date to obtain a date value**
- **Subtract two dates to find the number of days between those dates**

4-8

Using Arithmetic Operators with Dates

Because the database stores dates as numbers, you can use arithmetic operators to perform calculations such as addition and subtraction on dates. You can add and subtract number constants as well as dates.

You can perform the following operations on dates:

Operation	Result	Description
date + number	Date	Add a number of days to a date
date - number	Date	Subtract a number of days from a date
date - date	Number of days	Subtract one date from another
date + number/24	Date	Add a number of hours to a date

The Oracle Server interprets number constants in arithmetic date expressions as numbers of days. For example, `SYSDATE + 1` is tomorrow. `SYSDATE - 7` is one week ago. Subtracting the `HIRE_DATE` column of the `EMPLOYEES` table from `SYSDATE` returns the number of days since each employee was hired. You cannot multiply or divide `DATE` values.

Using Arithmetic Operators with Dates

```
SELECT last_name, hire_date,
       hire_date+30 "NEW DATE"
FROM   employees
WHERE  last_name='Grant';
```

LAST_NAME	HIRE_DATE	NEW DATE
Grant	24-MAY-99	23-JUN-99

Using Arithmetic Operators with Dates (continued)

The example in the slide displays the last name, hire date, and the date on which an employee's training period completes. The calculation simply adds 30 days to the `HIRE_DATE` to get the new date.

Using SYSDATE in Calculations

For how many weeks have the employees in department 10 worked ?

```
SELECT last_name, (SYSDATE-hire_date)/7
       "WEEKS AT WORK"
FROM  employees
WHERE department_id =10;
```

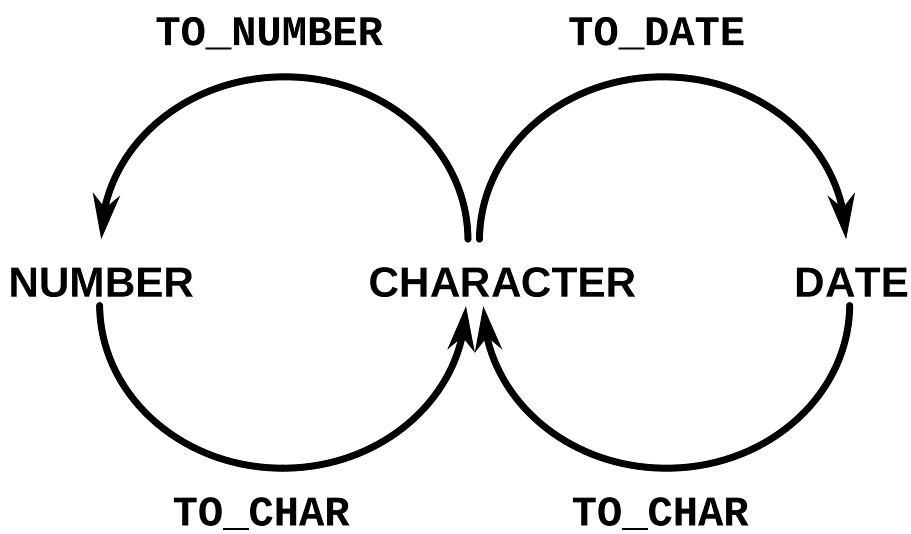
LAST_NAME	WEEKS AT WORK
Whalen	734.768019

Performing Calculations with Dates

The example in the slide displays the last name and the number of weeks the employee has worked for the company, for all employees in department 10. The example subtracts the date on which the employee was hired from the current date (SYSDATE) and divides the result by 7 to calculate the duration of employment in weeks.

Note: SYSDATE is a SQL function that returns the current date and time. Your results may differ from the example.

Explicit Data Type Conversion



ORACLE

4-11

Explicit Data Type Conversion

SQL provides three functions to convert a value from one data type to another:

Function	Purpose

`TO_CHAR(`*`number`*`|`*`date`*`,[`
`fmt``],`
`[`*`nlsparams`*`])`
	Converts a number or date value to a `VARCHAR2` character string with format model *fmt*.
	Number Conversion:
	The `NLSPARAMS` parameter specifies the following characters, which are returned by number format elements:
	• Decimal character
	• Group separator
	• Local currency symbol
	• International currency symbol
	If `NLSPARAMS` or any one of the parameters is omitted, this function uses the default parameter values for the session.

Explicit Data Type Conversion (continued)

Function	Purpose
	Date conversion: The `NLSPARAMS` parameter specifies the language in which month and day names and abbreviations are returned. This argument can have the form: `'NLS_DATE_LANGUAGE = language'` If this parameter is omitted, the default date language is used for the session
`TO_NUMBER(`*`char,[ fmt],`* *`[nlsparams]`*`)`	Converts a character string containing digits to a number in the format specified by the optional format model *fmt*. The `NLSPARAMS` parameter has the same purpose in this function as in the `TO_CHAR` function for number conversion.
`TO_DATE(`*`char`*`,[` *`fmt`*`],[`*`nlsparams`*`])`	Converts a character string representing a date to a date value according to the *fmt* specified. If *fmt* is omitted, the format is `DD-MON-RR`. The `NLSPARAMS` parameter has the same purpose in this function as in the `TO_CHAR` function for date conversion.

Note: This list is a subset of the available conversion functions.

For more information, see *Oracle Server SQL Reference, "Conversion Functions."*

Modifying the Display Format of Dates

27-JAN-01 → Saturday the 27th of January, 2001

27-JAN-01 → January 27, 2001

27-JAN-01 → 01/27/01

4-13

Displaying a Date in a Specific Format

So far in this course, all Oracle date values have appeared in the `DD-MON-RR` format. You can use the `TO_CHAR` function to convert a date from this default format to the one that you specify.

For example, 03-APR-71 can be displayed in many different formats including:

- 04/03/71
- April 3rd, 1971
- Third of April, nineteen seventy one
- Saturday, the 3rd of April, 1971

TO_CHAR Function with Dates

```
TO_CHAR(date, 'format_model')
```

The format model:

- **Is case sensitive and must be enclosed in single quotation marks**
- **Can include any valid date format element**
- **Has an `fm` element to remove padded blanks or suppress leading zeros**
- **Is separated from the date value by a comma**

4-14

Guidelines for Date Format Models

- The format model is case sensitive and must be enclosed in single quotation marks.
- The format model can include any valid date format element. Be sure to separate the date value from the format model with a comma.
- The names of days and months in the output are automatically padded with blanks.
- To remove padded blanks or to suppress leading zeros, use the fill mode (`fm`) element.
- You can resize the display width of the resulting character field with the *i*SQL*Plus `COLUMN` command, which is covered later in this course.
- The default column width is 80 characters.

Date Format Model Elements

YYYY	Full year in numbers
YEAR	Year spelled out
MM	Two-digit value for month
MONTH	Full name of the month
MON	Three-letter abbreviation of the month
DY	Three-letter abbreviation of the day of the week
DAY	Full name of the day

ORACLE

4-15

Sample Valid Date Format Elements

Element	Description
CC or SCC	One greater than the first two digits of a four-digit year; "S" prefixes BC dates with "-". For example, '20' from '1900'.
Years in dates YYYY or SYYYY	4-digit year; "S" prefixes BC dates with "-".
YYY or YY or Y	Last 3, 2, or 1 digits of year
Y,YYY	Year with comma in this position
IYYY, IYY, IY, I	4, 3, 2, or 1 digit year based on the ISO standard
SYEAR or YEAR	Year spelled out; S prefixes B.C. date with -
BC or AD	B.C./A.D. indicator without periods
B.C. or A.D.	B.C./A.D. indicator with periods

Sample Valid Date Format Elements (continued)

Element	Description
Q	Quarter of year
MM	Month, 2-digit value
MONTH	Name of month padded with blanks to length of 9 characters
MON	Name of month, three-letter abbreviation
RM	Roman numeral month
WW or W	Week of year or month
DDD or DD or D	Day of year, month, or week
DAY	Name of day padded with blanks to length of nine characters
DY	Name of day; three-letter abbreviation
J	Julian day; the number of days since 31 December 4713 B.C.
RR	Given a year with 2 digits: • If the year is <50 and the last 2 digits of the current year are >=50, the first 2 digits of the returned year are 1 greater than the first 2 digits of the current year. • If the year is >=50 and the last 2 digits of the current year are <50, the first 2 digits of the returned year are 1 less than the first 2 digits of the current year.

Using the TO_CHAR Function with Dates

```
SELECT last_name,
       TO_CHAR(hire_date, 'Month DDTH, YYYY')
AS     HIREDATE
FROM   employees
WHERE  job_id ='IT_PROG';
```

LAST_NAME	HIREDATE
Hunold	January 03RD, 1990
Ernst	May 21ST, 1991
Lorentz	February 07TH, 1999

Using the TO_CHAR Function with Dates

In the example in the slide, TO_CHAR is used to display the HIRE_DATE column in the following format: Month DDTH YYYY.

So the date 03-JAN-90 is displayed as January 03RD, 1990. Note the following points in the display:

- The alias HIREDATE is used to replace the entire TO_CHAR expression in the column heading.
- The day of the month is preceded by a 0. You can use the fm element to eliminate this digit.

When a record with a date column is inserted into a table using the DD-MON-YY date format, YY indicates the year in the 20th century if the SYSDATE is less than or equal to 31-Dec-1999 (for example, 31-DEC-92 is December 31, 1992). But if the SYSDATE is greater than 31-Dec-1999, then YY indicates a year in the 21st century (for example, if the SYSDATE is 01-Jan-2000, then 31-DEC-92 will be December 31, 2092). You can display the date with the century component by using the TO_CHAR function with the YYYY format.

Using the TO_CHAR Function with Dates

```
SELECT  employee_id,
        TO_CHAR(hire_date, 'MM/YY') AS MONTH
FROM    employees
WHERE   last_name ='Vargas';
```

EMPLOYEE_ID	MONTH
144	07/98

Using the TO_CHAR Function with Dates (continued)

The SQL statement in the slide displays the employee ID and hire date for the employee whose last name is Vargas. The TO_CHAR function is used to convert the display of the HIRE_DATE column from the DD-MON-YY format to the simpler MM/YY format.

The order of the date can also be rearranged:

```
SELECT employee_id, last_name,
       TO_CHAR(hire_date, 'YYYY-MON-DD') HIRED
FROM   employees;
```

Using the TO_CHAR Function with Dates (continued)

EMPLOYEE_ID	LAST_NAME	HIRED
100	King	1987-JUN-17
101	Kochhar	1989-SEP-21
102	De Haan	1993-JAN-13
103	Hunold	1990-JAN-03
104	Ernst	1991-MAY-21
107	Lorentz	1999-FEB-07
124	Mourgos	1999-NOV-16
141	Rajs	1995-OCT-17
142	Davies	1997-JAN-29
143	Matos	1998-MAR-15
144	Vargas	1998-JUL-09
149	Zlotkey	2000-JAN-29
174	Abel	1996-MAY-11
176	Taylor	1998-MAR-24
EMPLOYEE_ID	LAST_NAME	HIRED
178	Grant	1999-MAY-24
200	Whalen	1987-SEP-17
201	Hartstein	1996-FEB-17
202	Fay	1997-AUG-17
205	Higgins	1994-JUN-07
206	Gietz	1994-JUN-07

20 rows selected.

Using the TO_CHAR Function with Dates

```
SELECT last_name,
       TO_CHAR(hire_date, 'fmDD Month YYYY')
       AS HIREDATE
FROM   employees;
```

LAST_NAME	HIREDATE
King	17 June 1987
Kochhar	21 September 1989
De Haan	13 January 1993
Hunold	3 January 1990
Ernst	21 May 1991
Lorentz	7 February 1999
Mourgos	16 November 1999
Rajs	17 October 1995
Davies	29 January 1997
Matos	15 March 1998
Vargas	9 July 1998
Zlotkey	29 January 2000
Abel	11 May 1996

...

20 rows selected.

4-20

Using the TO_CHAR Function with Dates

The SQL statement in the slide displays the last name and hire dates for all employees. The HIRE_DATE is displayed in the format 17 June 1987.

Use of fm in the date format model results in the entire date string being displayed with single spaces between the day, month, and year, and justified to the left. The leading zeros from the day and the trailing spaces from the month are also removed.

Using the TO_CHAR Function with Dates

```
SELECT last_name, manager_id, salary,
       TO_CHAR(hire_date,'YYYY-MON-DD')
       AS HIREDATE
FROM   employees
WHERE  salary < 15000
AND    hire_date like '%90';
```

LAST_NAME	MANAGER_ID	SALARY	HIREDATE
Hunold	102	9000	1990-JAN-03

4-21

Using the TO_CHAR Function with Dates

The example in the slide selects any employee who earns less than 15000 and was hired in the nineties. The TO_CHAR function is used to convert the display of the HIRE_DATE column from a DD-MON-YY format to the YYYY-MON-DD format.

Note: If there was an employee whose hire date was in the 21st century, say 01-JAN-2090, the result would display that record also. The TO_CHAR function with the YYYY format ensures that the century is displayed.

Using the TO_CHAR Function with Dates

```
SELECT employee_id,last_name,department_id,
       TO_CHAR(hire_date,'MM-DD-YYYY')
       AS HIREDATE
FROM   employees
WHERE  hire_date NOT LIKE '%99';
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	HIREDATE
100	King	90	06-17-1987
101	Kochhar	90	09-21-1989
102	De Haan	90	01-13-1993
103	Hunold	60	01-03-1990
104	Ernst	60	05-21-1991
141	Rajs	50	10-17-1995
142	Davies	50	01-29-1997
143	Matos	50	03-15-1998
144	Vargas	50	07-09-1998
149	Zlotkey	80	01-29-2000
174	Abel	80	05-11-1996

...

17 rows selected.

ORACLE

Using the TO_CHAR Function with Dates (continued)

The expression in the slide returns the employee ID, last name, department ID, and hire date of the employees who were not hired in the year 99. The `TO_CHAR` function is used to convert the display of the `HIRE_DATE` column from a `DD-MON-YY` format to the `MM-DD-YYYY` format.

Using the TO_CHAR Function with Dates

```
SELECT  last_name, job_id, department_id,
        TO_CHAR(hire_date,'DD-MON-YYYY')
        AS HIREDATE
FROM    employees
ORDER   BY hire_date DESC;
```

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIREDATE
Zlotkey	SA_MAN	80	29-JAN-2000
Mourgos	ST_MAN	50	16-NOV-1999
Grant	SA_REP		24-MAY-1999
Lorentz	IT_PROG	60	07-FEB-1999
Vargas	ST_CLERK	50	09-JUL-1998
Taylor	SA_REP	80	24-MAR-1998
Matos	ST_CLERK	50	15-MAR-1998
Fay	MK_REP	20	17-AUG-1997
Davies	ST_CLERK	50	29-JAN-1997
Abel	SA_REP	80	11-MAY-1996

...

20 rows selected.

Using the TO_CHAR Function with Dates (continued)

The example in the slide sorts the result, beginning with the most recently hired employee. The `TO_CHAR` function is used to convert the display of the `HIRE_DATE` column from a `DD-MON-YY` format to the `DD-MON-YYYY` format.

Note: If there was an employee whose hire date was in the 21st century, say 01-JAN-2003, the result would display the record as the first record. The `TO_CHAR` function with the `YYYY` format ensures that the century is displayed.

Date Format Model Elements

- Time elements format the time portion of the date.

`HH24:MI:SS`	15:45:32

- Add character strings by enclosing them in double quotation marks.

`DD "of" MONTH`	12 of OCTOBER

- Number suffixes spell out numbers

`ddspth`	fourteenth

ORACLE

Time Formats

Use the formats listed in the following tables to display time information and literals and to change numerals to spelled numbers.

Element	Description
`AM` or `PM`	Meridian indicator
`A.M.` or `P.M.`	Meridian indicator with periods
`HH` or `HH12` or `HH24`	Hour of day or hour (1-12) or hour (0-23)
`MI`	Minute (0-59)
`SS`	Second (0-59)
`SSSSS`	Seconds past midnight (0-86399)

Other Formats

Element	Description
`/ . ,`	Punctuation is reproduced in the result
" of the "	Quoted string is reproduced in the result

Other Formats (continued)

Element	Description
TH	Ordinal number (for example, DDTH for 4TH)
SP	Spelled-out number (for example, DDSP for FOUR)
SPTH or THSP	Spelled-out ordinal numbers (for example, DDSPTH for FOURTH)

Using Format Models to Display Time

```
SELECT TO_CHAR(SYSDATE,'HH24:MI:SS') TIME
FROM   DUAL;
```

TIME
09:31:14

Date Format Models to Display Time

As described earlier, the Oracle Server stores dates including hours, minutes, and seconds. Time details can be displayed for a date by creating a date format model specifying the time details desired. The example in the slide shows the display of the 24-hour time for the current date using a date format model.

Example

The following example displays the 12-hour time with the addition of a meridian indicator to show a.m. or p.m.

```
SELECT TO_CHAR(SYSDATE,'HH12:MI:SS a.m.') TIME
FROM   DUAL;
```

TIME
09:57:01 a.m.

Date Format Models to Display Time (continued)

Example

The example below modifies the `SELECT` statement to display the `HIRE_DATE` in the following format:

```
SELECT last_name,
       TO_CHAR(hire_date, 'fmDdspth "of" Month
               YYYY HH:MI:SS AM') AS HIREDATE
FROM   employees;
```

LAST_NAME	HIREDATE
King	Seventeenth of June 1987 12:0:0 AM
Kochhar	Twenty-First of September 1989 12:0:0 AM
De Haan	Thirteenth of January 1993 12:0:0 AM
Hunold	Third of January 1990 12:0:0 AM
Ernst	Twenty-First of May 1991 12:0:0 AM
Lorentz	Seventh of February 1999 12:0:0 AM
Mourgos	Sixteenth of November 1999 12:0:0 AM
Rajs	Seventeenth of October 1995 12:0:0 AM
Davies	Twenty-Ninth of January 1997 12:0:0 AM
Matos	Fifteenth of March 1998 12:0:0 AM
Vargas	Ninth of July 1998 12:0:0 AM
Zlotkey	Twenty-Ninth of January 2000 12:0:0 AM
Abel	Eleventh of May 1996 12:0:0 AM
Taylor	Twenty-Fourth of March 1998 12:0:0 AM
LAST_NAME	HIREDATE
Grant	Twenty-Fourth of May 1999 12:0:0 AM
Whalen	Seventeenth of September 1987 12:0:0 AM
Hartstein	Seventeenth of February 1996 12:0:0 AM
Fay	Seventeenth of August 1997 12:0:0 AM
Higgins	Seventh of June 1994 12:0:0 AM
Gietz	Seventh of June 1994 12:0:0 AM

20 rows selected.

Instructor Note

TO_CHAR Function with Numbers

```
TO_CHAR(n,'fmt')
```

Use these formats with the TO_CHAR function to display a number value as a character:

9	Represents a number
0	Forces a zero to be displayed
$	Places a floating dollar sign
L	Uses the floating local currency symbol
.	Prints a decimal point
,	Places a thousand indicator

4-28

TO_CHAR Function with Numbers

Syntax:

```
TO_CHAR(n,'fmt')
```

The TO_CHAR function converts *n* of NUMBER datatype to a value of VARCHAR2 datatype, using the optional number format *fmt*. If you omit fmt, *n* is converted to a VARCHAR2 value exactly long enough to hold its significant digits.

Number Format Elements

If you are converting a number to a character data type, use the following elements:

Element	Description	Example	Result
9	Numeric position (number of 9s determine display width)	999999	1234
0	Display leading zeros	099999	001234
$	Floating dollar sign	$999999	$1234
L	Floating local currency symbol	L999999	FF1234

TO_CHAR Function with Numbers (continued)

Element	Description	Example	Result
.	Decimal point in position specified	999999.99	1234.00
,	Comma in position specified	999,999	1,234
MI	Minus signs to right (negative values)	999999MI	1234-
PR	Parenthesize negative numbers	999999PR	<1234>
EEEE	Scientific notation (format must specify four Es)	99.999EEEE	1.234E+03
V	Multiply by 10 *n* times (*n* = number of 9s after V)	9999V99	123400
B	Display zero values as blank, not 0	B9999.99	1234.00

Using the TO_CHAR Function with Numbers

```
SELECT  TO_CHAR(salary,'$99,999') SALARY
FROM    employees
WHERE   last_name = 'Hartstein';
```

SALARY
$13,000

Dollar sign

Thousand indicator

Using the TO_CHAR Function with Numbers

In the example in the slide, the TO_CHAR function formats the display of the numeric SALARY column. TO_CHAR converts the SALARY column to the character data type and inserts a dollar sign before the amount and a comma as a thousand indicator.

Guidelines

- The Oracle Server displays a string of hash signs (#) in place of a whole number whose digits exceed the number of digits provided in the format model.
- The Oracle Server rounds the stored decimal value to the number of decimal spaces provided in the format model.

Using the TO_NUMBER and TO_DATE Functions

Convert a character string to a number format using the TO_NUMBER function:

```
TO_NUMBER(char[, 'format_model'])
```

Convert a character string to a date format using the TO_DATE function:

```
TO_DATE(char[, 'format_model'])
```

4-31

The TO_NUMBER and TO_DATE Functions

You may want to convert a character string to either a number or a date. To accomplish this task, you use the TO_NUMBER or TO_DATE functions. The format model you choose is based on the previously demonstrated format elements.

Using the TO_NUMBER Function

```
SELECT   TO_NUMBER('1000')+salary AS NEW_SALARY
FROM     employees
WHERE    last_name = 'Matos';
```

NEW_SALARY
3600

```
SELECT TO_NUMBER ('$1,000','L9,999') as NEW_SALARY
FROM employees
WHERE  last_name = 'De Haan';
```

NEW_SALARY
1000

ORACLE

Using the TO_NUMBER Function

The first example in the slide takes the employee's salary raise, which is in the form of a character string, and converts it to a numeric value. It then adds the value to the employee's salary, which is also a numeric value.

The second example uses the L9,999 format model to return in the specified position the local currency symbol (the current value of the NLS_CURRENCY parameter).

Using the TO_DATE Function

```
SELECT TO_DATE('January 15, 1989, 11:00 A.M.',
               'Month dd, YYYY, HH:MI A.M.')
FROM DUAL;
```

TO_DATE('
15-JAN-89

```
SELECT  last_name, hire_date
FROM    employees
WHERE   hire_date = TO_DATE('May 24, 1999',
                            'Month DD, YYYY');
```

LAST_NAME	HIRE_DATE
Grant	24-MAY-99

4-33

Using the TO_DATE Function

The first example in the slide converts a character string into date.

The second example displays the last names and hire dates of all the employees who joined on May 24, 1999.

Date Functions

FUNCTION	DESCRIPTION
MONTHS_BETWEEN	**Number of months between two dates**
ADD_MONTHS	**Adds calendar months to the date specified**
NEXT_DAY	**Next day following the date specified**
LAST_DAY	**Last day of the month**
ROUND	**Round off date**
TRUNC	**Truncate date**

ORACLE

4-34

How Date Functions Work

Date functions operate on Oracle dates. All date functions return a value of a date data type except MONTHS_BETWEEN, which returns a numeric value. Some of the date functions are:

- MONTHS_BETWEEN(*date1, date2*): Finds the number of months between *date1* and *date2*. The result can be positive or negative. If *date1* is later than *date2*, the result is positive; if *date1* is earlier than *date2*, the result is negative. The non-integer part of the result represents a portion of the month.
- ADD_MONTHS(*date, n*): Adds *n* number of calendar months to date. *n* must be an integer and can be negative. (ADD_MONTHS function will take decimal numbers but all decimal point will be truncated.)
- NEXT_DAY(*date,char*): Returns the date of the first weekday named by *char* that is later than the date *date*. *Char* may be a number representing a day or a character string.
- LAST_DAY(*date*): Finds the date of the last day of the month that contains *date*.
- ROUND(*date[, fmt]*): Returns date rounded to the unit specified by the format model *fmt*. If the format model *fmt* is omitted, *date* is rounded to the nearest day.
- TRUNC(*date[, fmt]*): Returns date with the time portion of the day truncated to the unit specified by the format model *fmt*. If the format model *fmt* is omitted, date is truncated to the current day with the time as midnight.

The above list is a subset of the available date functions.

Using Date Functions

Use the `ADD_MONTHS` function to add months to a date.

```
SELECT last_name, hire_date,
       ADD_MONTHS(hire_date, 6) AS "+6 MONTHS"
FROM   employees
WHERE  last_name='Vargas';
```

LAST_NAME	HIRE_DATE	+6 MONTHS
Vargas	09-JUL-98	09-JAN-99

ORACLE

Examples of Date Functions

- **MONTHS_BETWEEN('01-SEP-95','11-JAN-94')**
  → **19.6774194**
- **ADD_MONTHS('11-JAN-94',6)** → **11-JUL-94**
- **NEXT_DAY('01-SEP-95','FRIDAY')** → **08-SEP-95**
- **LAST_DAY('01-SEP-95')** → **30-SEP-95**

ORACLE

4-36

Sample Date Functions

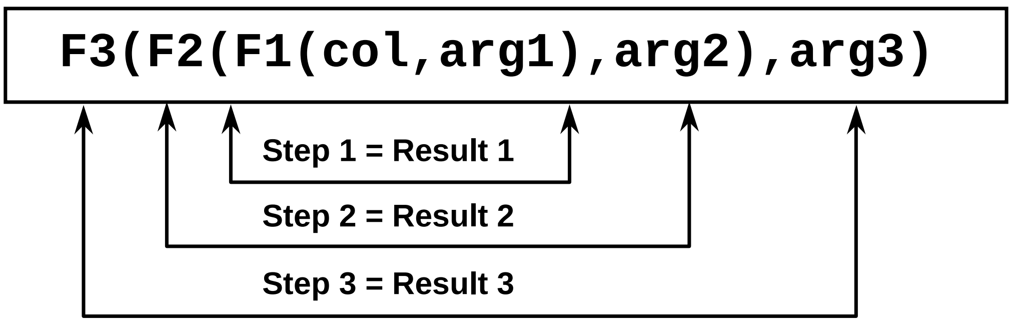
For all employees employed for fewer than 30 months, display the last name, hire date, number of months employed, six month review date, first Friday after hire date, and the last day of month when there were hired.

```
SELECT  last_name, hire_date,
        MONTHS_BETWEEN(SYSDATE, hire_date) TENURE,
        ADD_MONTHS(hire_date, 6) REVIEW,
        NEXT_DAY(hire_date, 'Friday'), LAST_DAY(hire_date)
FROM employees
WHERE MONTHS_BETWEEN (SYSDATE, hire_date) < 30;
```

LAST_NAME	HIRE_DATE	TENURE	REVIEW	NEXT_DAY(	LAST_DAY(
Mourgos	16-NOV-99	23	16-MAY-00	19-NOV-99	30-NOV-99
Zlotkey	29-JAN-00	20.5941099	29-JUL-00	04-FEB-00	31-JAN-00
Grant	24-MAY-99	28.7554002	24-NOV-99	28-MAY-99	31-MAY-99

Nesting Functions

- Single-row functions can be nested to any level.
- Nested functions are evaluated from the innermost level to the outermost level.



ORACLE

4-37

Nesting Functions

Single-row functions can be nested to any depth. Nested functions are evaluated from the innermost level to the outermost level. The following examples show you the flexibility of these functions.

Nesting Functions

Result 2

Result 1

```
SELECT last_name,
       NVL(TO_CHAR(manager_id),'No Manager')
FROM employees
WHERE manager_id IS NULL;
```

LAST_NAME	NVL(TO_CHAR(MANAGER_ID),'NOMANAGER')
King	No Manager

ORACLE

4-38

Nesting Functions (continued)

The example in the slide displays the head of the company, who has no manager. The evaluation of the SQL statement involves two steps:

1. Evaluate the inner function to convert a number value to a character string.

   `Result1 = TO_CHAR(manager_id)`

2. Evaluate the outer function to replace the null value with a text string.

   `NVL(Result1, 'No Manager')`

The entire expression becomes the column heading because no column alias was given.

Example

Using the `EMPLOYEES` table, display the date of the Friday that is six months from the hire date. The dates should be in the format Friday, December 18th, 1987. Order the results by hire date.

Nesting Functions

```
SELECT MONTHS_BETWEEN
      (TO_DATE('02-02-1995','MM-DD-YYYY'),
       TO_DATE('01-01-1995','MM-DD-YYYY'))
       AS "Months"
FROM DUAL;
```

Months
1.03225806

Nesting Functions (continued)

The example in the slide displays the months between 02-02-1995 and 01-01-1995.

1. Evaluate the inner function to convert the two strings, '02-02-1995' and '01-01-1995' to dates.
2. Evaluate the outer function to calculate the months between these two dates.

```
SELECT TO_CHAR(NEXT_DAY(ADD_MONTHS(hire_date, 6),'FRIDAY'),
               'fmDay, Month ddth, YYYY') "Next 6 Month Review"
FROM   employees
ORDER  BY  hire_date;
```

Using ROUND and TRUNC with Date Functions

- ROUND(TO_DATE('25-JUL-1995', 'DD-MON-YYYY'),'MONTH') → 01-AUG-95
- ROUND(TO_DATE('25-JUL-1995', 'DD-MON-YYYY'),'YEAR') → 01-JAN-96
- TRUNC(TO_DATE('25-JUL-1995', 'DD-MON-YYYY'),'MONTH') → 01-JUL-95
- TRUNC(TO_DATE('25-JUL-1995', 'DD-MON-YYYY'),'YEAR') → 01-JAN-95

ORACLE

4-40

Using ROUND and TRUNC with Date Functions

You can use the ROUND and TRUNC functions for number and date values. These functions round or truncate dates to the specified format model, such as to the nearest year or month.You can also round dates to the nearest day using no format model, setting the time element to 12:00 a.m. (midnight).

Example

Compare the hire dates for all employees who were employed in 1997. Display the last name and hire date. Also, display the month in which they were hired using the ROUND and TRUNC functions.

```
SELECT last_name, hire_date, ROUND(hire_date, 'MONTH') RND_MON,
       TRUNC(hire_date, 'MONTH') TRNC_MON
 FROM  employees
 WHERE TO_CHAR(hire_date,'DD-MON-YYYY') LIKE '%1997';
```

LAST_NAME	HIRE_DATE	RND_MON	TRNC_MON
Davies	29-JAN-97	01-FEB-97	01-JAN-97
Fay	17-AUG-97	01-SEP-97	01-AUG-97

When using the ROUND function with dates, the rules are as follows:

- When rounding to the closest year, round up July 1 and later dates.
- When rounding to the closest month, round up the 16th day of the month and later.
- When rounding to the nearest day, round up 12 noon and later times.

Summary

Use date and conversion functions to:

- **Convert column data types during calculations and display**
- **Perform calculations on data**
- **Modify individual data items**
- **Manipulate output for groups of rows**
- **Alter date formats for display**
- **Single-row functions can be nested to any depth**

4-41

SYSDATE and DUAL

SYSDATE is a date function that returns the current date and time. It is customary to select SYSDATE from a dummy table called DUAL.

Single-Row Functions

Single-row functions can manipulate:

- Conversion functions can convert character, date, and numeric values. Examples of conversion functions are:
  - TO_CHAR, TO_DATE, TO_NUMBER
- Date functions:
  - MONTHS_BETWEEN, ADD_MONTHS, NEXT_DAY, LAST_DAY, ROUND, TRUNC
  - Date values can also use arithmetic operators.
- Single-row functions can be used to convert character, date, and numeric values with various conversion functions like TO_CHAR, TO_DATE, and TO_NUMBER. Single-row functions can be nested to any level.
- Single-row functions can be nested to any depth. Nested functions are evaluated from the innermost level to the outermost level.

Practice 4 Overview

This practice covers the following topics:

- Using date functions to alter the display of dates
- Using SYSDATE in SELECT statements
- Using conversion functions to convert data from one data type to another

Practice 4 Overview

This practice contains a variety of exercises using date and conversion functions in the SELECT statement.

Note: Results of the practices will vary based on the value of SYSDATE.

Practice 4

1. Display the last name and hire date of all employees with the job ID `IT_PROG`. Display the hire date as shown:

LAST_NAME	HIRED_IN
Hunold	01/03/1990
Ernst	05/21/1991
Lorentz	02/07/1999

2. Determine the annual salary (excluding commission) and six-month review date for all employees with the job ID `ST_CLERK`. Give the column an alias of `REVIEW`.

LAST_NAME	SALARY*12	REVIEW
Rajs	42000	17-APR-1996
Davies	37200	29-JUL-1997
Matos	31200	15-SEP-1998
Vargas	30000	09-JAN-1999

3. Display the last name and number of days between today and the start date for all employees with the letter G as the first letter of their name.

LAST_NAME	DAYS_EMPLOYED
Gietz	2688.43075
Grant	876.430752

Note: Results will vary based on the value of `SYSDATE`

4. Display the number of months that Taylor has been employed with the company. Give the column an alias of `MONTHS`.

LAST_NAME	MONTHS
Taylor	42.7558498

Note: Results will vary due to the changing value of `SYSDATE`.

Practice 4 (continued)

5. For employees in department 20, display the last name and hire date as shown. Specify the alias as `DATE_HIRED` after your expression. Pay particular attention to the case used in the letters of the hire date.

LAST_NAME	DATE_HIRED
Hartstein	February, Seventeenth 1996
Fay	August, Seventeenth 1997

6. For employees in department 60, display each employee's last name, hire date, and salary review date. Assume that the review date is one year after the hire date. Give the review date column an alias of `REVIEW`. Order the output in ascending order of hire date.

LAST_NAME	HIRE_DATE	REVIEW
Hunold	03-JAN-1990	03-JAN-1991
Lorentz	07-FEB-1999	07-FEB-2000
Ernst	21-MAY-1991	21-MAY-1992

7. Display the last names of all employees who were hired after March 15, 1998. Use the date format 03/15/1998.

LAST_NAME
Lorentz
Mourgos
Vargas
Zlotkey
Taylor
Grant

6 rows selected.

Practice 4 (continued)

8. Create a single-column report that lists sales representatives (`JOB_ID = 'SA_REP'`) and their monthly salaries as shown in the following output. Pay particular attention to the case used in the letters and the formatting of the salary amounts.

MONTHLYSALARIES
Abel earns $11,000 a month
Taylor earns $8,600 a month
Grant earns $7,000 a month

If you want an extra challenge, try the following exercises:

9. Display the date of the first Monday in the year 2001. Give the column the heading as Monday.

Monday
01-JAN-2001

10. Display the last names and hire dates of all employees who have been with the company for more than 10 years.

LAST_NAME	HIREDATE
King	17-JUN-1987
Kochhar	21-SEP-1989
Hunold	03-JAN-1990
Ernst	21-MAY-1991
Whalen	17-SEP-1987

Note: The output will vary from year to year, depending on `SYSDATE`.

Practice 4 (continued)

11. Display the last name and hire date for all employees who were hired in 1987.

LAST_NAME	HIREDATE
King	17-JUN-1987
Whalen	17-SEP-1987

12. Display the last name and hire date for all employees whose job ID is `ST_CLERK`, starting with the clerk who was hired first and ending with the clerk who was hired most recently.

LAST_NAME	HIREDATE
Rajs	17-OCT-1995
Davies	29-JAN-1997
Matos	15-MAR-1998
Vargas	09-JUL-1998

13. Display the last name, hire date, hire date rounded to the `MONTH`, and hire date rounded to the `YEAR` for employees with an employee ID is greater than 170. The column headings should be as given below.

LAST_NAME	HIREDATE	ROUND_MONTH	ROUND_YEAR
Abel	11-MAY-1996	01-MAY-1996	01-JAN-1996
Taylor	24-MAR-1998	01-APR-1998	01-JAN-1998
Grant	24-MAY-1999	01-JUN-1999	01-JAN-1999
Whalen	17-SEP-1987	01-OCT-1987	01-JAN-1988
Hartstein	17-FEB-1996	01-MAR-1996	01-JAN-1996
Fay	17-AUG-1997	01-SEP-1997	01-JAN-1998
Higgins	07-JUN-1994	01-JUN-1994	01-JAN-1994
Gietz	07-JUN-1994	01-JUN-1994	01-JAN-1994

8 rows selected.

5

Displaying Data from Multiple Tables

ORACLE

Objectives

After completing this lesson, you should be able to do the following:

- Write SELECT statements using equality and nonequality joins to access data from more than one table
- Describe the Cartesian product
- Join a table to itself
- Join tables using SQL: 1999 Syntax

5-2

Lesson Aim

This lesson covers how to use different methods to obtain data from more than one table.

Obtaining Data from Multiple Tables

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90
103	Hunold	60
104	Ernst	60

...

20 rows selected.

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700

8 rows selected.

EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
100	10	Administration
100	20	Marketing
100	50	Shipping
100	60	IT
100	80	Sales
100	90	Executive

...

160 rows selected.

ORACLE

Data from Multiple Tables

Sometimes you need to use data from more than one table. In the slide example, the report displays data from two separate tables.

- Employee IDs exist in the `EMPLOYEES` table.
- Department IDs exist in both the `EMPLOYEES` and `DEPARTMENTS` tables.
- Department names exist in the `DEPARTMENTS` table.

To produce the report, you need to link the `EMPLOYEES` and `DEPARTMENTS` tables and access data from both of them.

Joining Tables Using Oracle Syntax

- **Use a join to query data from more than one table:**

```
SELECT   table1.column1, table2.column2
FROM     table1, table2
WHERE    table1.column1 = table2.column2;
```

- **Write the join condition in the `WHERE` clause.**
- **Prefix the column name with the table name when the same column name appears in more than one table.**

Joining Tables Using Oracle Syntax

When you require data from more than one table in the database, you use a join condition. Rows in one table can be joined to rows in another table according to common values existing in corresponding columns. Typically, when rows are joined using a common value, the column in the first table is a primary key and the column in the second table is a foreign key.

To display data from two or more related tables, write a simple join condition in the `WHERE` clause. In the syntax:

table.column	Denotes the table and column from which data is retrieved
table1.column1 = table2.column2	Is the condition that joins (or relates) the tables together

This example of a join condition is called an equijoin. It is based on the value in one column of a table being equal to the value in another table. Equijoins are discussed later in this lesson.

Guidelines

- When writing a `SELECT` statement that joins tables, precede the column name with the table name for clarity and to enhance database access.
- If the same column name appears in more than one table, the column name must be prefixed with the table name. If this not done the Oracle server returns the error `ORA-00918: column ambiguously defined.`
- To join *n* tables together, you need a minimum of *n-1* join conditions. For example, to join four tables, a minimum of three joins is required. This rule may not apply if your table has a concatenated primary key, in which case more than one column is required to uniquely identify each row.

For more information, see *Oracle9i SQL Reference, "SELECT."*

.

Cartesian Product

- A Cartesian product is formed when:
  - A join condition is omitted
  - A join condition is invalid
  - All rows in the first table are joined to all rows in the second table
- To avoid a Cartesian product, always include a valid join condition in a `WHERE` clause.

5-6

Cartesian Product

When a join condition is invalid or omitted completely, the result is a Cartesian product in which all combinations of rows are displayed. All rows in the first table are joined to all rows in the second table.

A Cartesian product tends to generate a large number of rows, and its result is rarely useful. You should always include a valid join condition in a `WHERE` clause, unless you have a specific need to combine all rows from all tables.

Generating a Cartesian Product

EMPLOYEES (20 rows)

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90
103	Hunold	60
104	Ernst	60

...

20 rows selected.

DEPARTMENTS (8 rows)

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700

8 rows selected.

Cartesian product: 20x8=160 rows

EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
100	10	Administration
100	20	Marketing
100	50	Shipping
100	60	IT
100	80	Sales
100	90	Executive

...

160 rows selected.

ORACLE

Cartesian Products (continued)

A Cartesian product is generated if a join condition is omitted. The example on the slide displays employee last name and department name from the `EMPLOYEES` and `DEPARTMENTS` tables. Because no `WHERE` clause has been specified, all rows (20 rows) from the `EMPLOYEES` table are joined with all rows (8 rows) in the `DEPARTMENTS` table, thereby generating 160 rows in the output.

```
SELECT last_name, department_name dept_name
FROM   employees, departments;
```

Cartesian Products (continued)

LAST_NAME	DEPT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration
Ernst	Administration
Lorentz	Administration
Mourgos	Administration
Rajs	Administration
Davies	Administration
Matos	Administration
Vargas	Administration
Zlotkey	Administration
Abel	Administration
Taylor	Administration

▪ ▪ ▪

160 rows selected.

Types of Joins

Oracle SQL Joins (8*i* and prior):

- **Equijoin**
- **Non-equijoin**
- **Outer join**
- **Self join**

SQL: 1999 Compliant Joins:

- **Cross joins**
- **Natural joins**

ORACLE

5-9

Types of Joins

The Oracle9*i* database offers join syntax that is SQL: 1999 Compliant. Prior to the 9*i* release, the join syntax was different from the ANSI standards. The new SQL: 1999 Compliant join syntax does not offer any performance benefits over the Oracle proprietary join syntax that existed in prior releases.

What Is an Equijoin?

EMPLOYEES

EMPLOYEE_ID	DEPARTMENT_ID
100	90
101	90
102	90
103	60
104	60
107	60
124	50
141	50
142	50
143	50
144	50

...

19 rows selected.

Rows from the DEPARTMENTS table

DEPARTMENT_ID	DEPARTMENT_NAME
90	Executive
90	Executive
90	Executive
60	IT
60	IT
60	IT
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
80	Sales
80	Sales

...

Foreign key **Primary key**

Equijoins

To determine an employee's department name, you compare the value in the DEPARTMENT_ID column in the EMPLOYEES table with the DEPARTMENT_ID values in the DEPARTMENTS table. The relationship between the EMPLOYEES and DEPARTMENTS tables is an equijoin: that is, values in the DEPARTMENT_ID column on both tables must be equal. Frequently, this type of join involves primary and foreign key complements.

Note: Equijoins are also called simple joins or inner joins.

Retrieving Records with Equijoins

```
SELECT employees.employee_id, employees.last_name,
       employees.department_id, departments.department_id,
       departments.department_name
FROM   employees, departments
WHERE  employees.department_id = departments.department_id ;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	Whalen	10	10	Administration
201	Hartstein	20	20	Marketing
202	Fay	20	20	Marketing
124	Mourgos	50	50	Shipping
141	Rajs	50	50	Shipping
142	Davies	50	50	Shipping
143	Matos	50	50	Shipping
144	Vargas	50	50	Shipping
103	Hunold	60	60	IT
104	Ernst	60	60	IT
107	Lorentz	60	60	IT
149	Zlotkey	80	80	Sales
174	Abel	80	80	Sales

...

19 rows selected.

ORACLE

5-11

Retrieving Records with Equijoins

In the slide example:

- The `SELECT` clause specifies the column names to retrieve:
  - employee last name, employee ID, and department ID, which are columns in the `EMPLOYEES` table
  - department ID, and department name, which are columns in the `DEPARTMENTS` table
- The `FROM` clause specifies the two tables that the must be accessed:
  - `EMPLOYEES` table
  - `DEPARTMENTS` table
- The `WHERE` clause specifies how the tables are to be joined:
  - `EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID`

Because the `DEPARTMENT_ID` column is common to both tables, it must be prefixed by the table name to avoid ambiguity.

Qualifying Ambiguous Column Names

- **Use table prefixes to qualify column names that are in multiple tables.**
- **Improve performance by using table prefixes.**
- **Distinguish columns that have identical names but reside in different tables by using column aliases.**

ORACLE

5-12

Qualifying Ambiguous Column Names

You need to qualify the names of the columns in the `WHERE` clause with the table name to avoid ambiguity. Without the table prefixes, the `DEPARTMENT_ID` column could be from either the `DEPARTMENTS` table or the `EMPLOYEES` table. It is necessary to add the table prefix to execute your query.

If there are no common column names between the two tables, there is no need to qualify the columns. However, using the table prefix improves performance, because you tell the Oracle Server exactly where to find the columns.

The requirement to qualify ambiguous column names is also applicable to columns that may be ambiguous in other clauses, such as the `SELECT` clause or the `ORDER BY` clause.

Additional Search Conditions Using the AND Operator

EMPLOYEES

LAST_NAME	DEPARTMENT_ID
Whalen	10
Hartstein	20
Fay	20
Mourgos	50
Rajs	50
Davies	50
Matos	50
Vargas	50
Hunold	60
Ernst	60

...

19 rows selected.

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
60	IT
60	IT
60	IT
80	Sales
80	Sales
80	Sales

...

ORACLE

5-13

Additional Search Conditions

In addition to the join, you may have to specify other criteria for your `WHERE` clause. For example, to display only employee Matos' department number and department name, you need an additional condition in the `WHERE` clause.

```
SELECT last_name, employees.department_id,
        department_name
FROM   employees, departments
WHERE  employees.department_id = departments.department_id
AND    last_name = 'Matos';
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Matos	50	Shipping

Using Additional Search Conditions with a Join

```
SELECT employees.employee_id, employees.department_id,
       departments.department_name
FROM   employees, departments
WHERE  employees.department_id = departments.department_id
AND    employee_id = 100;
```

EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
100	90	Executive

Using Additional Search Conditions with a Join

In the slide:

- The SELECT clause specifies the column names to retrieve:
  - EMPLOYEE_ID, DEPARTMENT_ID which are columns in the EMPLOYEES table
  - DEPARTMENT_NAME, which is a column in the DEPARTMENTS table
- The FROM clause specifies the two tables that must be accessed:
  - EMPLOYEES table
  - DEPARTMENTS table
- The WHERE clause specifies how the tables are to be joined and which rows to retrieve:
  - EMPLOYEES.DEPARTMENT_ID=DEPARTMENTS.DEPARTMENT_ID
  - EMPLOYEE_ID=100

The additional condition in the WHERE clause specifies employee whose EMPLOYEE_ID = 100, as the employee whose data you want to retrieve.

Using Additional Search Conditions with a Join

```
SELECT last_name,job_id, employees.department_id,
       departments.department_name
FROM   employees, departments
WHERE  employees.department_id=departments.department_id
AND    job_id IN ('SA_REP','MK_REP');
```

LAST_NAME	JOB_ID	DEPARTMENT_ID	DEPARTMENT_NAME
Fay	MK_REP	20	Marketing
Abel	SA_REP	80	Sales
Taylor	SA_REP	80	Sales

ORACLE

Using Additional Search Conditions with a Join

For example, to display the employee's last name, job ID, department number, and department name for the sales representatives and the marketing representatives, you need an additional condition in the `WHERE` clause.

In the slide:

- The `SELECT` clause specifies the column names to retrieve:
  – `LAST_NAME`, `JOB_ID` and `DEPARTMENT_ID` which are columns in the `EMPLOYEES` table
  – `DEPARTMENT_NAME` which is a column in the `DEPARTMENTS` table
- The `FROM` clause specifies the two tables that must be accessed.
  – `EMPLOYEES` table
  – `DEPARTMENTS` table
- The `WHERE` clause specifies how the tables are to be joined and which rows to retrieve:
  – `EMPLOYEES.DEPARTMENT_ID=DEPARTMENTS.DEPARTMENT_ID`
  – `JOB_ID IN ('SA_REP', 'MK_REP')`

The additional condition in the `WHERE` clause specifies the sales representatives and the marketing representatives as the employees whose data you want to retrieve.

Table Aliases

Simplify queries by using table aliases.

```
SELECT employees.employee_id, employees.last_name,
       employees.department_id,departments.department_name
FROM   employees, departments
WHERE  employees.department_id = departments.department_id;
```

… can be written as ...

```
SELECT e.employee_id, e.last_name, e.department_id,
       departments.department_name
FROM   employees e, departments
WHERE  e.department_id = departments.department_id;
```


Table Aliases

Qualifying column names with table names can be very time consuming, particularly if table names are lengthy. You can use table aliases instead of table names. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help reduce SQL code so that it uses less memory.

Notice how table aliases are identified in the `FROM` clause in the example. The table name is specified in full, followed by a space and then the table alias. The `EMPLOYEES` table has the alias *e*.

Guidelines

- Table aliases can be up to 30 characters in length, but the shorter they are the better.
- If a table alias is used for a particular table name in the `FROM` clause, that table alias must be substituted for the table name throughout the `SELECT` statement. This is particularly useful in cases when the table names are long.
- Table aliases should be meaningful.
- The table alias is valid only for the current `SELECT` statement.
- The use of aliases increases readability of the SQL code
- You cannot use the optional 'AS' keyword for table alias as in a column alias

Using Table Aliases

```
SELECT  e.last_name,
        e.department_id, d.department_name
FROM    employees e, departments d
WHERE   e.department_id = d.department_id;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping
Rajs	50	Shipping
Davies	50	Shipping
Matos	50	Shipping
Vargas	50	Shipping
Hunold	60	IT
Ernst	60	IT
Lorentz	60	IT

...

19 rows selected.

ORACLE

Using Table Aliases

The example in the slide uses table aliases to specify the tables from which data has to be retrieved.

In the following example, notice the points:

- The same join condition is used as in the slide example but neither of the `DEPARTMENT_ID` columns is actually retrieved in the `SELECT` list.
- The query retrieves data on employees who work in the accounting department by using the additional condition in the `WHERE` clause

```
SELECT  e.last_name,d.department_name
FROM   employees e,  departments d
WHERE  e.department_id = d.department_id
AND d.department_name = ’Accounting’;
```

LAST_NAME	DEPARTMENT_NAME
Higgins	Accounting
Gietz	Accounting

Joining More than Two Tables

EMPLOYEES

LAST_NAME	DEPARTMENT_ID
Whalen	10
Hartstein	20
Fay	20
Mourgos	50
Rajs	50
Davies	50
Matos	50
Vargas	50
Hunold	60
Ernst	60

...

19 rows selected.

DEPARTMENTS

DEPARTMENT_ID	LOCATION_ID
10	1700
20	1800
50	1500
60	1400
80	2500
90	1700
110	1700
190	1700

8 rows selected.

LOCATIONS

LOCATION_ID	CITY
1400	Southlake
1500	South San Francisco
1700	Seattle
1800	Toronto
2500	Oxford

- **To join *n* tables together, you need a minimum of *n-1* join conditions.**
- **For example, to join three tables, a minimum of two joins is required.**

Additional Search Conditions

Sometimes you may need to join more than two tables. For example, to display the last name, the department name, and the city for each employee, you have to join the `EMPLOYEES`, `DEPARTMENTS`, and `LOCATIONS` tables.

```
SELECT e.last_name, d.department_name, l.city
FROM   employees e, departments d, locations l
WHERE  e.department_id = d.department_id
AND    d.location_id = l.location_id;
```

LAST_NAME	DEPARTMENT_NAME	CITY
Hunold	IT	Southlake
Ernst	IT	Southlake
Lorentz	IT	Southlake
Mourgos	Shipping	South San Francisco
Rajs	Shipping	South San Francisco
Davies	Shipping	South San Francisco

...

19 rows selected.

Non-Equijoins

EMPLOYEES

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000
Ernst	6000
Lorentz	4200
Mourgos	5800
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500
Zlotkey	10500
Abel	11000

...

20 rows selected.

JOB_GRADES

GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

6 rows selected.

salary in the `EMPLOYEES` table must be between lowest salary and highest salary in the `JOB_GRADES` Table.

ORACLE

5-19

Non-Equijoins

A non-equijoin is a join condition containing something other than an equality operator.

The relationship between the `EMPLOYEES` table and the `JOB_GRADES` table is an example of a non-equijoin. A relationship between the two tables is that the `SALARY` column in the `EMPLOYEES` table must be between the values in the `LOWEST_SALARY` and `HIGHEST_SALARY` columns of the `JOB_GRADES` table. The relationship is obtained using an operator other than equals (=).

Retrieving Records with Nonequijoins

```
SELECT  e.last_name, e.salary, j.grade_level
FROM    employees e, job_grades j
WHERE   e.salary
BETWEEN j.lowest_sal AND j.highest_sal;
```

LAST_NAME	SALARY	GRA
Matos	2600	A
Vargas	2500	A
Lorentz	4200	B
Mourgos	5800	B
Rajs	3500	B
Davies	3100	B
Whalen	4400	B
Hunold	9000	C
Ernst	6000	C
Taylor	8600	C
Grant	7000	C
Fay	6000	C

...

20 rows selected.

ORACLE

5-20

Nonequijoins (continued)

The example in the slide creates a nonequijoin to evaluate an employee's job grade. The salary must be between any pair of the low and high salary ranges in the `JOB_GRADES` table.

It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:

- None of the rows in the `JOB_GRADES` table contain grades that overlap. That is, the salary value for an employee must lie between the low salary and high salary values of one of the rows in the `JOB_GRADES` table.
- All of the employees' salaries lie within the limits provided by the `JOB_GRADES` table. That is, no employee earns less than the lowest value contained in the `LOWEST_SAL` column or more than the highest value contained in the `HIGHEST_SAL` column.

Note: Other operators such as `<=` and `>=` could be used, but `BETWEEN` is the simplest. Remember to specify the low value first and the high value last when using `BETWEEN`. Table aliases have been specified for performance reasons, not because of possible ambiguity.

Using Multiple Joins

```
SELECT e.last_name, e.department_id,
       d.department_name, e.salary, j.grade_level
FROM   employees e, departments d, job_grades j
WHERE  e.department_id=d.department_id
AND    e.salary BETWEEN j.lowest_sal and j.highest_sal ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME	SALARY	GRA
Matos	50	Shipping	2600	A
Vargas	50	Shipping	2500	A
Lorentz	60	IT	4200	B
Mourgos	50	Shipping	5800	B
Rajs	50	Shipping	3500	B
Davies	50	Shipping	3100	B
Whalen	10	Administration	4400	B
Hunold	60	IT	9000	C
Ernst	60	IT	6000	C
Taylor	80	Sales	8600	C
Fay	20	Marketing	6000	C

---
19 rows selected.

ORACLE

Joining More Than Two Tables (continued)

In the slide:

- The `SELECT` clause specifies the column names to retrieve:
  - `LAST_NAME`, `DEPARTMENT_ID`, and `SALARY`, which are columns in the `EMPLOYEES` table
  - `DEPARTMENT_NAME`, which is a column in the `DEPARTMENTS` table
  - `GRADE_LEVEL`, which is a column in the `JOB_GRADES` table
- The `FROM` clause specifies the three tables that must be accessed:
  - `EMPLOYEES` table (alias E)
  - `DEPARTMENTS` table (alias D)
  - `JOB_GRADES` (alias J)
- The `WHERE` clause specifies how the tables are to be joined:
  - `E.DEPARTMENT_ID=D.DEPARTMENT_ID`
  - `E.SALARY BETWEEN J.LOWEST_SAL AND J.HIGHEST_SAL`

Note: The number of joins must at least equal the number of tables minus one.

Self Joins

EMPLOYEES (WORKER)

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103
107	Lorentz	103
124	Mourgos	100
141	Rajs	124
142	Davies	124
143	Matos	124
144	Vargas	124
149	Zlotkey	100
174	Abel	149

...

19 rows selected.

EMPLOYEES (MANAGER)

EMPLOYEE_ID	LAST_NAME
100	King
100	King
100	King
100	King
100	King
101	Kochhar
101	Kochhar
102	De Haan
103	Hunold
103	Hunold
124	Mourgos
124	Mourgos

`MANAGER_ID` in the `WORKER` table is equal to `EMPLOYEE_ID` in the `MANAGER` table.

5-22

Joining a Table to Itself

Sometimes you need to join a table to itself. This type of a join is called as self join. To find the name of each employee's manager, you need to join the `EMPLOYEES` table to itself, or perform a self join. For example, to find the name of Whalen's manager, you need to:

- Find Whalen in the `EMPLOYEES` table by looking at the `LAST_NAME` column.
- Find the manager number for Whalen by looking at the `MANAGER_ID` column. Whalen's manager number is 101.
- Find the name of the manager with `EMPLOYEE_ID` 101 by looking at the `LAST_NAME` column. Kochhar's employee number is 101, so Kochhar is Whalen's manager.

In this process, you look in the table twice. The first time you look in the table to find Whalen in the `LAST_NAME` column and `MANAGER_ID` value of 101. The second time you look in the `EMPLOYEE_ID` column to find 101 and the `LAST_NAME` column to find Kochhar.

Joining a Table to Itself

```
SELECT worker.last_name || ' works for '
       || manager.last_name
FROM   employees worker, employees manager
WHERE  worker.manager_id = manager.employee_id ;
```

WORKER.LAST_NAME||'WORKSFOR'||MANAGER.LAST_NAME
Kochhar works for King
De Haan works for King
Mourgos works for King
Zlotkey works for King
Hartstein works for King
Whalen works for Kochhar
Higgins works for Kochhar
Hunold works for De Haan
Ernst works for Hunold
Lorentz works for Hunold
Rajs works for Mourgos
Davies works for Mourgos

...

19 rows selected.

ORACLE

5-23

Joining a Table to Itself (continued)

The slide example joins the `EMPLOYEES` table to itself. To simulate two tables in the `FROM` clause, there are two aliases, namely worker and manager, for the same table, `EMPLOYEES`.

In this example, the `WHERE` clause contains the join that means "where a worker's manager number matches the employee number for the manager."

Joining Tables Using SQL: 1999 Syntax

Use a join to query data from more than one table.

```
SELECT table1.column, table2.column
FROM   table1
[CROSS JOIN table2] |
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
ON(table1.column_name = table2.column_name)] |
[LEFT|RIGHT|FULL OUTER JOIN table2
ON (table1.column_name = table2.column_name)];
```

ORACLE

5-24

Defining Joins

Using the SQL: 1999 syntax, you can obtain the same results as in the prior pages.

In the syntax:

`table1.column`	Denotes the table and column from which data is retrieved
`CROSS JOIN`	Returns a cartesian product from the two tables
`NATURAL JOIN`	Joins two tables based on the same column name
`JOIN` table	
`USING column_name`	Performs an equi-join based on the `column_name`
`JOIN` table `ON`	
`table1.column_name = table2.column_name clause LEFT/RIGHT/FULL OUTER`	Performs an equi-join based on the condition in the `ON`

For more information, *see Oracle SQL Reference, "SELECT."*

Creating Cross Joins

- The `CROSS JOIN` clause produces the cross-product of two tables.
- This is the same as a Cartesian product between the two tables.

```
SELECT last_name, department_name
FROM   employees
CROSS JOIN departments ;
```

LAST_NAME	DEPARTMENT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration
Ernst	Administration
Lorentz	Administration
Mourgos	Administration
Rajs	Administration

...

160 rows selected.

ORACLE

Creating Cross Joins

The above example gives the same results as the following:

```
SELECT last_name, department_name
FROM employees, departments;
```

LAST_NAME	DEPARTMENT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration
Ernst	Administration
Lorentz	Administration
Mourgos	Administration
Rajs	Administration

...

160 rows selected.

Creating Natural Joins

- The `NATURAL JOIN` clause is based on all columns in the two tables that have the same name.
- It selects rows from the two tables that have equal values in all matched columns.
- If the columns having the same names have different data types then an error is returned.

5-26

Creating Natural Joins

It was not possible to do a join without explicitly specifying the columns in the corresponding tables in prior releases of Oracle. In Oracle9*i* it is possible to let the join be completed automatically based on columns in the two tables which have matching data types and names, using the keywords `NATURAL JOIN.`

Note: The join can only happen on columns having the same names and data types in both the tables. If the columns have the same name, but different data types, then the `NATURAL JOIN` syntax will cause an error.

Retrieving Records with Natural Joins

```
SELECT department_id, department_name,
       location_id, city
FROM   departments
NATURAL JOIN locations  ;
```

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
60	IT	1400	Southlake
50	Shipping	1500	South San Francisco
10	Administration	1700	Seattle
90	Executive	1700	Seattle
110	Accounting	1700	Seattle
190	Contracting	1700	Seattle
20	Marketing	1800	Toronto
80	Sales	2500	Oxford

8 rows selected.

ORACLE

5-27

Retrieving Records with Natural Joins

In the above example the `LOCATIONS` table is joined to the `DEPARTMENT` table by the `LOCATION_ID` column, which is the only column of the same name in both tables. If other common columns were present then the join would have used them all.

Equijoins

The natural join can also be written as an equijoin:

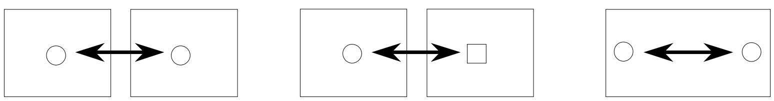
```
SELECT department_id, department_name,
       departments.location_id, city
FROM   departments, locations
WHERE  departments.location_id = locations.location_id;
```

Summary

```
SELECT   table1.column, table2.column
FROM     table1, table2
WHERE    table1.column1 = table2.column2;
```

- **Prefix the column name with the table name or alias if the same column name appears in more than one table**

Equijoin   **Nonequijoin**   **Selfjoin**



SQL: 1999 Compliant Joins

- **Cross joins**
- **Natural joins**

5-28

Summary

There are many ways to join tables. The common thread is to link them through a condition in the `WHERE` clause. The method you choose is based on the required result and the data structures you are using. Omission of the `WHERE` clause results in a Cartesian product, which displays all combinations of rows.

```
SELECT  table1.column, table2.column
FROM    table1, table2
WHERE   table1.column1 = table2.column2;
```

Table Aliases

- Table aliases speed up database access.
- Table aliases can help keep SQL code smaller, therefore conserving memory.

Types of Joins

- Equijoin
- Non-equijoin
- Outer join
- Self join

Using the SQL: 1999 joins, you can obtain the same results from more than one table

- Cross joins
- Natural joins

Practice 5 Overview

This practice covers the following topics:

- **Joining tables by using an equijoin**
- **Performing selfjoins**
- **Adding additional conditions**

Practice 5 Overview

This practice is intended to give you practical experience in extracting data from more than one table by joining and restricting rows in the `WHERE` clause.

Practice 5

1. Display the last name, department ID, and department name of all employees, in department name order.

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Higgins	110	Accounting
Gietz	110	Accounting
Whalen	10	Administration
King	90	Executive
Kochhar	90	Executive
De Haan	90	Executive
Hunold	60	IT
Ernst	60	IT
Lorentz	60	IT
Hartstein	20	Marketing
Fay	20	Marketing
Zlotkey	80	Sales
Taylor	80	Sales
Abel	80	Sales
LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Mourgos	50	Shipping
Rajs	50	Shipping
Davies	50	Shipping
Matos	50	Shipping
Vargas	50	Shipping

19 rows selected.

2. Display the last name, salary, and department name of all employees who earn more than $10,000.

LAST_NAME	SALARY	DEPARTMENT_NAME
Hartstein	13000	Marketing
Zlotkey	10500	Sales
Abel	11000	Sales
King	24000	Executive
Kochhar	17000	Executive
De Haan	17000	Executive
Higgins	12000	Accounting

7 rows selected.

Practice 5 (continued)

3. Display the last name, salary, and department name for all employees in the accounting department.

LAST_NAME	SALARY	DEPARTMENT_NAME
Higgins	12000	Accounting
Gietz	8300	Accounting

4. Display the last name, job, department name, and location ID for all employees whose office has the location ID 1400.

LAST_NAME	JOB_ID	DEPARTMENT_NAME	LOCATION_ID
Hunold	IT_PROG	IT	1400
Ernst	IT_PROG	IT	1400
Lorentz	IT_PROG	IT	1400

5. Display a list of employees including last name, job, salary, and grade level.

LAST_NAME	JOB_ID	SALARY	GRA
Matos	ST_CLERK	2600	A
Vargas	ST_CLERK	2500	A
Lorentz	IT_PROG	4200	B
Mourgos	ST_MAN	5800	B
Rajs	ST_CLERK	3500	B
Davies	ST_CLERK	3100	B
Whalen	AD_ASST	4400	B
Hunold	IT_PROG	9000	C
Ernst	IT_PROG	6000	C
Taylor	SA_REP	8600	C
Grant	SA_REP	7000	C
Fay	MK_REP	6000	C
Gietz	AC_ACCOUNT	8300	C
Zlotkey	SA_MAN	10500	D
LAST_NAME	JOB_ID	SALARY	GRA
Abel	SA_REP	11000	D
Hartstein	MK_MAN	13000	D
Higgins	AC_MGR	12000	D
King	AD_PRES	24000	E
Kochhar	AD_VP	17000	E
De Haan	AD_VP	17000	E

20 rows selected.

Practice 5 (continued)

6. Using question 5, show only employees in grade C.

LAST_NAME	JOB_ID	SALARY	GRA
Hunold	IT_PROG	9000	C
Ernst	IT_PROG	6000	C
Taylor	SA_REP	8600	C
Grant	SA_REP	7000	C
Fay	MK_REP	6000	C
Gietz	AC_ACCOUNT	8300	C

6 rows selected.

7. For employees in department 20, display the last name, department ID, the name of the employee's manager and department ID of their manager.

LAST_NAME	DEPARTMENT_ID	MANAGER	DEPARTMENT_ID
Hartstein	20	King	90
Fay	20	Hartstein	20

8. Find all employees who joined the company before their manager.

LAST_NAME	HIREDATE	MGR	HIREDATE
Whalen	17-SEP-1987	Kochhar	21-SEP-1989
Hunold	03-JAN-1990	De Haan	13-JAN-1993
Rajs	17-OCT-1995	Mourgos	16-NOV-1999
Davies	29-JAN-1997	Mourgos	16-NOV-1999
Matos	15-MAR-1998	Mourgos	16-NOV-1999
Vargas	09-JUL-1998	Mourgos	16-NOV-1999
Abel	11-MAY-1996	Zlotkey	29-JAN-2000
Taylor	24-MAR-1998	Zlotkey	29-JAN-2000
Grant	24-MAY-1999	Zlotkey	29-JAN-2000

9 rows selected.

Practice 5 (continued)

If you want an extra challenge, try the following exercises:

9. For each employee, display the last name, the last name of the employee's manager and the manager's department name.

LAST_NAME	MGR	MGR_DEPT
Fay	Hartstein	Marketing
Rajs	Mourgos	Shipping
Davies	Mourgos	Shipping
Matos	Mourgos	Shipping
Vargas	Mourgos	Shipping
Ernst	Hunold	IT
Lorentz	Hunold	IT
Abel	Zlotkey	Sales
Taylor	Zlotkey	Sales
Grant	Zlotkey	Sales
Kochhar	King	Executive
De Haan	King	Executive
Mourgos	King	Executive
Zlotkey	King	Executive
LAST_NAME	MGR	MGR_DEPT
Hartstein	King	Executive
Whalen	Kochhar	Executive
Higgins	Kochhar	Executive
Hunold	De Haan	Executive
Gietz	Higgins	Accounting

19 rows selected.

Practice 5 (continued)

10. Display the last name and the last name of the manager for all employees who work in the same department as their manager.

LAST_NAME	MGR
Kochhar	King
De Haan	King
Ernst	Hunold
Lorentz	Hunold
Rajs	Mourgos
Davies	Mourgos
Matos	Mourgos
Vargas	Mourgos
Abel	Zlotkey
Taylor	Zlotkey
Fay	Hartstein
Gietz	Higgins

12 rows selected.

11. Display the employee ID, last name, department ID, department name,and city for all employees whose last names begin with H.

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME	CITY
201	Hartstein	20	Marketing	Toronto
205	Higgins	110	Accounting	Seattle
103	Hunold	60	IT	Southlake

6

Aggregating Data by Using Group Functions

ORACLE

Objectives

After completing this lesson, you should be able to:

- Identify the available group functions
- Describe the use of group functions
- Use the `GROUP BY` clause to group data

6-2

Lesson Aim

This lesson further addresses functions. It focuses on obtaining summary information, such as averages, for groups of rows. It also discusses how to group rows in a table into smaller sets.

What Are Group Functions?

Group functions operate on sets of rows to give one result per group.

EMPLOYEES

DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
...	
20	13000
20	6000
110	12000
110	8300

The maximum salary in the EMPLOYEES table.

MAX(SALARY)
24000

ORACLE

Group Functions

Unlike single-row functions, group functions operate on sets of rows to give one result per group. These sets may be the whole table or the table split into groups.

Types of Group Functions

- **AVG**
- **COUNT**
- **MAX**
- **MIN**
- **SUM**

6-4

Group Functions

Each of the functions accepts an argument. The following table identifies the options that you can use in the syntax.

Function	**Description**
`AVG([DISTINCT|ALL]n)`	Average value of *n*, ignoring null values
`COUNT({*|[DISTINCT|ALL] expr})`	Number of rows, where *expr* evaluates to something other than null; count all selected rows using *, including duplicates and rows with nulls
`MAX([DISTINCT|ALL]expr)`	Maximum value of *expr*, ignoring null values
`MIN([DISTINCT|ALL]expr)`	Minimum value of *expr*, ignoring null values
`SUM([DISTINCT|ALL]n)`	Sum values of *n*, ignoring null values

Guidelines for Using Group Functions

Many aggregate functions accept these options:

- **DISTINCT**
- **ALL**
- **NVL**

Guidelines for Using Group Functions

- DISTINCT makes the function consider only nonduplicate values; ALL makes it consider every value including duplicates. The default is ALL.
- The data types for the arguments can be CHAR, VARCHAR2, NUMBER, or DATE.
- All group functions except COUNT(*) ignore null values. To substitute a value for null values, use the NVL function.

Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

```
SELECT AVG(salary) , SUM(salary)
FROM employees
WHERE job_id = 'ST_CLERK';
```

AVG(SALARY)	SUM(SALARY)
2925	11700

Using the AVG and SUM Functions

You can use AVG and SUM functions against columns that can store numeric data. The example in the slide uses the AVG and SUM functions to display the average salary and sum of monthly salaries for all clerks.

Note: You can use AVG and SUM functions only with numeric data types.

Using the MIN and MAX Functions

You can use MIN and MAX for any data type.

```
SELECT TO_CHAR(MIN(hire_date),'DD-MON-YYYY'),
       TO_CHAR(MAX(hire_date),'DD-MON-YYYY')
FROM employees;
```

TO_CHAR(MIN	TO_CHAR(MAX
17-JUN-1987	29-JAN-2000

Using MIN and MAX Functions with Dates and Character Data Types

You can use MAX and MIN functions for any data type. The example in the slide uses the MAX and MIN functions with date data types to display the hire dates of the most junior and most senior employee.

The following example displays the names of the first and last employees in an alphabetized list of all employees:

```
SELECT MIN(last_name), MAX(last_name)
FROM employees;
```

MIN(LAST_NAME)	MAX(LAST_NAME)
Abel	Zlotkey

Using the MIN and MAX Functions

You can use MIN and MAX for any data type.

```
SELECT MIN(salary) AS "Lowest Salary",
       MAX(salary) AS "Highest Salary"
FROM employees;
```

Lowest Salary	Highest Salary
2500	24000

Using the MIN and MAX Functions with Numeric Data

The example in the slide displays the lowest and the highest salary paid to an employee.

The following example displays the lowest salary boundary and highest salary boundary for employees:

```
SELECT MIN(salary) AS "Lowest Salary",
       MAX(salary) AS "Highest Salary"
FROM employees;
```

Lowest Salary	Highest Salary
2500	24000

Using the COUNT Function

COUNT(*) returns the number of rows in a query.

```
SELECT COUNT(*)
FROM employees
WHERE department_id = 20;
```

COUNT(*)
2

Using the COUNT Function

The `COUNT` function has two formats:

- `COUNT(*)`: Returns the number of rows in a query, including duplicate rows and rows containing null values.
- `COUNT(`*`expr`*`)`: Returns the number of nonnull rows in the column identified by expr.

The example in the slide uses `COUNT(*)` to display the number of employees in department 20.

The following example displays the total number of employees and the total number of managers in the `EMPLOYEES` table. Observe that the total number of managers is 19, because the employee with the employee ID 100 does not have a manager (that is does not have a value in the `MANAGER_ID` column).

```
SELECT COUNT(*), COUNT(manager_id)
FROM employees;
```

COUNT(*)	COUNT(MANAGER_ID)
20	19

Using the COUNT Function

COUNT(expr) returns the number of nonnull rows.

```
SELECT COUNT(commission_pct)
FROM employees
WHERE department_id = 80;
```

COUNT(COMMISSION_PCT)
3

Using the COUNT Function (continued)

The example in the slide displays the number of employees in department 80 who can earn a commission. The following example displays the number of departments in the EMPLOYEES table:

```
SELECT COUNT(department_id)
FROM employees;
```

COUNT(DEPARTMENT_ID)
19

The following example displays the number of distinct departments in the EMPLOYEES table:

```
SELECT COUNT(DISTINCT(department_id)) AS "Working Depts"
FROM   employees;
```

Working Depts
7

Group Functions and Null Values

Group functions ignore null values in the column.

```
SELECT AVG(commission_pct)
FROM   employees;
```

AVG(COMMISSION_PCT)
.2125

Group Functions and Null Values

All group functions except `COUNT(*)` ignore null values in the column. In the example in the slide, the average is calculated based only on the rows in the table where a valid value is stored in the `COMMISSION_PCT` column. The average is calculated as total commission being paid to all employees divided by the number of employees receiving commission.

There are four employees who receive commission.

Using the NVL Function with Group Functions

The NVL function forces group functions to include null values.

```
SELECT AVG(NVL(commission_pct,0))
FROM   employees;
```

AVG(NVL(COMMISSION_PCT,0))
.0425

Group Functions and Null Values (continued)

The NVL function forces group functions to include null values. In the example in the slide, the average is calculated based on all rows in the table regardless of whether null values are stored in the COMMISSION_PCT column. The average is calculated as total commission being paid to all employees divided by the total number of employees in the company, which is 20.

Using the NVL Function with Group Functions

Average commission for all people hired in 1999

```
SELECT  AVG(NVL(commission_pct,0))
FROM    employees
WHERE   hire_date
BETWEEN TO_DATE('01-JAN-1999','DD-MON-YYYY')
AND     TO_DATE('31-DEC-1999','DD-MON-YYYY');
```

AVG(NVL(COMMISSION_PCT,0))
.05

Group Functions and Null Values (continued)

The example in the slide calculates the average commission paid to employees who were hired in 1999, regardless of whether null values are stored in the `COMMISSION_PCT` column. The average is calculated as total commission being paid to all employees hired in 1999 (0.15) divided by the total number of employees who were hired in 1999, which is 1.

Creating Groups of Data

EMPLOYEES

DEPARTMENT_ID	SALARY
10	4400
20	13000
20	6000
50	5800
50	3500
50	3100
50	2500
50	2600
60	9000
60	6000
60	4200
80	10500
80	8600
80	11000
90	24000
90	17000

...

20 rows selected.

4400

9500

3500

6400

10033

The average salary in `EMPLOYEES` table for each department

DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

ORACLE

6-14

Groups of Data

Until now, all group functions have treated the table as one large group of information. At times, you need to divide the table of information into smaller groups. This can be done by using the `GROUP BY` clause.

Creating Groups of Data: GROUP BY Clause

Use the GROUP BY clause to divide rows in a table into smaller groups.

```
SELECT        column, group_function
FROM          table
[WHERE        condition]
[GROUP BY     group_by_expression]
[ORDER BY     column];
```


The GROUP BY Clause

You can use the GROUP BY clause to divide the rows in a table into groups. You can then use the group functions to return summary information for each group.

In the syntax:

group_by_expression: Specifies columns whose values determine the basis for grouping rows

Guidelines

- If you include a group function in a SELECT clause, you cannot select individual results as well unless the individual column appears in the GROUP BY clause. You will receive an error message if you fail to include the column list.
- You can use a WHERE clause to exclude rows before dividing them into groups.
- You must include the columns in the GROUP BY clause.
- You cannot use a column alias in the GROUP BY clause.
- By default, rows are sorted by ascending order of the columns included in the GROUP BY list. You can override this by using the ORDER BY clause.

Using the GROUP BY Clause

All columns in the SELECT list that are not in group functions must be in the GROUP BY clause.

```
SELECT   department_id, AVG(salary)
FROM     employees
GROUP BY department_id  ;
```

DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

8 rows selected.

Using the GROUP BY Clause

When using the GROUP BY clause, make sure that all columns in the SELECT list that are not in the group functions are included in the GROUP BY clause. The example in the slide displays the department ID and average salary for each department. Here is how the statement is evaluated:

- The SELECT clause specifies the columns to be retrieved:
  - Department ID column in the EMPLOYEES table
  - The average of all the salaries in the group you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMPLOYEES table.
- The WHERE clause specifies the rows to be retrieved. Because there is no WHERE clause, by default all rows are retrieved.
- The GROUP BY clause specifies how the rows should be grouped. Because the rows are being grouped by department ID, the AVG function that is being applied to the salary column calculates the average salary for each department.

Using the GROUP BY Clause

The `GROUP BY` column does not have to be in the `SELECT` list.

```
SELECT   AVG(salary)
FROM     employees
GROUP BY department_id ;
```

	AVG(SALARY)
	4400
	9500
	3500
	6400
	10033.3333
	19333.3333
	10150
	7000

8 rows selected.

ORACLE

6-17

Using the GROUP BY Clause (continued)

The `GROUP BY` column does not have to be in the `SELECT` clause. For example, the `SELECT` statement in the slide displays the average salary for each department without displaying the respective department ids. However, without the department ids, the results do not look meaningful.

Using the GROUP BY Clause

Display the number of people in each department.

```
SELECT   department_id, COUNT(*)
         AS "Dept Employees"
FROM     employees
GROUP BY department_id;
```

DEPARTMENT_ID	Dept Employees
10	1
20	2
50	5
60	3
80	3
90	3
110	2
	1

8 rows selected.

Using the GROUP BY Clause (continued)

The example in the slide above displays the number of employees who work in each department. As in the example on the previous slide, the DEPARTMENT_ID column is not required in the SELECT list, but it makes the output clear and meaningful.

The following example displays the lowest and highest salary for each job title:

```
SELECT job_id, MIN(salary), MAX(salary)
FROM employees
GROUP BY job_id;
```

JOB_ID	MIN(SALARY)	MAX(SALARY)
AC_ACCOUNT	8300	8300
AC_MGR	12000	12000
AD_ASST	4400	4400
AD_PRES	24000	24000
AD_VP	17000	17000
IT_PROG	4200	9000
MK_MAN	13000	13000

...

12 rows selected.

Using a Group Function in the ORDER BY Clause

```
SELECT  department_id, AVG(salary)
FROM  employees
GROUP BY department_id
ORDER BY AVG(salary);
```

DEPARTMENT_ID	AVG(SALARY)
50	3500
10	4400
60	6400
	7000
20	9500
80	10033.3333
110	10150
90	19333.3333

8 rows selected.

Using Group Functions in the ORDER BY Clause

You can use group functions in the `ORDER BY` clause to order the output of the groups that are displayed. In the example in the slide, the report displays the average salary for each department in ascending order.

In the following example, the report displays the maximum salary in each department. The output is ordered by the maximum salary amounts from lowest to highest:

```
SELECT department_id, MAX(salary)
FROM   employees
GROUP BY department_id
ORDER BY  MAX(salary);
```

Using Group Functions in the `ORDER BY` Clause (continued)

DEPARTMENT_ID	MAX(SALARY)
10	4400
50	5800
	7000
60	9000
80	11000
110	12000
20	13000
90	24000

8 rows selected.

Illegal Queries Using Group Functions

Any column or expression in the `SELECT` list that is not an aggregate function must be in the `GROUP BY` clause.

```
SELECT department_id, COUNT(last_name)
FROM employees;
```

```
SELECT department_id, COUNT(last_name)
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

Column missing in the GROUP BY clause

6-21

ORACLE

Illegal Queries Using Group Functions

Whenever you use a mixture of individual items (`DEPARTMENT_ID`) and group functions (`COUNT`) in the same `SELECT` statement, you must include a `GROUP BY` clause that specifies the individual items (in this case, `DEPARTMENT_ID`). If the `GROUP BY` clause is missing, the error message "not a single-group group function" appears and an asterisk (`*`) points to the offending column. You can correct the error by adding the `GROUP BY` clause.

```
SELECT department_id, COUNT(last_name)
FROM employees
GROUP BY department_id;
```

Illegal Queries Using Group Functions (continued)

DEPARTMENT_ID	COUNT(LAST_NAME)
10	1
20	2
50	5
60	3
80	3
90	3
110	2
	1

8 rows selected.

Summary

```
SELECT      column, group_function
FROM        table
[WHERE      condition]
[GROUP BY   group_by_expression]
[ORDER BY   column];
```


Summary

Group functions available in SQL include:

- AVG
- COUNT
- MAX
- MIN
- SUM

Use the GROUP BY clause to create subgroups.

Oracle Server evaluates the clauses in the following order:

- If the statement contains a WHERE clause, the server establishes the candidate rows.
- The server identifies the groups specified in the GROUP BY clause.

Practice 6 Overview

This practice covers the following topics:

- **Showing different queries that use group functions**
- **Grouping by rows to achieve more than one result**

ORACLE

6-24

Practice 6 Overview

At the end of this practice, you should be familiar with using group functions.

Note: Column aliases are used for the queries.

Practice 6

1. Determine the validity of the following statements. Circle either True or False.

   a. Group functions work across many rows to produce one result.

   True/False

   b. Group functions include nulls in calculations.

   True/False

2. Find the earliest hire date of an employee.

EARLIEST
17-JUN-1987

3. Find the highest salary paid to an employee.

MAX_SALARY
24000

4. Find the total monthly salary paid to all clerks.

CLERK_PAYROLL
11700

5. Display the maximum salary, the minimum salary, and the difference between them for staff who were hired in 1999.

MAX(SALARY)	MIN(SALARY)	DIFFERENCE
7000	4200	2800

6. Find the minimum, average, and maximum salaries of all employees.

LOWEST	AVERAGE	HIGHEST
2500	8775	24000

Practice 6 (continued)

7. Display the minimum and maximum salary for each job ID.

JOB_ID	MIN_SAL	MAX_SAL
AC_ACCOUNT	8300	8300
AC_MGR	12000	12000
AD_ASST	4400	4400
AD_PRES	24000	24000
AD_VP	17000	17000
IT_PROG	4200	9000
MK_MAN	13000	13000
MK_REP	6000	6000
SA_MAN	10500	10500
SA_REP	7000	11000
ST_CLERK	2500	3500
ST_MAN	5800	5800

12 rows selected.

If you want an extra challenge, try the following exercises:

8. Determine the number of managers without listing them.

Note: Think about the `MANAGER_ID` column rather than the `JOB_ID` column when determining the number of managers

No. of managers
8

Practice 6 (continued)

9. Find the average monthly salary and average annual income for each job ID. Remember that only sales people earn commission.

JOB_ID	AVERAGE_SALARY	AVERAGE_ANNUAL_INCOME
AC_ACCOUNT	8300	99600
AC_MGR	12000	144000
AD_ASST	4400	52800
AD_PRES	24000	288000
AD_VP	17000	204000
IT_PROG	6400	76800
MK_MAN	13000	156000
MK_REP	6000	72000
SA_MAN	10500	151200
SA_REP	8866.66667	130680
ST_CLERK	2925	35100
ST_MAN	5800	69600

12 rows selected.

10. Display the department ID and the total number of employees working for each department. Order the results in the descending order of the number of employees in each department.

DEPARTMENT_ID	TOTAL_EMPLOYEES
50	5
60	3
80	3
90	3
20	2
110	2
10	1
	1

8 rows selected.

Notes Page

Notes Page

7

Writing Subqueries

Objectives

After completing this lesson, you should be able to do the following:

- **Describe the types of problems that subqueries can solve**
- **Define subqueries**
- **List the types of subqueries**
- **Write single-row and multiple-row subqueries**

ORACLE

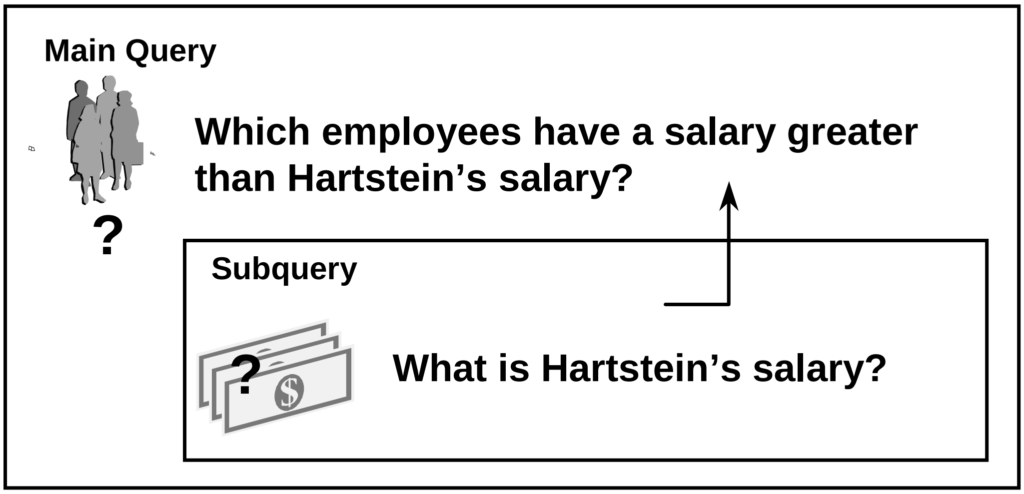
7-2

Lesson Aim

In this lesson you will learn about more advanced features of the `SELECT` statement. You can write subqueries in the `WHERE` clause of another SQL statement to obtain values based on an unknown conditional value. This lesson covers single-row subqueries and multiple-row subqueries.

Using a Subquery to Solve a Problem

Who has a salary greater than Hartstein's?



Using a Subquery to Solve a Problem

Suppose that you want to write a query to find out who earns a salary greater than Hartstein's salary. To solve this problem, you need two queries: one query to find out what Hartstein earns and a second query to find out who earns more than that amount.

You can solve this problem by combining the two queries, placing one query inside the other query.

An inner query, or subquery, returns a value that is used by the outer query or main query. Using a subquery is equivalent to performing two sequential queries and using the result of the first query as the search value in the second query.

Subqueries

```
SELECT   select_list
FROM     table
WHERE    expr operator
                     (SELECT      select_list
                      FROM        table);
```

- **The subquery (inner query) executes once before the main query.**
- **The result of the subquery is used by the main query (outer query).**

ORACLE

Using a Subquery

"Who has a salary greater than Hartstein'?"

```
SELECT last_name
FROM   employees           13000
WHERE  salary >
       (SELECT salary
        FROM   employees
        WHERE  last_name='Hartstein');
```

LAST_NAME
King
Kochhar
De Haan

7-5

Subqueries

A subquery is a `SELECT` statement that is embedded in a clause of another `SELECT` statement.

In the syntax:

`operator` Includes a comparison operator such as >, =, or `IN`

Note: Comparison operators fall into two classes: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators (`IN`, `ANY`, `ALL`).

The subquery is often called a nested `SELECT`, sub-`SELECT`, or inner `SELECT` statement. The subquery generally executes first, and its output is used to complete the query condition for the main or outer query.

Guidelines for Using Subqueries

- **Enclose subqueries in parentheses.**
- **Place subqueries on the right side of the comparison operator.**
- **Use single-row operators with single-row subqueries.**

ORACLE

7-6

Guidelines for Using Subqueries

- You must enclose a subquery in parentheses.
- A subquery should appear on the right side of the comparison operator.
- You must use a single row-operator with a single-row subquery

Types of Subqueries

- Single-row subquery

- Multiple-row subquery

7-7

Types of Subqueries

- Single-row subqueries return only one row from the inner `SELECT` statement.
- Multiple-row subqueries return more than one row from the inner `SELECT` statement.

Single-Row Subqueries

- Return only one row
- Use single-row comparison operators

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

7-8

Single-Row Subqueries

A single-row subquery returns one row from the inner SELECT statement. This type of subquery uses a single-row operator as listed in the slide

The following example displays the employees whose job ids are the same as that of employee 103:

```
SELECT   last_name, job_id
FROM     employees
WHERE    job_id =
                 (SELECT  job_id
                   FROM   employees
                   WHERE  employee_id = 103);
```

LAST_NAME	JOB_ID
Hunold	IT_PROG
Ernst	IT_PROG
Lorentz	IT_PROG

Executing Single-Row Subqueries

Who works in the same department as King?

```
SELECT last_name, department_id
FROM   employees
WHERE  department_id =
              (SELECT department_id
               FROM   employees
               WHERE  last_name='King');
```

90

LAST_NAME	DEPARTMENT_ID
King	90
Kochhar	90
De Haan	90

ORACLE

Using a Subquery

In the slide, the inner query determines the department number in which King works. The outer query takes the result of the inner query and uses it to display all the employees who work in the same department.

Executing Single-Row Subqueries

Who has the same manager as Ernst?

```
SELECT last_name, manager_id
FROM   employees                103
WHERE  manager_id =
                 (SELECT manager_id
                  FROM   employees
                  WHERE  last_name='Ernst');
```

LAST_NAME	MANAGER_ID
Ernst	103
Lorentz	103

Using a Subquery (continued)

In the example in the slide, the inner query determines the manager of Ernst. The outer query takes the result of the inner query (the employee ID of Ernst's manager) and uses this result to display all the employees who have the same manager as Ernst.

The following example finds the employees who have been in the organization longer than employee with the `EMPLOYEE_ID = 103` (Hunold):

```
SELECT last_name
FROM   employees
WHERE  hire_date <
                  (SELECT hire_date
                   FROM   employees
                   WHERE  employee_id = 103);
```

LAST_NAME
King
Kochhar
Whalen

Executing Single-Row Subqueries

Who has the same job as employee 176 and earns a higher salary than employee 143?

```
SELECT   last_name, job_id,salary
FROM     employees
WHERE    job_id =
                 (SELECT  job_id
                  FROM employees
                  WHERE employee_id = 176)

 AND salary >
                 (SELECT salary
                  FROM employees
                  WHERE employee_id = 143);
```

SA_REP

2600

LAST_NAME	JOB_ID	SALARY
Abel	SA_REP	11000
Taylor	SA_REP	8600
Grant	SA_REP	7000

7-11

Executing Single-Row Subqueries

The example in the slide displays employees whose job ID is the same as that of employee 176 and whose salary is greater than that of employee 143.

A `SELECT` statement can be considered as a query block. The example consists of three query blocks: the outer query and two inner queries. The inner query blocks are executed first, producing the query results: SA_REP and 2600, respectively. The outer query block then uses those values to complete its search conditions.

Because the inner queries return single values (SA_REP and 2600, respectively), this SQL statement is called a single-row subquery.

Using Group Functions in a Subquery

Display all employees who earn the minimum salary.

```
SELECT last_name, job_id, salary
FROM employees                  2500
WHERE salary =
                (SELECT MIN(salary)
                 FROM employees);
```

LAST_NAME	JOB_ID	SALARY
Vargas	ST_CLERK	2500

ORACLE

7-12

Using Group Functions in a Subquery

You can display data from a main query by using a group function in a subquery to return a single row. You place the subquery in parentheses and after the comparison operator.

The example in the slide displays the last name, job ID, and salary of all employees whose salary is equal to the minimum salary. The `MIN` group function returns a single value (2500) to the outer query.

The following example displays the employees who earn a salary greater than the average salary of a `AD_VP`:

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  salary >
               (SELECT AVG(salary)
                FROM   employees
                WHERE  job_id='AD_VP');
```

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000

Will This Statement Work?

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id =
                (SELECT job_id
                 FROM employees
                 WHERE last_name ='Smythe');
```

```
no rows selected
```

Subquery returns no values

Errors in Subqueries (continued)

Another common error in subqueries is no rows being returned by the inner query.

In the SQL statement in the slide, the subquery contains a `WHERE (LAST_NAME='Smythe')` clause. Presumably, the intention is to find the employee whose last name is Smythe. The statement seems to be correct but selects no rows when executed.

There is no employee with the last name as Smythe. So the subquery returns no rows. The outer query takes the result of the subquery and uses it in its `WHERE` clause.

What Is Wrong with This Statement?

```
SELECT employee_id, last_name
FROM   employees
WHERE  salary =
      (SELECT   MIN(salary)
       FROM      employees
       GROUP BY  department_id);
```

```
(SELECT MIN(salary)
    *
ERROR at line 4:
ORA-01427: single-row subquery returns more than one row
```

Single-row operator with multiple-row subquery

Errors in Subqueries

One common error in subqueries is more than one row returned for a single-row subquery. In the SQL statement in the slide, the subquery contains a `GROUP BY (DEPARTMENT_ID)` clause, which implies that the subquery will return multiple rows, one for each group it finds. In this case, the result of the subquery is 4400,6000,2500,4200,8600,7000,17000, and 8300.

The outer query takes the results of the subquery (4400,6000,2500,4200,8600,7000,17000, and 8300) and uses these results in its `WHERE` clause. The `WHERE` clause contains an equal (=) operator, a single-row comparison operator expecting only one value. The = operator cannot accept more than one value from the subquery and therefore generates the error.

Multiple-Row Subqueries

- **Multiple-row subqueries return more than one row.**
- **Use the `IN` multiple-row comparison operator to compare an expression to any member in the list that a subquery returns.**

7-15

Multiple-Row Subqueries

Subqueries that return more than one row are called multiple-row subqueries. You use a multiple-row operator, instead of a single-row operator, with a multiple-row subquery. The multiple-row operator expects one or more values. This lesson deals with the `IN` multiple-row comparison operator. There are two other multiple-row comparison operators, `ANY` and `ALL`, which are covered in other Oracle SQL courses.

Using Group Functions in a Multiple-Row Subquery

Display all employees who earn the same salary as the maximum salary for each department.

```
SELECT   last_name, salary, department_id
FROM     employees
WHERE    salary IN
                 (SELECT   MAX(salary)
                  FROM     employees
                  GROUP BY department_id);
```

4400,13000,5800,9000,11000,24000,12000

LAST_NAME	SALARY	DEPARTMENT_ID
Whalen	4400	10
Mourgos	5800	50
Grant	7000	
Hunold	9000	60
Abel	11000	80

...

8 rows selected.

Using Group Functions in a Multiple-Row Subquery

The example in the slide displays the employees who earn the same salary as the maximum salary for the departments.The inner query is executed first, producing a result that contains eight rows: 4400,13000,5800,9000,11000,24000, 12000,and 7000. These numbers represent the highest salary in each department. The main query block then uses the values returned by the inner query to complete its search condition. In fact, the main query would look like the following to the Oracle server:

```
SELECT    last_name, salary, department_id
FROM      employees
WHERE     salary IN (4400,13000,5800,9000,11000,24000,12000,7000);
```

Using Group Functions in a Multiple-Row Subquery

Display the employees who were hired on the same date as the longest serving employee in any department.

```
SELECT  last_name, salary, department_id,
        TO_CHAR(hire_date,'DD-MON-YYYY')HIREDATE
FROM    employees
WHERE   hire_date IN
                    (SELECT   MIN(hire_date)
                     FROM     employees
                     GROUP BY department_id);
```

LAST_NAME	SALARY	DEPARTMENT_ID	HIREDATE
King	24000	90	17-JUN-1987
Whalen	4400	10	17-SEP-1987
Hunold	9000	60	03-JAN-1990
Higgins	12000	110	07-JUN-1994

...

9 rows selected.

ORACLE

Using Group Functions in a Multiple-Row Subquery (continued)

The example in the slide displays the employees who were hired on the same date as the longest serving employee in any department.

The inner query is executed first, producing a result that contains eight rows: 17-SEP-87,17-FEB-96, 17-OCT-95,03-JAN-90,11-MAY-96,17-JUN-87, 07-JUN-94 and 24-MAY-99. These dates represent the hire dates of the first employees in each department. The main query block then uses the values returned by the inner query to complete its search condition. In fact, the main query would look like the following to the Oracle Server:

```
SELECT   last_name, salary, department_id, hire_date
FROM     employees
WHERE    hire_date IN ('17-SEP-87','17-FEB-96','17-OCT-95',
                       '03-JAN-90','11-MAY-96','17-JUN-87',
                       '07-JUN-94', '24-MAY-99');
```

Summary

Subqueries are useful when a query is based on unknown values.

```
SELECT   select_list
FROM     table
WHERE    expr operator
                   (SELECT select_list
                    FROM   table);
```


Summary

A subquery is a `SELECT` statement that is embedded in a clause of another SQL statement. Subqueries are useful when a query is based on unknown criteria.

Subqueries have the following characteristics:

- Can pass one row of data to a main statement that contains a single-row operator, such as =, <>, >, >=, <, or <=
- Can pass multiple rows of data to a main statement that contains a multiple-row operator, such as `IN`
- Are processed first by the Oracle server, and the `WHERE` clause uses the results
- Can contain group functions

Practice 7 Overview

This practice covers creating subqueries to query values based on unknown criteria.

Practice 7 Overview

This practice gives you experience in using nested `SELECT` statements in complex queries. You may want to consider creating the inner query first for these questions. Make sure that it runs and produces the data you expect before coding the outer query.

Practice 7

1. Answer the following questions:

   a. Which query runs first with a subquery?

   b. You cannot use the equal operator if the inner query returns more than one value.

      True/False

      i. If the answer is true, why, and what operator should be used?

      ii. If the answer is false, why?

2. Display the last name, manager ID, and salary for all employees in the same department as Matos.

LAST_NAME	MANAGER_ID	SALARY
Mourgos	100	5800
Rajs	124	3500
Davies	124	3100
Matos	124	2600
Vargas	124	2500

3. Display the employee ID, last name, and salary for all employees with a salary above the average salary.

EMPLOYEE_ID	LAST_NAME	SALARY
100	King	24000
101	Kochhar	17000
102	De Haan	17000
103	Hunold	9000
149	Zlotkey	10500
174	Abel	11000
201	Hartstein	13000
205	Higgins	12000

8 rows selected.

Practice 7 (continued)

4. Display the last name and salary for all employees who have the same manager as Zlotkey.

LAST_NAME	SALARY
Kochhar	17000
De Haan	17000
Mourgos	5800
Zlotkey	10500
Hartstein	13000

5. Find the employees who earn the same salary as the highest salary in each job ID. Sort in the descending order of salary.

LAST_NAME	JOB_ID	HIGHEST_SALARY
King	AD_PRES	24000
Kochhar	AD_VP	17000
De Haan	AD_VP	17000
Hartstein	MK_MAN	13000
Higgins	AC_MGR	12000
Abel	SA_REP	11000
Zlotkey	SA_MAN	10500
Hunold	IT_PROG	9000
Gietz	AC_ACCOUNT	8300
Ernst	IT_PROG	6000
Fay	MK_REP	6000
Mourgos	ST_MAN	5800
Whalen	AD_ASST	4400
Rajs	ST_CLERK	3500

14 rows selected.

Practice 7 (continued)

6. Find the employees who earn the same salary as the lowest salary for a job. Sort in the ascending order of salary.

LAST_NAME	JOB_ID	LOWEST_SALARY
Vargas	ST_CLERK	2500
Lorentz	IT_PROG	4200
Whalen	AD_ASST	4400
Mourgos	ST_MAN	5800
Ernst	IT_PROG	6000
Fay	MK_REP	6000
Grant	SA_REP	7000
Gietz	AC_ACCOUNT	8300
Zlotkey	SA_MAN	10500
Higgins	AC_MGR	12000
Hartstein	MK_MAN	13000
Kochhar	AD_VP	17000
De Haan	AD_VP	17000
King	AD_PRES	24000

14 rows selected.

7. Display all the employees who have worked longer than Gietz.

LAST_NAME	HIREDATE
King	17-JUN-1987
Kochhar	21-SEP-1989
De Haan	13-JAN-1993
Hunold	03-JAN-1990
Ernst	21-MAY-1991
Whalen	17-SEP-1987

6 rows selected.

Practice 7 (continued)

If you want an extra challenge, try the following exercises:

8. Display the last name and job ID for all the employees (excluding salesmen) with an annual salary greater than the average annual remuneration `AVG(12*salary*(1+NVL(commission_pct,0)))` for salesmen.

Hint: (`JOB_ID = 'SA_REP'`)

LAST_NAME	JOB_ID
King	AD_PRES
Kochhar	AD_VP
De Haan	AD_VP
Hartstein	MK_MAN
Higgins	AC_MGR

9. Display the names and salaries for all employees who work out of the Oxford office.

Hint: Use the `LOCATIONS` table to retrieve the city

OXFORDTEAM	SALARY
Zlotkey	10500
Abel	11000
Taylor	8600

Practice 7 (continued)

10. Display the employee ID and last names for all employees who report to King.

EMPLOYEE_ID	LAST_NAME
101	Kochhar
102	De Haan
124	Mourgos
149	Zlotkey
201	Hartstein

11. Display all the employees whose manager works in department 20.

LAST_NAME
Fay

12. Display the department ID, last names and job ids for all employees who work in the sales department.

DEPARTMENT_ID	LAST_NAME	JOB_ID
80	Zlotkey	SA_MAN
80	Abel	SA_REP
80	Taylor	SA_REP

8

*i*SQL*Plus

Objectives

After completing this lesson, you should be able to do the following:

- Produce queries that require an input variable
- Customize the *i*SQL*Plus environment
- Produce more readable output
- Create and execute script files

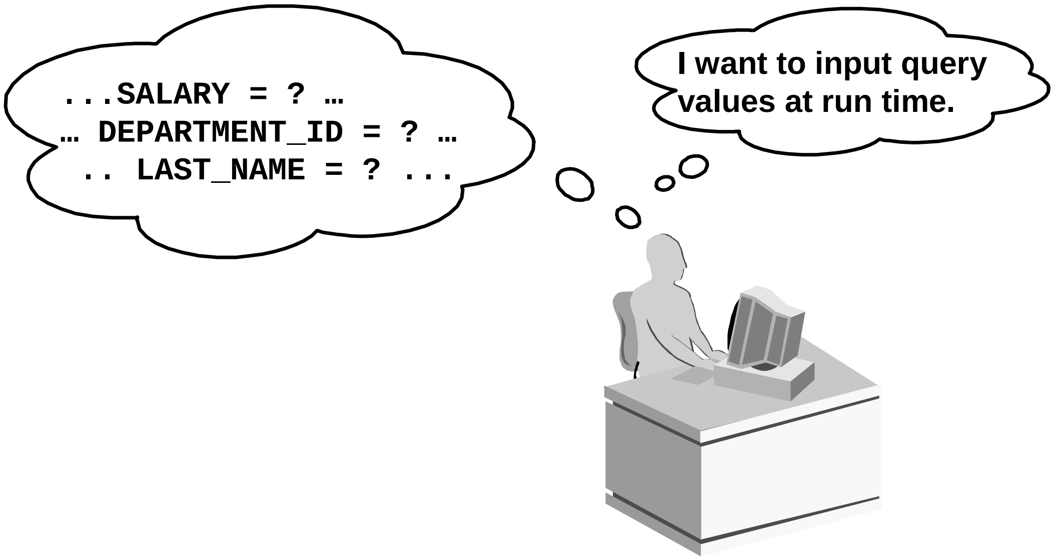
ORACLE

8-2

Lesson Aim

In this lesson, you will learn how to include *i*SQL*Plus commands to produce more readable SQL output.

Interactive Reports



User

8-3

Interactive Reports

The examples so far have not been interactive in any way. The user triggers the report and the report runs without further prompting. The range of data is determined by the fixed `WHERE` clause in the *i*SQL*Plus script file.

Using *i*SQL*Plus, you can create reports that prompt users to supply their own values to restrict the range of data returned. To create interactive reports, you can embed substitution variables in a command file or in a single SQL statement. Think of a variable as a container in which you temporarily store values.

Substitution Variables

- Use *i*SQL*Plus substitution variables to store values temporarily or permanently
  - Single ampersand (&)
  - Double ampersand (&&)
  - `DEFINE` command
- Pass variable values between SQL statements

8-4

Substitution Variables

In *i*SQL*Plus you can use a single ampersand (&) substitution variable to store values temporarily. `&user_variable` and `&&user_variable` indicate a substitution variable in a SQL or *i*SQL*Plus command. *i*SQL*Plus substitutes the value of the specified user variable for each substitution variable it encounters. If the user variable is undefined, *i*SQL*Plus prompts you for a value each time an "&" variable is found, and the first time an "&&" variable is found. You can predefine variables by using the `DEFINE` command. `DEFINE` creates and assigns a value to a variable.

Examples of Restricted Ranges of Data

- Report figures for the current quarter or specified date range only
- Report on data relevant only to the user requesting the report
- Display personnel in a given department

Other Interactive Effects

Interactive effects are not restricted to direct user interaction with the `WHERE` clause. The same principles can be used to achieve other goals, including:

- Dynamically altering headers and footers
- Obtaining input values from a file rather than from a person
- Passing values from one SQL statement to another

Using the & Substitution Variable

Use a variable prefixed with an ampersand to prompt the user for a value.

```
SELECT *
FROM   departments
WHERE  department_id = &DEPARTMENT;
```

user input

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
20	Marketing	201	1800

Single Ampersand Substitution Variable

When running a report, users often want to restrict the data returned dynamically. *i*SQL*Plus provides this flexibility by means of user variables. Use an ampersand to identify each variable in your SQL statement. You do not need to pre-define the value of each variable.

Notation	Description
`&user_variable`	Indicates a variable in a SQL statement, if the value of the variable does not exist, *i*SQL*Plus prompts the user for a value. *i*SQL*Plus discards a new variable once it is used.

The example in the slide creates a SQL statement to prompt the user for a department number at run time and displays all the details for that department from the `DEPARTMENTS` table.

Note: With the single ampersand, the user is prompted every time the command is executed, if the value of the variable does not exist. If the user does not enter a value for the substitution variable, *i*SQL*Plus displays an error message.

Using the SET VERIFY Command

Toggling the display of the text of a command before and after *i*SQL*Plus replaces substitution variables with values.

```
SET VERIFY ON
SELECT   employee_id, last_name, salary
FROM     employees
WHERE    employee_id = &employee_num;
```

Define Substitution Variables

"employee_num" 103

Submit for Execution | Cancel

```
old 3: WHERE employee_id = &employee_num
new 3: WHERE employee_id = 103
```

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000

Using the `SET VERIFY` Command

To confirm the changes in the SQL statement, use the *i*SQL*Plus `SET VERIFY` command. Setting `SET VERIFY` to `ON` forces *i*SQL*Plus to display the text of a command before and after it replaces substitution variables with values.

The example in the slide and the following example display the old as well as the new value of the text.

Example

The following example displays the old as well as the new value of the text:

```
SET VERIFY ON
SELECT  department_name, location_id
FROM    departments
WHERE   department_id = &dept_number;
```

Define Substitution Variables

"dept_number" 80

Submit for Execution | Cancel

Using the `SET VERIFY` Command (continued)

old 3: WHERE department_id = &dept_number
new 3: WHERE department_id = 80

DEPARTMENT_NAME	LOCATION_ID
Sales	2500

Character and Date Values with Substitution Variables

Use single quotation marks for date and character values.

```
SELECT last_name, department_id, salary*12
FROM   employees
WHERE  job_id ='&job_title';
```

LAST_NAME	DEPARTMENT_ID	SALARY*12
Kochhar	90	204000
De Haan	90	204000

8-8

Specifying Character and Date Values with Substitution Variables

In a `WHERE` clause, date and character values must be enclosed in single quotation marks. The same rule applies to the substitution variables. To avoid entering the quotation marks at run time, enclose the variable in single quotation marks within the SQL statement itself.

The slide shows a query to retrieve the employee name, department number, and annual salary of all employees based on the job title entered at the prompt by the user.

Note: You can also use functions like `UPPER`,`LOWER` and `INITCAP` with the ampersand. If you use `UPPER('&job_title')` , the user does not have to enter the job title in capitals. The following example displays the location and department number for the department name that the user enters. The `INITCAP` function enables the user to type the department name in any case:

```
SELECT location_id, department_id
FROM   departments
WHERE  department_name =INITCAP('&dept_name');
```

You might want to use `WHERE UPPER(department_name) = UPPER('&dept_name')` when you do not know the exact format in which `DEPARTMENT_NAME` is stored in the table.

Character and Date Values with Substitution Variables

Use single quotation marks for date and character values.

```
SELECT last_name, salary
FROM   employees
WHERE  hire_date= TO_DATE('&hire_date','DD-MON-YYYY');
```

LAST_NAME	SALARY
Higgins	12000
Gietz	8300

Specifying Character and Date Values with Substitution Variables (continued)

The slide shows a query to retrieve the name and salary of all employees hired on the date entered at the prompt by the user.

Note: The user must enter the hire date in `DD-MON-YYYY` format for the query to execute correctly as the `WHERE` clause expects the hire date to be entered in the `DD-MON-YYYY` format.

Specifying Column Names, Expressions, and Text at Run Time

Use substitution variables to supplement:

- A `WHERE` condition
- An `ORDER BY` clause
- A column expression
- A table name
- An entire `SELECT` statement

8-10

Specifying Column Names, Expressions, and Text at Run Time

Not only can you use the substitution variables in the `WHERE` clause of a SQL statement, you can also use them to substitute column names, expressions, or text.

Specifying Column Names at Run Time

```
SELECT last_name, &column_name
FROM   employees;
```

Define Substitution Variables

"column_name" department_id

Submit for Execution | Cancel

LAST_NAME	DEPARTMENT_ID
King	90
Kochhar	90
De Haan	90
Hunold	60
Ernst	60
Lorentz	60
Mourgos	50

...

20 rows selected.

ORACLE

Specifying Column Names, Expressions, and Text at Run Time (continued)

The example in the slide displays the employee name and any other column specified by the user at run time, from the `EMPLOYEES` table.

The following example uses a substitution variable with an `ORDER BY` clause to prompt the user for the order of the output:

```
SELECT   employee_id, department_id, manager_id
FROM     employees
WHERE    department_id = 80
ORDER BY &order_column;
```

Define Substitution Variables

"order_column" manager_id

Submit for Execution | Cancel

Specifying Column Names, Expressions, and Text at Run Time (continued)

```
old 4: ORDER BY &order_column
new 4: ORDER BY manager_id
```

EMPLOYEE_ID	DEPARTMENT_ID	MANAGER_ID
149	80	100
174	80	149
176	80	149

Observe in the output above that the results set is sorted according to the `MANAGER_ID` column.

Specifying Column Names and Expressions at Run Time

```
SELECT employee_id, job_id, &column_name
FROM   employees
WHERE  &condition;
```

Define Substitution Variables

"column_name" salary

"condition" job_id='ST_CLERK'

Submit for Execution   Cancel

EMPLOYEE_ID	JOB_ID	SALARY
141	ST_CLERK	3500
142	ST_CLERK	3100
143	ST_CLERK	2600
144	ST_CLERK	2500

ORACLE

8-13

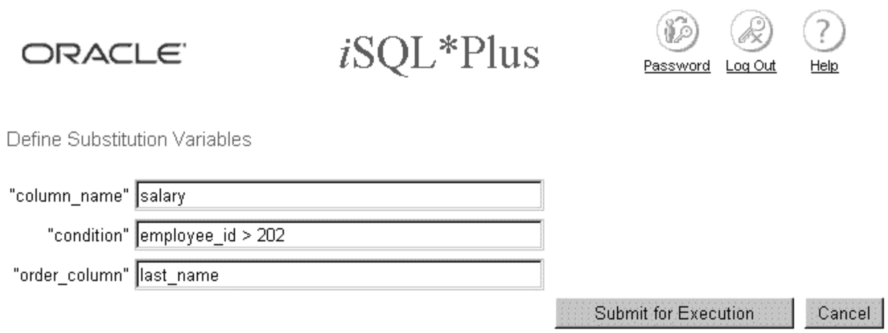
Specifying Column Names, Expressions, and Text at Run Time (continued)

The example in the slide displays the employee number, job title, and any other column specified by the user at run time, from the `EMPLOYEES` table. The user can also specify the condition for retrieval of rows and the column name by which the data is to be ordered.

Note: The condition that the user enters at the second prompt does not have to involve the column name the user enters at the first prompt.

Specifying Column Names, Expressions, and Text at Run Time

```
SELECT   employee_id, last_name,&column_name
FROM     employees
WHERE    &condition
ORDER BY &order_column;
```



Specifying Column Names, Expressions, and Text at Run Time (continued)

The example in the slide displays the employee number, name, job title, and any other column specified by the user at run time, from the EMPLOYEES table. The user can also specify the condition for retrieval of rows and the column name by which the data is to be ordered. The output of the query is as follows:

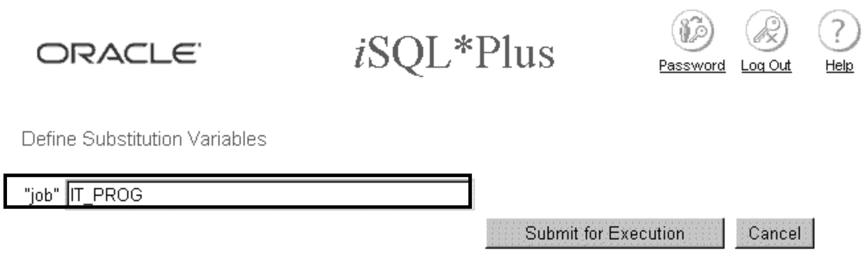
```
old 1: SELECT employee_id, last_name,&column_name
new 1: SELECT employee_id, last_name,salary
old 3: WHERE &condition
new 3: WHERE employee_id > 202
old 4: ORDER BY &order_column
new 4: ORDER BY last_name
```

EMPLOYEE_ID	LAST_NAME	SALARY
206	Gietz	8300
205	Higgins	12000

Using the && Substitution Variable

If the variable is prefixed with a double ampersand (&&), *i*SQL*Plus will prompt for the value only once.

```
SELECT employee_id,last_name,department_id
FROM employees
WHERE job_id = '&&JOB' ;
```



Double Ampersand Substitution Variable

*i*SQL*Plus stores the first value supplied and uses it again whenever the query is run, without prompting for it as shown in the following example:

```
SELECT employee_id,last_name,department_id
FROM   employees
WHERE  job_id = '&&JOB';
```

```
old 3: WHERE job_id = '&&JOB'
new 3: WHERE job_id = 'IT_PROG'
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
103	Hunold	60
104	Ernst	60
107	Lorentz	60

Double Ampersand Substitution Variable (continued)

Observe that when the command is run the second time, the user is not prompted for a value of the `JOB` variable.

This double ampersand substitution variable defines the variable on the first execution of a query. The value remains stored in the variable until the end of the *i*SQL*Plus session, or until it is `UNDEFINED`.

The `UNDEFINE` command clears the variable definition. For example, the following command undefines the variable `JOB`.

```
UNDEFINE JOB
```

Defining User Variables

- **You can use one of two *i*SQL*Plus commands to predefine variables:**
  - **`DEFINE` creates a `CHAR` data type user variable**
  - **`ACCEPT` reads user input and stores it in a variable**
- **When using the `DEFINE` command, use single quotation marks around any value that includes a space**

Defining User Variables

You can predefine user variables before executing a `SELECT` statement. *i*SQL*Plus provides the `DEFINE` command for defining and setting user variables.

Command	**Description**
`DEFINE` *`variable`* `=` *`value`*	Creates a `CHAR` data type user variable and assigns a value to it
`DEFINE` *`variable`*	Displays the variable, its value, and its data type
`DEFINE`	Displays all user variables with value and data type

Note: *i*SQL*Plus commands can continue onto multiple lines, but require a hyphen.

DEFINE and UNDEFINE Commands

- **A variable remains defined until you either:**
  - **Use the `UNDEFINE` command to clear it**

  or
  - **Exit *i*SQL*Plus**
- **You can verify your changes with the `DEFINE` command**

The `DEFINE` and `UNDEFINE` Commands

Variables are defined until you do one of the following:

- Issue the `UNDEFINE` command on a variable
- Exit *i*SQL*Plus

When you undefine variables, you can verify your changes with the `DEFINE` command. When you exit *i*SQL*Plus, variables defined during that session are lost.

Guidelines

- The `DEFINE` command creates a variable if the variable does not exist; this command automatically redefines a variable if it exists.
- When using the `DEFINE` command, use single quotation marks to enclose a string that contains an embedded space.

Using the DEFINE Command

- Create a variable to hold the department name.

```
DEFINE deptname = sales
```

SP2-0863: iSQL*Plus processing completed

- Use the variable as you would any other variable.

```
SELECT *
FROM   departments
WHERE  department_name = INITCAP('&deptname');
```

old 3: WHERE department_name = INITCAP('&deptname')
new 3: WHERE department_name = INITCAP('sales')

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
80	Sales	149	2500

Using the DEFINE Command

You can use the DEFINE command to create a variable and then use it as you would any other substitution variable. The example in the slide creates a variable DEPARTMENT_NAME that contains the department name, Sales. The SQL statement then uses this variable to display the number, name, and location of the sales department.

Use the UNDEFINE command to erase the variable:

```
UNDEFINE deptname
DEFINE   deptname
symbol   deptname is UNDEFINED
```

Variables can also be used to define expressions that would be used frequently in SELECT statements throughout the session. It would be useful to DEFINE variables and reference the variable in the SELECT statement rather than repeatedly typing out the expression. The example below illustrates the usage of variables.

```
DEFINE val = 12*salary*(1+NVL(commission_pct,0))
SELECT * FROM employees
WHERE    &val > 25000;
```

Customizing the *i*SQL*Plus Environment

- **Use the SET commands to control the current session.**

```
SET system_variable value
```

- **Verify what you have set by using the SHOW command.**

```
SET ECHO ON
```

```
SHOW ECHO
```

```
SHOW ECHO
echo ON
```


Customizing the *i*SQL*Plus Environment

You can use the `SET` commands to control the environment in which *i*SQL*Plus is currently operating.

In the syntax:

`system_variable`	Controls one aspect of the session environment
`value`	Is a value for the system variable

You can verify what you have set by using the `SHOW` command. The `SHOW` command in the slide checks whether `ECHO` had been set on or off. To see all `SET` variable values, use the `SHOW ALL` command.

SET Command Variables

- **FEEDBACK**
- **HEADING**
- **LINESIZE**
- **LONG**
- **PAGESIZE**
- **PAUSE**

SET Command Variables

SET Variable and Values	**Description**
`FEED[BACK] {`**6**`|`*n*`|OFF|`**ON**`}`	Displays the number of records returned by a query when the query selects at least *n* records.
`HEA[DING] {OFF|`**ON**`}`	Determines whether column headings are displayed in reports.
`LIN[ESIZE] {`**80**`|`*n*`}`	Sets the number of characters per line to *n* for reports.
`LONG {`**80**`|`*n*`}`	Sets the maximum width for displaying `LONG` values.
`PAGES[IZE] {`**24**`|`*n*`}`	Specifies the number of lines per page of output.
`PAU[SE] {`**OFF**`|ON|`*text*`}`	Controls scrolling of the terminal. You must press [Return] after seeing each pause.

Note: The value *n* represents a numeric value. The values shown in bold face in the table are default values. If you enter no value with the variable, *i*SQL*Plus assumes the default value.

Using SET Command Variables

SET feedback

```
SET feedback 1
SELECT last_name, department_id, hire_date
FROM   employees
WHERE  last_name='King';
```

LAST_NAME	DEPARTMENT_ID	HIRE_DATE
King	90	17-JUN-87

1 row selected.

ORACLE

Using SET Feedback

The example in the slide shows the use of the `SET` feedback command. `FEED[BACK] {6|n|ON|OFF}`displays the number of records returned by a query when a query selects at least *n* records.`ON` or `OFF` turns this display on or off. Turning feedback `ON` sets n to 1. Setting feedback to zero is equivalent to turning it `OFF`.

Using SET Command Variables

SET heading

```
SET heading off
SELECT employee_id, last_name, manager_id
FROM   employees
WHERE  last_name='Kochhar';
```

101	Kochhar	100

1 row selected.

Using SET Heading

The example in the slide turns off the headings for columns. Queries entered from the *i*SQL*Plus command line will return data for any column that appears in the `SELECT` list but do not display the heading for the column.

*i*SQL*Plus Format Commands

- **`COLUMN [column option]`**
- **`BREAK  [ON report_element]`**

Obtaining More Readable Reports

You can control the report features by using the following commands:

`COL[UMN] [column option]`: Controls column formats

`BRE[AK] [ON report_element]`: Suppresses duplicate values and sections rows of data with line feeds

Guidelines

- All format commands remain in effect until the end of the *i*SQL*Plus session or until the format setting is overwritten or cleared.
- Remember to reset your *i*SQL*Plus settings to default values after every report.
- There is no command for setting a *i*SQL*Plus variable to its default value; you must know the specific value or log out and log in again.
- If you give an alias to your column, you must use the alias name, not the column name.

The `COLUMN` Command

Controls display of a column:

```
COL[UMN] [{column|alias} [option]]
```

- **`CLE[AR]` clears any column formats.**
- **`FOR[MAT]` format uses a format model to change the display of the column.**
- **`HEA[DING]` text sets the column heading.**
- **`JUS[TIFY]` {align} aligns the column heading at left, center,or right.**

ORACLE

`COLUMN` Command Options

Option	Description
`CLE[AR]`	Clears any column formats.
`FOR[MAT]` *format*	Changes the display of the column data.
`HEA[DING]` *text*	Sets the column heading. A vertical line | forces a line feed in the heading if you do not use justification.
`JUS[TIFY] {align}`	Aligns the column heading (not the data) at the left, center or right of the column.
`NOPRI[NT]`	Hides the column.
`NUL[L]` *text*	Specifies text to be displayed for null values.
`PRI[NT]`	Shows the column.
`TRU[NCATED]`	Truncates the string at the end of the first line of display.
`WRA[PPED]`	Wraps the end of the string to the next line.

Using the COLUMN Command

- Create a column heading for the LAST_NAME column.

```
COLUMN last_name HEADING 'Employee|Name' FORMAT A15
```

- Display the current setting for the LAST_NAME column.

```
COLUMN last_name
```

- Clear settings for the LAST_NAME column.

```
COLUMN last_name CLEAR
```

ORACLE

8-26

Display or Clear Settings

To show or clear the current `COLUMN` command settings, use the following commands:

Command	Description
`COL[UMN]` *`column`*	Displays the current settings for the specified column
`COL[UMN]`	Displays the current settings for all columns
`COL[UMN] column CLE[AR]`	Clears the settings for the specified column
`CLE[AR] COL[UMN]`	Clears the settings for all columns

Note: If you have a lengthy command, you can continue it on the next line by ending the current line with a hyphen.

Using the COLUMN Command

```
COLUMN last_name HEADING 'Employee|Name' FORMAT A15
COLUMN last_name
```

COLUMN last_name HEADING 'Employee|Name' FORMAT A15
COLUMN last_name
COLUMN last_name ON
HEADING 'Employee|Name'
headsep '|'
FORMAT A15

```
SELECT last_name, salary, manager_id
FROM   employees
WHERE  last_name ='Lorentz';
```

Employee Name	SALARY	MANAGER_ID
Lorentz	4200	103

1 row selected.

Using the COLUMN Command

The example in the slide sets the heading for the DEPARTMENT_NAME column to Employee Name. The '|' character is used to wrap the heading to two lines. The width of the column is set to 15 characters with the FORMAT command.

The following example gives the MANAGER_ID column the title of Manager

```
COLUMN manager_id HEADING 'Manager'
SELECT last_name, manager_id
FROM   employees
WHERE  last_name ='Davies';
```

LAST_NAME	Manager
Davies	124

.

Using the COLUMN Command

```
COLUMN salary FORMAT $99,999.00

COLUMN manager_id FORMAT 999999999 NULL 'No manager'

SELECT last_name, salary, manager_id
FROM   employees
WHERE  last_name ='King';
```

LAST_NAME	SALARY	MANAGER_ID
King	$24,000.00	No manager

Using the COLUMN Command (continued)

In the example in the slide, the COLUMN command formats the display of the data in the column with a $ symbol, two decimal places, and the comma. The MANAGER_ID column is formatted to contain nine digits, and the text to display for null values in the column is set to No Manager.

The following command specifies the text to display for null values in the COMMISSION_PCT column. Any employee who does not earn commission (who is not a salesman) has -No Comm- displayed in the COMMISSION_PCT column.

```
COLUMN commission_pct NULL '-No Comm-'
SELECT last_name,commission_pct
FROM   employees
WHERE  department_id IN (80,20);
```

LAST_NAME	COMMISSION_PCT
Hartstein	-No Comm-
Fay	-No Comm-
Zlotkey	.2
Abel	.3
Taylor	.2

COLUMN Format Models

Element	Description	Example	Result
A*n*	Sets a display width of *n*	N/A	N/A
9	Single zero-suppression digit	99999	1234
0	Enforces leading zero	09999	01234
$	Floating dollar sign	$9999	$1234
L	Local currency	L9999	L1234
.	Position of decimal point	9999.99	1234.00
,	Thousand separator	9,999	1,234

ORACLE

8-29

COLUMN Format Models

The slide shows sample COLUMN format models.

The Oracle Server displays a string of hash signs (#) in place of a whole number whose digits exceed the number of digits provided in the format model. It also displays pound signs in place of a value whose format model is alphanumeric but whose actual value is numeric.

Using the BREAK Command

Suppresses duplicates and sections rows:

- **To suppress duplicates**

```
BREAK ON last_name ON job_id
```

- **To section rows at break values**

```
BREAK ON last_name SKIP 4 ON job_id SKIP  2
```


The BREAK Command

Use the BREAK command to place a space between rows and suppress duplicate values. The column you specify in a BREAK command is called a break column. By including the break column in your ORDER BY clause, you create meaningful subsets of records in your output.

Syntax

```
BREAK on column[|alias|row] [skip n|dup|page] on .. [on report]
```

In the syntax: `page` Goes to a new page when the break value changes

Breaks can be active on:

- Column
- Row
- Page
- Report

`duplicate` Displays duplicate values

Clear all Break settings by using the CLEAR command:

```
CLEAR BREAK
```

Using the BREAK Command

```
BREAK ON job_id
SELECT  job_id, last_name
FROM    employees
ORDER BY job_id;
```

JOB_ID	LAST_NAME
AC_ACCOUNT	Gietz
AC_MGR	Higgins
AD_ASST	Whalen
AD_PRES	King
AD_VP	Kochhar
	De Haan
IT_PROG	Hunold
	Ernst
	Lorentz
MK_MAN	Hartstein
MK_REP	Fay

...

20 rows selected.

Using the BREAK Command

The example in the slide uses the `BREAK ON` command to suppress the repetition of job titles in the output. The `ORDER BY` job clause ensures that job titles are grouped for clear output.

Creating a Script File to Run a Report

1. Create the SQL `SELECT` statement.
2. Save the `SELECT` statement to a script file with a `.sql` extension.
3. Load the script file into an editor.
4. Add formatting commands before the `SELECT` statement.
5. Verify that the termination character follows the `SELECT` statement.

8-32

Creating the Script File

You can either enter each of the *i*SQL*Plus commands at the SQL prompt or put all the commands, including the `SELECT` statement, in a command (or script) file. A typical script consists of at least one `SELECT` statement and several *i*SQL*Plus commands.

How to Create a Script File

1. Create the SQL `SELECT` statement at the SQL prompt. Make sure that the data required for the report is accurate before you save the statement to a file and apply formatting commands. If you intend to use breaks ensure that the relevant `ORDER BY` clause is included.
2. Save the `SELECT` statement to a script file with a `.sql` extension. Click the Save Script button in the *i*SQL*Plus window to save the script.
3. Edit the script file to enter the *i*SQL*Plus commands.
4. Add the required formatting commands before the `SELECT` statement. Be careful not to place *i*SQL*Plus commands in the `SELECT` statement.
5. Verify that the `SELECT` statement is followed by a run character, either a semicolon (`;`) or a slash (`/`).

Creating a Script File to Run a Report

6. Clear formatting commands after the `SELECT` statement.
7. Save the script file.
8. In *i*SQL*Plus, browse to locate the script file.
9. Load the script file.
10. Execute the script.

8-33

Creating the Script File (continued)

How to Create a Script File (continued)

6. Add the format-clearing *i*SQL*Plus commands after the run character, or call a reset file that contains all the format-clearing commands.
7. Save the script file with your changes.
8. In *i*SQL*Plus, browse to locate the script file.
9. Load the script file.
10. Execute the script.

Guidelines

- You can include blank lines between *i*SQL*Plus commands in a script.
- You can abbreviate *i*SQL*Plus commands.
- Include reset commands at the end of the file to restore the original *i*SQL*Plus environment.

Sample Report

Job Category	Employee	Salary
AD_PRES	King	$24,000.00
AD_VP	Kochhar	$17,000.00
	De Haan	$17,000.00
SA_MAN	Zlotkey	$10,500.00
SA_REP	Abel	$11,000.00
MK_MAN	Hartstein	$13,000.00
AC_MGR	Higgins	$12,000.00

ORACLE

Example

Create a script file to create a report that displays the job title, name, and salary of every employee whose salary is more than $10000. Rename the job ID column as Job Category and split it into two lines. Rename the employee last name column as Employee. Rename the salary column Salary and have the output displayed as $2,500.00. Order the output in the order of job first and then by last name.

```
SET PAGESIZE 37
SET LINESIZE 60
SET FEEDBACK OFF
COLUMN job_id HEADING ’Job|Category’ FORMAT A15
COLUMN last_name HEADING ’Employee’ FORMAT A15
COLUMN salary HEADING ’Salary’ FORMAT $99,999.99
REM ** Insert SELECT statement
SELECT   job_id, last_name, salary
FROM     employees
WHERE    salary > 10000;
```

Remember to clear the settings after the report is produced

`REM` indicates a remark or comment in *i*SQL*Plus.

`FORMAT A15` indicates alphanumeric data.

Summary

Substitution variables can be used in script files with the following:

- **Single ampersands**
- **Double ampersands**

ORACLE

8-35

Summary

Substitution variables are useful for running reports. They give you the flexibility to replace values in a `WHERE` clause, column names, and expressions. You can customize reports by writing script files with:

- Single ampersand substitution variables
- Double ampersands substitution variables

Practice 8 Overview

This practice covers the following topics:

- **Creating a query to display values using substitution variables**
- **Starting a command file containing variables**

Practice 8 Overview

In this practice you use substitution variables to create run time selection criteria to create files that can be run interactively.

Practice 8

1. a. A single ampersand substitution variable prompts only once.

   True/False

   b. The `DEFINE` command is a SQL statement.

   True/False

2. Write a statement that prompts a user for a department number at run time and then displays the employee last name, ID, and salary, for each employee in the department:

```
old 3: WHERE department_id = &department_number
new 3: WHERE department_id = 50
```

LAST_NAME	EMPLOYEE_ID	SALARY
Mourgos	124	5800
Rajs	141	3500
Davies	142	3100
Matos	143	2600
Vargas	144	2500

3. Write a script that prompts the user for two dates in the `DD-MON-YYYY` format. The script displays the employee last name, number, salary and hire date for each employee hired between these two dates. Save the script as `8Lab3.sql` using the `Save` Script button.

   Note: Enter the date in the `DD-MON-YYYY` format.

```
old 4: TO_DATE( '&low_date','DD-MON-YYYY')
new 4: TO_DATE( '01-JAN-1995','DD-MON-YYYY')
old 5: AND TO_DATE( '&high_date','DD-MON-YYYY')
new 5: AND TO_DATE( '31-DEC-1998','DD-MON-YYYY')
```

LAST_NAME	SALARY	HIREDATE
Rajs	141	17-OCT-1995
Davies	142	29-JAN-1997
Matos	143	15-MAR-1998
Vargas	144	09-JUL-1998
Abel	174	11-MAY-1996
Taylor	176	24-MAR-1998
Hartstein	201	17-FEB-1996
Fay	202	17-AUG-1997

8 rows selected.

A

Manipulating Data

Objectives

After completing this lesson, you should be able to do the following:

- **Describe each DML statement**
- **Insert rows into a table**
- **Update rows in a table**
- **Delete rows from a table**
- **Merge rows in a table**
- **Control transactions**

A-2

Lesson Aim

In this lesson, you will learn how to insert rows into a table, update existing rows in a table, and delete existing rows from a table. You will also learn how to control transactions with the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements.

Data Manipulation Language

- **A DML statement is executed when you:**
  - **Add new rows to a table**
  - **Modify existing rows in a table**
  - **Remove existing rows from a table**
- **A transaction consists of a collection of DML statements that form a logical unit of work.**

A-3

Data Manipulation Language

Data manipulation language (DML) is a core part of SQL. When you want to add, update, or delete data in the database, you execute a DML statement. A collection of DML statements that form a logical unit of work is called a transaction.

Consider a banking database. When a bank customer transfers money from a savings account to a checking account, the transaction might consist of three separate operations: decrease the savings account, increase the checking account, and record the transaction in the transaction journal. The Oracle Server must guarantee that all three SQL statements are performed to maintain the accounts in proper balance. When something prevents one of the statements in the transaction from executing, the other statements of the transaction must be undone.

The INSERT Statement Syntax

- Add new rows to a table by using the INSERT statement.

```
INSERT INTO   table [(column [, column...])]
VALUES        (value [, value...]);
```

- Only one row is inserted at a time with this syntax.

ORACLE

Adding a New Row to a Table (continued)

You can add new rows to a table by issuing the `INSERT` statement.

In the syntax:

`table`	is the name of the table
`column`	is the name of the column in the table to populate
`value`	is the corresponding value for the column

Note: This statement with the `VALUES` clause adds only one row at a time to a table.

Inserting New Rows

- **Insert a new row containing values for each column.**
- **List values in the default order of the columns in the table.**
- **Optionally, list the columns in the `INSERT` clause.**

```
INSERT INTO departments(department_id, department_name,
                        manager_id, location_id)
VALUES      (70, 'Public Relations', 100, 1700);
1 row created.
```

- **Enclose character and date values within single quotation marks.**

A-5

Adding a New Row to a Table (continued)

Because you can insert a new row that contains values for each column, the column list is not required in the `INSERT` clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table and a value must be provided for each column.

For clarity, use the column list in the `INSERT` clause.

Enclose character and date values within single quotation marks; it is not recommended to enclose numeric values within single quotation marks.

Number values should not be enclosed in single quotes, because implicit conversion may take place for numeric values assigned to `NUMBER` data type columns if single quotes are included.

Inserting Rows with Null Values

- **Implicit method: Omit the column from the column list.**

```
INSERT INTO   departments (department_id,
                           department_name )
VALUES        (30, 'Purchasing');
1 row created.
```

- **Explicit method: Specify the `NULL` keyword in the `VALUES` clause.**

```
INSERT INTO   departments
VALUES        (100, 'Finance', NULL, NULL);
1 row created.
```

ORACLE

Methods for Inserting Null Values

Method	**Description**
Implicit	Omit the column from the column list.
Explicit	Specify the `NULL` keyword in the `VALUES` list, specify the empty string (`''`) in the `VALUES` list for character strings and dates.

Be sure that the targeted column allows null values by verifying the Null? status with the *i*SQL*Plus `DESCRIBE` command.

The Oracle Server automatically enforces all data types, data ranges, and data integrity constraints. Any column that is not listed explicitly obtains a null value in the new row.

Common errors that can occur during user input:

- Mandatory value missing for a `NOT NULL` column
- Duplicate value violates uniqueness constraint
- Foreign key constraint violated
- `CHECK` constraint violated
- Data type mismatch
- Value too wide to fit in column

Inserting Special Values

The SYSDATE function records the current date and time.

```
INSERT INTO employees (employee_id,
                 first_name, last_name,
                 email, phone_number,
                 hire_date, job_id, salary,
                 commission_pct, manager_id,
                 department_id)
VALUES          (113,
                 'Louis', 'Popp',
                 'LPOPP', '515.124.4567',
                 SYSDATE, 'AC_ACCOUNT', 6900,
                 NULL, 205, 100);
1 row created.
```

A-7

Inserting Special Values by Using SQL Functions

You can use functions to enter special values in your table.

The slide example records information for employee Popp in the EMPLOYEES table. It supplies the current date and time in the HIRE_DATE column. It uses the SYSDATE function for current date and time.

You can also use the USER function when inserting rows in a table. The USER function records the current username.

Confirming Additions to the Table

```
SELECT employee_id, last_name, job_id, hire_date,
       commission_pct
FROM   employees
WHERE  employee_id = 113;
```

Inserting Specific Date Values

- **Add a new employee.**

```
INSERT INTO employees
VALUES      (114,
             'Den', 'Raphealy',
             'DRAPHEAL', '515.127.4561',
             TO_DATE('FEB 3, 1999', 'MON DD, YYYY'),
             'AC_ACCOUNT', 11000, NULL, 100, 30);
1 row created.
```

ORACLE

Inserting Specific Date and Time Values

The DD-MON-YY format is usually used to insert a date value. With this format, recall that the century defaults to the current century. Because the date also contains time information, the default time is midnight (00:00:00).

If a date must be entered in a format other than the default format, for example, with another century, or a specific time, you must use the TO_DATE function.

The example on the slide records information for employee Raphealy in the EMPLOYEES table. It sets the HIRE_DATE column to be February 3, 1999. If we used the following statement instead of the one shown on the slide, the year of the HIRE_DATE is interpreted as 2099.

```
INSERT INTO employees
  VALUES      (114,
               'Den', 'Raphealy',
               'DRAPHEAL', '515.127.4561',
               '03-FEB-99',
               'AC_ACCOUNT', 11000, NULL, 100, 30);
```

If the RR format is used, the system provides the correct century automatically, even if it is not the current one.

Creating a Script

- **The DEFINE command creates an *i*SQL*Plus variable and stores the value.**
- **& is a placeholder for the variable value.**

```
DEFINE department_id = 40
DEFINE department_name = "Human Resources"
DEFINE location = 2500

INSERT INTO    departments
            (department_id, department_name, location_id)
VALUES      (&department_id, '&department_name',
               &location);
```


Creating a Script

The DEFINE command specifies a user variable and assigns it a CHAR value, or lists the value and variable type of a single variable or all variables.

Syntax

```
DEF[INE] [variable]|[variable = text]
```

Where:

Variable	Represents the user variable whose value you wish to assign or list.
text	Represents the CHAR value you wish to assign to variable. Enclose text in single quotes if it contains punctuation or blanks.
variable = text	Defines (names) a user variable and assigns it a CHAR value.

Enter DEFINE followed by variable to list the value and type of variable. Enter DEFINE with no clauses to list the values and types of all user variables. You can save your command with substitution variables to a file and execute the commands in the file. The example on the slide records information for a department in the DEPARTMENTS table.

Do not prefix the *i*SQL*Plus substitution parameter with the ampersand (&) when referencing it in the DEFINE command. Use a dash (-) to continue an *i*SQL*Plus command on the next line.

Copying Rows from Another Table

- Write your **INSERT** statement with a subquery.

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
  SELECT employee_id, last_name, salary, commission_pct
  FROM   employees
  WHERE  job_id LIKE '%REP%';
4 rows created.
```

- Do not use the **VALUES** clause.
- Match the number of columns in the **INSERT** clause to those in the subquery.

Copying Rows from Another Table

You can use the `INSERT` statement to add rows to a table where the values are derived from existing tables. In place of the `VALUES` clause, you use a subquery.

Syntax

```
INSERT INTO table [ column (, column) ] subquery;
```

In the syntax:

`table`	is the table name
`column`	is the name of the column in the table to populate
`subquery`	is the subquery that returns rows into the table

The number of columns and their data types in the column list of the `INSERT` clause must match the number of values and their data types in the subquery. To create a copy the rows of a table, use `SELECT *` in the subquery.

```
INSERT INTO copy_emp
SELECT *
FROM employees;
```

For more information, see *Oracle SQL Reference, "SELECT," Subqueries section.*

Using a Subquery in an `INSERT` statement

```
INSERT INTO
        (SELECT employee_id, last_name,
                email, hire_date, job_id, salary,
                department_id
         FROM   employees
         WHERE  department_id = 50)
VALUES (99999, 'Taylor', 'DTAYLOR',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000, 50);

1 row created.
```

ORACLE

A-11

Using a Subquery in an `INSERT` Statement

You can use a subquery in place of the table name in the `INTO` clause of the `INSERT` statement. Use the `INSERT INTO` clause to specify the target object or objects into which Oracle is to insert data.

The select list of this subquery must have the same number of columns as the column list of the `VALUES` clause. Any rules on the columns of the base table must be followed in order for the `INSERT` statement to work successfully. For example, you could not put in a duplicate employee Id, nor leave out a value for a mandatory not null column.

Using the `WITH CHECK OPTION` Keyword on DML Statements

- **A subquery is used to identify the table and columns of the DML statement.**
- **The `WITH CHECK OPTION` keyword prohibits you from changing rows that are not in the subquery.**

```
INSERT INTO (SELECT employee_id, last_name, email,
                    hire_date, job_id, salary,department_id
              FROM   employees
              WHERE  department_id = 50 WITH CHECK OPTION)
VALUES (99998, 'Smith', 'JSMITH',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000,10);
```

```
INSERT INTO (SELECT employee_id, last_name, email,
     *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

ORACLE

A-12

The `WITH CHECK OPTION` Keyword

Specify `WITH CHECK OPTION` to indicate that, if the subquery is used in place of a table in an `INSERT`, `UPDATE`, or `DELETE` statement, no changes to that table are permitted which would produce rows that are not included in the subquery.

In the example shown, the `WITH CHECK OPTION` keyword is used. The subquery identifes rows that are in department 50. The value provided for `DEPARTMENT_ID` it in the `VALUES` list is 10, which violates the `CHECK OPTION`. The above query can be rewritten as:

The `WITH CHECK OPTION` Keyword (Continued)

```
INSERT INTO  (SELECT employee_id, last_name, email,
                     hire_date, job_id, salary,department_id
              FROM   employees
              WHERE  department_id = 50 WITH CHECK OPTION)
VALUES  (99998, 'Smith', 'JSMITH',
         TO_DATE('07-JUN-99', 'DD-MON-RR'),
         'ST_CLERK', 5000,50);
```

The following statement is legal even though the value for `DEPARTMENT_ID` violates the condition of the subquery where_clause:

```
INSERT INTO  (SELECT employee_id, last_name, email,
                     hire_date, job_id, salary,department_id
              FROM   employees
              WHERE  department_id = 50)
VALUES (99998, 'Smith', 'JSMITH',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000,10);
```

Overview of the Explict Default Feature

- The explicit default feature allows you to use the `DEFAULT` keyword where the column default value is desired.
- The addition of this feature is for compliance with the SQL: 1999 Standard.
- This allows the user to control where and when the default value should be applied to data.
- Explicit defaults can be used in `INSERT` and `UPDATE` statements.

ORACLE

A-14

Explicit Defaults

The `DEFAULT` keyword can be used in `INSERT` and `UPDATE` statements to identify a default column value. If no default value exists, a null value is used.

Using Explicit Default Values

- **DEFAULT with INSERT:**

```
INSERT INTO departments
  (department_id, department_name, manager_id)
VALUES (300, 'Engineering', DEFAULT);
```


Using Explicit Default Values

Specify DEFAULT to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, Oracle sets the column to null.

In the example shown, the INSERT statement uses a default value for the MANAGER_ID column. If there is no default value defined for the column, a null value is inserted instead.

The UPDATE Statement Syntax

- **Modify existing rows with the UPDATE statement.**

```
UPDATE       table
SET          column = value [, column = value, ...]
[WHERE       condition];
```

- **Update more than one row at a time, if required.**

A-16

Updating Rows

You can modify existing rows by using the `UPDATE` statement.

In the syntax:

`table`	is the name of the table
`column`	is the name of the column in the table to populate
`value`	is the corresponding value or subquery for the column
`condition`	identifies the rows to be updated and is composed of column names expressions, constants, subqueries, and comparison operators

Confirm the update operation by querying the table to display the updated rows.

For more information, see *Oracle SQL Reference, "UPDATE."*

Note: In general, use the primary key to identify a single row. Using other columns may unexpectedly cause several rows to be updated. For example, identifying a single row in the `EMPLOYEES` table by name is dangerous because more than one employee may have the same name.

Updating Rows in a Table

- Specific row or rows are modified if you specify the `WHERE` clause.

```
UPDATE employees
SET    department_id = 70
WHERE  employee_id = 113;
1 row updated.
```

- All rows in the table are modified if you omit the `WHERE` clause.

```
UPDATE   copy_emp
SET      department_id = 110;
23 rows updated.
```

ORACLE

A-17

Updating Rows (continued)

The `UPDATE` statement modifies specific rows, if the `WHERE` clause is specified. The slide example transfers employee 113 (Popp) to department 70.

If you omit the `WHERE` clause, all the rows in the table are modified.

Note: The `COPY_EMP` table has the same data as the `EMPLOYEES` table.

Updating Two Columns with a Subquery

Update employee 114's job and department to match that of employee 205.

```
UPDATE employees
SET    job_id  =   (SELECT  job_id
                    FROM    employees
                    WHERE   employee_id = 205),
       salary  =   (SELECT  salary
                    FROM    employees
                    WHERE   employee_id = 205)
WHERE  employee_id    =  114;
1 row updated.
```


Updating Two Columns with a Subquery

You can update multiple columns in the SET clause of an UPDATE statement by writing multiple subqueries.

Syntax

```
UPDATE table
  SET     column  =
                       (SELECT  column
                        FROM table
                        WHERE condition)
                        [ ,column  =
                       (SELECT  column
                        FROM table
                        WHERE condition)]
  [WHERE  condition ] ;
```

Note: If no rows are updated, a message "0 rows updated." is returned:

Updating Rows Based on Another Table

Use subqueries in UPDATE statements to update rows in a table based on values from another table.

```
UPDATE  copy_emp
SET     department_id  =  (SELECT department_id
                           FROM employees
                           WHERE employee_id = 100)
WHERE   job_id         =  (SELECT job_id
                           FROM employees
                           WHERE employee_id = 200);
1 row updated.
```


Updating Rows Based on Another Table

You can use subqueries in UPDATE statements to update rows in a table. The example on the slide updates the COPY_EMP table based on the values from the EMPLOYEES table. It changes the department ID of all the employees who have the same JOB_ID as the employee with the EMPLOYEE_ID 200. It updates their department ID to the same department ID as the employee with the EMPLOYEE_ID 100.

Updating Rows: Integrity Constraint Error

```
UPDATE employees
SET    department_id = 55
WHERE  department_id = 110;
```

Department number 55 does not exist

```
UPDATE employees
*
ERROR at line 1:
ORA-02291: integrity constraint (SQL_EU.EMP_DEPT_FK) violated - parent key not found
```

ORACLE

Integrity Constraint Error

If you attempt to update a record with a value that is tied to an integrity constraint, an error is returned.

In the example on the slide, department number 55 does not exist in the parent table, `DEPARTMENTS`, and so you receive the parent key violation `ORA-02291.`

Note: Integrity constraints ensure that the data adheres to a predefined set of rules. A subsequent lesson covers integrity constraints in greater depth.

The DELETE Statement

You can remove existing rows from a table by using the DELETE statement.

```
DELETE [FROM]   table
[WHERE      condition];
```


Deleting Rows

You can remove existing rows by using the DELETE statement.

In the syntax:

table	is the table name.
condition	identifies the rows to be deleted and is composed of column names, expressions, constants, subqueries, and comparison operators.

Note: If no rows are deleted, a message "0 rows deleted." is returned:

For more information, *see Oracle SQL Reference, "DELETE."*

Deleting Rows from a Table

- **Specific rows are deleted if you specify the `WHERE` clause.**

```
 DELETE FROM departments
 WHERE  department_name = 'Finance';
1 row deleted.
```

- **All rows in the table are deleted if you omit the `WHERE` clause.**

```
DELETE FROM  copy_emp;
23 rows deleted.
```

ORACLE

Deleting Rows (continued)

You can delete specific rows by specifying the `WHERE` clause in the `DELETE` statement. The slide example deletes the Finance department from the `DEPARTMENTS` table. You can confirm the delete operation by displaying the deleted rows using the `SELECT` statement.

```
SELECT   *
 FROM    departments
 WHERE   department_name = 'Finance';
 no rows selected.
```

If you omit the `WHERE` clause, all rows in the table are deleted. The second example on the slide deletes all the rows from the `COPY_EMP` table because no `WHERE` clause has been specified.

Example

Remove rows identified in the `WHERE` clause.

```
DELETE FROM  employees
 WHERE        employee_id = 114;
 1 row deleted.
 DELETE FROM  departments
 WHERE        department_id IN (30, 40);
 2 rows deleted.
```

Deleting Rows Based on Another Table

Use subqueries in `DELETE` statements to remove rows from a table based on values from another table.

```
DELETE FROM employees
WHERE  department_id =
                       (SELECT department_id
                        FROM   departments
                        WHERE  department_name LIKE '%Public%');
1 row deleted.
```


Deleting Rows Based on Another Table

You can use subqueries to delete rows from a table based on values from another table. The example on the slide deletes all the employees who are in a department where the department name contains the string "Public." The subquery searches the `DEPARTMENTS` table to find the department number based on the department name containing the string "Public." The subquery then feeds the department number to the main query, which deletes rows of data from the `EMPLOYEES` table based on this department number.

Deleting Rows: Integrity Constraint Error

```
DELETE FROM departments
WHERE       department_id = 60;
```

You cannot delete a row that contains a primary key that is used as a foreign key in another table.

```
DELETE FROM departments
*
ERROR at line 1:
ORA-02292: integrity constraint (SQL_EU.EMP_DEPT_FK) violated - child record found
```

A-24

Integrity Constraint Error

If you attempt to delete a record with a value that is tied to an integrity constraint, an error is returned.

The example on the slide tries to delete department number 60 from the `DEPARTMENTS` table, but it results in an error because department number is used as a foreign key in the `EMPLOYEES` table. If the parent record that you attempt to delete has child records, then you receive the child record found violation `ORA-02292.`

The following statement works because there are no employees in department 70:

```
DELETE FROM  departments
WHERE        department_id = 70;
1 row deleted.
```

.

The MERGE Statement

- Provides the ability to conditionally update or insert data into a database table
- It performs an UPDATE if the row exists and an INSERT if it is a new row.
  - Avoids multiple updates
  - Increases performance and ease of use
  - Is useful in data warehousing applications

ORACLE

A-25

MERGE Statements

SQL has been extended to include the MERGE statement. This statement allows you to update or insert a row conditionally into a table, thus avoiding multiple UPDATE statements. The decision whether to update or insert into the target table is based on a condition in the ON clause.

Since the MERGE command combines the INSERT and UPDATE commands, you need both INSERT and UPDATE privileges on the target table and the SELECT privilege on the source table.

The MERGE statement is deterministic. You cannot update the same row of the target table multiple times in the same MERGE statement.

An alternative approach is to use PL/SQL loops and multiple DML statements. The MERGE statement, however, is easy to use, and more simply expressed as a single SQL statement.

The MERGE statement is suitable in a number of data warehousing applications. For example, in a data warehousing application, you may need to work with data coming from multiple sources, some of which may be duplicates. The MERGE statement allows you to conditionally add or modify rows.

The MERGE Statement Syntax

You can conditionally insert or update rows in a table by using the MERGE statement.

```
MERGE INTO table_name table_alias
  USING (table|view|sub_query) AS alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
    col1 = col_val1,
    col2 = col2_val
  WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```

A-26

Merging Rows

You can update existing rows and insert new rows conditionally by using the MERGE statement.

In the syntax:

INTO clause	specifies the target table you are updating or inserting into
USING clause	identifies the source of the data to be updated or inserted. This can be a table, view or subquery
ON clause	The condition upon which the MERGE operation either updates or inserts
WHEN MATCHED | WHEN NOT MATCHED	Instructs the server how to respond to the results of the join condition

For more information, see *Oracle SQL Reference, "MERGE."*

Merging Rows

Insert or update rows in the COPY_EMP table to match the EMPLOYEES table

```
MERGE INTO copy_emp c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET
     c.first_name     = e.first_name,
     c.last_name      = e.last_name,
     ...
     c.department_id  = e.department_id
WHEN NOT MATCHED THEN
 INSERT VALUES(e.employee_id, e.first_name, e.last_name,
          e.email, e.phone_number, e.hire_date, e.job_id,
          e.salary, e.commission_pct, e.manager_id,
          e.department_id);
```

ORACLE

A-27

Example of Merging Rows

The example shown matches the EMPLOYEE_ID in the COPY_EMP table to the EMPLOYEE_ID in the EMPLOYEES table. If a match is found, the row in the COPY_EMP table is updated to match the row in the EMPLOYEES table. If the row is not found, it is inserted into the COPY_EMP table. The complete code for the example in the slide is given in the next page.

Example of Merging Rows (Continued)

```
MERGE INTO copy_emp c
    USING employees e
    ON (c.employee_id = e.employee_id)
  WHEN MATCHED THEN
    UPDATE SET
      c.first_name     = e.first_name,
      c.last_name      = e.last_name,
      c.email          = e.email,
      c.phone_number   = e.phone_number,
      c.hire_date      = e.hire_date,
      c.job_id         = e.job_id,
      c.salary         = e.salary,
      c.commission_pct = e.commission_pct,
      c.manager_id     = e.manager_id,
      c.department_id  = e.department_id
  WHEN NOT MATCHED THEN
    INSERT VALUES(e.employee_id, e.first_name, e.last_name,
         e.email, e.phone_number, e.hire_date, e.job_id,
         e.salary, e.commission_pct, e.manager_id,
         e.department_id);
```

Merging Rows

```
SELECT *
FROM COPY_EMP;

no rows selected
```

```
MERGE INTO copy_emp c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET
     ...
WHEN NOT MATCHED THEN
 INSERT VALUES...;
```

```
SELECT *
FROM COPY_EMP;

20 rows selected.
```


Example of Merging Rows

The condition C.EMPLOYEE_ID = E.EMPLOYEE_ID is evaluated. Because the COPY_EMP table is empty, the condition returns false: there are no matches. The logic falls into the WHEN NOT MATCHED clause and the MERGE command inserts the rows of the EMPLOYEES table into the COPY_EMP table.

If rows existed in the COPY_EMP table and employee IDs matched in both tables (the COPY_EMP and EMPLOYEES tables), the existing rows in the COPY_EMP table would be updated to match the EMPLOYEES table.

Database Transactions

A database transaction consists of one of the following:

- **DML statements which constitute one consistent change to the data**
- **One DDL statement**
- **One DCL statement**

A-30

Database Transactions

The Oracle Server ensures data consistency based on transactions. Transactions give you more flexibility and control when changing data, and they ensure data consistency in the event of user process failure or system failure.

Transactions consist of DML statements that make up one consistent change to the data. For example, a transfer of funds between two accounts should include the debit to one account and the credit to another account in the same amount. Both actions should either fail or succeed together; the credit should not be committed without the debit.

An implicit transaction is started when a DDL or DCL statement is issued. A DDL statement or a DCL statement is automatically committed and therefore implicitly ends a transaction.

Statements in a Transaction

Type	Description
Data manipulation language (DML)	Consists of any number of DML statements that the Oracle Server treats as a single entity or a logical unit of work
Data definition language (DDL)	Consists of only one DDL statement
Data control language (DCL)	Consists of only one DCL statement

Database Transactions

- **Begin when the first DML SQL statement is executed**
- **End with one of the following events:**
  - **A `COMMIT` or `ROLLBACK` statement is issued**
  - **A DDL or DCL statement executes (automatic commit)**
  - **The user exits *i*SQL*Plus**
  - **The system crashes**

When Does a Transaction Start and End?

A transaction begins when the first DML statement is encountered, and ends when one of the following occurs:

- A `COMMIT` or `ROLLBACK` statement is issued
- A DDL statement, such as `CREATE`, is issued
- A DCL statement is issued
- The user exits *i*SQL*Plus
- A machine fails or the system crashes

After one transaction ends, the next executable SQL statement automatically starts the next transaction.

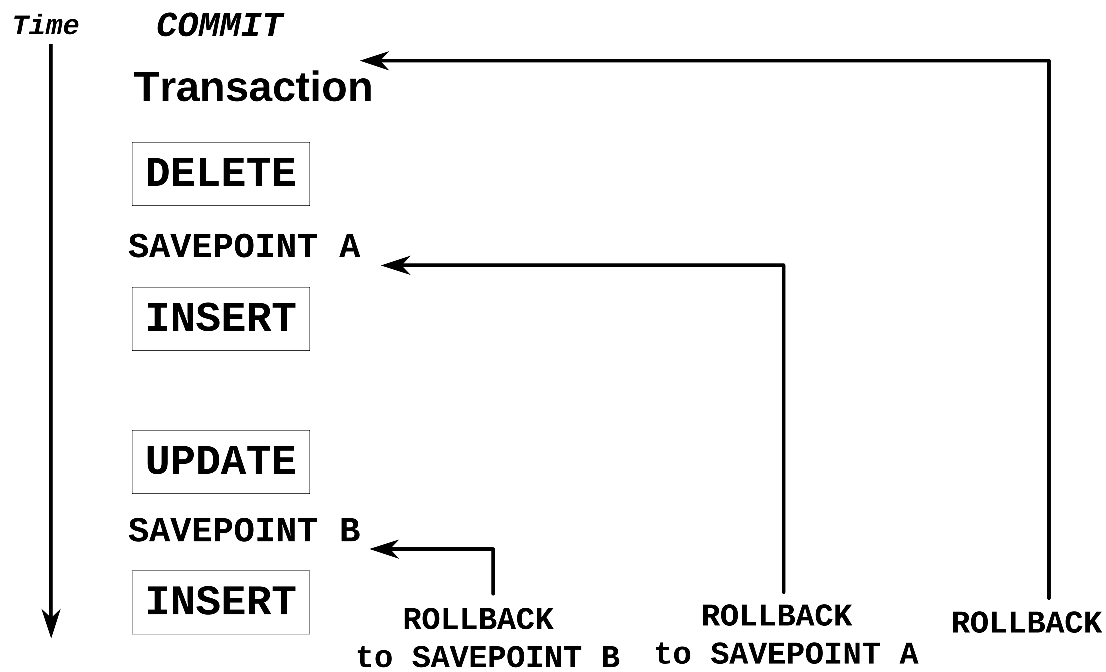
A DDL statement or a DCL statement is automatically committed and therefore implicitly ends a transaction.

Advantages of COMMIT and ROLLBACK Statements

COMMIT and ROLLBACK statements enable you to:

- **Ensure data consistency**
- **Preview data changes before making changes permanent**
- **Group logically related operations**

Controlling Transactions



A-33

Explicit Transaction Control Statements

You can control the logic of transactions by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements.

Statement	Description
`COMMIT`	Ends the current transaction by making all pending data changes permanent
`SAVEPOINT` *`name`*	Marks a savepoint within the current transaction
`ROLLBACK`	`ROLLBACK` ends the current transaction by discarding all pending data changes
`ROLLBACK TO` *`SAVEPOINT name`*	`ROLLBACK TO SAVEPOINT` rolls back the current transaction to the specified savepoint, thereby discarding any changes and or savepoints created after the savepoint to which you are rolling back. If you omit the `TO SAVEPOINT` clause, the `ROLLBACK` statement rolls back the entire transaction. As savepoints are logical, there is no way to list the savepoints you have created.

Rolling Back Changes to a Marker

- Create a marker in a current transaction by using the `SAVEPOINT` statement.
- Roll back to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

```
UPDATE...
SAVEPOINT update_done;
Savepoint created.
INSERT...
ROLLBACK TO update_done;
Rollback complete.
```


Rolling Back Changes to a Savepoint

You can create a marker in the current transaction by using the `SAVEPOINT` statement which divides the transaction into smaller sections. You can then discard pending changes up to that marker by using the `ROLLBACK TO SAVEPOINT` statement.

If you create a second savepoint with the same name as an earlier savepoint, the earlier savepoint is deleted.

Note: `SAVEPOINT` is not ANSI standard SQL.

Implicit Transaction Processing

- An automatic commit occurs under the following circumstances:
  - DDL statement is issued
  - DCL statement is issued
  - Normal exit from *i*SQL*Plus, without explicitly issuing `COMMIT` or `ROLLBACK` statements
- An automatic rollback occurs under an abnormal termination of *i*SQL*Plus or a system failure.

ORACLE

A-35

Implicit Transaction Processing

Status	Circumstances
Automatic commit	DDL statement or DCL statement is issued. *i*SQL*Plus exited normally, without explicitly issuing `COMMIT` or `ROLLBACK` commands.
Automatic rollback	Abnormal termination of *i*SQL*Plus or system failure.

Note: A third command is available in *i*SQL*Plus. The `AUTOCOMMIT` command can be toggled on or off. If set to on, each individual DML statement is committed as soon as it is executed. You cannot roll back the changes. If set to off, the `COMMIT` statement can still be issued explicitly. Also, the `COMMIT` statement is issued when a DDL statement is issued or when you exit from *i*SQL*Plus.

System Failures

When a transaction is interrupted by a system failure, the entire transaction is automatically rolled back. This prevents the error from causing unwanted changes to the data and returns the tables to their state at the time of the last commit. In this way, the Oracle Server protects the integrity of the tables.

From *i*SQL*Plus, a normal exit from the session is accomplished by clicking on the Exit button. With *i*SQL*Plus, a normal exit is accomplished by typing the command Exit at the prompt. Closing the window is interpreted as an abnormal exit.

State of the Data before `COMMIT` or `ROLLBACK`

- **The previous state of the data can be recovered.**
- **The current user can review the results of the DML operations by using the `SELECT` statement.**
- **Other users *cannot* view the results of the DML statements by the current user.**
- **The affected rows are *locked*; other users cannot change the data within the affected rows.**

Committing Changes

Every data change made during the transaction is temporary until the transaction is committed.

State of the data before `COMMIT` or `ROLLBACK` statements are issued:

- Data manipulation operations primarily affect the database buffer; therefore, the previous state of the data can be recovered.
- The current user can review the results of the data manipulation operations by querying the tables.
- Other users cannot view the results of the data manipulation operations made by the current user. The Oracle Server institutes read consistency to ensure that each user sees data as it existed at the last commit.
- The affected rows are locked; other users cannot change the data in the affected rows.

State of the Data after `COMMIT`

- Data changes are made permanent in the database.
- The previous state of the data is permanently lost.
- All users can view the results.
- Locks on the affected rows are released; those rows are available for other users to manipulate.
- All savepoints are erased.

A-37

Committing Changes (continued)

Make all pending changes permanent by using the `COMMIT` statement. Following a `COMMIT` statement:

- Data changes are written to the database.
- The previous state of the data is permanently lost.
- All users can view the results of the transaction.
- The locks on the affected rows are released; the rows are now available for other users to perform new data changes.
- All savepoints are erased.

Committing Data

- **Make the changes.**

```
DELETE FROM employees
WHERE  employee_id = 99999;
1 row deleted.

INSERT INTO departments
VALUES (290, 'Corporate Tax', NULL, 1700);
1 row inserted.
```

- **Commit the changes.**

```
COMMIT;
Commit complete.
```

ORACLE

A-38

Committing Changes (continued)

The slide example deletes a row from the EMPLOYEES table and inserts a new row into the DEPARTMENTS table. It then makes the change permanent by issuing the COMMIT statement.

Example

Remove departments 290 and 300 in the DEPARTMENTS table, update a row in the COPY_EMP table. Make the data change permanent.

```
DELETE FROM departments
WHERE  department_id IN (290, 300);
2 rows deleted.


UPDATE copy_emp
SET department_id = 80
WHERE employee_id = 206;
1 row updated.
COMMIT;
Commit Complete.
```

State of the Data after ROLLBACK

- Discard all pending changes by using the ROLLBACK statement.
- Data changes are undone.
- Previous state of the data is restored.
- Locks on the affected rows are released.

```
DELETE FROM  copy_emp;
22 rows deleted.
ROLLBACK;
Rollback complete.
```

ORACLE

A-39

Rolling Back Changes

Discard all pending changes by using the ROLLBACK statement. Following a ROLLBACK statement:

- Data changes are undone.
- The previous state of the data is restored.
- The locks on the affected rows are released.

Example

While attempting to remove a record from the TEST table, you can accidentally empty the table. You can correct the mistake, reissue the proper statement, and make the data change permanent.

Rolling Back Changes (Continued)

```
DELETE FROM  test;
   25,000 rows deleted.

   ROLLBACK;
   Rollback complete.

   DELETE FROM  test
   WHERE        id = 100;
   1 row deleted.

   SELECT   *
   FROM     test
   WHERE    id = 100;
   No rows selected.

   COMMIT;
   Commit complete.
```

.

Statement-Level Rollback

- **If a single DML statement fails during execution, only that statement is rolled back.**
- **The Oracle Server implements an implicit savepoint.**
- **All other changes are retained.**
- **The user should terminate transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.**

A-41

Statement-Level Rollbacks

Part of a transaction can be discarded by an implicit rollback if a statement execution error is detected. If a single DML statement fails during execution of a transaction, its effect is undone by a statement-level rollback, but the changes made by the previous DML statements in the transaction are not discarded. They can be committed or rolled back explicitly by the user.

Oracle issues an implicit commit before and after any data definition language (DDL) statement. So, even if your DDL statement does not execute successfully, you cannot roll back the previous statement because the server issued a commit.

Terminate your transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.

Read Consistency

- **Read consistency guarantees a consistent view of the data at all times.**
- **Changes made by one user do not conflict with changes made by another user.**
- **Read consistency ensures that on the same data:**
  - **Readers do not wait for writers**
  - **Writers do not wait for readers**

Read Consistency

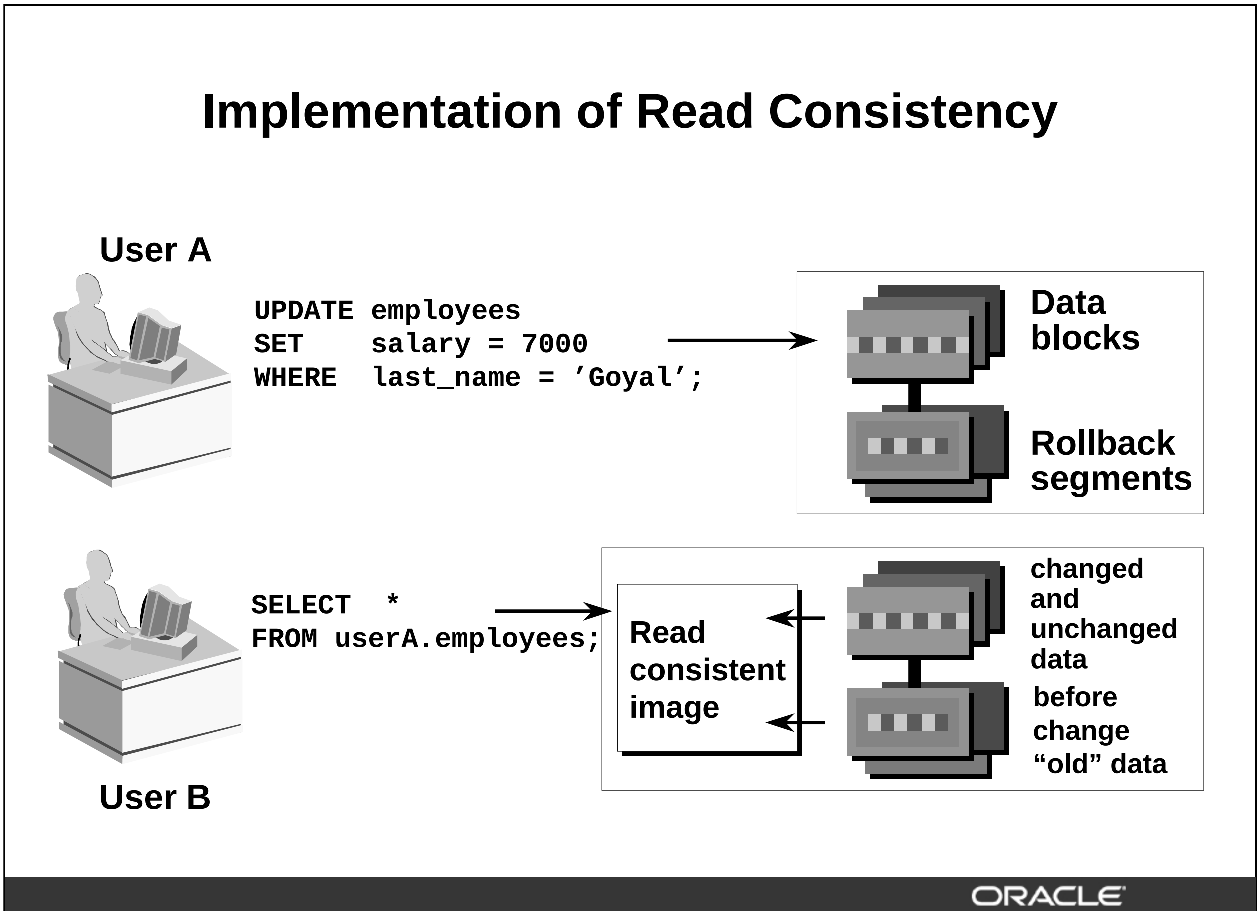
Database users access the database in two ways:

- Read operations (`SELECT` statement)
- Write operations (`INSERT`, `UPDATE`, `DELETE` statements)

You need read consistency so that the following occur:

- The database reader and writer are ensured a consistent view of the data.
- Readers do not view data that is in the process of being changed.
- Writers are ensured that the changes to the database are done in a consistent way.
- Changes made by one writer do not disrupt or conflict with changes another writer is making.

The purpose of read consistency is to ensure that each user sees data as it existed at the last commit, before a DML operation started.



A-43

Implementation of Read Consistency

Read consistency is an automatic implementation. It keeps a partial copy of the database in rollback segments.

When an insert, update, or delete operation is made to the database, the Oracle Server takes a copy of the data before it is changed and writes it to a rollback segment.

All readers, except the one who issued the change, still see the database as it existed before the changes started; they view the rollback segment's "snapshot" of the data.

Before changes are committed to the database, only the user who is modifying the data sees the database with the alterations; everyone else sees the snapshot in the rollback segment. This guarantees that readers of the data read consistent data that is not currently undergoing change.

When a DML statement is committed, the change made to the database becomes visible to anyone executing a `SELECT` statement. The space occupied by the "old" data in the rollback segment file is freed for reuse.

If the transaction is rolled back, the changes are undone:

- The original, older version, of the data in the rollback segment is written back to the table.
- All users see the database as it existed before the transaction began.

Locking

In an Oracle database, locks:

- Prevent destructive interaction between concurrent transactions
- Require no user action
- Automatically use the lowest level of restrictiveness
- Are held for the duration of the transaction
- Oracle database locks are of two types:
  - Explicit locking
  - Implicit locking

A-44

What Are Locks?

Locks are mechanisms that prevent destructive interaction between transactions accessing the same resource, either a user object (such as tables or rows) or a system object not visible to users (such as shared data structures and data dictionary rows).

How the Oracle Database Locks Data

Oracle locking is performed automatically and requires no user action. Implicit locking occurs for SQL statements as necessary, depending on the action requested. Implicit locking occurs for all SQL statements except `SELECT`.

The users can also lock data manually, which is called explicit locking.

Implicit Locking

- **Two lock modes**
  - **Exclusive: Locks out other users**
  - **Share: Allows other users to access**
- **High level of data concurrency**
  - **DML: Table share, row exclusive**
  - **Queries: No locks required**
  - **DDL: Protects object definitions**
- **Locks held until commit or rollback**

A-45

DML Locking

When performing data manipulation language (DML) operations, the Oracle Server provides data concurrency through DML locking. DML locks occur at two levels:

- A share lock is automatically obtained at the table level during DML operations. Share lock mode allows several transactions to acquire share locks on the same resource.
- An exclusive lock is acquired automatically for each row modified by a DML statement. Exclusive locks prevent the row from being changed by other transactions until the transaction is committed or rolled back. This lock ensures that no other user can modify the same row at the same time and overwrite changes not yet committed by another user.
- DDL locks occur when modifying a database object such as a table.

Summary

In this lesson, you should have learned how to use DML statements and control transactions.

Statement	Description
`INSERT`	Adds a new row to the table
`UPDATE`	Modifies existing rows in the table
`DELETE`	Removes existing rows from the table
`MERGE`	Conditionally inserts or updates data in a table
`COMMIT`	Makes all pending changes permanent
`SAVEPOINT`	Allows a rollback to the savepoint marker
`ROLLBACK`	Discards all pending data changes

ORACLE

A-46

Summary

In this lesson, you should have learned how to manipulate data in the Oracle database by using the `INSERT`, `UPDATE`, and `DELETE` statements. Control data changes by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements.

The Oracle Server guarantees a consistent view of data at all times.

Locking can be implicit or explicit.

Practice A Overview

This practice covers the following topics:

- **Inserting rows into the tables**
- **Updating and deleting rows in the table**
- **Controlling transactions**

A-47

Practice A Overview

In this practice, you will add rows to the MY_EMPLOYEE table, update and delete data from the table, and control your transactions.

Practice A

Insert data into the `MY_EMPLOYEE` table.

1. Run the statement in the `labA_1.sql` script to build the `MY_EMPLOYEE` table that will be used for the lab.
2. Describe the structure of the `MY_EMPLOYEE` table to identify the column names.

Name	Null?	Type
ID	NOT NULL	NUMBER(4)
LAST_NAME		VARCHAR2(25)
FIRST_NAME		VARCHAR2(25)
USERID		VARCHAR2(8)
SALARY		NUMBER(9,2)

3. Add the first row of data to the `MY_EMPLOYEE` table from the following sample data. Do not list the columns in the `INSERT` clause.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860
3	Biri	Ben	bbiri	1100
4	Newman	Chad	cnewman	750
5	Ropeburn	Audrey	aropebur	1550

4. Populate the `MY_EMPLOYEE` table with the second row of sample data from the preceding list. This time, list the columns explicitly in the `INSERT` clause.
5. Confirm your addition to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860

Practice A (continued)

6. Write an insert statement in a text file named `loademp.sql` to load rows into the `MY_EMPLOYEE` table. Concatenate the first letter of the first name and the first seven characters of the last name to produce the userid.
7. Populate the table with the next three rows of sample data by running the insert statement in the script that you created.
8. Confirm your additions to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860
3	Biri	Ben	bbiri	1100
4	Newman	Chad	cnewman	750
5	Ropeburn	Audrey	aropebur	1550

9. Make the data additions permanent.

Update and delete data in the `MY_EMPLOYEE` table.

10. Change the last name of employee 3 to Drexler.
11. Change the salary to 1000 for all employees with a salary less than 900.
12. Verify your changes to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
2	Dancs	Betty	bdancs	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	cnewman	1000
5	Ropeburn	Audrey	aropebur	1550

13. Delete Betty Dancs from the `MY_EMPLOYEE` table.
14. Confirm your changes to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	cnewman	1000
5	Ropeburn	Audrey	aropebur	1550

Practice A (continued)

15. Commit all pending changes

Control data transaction to the `MY_EMPLOYEE` table.

16. Populate the table with the to add Betty Dancs data by using the script that you created in step 6.
17. Confirm your addition to the table.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	cnewman	1000
5	Ropeburn	Audrey	aropebur	1550
2	Betty	Dancs	dbetty	860

18. Mark an intermediate point in the processing of the transaction.
19. Empty the entire table.
20. Confirm that the table is empty.
21. Discard the most recent `DELETE` operation without discarding the earlier `INSERT` operation.
22. Confirm that the the most recent `DELETE` has been discarded.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	1000
3	Drexler	Ben	bbiri	1100
4	Newman	Chad	cnewman	1000
5	Ropeburn	Audrey	aropebur	1550
2	Betty	Dancs	dbetty	860

Reporting with SQL*Plus

SQL*Plus SET Command Variables

- The SET command affects the way that SQL*Plus runs commands
- SET commands can be used to control:
  - Number of blank lines between records
  - Number of spaces between columns
  - Characters used to underline column headings
  - Value to display for null values

ORACLE

SQL*Plus SET Command Variables

SQL*Plus enables you to execute SQL commands and PL/SQL blocks, and to perform many additional tasks as well. Through SET commands, you can control the format in which the output of a query is displayed to the user.

Additional SET Command Variables

- **RECSEP**
- **RECSEPCHAR**
- **SPACE**
- **UNDERLINE**
- **WRAP**
- **NULL**
- **HEADSEP**
- **NEWPAGE**

B-3

SET Command Values

SET Value	Description
`RECSEP {WR[APPED]| EA[CH]\OFF}`	Controls the printing of record separators. `WRAPPED` prints a record separator only after wrapped lines; `EACH` prints a record separator following each row.
`RECSEPCHAR{_|c}`	Character printed between records
`SPA[CE] {1|n}`	Sets the number of spaces between columns
`UND[ERLINE] {-|C|ON|OFF}`	Sets the characters to use to underline column headings
`WRA[P] {OFF|ON}`	Controls the truncation of data item display
`NULL text`	Sets the text that represents a null value in the result of a SQL `SELECT` statement
`HEADSEP`	Specifies the character to be used between column headings
`NEWP[AGE] {1|n}`	Sets the number of blank lines before the top of each page (0=formfeed)

Using SET Command Variables

SET RECSEP and SET RECSEPCHAR _

```
SET RECSEP EACH
SET RECSEPCHAR
SELECT employee_id, last_name
FROM   employees
WHERE  employee_id < 103;
```

```
EMPLOYEE_ID LAST_NAME
----------- ------------------------
        100 King
________________________________________
        101 Kochhar
________________________________________
        102 De Haan
________________________________________
```

B-4

Using `SET RECSEP EACH` and `SET RECSEPCHAR _`

The example in the slide uses two SET commands to separate each row returned by the `SELECT` statement. The `SET RECSEP EACH` command prints one blank line between each record in the output. The `SET RECSEPCHAR_` command changes the record separator character from a space to an underline.

Using SET Command Variables

SET SPACE and SET UNDERLINE

```
SET SPACE 2
SET UNDERLINE =
SELECT employee_id, last_name, salary
FROM   employees
WHERE employee_id < 103;
```

```
EMPLOYEE_ID  LAST_NAME                     SALARY
===========  =========================  ==========
        100  King                            24000
        101  Kochhar                         17000
        102  De Haan                         17000
```

ORACLE

Using SET SPACE and SET UNDERLINE

The example in the slide uses two SET commands to alter the spacing between the columns returned by the SELECT statement. The SET SPACE 2 command places two spaces between columns. The SET UNDERLINE = command sets the underline character beneath column headings to an equal sign.

Using SET Command Variables

SET NULL

```
SET NULL Null
SELECT employee_id, last_name, commission_pct
FROM   employees
WHERE  department_id IN (20,80);
```

```
                                           COMMISSION_PCT
EMPLOYEE_ID LAST_NAME
----------- ---------------------- --------------
        149 Zlotkey                             .2
        174 Abel                                .3
        176 Taylor                              .2
        201 Hartstein                      Null
        202 Fay                            Null
```

B-6

Using the Additional SET Command Variables

The example in the slide uses the `SET NULL` command to display all null values as Null.

Using SET Command Variables

SET NEWPAGE

```
SET NEWPAGE 3
SELECT employee_id, last_name, manager_id
FROM   employees
WHERE  employee_id < 103;
```

```



EMPLOYEE_ID LAST_NAME                 MANAGER_ID
----------- ------------------------- ----------
        100 King
        101 Kochhar                          100
        102 De Haan                          100
```


Using SET NEWPAGE

The example in the slide uses the `SET NEWPAGE 3` command to set the number of blank lines that are displayed before each page in the output to 3.

The TTITLE and BTITLE Commands

Display headers and footers:

```
TTI[TLE] [text|OFF|ON]
```

Set the report header:

```
TTITLE 'Salary|Report'
```

Set the report footer:

```
BTITLE 'Confidential'
```

ORACLE

B-8

Using the TTITLE and BTITLE Commands

Use the `TTITLE` command to format page headers and the `BTITLE` command for footers. Footers appear at the bottom of the page according to the PAGESIZE value.

The syntax for `BTITLE` and `TTITLE` is identical. Only the syntax for `TTITLE` is shown. You can use the vertical bar (`|`) to split the text of the title across several lines.

In the syntax:

`text`	Represents the title text; enter single quotes if the text is more than one word.

The `TTITLE` example in the slide sets the report header to display Salary centered on one line and Report centered below it. The `BTITLE` example sets the report footer to display Confidential.

Note: The slide gives an abridged syntax for `TTITLE` and `BTITLE`. Various options for `TTITLE` and `BTITLE` are covered in other Oracle SQL courses

Use `TTITLE OFF` and `BTITLE` OFF to clear header and footer settings.

Using the TTITLE Command

```
TTITLE 'Salary|Report'
SELECT employee_id, last_name, salary, manager_id
FROM   employees;
```

```
Tue Jun 19                                    page    1
                      Salary
                      Report

EMPLOYEE_ID LAST_NAME              SALARY     MANAGER_ID
----------- --------------------- ---------- ----------
        100 King                       24000
        101 Kochhar                    17000        100
        102 De Haan                    17000        100
        103 Hunold                      9000        102
. . .
20 rows selected.
```

ORACLE

Using the BTITLE Command

```
BTITLE 'Confidential'
SELECT employee_id, last_name, salary, manager_id
FROM   employees;
```

```
EMPLOYEE_ID LAST_NAME                  SALARY     MANAGER_ID
----------- ------------------- ---------- ----------
        100 King                       24000
. . .
        205 Higgins                    12000        101
        206 Gietz                       8300        205

                   Confidential

20 rows selected.
```

B-10

Using the BTITLE Command

The example in the slide uses the BTITLE command to add the page footer Confidential to the bottom of each page of the report.

Tips for Using BTITLE and TTITLE

- Split the header or footer onto several lines with the vertical bar (|).
- TTITLE and BTITLE remain in effect until reset to another title or turned off, or until the SQL session is over.
- By default, the TITLE command centers the heading, displays the date in the left corner, and displays the page number in the right corner.

The following example puts the title on the left:

```
 TTITLE LEFT 'Departments'
SELECT department_name
FROM   departments;
Departments
DEPARTMENT_NAME
------------------------------
Administration
. . .
8 rows selected.
```

Additional `COLUMN` Command Options

Control display of columns and headings:

```
COL[UMN] [{column|alias} [option]]
```

- **`NEW_VALUE`: Prints data in the title**
- **`NOPRINT`: Excludes data from output**
- **`CLEAR`: Resets column display attributes**

`COLUMN` Command Options

Option	Description
`NEW_V[ALUE]`	Specifies a variable to hold a column value that can be used in the `TTITLE` command
`NOPRI[NT]` `PRI[NT]`	Controls whether or not a column is printed
`CLE[AR]` `DEF[AULT]`	Resets the display attributes for the column to the default values

Using the NEW_VALUE Command

```
COLUMN department_id new_value deptnum FORMAT 99
TTITLE SKIP 1 CENTER 'Report for Dept:' deptnum -
SKIP 2 CENTER
BREAK on department_id SKIP PAGE ON department_id
SELECT   last_name, manager_id, department_id, salary
FROM     employees
ORDER BY department_id;
```


Using the NEW_VALUE Option of the COLUMN Command

The example in the slide creates a report that separates the output onto separate pages according to department. The statement uses the NEW_VALUE COLUMN option to create a variable (deptnum) for the department number in the report, which is then used in the TTITLE statement as part of the header for each page. The BREAK ON command creates new sections in the report for each department.

```
           Report for Dept:        10

LAST_NAME                 MANAGER_ID DEPARTMENT_ID     SALARY
------------------------- ---------- ------------- ----------
Whalen                           101            10       4400
                           Report for Dept:        20
LAST_NAME                 MANAGER_ID DEPARTMENT_ID     SALARY
------------------------- ---------- ------------- ----------
Hartstein                        100            20      13000
Fay                              201                     6000
. . .
20 rows selected.
```

Using the NOPRINT Command

```
COLUMN department_id NOPRINT new_value deptnum FORMAT 99
TTITLE SKIP 1 'Report for Dept:' deptnum -
SKIP 2 CENTER
BREAK ON department_id SKIP PAGE ON department_id
SELECT   last_name, manager_id, department_id, salary
FROM     employees
ORDER BY department_id;
```


Using the NOPRINT Option of the COLUMN Command

The NOPRINT COLUMN command option provides a method of not displaying the column that is used in the NEW_VALUE command, in this case DEPARTMENT_ID. The example in the slide creates the same report as the example in the previous slide but uses the NOPRINT COLUMN option to hide the DEPARTMENT_ID column in the output.

```
Report for Dept:        10
LAST_NAME                 MANAGER_ID     SALARY
------------------------- ---------- ----------
Whalen                           101       4400
Report for Dept:        20
LAST_NAME                 MANAGER_ID     SALARY

------------------------- ---------- ----------
Hartstein                        100      13000
Fay                              201       6000
. . .
20 rows selected.
```

Using the CLEAR Command

Use the CLEAR command to reset the display attributes for columns and headings to the default values.

```
COLUMN department_id   CLEAR
COLUMN department_name CLEAR
CLEAR BREAK
```


Using the CLEAR command

The CLEAR command resets or erases the current value or setting for the specified option. The examples on the slide clear the settings for the DEPARTMENT_ID and DEPARTMENT_NAME columns. CLEAR BREAK removes the break definition set by the BREAK command.

The COMPUTE Command

Calculates and displays summary lines

```
COMP[UTE] [function [LABEL labelname] …
       OF {expr|column|alias} …
       ON {expr|column|alias|REPORT|FORM}]
```

Uses various standard computations including:

- **AVG**
- **COUNT**
- **MAXIMUM**
- **MINIMUM**

ORACLE

B-15

The COMPUTE Command

The COMPUTE command calculates and prints summary lines using the following standard computations on subsets of selected rows, or lists all COMPUTE definitions.

Function	Computes	Applies to Data types
AVG	Average of non null values	NUMBER
COUNT	Count of non null values	All types
MAX[IMUM]	Maximum value	NUMBER, CHAR, VARCHAR2
MIN[IMUM]	Minimum value	NUMBER, CHAR, VARCHAR2
STD	Standard deviation of non null values	NUMBER
SUM	Sum of non null values	NUMBER
VAR[IANCE]	Variance on non null values	NUMBER

Using the COMPUTE Command

```
BREAK ON JOB_ID SKIP 1
COMPUTE SUM OF salary ON job_id
SELECT job_id, last_name, salary
FROM   employees
WHERE  job_id IN ('ST_CLERK', 'SA_REP')
ORDER BY job_id, salary;
```

```
JOB_ID     LAST_NAME                     SALARY
---------- ------------------------- ----------
SA_REP     Grant                           7000
           Taylor                          8600
           Abel                           11000
**********                           ----------
sum                                       26600

ST_CLERK   Vargas                          2500
. . .
7 rows selected.
```


The COMPUTE Command (continued)

The example in the slide uses the COMPUTE command to calculate and display the total salary for each job title listed in the WHERE clause. The result is a list of employees for each job title, with a salary total at the end of each job section. The sectioning of the output can be performed on any column in the table. The ORDER BY clause is used on the JOB_ID column to ensure that jobs are grouped together for each total.

Using BREAK with COMPUTE

```
BREAK ON department_id SKIP 2
COMPUTE MAX OF salary ON department_id
SELECT department_id, last_name, salary
FROM   employees
WHERE  job_id IN ('ST_CLERK', 'SA_REP')
ORDER BY department_id, salary;
```

```
DEPARTMENT_ID LAST_NAME                     SALARY
------------- ------------------------- ----------
           50 Vargas                          2500
              Matos                           2600
              . . .
*************                           ----------
maximum                                       3500


           80 Taylor                          8600
. . .
7 rows selected.
```

B-17

Using BREAK with the COMPUTE Command

The example in the slide uses the BREAK and SKIP options with the COMPUTE command to display blank lines between each total.

The "maximum" line generated by the COMPUTE statement represents the highest salary of a clerk, or sales representative in each department.

Using LABEL with COMPUTE

```
BREAK ON department_id SKIP 2
COMPUTE MAX LABEL Max_Sal OF salary ON department_id
SELECT department_id, last_name, salary
FROM   employees
WHERE  job_id IN ('ST_CLERK', 'SA_REP')
ORDER BY department_id, salary;
```

```
DEPARTMENT_ID LAST_NAME                     SALARY
------------- ------------------------- ----------
           50 Vargas                          2500
              Matos                           2600
              Davies                          3100
              Rajs                            3500
*************                           ----------
Max_Sal                                       3500
. . .
7 rows selected.
```

B-18

Using LABEL with the COMPUTE Command

The example in the slide uses the `LABEL` option with the `COMPUTE` command to rename the maximum row created by the `COMPUTE` statement.

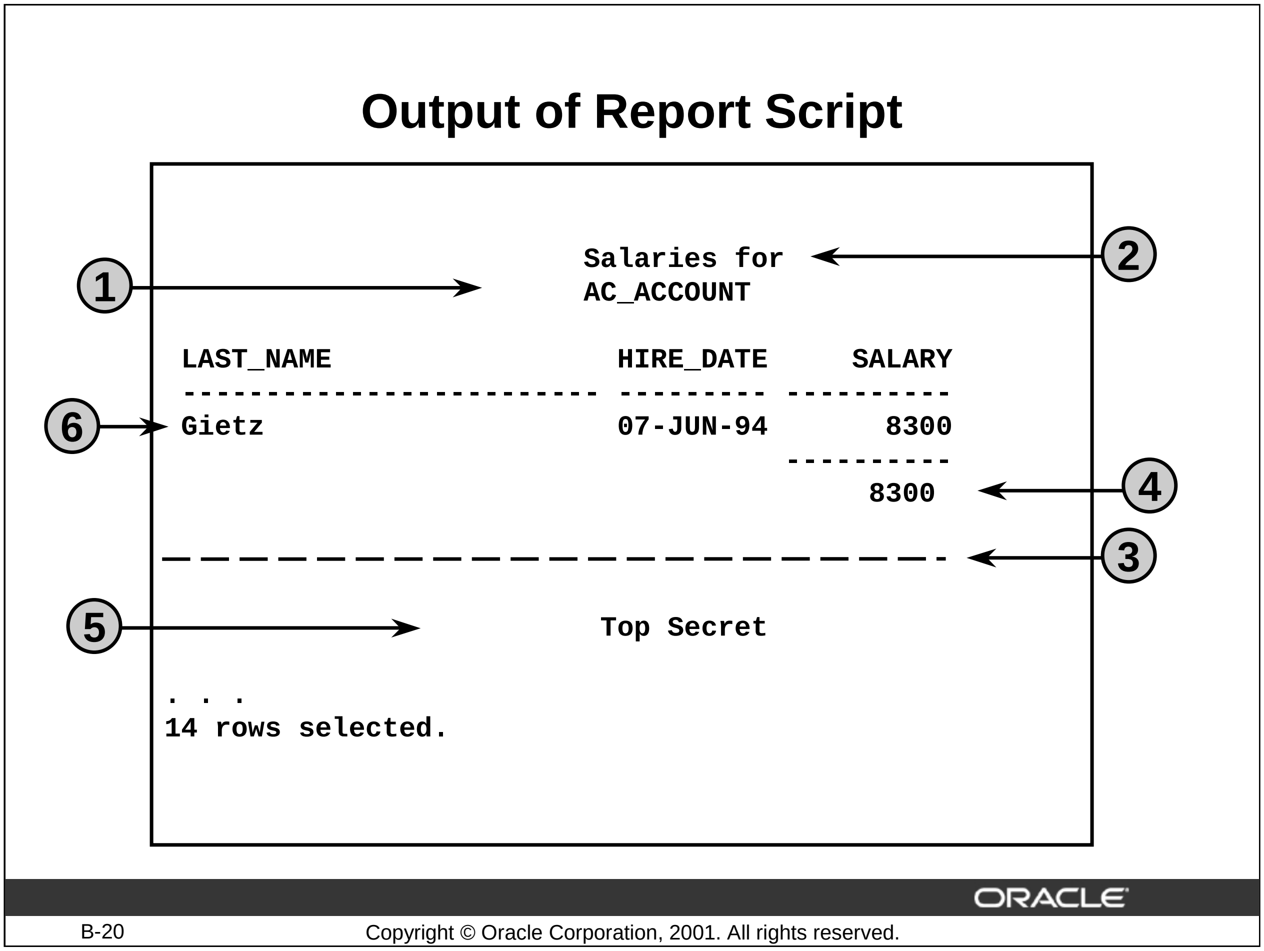
Example of Creating a Report Script

```
COLUMN job_id NOPRINT new_value jobname FORMAT A9
TTITLE -
SKIP 1 CENTER 'Salaries for' -
SKIP 1 CENTER jobname -
SKIP 2
BREAK on job_id SKIP PAGE
COMPUTE AVG LABEL '' OF salary ON job_id
BTITLE 'Top Secret'
SELECT job_id, last_name, hire_date, salary
FROM   employees
ORDER BY job_id, salary;
```

B-19

How to Create a Report Script

1. Use the `NEW_VALUE COLUMN` command to create a variable to hold the job title for each page of the report.
2. Use the `TTITLE` command to create a heading for each report page:
   - Add a centered heading, Salaries for job name, to the top of the report. Split the heading onto two lines.
   - Leave a blank line.
3. Use the `BREAK` command to start a new report page for each job title.
4. Use the `COMPUTE` command to calculate the average salary for each job.
5. Use the `BTITLE` command to add a Top Secret footer to each page of the report.
6. Add the query for the report to list all employees. Group the output by job to ensure that the rows are listed correctly above the averages for each job.
7. Save the script to a file.
8. Use the `START` command to start the script.



Output of Report Script

1. The job ID changes for each page of the report.
2. The heading appears centered above the report.
3. A new page is started when all employees for a job have been listed and the average salary has been calculated.
4. The average salary is computed for the job listed.
5. The Top Secret footer appears at the bottom of each page of the report.
6. All employees for each job title are listed.

C

Practice Solutions

Practice 1 Solutions

1. Initiate an iSQL*Plus session by using the user ID and password provided by the instructor.

2. SQL commands are always held in the buffer.

   True

3. *i*SQL*Plus commands are used to query data.

   False. The SQL command `SELECT` is used to query data.

4. Show the structure of the `DEPARTMENTS` table.

   ```
   DESC departments
   ```

5. Select all information from the `DEPARTMENTS` table.

   ```
   SELECT *
   FROM departments;
   ```

6. Show the structure of the `EMPLOYEES` table.

   ```
   DESC employees
   ```

   Using this table, perform the following actions:

7. Display the last name and hire date for each employee.

   ```
   SELECT last_name, hire_date
   FROM   employees;
   ```

8. Display the hire date and last name for each employee, with the hire date appearing first.

   ```
   SELECT hire_date, last_name
   FROM   employees;
   ```

Practice 1 Solutions (continued)

9. Display the last name, hire date, and annual salary, excluding commission, for each employee. Label the annual salary column as ANNUAL

```
SELECT last_name, hire_date, salary*12 ANNUAL
FROM   employees;
```

10. List all the specific job ids that exist in the organization.

```
SELECT DISTINCT job_id
FROM employees;
```

11. Select the last_name, department ID, and hire date for all employees. Display the data as shown:

```
SELECT last_name||’ has worked in department
   ’||department_id||’ since ’||hire_date AS
   "WHO, WHERE, AND WHEN"
FROM employees;
```

Practice 2 Solutions

1. You can order by a column that you have not selected.

   True

2. This statement will execute successfully.

```
SELECT *
FROM   employees
WHERE  salary*12=9600;
```

   True. Demonstrate this, if required.

3. Display the last name of the with the employee ID 104.

```
SELECT last_name
FROM   employees
WHERE  employee_id=104;
```

4. Display the last name, manager ID, and salary for all employees in department 20.

```
SELECT last_name, manager_id,salary
FROM   employees
WHERE  department_id = 20;
```

5. Display the last name and hire date of all employees whose last name begins with the letter H.

.

```
SELECT last_name, hire_date
FROM   employees
WHERE last_name LIKE 'H%';
```

6. Display the last name, manager ID, and salary for all employees whose salary is in the range of $6000 through $8000.

```
SELECT last_name, manager_id, salary
FROM   employees
WHERE  salary BETWEEN 6000 AND 8000;
```

Practice 2 Solutions (continued)

7. Display the employee ID and last name for all clerks (`JOB_ID = ST_CLERK`) and who work for manager 100 or 124.

```
SELECT employee_id, last_name
FROM   employees
WHERE  job_id='ST_CLERK'
AND    manager_id IN (100,124);
```

8. Display the employee ID, last name, and manager ID for all employees whose salary is greater than $2500 and who work in department 50.

```
SELECT employee_id, last_name, manager_id
FROM   employees
WHERE  salary>2500
AND    department_id=50;
```

9. Display the last names and salary for all employees who work for the manager with the manager ID 124, starting with the employee with the highest salary and ending with the employee with the lowest salary.

.

```
SELECT   last_name, salary
FROM     employees
WHERE    manager_id =124
ORDER BY salary DESC;
```

10. Display the last name, job ID, and salary for all non sales employees who are earning less than $2000 or more than $15000.

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id <> 'SA_MAN'
AND    salary NOT BETWEEN 2000 AND 15000;
```

Practice 3 Solutions

1. Single-row functions work on many rows to produce a single result.

   False

2. Display the last name and salary plus $600 for all employees in department 20. The name should be displayed in capitals.

```
SELECT UPPER(last_name) AS NAME, salary + 600
FROM   employees
WHERE  department_id = 20;
```

3. Display the employee ID, last name, and salary increased by 15% and expressed as a whole number, for all employees in department 20. Round up any cents in the new salary amounts to the nearest dollar. Give the column the heading, `SAL+15%,` as shown:

```
SELECT employee_id, last_name,
       ROUND(salary * 1.15) AS "SAL+15%"
FROM   employees
WHERE  department_id=20;
```

4. Produce the following list of employees and their jobs.

```
SELECT last_name||' works as a '
       ||LOWER(job_id) AS "Employees and Jobs"
FROM   employees;
```

5. Display the employee ID, last name, monthly commission percentage, and monthly commission pct rounded to two decimal places for all sales people. (`JOB_ID = 'SA_MAN' or JOB_ID ='SA_REP'`)

   Note: `COMMISSION_PCT` is an annual figure.

```
SELECT employee_id, last_name,commission_pct/12,
       ROUND(commission_pct/12, 2) COMM_ROUNDED
FROM   employees
WHERE  job_id in ('SA_MAN', 'SA_REP');
```

Practice 3 Solutions (continued)

6. Produce a one-column report showing the first name and last name of each employee separated by a dash (-). Give the column the heading Employee Details, as shown:

```
SELECT first_name||'- '||last_name AS
         "Employees Details"
FROM   Employees;
```

7. Display the last name, job ID, and total annual income (including commission where applicable) for all employees.

```
SELECT last_name, job_id,
        salary * 12 * (1+NVL(commission_pct,0)) ANNUAL_SAL
FROM employees;
```

8. Display the employee number, name, and salary plus the commission amount increased by 20% for all employees.

```
SELECT employee_id,last_name,
        salary*(1+NVL(commission_pct*0.2,0)) "NEW SALARY"
FROM   employees;
```

Practice 4 Solutions

1. Display the last name and hire date of all employees with the job ID IT_PROG. Display the hire date as shown:

```
SELECT last_name, TO_CHAR(hire_date,'MM/DD/YYYY') HIRED_IN
FROM   employees
WHERE  job_id ='IT_PROG';
```

2. Determine the annual salary (excluding commission) and six-month review date for all employees with the job ID ST_CLERK. Give the column an alias of REVIEW.

```
SELECT last_name, salary*12,
       ADD_MONTHS(hire_date,6) REVIEW
 FROM   employees
 WHERE  job_id ='ST_CLERK';
```

3. Display the last name and number of days between today and the start date for all employees with the letter G as the first letter of their name.

```
SELECT last_name, SYSDATE-hire_date DAYS_EMPLOYED
FROM   employees
WHERE last_name LIKE 'G%';
```

4. Display the number of months that Taylor has been employed with the company. Give the column an alias of MONTHS

```
SELECT last_name, MONTHS_BETWEEN(SYSDATE,hire_date) MONTHS
FROM   employees
WHERE  last_name='Taylor';
```

Practice 4 Solutions (continued)

5. For employees in department 20, display the last name and hire date as shown. Specify the alias as `DATE_HIRED` after your expression. Pay particular attention to the case used in the letters of the hire date.

```
SELECT last_name,
       TO_CHAR(hire_date,'fmMonth, Ddspth YYYY') AS
  DATE_HIRED
FROM   employees
WHERE  department_id=20;
```

6. For employees in department 60, display each employee's last name, hire date, and salary review date. Assume that the review date is one year after the hire date. Give the review date column an alias of `REVIEW`. Order the output in ascending order of hire date.

```
SELECT   last_name, TO_CHAR(hire_date,'DD-MON-YYYY')
  HIRE_DATE,
         TO_CHAR(ADD_MONTHS(hire_date,12),'DD-MON-YYYY')
  REVIEW
FROM     employees
WHERE    department_id=60
ORDER BY hire_date;
```

7. Display the last names of all employees who were hired after March 15, 1998. Use the date format 03/15/1998.

```
SELECT last_name
FROM   employees
WHERE  HIRE_DATE > TO_DATE('03/15/1998','MM/DD/YYYY');
```

8. Create a single-column report that lists sales representatives (`JOB_ID = 'SA_REP'`) and their monthly salaries as shown in the following output. Pay particular attention to the case used in the letters and the formatting of the salary amounts.

```
SELECT INITCAP(last_name)||
      ' earns'||TO_CHAR(salary,'$99,999')||' a month'
MONTHLYSALARIES
FROM   employees
WHERE  job_id ='SA_REP';
```

Practice 4 Solutions (continued)

9. Display the date of the first Monday in the year 2001. Give the column the heading as Monday.

```
SELECT NEXT_DAY('31-DEC-2000','Monday') AS   "Monday"
FROM   DUAL;
```

10. Display the last names and hire dates of all employees who have been with the company for more than 10 years.

```
SELECT last_name, TO_CHAR(hire_date,'DD-MON-YYYY') HIREDATE
FROM   employees
WHERE  MONTHS_BETWEEN(SYSDATE,HIRE_DATE)/12>10;
```

11. Display the last name and hire date for all employees who were hired in 1987.

```
SELECT last_name, TO_CHAR(hire_date,'DD-MON-YYYY') HIREDATE
FROM   employees
WHERE  TO_CHAR(hire_date,'DD-MON-YYYY') LIKE '%1987';
```

12. Display the last name and hire date for all employees whose job ID is `ST_CLERK`, starting with the clerk who was hired first and ending with the clerk who was hired most recently.

```
SELECT    last_name, TO_CHAR(hire_date,'DD-MON-YYYY') HIREDATE
FROM      employees
WHERE     job_id ='ST_CLERK'
ORDER BY hire_date;
```

13. Display the last name, hire date, hire date rounded to the MONTH, and hire date rounded to the YEAR for employees with an employee ID is greater than 170. The column headings should be as given below.

```
SELECT last_name, TO_CHAR(hire_date,'DD-MON-YYYY') HIREDATE,
       TO_CHAR(round(hire_date,'MONTH'),'DD-MON-YYYY')
       ROUND_MONTH,
       TO_CHAR(round(hire_date,'YEAR'),'DD-MON-YYYY') ROUND_YEAR
FROM employees
WHERE employee_id > 170;
```

Practice 5 Solutions

1. Display the last name, department ID, and department name of all employees, in department name order.

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e, departments d
WHERE  e.department_id=d.department_id
ORDER BY d.department_name;
```

2. Display the last name, salary, and department name of all employees who earn more than $10,000.

```
SELECT e.last_name, e.salary, d.department_name
FROM employees e, departments d
WHERE  e.department_id=d.department_id
AND    e.salary>10000;
```

3. Display the last name, salary, and department name for all employees in the accounting department.

```
SELECT e.last_name, e.salary, d.department_name
FROM   employees e, departments d
WHERE  e.department_id=d.department_id
AND    d.department_name='Accounting';
```

4. Display the last name, job, department name, and location ID for all employees whose office has the location ID 1400.

```
SELECT e.last_name, e.job_id,
       d.department_name, d.location_id
FROM   employees e, departments d
WHERE  e.department_id=d.department_id
AND    d.location_id=1400;
```

Practice 5 Solutions (continued)

5. Display a list of employees including last name, job, salary, and grade level.

```
SELECT e.last_name, e.job_id, e.salary, j.grade_level
FROM   employees e, job_grades j
WHERE  e.salary BETWEEN j.lowest_sal AND j.highest_sal;
```

6. Using question 5, show only employees in grade C.

```
SELECT e.last_name, e.job_id, e.salary, j.grade_level
FROM   employees e, job_grades j
WHERE  e.salary BETWEEN j.lowest_sal AND j.highest_sal
AND    j.grade_level='C';
```

7. For employees in department 20, display the last name, department ID, the name of the employee's manager and department ID of their manager.

```
SELECT e.last_name, e.department_id,
       m.last_name MANAGER, m.department_id
FROM   employees e, employees m
WHERE  e.manager_id=m.employee_id
AND    e.department_id=20;
```

8. Find all employees who joined the company before their manager.

```
SELECT e.last_name, to_char(e.hire_date,'DD-MON-YYYY')
       HIREDATE, m.last_name MGR,
       to_char(m.hire_date,'DD-MON-YYYY') HIREDATE
FROM   employees e, employees m
WHERE  e.manager_id=m.employee_id
AND    e.hire_date<m.hire_date;
```

9. For each employee, display the last name, the last name of the employee's manager and the manager's department name.

```
SELECT e.last_name, m.last_name MGR,
       d.department_name MGR_DEPT
FROM   employees e, employees m, departments d
WHERE  e.manager_id=m.employee_id
AND    m.department_id=d.department_id;
```

Practice 5 Solutions (continued)

10. Display the last name and the last name of the manager for all employees who work in the same department as their manager.

```
SELECT e.last_name, m.last_name MGR
FROM   employees e, employees m
WHERE  e.manager_id=m.employee_id
AND    e.department_id=m.department_id;
```

11. Display the employee ID, last name, department ID, department name,and city for all employees those last names begin with H.

```
SELECT employee_id, last_name, e.department_id,
       department_name, city
FROM   employees e , departments d, locations l
WHERE  e.department_id = d.department_id
AND    d.location_id = l.location_id and
       last_name like ’H%’
```

Practice 6 Solutions

1. Determine the validity of the following statements. Circle either True or False.
   a. Group functions work across many rows to produce one result.
   True

   b. Group functions include nulls in calculations.
   False

2. Find the earliest hire date of an employee.

```
SELECT TO_CHAR(MIN(hire_date),'DD-MON-YYYY') EARLIEST
FROM   employees;
```

3. Find the highest salary paid to an employee.

```
SELECT MAX(salary) MAX_SALARY
FROM   employees;
```

4. Find the total monthly salary paid to all clerks.

```
SELECT SUM(salary) CLERK_PAYROLL
FROM   employees
WHERE  job_id='ST_CLERK';
```

5. Display the maximum salary, the minimum salary, and the difference between them for staff who were hired in 1999.

```
SELECT MAX(salary), MIN(salary),
       MAX(salary)-MIN(salary) DIFFERENCE
FROM   employees
WHERE  hire_date
BETWEEN TO_DATE('01-JAN-1999','DD-MON-YYYY') AND
TO_DATE('31-DEC-1999','DD-MON-YYYY');
```

6. Find the minimum, average, and maximum salaries of all employees.

```
SELECT MIN(salary) LOWEST, AVG(salary) AVERAGE,
       MAX(salary) HIGHEST
FROM   employees;
```

Practice 6 Solutions (continued)

7. Display the minimum and maximum salary for each job ID.

```
SELECT   job_id, MIN(salary) MIN_SAL, MAX(salary) MAX_SAL
FROM     employees
GROUP BY job_id;
```

8. Determine the number of managers without listing them.

```
SELECT count(distinct(manager_id)) "No. of managers"
FROM   employees;
```

9. Find the average monthly salary and average annual income for each job ID. Remember that only salesmen earn commission.

```
SELECT  job_id, AVG(salary) Average_Salary,
        AVG(12*salary*(1+NVL(commission_pct,0)))
        Average_Annual_Income
 FROM     employees
 GROUP BY job_id;
```

10. Display the department numbers and the total number of employees working for each department. Order the results in the descending order of the number of employees in each department.

```
SELECT department_id,count(*) TOTAL_EMPLOYEES
FROM   employees
GROUP BY department_id
ORDER BY count(*) DESC;
```

Practice 7 Solutions

1. Answer the following questions:
    a. Which query runs first with a subquery?

    Inner query.

    b. You cannot use the equal operator if the inner query returns more than one value.

    i. If the answer is true, why, and what operator should be used ?

    ii. If the answer is false, why

    True

    The equal operator expects one value in return. Use the IN operator.

2. Display the last name, manager ID, and salary for all employees in the same department as Matos.

```
SELECT last_name, manager_id, salary
FROM   employees
WHERE  department_id =
                      (SELECT department_id
                       FROM   employees
                       WHERE last_name ='Matos');
```

3. Display the employee ID, last name, and salary for all employees with a salary above the average salary.

```
SELECT employee_id, last_name, salary
FROM   employees
WHERE  salary >
              (SELECT AVG(salary)
               FROM   employees);
```

4. Display the last name and salary for all employees who have the same manager as Zlotkey.

```
SELECT last_name, salary
FROM   employees
WHERE  manager_id =
                   (SELECT manager_id
                    FROM   employees
                      WHERE last_name ='Zlotkey');
```

Practice 7 Solutions (continued)

5. Find the employees who earn the same salary as the highest salary in each job ID. Sort in descending order of the salary.

```
SELECT   last_name, job_id, salary Highest_salary
FROM     employees
WHERE    salary IN
                  (SELECT   MAX(salary)
                   FROM     employees
                        GROUP BY job_id)
ORDER BY salary DESC;
```

6. Find the employees who earn the same salary as the lowest salary for a job. Sort in ascending order of the salary.

```
SELECT   last_name, job_id, salary LOWEST_SALARY
FROM     employees
WHERE    salary IN
                   (SELECT   MIN(salary)
                    FROM    employees
                    GROUP BY job_id)
 ORDER BY salary;
```

Practice 7 Solutions (continued)

7. Display all the employees who have worked longer than Gietz.

```
SELECT last_name, TO_CHAR(hire_date,'DD-MON-YYYY') HIREDATE
FROM   employees
WHERE  hire_date < (SELECT hire_date
                    FROM   employees
                    WHERE  last_name ='Gietz');
```

8. Display the last name and job ID for all the employees (excluding sales people) with an annual salary greater than the average annual remuneration `AVG(12*salary*(1+NVL(commission_pct,0)))` for sales people.
   Hint: (`JOB_ID = 'SA_REP'`)

```
SELECT last_name, job_id
FROM   employees
WHERE  salary*12 >
   (SELECT AVG(12*salary*(1+NVL(commission_pct,0)))
    FROM   employees
    WHERE  job_id = 'SA_REP')
AND     job_id <> 'SA_REP';
```

9. Display the names and salaries for all employees who work out of the Oxford office.
   `Hint`: Use the `LOCATIONS` table to retrieve the city.

```
SELECT last_name OXFORDTEAM, salary
FROM   employees
WHERE  department_id =
       (SELECT department_id
        FROM departments
        WHERE location_id =
                 (SELECT location_id
                  FROM locations
                  WHERE city = 'Oxford'));
```

Practice 7 Solutions (continued)

10. Display the employee ID and last names for all employees who report to King.

```
SELECT employee_id, last_name
FROM   employees
WHERE  manager_id =
                (SELECT employee_id
                 FROM   employees
                 WHERE  last_name='King');
```

11. Display all the employees whose manager works in department 20.

```
SELECT last_name
FROM   employees
WHERE  manager_id IN
                   (SELECT employee_id
                    FROM   employees
                    WHERE  department_id = 20);
```

12. Display the department ID, last names and job ids for all employees who work in the sales department.

```
SELECT department_id, last_name, job_id
FROM   employees
WHERE  department_id =
                   (SELECT department_id
                    FROM   departments
                    WHERE  department_name = 'Sales');
```

Practice 8 Solutions

1. a. A single ampersand substitution variable prompts only once.

   False. The single ampersand substitution variable prompts every time the command is executed.

   b. The `DEFINE` command is a SQL statement

   False. The `DEFINE` command is a iSQL*Plus command. It is issued within a SQL script file.

2. Write a statement that prompts a user for a department number at run time and then displays the employee last name, number, and salary for each employee in the department:

```
SELECT last_name, employee_id, salary
FROM   employees
WHERE  department_id = &department_number;
```

3. Write a script that prompts the user for two dates in the `DD-MON-YYYY` format. The script displays the employee last name, number, salary and hire date of each employee hired between these two dates. Save the script as `8Lab3.sql` using the `Save Script` button.

```
SELECT last_name, employee_id salary,
         TO_CHAR(hire_date,'DD-MON-YYYY') HIREDATE
FROM   employees
WHERE  hire_date BETWEEN
TO_DATE( '&low_date','DD-MON-YYYY')
AND  TO_DATE( '&high_date','DD-MON-YYYY');
```

Practice A Solutions

Insert data into the MY_EMPLOYEE table.

1. Run the statement in the labA_1.sql script to build the MY_EMPLOYEE table that will be used for the lab.

```
CREATE TABLE my_employee
(id  NUMBER(4) CONSTRAINT my_employee_id_nn NOT NULL,
 last_name VARCHAR2(25),
 first_name VARCHAR2(25),
 userid  VARCHAR2(8),
 salary  NUMBER(9,2));
```

2. Describe the structure of the MY_EMPLOYEE table to identify the column names.

```
DESCRIBE my_employee
```

3. Add the first row of data to the MY_EMPLOYEE table from the following sample data. Do not list the columns in the INSERT clause.

ID	LAST_NAME	FIRST_NAME	USERID	SALARY
1	Patel	Ralph	rpatel	895
2	Dancs	Betty	bdancs	860
3	Biri	Ben	bbiri	1100
4	Newman	Chad	cnewman	750
5	Ropeburn	Audrey	aropebur	1550

```
INSERT INTO my_employee
VALUES (1, 'Patel', 'Ralph', 'rpatel', 895);
```

4. Populate the MY_EMPLOYEE table with the second row of sample data from the preceding list. This time, list the columns explicitly in the INSERT clause.

```
INSERT INTO my_employee (id, last_name, first_name,
                         userid, salary)
 VALUES (2, 'Dancs', 'Betty', 'bdancs', 860);
```

5. Confirm your addition to the table.

```
SELECT   *
FROM     my_employee;
```

Practice A Solutions (continued)

6. Write an insert statement in a text file named `loademp.sql` to load rows into the `MY_EMPLOYEE` table. Concatenate the first letter of the first name and the first seven characters of the last name to produce the userid.

```
SET VERIFY OFF
INSERT INTO my_employee
VALUES (&p_id, '&p_last_name', '&p_first_name',
        lower(substr('&p_first_name', 1, 1) ||
        substr('&p_last_name', 1, 7)), &p_salary);
SET VERIFY ON
```

7. Populate the table with the next three rows of sample data by running the insert statement in the script that you created.

```
SET VERIFY OFF
 INSERT INTO my_employee
 VALUES (&p_id, '&p_last_name', '&p_first_name',
         lower(substr('&p_first_name', 1, 1) ||
         substr('&p_last_name', 1, 7)), &p_salary);
 SET VERIFY ON
```

8. Confirm your additions to the table.

```
SELECT   *
FROM my_employee;
```

9. Make the data additions permanent.

```
COMMIT;
```

Practice A Solutions (continued)

Update and delete data in the `MY_EMPLOYEE` table.

10. Change the last name of employee 3 to Drexler.

```
UPDATE  my_employee
SET     last_name = 'Drexler'
WHERE   id = 3;
```

11. Change the salary to 1000 for all employees with a salary less than 900.

```
UPDATE  my_employee
SET     salary = 1000
WHERE   salary < 900;
```

12. Verify your changes to the table.

```
SELECT  id,last_name, first_name,userid,salary
FROM    my_employee;
```

13. Delete Betty Dancs from the `MY_EMPLOYEE` table.

```
DELETE
FROM  my_employee
WHERE last_name = 'Dancs';
```

14. Confirm your changes to the table.

```
SELECT  *
FROM    my_employee;
```

15. Commit all pending changes.

```
COMMIT;
```

Control data transaction to the `MY_EMPLOYEE` table.

16. Populate the table with the to add Betty Dancs data by using the script that you created in step 6.

Practice A Solutions (continued)

```
SET VERIFY OFF
INSERT INTO my_employee VALUES (&p_id, '&p_last_name',
     '&p_first_name',lower(substr('&p_first_name', 1, 1)
        || substr('&p_last_name', 1, 7)), &p_salary);
SET VERIFY ON
```

17. Confirm your addition to the table.

```
SELECT *
  FROM      my_employee;
```

18. Mark an intermediate point in the processing of the transaction.

```
SAVEPOINT step_18;
```

19. Empty the entire table.

```
DELETE
 FROM  my_employee;
```

20. Confirm that the table is empty.

```
SELECT *
  FROM   my_employee;
```

21. Discard the most recent `DELETE` operation without discarding the earlier `INSERT` operation.

```
ROLLBACK TO step_18;
```

22. Confirm that the the most recent `DELETE` has been discarded.

```
SELECT *
FROM   my_employee;
```

D

Table Descriptions and Data

COUNTRIES Table

```
DESCRIBE countries
```

Name	Null?	Type
COUNTRY_ID	NOT NULL	CHAR(2)
COUNTRY_NAME		VARCHAR2(40)
REGION_ID		NUMBER

```
SELECT * FROM countries;
```

CO	COUNTRY_NAME	REGION_ID
CA	Canada	2
DE	Germany	1
UK	United Kingdom	1
US	United States of America	2

DEPARTMENTS Table

```
DESCRIBE departments
```

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

```
SELECT * FROM departments;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting		1700

8 rows selected.

EMPLOYEES Table

```
DESCRIBE employees
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

```
SELECT * FROM employees;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE
100	Steven	King	SKING	515.123.4567	17-JUN-87
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98
144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-98
149	Eleni	Zlotkey	EZLOTKEY	011.44.1344.429018	29-JAN-00
174	Ellen	Abel	EABEL	011.44.1644.429267	11-MAY-96
176	Jonathon	Taylor	JTAYLOR	011.44.1644.429265	24-MAR-98
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-99
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94

20 rows selected.

EMPLOYEES Table (continued)

JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
AD_PRES	24000			90
AD_VP	17000		100	90
AD_VP	17000		100	90
IT_PROG	9000		102	60
IT_PROG	6000		103	60
IT_PROG	4200		103	60
ST_MAN	5800		100	50
ST_CLERK	3500		124	50
ST_CLERK	3100		124	50
ST_CLERK	2600		124	50
ST_CLERK	2500		124	50
SA_MAN	10500	.2	100	80
SA_REP	11000	.3	149	80
SA_REP	8600	.2	149	80
SA_REP	7000	.15	149	
AD_ASST	4400		101	10
MK_MAN	13000		100	20
MK_REP	6000		201	20
AC_MGR	12000		101	110
AC_ACCOUNT	8300		205	110

20 rows selected.

JOBS Table

```
DESCRIBE jobs
```

Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)

```
SELECT * FROM jobs;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
AC_MGR	Accounting Manager	8200	16000
AC_ACCOUNT	Public Accountant	4200	9000
SA_MAN	Sales Manager	10000	20000
SA_REP	Sales Representative	6000	12000
ST_MAN	Stock Manager	5500	8500
ST_CLERK	Stock Clerk	2000	5000
IT_PROG	Programmer	4000	10000
MK_MAN	Marketing Manager	9000	15000
MK_REP	Marketing Representative	4000	9000

12 rows selected.

`JOB_GRADES` Table

```
DESCRIBE job_grades
```

Name	Null?	Type
GRADE_LEVEL		VARCHAR2(3)
LOWEST_SAL		NUMBER
HIGHEST_SAL		NUMBER

```
SELECT * FROM job_grades;
```

GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

6 rows selected.

JOB_HISTORY Table

```
DESCRIBE job_history
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

```
SELECT * FROM job_history;
```

EMPLOYEE_ID	START_DAT	END_DATE	JOB_ID	DEPARTMENT_ID
102	13-JAN-93	24-JUL-98	IT_PROG	60
101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
101	28-OCT-93	15-MAR-97	AC_MGR	110
201	17-FEB-96	19-DEC-99	MK_REP	20
114	24-MAR-98	31-DEC-99	ST_CLERK	50
122	01-JAN-99	31-DEC-99	ST_CLERK	50
200	17-SEP-87	17-JUN-93	AD_ASST	90
176	24-MAR-98	31-DEC-98	SA_REP	80
176	01-JAN-99	31-DEC-99	SA_MAN	80
200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

10 rows selected.

LOCATIONS Table

```
DESCRIBE locations
```

Name	Null?	Type
LOCATION_ID	NOT NULL	NUMBER(4)
STREET_ADDRESS		VARCHAR2(40)
POSTAL_CODE		VARCHAR2(12)
CITY	NOT NULL	VARCHAR2(30)
STATE_PROVINCE		VARCHAR2(25)
COUNTRY_ID		CHAR(2)

```
SELECT * FROM locations;
```

LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	CO
1400	2014 Jabberwocky Rd	26192	Southlake	Texas	US
1500	2011 Interiors Blvd	99236	South San Francisco	California	US
1700	2004 Charade Rd	98199	Seattle	Washington	US
1800	460 Bloor St. W.	ON M5S 1X8	Toronto	Ontario	CA
2500	Magdalen Centre, The Oxford Science Park	OX9 9ZB	Oxford	Oxford	UK

REGIONS Table

```
DESCRIBE regions
```

Name	Null?	Type
REGION_ID	NOT NULL	NUMBER
REGION_NAME		VARCHAR2(25)

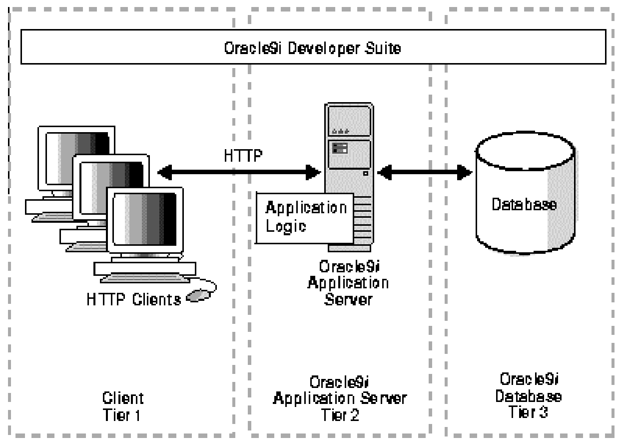
```
SELECT * FROM regions;
```

REGION_ID	REGION_NAME
1	Europe
2	Americas
3	Asia
4	Middle East and Africa

E

Oracle9*i* Architecture

Oracle9*i* Architecture



E-2

Oracle9*i* Architecture

Oracle9*i* components include the following:

- Oracle9*i* Database
- Oracle9*i* Application Server
- Oracle9*i* Developer Suite

Oracle9*i* Database

The Oracle9*i* Database introduces the following advanced and automated design features that refine Oracle9*i* Application Server and Oracle9*i* Developer Suite to optimize performance for traditional applications and the emerging hosted application market.

- **Oracle9*i* Real Application Clusters:** The next evolutionary step after Oracle Parallel Server, Oracle9*i* Real Application Clusters provides out-of-the-box, linear scaling transparency, compatibility with all applications without redesign, and the ability to rapidly add nodes and disks.
- **Systems Management**: Integrated system management products create a complete view of all critical components that drive e-business processes. From the client and application server to the database and host, Oracle9*i* quickly and completely assesses the overall health of an e-business infrastructure.
- **High Availability and Security**: Setting a new standard for high availability, Oracle9*i* introduces powerful new functionality in the areas of disaster recovery, system fault recovery, and planned downtime. Oracle9*i* offers the most secure Internet platform for protecting company information through multiple layers of security for data, users, and companies.

Oracle9*i* Architecture (Continued)

Included are features for building Internet-scale applications, for providing security for users, and for keeping data from different hosted user communities separate.

Oracle9*i* Application Server

Recognized as the leading application server for database-driven Web sites, Oracle9*i* Application Server offers the industry's most innovative and comprehensive set of middle-tier services.

- **Comprehensive Middle-tier Services**: Continued innovation within comprehensive middle-tier services, ranging from self-service enterprise portals, to e-stores and supplier exchange, sustains the Oracle9*i* Application Server as the industry's preferred application server for database-driven Web sites.
- **New Caching Technology**: The new caching technology in Oracle9*i* can dramatically increase Web-site performance, scalability, and availability. Greater numbers of users can be provided with more personalized, dynamic Web content without adding more application or database servers.
- **Scalability and Performance**: Superb scalability and performance is now made available for all Web applications. Oracle Portal services make it easy for Web site developers to deploy enterprise portals with centralized management and unified security. Standard Java, with rich XML and content management support, as well as back-office transactional applications built using Oracle Forms Developer, can easily be deployed.
- **Wireless Device Access**: Oracle9*i*AS Wireless can provide access to all your existing applications and content from any wireless Web device.
- **Business Intelligence**: Oracle9*i* Application Server has built-in reporting and ad-hoc query functionality to derive business intelligence after Web site deployment.

Oracle9*i* Developer Suite

Oracle9*i* Developer Suite (formerly known as Oracle Internet Developer Suite) is a complete, integrated suite of development tools for rapidly developing transactional Internet applications and Web services using Java and XML. Oracle9*i* Developer Suite supports any language, any operating system, any development style, any phase of the development life-cycle, and any of the latest Internet standards.

Components of Oracle9i Developer Suite:

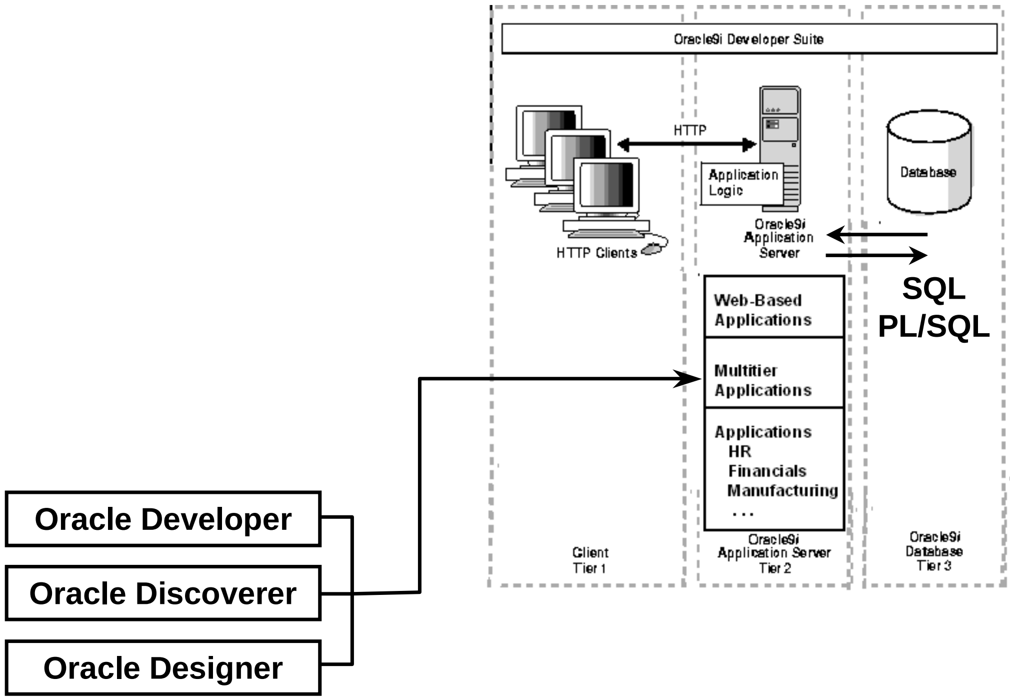
Internet Applications: Oracle Designer including Oracle Software Configuration Manager (Oracle SCM), Oracle Forms Developer, and Oracle JDeveloper including Oracle Business Components for Java.

Business Intelligence: Oracle Reports Developer, Oracle Discoverer, and Oracle Warehouse Builder.

How are these components integrated ?

The diagram on the slide depicts a three-tier database architecture. Client machines in the first tier connect through the HTTP protocol with an Oracle9*i* Application Server in the second tier. The Oracle9*i* Application Server connects with an Oracle9*i* database in the third tier. The Oracle9*i* Developer Suite interfaces with all the tiers in the architecture.

Oracle’s Complete Solution



ORACLE

E-4

Oracle’s Complete Solution

The Oracle object relational database management system (ORDBMS) is the Oracle core product. It includes the Oracle9*i* Server and several tools that assist users in maintaining, monitoring, and using data. The Oracle data dictionary is one of the most important components of the server. It consists of a set of tables and views that provide a read-only reference to the database.

The ORDBMS handles various tasks such as the following:

- Managing the storage and definition of data
- Controlling and restricting data access and concurrency
- Providing backup and recovery
- Interpreting SQL and PL/SQL statements

Note: PL/SQL is an Oracle procedural language that extends SQL by adding application logic.

SQL and PL/SQL statements are used by all programs and users to access and manipulate data stored in the Oracle database. In some application programs, you may access the database without directly writing SQL or PL/SQL commands. For example you may click a button or select a check box, but the applications implicitly use SQL or PL/SQL to execute the request.

*i*SQL*Plus is an Oracle tool that recognizes and submits SQL and PL/SQL statements to the server for execution and contains its own command language.

Oracle offers a wide variety of state-of-the-art graphical user interface (GUI) driven tools to build business applications as well as a large suite of software applications for many areas of business and industry.

Index

Symbols

A

B

C

T

U

W