Oracle Database: Introduction to SQL Accelerated - IBM Graduate Program

Student Guide – Vol 1 (Oracle Database 10*g*: SQL Fundamentals I)

D59008GC10
Edition 1.0
June 2010

ORACLE

Contents

Introduction

Lesson Objectives

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

- List the features of Oracle10*g*
- Discuss the theoretical and physical aspects of a relational database
- Describe the Oracle implementation of RDBMS and ORDBMS
- The Data Models and their purposes
- Understand the goals of the course

ORACLE

Objectives

In this lesson, you gain an understanding of the relational database management system (RDBMS) and the object relational database management system (ORDBMS). You are also introduced to the following:

- SQL statements that are specific to Oracle
- *i*SQL*Plus, which is an environment used for executing SQL statements and for formatting and reporting purposes

Goals of the Course

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

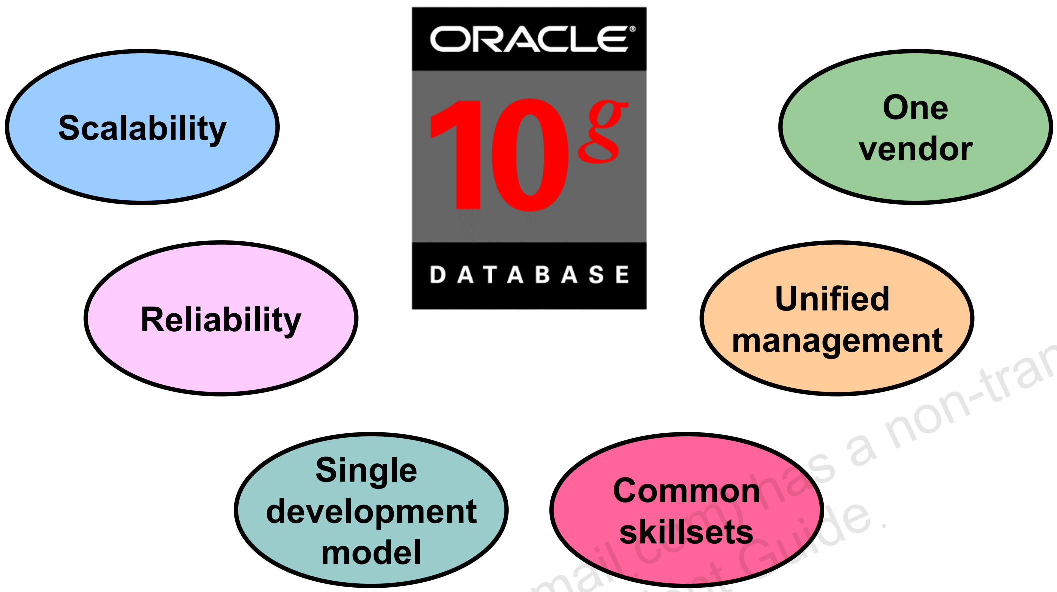
- Identify the major structural components of Oracle Database 10*g*
- Retrieve row and column data from tables with the `SELECT` statement
- Create reports of sorted and restricted data
- Employ SQL functions to generate and retrieve customized data
- Run data manipulation language (DML) statements to update data in Oracle Database 10*g*
- Obtain metadata by querying the dictionary views

ORACLE

Goals of the Course

This course offers you an introduction to Oracle Database 10*g* database technology. In this class, you learn the basic concepts of relational databases and the powerful SQL programming language. This course provides the essential SQL skills that enable you to write queries against single and multiple tables, manipulate data in tables, create database objects, and query metadata.

Oracle10*g*



Oracle10*g* Features

The Oracle10*g* release offers a comprehensive high-performance infrastructure, including:

- Scalability from departments to enterprise e-business sites
- Robust, reliable, available, and 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

In addition to providing the benefits listed above, the Oracle10*g* release contains the database for the grid. Grid computing can dramatically lower the cost of computing, extend the availability of computing resources, and deliver higher productivity and quality.

The basic idea of grid computing is the notion of computing as a utility, analogous to the electric power grid or the telephone network. As a client of the grid, you do not care where your data is or where your computation is done. You want to have your computation done and to have your information delivered to you when you want it. From the server side, grid is about virtualization and provisioning. You pool all your resources together and provision these resources dynamically based on the needs of your business, thus achieving better resource utilization at the same time.

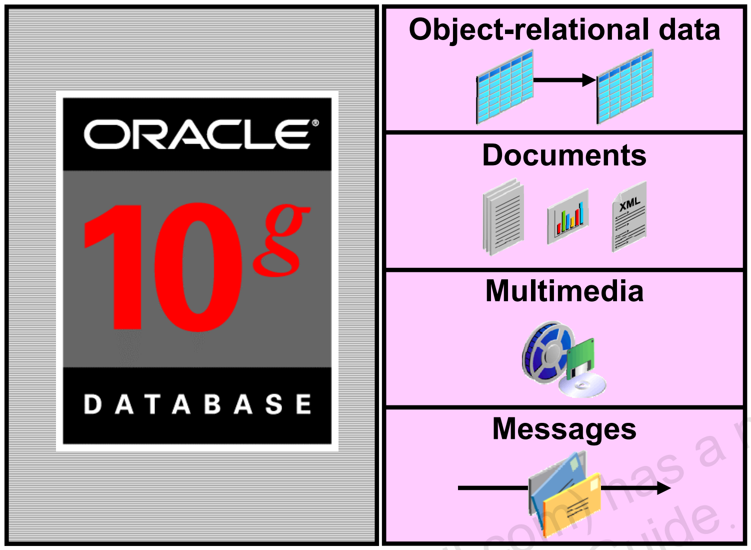
Oracle10*g*

Oracle10*g*

The three grid-infrastructure products of the Oracle10*g* release are:

- Oracle Database 10*g*
- Oracle Application Server 10*g*
- Oracle Enterprise Manager 10*g* Grid Control

Oracle Database 10*g*



Oracle Database 10*g*

Oracle Database 10*g* is designed to store and manage enterprise information. Oracle Database 10*g* cuts management costs and provides a high quality of service. Reduced configuration and management requirements and automatic SQL tuning have significantly reduced the cost of maintaining the environment.

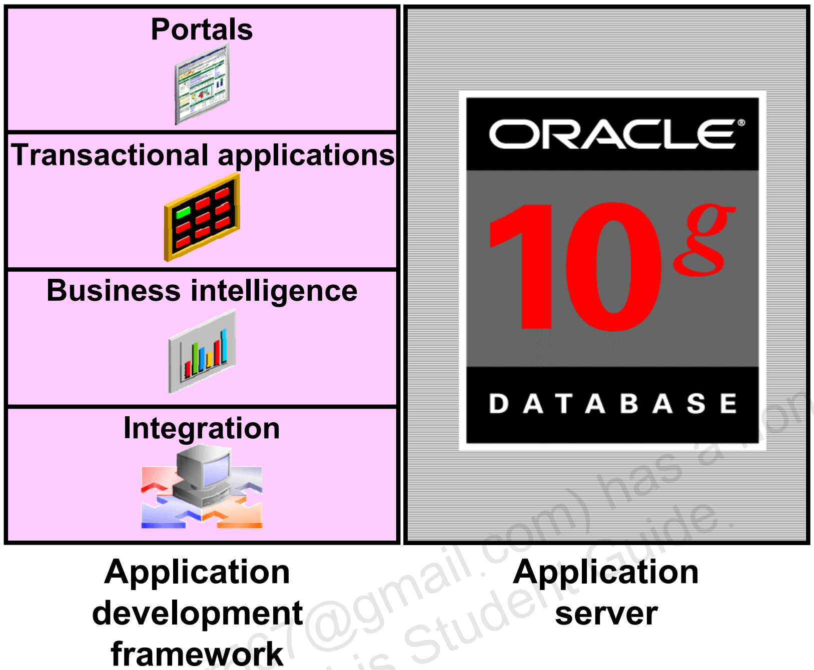
Oracle Database 10*g* contributes to the grid-infrastructure products of the Oracle 10*g* release. Grid computing is all about computing as a utility. If you are a client, you need not know where your data resides and which computer stores it. You should be able to request information or computation on your data and have it delivered to you.

Oracle Database 10*g* 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 such as:

- 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. Oracle Database 10*g* 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.

Oracle Application Server 10*g*



Oracle Application Server 10*g*

Oracle Application Server 10*g* provides a complete infrastructure platform for developing and deploying enterprise applications, integrating many functions including a J2EE and Web services run-time environment, an enterprise portal, an enterprise integration broker, business intelligence, Web caching, and identity management services.

Oracle Application Server 10*g* adds new grid computing features, building on the success of Oracle9*i* Application Server, which has hundreds of customers running production enterprise applications.

Oracle Application Server 10*g* is the only application server to include services for all the different server applications that you might want to run, including:

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

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

Oracle Enterprise Manager 10*g* Grid Control

- Software provisioning
- Application service level monitoring

Oracle Enterprise Manager 10*g* Grid Control

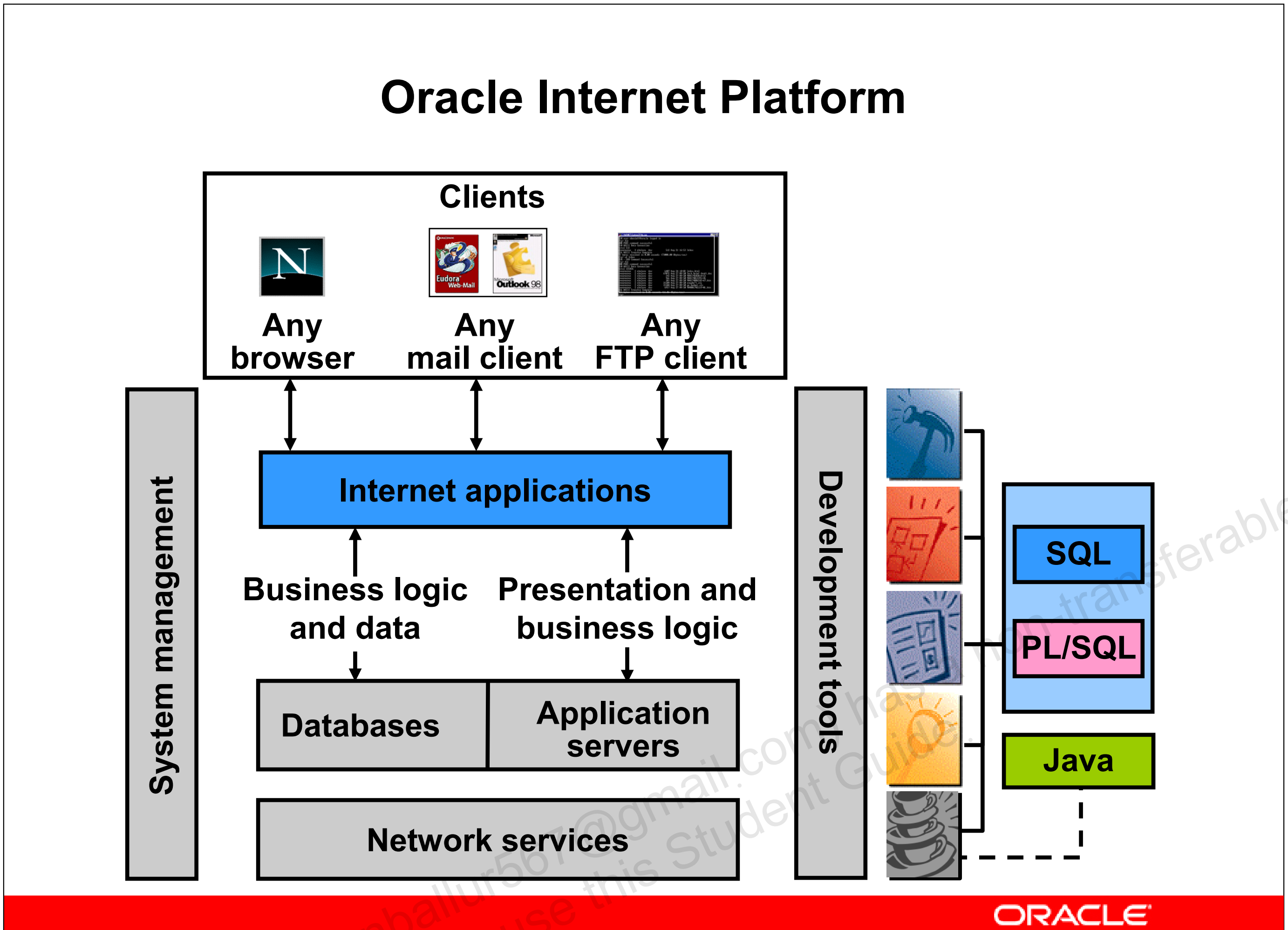
Oracle Enterprise Manager 10*g* Grid Control is the complete, integrated, central management console and underlying framework that automates administrative tasks across sets of systems in a grid environment. With Oracle Grid Control, you can group multiple hardware nodes, databases, application servers, and other targets into single logical entities. By executing jobs, enforcing standard policies, monitoring performance, and automating many other tasks across a group of targets instead of on many systems individually, Grid Control enables scaling with a growing grid.

Software Provisioning

With Grid Control, Oracle 10*g* automates installation, configuration, and cloning of Application Server 10*g* and Database 10*g* across multiples nodes. Oracle Enterprise Manager provides a common framework for software provisioning and management, enabling administrators to create, configure, deploy, and utilize new servers with new instances of the application server and database as they are needed.

Application Service Level Monitoring

Oracle Grid Control views the availability and performance of the grid infrastructure as a unified whole, as a user would experience it, rather than as isolated storage units, processing boxes, databases, and application servers.



Oracle Internet Platform

To develop an e-commerce application, you need a product that can store and manage the data, a product that can provide a run-time environment for your applications implementing business logic, and a product that can monitor and diagnose the application after it is integrated. The Oracle 10*g* products provide all the necessary components to develop your application.

Oracle offers a comprehensive high-performance Internet platform for e-commerce and data warehousing. The integrated Oracle Internet Platform includes everything needed to develop, deploy, and manage Internet applications, including these 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. Oracle Developer Suite includes tools to develop forms and reports and to build data warehouses. Stored procedures, functions, and packages can be written using SQL, PL/SQL, or Java.

Relational and Object Relational Database Management Systems

- Relational model and object relational model
- User-defined data types and objects
- Fully compatible with relational database
- Support of multimedia and large objects
- High-quality database server features

About the Oracle Server

The Oracle server supports both the relational and object relational models.

The Oracle server extends the data-modeling capabilities to support an object relational database model that brings object-oriented programming, complex data types, complex business objects, and full compatibility with the relational world.

It includes several features for improved performance and functionality of online transaction processing (OLTP) applications, such as better sharing of run-time data structures, larger buffer caches, and deferrable constraints. Data warehouse applications benefit from enhancements such as parallel execution of insert, update, and delete operations; partitioning; and parallel-aware query optimization. Operating within the Network Computing Architecture (NCA) framework, the Oracle model supports client/server and Web-based applications that are distributed and multitiered.

For more information about the relational and object relational model, see the *Database Concepts* manual.

Relational Database Concept

- Dr. E. F. Codd proposed the relational model for database systems in 1970.
- It is the basis for the relational database management system (RDBMS).
- 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

Relational Model

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

The common models used at that time were hierarchical and network, or even 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, a number of innovative vendors, such as Oracle, supplemented the RDBMS with a suite of powerful application development and user products, providing a total solution.

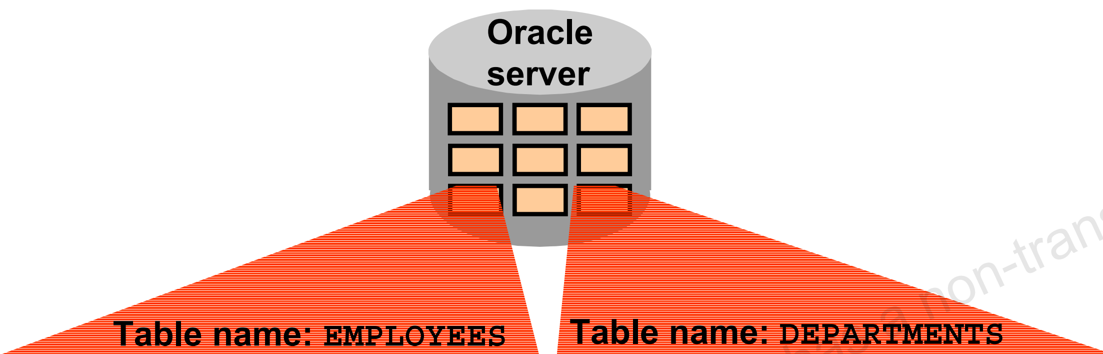
Components of the Relational Model

- 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 *An Introduction to Database Systems, Eighth Edition* (Addison-Wesley: 2004), written by Chris Date.

Definition of a Relational Database

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



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PI
100	Steven	King	SKING	51
101	Neena	Kochhar	NKOCHHAR	51
102	Lex	De Haan	LDEHAAN	51

…

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID
10	Administration	200
20	Marketing	201
50	Shipping	124

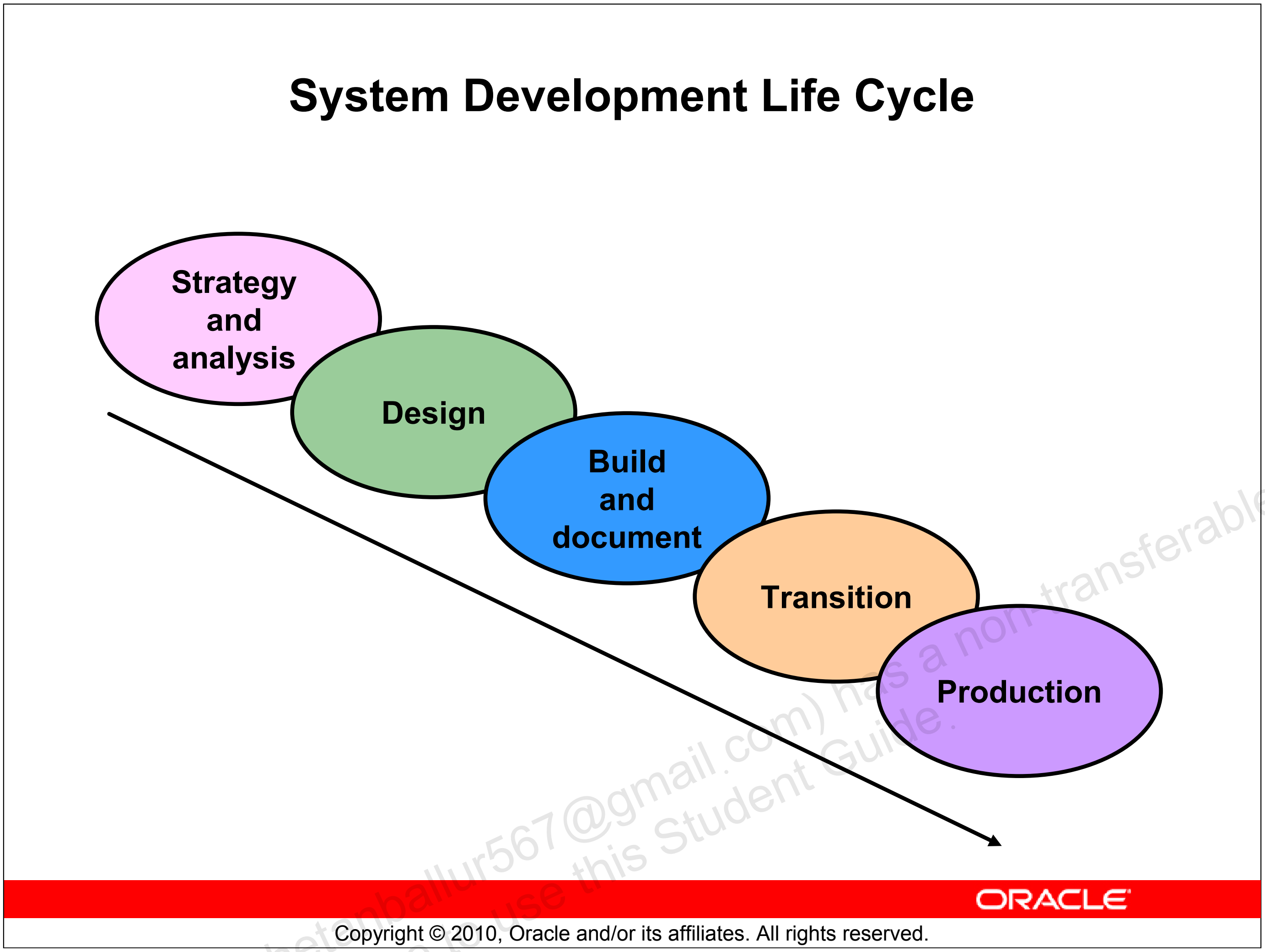
…

ORACLE

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 create several tables to store different pieces of information about your employees, such as an employee table, a department table, and a salary table.



System Development Life Cycle

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

Strategy and Analysis Phase

- 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.

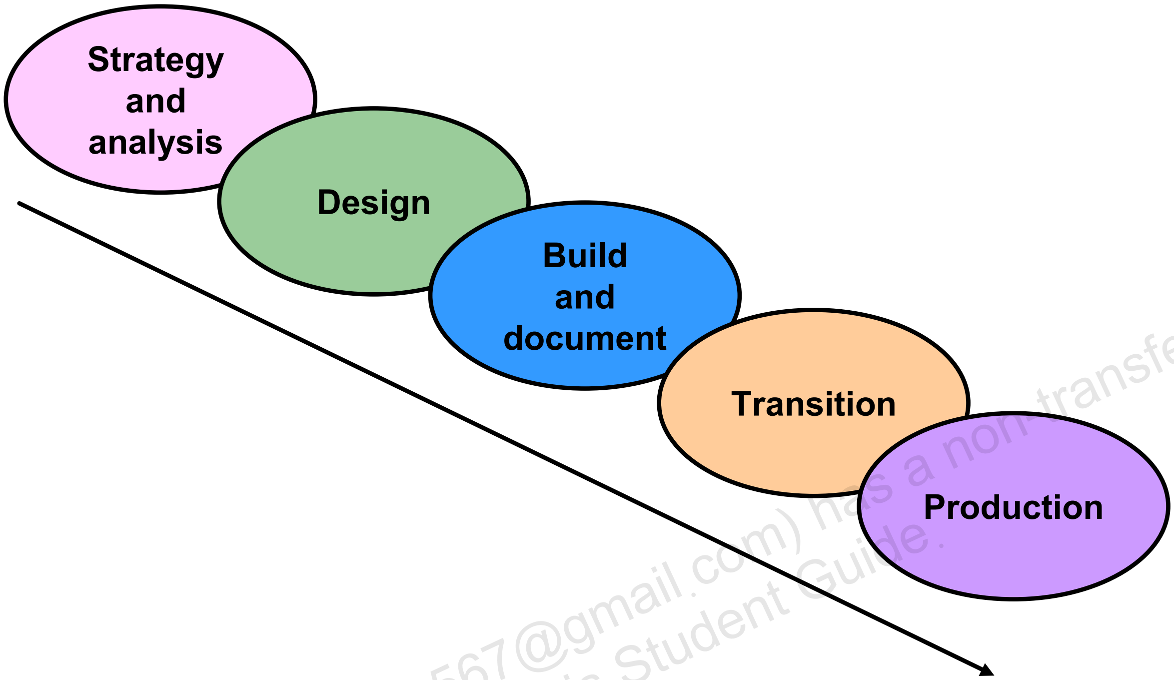
Design Phase

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

Build and Documentation Phase

- 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 operations manuals to support the use and operation of the system.

System Development Life Cycle



System Development Life Cycle (continued)

Transition Phase

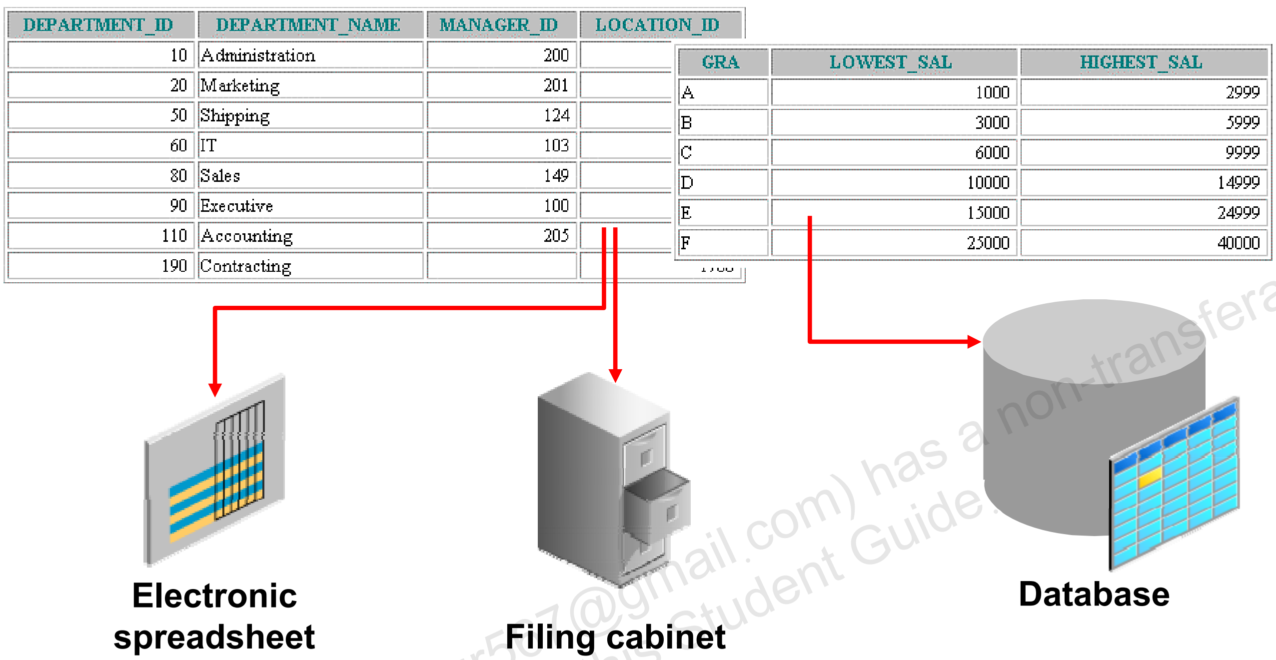
Refine the prototype. Move an application into production with user-acceptance testing, conversion of existing data, and parallel operations. Make any modifications required.

Production Phase

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	
20	Marketing	201	
50	Shipping	124	
60	IT	103	
80	Sales	149	
90	Executive	100	
110	Accounting	205	
190	Contracting		

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

Storing Information

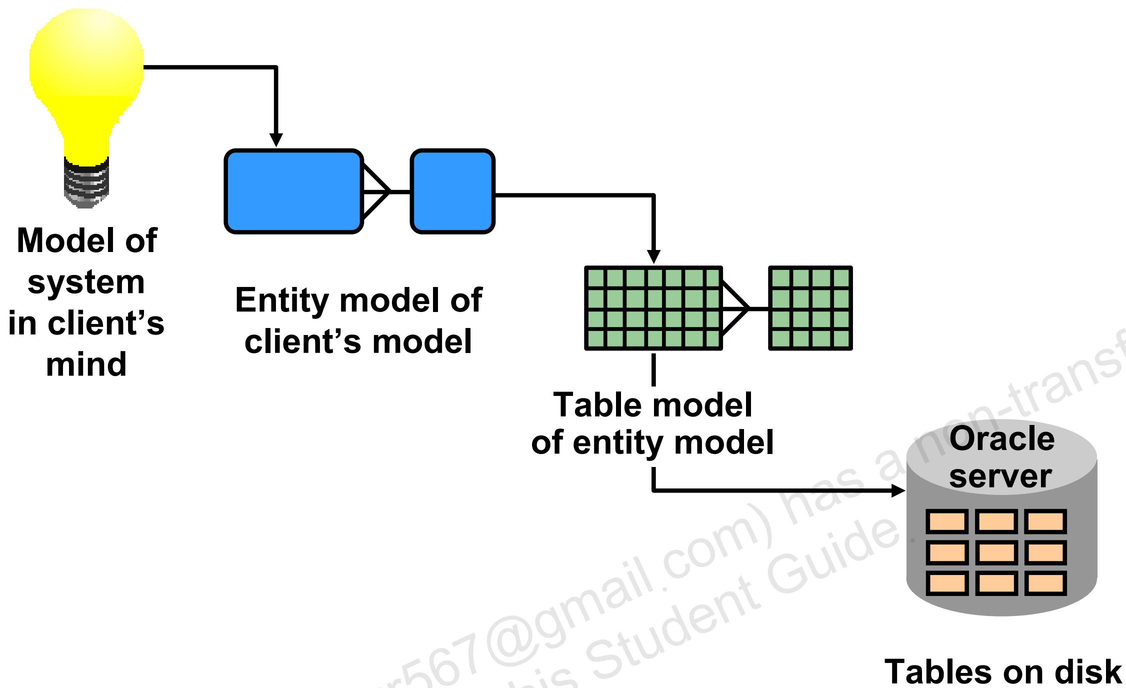
Every organization has some information needs. A library keeps a list of members, books, due dates, and fines. A company needs to save information about employees, departments, and salaries. These pieces of information are called *data*.

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

A *database* is an organized collection of information.

To manage databases, you need a database management system (DBMS). A DBMS is a program that stores, retrieves, and modifies data in databases on request. There are four main types of databases: *hierarchical*, *network*, *relational*, and (most recently) *object relational*.

Data Models



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 database design.

Purpose of Models

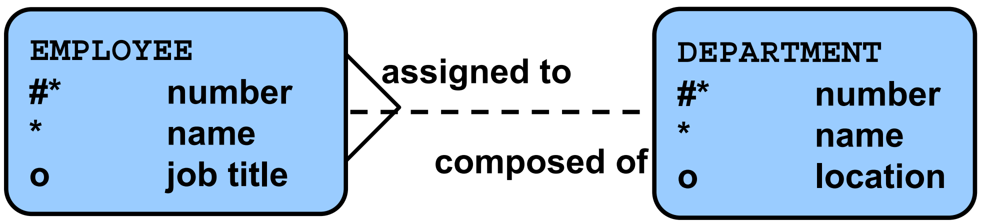
Models help communicate the concepts that are in people's minds. They can be used to do the following:

- 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.

Entity Relationship Model

- Create an entity relationship diagram from business specifications or narratives:



- Scenario
  – “. . . Assign one or more employees to a department . . .”
  – “. . . Some departments do not yet have assigned employees . . .”

ORACLE

ER Modeling

In an effective system, data is divided into discrete categories or entities. An entity relationship (ER) model is an illustration of various entities in a business and the relationships among them. An ER model is derived from business specifications or narratives and built during the analysis phase of the system development life cycle. ER models separate the information required by a business from the activities performed within a business. Although businesses can change their activities, the type of information tends to remain constant. Therefore, the data structures also tend to be constant.

ER Modeling (continued)

Benefits of ER Modeling

- Documents information for the organization in a clear, precise format
- Provides a clear picture of the scope of the information requirement
- Provides an easily understood pictorial map for database design
- Offers an effective framework for integrating multiple applications

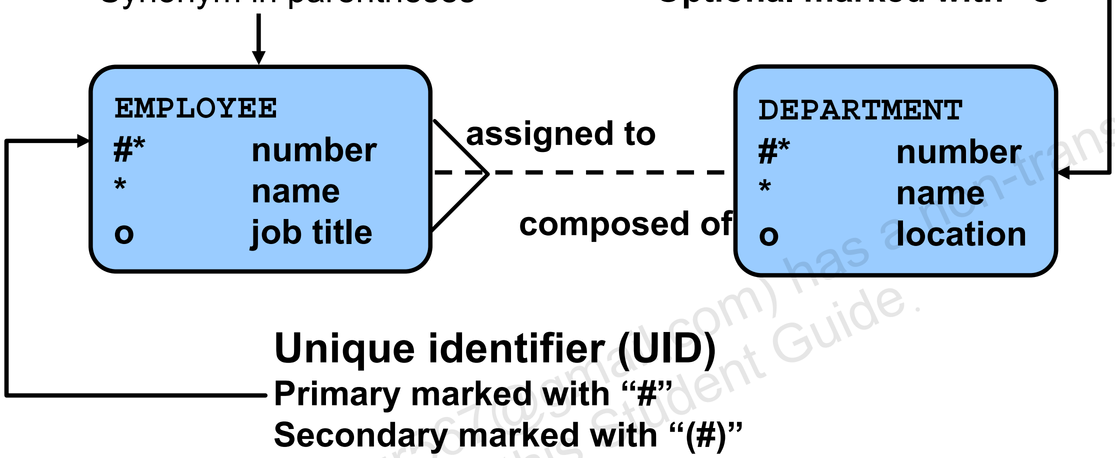
Key Components

- **Entity:** A thing of significance about which information needs to be known. Examples are departments, employees, and orders.
- **Attribute:** Something that describes or qualifies an entity. For example, for the employee entity, the attributes would be the employee number, name, job title, hire date, department number, and so on. Each of the attributes is either required or optional. This state is called *optionality*.
- **Relationship:** A named association between entities showing optionality and degree. Examples are employees and departments, and orders and items.

Entity Relationship Modeling Conventions

Entity
- Singular, unique name
- Uppercase
- Soft box
- Synonym in parentheses

Attribute
- Singular name
- Lowercase
- Mandatory marked with *
- Optional marked with “o”



ORACLE

ER Modeling Conventions

Entities

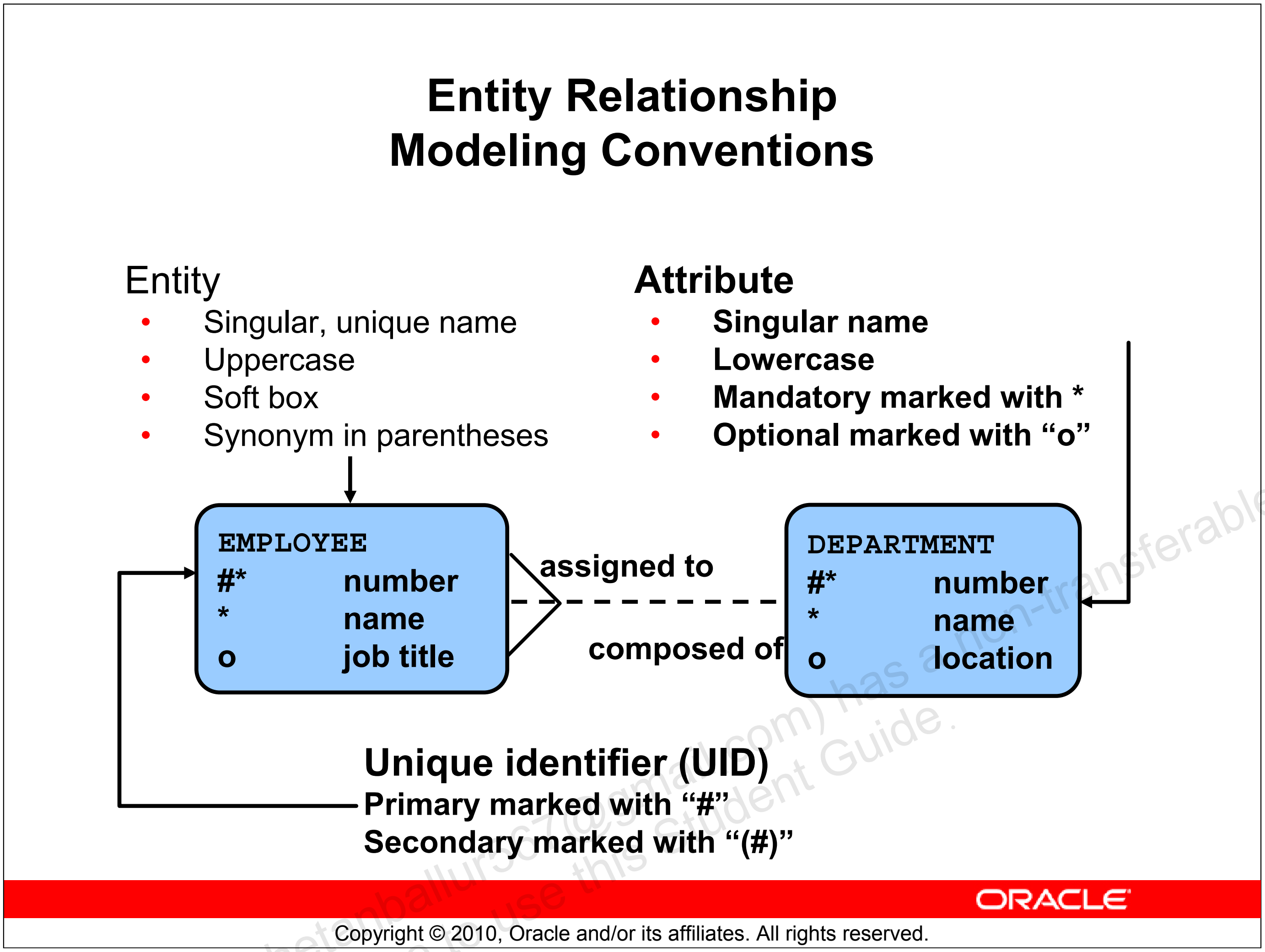
To represent an entity in a model, use the following conventions:
- Singular, unique entity name
- Entity name in uppercase
- Soft box
- Optional synonym names in uppercase within parentheses: ( )

Attributes

To represent an attribute in a model, use the following conventions:
- Singular name in lowercase
- Asterisk (*) tag for mandatory attributes (that is, values that *must* be known)
- Letter “o” tag for optional attributes (that is, values that *may* be known)

Relationships

Symbol	Description
Dashed line	Optional element indicating “maybe”
Solid line	Mandatory element indicating “must be”
Crow’s foot	Degree element indicating “one or more”
Single line	Degree element indicating “one and only one”



ER Modeling Conventions (continued)

Relationships

Each direction of the relationship contains:

- **A label:** for example, *taught by* or *assigned to*
- **An optionality:** either *must be* or *maybe*
- **A degree:** either *one and only one* or *one or more*

Note: The term *cardinality* is a synonym for the term *degree*.

Each source entity {may be | must be} relationship name {one and only one | one or more} destination entity.

Note: The convention is to read clockwise.

Unique Identifiers

A unique identifier (UID) is any combination of attributes or relationships, or both, that serves to distinguish occurrences of an entity. Each entity occurrence must be uniquely identifiable.

- **Tag each attribute that is part of the UID with a number sign:** #
- **Tag secondary UIDs with a number sign in parentheses:** (#)

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

…

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

ORACLE

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). With an RDBMS, you can relate the data in one table to the data in another by using the foreign keys. A foreign key is a column (or a set of columns) that refers to a primary key in the same table or another table.

You can use the ability to relate data in one table to data in another 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

- You cannot use duplicate values in a primary key.
- Primary keys generally cannot be changed.
- 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 it must be null.
- A foreign key must reference either a primary key or a unique key column.

Relational Database Terminology

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	SALARY	COMMISSION_PCT	DEPARTMENT_ID
100	King	Steven	24000		90
101	Kochhar	Neena	17000		90
102	De Haan	Lex	17000		90
103	Hunold	Alexander	9000		60
104	Ernst	Bruce	6000		60
107	Lorentz	Diana	4200		60
124	Mourgos	Kevin	5800		50
141	Rajs	Trenna	3500		50
142	Davies	Curtis	3100		50
143	Matos	Randall	2600		50
144	Vargas	Peter	2500		50
149	Zlotkey	Eleni	10500	.2	80
174	Abel	Ellen	11000	.3	80
176	Taylor	Jonathon	8600	.2	80
178	Grant	Kimberely	7000	.15	
200	Whalen	Jennifer	4400		10
201	Hartstein	Michael	13000		20
202	Fay	Pat	6000		20
205	Higgins	Shelley	12000		110
206	Gietz	William	8300		110

ORACLE

Terminology Used in a Relational Database

A relational database can contain one or many tables. A *table* is the basic storage structure of an RDBMS. A table holds all the data necessary about something in the real world, such as 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 permits 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 number identifies a *unique* employee in the `EMPLOYEES` table. In this example, the employee number 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 this example, the data is the salaries of all the employees. Column order is insignificant when storing data; specify the column order when the data is retrieved.

Terminology Used in a Relational Database (continued)

4. A column containing the department number, which is also a *foreign key*. A foreign key is a column 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.
5. A *field* can be found at the intersection of a row and a column. There can be only one value in it.
6. A field may have no value in it. This is called a null value. In the EMPLOYEES table, only those employees who have the role of sales representative have a value in the COMMISSION_PCT (commission) field.

Relational Database Properties

A relational database:

- Can be accessed and modified by executing structured query language (SQL) statements
- Contains a collection of tables with no physical pointers
- Uses a set of operators

Properties of a Relational Database

In a relational database, you do not specify the access route to the tables, and you do not need to know how the data is arranged physically.

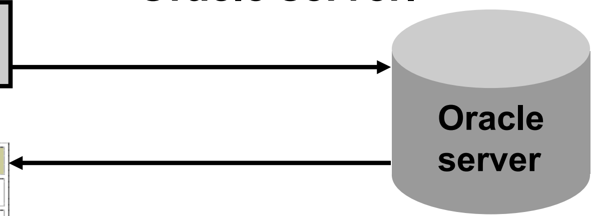
To access the database, you execute a structured query language (SQL) statement, which is the American National Standards Institute (ANSI) standard language for operating relational databases. The language contains a large set of operators for partitioning and combining relations. The database can be modified by using the SQL statements.

Communicating with an RDBMS Using SQL

SQL statement is entered.

```
SELECT department_name
FROM   departments;
```

Statement is sent to Oracle server.



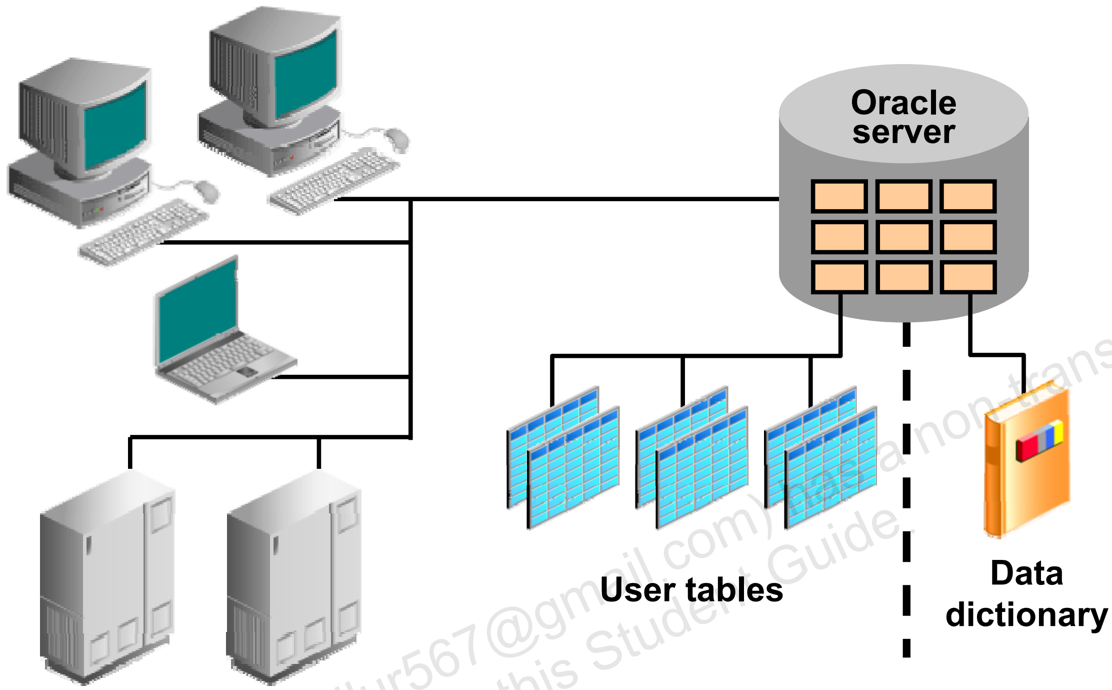
DEPARTMENT_NAME
Administration
Marketing
Shipping
IT
Sales
Executive
Accounting
Contracting

Structured Query Language

Using SQL, you can communicate with the Oracle server. SQL has the following advantages:

- Efficient
- Easy to learn and use
- Functionally complete (With SQL, you can define, retrieve, and manipulate data in the tables.)

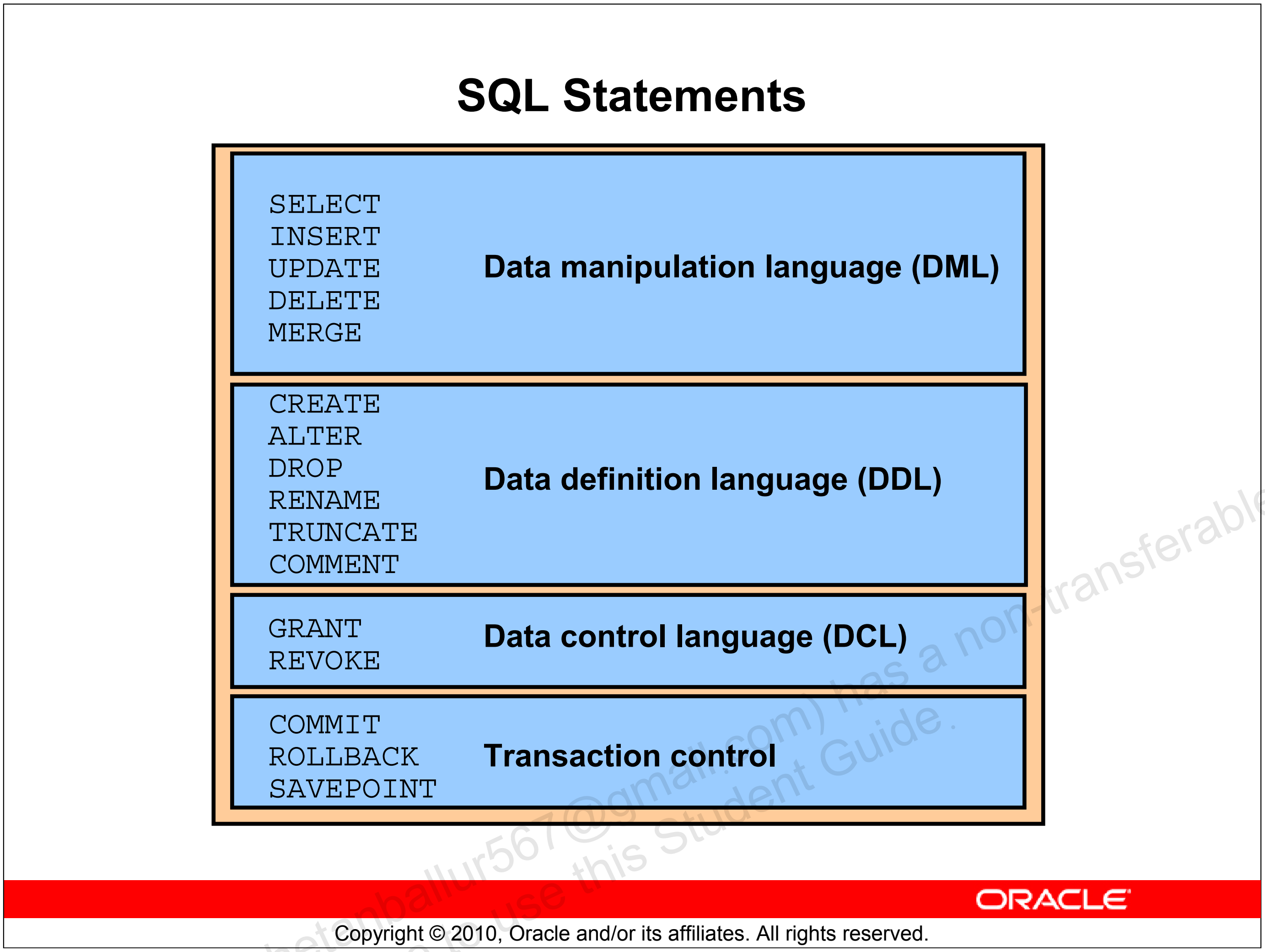
Oracle's Relational Database Management System



Oracle's Relational Database Management System

Oracle provides a flexible RDBMS called Oracle Database 10*g*. Using its features, you can store and manage data with all the advantages of a relational structure plus PL/SQL, an engine that provides you with the ability to store and execute program units. Oracle Database 10*g* also supports Java and XML. The Oracle server offers the options of retrieving data based on optimization techniques. It includes security features that control how a database is accessed and used. Other features include consistency and protection of data through locking mechanisms.

The Oracle10*g* release provides an open, comprehensive, and integrated approach to information management. An Oracle server consists of an Oracle database and an Oracle server instance. Every time a database is started, a system global area (SGA) is allocated and Oracle background processes are started. The SGA is an area of memory that is used for database information shared by the database users. The combination of the background processes and memory buffers is called an Oracle *instance*.



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 American National Standards Institute (ANSI) and International Standards Organization (ISO). Both ANSI and ISO have accepted SQL as the standard language for relational databases.

Statement	Description
SELECT INSERT UPDATE DELETE MERGE	Retrieves data from the database, enters new rows, changes existing rows, and removes unwanted rows from tables in the database, respectively. Collectively known as *data manipulation language* (DML).
CREATE ALTER DROP RENAME TRUNCATE COMMENT	Sets up, changes, and removes data structures from tables. Collectively known as *data definition language* (DDL).
GRANT REVOKE	Gives or removes access rights to both the Oracle database and the structures within it.
COMMIT ROLLBACK SAVEPOINT	Manages the changes made by DML statements. Changes to the data can be grouped together into logical transactions.

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
				1.2874	15-MAR-98	ST_CLERK	26
				1.2004	09-JUL-98	ST_CLERK	25

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

ORACLE

Tables Used in the Course

The following main tables are used in this course:

- `EMPLOYEES` table: Gives details of all the employees
- `DEPARTMENTS` table: Gives details of all the departments
- `JOB_GRADES` table: Gives details of salaries for various grades

Note: The structure and data for all the tables are provided in Appendix A.

Summary

- Oracle Database 10*g* is the database for grid computing.
- The database is based on the object relational database management system.
- Relational databases are composed of relations, managed by relational operations, and governed by data integrity constraints.
- With the Oracle server, you can store and manage information by using the SQL language and PL/SQL engine.

Summary

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

Oracle Corporation produces products and services to meet your RDBMS needs. The main products are the following:

- Oracle Database 10*g*, with which you store and manage information by using SQL
- Oracle Application Server 10*g*, with which you run all of your applications
- Oracle Enterprise Manager 10*g* Grid Control, which you use to manage and automate administrative tasks across sets of systems in a grid environment

SQL

The Oracle server supports ANSI-standard SQL and contains extensions. SQL is the language that is used to communicate with the server to access, manipulate, and control data.

1

Writing SQL `SELECT` Statements

Objectives

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

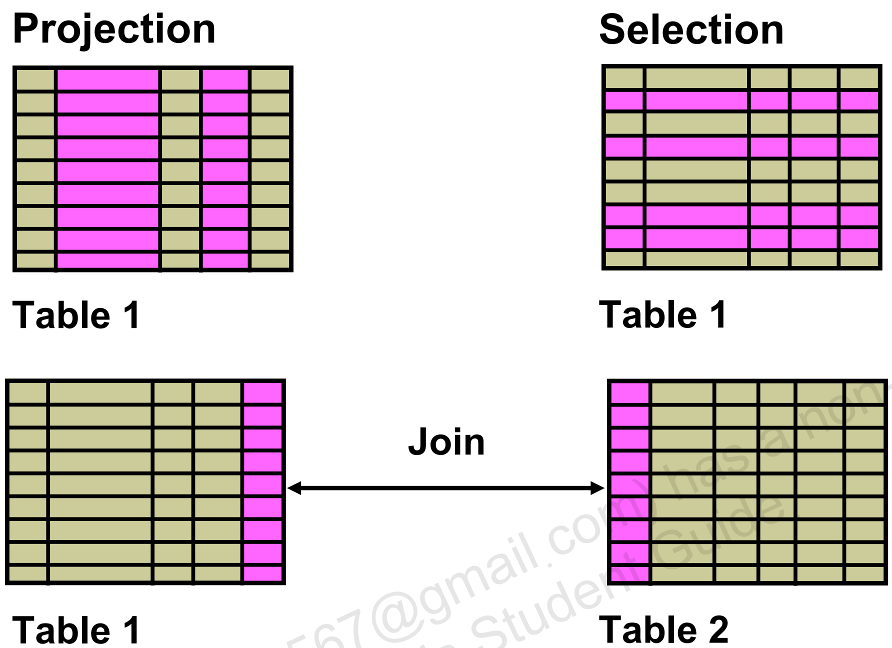
- List the capabilities of SQL `SELECT` statements
- Execute a basic `SELECT` statement
- Differentiate between SQL statements and *i*SQL*Plus commands
- Using the *i*SQL*Plus environment

Objectives

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 are needed to perform these actions. You may want to create `SELECT` statements that can be used more than once.

This lesson also covers the *i*SQL*Plus environment in which you execute SQL statements.

Capabilities of SQL SELECT Statements



Capabilities of SQL SELECT Statements

A SELECT statement retrieves information from the database. With a SELECT statement, you can use the following capabilities:

- **Projection:** Choose the columns in a table that are returned by a query. Choose as few or as many of the columns as needed.
- **Selection:** Choose the rows in a table that are returned by a query. Various criteria can be used to restrict the rows that are retrieved.
- **Joining:** Bring together data that is stored in different tables by specifying the link between them. SQL joins are covered in more detail in the lesson titled “Displaying Data from Multiple Tables.”

Basic SELECT Statement

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

- SELECT identifies the columns to be displayed.
- FROM identifies the table containing those columns.

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 identifies the table containing the columns that are listed in the SELECT clause

In the syntax:

SELECT	is a list of one or more columns
*	selects all columns
DISTINCT	suppresses duplicates
column|*expression*	selects the named column or the expression
alias	gives selected columns different headings
FROM *table*	specifies the table containing the columns

Note: Throughout this course, the words *keyword*, *clause*, and *statement* are used as follows:

- 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.

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 of All Rows

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

You can also display all columns in the table by listing all the columns after the SELECT keyword. For example, the following SQL statement (like the example in the slide) displays all columns and all rows of the DEPARTMENTS table:

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

Selecting Specific Columns

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

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

8 rows selected.

ORACLE

Selecting Specific Columns of 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 location numbers from the DEPARTMENTS table.

In the SELECT clause, specify the columns that you want, in the order in which you want them to appear in the output. For example, to display location before department number going from left to right, you use the following statement:

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

LOCATION_ID	DEPARTMENT_ID
1700	10
1800	20
1500	50

...

8 rows selected.

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.
- Indents are used to enhance readability.
- In *i*SQL*Plus, SQL statements can optionally be terminated by a semicolon (;). Semicolons are required if you execute multiple SQL statements.
- In SQL*Plus, you are required to end each SQL statement with a semicolon (;).

ORACLE

Writing SQL Statements

Using the following simple rules and guidelines, you can construct valid statements that are both easy to read and easy to edit:

- SQL statements are not case sensitive (unless indicated).
- SQL statements can be entered on one or many lines.
- Keywords cannot be split across lines or abbreviated.
- Clauses are usually placed on separate lines for readability and ease of editing.
- Indents should be used to make code more readable.
- Keywords typically are entered in uppercase; all other words, such as table names and columns, are entered in lowercase.

Executing SQL Statements

Using *i*SQL*Plus, click the Execute button to run the command or commands in the editing window.

Using SQL*Plus, terminate the SQL statement with a semicolon and then press the Enter key to run the command.

Column Heading Defaults

- *i*SQL*Plus:
  – Default heading alignment: Center
  – Default heading display: Uppercase
- SQL*Plus:
  – Character and Date column headings are left-aligned
  – Number column headings are right-aligned
  – Default heading display: Uppercase

Column Heading Defaults

In *i*SQL*Plus, column headings are displayed in uppercase and centered.

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

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
...		
Higgins	07-JUN-94	12000
Gietz	07-JUN-94	8300

20 rows selected.

You can override the column heading display with an alias. Column aliases are covered later in this lesson.

Arithmetic Expressions

Create expressions with number and date data by using arithmetic operators.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

Arithmetic Expressions

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

Arithmetic Operators

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

Note: With the `DATE` and `TIMESTAMP` data types, you can use the addition and subtraction operators only.

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

...

20 rows selected.

Using Arithmetic Operators

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

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

Note: The Oracle server ignores blank spaces before and after the arithmetic operator.

Operator Precedence

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

You can use parentheses to force the expression that is enclosed by parentheses to be evaluated first.

Rules of Precedence:

- Multiplication and division occur before addition and subtraction.
- Operators of the same priority are evaluated from left to right.
- Parentheses are used to override the default precedence or to clarify the statement.

Operator Precedence

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

1

LAST_NAME	SALARY	12*SALARY+100
King	24000	288100
Kochhar	17000	204100
De Haan	17000	204100

...

20 rows selected.

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

2

LAST_NAME	SALARY	12*(SALARY+100)
King	24000	289200
Kochhar	17000	205200
De Haan	17000	205200

...

20 rows selected.

ORACLE

Operator Precedence (continued)

The first example in the slide displays the last name, salary, and annual compensation of employees. It calculates the annual compensation by multiplying the monthly salary by 12, plus a one-time bonus of $100. Note that multiplication is performed before addition.

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

Using Parentheses

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

The second example in the slide displays the last name, salary, and annual compensation of employees. It calculates the annual compensation as follows: adding a monthly bonus of $100 to the monthly salary, and then multiplying that subtotal by 12. Because of the parentheses, addition takes priority over multiplication.

Defining a Null Value

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

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

LAST_NAME	JOB_ID	SALARY	COMMISSION_PCT
King	AD_PRES	24000	
Kochhar	AD_VP	17000	
...			
Zlotkey	SA_MAN	10500	.2
Abel	SA_REP	11000	.3
Taylor	SA_REP	8600	.2
...			
Gietz	AC_ACCOUNT	8300	

20 rows selected.

ORACLE

Null Values

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

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

Columns of any data type can contain nulls. However, some constraints (`NOT NULL` and `PRIMARY KEY`) prevent nulls from being used in the column.

In the `COMMISSION_PCT` column in the `EMPLOYEES` table, notice that only a sales manager or sales representative can earn a commission. Other employees are not entitled to earn commissions. A null represents that fact.

Null Values in Arithmetic Expressions

Arithmetic expressions containing a null value evaluate to null.

```
SELECT last_name, 12*salary*commission_pct
FROM   employees;
```

LAST_NAME	12*SALARY*COMMISSION_PCT
King	
Kochhar	

…

Zlotkey	25200
Abel	39600
Taylor	20640

…

Gietz	

20 rows selected.

ORACLE

Null Values in Arithmetic Expressions

If any column value in an arithmetic expression is null, the result is null. For example, if you attempt to perform division by 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, employee King does not get any commission. Because the `COMMISSION_PCT` column in the arithmetic expression is null, the result is null.

For more information, see “Basic Elements of SQL” in *SQL Reference*.

Defining a Column Alias

A column alias:

- Renames a column heading
- Is useful with calculations
- Immediately follows the column name (There can also be the optional `AS` keyword between the column name and alias.)
- Requires double quotation marks if it contains spaces or special characters or if it is case sensitive

Column Aliases

When displaying the result of a query, *i*SQL*Plus normally uses the name of the selected column as the column heading. This heading may not be descriptive and, therefore, maybe difficult to understand. You can change a column heading by using a column alias.

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

Using Column Aliases

```
SELECT last_name AS name, commission_pct comm
FROM   employees;
```

NAME	COMM
King	
Kochhar	
De Haan	

...

20 rows selected.

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

Name	Annual Salary
King	288000
Kochhar	204000
De Haan	204000

...

20 rows selected.

ORACLE

Column Aliases (continued)

The first example displays the names and the commission percentages of all the employees. Notice that the optional `AS` keyword has been used before the column alias name. The result of the query is the same whether the `AS` keyword is used or not. Also notice that the SQL statement has the column aliases, `name` and `comm`, in lowercase, whereas the result of the query displays the column headings in uppercase. As mentioned in a previous slide, column headings appear in uppercase by default.

The second example displays the last names and annual salaries of all the employees. Because `Annual Salary` contains a space, it has been enclosed in double quotation marks. Notice that the column heading in the output is exactly the same as the column alias.

Concatenation Operator

A concatenation operator:

- Links columns or character strings to other columns
- Is represented by two vertical bars (||)
- Creates a resultant column that is a character expression

```
SELECT   last_name||job_id AS "Employees"
FROM     employees;
```

Employees
KingAD_PRES
KochharAD_VP
De HaanAD_VP

...

20 rows selected.

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.

In the example, `LAST_NAME` and `JOB_ID` are concatenated, and they are given the alias `Employees`. Notice that the employee last name and job code are combined to make a single output column.

The `AS` keyword before the alias name makes the `SELECT` clause easier to read.

Null Values with the Concatenation Operator

If you concatenate a null value with a character string, the result is a character string. `LAST_NAME || NULL` results in `LAST_NAME`.

Literal Character Strings

- A literal is a character, a number, or a date that is included in the `SELECT` statement.
- Date and character literal values must be enclosed by single quotation marks.
- Each character string is output once for each row returned.

Literal Character Strings

A literal is a character, a number, or a date that is included in the `SELECT` list and that is not a column name or a column alias. It is printed for each row returned. Literal strings of free-format text can be included in the query result and are treated the same as a column in the `SELECT` list.

Date and character literals *must* be enclosed by single quotation marks (`' '`); number literals need not be so enclosed.

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

...

20 rows selected.

ORACLE

Literal Character Strings (continued)

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

In the following example, the last name and salary for each employee are 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
Mourgos: 1 Month salary = 5800
Rajs: 1 Month salary = 3500

...

20 rows selected.

Alternative Quote (q) Operator

- Specify your own quotation mark delimiter
- Choose any delimiter
- Increase readability and usability

```
SELECT department_name ||
       q'[, it's assigned Manager Id: ]'
       || manager_id
       AS "Department and Manager"
FROM departments;
```

Department and Manager
Administration, it's assigned manager ID: 200
Marketing, it's assigned manager ID: 201
Shipping, it's assigned manager ID: 124

...

8 rows selected.

ORACLE

Alternative Quote (q) Operator

Many SQL statements use character literals in expressions or conditions. If the literal itself contains a single quotation mark, you can use the quote (q) operator and choose your own quotation mark delimiter.

You can choose any convenient delimiter, single-byte or multibyte, or any of the following character pairs: [ ], { }, ( ), or < >.

In the example shown, the string contains a single quotation mark, which is normally interpreted as a delimiter of a character string. By using the q operator, however, the brackets [] are used as the quotation mark delimiter. The string between the brackets delimiters is interpreted as a literal character string.

Duplicate Rows

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

```
SELECT  department_id
FROM    employees;
```

1

DEPARTMENT_ID
90
90
90

...

20 rows selected.

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

2

DEPARTMENT_ID
10
20
50

...

8 rows selected.

ORACLE

Duplicate Rows

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

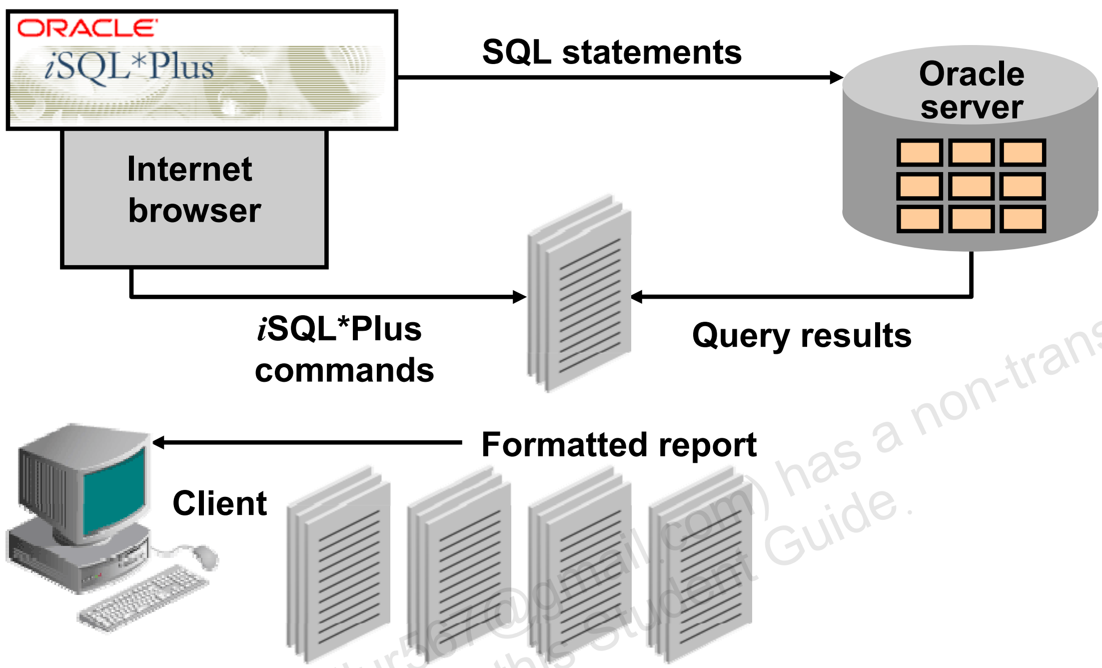
To eliminate duplicate rows in the result, include the DISTINCT keyword in the SELECT clause immediately after the SELECT keyword. In the second example in the slide, the EMPLOYEES table actually contains 20 rows, but there are only seven unique department numbers in the table.

You can specify multiple columns after the DISTINCT qualifier. The DISTINCT qualifier affects all the selected columns, and the result is every distinct combination of the columns.

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

Using the *i*SQL*Plus Environment

SQL and iSQL*Plus Interaction



SQL and *i*SQL*Plus

SQL is a command language for communication with the Oracle server from any tool or application. Oracle SQL contains many extensions.

*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
- Is an English-like language

Features of *i*SQL*Plus

- Is accessed from a browser
- Accepts SQL 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
- Keyword cannot be abbreviated
- Statements manipulate data and table definitions in the database

***i*SQL*Plus**

- An environment
- Oracle-proprietary
- 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

SQL statements

*i*SQL*Plus commands

ORACLE

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
Retrieves data; 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 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 table structures
- Enter, execute, and edit SQL statements
- Save or append SQL statements to files
- Execute or edit statements that are stored in saved script files

ORACLE

*i*SQL*Plus

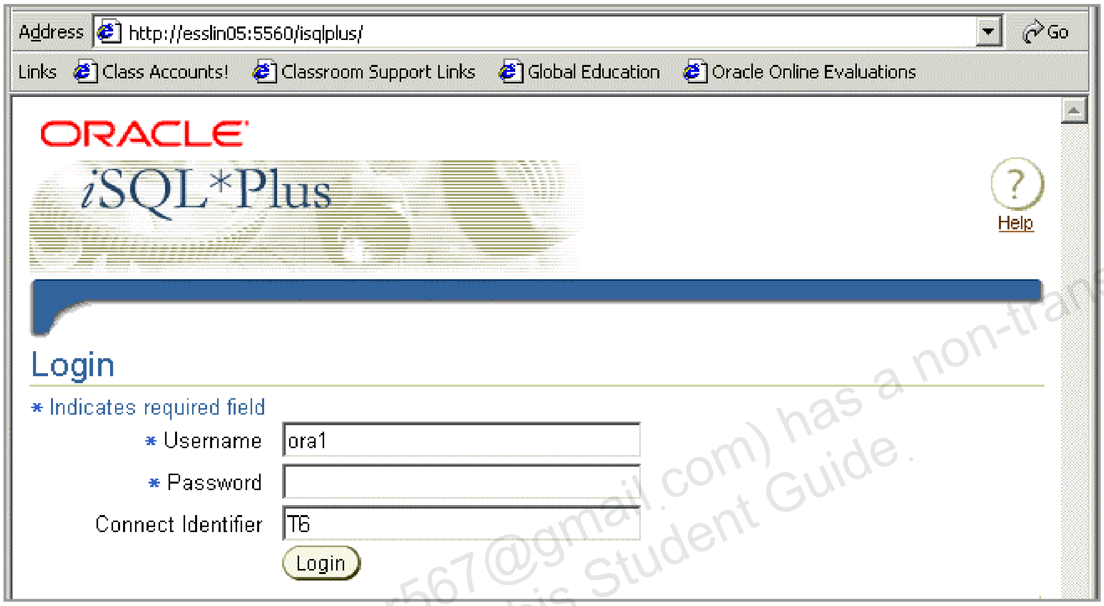
*i*SQL*Plus is an environment in which you can do the following:
- Execute SQL statements to retrieve, modify, add, and remove data from the database
- Format, perform calculations on, store, and print query results in the form of reports
- Create script files to store SQL statements for repeated use in the future

*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 in text script files and runs 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	Enables you 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 In to *i*SQL*Plus

From your browser environment:

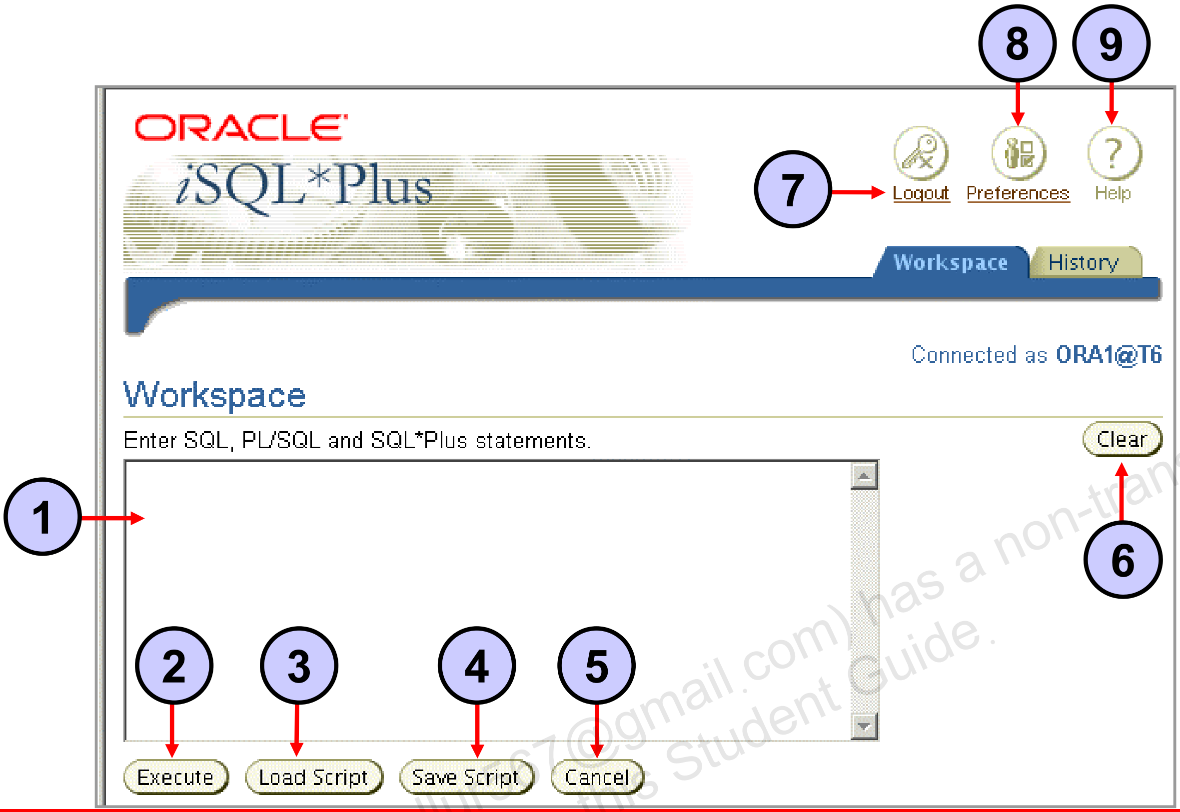


Logging In to *i*SQL*Plus

To log in from a browser environment:

1. Start the browser.
2. Enter the URL address of the *i*SQL*Plus environment.
3. On the Login page, enter appropriate values in the Username, Password, and Connect Identifier fields.

*i*SQL*Plus Environment



*i*SQL*Plus Environment

In the browser, the *i*SQL*Plus Workspace page has several key areas:

1. **Text box:** Area where you type the SQL statements and *i*SQL*Plus commands
2. **Execute button:** Click to execute the statements and commands in the text box
3. **Load Script button:** Brings up a form where you can identify a path and file name or a URL that contains SQL, PL/SQL, or SQL*Plus commands and load them into the text box
4. **Save Script button:** Saves the contents of the text box to a file
5. **Cancel button:** Stops the execution of the command in the text box
6. **Clear Screen button:** Click to clear text from the text box
7. **Logout icon:** Click to end the *i*SQL*Plus session and return to the *i*SQL*Plus Login page
8. **Preferences icon:** Click to change your interface configuration, system configuration, or password
9. **Help icon:** Provides access to *i*SQL*Plus help documentation

Displaying Table Structure

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

```
DESC[RIBE] tablename
```


Displaying the Table Structure

In *i*SQL*Plus, you can display the structure of a table by using the `DESCRIBE` command. The command displays the column names and data types, and it shows you whether a column *must* contain data (that is, whether the column has a `NOT NULL` constraint).

In the syntax, *`tablename`* is the name of any existing table, view, or synonym that is accessible to the user.

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

Displaying the Table Structure (continued)

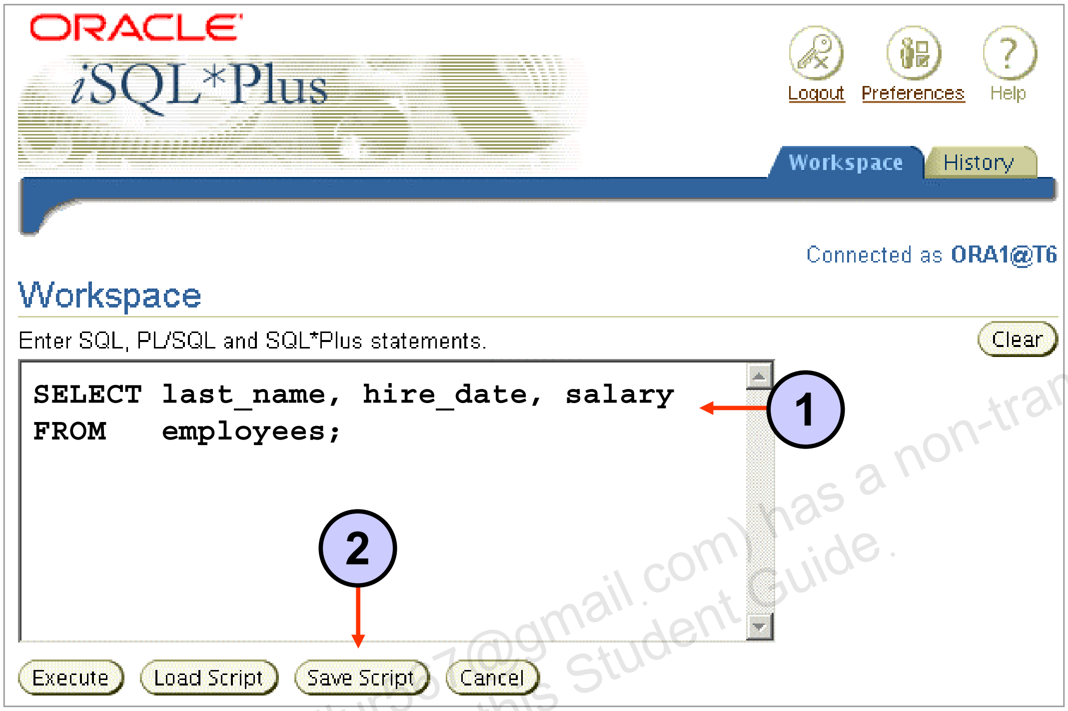
The example in the slide displays the information about the structure of the `EMPLOYEES` table.

In the resulting display, *Null?* indicates that the values for this column maybe unknown. `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, A.D 9999.
`CHAR(`*s*`)`	Fixed-length character value of size *s*

Interacting with Script Files



ORACLE

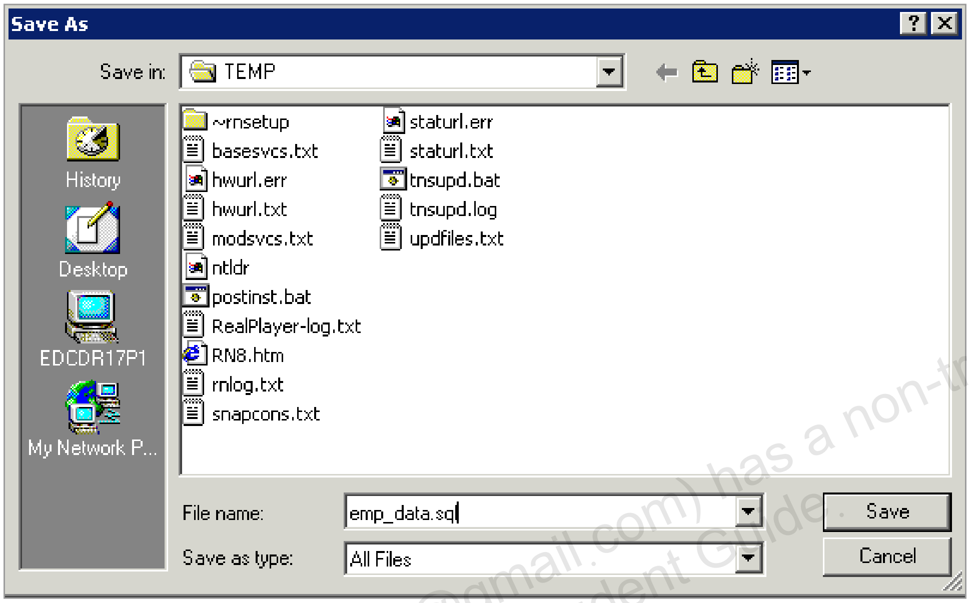
Interacting with Script Files

Placing Statements and Commands into a Text Script File

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

1. Type the SQL statements in the text box in *i*SQL*Plus.
2. Click the Save Script button. This opens the Windows File Save dialog box. Identify the name of the file. Note that the file extension defaults to `.sql`. You can change the file type to a text file or save it as a `.sql` file.

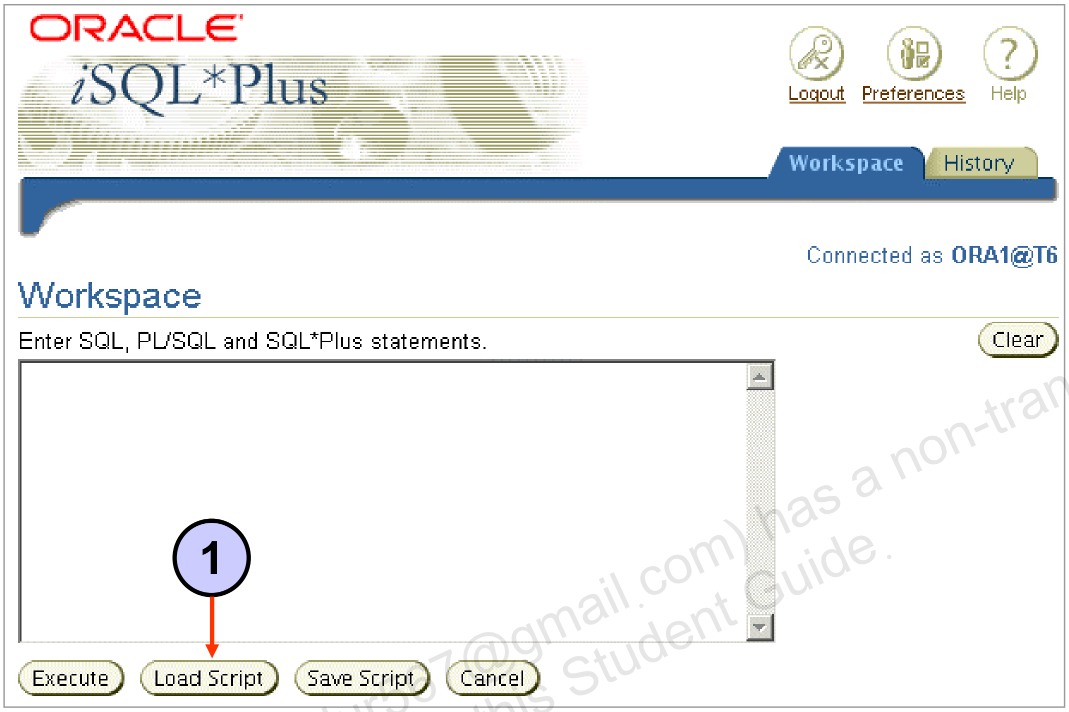
Interacting with Script Files



Interacting with Script Files (continued)

In the example shown, the SQL `SELECT` statement typed in the text box is saved to a file named `emp_data.sql`. You can choose the type of the file, name of the file, and location of where you want to save the script file.

Interacting with Script Files



ORACLE

Interacting with Script Files (continued)

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:

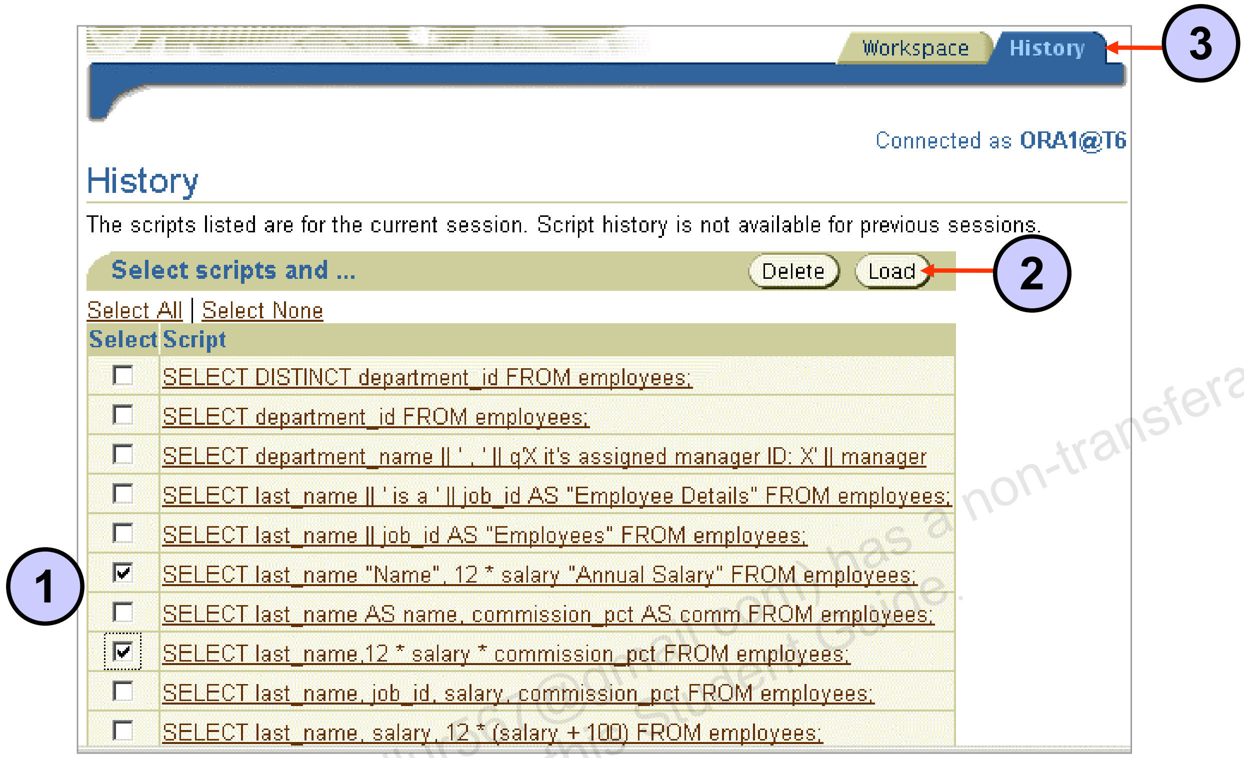
1. Click the Load Script button. This opens a form where you can enter the name of the file or a URL containing the SQL, PL/SQL, or SQL*Plus commands that you want to enter in the text box.

Interacting with Script Files

Interacting with Script Files (continued)

2. Enter the script name and path, or the URL location. Or you can click the Browse button to find the script name and location.
3. Click the Load button to bring the contents of the file or URL location into the text box.

*i*SQL*Plus History Page



Running Previous Statements

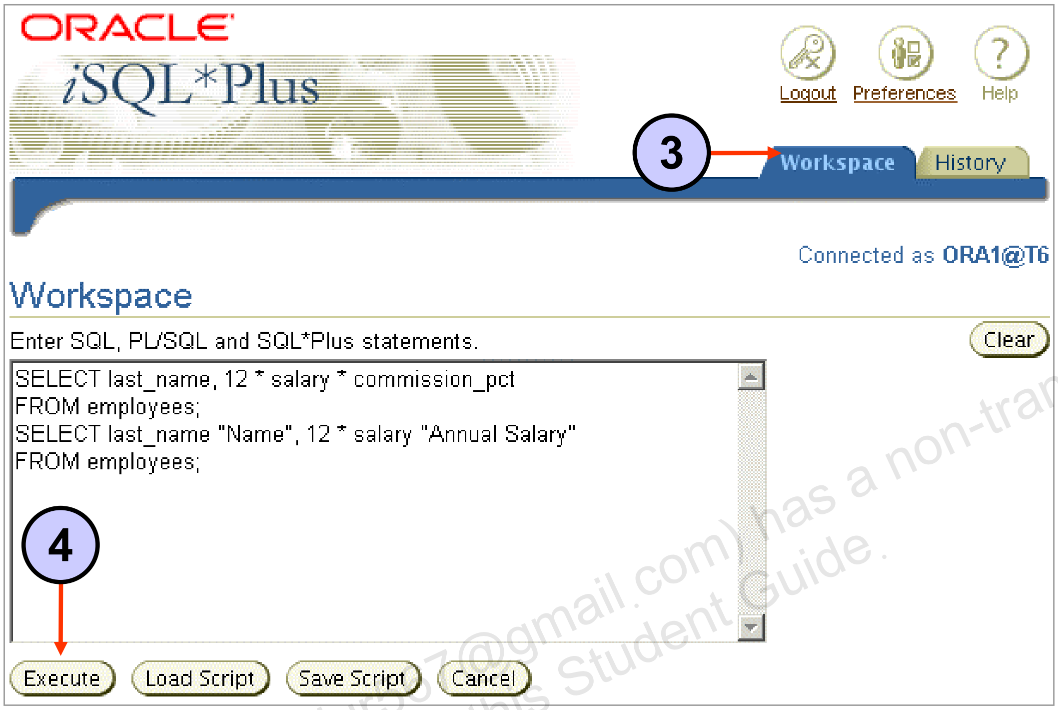
The History page in *i*SQL*Plus lets you execute previously run statements in your session. The History page shows your most recently run SQL statements and *i*SQL*Plus commands. To rerun the statements:

1. Select the statement that you want to execute.
2. Click the Load button.

Note

- You can control the number of statements that are shown on the History page with Preferences settings.
- You can choose to delete selected statements by clicking the Delete button.

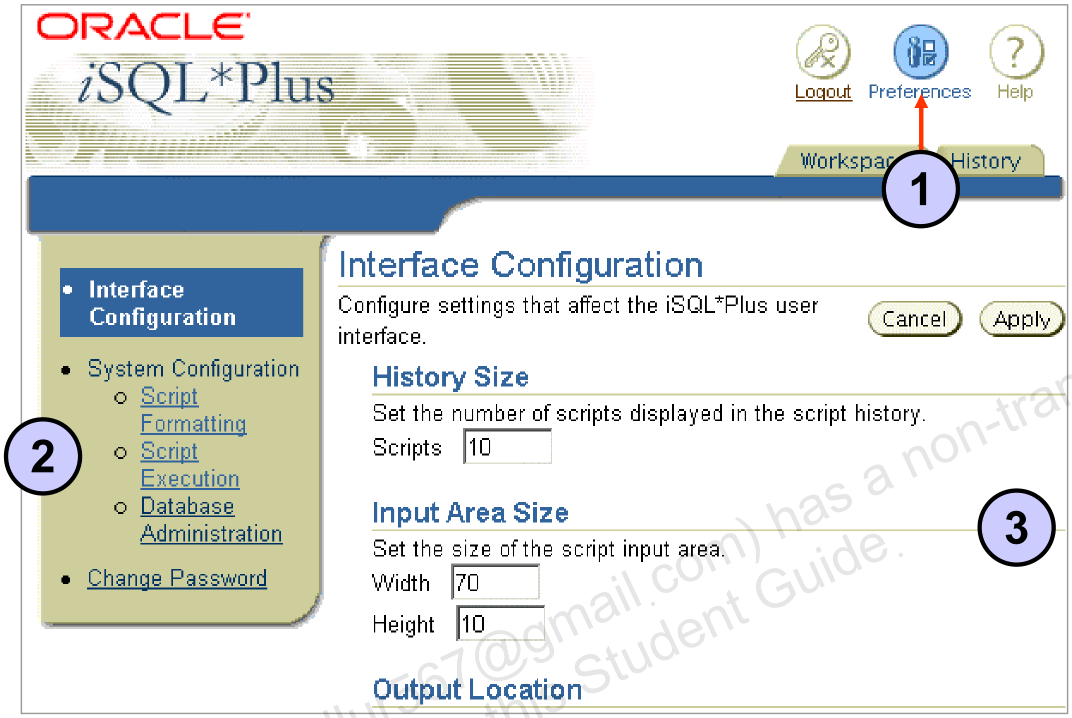
*i*SQL*Plus History Page



Running Previous Statements (continued)

3. Return to the Workspace page.
4. Click the Execute button to run the commands that have been loaded into the text box.

Setting *i*SQL*Plus Preferences

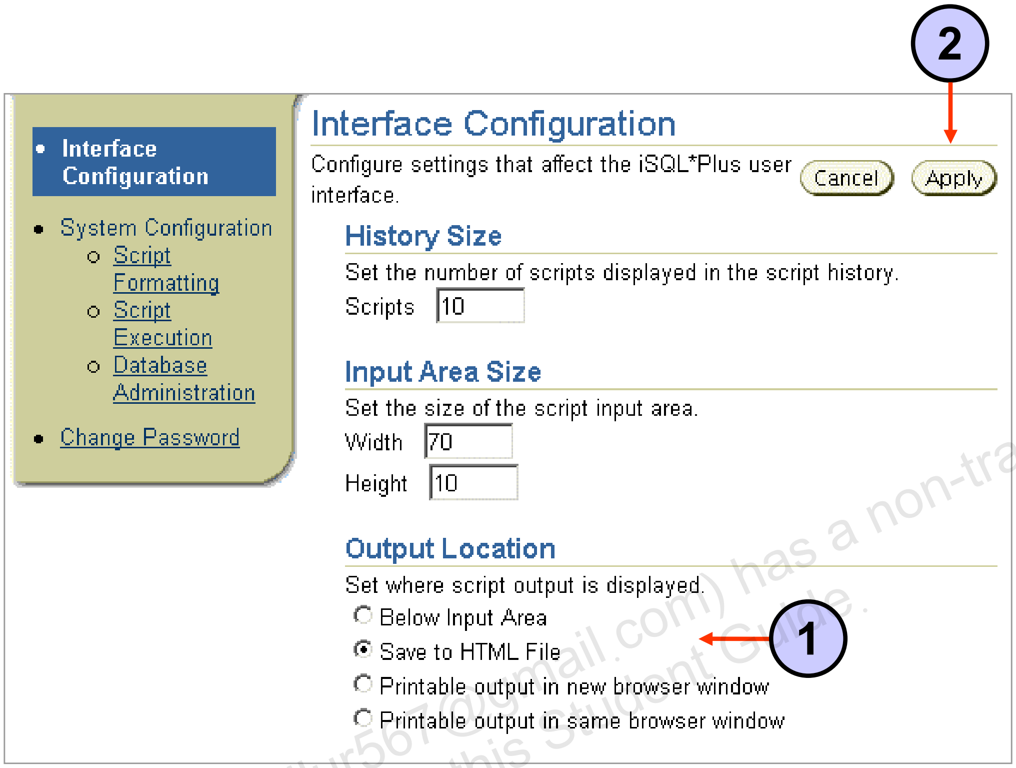


ORACLE

*i*SQL*Plus Preferences

1. You can set preferences for your *i*SQL*Plus session by clicking the Preferences icon.
2. The preferences are divided into categories. You can set preferences for script formatting, script execution, and database administration, and you can change your password.
3. When you choose a preference category, a form is displayed that lets you set the preferences for that category.

Setting the Output Location Preference



ORACLE

Changing the Output Location

You can send the results that are generated by a SQL statement or *i*SQL*Plus command to the screen (the default), a file, or another browser window.

On the Preferences page:

1. Select an Output Location option.
2. Click the Apply button.

Summary

In this lesson, you should have learned how to:

- Write a `SELECT` statement that:
  – Returns all rows and columns from a table
  – Returns specified columns from a table
  – Uses column aliases to display more descriptive column headings
- Use the *i*SQL*Plus environment to write, save, and execute SQL statements and *i*SQL*Plus commands

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


`SELECT` Statement

In this lesson, you should have learned how to retrieve data from a database table with the `SELECT` statement.

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

In the syntax:

`SELECT`	is a list of one or more columns
`*`	selects all columns
`DISTINCT`	suppresses duplicates
`column|expression`	selects the named column or the expression
`alias`	gives selected columns different headings
`FROM` *`table`*	specifies the table containing the columns

*i*SQL*Plus

*i*SQL*Plus is an execution environment that you can use to send SQL statements to the database server and to edit and save SQL statements. Statements can be executed 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 and specifying column names
- Using *i*SQL*Plus

Practice 1: Overview

This is the first of many practices in this course. The solutions (if you require them) can be found in Appendix A. Practices are intended to cover all topics that are presented in the corresponding lesson.

Note the following location for the lab files:

`E:\labs\SQL1\labs`

If you are asked to save any lab files, save them at this location.

To start *i*SQL*Plus, start your browser. You need to enter a URL to access *i*SQL*Plus. The URL requires the host name, which your instructor will provide. Enter the following command, replacing the host name with the value that your instructor provides:

```
http://<HOSTNAME:5560>/isqlplus
```

In any practice, there maybe exercises that are prefaced with the phrases "If you have time" or "If you want an extra challenge." Work on these exercises only if you have completed all other exercises in the allocated time and would like a further challenge to your skills.

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

Practice 1

Part 1

Test your knowledge:

1. Initiate an *i*SQL*Plus session using the user ID and password that are provided by the instructor.
2. *i*SQL*Plus commands access the database.

   True/False

3. The following `SELECT` statement executes successfully:

```
SELECT last_name, job_id, salary AS Sal
FROM   employees;
```

   True/False

4. The following `SELECT` statement executes successfully:

```
SELECT *
FROM   job_grades;
```

   True/False

5. There are four coding errors in the following statement. Can you identify them?

```
SELECT    employee_id, last_name
sal x 12  ANNUAL SALARY
FROM      employees;
```

Part 2

Note the following location for the lab files:

`E:\labs\SQL1\labs`

If you are asked to save any lab files, save them at this location.

*To start iSQL*Plus, start your browser. You need to enter a URL to access iSQL*Plus. The URL requires the host name, which your instructor will provide. Enter the following command, replacing the host name with the value that your instructor provides:*

`http://<HOSTNAME:5560>/isqlplus`

You have been hired as a SQL programmer for Acme Corporation. Your first task is to create some reports based on data from the Human Resources tables.

6. Your first task is to determine the structure of the `DEPARTMENTS` table and its contents.

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

Practice 1 (continued)

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.

7. You need to determine 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)

The HR department wants a query to display the last name, job code, hire date, and employee number for each employee, with employee number appearing first. Provide an alias `STARTDATE` for the `HIRE_DATE` column. Save your SQL statement to a file named `lab_01_07.sql` so that you can dispatch this file to the HR department.

Practice 1 (continued)

8. Test your query in the `lab_01_07.sql` file to ensure that it runs correctly.

EMPLOYEE_ID	LAST_NAME	JOB_ID	STARTDATE
100	King	AD_PRES	17-JUN-87
101	Kochhar	AD_VP	21-SEP-89
102	De Haan	AD_VP	13-JAN-93
103	Hunold	IT_PROG	03-JAN-90
104	Ernst	IT_PROG	21-MAY-91
107	Lorentz	IT_PROG	07-FEB-99
124	Mourgos	ST_MAN	16-NOV-99
141	Rajs	ST_CLERK	17-OCT-95
142	Davies	ST_CLERK	29-JAN-97
143	Matos	ST_CLERK	15-MAR-98
144	Vargas	ST_CLERK	09-JUL-98
149	Zlotkey	SA_MAN	29-JAN-00
174	Abel	SA_REP	11-MAY-96
176	Taylor	SA_REP	24-MAR-98
...			
206	Gietz	AC_ACCOUNT	07-JUN-94

20 rows selected.

9. The HR department needs a query to display all unique job codes from the `EMPLOYEES` table.

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)

Part 3

If you have time, complete the following exercises:

10. The HR department wants more descriptive column headings for its report on employees. Copy the statement from `lab_01_07.sql` to the *i*SQL*Plus text box. Name the column headings `Emp #`, `Employee`, `Job`, and `Hire Date`, respectively. Then run your query again.

Emp #	Employee	Job	Hire Date
100	King	AD_PRES	17-JUN-87
101	Kochhar	AD_VP	21-SEP-89
102	De Haan	AD_VP	13-JAN-93
103	Hunold	IT_PROG	03-JAN-90
104	Ernst	IT_PROG	21-MAY-91
107	Lorentz	IT_PROG	07-FEB-99
124	Mourgos	ST_MAN	16-NOV-99
141	Rajs	ST_CLERK	17-OCT-95
142	Davies	ST_CLERK	29-JAN-97
143	Matos	ST_CLERK	15-MAR-98
144	Vargas	ST_CLERK	09-JUL-98
...			
206	Gietz	AC_ACCOUNT	07-JUN-94

20 rows selected.

11. The HR department has requested a report of all employees and their job IDs. Display the last name concatenated with the job ID (separated by a comma and space) and name the column `Employee and Title`.

Employee and Title
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
...
Gietz, AC_ACCOUNT

20 rows selected.

Practice 1 (continued)

If you want an extra challenge, complete the following exercise:

12. To familiarize yourself with the data in the `EMPLOYEES` table, create a query to display all the data from that table. Separate each column output by a comma. Name the column title `THE_OUTPUT`.

THE_OUTPUT
100,Steven,King,SKING,515.123.4567,AD_PRES,,17-JUN-87,24000,,90
101,Neena,Kochhar,NKOCHHAR,515.123.4568,AD_VP,100,21-SEP-89,17000,,90
102,Lex,De Haan,LDEHAAN,515.123.4569,AD_VP,100,13-JAN-93,17000,,90
103,Alexander,Hunold,AHUNOLD,590.423.4567,IT_PROG,102,03-JAN-90,9000,,60
104,Bruce,Ernst,BERNST,590.423.4568,IT_PROG,103,21-MAY-91,6000,,60
107,Diana,Lorentz,DLORENTZ,590.423.5567,IT_PROG,103,07-FEB-99,4200,,60
124,Kevin,Mourgos,KMOURGOS,650.123.5234,ST_MAN,100,16-NOV-99,5800,,50
141,Trenna,Rajs,TRAJS,650.121.8009,ST_CLERK,124,17-OCT-95,3500,,50
142,Curtis,Davies,CDAVIES,650.121.2994,ST_CLERK,124,29-JAN-97,3100,,50
143,Randall,Matos,RMATOS,650.121.2874,ST_CLERK,124,15-MAR-98,2600,,50
144,Peter,Vargas,PVARGAS,650.121.2004,ST_CLERK,124,09-JUL-98,2500,,50
...
206,William,Gietz,WGIETZ,515.123.8181,AC_ACCOUNT,205,07-JUN-94,8300,,110

20 rows selected.

Restricting and Sorting Data

Objectives

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

- Limit the rows that are retrieved by a query
- Sort the rows that are retrieved by a query
- Use ampersand substitution in *i*SQL*Plus to restrict and sort output at run time

Objectives

When retrieving data from the database, you may need to do the following:

- Restrict the rows of data that are displayed
- Specify the order in which the rows are displayed

This lesson explains the SQL statements that you use to perform these actions.

Limiting Rows Using a Selection

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
104	Ernst	IT_PROG	60
107	Lorentz	IT_PROG	60
124	Mourgos	ST_MAN	50

...

20 rows selected.

"retrieve all employees in department 90"

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90

ORACLE

Limiting Rows Using a Selection

In the example in the slide, assume that you want to display all the employees in department 90. The rows with a value of 90 in the `DEPARTMENT_ID` column are the only ones that are returned. This method of restriction is the basis of the `WHERE` clause in SQL.

Limiting the Rows That Are Selected

- Restrict the rows that are returned by using the `WHERE` clause:

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

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

ORACLE

Limiting the Rows That Are Selected

You can restrict the rows that are returned from the query by using the `WHERE` clause. A `WHERE` clause contains a condition that must be met, and it directly follows the `FROM` clause. If the condition is true, the row meeting the condition is returned.

In the syntax:

`WHERE`	restricts the query to rows that meet a condition
`condition`	is composed of column names, expressions, constants, and a comparison operator

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

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

Using the WHERE Clause

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

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90

Using the `WHERE` Clause

In the example, the `SELECT` statement retrieves the employee ID, name, job ID, and department number of all employees who are in department 90.

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.

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


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 searches are case sensitive. In the following example, no rows are returned because the `EMPLOYEES` table stores all the last names in mixed case:

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

Oracle databases store 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: For details about the RR format and about changing the default date format, see the lesson titled "Using Single-Row Functions to Customize Output."

Comparison Conditions

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

Comparison Conditions

Comparison conditions are used in conditions that compare one expression to another value or expression. They are used in the `WHERE` clause in the following format:

Syntax

```
... WHERE expr operator value
```

Example

```
... WHERE hire_date = '01-JAN-95'
... WHERE salary >= 6000
... WHERE last_name = 'Smith'
```

An alias cannot be used in the `WHERE` clause.

Note: The symbols `!=` and `^=` can also represent the *not equal to* condition.

Using Comparison Conditions

```
SELECT last_name, salary
FROM   employees
WHERE  salary <= 3000 ;
```

LAST_NAME	SALARY
Matos	2600
Vargas	2500

Using Comparison Conditions

In the example, the SELECT statement retrieves the last name and salary from the EMPLOYEES table for any employee whose salary is less than or equal to $3,000. Note that there is an explicit value supplied to the WHERE clause. The explicit value of 3000 is compared to the salary value in the SALARY column of the EMPLOYEES table.

Using the BETWEEN Condition

Use the BETWEEN condition to display rows based on a range of values:

```
SELECT last_name, salary
FROM   employees
WHERE  salary BETWEEN 2500 AND 3500 ;
```

Lower limit   Upper limit

LAST_NAME	SALARY
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500

ORACLE

Using the BETWEEN Condition

You can display rows based on a range of values using the BETWEEN range condition. 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 $2,500 and $3,500.

Values that are specified with the BETWEEN condition are inclusive. You must specify the lower limit first.

You can also use the BETWEEN condition on character values:

```
SELECT last_name
FROM   employees
WHERE  last_name BETWEEN 'King' AND 'Smith';
```

Using the `IN` Condition

Use the `IN` membership condition to test for values in a list:

```
SELECT employee_id, last_name, salary, manager_id
FROM   employees
WHERE  manager_id IN (100, 101, 201) ;
```

EMPLOYEE_ID	LAST_NAME	SALARY	MANAGER_ID
202	Fay	6000	201
200	Whalen	4400	101
205	Higgins	12000	101
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` Condition

To test for values in a specified set of values, use the `IN` condition. The `IN` condition is also known as the *membership condition*.

The slide example displays employee numbers, last names, salaries, and manager's employee numbers for all the employees whose manager's employee number is 100, 101, or 201.

The `IN` condition can be used with any data type. The following example returns a row from the `EMPLOYEES` table for any employee whose last name is included in the list of names in the `WHERE` clause:

```
SELECT employee_id, manager_id, department_id
FROM   employees
WHERE  last_name IN ('Hartstein', 'Vargas');
```

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

Using the LIKE Condition

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

```
SELECT   first_name
FROM     employees
WHERE    first_name LIKE 'S%' ;
```


Using the LIKE Condition

You may not always know the exact value to search for. You can select rows that match a character pattern by using the LIKE condition. The character pattern–matching operation is referred to as a *wildcard* search. Two symbols can be used to construct the search string.

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

The SELECT statement in the slide returns the employee first name from the EMPLOYEES table for any employee whose first name begins with the letter *S*. Note the uppercase *S*. Names beginning with an *s* are not returned.

The LIKE condition can be used as a shortcut for some BETWEEN comparisons. The following example displays the last names and hire dates of all employees who joined between January 1995 and December 1995:

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date LIKE '%95';
```

Using the LIKE Condition

- You can combine pattern-matching characters:

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

LAST_NAME
Kochhar
Lorentz
Mourgos

- You can use the ESCAPE identifier to search for the actual % and _ symbols.

Combining Wildcard Characters

The % and _ symbols can be used in any combination with literal characters. The example in the slide displays the names of all employees whose last names have the letter *o* as the second character.

ESCAPE Option

When you need to have an exact match for the actual % and _ characters, use the ESCAPE option. This option specifies what the escape character is. If you want to search for strings that contain SA_, you can use the following SQL statement:

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

EMPLOYEE_ID	LAST_NAME	JOB_ID
149	Zlotkey	SA_MAN
174	Abel	SA_REP
176	Taylor	SA_REP
178	Grant	SA_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.

Using the NULL Conditions

Test for nulls with the `IS NULL` operator.

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

LAST_NAME	MANAGER_ID
King	

Using the NULL Conditions

The `NULL` conditions include the `IS NULL` condition and the `IS NOT NULL` condition.

The `IS NULL` condition tests for nulls. A null value means the value is unavailable, unassigned, unknown, or inapplicable. Therefore, you cannot test with = because a null cannot be equal or unequal to any value. The slide example retrieves the last names and managers of all employees who do not have a manager.

Here is another example: To display last name, job ID, and commission for all employees who are *not* entitled to receive a commission, use the following SQL 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	
...		
Higgins	AC_MGR	
Gietz	AC_ACCOUNT	

16 rows selected.

Logical Conditions

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

Logical Conditions

A logical condition combines the result of two component conditions to produce a single result based on those conditions, or it inverts the result of a single condition. A row is returned only if the overall result of the condition is true.

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 several conditions in one WHERE clause using the AND and OR operators.

Using the AND Operator

AND requires both conditions to be true:

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >=10000
AND    job_id LIKE '%MAN%' ;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
149	Zlotkey	SA_MAN	10500
201	Hartstein	MK_MAN	13000

Using the AND Operator

In the example, both conditions must be true for any record to be selected. Therefore, only employees who have a job title that contains the string 'MAN' *and* earn $10,000 or more are selected.

All character searches are case sensitive. No rows are returned if 'MAN' is not uppercase. Character strings must be enclosed in quotation marks.

AND Truth Table

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

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Using the OR Operator

OR requires either condition to be true:

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary >= 10000
OR     job_id LIKE '%MAN%' ;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
100	King	AD_PRES	24000
101	Kochhar	AD_VP	17000
102	De Haan	AD_VP	17000
124	Mourgos	ST_MAN	5800
149	Zlotkey	SA_MAN	10500
174	Abel	SA_REP	11000
201	Hartstein	MK_MAN	13000
205	Higgins	AC_MGR	12000

8 rows selected.

Using the OR Operator

In the example, either condition can be true for any record to be selected. Therefore, any employee who has a job ID that contains the string 'MAN' *or* earns $10,000 or more is selected.

OR Truth Table

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

OR	**TRUE**	**FALSE**	**NULL**
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Using the NOT Operator

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

LAST_NAME	JOB_ID
King	AD_PRES
Kochhar	AD_VP
De Haan	AD_VP
Mourgos	ST_MAN
Zlotkey	SA_MAN
Whalen	AD_ASST
Hartstein	MK_MAN
Fay	MK_REP
Higgins	AC_MGR
Gietz	AC_ACCOUNT

10 rows selected.

ORACLE

Using the NOT Operator

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

NOT Truth Table

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

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

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

```
... WHERE  job_id     NOT  IN ('AC_ACCOUNT', 'AD_VP')
... WHERE  salary     NOT  BETWEEN  10000 AND  15000
... WHERE  last_name NOT  LIKE '%A%'
... WHERE  commission_pct  IS   NOT  NULL
```

Rules of Precedence

Operator	Meaning
1	Arithmetic operators
2	Concatenation operator
3	Comparison conditions
4	`IS [NOT] NULL`, `LIKE`, `[NOT] IN`
5	`[NOT] BETWEEN`
6	Not equal to
7	`NOT` logical condition
8	`AND` logical condition
9	`OR` logical condition

You can use parentheses to override rules of precedence.

Rules of Precedence

The rules of precedence determine the order in which expressions are evaluated and calculated. The table lists the default order of precedence. You can override the default order by using parentheses around the expressions that you want to calculate first.

Rules of Precedence

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id = 'SA_REP'
OR     job_id = 'AD_PRES'
AND    salary > 15000;
```

1

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000
Abel	SA_REP	11000
Taylor	SA_REP	8600
Grant	SA_REP	7000

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  (job_id = 'SA_REP'
OR     job_id = 'AD_PRES')
AND    salary > 15000;
```

2

LAST_NAME	JOB_ID	SALARY
King	AD_PRES	24000

ORACLE

Rules of Precedence (continued)

1. Example of the Precedence of the AND Operator

In this example, there are two conditions:

- The first condition is that the job ID is AD_PRES *and* the salary is greater than $15,000.
- The second condition is that the job ID is SA_REP.

Therefore, the SELECT statement reads as follows:

"Select the row if an employee is a president *and* earns more than $15,000, *or* if the employee is a sales representative."

2. Example of Using Parentheses

In this example, there are two conditions:

- The first condition is that the job ID is AD_PRES *or* SA_REP.
- The second condition is that salary is greater than $15,000.

Therefore, the SELECT statement reads as follows:

"Select the row if an employee is a president *or* a sales representative, *and* if the employee earns more than $15,000."

Using the ORDER BY Clause

- Sort retrieved 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, hire_date
FROM     employees
ORDER BY hire_date ;
```

LAST_NAME	JOB_ID	DEPARTMENT_ID	HIRE_DATE
King	AD_PRES	90	17-JUN-87
Whalen	AD_ASST	10	17-SEP-87
Kochhar	AD_VP	90	21-SEP-89
Hunold	IT_PROG	60	03-JAN-90
Ernst	IT_PROG	60	21-MAY-91

...

20 rows selected.

Using the ORDER BY Clause

The order of rows that are returned in a query result is undefined. The ORDER BY clause can be used to sort the rows. If you use the ORDER BY clause, it must be the last clause of the SQL statement. You can specify an expression, an alias, or a column position as the sort condition.

Syntax

```
SELECT          expr
FROM            table
[WHERE          condition(s)]
[ORDER BY  {column, expr, numeric_position} [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

If the ORDER BY clause is not used, the sort order is undefined, and the Oracle server may not fetch rows in the same order for the same query twice. Use the ORDER BY clause to display the rows in a specific order.

Sorting

- Sorting in descending order:

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

1

- Sorting by column alias:

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

2

- Sorting by multiple columns:

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

3

ORACLE

Default Ordering of Data

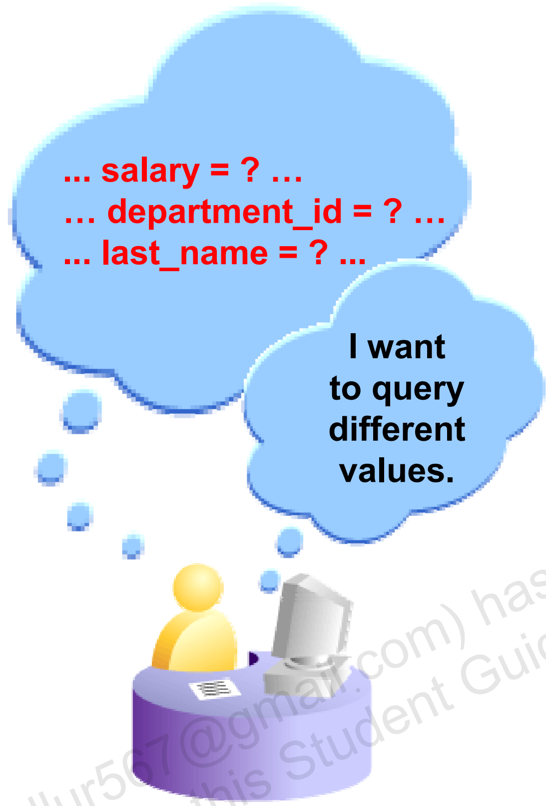
The default sort order is ascending:

- 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-92 before 01-JAN-95).
- 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.
- You can sort by a column that is not in the `SELECT` list.

Examples

1. To reverse the order in which rows are displayed, specify the `DESC` keyword after the column name in the `ORDER BY` clause. The slide example sorts the result by the most recently hired employee.
2. You can use a column alias in the `ORDER BY` clause. The slide example sorts the data by annual salary.
3. You can sort query results by more than one column. The sort limit is the number of columns in the given table. In the `ORDER BY` clause, specify the columns and separate the column names using commas. If you want to reverse the order of a column, specify `DESC` after its name.

Substitution Variables



Substitution Variables

The examples so far have been hard-coded. In a finished application, the user would trigger the report, and the report would run without further prompting. The range of data would be predetermined 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 by using substitution variables. You can embed *substitution variables* in a command file or in a single SQL statement. A variable can be thought of as a container in which the values are temporarily stored. When the statement is run, the value is substituted.

Substitution Variables

- Use *i*SQL*Plus substitution variables to:
  - Temporarily store values with single-ampersand (`&`) and double-ampersand (`&&`) substitution
- Use substitution variables to supplement the following:
  - `WHERE` conditions
  - `ORDER BY` clauses
  - Column expressions
  - Table names
  - Entire `SELECT` statements

Substitution Variables (continued)

In *i*SQL*Plus, you can use single-ampersand (`&`) substitution variables to temporarily store values.

You can predefine variables in *i*SQL*Plus by using the `DEFINE` command. `DEFINE` creates and assigns a value to a variable.

Examples of Restricted Ranges of Data

- Reporting figures only for the current quarter or specified date range
- Reporting on data relevant only to the user requesting the report
- Displaying personnel only within 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, such as:

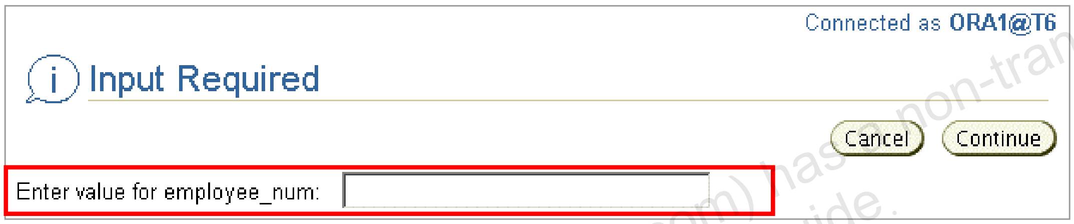
- Obtaining input values from a file rather than from a person
- Passing values from one SQL statement to another

*i*SQL*Plus does not support validation checks (except for data type) on user input.

Using the & Substitution Variable

Use a variable prefixed with an ampersand (&) to prompt the user for a value:

```
SELECT  employee_id, last_name, salary, department_id
FROM    employees
WHERE   employee_id = &employee_num ;
```



Single-Ampersand Substitution Variable

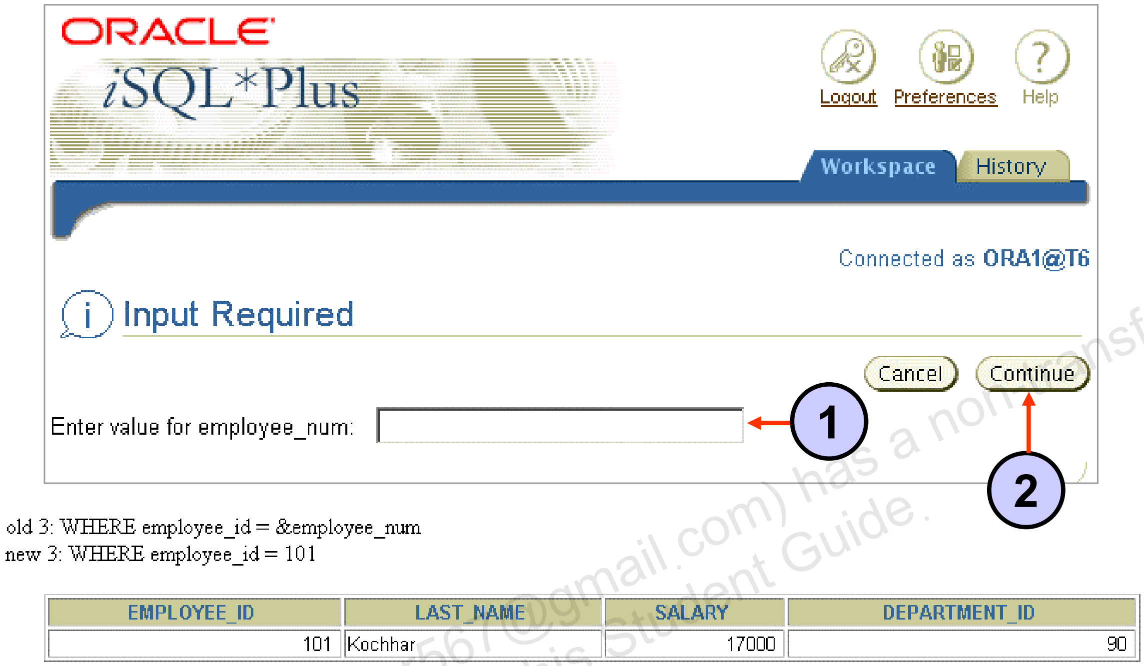
When running a report, users often want to restrict the data that is returned dynamically. *i*SQL*Plus provides this flexibility with user variables. Use an ampersand (&) to identify each variable in your SQL statement. You do not need to define the value of each variable.

Notation	Description
&user_variable	Indicates a variable in a SQL statement; if the variable does not exist, *i*SQL*Plus prompts the user for a value (*i*SQL*Plus discards a new variable after it is used.)

The example in the slide creates an *i*SQL*Plus substitution variable for an employee number. When the statement is executed, *i*SQL*Plus prompts the user for an employee number and then displays the employee number, last name, salary, and department number for that employee.

With the single ampersand, the user is prompted every time the command is executed, if the variable does not exist.

Using the & Substitution Variable



Single-Ampersand Substitution Variable (continued)

When *i*SQL*Plus detects that the SQL statement contains an ampersand, you are prompted to enter a value for the substitution variable that is named in the SQL statement.

After you enter a value and click the Continue button, the results are displayed in the output area of your *i*SQL*Plus session.

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' ;
```

Input Required

Cancel Continue

Enter value for job_title: IT_PROG

LAST_NAME	DEPARTMENT_ID	SALARY*12
Hunold	60	108000
Ernst	60	72000
Lorentz	60	50400

ORACLE

Specifying Character and Date Values with Substitution Variables

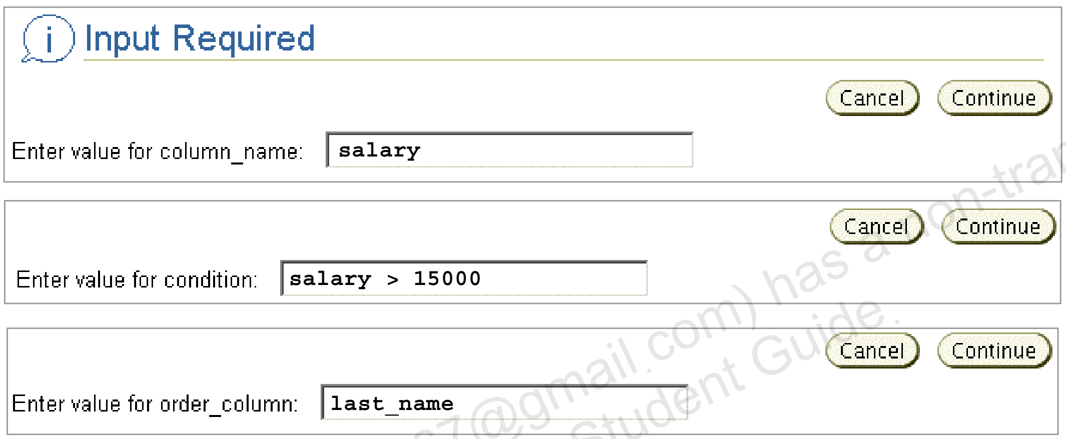
In a `WHERE` clause, date and character values must be enclosed in single quotation marks. The same rule applies to the substitution variables.

Enclose the variable in single quotation marks within the SQL statement itself.

The slide shows a query to retrieve the employee names, department numbers, and annual salaries of all employees based on the job title value of the *i*SQL*Plus substitution variable.

Specifying Column Names, Expressions, and Text

```
SELECT   employee_id, last_name, job_id,&column_name
FROM     employees
WHERE    &condition
ORDER BY &order_column ;
```



ORACLE

Specifying Column Names, Expressions, and Text

Not only can you use the substitution variables in the WHERE clause of a SQL statement, but these variables can also be used to substitute for column names, expressions, or text.

Example

The slide example displays the employee number, name, job title, and any other column that is specified by the user at run time, from the EMPLOYEES table. For each substitution variable in the SELECT statement, you are prompted to enter a value, and you then click the Continue button to proceed.

If you do not enter a value for the substitution variable, you get an error when you execute the preceding statement.

Note: A substitution variable can be used anywhere in the SELECT statement, except as the first word entered at the command prompt.

Using the `&&` Substitution Variable

Use the double ampersand (`&&`) if you want to reuse the variable value without prompting the user each time:

```
SELECT   employee_id, last_name, job_id, &&column_name
FROM     employees
ORDER BY &column_name ;
```

Input Required

Cancel Continue

Enter value for column_name: department_id

EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
200	Whalen	AD_ASST	10
201	Hartstein	MK_MAN	20

...

20 rows selected.

ORACLE

Double-Ampersand Substitution Variable

You can use the double-ampersand (`&&`) substitution variable if you want to reuse the variable value without prompting the user each time. The user sees the prompt for the value only once. In the example in the slide, the user is asked to give the value for variable `column_name` only once. The value that is supplied by the user (`department_id`) is used for both display and ordering of data.

*i*SQL*Plus stores the value that is supplied by using the `DEFINE` command; it uses it again whenever you reference the variable name. After a user variable is in place, you need to use the `UNDEFINE` command to delete it as follows:

```
UNDEFINE column_name
```

Using the *i*SQL*Plus `DEFINE` Command

- Use the *i*SQL*Plus `DEFINE` command to create and assign a value to a variable.
- Use the *i*SQL*Plus `UNDEFINE` command to remove a variable.

```
DEFINE employee_num = 200

SELECT employee_id, last_name, salary, department_id
FROM   employees
WHERE  employee_id = &employee_num ;

UNDEFINE employee_num
```


ORACLE

Using the *i*SQL*Plus `DEFINE` Command

The example shown creates an *i*SQL*Plus substitution variable for an employee number by using the `DEFINE` command. At run time, this displays the employee number, name, salary, and department number for that employee.

Because the variable is created using the *i*SQL*Plus `DEFINE` command, the user is not prompted to enter a value for the employee number. Instead, the defined variable value is automatically substituted in the `SELECT` statement.

The `EMPLOYEE_NUM` substitution variable is present in the session until the user undefines it or exits the *i*SQL*Plus session.

Using the VERIFY Command

Use the VERIFY command to toggle the display of the substitution variable, both before and after *i*SQL*Plus replaces substitution variables with values:

```
SET VERIFY ON
SELECT employee_id, last_name, salary, department_id
FROM   employees
WHERE  employee_id = &employee_num;
```

"employee_num" 200

```
old   3: WHERE  employee_id = &employee_num
new   3: WHERE  employee_id = 200
```


Using the VERIFY Command

To confirm the changes in the SQL statement, use the *i*SQL*Plus VERIFY command. Setting SET VERIFY 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 displays the old as well as the new value of the EMPLOYEE_ID column.

*i*SQL*Plus System Variables

*i*SQL*Plus uses various system variables that control the working environment. One of those variables is VERIFY. To obtain a complete list of all system variables, you can issue the SHOW ALL command.

Summary

In this lesson, you should have learned how to:

- Use the `WHERE` clause to restrict rows of output:
  - Use the comparison conditions
  - Use the `BETWEEN`, `IN`, `LIKE`, and `NULL` conditions
  - Apply the logical `AND`, `OR`, and `NOT` operators
- Use the `ORDER BY` clause to sort rows of output:

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

- Use ampersand substitution in *i*SQL*Plus to restrict and sort output at run time

ORACLE

Summary

In this lesson, you should have learned about restricting and sorting rows that are returned by the `SELECT` statement. You should also have learned how to implement various operators and conditions.

By using the *i*SQL*Plus substitution variables, you can add flexibility to your SQL statements. You can query users at run time and enable them to specify criteria.

Practice 2: Overview

This practice covers the following topics:

- Selecting data and changing the order of the rows that are displayed
- Restricting rows by using the `WHERE` clause
- Sorting rows by using the `ORDER BY` clause
- Using substitution variables to add flexibility to your SQL `SELECT` statements

Practice 2: Overview

In this practice, you build more reports, including statements that use the `WHERE` clause and the `ORDER BY` clause. You make the SQL statements more reusable and generic by including ampersand substitution.

Practice 2

The HR department needs your assistance with creating some queries.

1. Because of budget issues, the HR department needs a report that displays the last name and salary of employees who earn more than $12,000. Place your SQL statement in a text file named `lab_02_01.sql`. Run your query.

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hartstein	13000

2. Create a report that displays the last name and department number for employee number 176.

LAST_NAME	DEPARTMENT_ID
Taylor	80

3. The HR department needs to find high-salary and low-salary employees. Modify `lab_02_01.sql` to display the last name and salary for any employee whose salary is not in the range of $5,000 to $12,000. Place your SQL statement in a text file named `lab_02_03.sql`.

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Lorentz	4200
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500
Whalen	4400
Hartstein	13000

10 rows selected.

Practice 2 (continued)

4. Create a report to display the last name, job ID, and start date for the employees with the last names of Matos and Taylor. Order the query in ascending order by start date.

LAST_NAME	JOB_ID	HIRE_DATE
Matos	ST_CLERK	15-MAR-98
Taylor	SA_REP	24-MAR-98

5. Display the last name and department number of all employees in departments 20 or 50 in ascending alphabetical order by name.

LAST_NAME	DEPARTMENT_ID
Davies	50
Fay	20
Hartstein	20
Matos	50
Mourgos	50
Rajs	50
Vargas	50

7 rows selected.

6. Modify `lab_02_03.sql` to display the last name and salary of employees who earn between $5,000 and $12,000 and are in department 20 or 50. Label the columns `Employee` and `Monthly Salary`, respectively. Resave `lab_02_03.sql` as `lab_02_06.sql`. Run the statement in `lab_02_06.sql`.

Employee	Monthly Salary
Fay	6000
Mourgos	5800

Practice 2 (continued)

7. The HR department needs a report that displays the last name and hire date for all employees who were hired in 1994.

LAST_NAME	HIRE_DATE
Higgins	07-JUN-94
Gietz	07-JUN-94

8. Create a report to display the last name and job title of all employees who do not have a manager.

LAST_NAME	JOB_ID
King	AD_PRES

9. Create a report to display the last name, salary, and commission of all employees who earn commissions. Sort data in descending order of salary and commissions.

LAST_NAME	SALARY	COMMISSION_PCT
Abel	11000	.3
Zlotkey	10500	.2
Taylor	8600	.2
Grant	7000	.15

10. Members of the HR department want to have more flexibility with the queries that you are writing. They would like a report that displays the last name and salary of employees who earn more than an amount that the user specifies after a prompt. (You can use the query that you created in practice exercise 1 and modify it.) Save this query to a file named `lab_02_10.sql`. If you enter `12000` when prompted, the report displays the following results:

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hartstein	13000

Practice 2 (continued)

11. The HR department wants to run reports based on a manager. Create a query that prompts the user for a manager ID and generates the employee ID, last name, salary, and department for that manager's employees. The HR department wants the ability to sort the report on a selected column. You can test the data with the following values:

manager ID = 103, sorted by employee last name:

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
104	Ernst	6000	60
107	Lorentz	4200	60

manager ID = 201, sorted by salary:

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
202	Fay	6000	20

manager ID = 124, sorted by employee ID:

EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
141	Rajs	3500	50
142	Davies	3100	50
143	Matos	2600	50
144	Vargas	2500	50

Practice 2 (continued)

If you have time, complete the following exercises:

12. Display all employee last names in which the third letter of the name is *a*.

LAST_NAME
Grant
Whalen

13. Display the last names of all employees who have both an *a* and an *e* in their last name.

LAST_NAME
Davies
De Haan
Hartstein
Whalen

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

14. Display the last name, job, and salary for all employees whose jobs are either sales representative or stock clerk and whose salaries are not equal to $2,500, $3,500, or $7,000.

LAST_NAME	JOB_ID	SALARY
Abel	SA_REP	11000
Taylor	SA_REP	8600
Davies	ST_CLERK	3100
Matos	ST_CLERK	2600

15. Modify `lab_02_06.sql` to display the last name, salary, and commission for all employees whose commission amount is 20%. Resave `lab_02_06.sql` as `lab_02_15.sql`. Rerun the statement in `lab_02_15.sql`.

Employee	Monthly Salary	COMMISSION_PCT
Zlotkey	10500	.2
Taylor	8600	.2

SQL Functions

Objectives

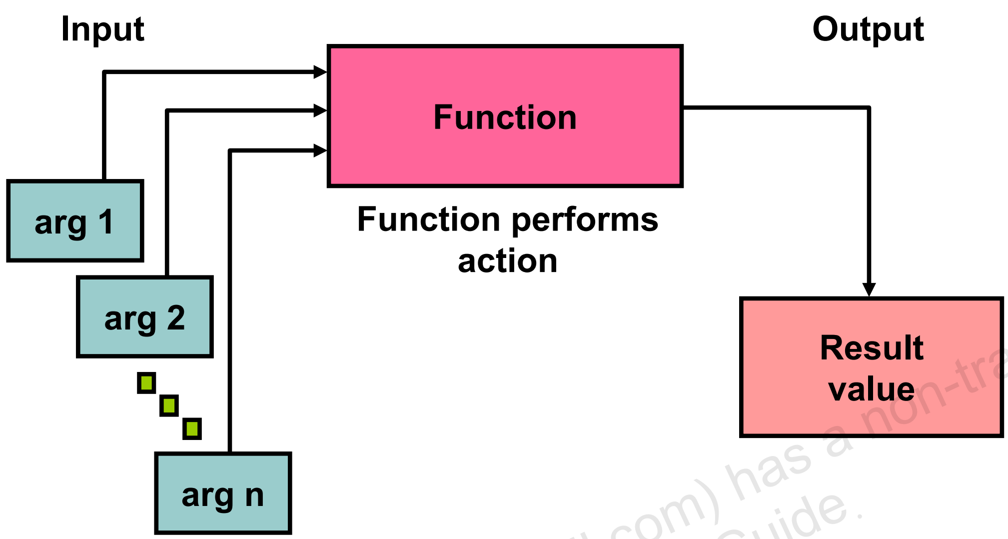
After completing this lesson, you should be able to do the following:

- Describe various types of functions that are available in SQL
- Use character, number, and date functions in `SELECT` statements
- Describe the use of conversion functions

Objectives

Functions make the basic query block more powerful, and they are used to manipulate data values. This is the first of two lessons that explore functions. It focuses on single-row character, number, and date functions, as well as those functions that convert data from one type to another (for example, conversion from character data to numeric data).

SQL Functions



SQL Functions

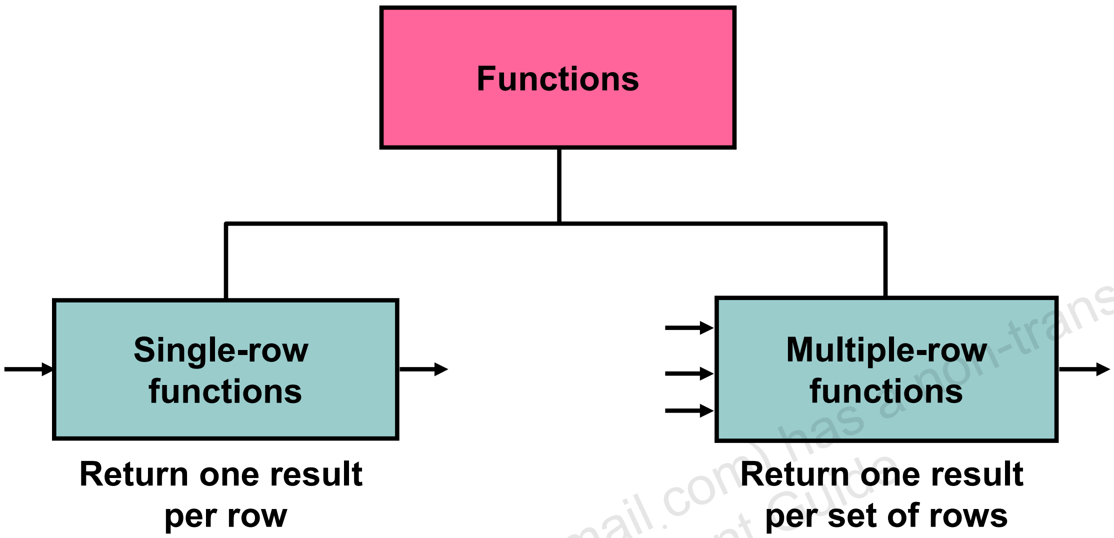
Functions are a very powerful feature of SQL. They can be used to do the following:

- 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 sometimes take arguments and always return a value.

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

Two Types of SQL Functions



SQL Functions (continued)

There are two types of functions:

- Single-row functions
- Multiple-row functions

Single-Row Functions

These functions operate on single rows only and return one result per row. There are different types of single-row functions. This lesson covers the following ones:

- Character
- Number
- Date
- Conversion
- General

Multiple-Row Functions

Functions can manipulate groups of rows to give one result per group of rows. These functions are also known as *group functions* (covered in lesson 4).

Note: For more information and a complete list of available functions and their syntax, see *Oracle SQL Reference*.

Single-Row Functions

Single-row functions:
- Manipulate data items
- Accept arguments and return one value
- Act on each row that is returned
- Return one result per row
- May modify the data type
- Can be nested
- Accept arguments that can be a column or an expression

```
function_name [(arg1, arg2,...)]
```

ORACLE

Single-Row Functions

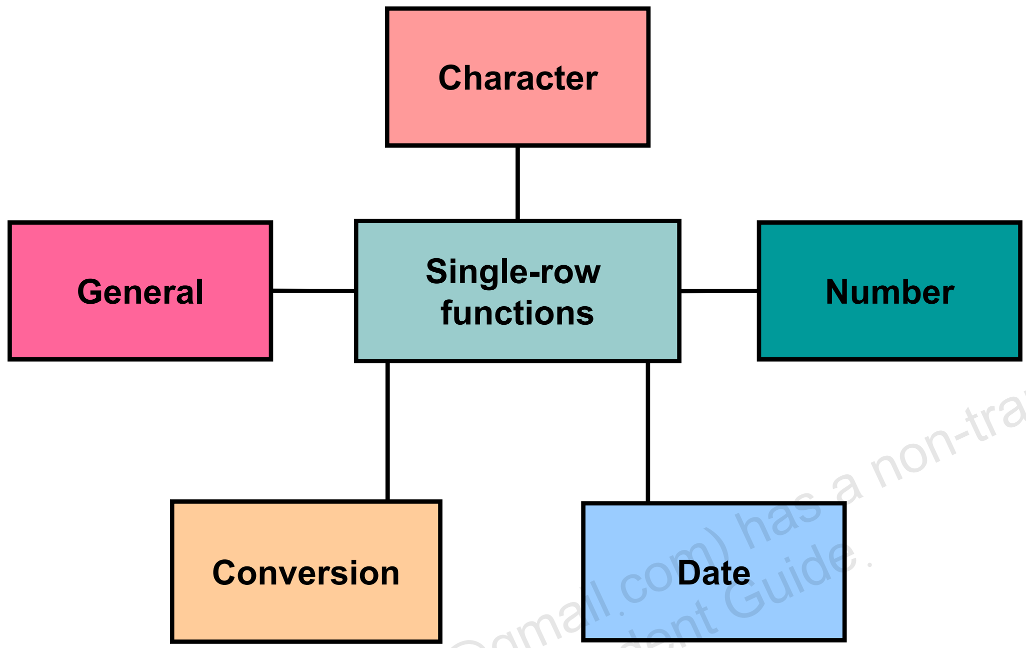
Single-row functions are used to manipulate data items. They accept one or more arguments and return one value for each row that is returned by the query. An argument can be one of the following:
- User-supplied constant
- Variable value
- Column name
- Expression

Features of single-row functions include:
- Acting on each row that is returned in the query
- Returning one result per row
- Possibly returning a data value of a different type than the one that is referenced
- Possibly expecting one or more arguments
- Can be used in `SELECT`, `WHERE`, and `ORDER BY` clauses; can be nested

In the syntax:

function_name	is the name of the function
arg1, arg2	is any argument to be used by the function. This can be represented by a column name or expression.

Single-Row Functions

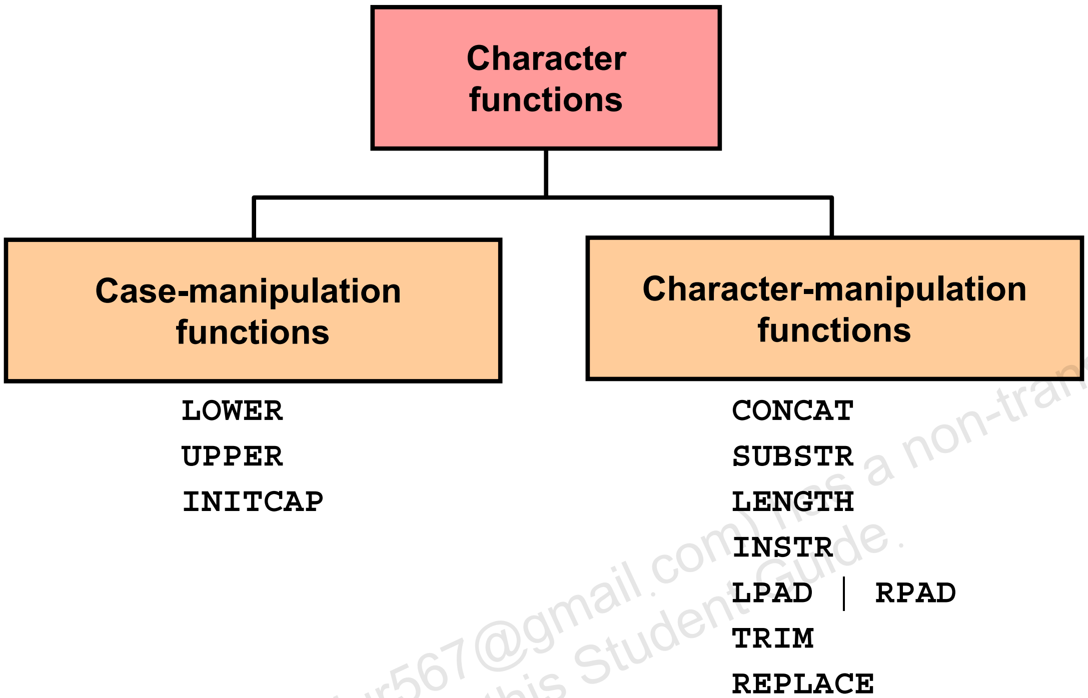


Single-Row Functions (continued)

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
- **Date functions:** Operate on values of the `DATE` data type (All date functions return a value of `DATE` data type except the `MONTHS_BETWEEN` function, which returns a number.)
- **Conversion functions:** Convert a value from one data type to another
- **General functions:**
  - `NVL`
  - `NVL2`
  - `NULLIF`
  - `COALESCE`
  - `CASE`
  - `DECODE`

Character Functions



Character Functions

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

- Case-manipulation functions
- Character-manipulation functions

<table>
<tr><th>Function</th><th>Purpose</th></tr>
<tr><td><code>LOWER(column|expression)</code></td><td>Converts alpha character values to lowercase</td></tr>
<tr><td><code>UPPER(column|expression)</code></td><td>Converts alpha character values to uppercase</td></tr>
<tr><td><code>INITCAP(column|expression)</code></td><td>Converts alpha character values to uppercase for the first letter of each word; all other letters in lowercase</td></tr>
<tr><td><code>CONCAT(column1|expression1, column2|expression2)</code></td><td>Concatenates the first character value to the second character value; equivalent to concatenation operator (||)</td></tr>
<tr><td><code>SUBSTR(column|expression,m[,n])</code></td><td>Returns specified characters from character value starting at character position m, n characters long (If m is negative, the count starts from the end of the character value. If n is omitted, all characters to the end of the string are returned.)</td></tr>
</table>

Note: The functions discussed in this lesson are only some of the available functions.

Character Functions (continued)

Function	Purpose
`LENGTH(`*`column`*`|`*`expression`*`)`	Returns the number of characters in the expression
`INSTR(`*`column|expression,`* *`'string', [,m], [n]`* `)`	Returns the numeric position of a named string. Optionally, you can provide a position *m* to start searching, and the occurrence *n* of the string. *m* and *n* default to 1, meaning start the search at the beginning of the search and report the first occurrence.
`LPAD(`*`column`*`|`*`expression, n,`* `'`*`string`*`')` `RPAD(`*`column`*`|`*`expression, n,`* `'`*`string`*`')`	Pads the character value right-justified to a total width of *n* character positions Pads the character value left-justified to a total width of *n* character positions
`TRIM(`*`leading|trailing|both, trim_character FROM trim_source`*`)`	Enables you to trim heading or trailing characters (or both) from a character string. If *`trim_character`* or *`trim_source`* is a character literal, you must enclose it in single quotation marks. This is a feature that is available in Oracle8*i* and later versions.
`REPLACE(`*`text, search_string, replacement_string`*`)`	Searches a text expression for a character string and, if found, replaces it with a specified replacement string

Case-Manipulation Functions

These functions convert case for character strings:

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

Case-Manipulation Functions

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

- **LOWER:** Converts mixed-case or uppercase character strings to lowercase
- **UPPER:** Converts mixed-case or lowercase character strings to uppercase
- **INITCAP:** Converts the first letter of each word to uppercase and remaining letters to lowercase

```
SELECT 'The job id for '||UPPER(last_name)||' is '
 ||LOWER(job_id) AS "EMPLOYEE DETAILS"
FROM   employees;
```

EMPLOYEE DETAILS
The job id for KING is ad_pres
The job id for KOCHHAR is ad_vp
The job id for DE HAAN is ad_vp
...
The job id for HIGGINS is ac_mgr
The job id for GIETZ is ac_account

20 rows selected.

Using Case-Manipulation Functions

Display the employee number, name, and department number for employee Higgins:

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

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

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
205	Higgins	110

Using Case-Manipulation Functions

The slide example displays the employee number, name, and department number of employee Higgins.

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

The WHERE clause of the second SQL statement specifies that the employee name in the EMPLOYEES table is compared to higgins, converting the LAST_NAME column to lowercase for comparison purposes. Because both names are now lowercase, a match is found and one row is selected. The WHERE clause can be rewritten in the following manner to produce the same result:

```
...WHERE last_name = 'Higgins'
```

The name in the output appears as it was stored in the database. To display the name in uppercase, use the UPPER function in the SELECT statement.

```
SELECT employee_id, UPPER(last_name), department_id
FROM   employees
WHERE  INITCAP(last_name) = 'Higgins';
```

Character-Manipulation Functions

These functions manipulate character strings:

Function	Result
`CONCAT('Hello', 'World')`	`HelloWorld`
`SUBSTR('HelloWorld',1,5)`	`Hello`
`LENGTH('HelloWorld')`	`10`
`INSTR('HelloWorld', 'W')`	`6`
`LPAD(salary,10,'*')`	`*****24000`
`RPAD(salary, 10, '*')`	`24000*****`
`REPLACE ('JACK and JUE','J','BL')`	`BLACK and BLUE`
`TRIM('H' FROM 'HelloWorld')`	`elloWorld`

ORACLE

Character-Manipulation Functions

CONCAT, SUBSTR, LENGTH, INSTR, LPAD, RPAD, and TRIM are the character-manipulation functions that are covered in this lesson.

- **CONCAT:** Joins values together (You are limited to using two parameters with CONCAT.)
- **SUBSTR:** Extracts a string of determined length
- **LENGTH:** Shows the length of a string as a numeric value
- **INSTR:** Finds the numeric position of a named character
- **LPAD:** Pads the character value right-justified
- **RPAD:** Pads the character value left-justified
- **TRIM:** Trims heading or trailing characters (or both) from a character string (If *trim_character* or *trim_source* is a character literal, you must enclose it in single quotation marks.)

Note: You can use functions such as UPPER and LOWER with ampersand substitution. For example, use UPPER('&job_title') so that the user does not have to enter the job title in a specific case.

Using the Character-Manipulation Functions

```
SELECT employee_id, CONCAT(first_name, last_name) NAME,
       job_id, LENGTH (last_name),
       INSTR(last_name, 'a') "Contains 'a'?"
FROM   employees
WHERE  SUBSTR(job_id, 4) = 'REP';
```

EMPLOYEE_ID	NAME	JOB_ID	LENGTH(LAST_NAME)	Contains 'a'?
174	EllenAbel	SA_REP	4	0
176	JonathonTaylor	SA_REP	6	2
178	KimberelyGrant	SA_REP	5	3
202	PatFay	MK_REP	3	2

Using the Character-Manipulation Functions

The slide example displays employee first names and last names joined together, the length of the employee last name, and the numeric position of the letter *a* in the employee last name for all employees who have the string REP contained in the job ID starting at the fourth position of the job ID.

Example

Modify the SQL statement in the slide to display the data for those employees whose last names end with the letter *n*.

```
SELECT employee_id, CONCAT(first_name, last_name) NAME,
LENGTH (last_name), INSTR(last_name, 'a') "Contains 'a'?"
FROM   employees
WHERE  SUBSTR(last_name, -1, 1) = 'n';
```

EMPLOYEE_ID	NAME	LENGTH(LAST_NAME)	Contains 'a'?
102	LexDe Haan	7	5
200	JenniferWhalen	6	3
201	MichaelHartstein	9	2

Number Functions

- `ROUND`: Rounds value to specified decimal
- `TRUNC`: Truncates value to specified decimal
- `MOD`: Returns remainder of division

Function	Result
`ROUND(45.926, 2)`	`45.93`
`TRUNC(45.926, 2)`	`45.92`
`MOD(1600, 300)`	`100`

Number Functions

Number functions accept numeric input and return numeric values. This section describes some of the number functions.

Function	Purpose
ROUND(*column*|*expression*, *n*)	Rounds the column, expression, or value to *n* decimal places or, if *n* is omitted, no decimal places (If *n* is negative, numbers to 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, *n* defaults to zero
MOD(*m*,*n*)	Returns the remainder of *m* divided by *n*

Note: This list contains only some of the available number functions.

For more information, see “Number Functions” in *Oracle SQL Reference.*

Using the ROUND Function

```
SELECT ROUND(45.923,2), ROUND(45.923,0),
       ROUND(45.923,-1)
FROM   DUAL;
```

ROUND(45.923,2)	ROUND(45.923,0)	ROUND(45.923,-1)
45.92	46	50

DUAL is a dummy table that you can use to view results from functions and calculations.

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. Conversely, if the second argument is –2, the value is rounded to two decimal places to the left (rounded to the nearest unit of 10).

The ROUND function can also be used with date functions. You will see examples later in this lesson.

DUAL Table

The DUAL table is owned by the user SYS and can be accessed by all users. It contains one column, DUMMY, and one row with the value X. The DUAL table is useful when you want to return a value once only (for example, the value of a constant, pseudocolumn, or expression that is not derived from a table with user data). The DUAL table is generally used for SELECT clause syntax completeness, because both SELECT and FROM clauses are mandatory, and several calculations do not need to select from actual tables.

Using the TRUNC Function

```
SELECT TRUNC(45.923,2), TRUNC(45.923),
       TRUNC(45.923,-1)
FROM   DUAL;
```

TRUNC(45.923,2)	TRUNC(45.923)	TRUNC(45.923,-1)
45.92	45	40

TRUNC Function

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

The TRUNC function works with arguments similar to those of the ROUND function. 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. Conversely, if the second argument is –2, the value is truncated to two decimal places to the left. If the second argument is –1, the value is truncated to one decimal place to the left.

Like the ROUND function, the TRUNC function can be used with date functions.

Using the MOD Function

For all employees with job title of Sales Representative, calculate the remainder of the salary after it is divided by 5,000.

```
SELECT last_name, salary, MOD(salary, 5000)
FROM   employees
WHERE  job_id = 'SA_REP';
```

LAST_NAME	SALARY	MOD(SALARY,5000)
Abel	11000	1000
Taylor	8600	3600
Grant	7000	2000

MOD Function

The MOD function finds the remainder of the first argument divided by the second argument. The slide example calculates the remainder of the salary after dividing it by 5,000 for all employees whose job ID is SA_REP.

Note: The MOD function is often used to determine if a value is odd or even.

Working with Dates

- The Oracle database stores dates in an internal numeric format: century, year, month, day, hours, minutes, and seconds.
- The default date display format is DD-MON-RR.
  - Enables you to store 21st-century dates in the 20th century by specifying only the last two digits of the year
  - Enables you to store 20th-century dates in the 21st century in the same way

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date < '01-FEB-88';
```

LAST_NAME	HIRE_DATE
King	17-JUN-87
Whalen	17-SEP-87

Oracle Date Format

The Oracle database stores dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds.

The default display and input format for any date is DD-MON-RR. Valid Oracle dates are between January 1, 4712 B.C., and December 31, 9999 A.D.

In the example in the slide, the `HIRE_DATE` column output 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` such as 17-JUN-87 is displayed as day, month, and year, there is also *time* and *century* information associated with the date. The complete data might be June 17, 1987, 5:10:43 p.m.

Oracle Date Format (continued)

This data is stored internally as follows:

CENTURY	YEAR	MONTH	DAY	HOUR	MINUTE	SECOND
19	87	06	17	17	10	43

Centuries and the Year 2000

When a record with a date column is inserted into a table, the *century* information is picked up from the `SYSDATE` function. However, when the date column is displayed on the screen, the century component is not displayed (by default).

The `DATE` data type 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 1987 or 2004, and not just as 87 or 04.

Working with Dates

SYSDATE is a function that returns:

- Date
- Time

SYSDATE Function

SYSDATE is a date function that returns the current database server 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.

Example

Display the current date using the DUAL table.

```
SELECT SYSDATE
FROM   DUAL;
```

SYSDATE
28-SEP-01

Arithmetic with Dates

- Add or subtract a number to or from a date for a resultant date value.
- Subtract two dates to find the number of days between those dates.
- Add hours to a date by dividing the number of hours by 24.

Arithmetic with Dates

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

You can perform the following operations:

Operation	Result	Description
date + number	Date	Adds a number of days to a date
date – number	Date	Subtracts a number of days from a date
date – date	Number of days	Subtracts one date from another
date + number/24	Date	Adds a number of hours to a date

Using Arithmetic Operators with Dates

```
SELECT last_name, (SYSDATE-hire_date)/7 AS WEEKS
FROM   employees
WHERE  department_id = 90;
```

LAST_NAME	WEEKS
King	744.245395
Kochhar	626.102538
De Haan	453.245395

ORACLE

Arithmetic with Dates (continued)

The example in the slide displays the last name and the number of weeks employed for all employees in department 90. It subtracts the date on which the employee was hired from the current date (SYSDATE) and divides the result by 7 to calculate the number of weeks that a worker has been employed.

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

If a more current date is subtracted from an older date, the difference is a negative number.

Date Functions

Function	Result
`MONTHS_BETWEEN`	Number of months between two dates
`ADD_MONTHS`	Add calendar months to date
`NEXT_DAY`	Next day of the date specified
`LAST_DAY`	Last day of the month
`ROUND`	Round date
`TRUNC`	Truncate date

Date Functions

Date functions operate on Oracle dates. All date functions return a value of `DATE` data type except `MONTHS_BETWEEN`, which returns a numeric value.

- **`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 noninteger part of the result represents a portion of the month.
- **`ADD_MONTHS(`*`date, n`*`):`** Adds *`n`* number of calendar months to *`date`*. The value of *`n`* must be an integer and can be negative.
- **`NEXT_DAY(`*`date, 'char'`*`):`** Finds the date of the next specified day of the week (*`'char'`*) following *`date`*. The value of *`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 that is 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 that is specified by the format model *`fmt`*. If the format model *`fmt`* is omitted, *`date`* is truncated to the nearest day.

This list is a subset of the available date functions. The format models are covered later in this lesson. Examples of format models are month and year.

Using Date Functions

Function	Result
`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-FEB-95')`	'28-FEB-95'

ORACLE

Date Functions (continued)

For example, display the employee number, hire date, number of months employed, six-month review date, first Friday after hire date, and last day of the hire month for all employees who have been employed for fewer than 80 months.

```
SELECT employee_id, 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) < 80;
```

EMPLOYEE_ID	HIRE_DATE	TENURE	REVIEW	NEXT_DAY (HIRE_DATE	LAST_DAY (HIRE_DATE
124	16-NOV-99	76.7452677	16-MAY-00	19-NOV-99	30-NOV-99
149	29-JAN-00	74.3259129	29-JUL-00	04-FEB-00	31-JAN-00

Using Date Functions

Assume `SYSDATE = '25-JUL-03'`:

Function	Result
ROUND(SYSDATE,'MONTH')	01-AUG-03
ROUND(SYSDATE ,'YEAR')	01-JAN-04
TRUNC(SYSDATE ,'MONTH')	01-JUL-03
TRUNC(SYSDATE ,'YEAR')	01-JAN-03

Date Functions (continued)

The ROUND and TRUNC functions can be used for number and date values. When used with dates, these functions round or truncate to the specified format model. Therefore, you can round dates to the nearest year or month.

Example

Compare the hire dates for all employees who started in 1997. Display the employee number, hire date, and start month using the ROUND and TRUNC functions.

```
SELECT employee_id, hire_date,
 ROUND(hire_date, 'MONTH'), TRUNC(hire_date, 'MONTH')
FROM   employees
WHERE  hire_date LIKE '%97';
```

EMPLOYEE_ID	HIRE_DATE	ROUND(HIR	TRUNC(HIR
142	29-JAN-97	01-FEB-97	01-JAN-97
202	17-AUG-97	01-SEP-97	01-AUG-97

Practice 3: Overview of Part 1

This practice covers the following topics:

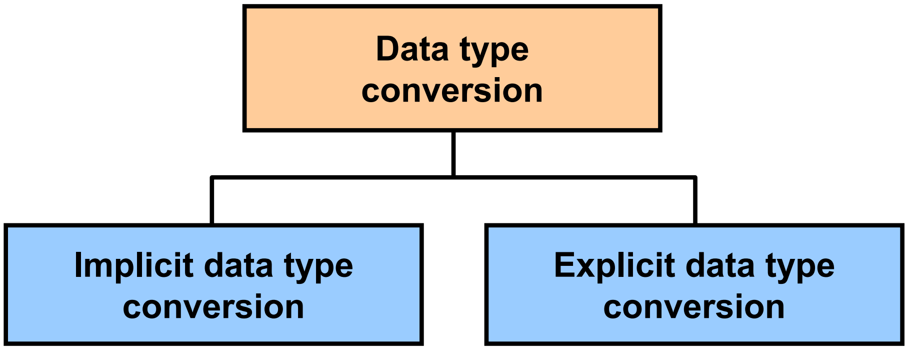
- Writing a query that displays the current date
- Creating queries that require the use of numeric, character, and date functions
- Performing calculations of years and months of service for an employee

Practice 3: Overview of Part 1

Part 1 of this lesson's practice provides a variety of exercises using different functions that are available for character, number, and date data types.

For Part 1, complete questions 1–6 at the end of this lesson.

Conversion Functions



Conversion Functions

In addition to Oracle data types, columns of tables in an Oracle database can be defined using ANSI, DB2, and SQL/DS data types. However, the Oracle server internally converts such data types to Oracle data types.

In some cases, the Oracle server uses data of one data type where it expects data of a different data type. When this happens, the Oracle server can automatically convert the data to the expected data type. This data type conversion can be done *implicitly* by the Oracle server or *explicitly* by the user.

Implicit data type conversions work according to the rules that are explained in the next two slides.

Explicit data type conversions are done by using the conversion functions. Conversion functions convert a value from one data type to another. Generally, the form of the function names follows the convention `data type TO data type`. The first data type is the input data type; the second data type is the output.

Note: Although implicit data type conversion is available, it is recommended that you do explicit data type conversion to ensure the reliability of your SQL statements.

Implicit Data Type Conversion

For assignments, the Oracle server can automatically convert the following:

From	To
`VARCHAR2 or CHAR`	`NUMBER`
`VARCHAR2 or CHAR`	`DATE`
`NUMBER`	`VARCHAR2`
`DATE`	`VARCHAR2`

Implicit Data Type Conversion

The assignment succeeds if the Oracle server can convert the data type of the value used in the assignment to that of the assignment target.

For example, the expression `hire_date > '01-JAN-90'` results in the implicit conversion from the string `'01-JAN-90'` to a date.

Implicit Data Type Conversion

For expression evaluation, the Oracle Server can automatically convert the following:

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE

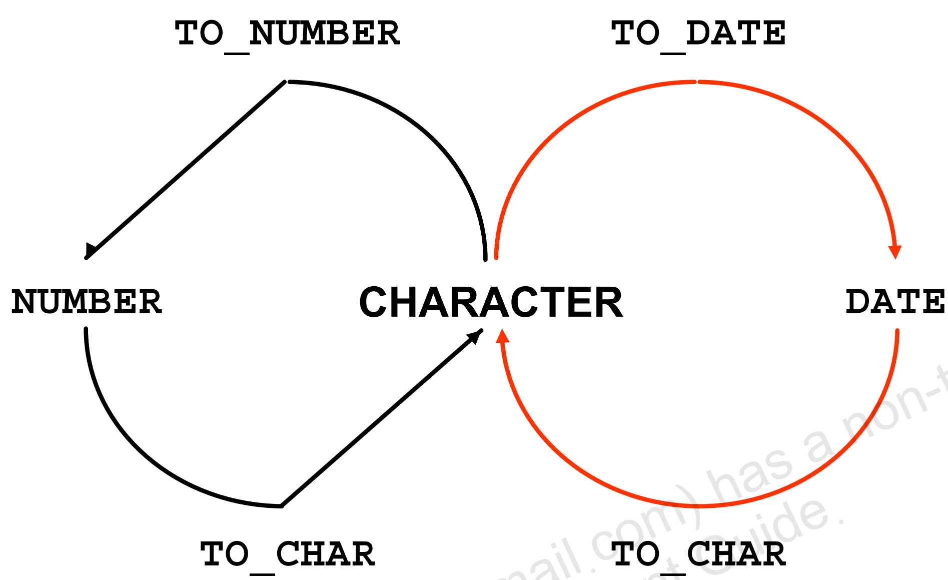
Implicit Data Type Conversion (continued)

In general, the Oracle server uses the rule for expressions when a data type conversion is needed in places that are not covered by a rule for assignment conversions.

For example, the expression `salary = '20000'` results in the implicit conversion of the string `'20000'` to the number 20000.

Note: `CHAR` to `NUMBER` conversions succeed only if the character string represents a valid number.

Explicit Data Type Conversion

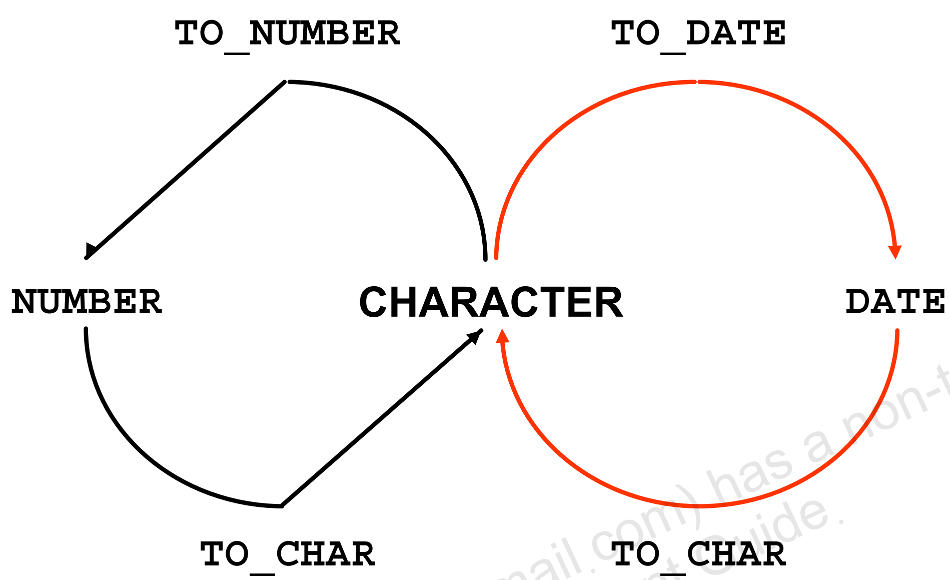


Explicit Data Type Conversion

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 other parameter is omitted, this function uses the default parameter values for the session.

Explicit Data Type Conversion



Explicit Data Type Conversion (continued)

Function	Purpose
`TO_CHAR(`*`number`*`|`*`date`*`,[ `*`fmt`*`], [`*`nlsparams`*`])`	**Date conversion:** The `nlsparams` parameter specifies the language in which month and day names and abbreviations are returned. If this parameter is omitted, this function uses the default date languages 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`* that is specified. If *`fmt`* is omitted, the format is DD-MON-YY. The `nlsparams` parameter has the same purpose in this function as in the `TO_CHAR` function for date conversion.

Explicit Data Type Conversion (continued)

Note: The list of functions mentioned in this lesson includes only some of the available conversion functions.

For more information, see “Conversion Functions” in *Oracle SQL Reference*.

Using the `TO_CHAR` Function with Dates

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

The format model:

- Must be enclosed by single quotation marks
- Is case sensitive
- 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

Displaying a Date in a Specific Format

Previously, all Oracle date values were displayed in the DD-MON-YY format. You can use the `TO_CHAR` function to convert a date from this default format to one that you specify.

Guidelines

- The format model must be enclosed by single quotation marks and is case sensitive.
- The format model can include any valid date format element. Be sure to separate the date value from the format model by 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 format the resulting character field with the *i*SQL*Plus `COLUMN` command (covered in a later lesson).

```
SELECT employee_id, TO_CHAR(hire_date, 'MM/YY') Month_Hired
FROM   employees
WHERE  last_name = 'Higgins';
```

EMPLOYEE_ID	MONTH
205	06/94

Elements of the Date Format Model

Element	Result
YYYY	Full year in numbers
YEAR	Year spelled out (in English)
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 of the week
DD	Numeric day of the month

Sample Format Elements of Valid Date Formats

Element	Description
SCC or CC	Century; server prefixes B.C. date with -
Years in dates YYYY or SYYYY	Year; server prefixes B.C. date with -
YYY or YY or Y	Last three, two, or one digits of year
Y,YYY	Year with comma in this position
IYYY, IYY, IY, I	Four-, three-, two-, or one-digit year based on the ISO standard
SYEAR or YEAR	Year spelled out; server prefixes B.C. date with -
BC or AD	Indicates B.C. or A.D. year
B.C. or A.D.	Indicates B.C. or A.D. year using periods
Q	Quarter of year
MM	Month: two-digit value
MONTH	Name of month padded with blanks to length of nine 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 a length of nine characters
DY	Name of day; three-letter abbreviation
J	Julian day; the number of days since December 31, 4713 B.C.

Elements of the Date Format Model

- Time elements format the time portion of the date:

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

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

`DD "of" MONTH`	`12 of OCTOBER`

- Number suffixes spell out numbers:

`ddspth`	`fourteenth`

Date Format Elements: Time Formats

Use the formats that are 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.

Specifying Suffixes to Influence Number Display

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 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

...

20 rows selected.

Using the TO_CHAR Function with Dates

The SQL statement in the slide displays the last names and hire dates for all the employees. The hire date appears as 17 June 1987.

Example

Modify the slide example to display the dates in a format that appears as "Seventeenth of June 1987 12:00:00 AM."

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

LAST_NAME	HIREDATE
King	Seventeenth of June 1987 12:00:00 AM
Kochhar	Twenty-First of September 1989 12:00:00 AM

...

Notice that the month follows the format model specified; in other words, the first letter is capitalized and the rest are lowercase.

Using the `TO_CHAR` Function with Numbers

```
TO_CHAR(number, 'format_model')
```

These are some of the format elements that you can use with the `TO_CHAR` function to display a number value as a character:

Element	Result
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
,	Prints a comma as thousands indicator

ORACLE

Using the `TO_CHAR` Function with Numbers

When working with number values such as character strings, you should convert those numbers to the character data type using the `TO_CHAR` function, which translates a value of `NUMBER` data type to `VARCHAR2` data type. This technique is especially useful with concatenation.

Using the `TO_CHAR` Function with Numbers (continued)

Number Format Elements

If you are converting a number to the character data type, you can use the following format 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
D	Returns in the specified position the decimal character. The default is a period (.).	99D99	99.99
.	Decimal point in position specified	999999.99	1234.00
G	Returns the group separator in the specified position. You can specify multiple group separators in a number format model.	9,999	9G999
,	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
U	Returns in the specified position the "Euro" (or other) dual currency	U9999	€1234
V	Multiply by 10 *n* times (*n* = number of 9s after V)	9999V99	123400
S	Returns the negative or positive value	S9999	-1234 or +1234
B	Display zero values as blank, not 0	B9999.99	1234.00

Using the `TO_CHAR` Function with Numbers

```
SELECT TO_CHAR(salary, '$99,999.00') SALARY
FROM   employees
WHERE  last_name = 'Ernst';
```

SALARY
$6,000.00

Guidelines

- The Oracle server displays a string of number signs (#) in place of a whole number whose digits exceed the number of digits that is provided in the format model.
- The Oracle server rounds the stored decimal value to the number of decimal places that is 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'])
```

- These functions have an fx modifier. This modifier specifies the exact matching for the character argument and date format model of a TO_DATE function.

ORACLE

Using 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, use the TO_NUMBER or TO_DATE functions. The format model that you choose is based on the previously demonstrated format elements.

The fx modifier specifies exact matching for the character argument and date format model of a TO_DATE function:

- Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.
- The character argument cannot have extra blanks. Without fx, Oracle ignores extra blanks.
- Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without fx, numbers in the character argument can omit leading zeros.

Using the `TO_NUMBER` and `TO_DATE` Functions (continued)

Example

Display the name and hire date for all employees who started on May 24, 1999. There are two spaces after the month *May* and the number *24* in the following example. Because the `fx` modifier is used, an exact match is required and the spaces after the word *May* are not recognized:

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date = TO_DATE('May   24, 1999', 'fxMonth DD, YYYY');
*
WHERE  hire_date = TO_DATE('May   24, 1999', 'fxMonth DD, YYYY')
                             *
ERROR at line 3:
ORA-01858: a non-numeric character was found where a numeric was
  expected
```

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>

ORACLE

RR Date Format Element

The RR date format is similar to the YY element, but you can use it to specify different centuries. 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 in the slide summarizes the behavior of the RR element.

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

Example of RR Date Format

To find employees hired before 1990, use the RR date format, which produces the same results whether the command is run in 1999 or now:

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')
FROM   employees
WHERE hire_date < TO_DATE('01-Jan-90','DD-Mon-RR');
```

LAST_NAME	TO_CHAR(HIR
King	17-Jun-1987
Kochhar	21-Sep-1989
Whalen	17-Sep-1987

Example of RR Date Format

To find employees who were hired before 1990, the RR format can be used. Because the current year is greater than 1999, the RR format interprets the year portion of the date from 1950 to 1999.

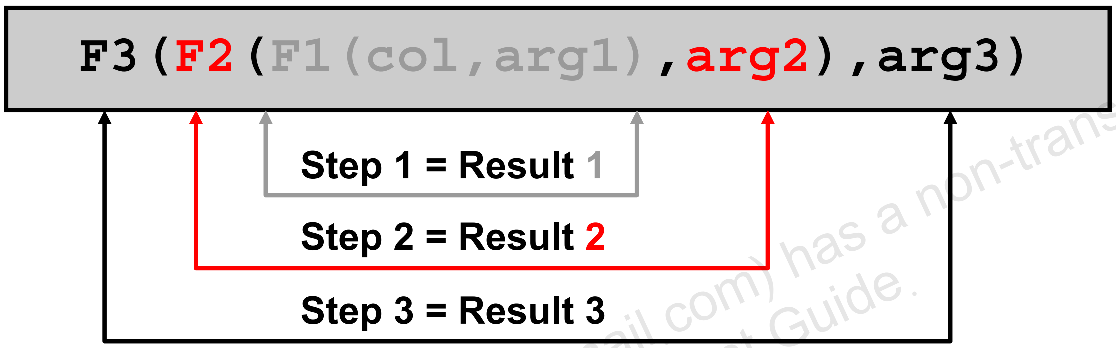
The following command, on the other hand, results in no rows being selected because the YY format interprets the year portion of the date in the current century (2090).

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-yyyy')
FROM   employees
WHERE  TO_DATE(hire_date, 'DD-Mon-yy') < '01-Jan-1990';

no rows selected
```

Nesting Functions

- Single-row functions can be nested to any level.
- Nested functions are evaluated from the deepest level to the least deep level.



Nesting Functions

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

Nesting Functions

```
SELECT last_name,
  UPPER(CONCAT(SUBSTR (LAST_NAME, 1, 8), '_US'))
FROM   employees
WHERE  department_id = 60;
```

LAST_NAME	UPPER(CONCAT(SUBSTR(LAST_NAME,1,8
Hunold	HUNOLD_US
Ernst	ERNST_US
Lorentz	LORENTZ_US

Nesting Functions (continued)

The slide example displays the last names of employees in department 60. The evaluation of the SQL statement involves three steps:

1. The inner function retrieves the first eight characters of the last name.
   `Result1 = SUBSTR (LAST_NAME, 1, 8)`
2. The outer function concatenates the result with `_US`.
   `Result2 = CONCAT(Result1, '_US')`
3. The outermost function converts the results to uppercase.

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

Example

Display the date of the next Friday that is six months from the hire date. The resulting date should appear as Friday, August 13th, 1999. Order the results by hire date.

```
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;
```

General Functions

The following functions work with any data type and pertain to using nulls:

- `NVL (expr1, expr2)`
- `NVL2 (expr1, expr2, expr3)`
- `NULLIF (expr1, expr2)`
- `COALESCE (expr1, expr2, ..., exprn)`

ORACLE

General Functions

These functions work with any data type and pertain to the use of null values in the expression list.

Function	Description
NVL	Converts a null value to an actual value
NVL2	If `expr1` is not null, NVL2 returns `expr2`. If `expr1` is null, NVL2 returns `expr3`. The argument `expr1` can have any data type.
NULLIF	Compares two expressions and returns null if they are equal; returns the first expression if they are not equal
COALESCE	Returns the first non-null expression in the expression list

Note: For more information about the hundreds of functions available, see "Functions" in *Oracle SQL Reference*.

NVL Function

Converts a null value to an actual value:

- Data types that can be used are date, character, and number.
- Data types must match:
  - `NVL(commission_pct,0)`
  - `NVL(hire_date,'01-JAN-97')`
  - `NVL(job_id,'No Job Yet')`

NVL Function

To convert a null value to an actual value, use the NVL function.

Syntax

`NVL (expr1, expr2)`

In the syntax:

- *expr1* is the source value or expression that may contain a null
- *expr2* is the target value for converting the null

You can use the NVL function to convert any data type, but the return value is always the same as the data type of *expr1*.

NVL Conversions for Various Data Types

Data Type	Conversion Example
NUMBER	NVL(*number_column*,9)
DATE	NVL(*date_column*, '01-JAN-95')
CHAR or VARCHAR2	NVL(*character_column*, 'Unavailable')

Using the NVL Function

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

LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
King	24000	0	288000
Kochhar	17000	0	204000
De Haan	17000	0	204000
Hunold	9000	0	108000
Ernst	6000	0	72000
Lorentz	4200	0	50400
Mourgos	5800	0	69600
Rajs	3500	0	42000

...

20 rows selected.

1 2

ORACLE

Using the NVL Function

To calculate the annual compensation of all employees, you need to multiply the monthly salary by 12 and then add the commission percentage to the result:

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

LAST_NAME	SALARY	COMMISSION_PCT	AN_SAL
...			
Vargas	2500		
Zlotkey	10500	.2	151200
Abel	11000	.3	171600
Taylor	8600	.2	123840
...			

Notice that the annual compensation is calculated for only those employees who earn a commission. If any column value in an expression is null, the result is null. To calculate values for all employees, 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 null values to zero.

Using the NVL2 Function

```
SELECT last_name,  salary, commission_pct,
       NVL2(commission_pct,
            'SAL+COMM', 'SAL') income
FROM   employees WHERE department_id IN (50, 80);
```

LAST_NAME	SALARY	COMMISSION_PCT	INCOME
Zlotkey	10500	.2	SAL+COMM
Abel	11000	.3	SAL+COMM
Taylor	8600	.2	SAL+COMM
Mourgos	5800		SAL
Rajs	3500		SAL
Davies	3100		SAL
Matos	2600		SAL
Vargas	2500		SAL

8 rows selected.

ORACLE

Using the NVL2 Function

The NVL2 function examines the first expression. If the first expression is not null, then the NVL2 function returns the second expression. If the first expression is null, then the third expression is returned.

Syntax

```
NVL2(expr1, expr2, expr3)
```

In the syntax:

- *expr1* is the source value or expression that may contain null
- *expr2* is the value that is returned if *expr1* is not null
- *expr3* is the value that is returned if *expr1* is null

In the example shown in the slide, the COMMISSION_PCT column is examined. If a value is detected, the second expression of SAL+COMM is returned. If the COMMISSION_PCT column holds a null value, the third expression of SAL is returned.

The argument *expr1* can have any data type. The arguments *expr2* and *expr3* can have any data types except LONG. If the data types of *expr2* and *expr3* are different, the Oracle server converts *expr3* to the data type of *expr2* before comparing them unless *expr3* is a null constant. In the latter case, a data type conversion is not necessary. The data type of the return value is always the same as the data type of *expr2*, unless *expr2* is character data, in which case the return value's data type is VARCHAR2.

Using the `NULLIF` Function

```
SELECT first_name, LENGTH(first_name) "expr1",
       last_name,  LENGTH(last_name)  "expr2",
       NULLIF(LENGTH(first_name), LENGTH(last_name)) result
FROM   employees;
```

FIRST_NAME	expr1	LAST_NAME	expr2	RESULT
Steven	6	King	4	6
Neena	5	Kochhar	7	5
Lex	3	De Haan	7	3
Alexander	9	Hunold	6	9
Bruce	5	Ernst	5	
Diana	5	Lorentz	7	5
Kevin	5	Mourgos	7	5
Trenna	6	Rajs	4	6
Curtis	6	Davies	6	

...

20 rows selected.

Using the `NULLIF` Function

The `NULLIF` function compares two expressions. If they are equal, the function returns null. If they are not equal, the function returns the first expression. You cannot specify the literal `NULL` for the first expression.

Syntax

```
NULLIF (expr1, expr2)
```

In the syntax:

- *`expr1`* is the source value compared to *`expr2`*
- *`expr2`* is the source value compared with *`expr1`* (If it is not equal to *`expr1`*, *`expr1`* is returned.)

In the example shown in the slide, the length of the first name in the `EMPLOYEES` table is compared to the length of the last name in the `EMPLOYEES` table. When the lengths of the names are equal, a null value is displayed. When the lengths of the names are not equal, the length of the first name is displayed.

Note: The `NULLIF` function is logically equivalent to the following `CASE` expression. The `CASE` expression is discussed on a subsequent page:

```
CASE WHEN expr1 = expr 2 THEN NULL ELSE expr1 END
```

Using the `COALESCE` Function

- The advantage of the `COALESCE` function over the `NVL` function is that the `COALESCE` function can take multiple alternate values.
- If the first expression is not null, the `COALESCE` function returns that expression; otherwise, it does a `COALESCE` of the remaining expressions.

Using the `COALESCE` Function

The `COALESCE` function returns the first non-null expression in the list.

Syntax

```
COALESCE (expr1, expr2, ... exprn)
```

In the syntax:

- *`expr1`* returns this expression if it is not null
- *`expr2`* returns this expression if the first expression is null and this expression is not null
- *`exprn`* returns this expression if the preceding expressions are null

All expressions must be of the same data type.

Using the COALESCE Function

```
SELECT last_name,
       COALESCE(manager_id,commission_pct, -1) comm
FROM   employees
ORDER BY commission_pct;
```

LAST_NAME	COMM
Grant	149
Zlotkey	100
Taylor	149
Abel	149
King	-1
Kochhar	100
De Haan	100

...

20 rows selected.

Using the COALESCE Function (continued)

In the example shown in the slide, if the MANAGER_ID value is not null, it is displayed. If the MANAGER_ID value is null, then the COMMISSION_PCT is displayed. If the MANAGER_ID and COMMISSION_PCT values are null, then the value –1 is displayed.

Conditional Expressions

- Provide the use of IF-THEN-ELSE logic within a SQL statement
- Use two methods:
  - `CASE` expression
  - `DECODE` function

Conditional Expressions

Two methods used to implement conditional processing (IF-THEN-ELSE logic) in a SQL statement are the `CASE` expression and the `DECODE` function.

Note: The `CASE` expression complies with ANSI SQL. The `DECODE` function is specific to Oracle syntax.

CASE Expression

Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```
CASE expr WHEN comparison_expr1 THEN return_expr1
         [WHEN comparison_expr2 THEN return_expr2
          WHEN comparison_exprn THEN return_exprn
          ELSE else_expr]
END
```


CASE Expression

CASE expressions let you use IF-THEN-ELSE logic in SQL statements without having to invoke procedures.

In a simple CASE expression, the Oracle server searches for the first WHEN ... THEN pair for which expr is equal to comparison_expr and returns return_expr. If none of the WHEN ... THEN pairs meet this condition, and if an ELSE clause exists, then the Oracle server returns else_expr. Otherwise, the Oracle server returns null. You cannot specify the literal NULL for all the return_exprs and the else_expr.

All of the expressions (expr, comparison_expr, and return_expr) must be of the same data type, which can be CHAR, VARCHAR2, NCHAR, or NVARCHAR2.

Using the CASE Expression

Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

```
SELECT last_name, job_id, salary,
       CASE job_id WHEN 'IT_PROG'  THEN  1.10*salary
                   WHEN 'ST_CLERK' THEN  1.15*salary
                   WHEN 'SA_REP'   THEN  1.20*salary
       ELSE      salary END     "REVISED_SALARY"
FROM   employees;
```

LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
...			
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

Using the CASE Expression

In the SQL statement in the slide, the value of JOB_ID is decoded. If JOB_ID is IT_PROG, the salary increase is 10%; if JOB_ID is ST_CLERK, the salary increase is 15%; if JOB_ID is SA_REP, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be written with the DECODE function.

This is an example of a searched CASE expression. In a searched CASE expression, the search occurs from left to right until an occurrence of the listed condition is found, and then it returns the return expression. If no condition is found to be true, and if an ELSE clause exists, the return expression in the ELSE clause is returned; otherwise, NULL is returned.

```
SELECT last_name,salary,
(CASE WHEN salary<5000 THEN 'Low'
      WHEN salary<10000 THEN 'Medium'
      WHEN salary<20000 THEN 'Good'
      ELSE 'Excellent'
END) qualified_salary
FROM employees;
```

`DECODE` Function

Facilitates conditional inquiries by doing the work of a `CASE` expression or an IF-THEN-ELSE statement:

```
DECODE(col|expression, search1, result1
                      [, search2, result2,...,]
                      [, default])
```


`DECODE` Function

The `DECODE` function decodes an expression in a way similar to the IF-THEN-ELSE logic that is used in various languages. The `DECODE` function decodes *`expression`* after comparing it to each *`search`* value. If the expression is the same as *`search`*, *`result`* is returned.

If the default value is omitted, a null value is returned where a search value does not match any of the result values.

Using the DECODE Function

```
SELECT last_name, job_id, salary,
       DECODE(job_id, 'IT_PROG',  1.10*salary,
                      'ST_CLERK', 1.15*salary,
                      'SA_REP',   1.20*salary,
              salary)
       REVISED_SALARY
FROM   employees;
```

LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...			
Lorentz	IT_PROG	4200	4620
Mourgos	ST_MAN	5800	5800
Rajs	ST_CLERK	3500	4025
...			
Gietz	AC_ACCOUNT	8300	8300

20 rows selected.

ORACLE

Using the DECODE Function

In the SQL statement in the slide, the value of JOB_ID is tested. If JOB_ID is IT_PROG, the salary increase is 10%; if JOB_ID is ST_CLERK, the salary increase is 15%; if JOB_ID is SA_REP, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be expressed in pseudocode as an IF-THEN-ELSE statement:

```
IF job_id = 'IT_PROG'      THEN  salary = salary*1.10
IF job_id = 'ST_CLERK'     THEN  salary = salary*1.15
IF job_id = 'SA_REP'       THEN  salary = salary*1.20
ELSE salary = salary
```

Using the `DECODE` Function

Display the applicable tax rate for each employee in department 80:

```
SELECT last_name, salary,
       DECODE (TRUNC(salary/2000, 0),
                          0, 0.00,
                          1, 0.09,
                          2, 0.20,
                          3, 0.30,
                          4, 0.40,
                          5, 0.42,
                          6, 0.44,
                             0.45) TAX_RATE
FROM   employees
WHERE  department_id = 80;
```


Using the `DECODE` function (continued)

This slide shows another example using the `DECODE` function. In this example, you determine the tax rate for each employee in department 80 based on the monthly salary. The tax rates are as follows:

Monthly Salary Range	*Tax Rate*
$0.00–1,999.99	00%
$2,000.00–3,999.99	09%
$4,000.00–5,999.99	20%
$6,000.00–7,999.99	30%
$8,000.00–9,999.99	40%
$10,000.00–11,999.99	42%
$12,200.00–13,999.99	44%
$14,000.00 or greater	45%

LAST_NAME	SALARY	TAX_RATE
Zlotkey	10500	.42
Abel	11000	.42
Taylor	8600	.4

Summary

In this lesson, you should have learned how to:

- Perform calculations on data using functions
- Modify individual data items using functions
- Manipulate output for groups of rows using functions
- Alter date formats for display using functions
- Convert column data types using functions
- Use NVL functions
- Use IF-THEN-ELSE logic

Summary

Single-row functions can be nested to any level. Single-row functions can manipulate the following:

- Character data: LOWER, UPPER, INITCAP, CONCAT, SUBSTR, INSTR, LENGTH
- Number data: ROUND, TRUNC, MOD
- Date data: MONTHS_BETWEEN, ADD_MONTHS, NEXT_DAY, LAST_DAY, ROUND, TRUNC

Remember the following:

- Date values can also use arithmetic operators.
- Conversion functions can convert character, date, and numeric values: TO_CHAR, TO_DATE, TO_NUMBER
- There are several functions that pertain to nulls, including NVL, NVL2, NULLIF, and COALESCE.
- IF-THEN-ELSE logic can be applied within a SQL statement by using the CASE expression or the DECODE function.

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.

Practice 3: Overview of Part 2

This practice covers the following topics:

- Creating queries that require the use of numeric, character, and date functions
- Using concatenation with functions
- Writing non-case-sensitive queries to test the usefulness of character functions
- Performing calculations of years and months of service for an employee
- Determining the review date for an employee

Practice 3: Overview of Part 2

Part 2 of this lesson's practice provides a variety of exercises using different functions that are available for character, number, and date data types. For Part 2, complete exercises 7–14.

Remember that for nested functions, the results are evaluated from the innermost function to the outermost function.

Practice 3

Part 1

1. Write a query to display the current date. Label the column Date.

Date
31-DEC-03

2. The HR department needs a report to display the employee number, last name, salary, and salary increased by 15.5% (expressed as a whole number) for each employee. Label the column New Salary. Place your SQL statement in a text file named lab_03_02.sql.

3. Run your query in the file lab_03_02.sql.

EMPLOYEE_ID	LAST_NAME	SALARY	New Salary
100	King	24000	27720
101	Kochhar	17000	19635
...			
202	Fay	6000	6930
205	Higgins	12000	13860
206	Gietz	8300	9587

20 rows selected.

4. Modify your query lab_03_02.sql to add a column that subtracts the old salary from the new salary. Label the column Increase. Save the contents of the file as lab_03_04.sql. Run the revised query.

EMPLOYEE_ID	LAST_NAME	SALARY	New Salary	Increase
100	King	24000	27720	3720
101	Kochhar	17000	19635	2635
102	De Haan	17000	19635	2635
...				
202	Fay	6000	6930	930
205	Higgins	12000	13860	1860
206	Gietz	8300	9587	1287

20 rows selected.

Practice 3 (continued)

5. Write a query that displays the last name (with the first letter uppercase and all other letters lowercase) and the length of the last name for all employees whose name starts with the letters *J, A,* or *M*. Give each column an appropriate label. Sort the results by the employees' last names.

Name	Length
Abel	4
Matos	5
Mourgos	7

Rewrite the query so that the user is prompted to enter a letter that starts the last name. For example, if the user enters `H` when prompted for a letter, then the output should show all employees whose last name starts with the letter *H*.

Name	Length
Hartstein	9
Higgins	7
Hunold	6

Practice 3 (continued)

6. The HR department wants to find the length of employment for each employee. For each employee, display the last name and calculate the number of months between today and the date on which the employee was hired. Label the column `MONTHS_WORKED`. Order your results by the number of months employed. Round the number of months up to the closest whole number. **Note:** Your results will differ.

LAST_NAME	MONTHS_WORKED
Zlotkey	47
Mourgos	50
Grant	55
Lorentz	59
Vargas	66
Taylor	69
Matos	70
Fay	76
Davies	83
Abel	92
Hartstein	94
Rajs	98
Higgins	115
Gietz	115
De Haan	132
Ernst	151
Hunold	168
Kochhar	171
Whalen	195
King	198

20 rows selected.

Practice 3 (continued)

Part 2

7. Create a report that produces the following for each employee:
<employee last name> earns <salary> monthly but wants <3 times salary>. Label the column Dream Salaries.

Dream Salaries
King earns $24,000.00 monthly but wants $72,000.00.
Kochhar earns $17,000.00 monthly but wants $51,000.00.
De Haan earns $17,000.00 monthly but wants $51,000.00.
...
Hartstein earns $13,000.00 monthly but wants $39,000.00.
Fay earns $6,000.00 monthly but wants $18,000.00.
Higgins earns $12,000.00 monthly but wants $36,000.00.
Gietz earns $8,300.00 monthly but wants $24,900.00.

20 rows selected.

If you have time, complete the following exercises:

8. Create a query to display the last name and salary for all employees. Format the salary to be 15 characters long, left-padded with the $ symbol. Label the column SALARY.

LAST_NAME	SALARY
King	$$$$$$$$$$24000
Kochhar	$$$$$$$$$$17000
De Haan	$$$$$$$$$$17000
Hunold	$$$$$$$$$$$9000
...	
Fay	$$$$$$$$$$$6000
Higgins	$$$$$$$$$$12000
Gietz	$$$$$$$$$$$8300

20 rows selected.

Practice 3 (continued)

9. Display each employee's last name, hire date, and salary review date, which is the first Monday after six months of service. Label the column `REVIEW`. Format the dates to appear in the format similar to "Monday, the Thirty-First of July, 2000."

LAST_NAME	HIRE_DATE	REVIEW
King	17-JUN-87	Monday, the Twenty-First of December, 1987
Kochhar	21-SEP-89	Monday, the Twenty-Sixth of March, 1990
De Haan	13-JAN-93	Monday, the Nineteenth of July, 1993
Hunold	03-JAN-90	Monday, the Ninth of July, 1990
Ernst	21-MAY-91	Monday, the Twenty-Fifth of November, 1991
Lorentz	07-FEB-99	Monday, the Ninth of August, 1999
...		
Higgins	07-JUN-94	Monday, the Twelfth of December, 1994
Gietz	07-JUN-94	Monday, the Twelfth of December, 1994

20 rows selected.

10. Display the last name, hire date, and day of the week on which the employee started. Label the column `DAY`. Order the results by the day of the week, starting with Monday.

LAST_NAME	HIRE_DATE	DAY
Grant	24-MAY-99	MONDAY
Ernst	21-MAY-91	TUESDAY
Mourgos	16-NOV-99	TUESDAY
Taylor	24-MAR-98	TUESDAY
...		
Lorentz	07-FEB-99	SUNDAY
Fay	17-AUG-97	SUNDAY
Matos	15-MAR-98	SUNDAY

20 rows selected.

Practice 3 (continued)

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

11. Create a query that displays the employees' last names and commission amounts. If an employee does not earn commission, show "No Commission." Label the column `COMM`.

LAST_NAME	COMM
King	No Commission
Kochhar	No Commission
...	
Zlotkey	.2
Abel	.3
Taylor	.2
Grant	.15
Whalen	No Commission
Hartstein	No Commission
Fay	No Commission
Higgins	No Commission
Gietz	No Commission

20 rows selected.

12. Create a query that displays the first eight characters of the employees' last names and indicates the amounts of their salaries with asterisks. Each asterisk signifies a thousand dollars. Sort the data in descending order of salary. Label the column `EMPLOYEES_AND_THEIR_SALARIES`.

EMPLOYEES_AND_THEIR_SALARIES
King ************************
Kochhar *****************
De Haan *****************
Hartstei *************
Higgins ************
...
Matos **
Vargas **

20 rows selected.

Practice 3 (continued)

13. Using the DECODE function, write a query that displays the grade of all employees based on the value of the column JOB_ID, using the following data:

Job	*Grade*
AD_PRES	A
ST_MAN	B
IT_PROG	C
SA_REP	D
ST_CLERK	E
None of the above	0

JOB_ID	GRA
AC_ACCOUNT	0
AC_MGR	0
AD_ASST	0
AD_PRES	A
AD_VP	0
AD_VP	0
IT_PROG	C
IT_PROG	C
IT_PROG	C
MK_MAN	0
MK_REP	0
SA_MAN	0
SA_REP	D
SA_REP	D
SA_REP	D
ST_CLERK	E
ST_CLERK	E
ST_CLERK	E
ST_CLERK	E
ST_MAN	B

20 rows selected.

14. Rewrite the statement in the preceding exercise using the CASE syntax.

Aggregating Data Using Group Functions

Objectives

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

- Identify the available group functions
- Describe the use of group functions
- Group data by using the `GROUP BY` clause
- Include or exclude grouped rows by using the `HAVING` clause

Objectives

This lesson further addresses functions. It focuses on obtaining summary information (such as averages) for groups of rows. It discusses how to group rows in a table into smaller sets and how to specify search criteria for groups of rows.

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
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600
	7000
10	4400

...

20 rows selected.

Maximum salary in EMPLOYEES table

MAX(SALARY)
24000

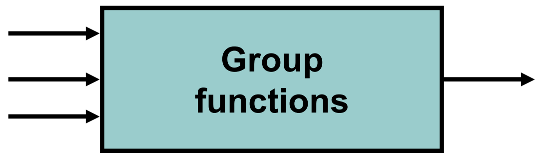
ORACLE

Group Functions

Unlike single-row functions, group functions operate on sets of rows to give one result per group. These sets may comprise the entire table or the table split into groups.

Types of Group Functions

- AVG
- COUNT
- MAX
- MIN
- STDDEV
- SUM
- VARIANCE



Types of Group Functions

Each of the functions accepts an argument. The following table identifies the options that you can use in the syntax:

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td><code>AVG([DISTINCT|ALL]n)</code></td><td>Average value of n, ignoring null values</td></tr>
<tr><td><code>COUNT({*|[DISTINCT|ALL]expr })</code></td><td>Number of rows, where expr evaluates to something other than null (count all selected rows using *, including duplicates and rows with nulls)</td></tr>
<tr><td><code>MAX([DISTINCT|ALL]expr)</code></td><td>Maximum value of expr, ignoring null values</td></tr>
<tr><td><code>MIN([DISTINCT|ALL]expr)</code></td><td>Minimum value of expr, ignoring null values</td></tr>
<tr><td><code>STDDEV([DISTINCT|ALL]x)</code></td><td>Standard deviation of n, ignoring null values</td></tr>
<tr><td><code>SUM([DISTINCT|ALL]n)</code></td><td>Sum values of n, ignoring null values</td></tr>
<tr><td><code>VARIANCE([DISTINCT|ALL]x)</code></td><td>Variance of n, ignoring null values</td></tr>
</table>

Group Functions: Syntax

```
SELECT      [column,] group_function(column), ...
FROM        table
[WHERE      condition]
[GROUP BY   column]
[ORDER BY   column];
```


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` and therefore does not need to be specified.
- The data types for the functions with an `expr` argument may be `CHAR`, `VARCHAR2`, `NUMBER`, or `DATE`.
- All group functions ignore null values. To substitute a value for null values, use the `NVL`, `NVL2`, or `COALESCE` functions.

Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

```
SELECT AVG(salary), MAX(salary),
       MIN(salary), SUM(salary)
FROM   employees
WHERE  job_id LIKE '%REP%';
```

AVG(SALARY)	MAX(SALARY)	MIN(SALARY)	SUM(SALARY)
8150	11000	6000	32600

Using the Group Functions

You can use AVG, SUM, MIN, and MAX functions against columns that can store numeric data. The example in the slide displays the average, highest, lowest, and sum of monthly salaries for all sales representatives.

Using the MIN and MAX Functions

You can use MIN and MAX for numeric, character, and date data types.

```
SELECT MIN(hire_date), MAX(hire_date)
FROM   employees;
```

MIN(HIRE_	MAX(HIRE_
17-JUN-87	29-JAN-00

Using the Group Functions (continued)

You can use the MAX and MIN functions for numeric, character, and date data types. The slide example displays the most junior and most senior employees.

The following example displays the employee last name that is first and the employee last name that is last in an alphabetized list of all employees:

```
SELECT MIN(last_name), MAX(last_name)
FROM   employees;
```

MIN(LAST_NAME)	MAX(LAST_NAME)
Abel	Zlotkey

Note: The AVG, SUM, VARIANCE, and STDDEV functions can be used only with numeric data types. MAX and MIN cannot be used with LOB or LONG data types.

Using the COUNT Function

COUNT(*) returns the number of rows in a table:

1

```
SELECT COUNT(*)
FROM   employees
WHERE  department_id = 50;
```

COUNT(*)
5

COUNT(*expr*) returns the number of rows with non-null values for the *expr*:

2

```
SELECT COUNT(commission_pct)
FROM   employees
WHERE  department_id = 80;
```

COUNT(COMMISSION_PCT)
3

COUNT Function

The COUNT function has three formats:

- COUNT(*)
- COUNT(*expr*)
- COUNT(DISTINCT *expr*)

COUNT(*) returns the number of rows in a table that satisfy the criteria of the SELECT statement, including duplicate rows and rows containing null values in any of the columns. If a WHERE clause is included in the SELECT statement, COUNT(*) returns the number of rows that satisfy the condition in the WHERE clause.

In contrast, COUNT(*expr*) returns the number of non-null values that are in the column identified by *expr*.

COUNT(DISTINCT *expr*) returns the number of unique, non-null values that are in the column identified by *expr*.

Examples

1. The slide example displays the number of employees in department 50.
2. The slide example displays the number of employees in department 80 who can earn a commission.

Using the `DISTINCT` Keyword

- `COUNT(DISTINCT expr)` returns the number of distinct non-null values of the *expr*.
- To display the number of distinct department values in the `EMPLOYEES` table:

```
SELECT COUNT(DISTINCT department_id)
FROM   employees;
```

COUNT(DISTINCTDEPARTMENT_ID)
7

`DISTINCT` Keyword

Use the `DISTINCT` keyword to suppress the counting of any duplicate values in a column.

The example in the slide displays the number of distinct department values that are in the `EMPLOYEES` table.

Group Functions and Null Values

Group functions ignore null values in the column:

1

```
SELECT AVG(commission_pct)
FROM   employees;
```

AVG(COMMISSION_PCT)
.2125

The NVL function forces group functions to include null values:

2

```
SELECT AVG(NVL(commission_pct, 0))
FROM   employees;
```

AVG(NVL(COMMISSION_PCT,0))
.0425

ORACLE

Group Functions and Null Values

All group functions ignore null values in the column.

The NVL function forces group functions to include null values.

Examples

1. The average is calculated based on *only* those rows in the table where a valid value is stored in the COMMISSION_PCT column. The average is calculated as the total commission that is paid to all employees divided by the number of employees receiving a commission (four).
2. 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 the total commission that is paid to all employees divided by the total number of employees in the company (20).

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

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

Creating Groups of Data

Until this point in our discussion, all group functions have treated the table as one large group of information.

At times, however, 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 Syntax

```
SELECT     column, group_function(column)
FROM       table
[WHERE     condition]
[GROUP BY  group_by_expression]
[ORDER BY  column];
```

You can divide rows in a table into smaller groups by using the GROUP BY clause.

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 receive an error message if you fail to include the column list in the GROUP BY clause.
- Using a WHERE clause, you can 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.

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 group functions are included in the GROUP BY clause. The example in the slide displays the department number and the average salary for each department. Here is how this SELECT statement, containing a GROUP BY clause, is evaluated:

- The SELECT clause specifies the columns to be retrieved, as follows:
  - Department number column in the EMPLOYEES table
  - The average of all the salaries in the group that 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, all rows are retrieved by default.
- The GROUP BY clause specifies how the rows should be grouped. The rows are grouped by department number, so the AVG function that is 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

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 salaries for each department without displaying the respective department numbers. Without the department numbers, however, the results do not look meaningful.

You can use the 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
...	
90	19333.3333

8 rows selected.

Grouping by More Than One Column

EMPLOYEES

DEPARTMENT_ID	JOB_ID	SALARY
90	AD_PRES	24000
90	AD_VP	17000
90	AD_VP	17000
60	IT_PROG	9000
60	IT_PROG	6000
60	IT_PROG	4200
50	ST_MAN	5800
50	ST_CLERK	3500
50	ST_CLERK	3100
50	ST_CLERK	2600
50	ST_CLERK	2500
80	SA_MAN	10500
80	SA_REP	11000
80	SA_REP	8600

...

20	MK_REP	6000
110	AC_MGR	12000
110	AC_ACCOUNT	8300

20 rows selected.

Add the salaries in the `EMPLOYEES` table for each job, grouped by department

DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

ORACLE

Groups Within Groups

Sometimes you need to see results for groups within groups. The slide shows a report that displays the total salary that is paid to each job title in each department.

The `EMPLOYEES` table is grouped first by department number and then by job title within that grouping. For example, the four stock clerks in department 50 are grouped together, and a single result (total salary) is produced for all stock clerks in the group.

Using the GROUP BY Clause on Multiple Columns

```
SELECT   department_id dept_id, job_id, SUM(salary)
FROM     employees
GROUP BY department_id, job_id ;
```

DEPT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

ORACLE

Groups Within Groups (continued)

You can return summary results for groups and subgroups by listing more than one GROUP BY column. You can determine the default sort order of the results by the order of the columns in the GROUP BY clause. In the slide example, the SELECT statement containing a GROUP BY clause is evaluated as follows:

- The SELECT clause specifies the column to be retrieved:
  - Department number in the EMPLOYEES table
  - Job ID in the EMPLOYEES table
  - The sum of all the salaries in the group that you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMPLOYEES table.
- The GROUP BY clause specifies how you must group the rows:
  - First, the rows are grouped by department number.
  - Second, the rows are grouped by job ID in the department number groups.

So the SUM function is applied to the salary column for all job IDs in each department number group.

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

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, then the error message "not a single-group group function" appears and an asterisk (*) points to the offending column. You can correct the error in the slide by adding the GROUP BY clause:

```
SELECT   department_id, count(last_name)
FROM     employees
GROUP BY department_id;
```

DEPARTMENT_ID	COUNT(LAST_NAME)
10	1
20	2
...	
	1

8 rows selected.

Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause.

Illegal Queries Using Group Functions

- You cannot use the `WHERE` clause to restrict groups.
- You use the `HAVING` clause to restrict groups.
- You cannot use group functions in the `WHERE` clause.

```
SELECT    department_id, AVG(salary)
FROM      employees
WHERE     AVG(salary) > 8000
GROUP BY department_id;
```

```
WHERE  AVG(salary) > 8000
       *
ERROR at line 3:
ORA-00934: group function is not allowed here
```

Cannot use the `WHERE` clause to restrict groups

Illegal Queries Using Group Functions (continued)

The `WHERE` clause cannot be used to restrict groups. The `SELECT` statement in the slide example results in an error because it uses the `WHERE` clause to restrict the display of average salaries of those departments that have an average salary greater than $8,000.

You can correct the error in the example by using the `HAVING` clause to restrict groups:

```
SELECT    department_id, AVG(salary)
FROM      employees
HAVING    AVG(salary) > 8000
GROUP BY department_id;
```

DEPARTMENT_ID	AVG(SALARY)
20	9500
80	10033.3333
90	19333.3333
110	10150

Restricting Group Results

EMPLOYEES

DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600
...	
20	6000
110	12000
110	8300

20 rows selected.

The maximum salary per department when it is greater than $10,000

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

ORACLE

Restricting Group Results

In the same way that you use the `WHERE` clause to restrict the rows that you select, you use the `HAVING` clause to restrict groups. To find the maximum salary in each of the departments that have a maximum salary greater than $10,000, you need to do the following:

1. Find the average salary for each department by grouping by department number.
2. Restrict the groups to those departments with a maximum salary greater than $10,000.

Restricting Group Results with the HAVING Clause

When you use the HAVING clause, the Oracle server restricts groups as follows:

1. Rows are grouped.
2. The group function is applied.
3. Groups matching the HAVING clause are displayed.

```
SELECT     column, group_function
FROM       table
[WHERE     condition]
[GROUP BY group_by_expression]
[HAVING    group_condition]
[ORDER BY column];
```


Restricting Group Results with the HAVING Clause

You use the HAVING clause to specify which groups are to be displayed, thus further restricting the groups on the basis of aggregate information.

In the syntax, *group_condition* restricts the groups of rows returned to those groups for which the specified condition is true.

The Oracle server performs the following steps when you use the HAVING clause:

1. Rows are grouped.
2. The group function is applied to the group.
3. The groups that match the criteria in the HAVING clause are displayed.

The HAVING clause can precede the GROUP BY clause, but it is recommended that you place the GROUP BY clause first because that is more logical. Groups are formed and group functions are calculated before the HAVING clause is applied to the groups in the SELECT list.

Using the HAVING Clause

```
SELECT   department_id, MAX(salary)
FROM     employees
GROUP BY department_id
HAVING   MAX(salary)>10000 ;
```

DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

Using the HAVING Clause

The slide example displays department numbers and maximum salaries for those departments with a maximum salary that is greater than $10,000.

You can use the GROUP BY clause without using a group function in the SELECT list.

If you restrict rows based on the result of a group function, you must have a GROUP BY clause as well as the HAVING clause.

The following example displays the department numbers and average salaries for those departments with a maximum salary that is greater than $10,000:

```
SELECT   department_id, AVG(salary)
FROM     employees
GROUP BY department_id
HAVING   max(salary)>10000;
```

DEPARTMENT_ID	AVG(SALARY)
20	9500
80	10033.3333
90	19333.3333
110	10150

Using the HAVING Clause

```
SELECT   job_id, SUM(salary) PAYROLL
FROM     employees
WHERE    job_id NOT LIKE '%REP%'
GROUP BY job_id
HAVING   SUM(salary) > 13000
ORDER BY SUM(salary);
```

JOB_ID	PAYROLL
IT_PROG	19200
AD_PRES	24000
AD_VP	34000

Using the HAVING Clause (continued)

The slide example displays the job ID and total monthly salary for each job that has a total payroll exceeding $13,000. The example excludes sales representatives and sorts the list by the total monthly salary.

Nesting Group Functions

Display the maximum average salary:

```
SELECT   MAX(AVG(salary))
FROM     employees
GROUP BY department_id;
```

MAX(AVG(SALARY))
19333.3333

Nesting Group Functions

Group functions can be nested to a depth of two. The slide example displays the maximum average salary.

Summary

In this lesson, you should have learned how to:

- Use the group functions COUNT, MAX, MIN, and AVG
- Write queries that use the GROUP BY clause
- Write queries that use the HAVING clause

```
SELECT     column, group_function
FROM       table
[WHERE     condition]
[GROUP BY  group_by_expression]
[HAVING    group_condition]
[ORDER BY  column];
```

ORACLE

Summary

Several group functions are available in SQL, such as the following:
AVG, COUNT, MAX, MIN, SUM, STDDEV, and VARIANCE

You can create subgroups by using the GROUP BY clause. Groups can be restricted using the HAVING clause.

Place the HAVING and GROUP BY clauses after the WHERE clause in a statement. The order of the HAVING and GROUP clauses following the WHERE clause is not important. Place the ORDER BY clause last.

The Oracle server evaluates the clauses in the following order:

1. If the statement contains a WHERE clause, the server establishes the candidate rows.
2. The server identifies the groups that are specified in the GROUP BY clause.
3. The HAVING clause further restricts result groups that do not meet the group criteria in the HAVING clause.

Note: For a complete list of the group functions, see *Oracle SQL Reference*.

Practice 4: Overview

This practice covers the following topics:

- Writing queries that use the group functions
- Grouping by rows to achieve more than one result
- Restricting groups by using the `HAVING` clause

Practice 4: Overview

At the end of this practice, you should be familiar with using group functions and selecting groups of data.

Practice 4

Determine the validity of the following three statements. Circle either True or False.

1. Group functions work across many rows to produce one result per group.
   True/False

2. Group functions include nulls in calculations.
   True/False

3. The `WHERE` clause restricts rows before inclusion in a group calculation.
   True/False

The HR department needs the following reports:

4. Find the highest, lowest, sum, and average salary of all employees. Label the columns `Maximum`, `Minimum`, `Sum`, and `Average`, respectively. Round your results to the nearest whole number. Place your SQL statement in a text file named `lab_04_04.sql`.

Maximum	Minimum	Sum	Average
24000	2500	175500	8775

5. Modify the query in `lab_04_04.sql` to display the minimum, maximum, sum, and average salary for each job type. Resave `lab_04_04.sql` as `lab_04_05.sql`. Run the statement in `lab_04_05.sql`.

JOB_ID	Maximum	Minimum	Sum	Average
AC_ACCOUNT	8300	8300	8300	8300
AC_MGR	12000	12000	12000	12000
AD_ASST	4400	4400	4400	4400
AD_PRES	24000	24000	24000	24000
AD_VP	17000	17000	34000	17000
IT_PROG	9000	4200	19200	6400
MK_MAN	13000	13000	13000	13000
MK_REP	6000	6000	6000	6000
SA_MAN	10500	10500	10500	10500
SA_REP	11000	7000	26600	8867
ST_CLERK	3500	2500	11700	2925
ST_MAN	5800	5800	5800	5800

12 rows selected.

6. Write a query to display the number of people with the same job.

JOB_ID	COUNT(*)
AC_ACCOUNT	1
AC_MGR	1
AD_ASST	1
AD_PRES	1
AD_VP	2
IT_PROG	3
MK_MAN	1
MK_REP	1
SA_MAN	1
SA_REP	3
ST_CLERK	4
ST_MAN	1

12 rows selected.

Generalize the query so that the user in the HR department is prompted for a job title. Save the script to a file named `lab_04_06.sql`.

7. Determine the number of managers without listing them. Label the column `Number of Managers`. *Hint: Use the* `MANAGER_ID` *column to determine the number of managers.*

Number of Managers
8

8. Find the difference between the highest and lowest salaries. Label the column `DIFFERENCE`.

DIFFERENCE
21500

If you have time, complete the following exercises:

9. Create a report to display the manager number and the salary of the lowest-paid employee for that manager. Exclude anyone whose manager is not known. Exclude any groups where the minimum salary is $6,000 or less. Sort the output in descending order of salary.

MANAGER_ID	MIN(SALARY)
102	9000
205	8300
149	7000

Practice 4 (continued)

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

10. Create a query to display the total number of employees and, of that total, the number of employees hired in 1995, 1996, 1997, and 1998. Create appropriate column headings.

TOTAL	1995	1996	1997	1998
20	1	2	2	3

11. Create a matrix query to display the job, the salary for that job based on department number, and the total salary for that job, for departments 20, 50, 80, and 90, giving each column an appropriate heading.

Job	Dept 20	Dept 50	Dept 80	Dept 90	Total
AC_ACCOUNT					8300
AC_MGR					12000
AD_ASST					4400
AD_PRES				24000	24000
AD_VP				34000	34000
IT_PROG					19200
MK_MAN	13000				13000
MK_REP	6000				6000
SA_MAN			10500		10500
SA_REP			19600		26600
ST_CLERK		11700			11700
ST_MAN		5800			5800

12 rows selected.

5

Displaying Data from Multiple Tables

Objectives

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

- Write `SELECT` statements to access data from more than one table using equijoins and nonequijoins
- Join a table to itself by using a self-join
- View data that generally does not meet a join condition by using outer joins
- Generate a Cartesian product of all rows from two or more tables

Objectives

This lesson explains how to obtain data from more than one table. A *join* is used to view information from multiple tables. Therefore, you can *join* tables together to view information from more than one table.

Note: Information on joins is found in "SQL Queries and Subqueries: Joins" in *Oracle SQL Reference*.

Obtaining Data from Multiple Tables

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
...		
202	Fay	20
205	Higgins	110
206	Gietz	110

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

EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
...		
102	90	Executive
205	110	Accounting
206	110	Accounting

ORACLE

Obtaining 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.

Types of Joins

Joins that are compliant with the SQL:1999 standard include the following:

- Cross joins
- Natural joins
- `USING` clause
- Full (or two-sided) outer joins
- Arbitrary join conditions for outer joins

Types of Joins

To join tables, you can use join syntax that is compliant with the SQL:1999 standard.

Note: Before the Oracle9*i* release, the join syntax was different from the ANSI standards. The SQL:1999–compliant join syntax does not offer any performance benefits over the Oracle-proprietary join syntax that existed in prior releases. For detailed information about the proprietary join syntax, see Appendix B.

Joining Tables Using SQL:1999 Syntax

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

```
SELECT   table1.column, table2.column
FROM     table1
[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)]|
[CROSS JOIN table2];
```


Defining Joins

In the syntax:

`table1.column` denotes the table and column from which data is retrieved

`NATURAL JOIN` joins two tables based on the same column name

`JOIN` *`table`* `USING` *`column_name`* performs an equijoin based on the column name

`JOIN` *`table ON table1.column_name`* performs an equijoin based on the condition in the ON clause, `=` *`table2.column_name`*

`LEFT/RIGHT/FULL OUTER` is used to perform outer joins

`CROSS JOIN` returns a Cartesian product from the two tables

For more information, see "`SELECT`" in *Oracle SQL Reference*.

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, an error is returned.

Creating Natural Joins

You can join tables automatically based on columns in the two tables that have matching data types and names. You do this by using the keywords `NATURAL JOIN`.

Note: The join can happen on only those columns that have the same names and data types in both tables. If the columns have the same name but different data types, then the `NATURAL JOIN` syntax causes 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

Retrieving Records with Natural Joins

In the example in the slide, 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, the join would have used them all.

Natural Joins with a WHERE Clause

Additional restrictions on a natural join are implemented by using a WHERE clause. The following example limits the rows of output to those with a department ID equal to 20 or 50:

```
SELECT   department_id, department_name,
         location_id, city
FROM     departments
NATURAL  JOIN locations
WHERE    department_id IN (20, 50);
```

Creating Joins with the `USING` Clause

- If several columns have the same names but the data types do not match, the `NATURAL JOIN` clause can be modified with the `USING` clause to specify the columns that should be used for an equijoin.
- Use the `USING` clause to match only one column when more than one column matches.
- Do not use a table name or alias in the referenced columns.
- The `NATURAL JOIN` and `USING` clauses are mutually exclusive.

`USING` Clause

Natural joins use all columns with matching names and data types to join the tables. The `USING` clause can be used to specify only those columns that should be used for an equijoin. The columns that are referenced in the `USING` clause should not have a qualifier (table name or alias) anywhere in the SQL statement.

For example, the following statement is valid:

```
SELECT l.city, d.department_name
FROM   locations l JOIN departments d USING (location_id)
WHERE  location_id = 1400;
```

The following statement is invalid because the `LOCATION_ID` is qualified in the `WHERE` clause:

```
SELECT l.city, d.department_name
FROM locations l JOIN departments d USING (location_id)
WHERE d.location_id = 1400;
ORA-25154: column part of USING clause cannot have qualifier
```

The same restriction also applies to `NATURAL` joins. Therefore, columns that have the same name in both tables must be used without any qualifiers.

Joining Column Names

EMPLOYEES

EMPLOYEE_ID	DEPARTMENT_ID
200	10
201	20
202	20
124	50
141	50
142	50
143	50
144	50
103	60
104	60
107	60
149	80
174	80
176	80

…

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

…

Foreign key **Primary key**

ORACLE

The `USING` Clause for 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 in 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 the USING Clause

```
SELECT employees.employee_id, employees.last_name,
       departments.location_id, department_id
FROM   employees JOIN departments
USING (department_id) ;
```

EMPLOYEE_ID	LAST_NAME	LOCATION_ID	DEPARTMENT_ID
200	Whalen	1700	10
201	Hartstein	1800	20
202	Fay	1800	20
124	Mourgos	1500	50
141	Rajs	1500	50
142	Davies	1500	50
144	Vargas	1500	50
143	Matos	1500	50

...

19 rows selected.

Retrieving Records with the USING Clause

The slide example joins the DEPARTMENT_ID column in the EMPLOYEES and DEPARTMENTS tables, and thus shows the location where an employee works.

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.
- Use column aliases to distinguish columns that have identical names but reside in different tables.
- Do not use aliases on columns that are identified in the `USING` clause and listed elsewhere in the SQL statement.

Qualifying Ambiguous Column Names

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

```
SELECT employees.employee_id, employees.last_name,
       departments.department_id, departments.location_id
FROM   employees JOIN departments
ON     employees.department_id = departments.department_id;
```

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.

Note: When joining with the `USING` clause, you cannot qualify a column that is used in the `USING` clause itself. Furthermore, if that column is used anywhere in the SQL statement, you cannot alias it.

Using Table Aliases

- Use table aliases to simplify queries.
- Use table aliases to improve performance.

```
SELECT e.employee_id, e.last_name,
       d.location_id, department_id
FROM   employees e JOIN departments d
USING (department_id) ;
```


Using 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 to keep SQL code smaller, therefore using 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 been given an alias of `e`, and the `DEPARTMENTS` table an alias of `d`.

Guidelines

- Table aliases can be up to 30 characters in length, but shorter aliases are better than longer ones.
- If a table alias is used for a particular table name in the `FROM` clause, then that table alias must be substituted for the table name throughout the `SELECT` statement.
- Table aliases should be meaningful.
- The table alias is valid for only the current `SELECT` statement.

Creating Joins with the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- Use the ON clause to specify arbitrary conditions or specify columns to join.
- The join condition is separated from other search conditions.
- The ON clause makes code easy to understand.

ON Clause

Use the ON clause to specify a join condition. This lets you specify join conditions separate from any search or filter conditions in the WHERE clause.

Retrieving Records with the ON Clause

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id);
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500

...

19 rows selected.

ORACLE

Creating Joins with the ON Clause

In this example, the DEPARTMENT_ID columns in the EMPLOYEES and DEPARTMENTS table are joined using the ON clause. Wherever a department ID in the EMPLOYEES table equals a department ID in the DEPARTMENTS table, the row is returned.

You can also use the ON clause to join columns that have different names.

Self-Joins Using the ON Clause

EMPLOYEES (WORKER)

EMPLOYEE_ID	LAST_NAME	MANAGER_ID
100	King	
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103
107	Lorentz	103
124	Mourgos	100

...

EMPLOYEES (MANAGER)

EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos

...

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

ORACLE

Joining a Table to Itself

Sometimes you need to join a table to itself. 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 Lorentz's manager, you need to:

- Find Lorentz in the EMPLOYEES table by looking at the LAST_NAME column
- Find the manager number for Lorentz by looking at the MANAGER_ID column. Lorentz's manager number is 103.
- Find the name of the manager with EMPLOYEE_ID 103 by looking at the LAST_NAME column. Hunold's employee number is 103, so Hunold is Lorentz's manager.

In this process, you look in the table twice. The first time you look in the table to find Lorentz in the LAST_NAME column and MANAGER_ID value of 103. The second time you look in the EMPLOYEE_ID column to find 103 and the LAST_NAME column to find Hunold.

Self-Joins Using the ON Clause

```
SELECT e.last_name emp, m.last_name mgr
FROM   employees e JOIN employees m
ON    (e.manager_id = m.employee_id);
```

EMP	MGR
Hartstein	King
Zlotkey	King
Mourgos	King
De Haan	King
Kochhar	King

...

19 rows selected.

Joining a Table to Itself (continued)

The ON clause can also be used to join columns that have different names, within the same table or in a different table.

The example shown is a self-join of the EMPLOYEES table, based on the EMPLOYEE_ID and MANAGER_ID columns.

Applying Additional Conditions to a Join

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id)
AND    e.manager_id = 149 ;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
174	Abel	80	80	2500
176	Taylor	80	80	2500

ORACLE

Applying Additional Conditions to a Join

You can apply additional conditions to the join.

The example shown performs a join on the EMPLOYEES and DEPARTMENTS tables and, in addition, displays only employees who have a manager ID of 149. To add additional conditions to the ON clause, you can add AND clauses. Alternatively, you can use a WHERE clause to apply additional conditions:

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON    (e.department_id = d.department_id)
WHERE  e.manager_id = 149;
```

Creating Three-Way Joins with the `ON` Clause

```
SELECT employee_id, city, department_name
FROM   employees e
JOIN   departments d
ON     d.department_id = e.department_id
JOIN   locations l
ON     d.location_id = l.location_id;
```

EMPLOYEE_ID	CITY	DEPARTMENT_NAME
103	Southlake	IT
104	Southlake	IT
107	Southlake	IT
124	South San Francisco	Shipping
141	South San Francisco	Shipping
142	South San Francisco	Shipping
143	South San Francisco	Shipping
144	South San Francisco	Shipping

...

19 rows selected.

ORACLE

Three-Way Joins

A three-way join is a join of three tables. In SQL:1999–compliant syntax, joins are performed from left to right. So, the first join to be performed is `EMPLOYEES JOIN DEPARTMENTS`. The first join condition can reference columns in `EMPLOYEES` and `DEPARTMENTS` but cannot reference columns in `LOCATIONS`. The second join condition can reference columns from all three tables.

Nonequijoins

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
Taylor	8600

...

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

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

Nonequijoins

A nonequijoin 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 nonequijoin. 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 equality (=).

Retrieving Records with Nonequijoins

```
SELECT e.last_name, e.salary, j.grade_level
FROM   employees e JOIN job_grades j
ON     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

...

20 rows selected.

ORACLE

Nonequijoins (continued)

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

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 grade table contain grades that overlap. That is, the salary value for an employee can lie only between the low salary and high salary values of one of the rows in the salary grade table.
- All of the employees' salaries lie within the limits that are provided by the job grade 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 conditions (such as `<=` and `>=`) can 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 in the slide example for performance reasons, not because of possible ambiguity.

Outer Joins

DEPARTMENTS

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

8 rows selected.

EMPLOYEES

DEPARTMENT_ID	LAST_NAME
90	King
90	Kochhar
90	De Haan
60	Hunold
60	Ernst
60	Lorentz
50	Mourgos
50	Rajs
50	Davies
50	Matos
50	Vargas
80	Zlotkey

…

20 rows selected.

There are no employees in department 190.

ORACLE

Returning Records with No Direct Match with Outer Joins

If a row does not satisfy a join condition, the row does not appear in the query result. For example, in the equijoin condition of `EMPLOYEES` and `DEPARTMENTS` tables, department ID 190 does not appear because there are no employees with that department ID recorded in the `EMPLOYEES` table. Instead of seeing 20 employees in the result set, you see 19 records.

To return the department record that does not have any employees, you can use an outer join.

INNER Versus OUTER Joins

- In SQL:1999, the join of two tables returning only matched rows is called an inner join.
- A join between two tables that returns the results of the inner join as well as the unmatched rows from the left (or right) tables is called a left (or right) outer join.
- A join between two tables that returns the results of an inner join as well as the results of a left and right join is a full outer join.

INNER Versus OUTER Joins

Joining tables with the `NATURAL JOIN`, `USING`, or `ON` clauses results in an inner join. Any unmatched rows are not displayed in the output. To return the unmatched rows, you can use an outer join. An outer join returns all rows that satisfy the join condition and also returns some or all of those rows from one table for which no rows from the other table satisfy the join condition.

There are three types of outer joins:
- `LEFT OUTER`
- `RIGHT OUTER`
- `FULL OUTER`

LEFT OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e LEFT OUTER JOIN departments d
ON    (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
...		
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		

20 rows selected.

ORACLE

Example of `LEFT OUTER JOIN`

This query retrieves all rows in the `EMPLOYEES` table, which is the left table even if there is no match in the `DEPARTMENTS` table.

RIGHT OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e RIGHT OUTER JOIN departments d
ON     (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
Davies	50	Shipping
...		
Kochhar	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
	190	Contracting

20 rows selected.

ORACLE

Example of RIGHT OUTER JOIN

This query retrieves all rows in the DEPARTMENTS table, which is the right table even if there is no match in the EMPLOYEES table.

FULL OUTER JOIN

```
SELECT e.last_name, d.department_id, d.department_name
FROM   employees e FULL OUTER JOIN departments d
ON   (e.department_id = d.department_id) ;
```

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
...		
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		
	190	Contracting

21 rows selected.

ORACLE

Example of FULL OUTER JOIN

This query retrieves all rows in the EMPLOYEES table, even if there is no match in the DEPARTMENTS table. It also retrieves all rows in the DEPARTMENTS table, even if there is no match in the EMPLOYEES table.

Cartesian Products

- 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.

ORACLE

Cartesian Products

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 the result is rarely useful. You should always include a valid join condition unless you have a specific need to combine all rows from all tables.

Cartesian products are useful for some tests when you need to generate a large number of rows to simulate a reasonable amount of data.

Generating a Cartesian Product

EMPLOYEES (20 rows)

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
…		
202	Fay	20
205	Higgins	110
206	Gietz	110

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:
20 x 8 = 160 rows

EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
100	90	1700
101	90	1700
102	90	1700
103	60	1700
104	60	1700
107	60	1700
…		

160 rows selected.

Cartesian Products (continued)

A Cartesian product is generated if a join condition is omitted. The example in the slide displays employee last name and department name from the EMPLOYEES and DEPARTMENTS tables. Because no join condition 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.

Creating Cross Joins

- The `CROSS JOIN` clause produces the cross-product of two tables.
- This is also called 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

...

160 rows selected.

Creating Cross Joins

The example in the slide produces a Cartesian product of the `EMPLOYEES` and `DEPARTMENTS` tables.

Summary

In this lesson, you should have learned how to use joins to display data from multiple tables by using:

- Equijoins
- Nonequijoins
- Outer joins
- Self-joins
- Cross joins
- Natural joins
- Full (or two-sided) outer joins

ORACLE

Summary

There are multiple ways to join tables.

Types of Joins

- Equijoins
- Nonequijoins
- Outer joins
- Self-joins
- Cross joins
- Natural joins
- Full (or two-sided) outer joins

Cartesian Products

A Cartesian product results in a display of all combinations of rows. This is done by either omitting the `WHERE` clause or specifying the `CROSS JOIN` clause.

Table Aliases

- Table aliases speed up database access.
- Table aliases can help to keep SQL code smaller by conserving memory.

Practice 5: Overview

This practice covers the following topics:

- Joining tables using an equijoin
- Performing outer and self-joins
- Adding conditions

Practice 5: Overview

This practice is intended to give you practical experience in extracting data from more than one table using SQL:1999–compliant joins.

Practice 5

1. Write a query for the HR department to produce the addresses of all the departments. Use the `LOCATIONS` and `COUNTRIES` tables. Show the location ID, street address, city, state or province, and country in the output. Use a `NATURAL JOIN` to produce the results.

LOCATION_ID	STREET_ADDRESS	CITY	STATE_PROVINCE	COUNTRY_NAME
1400	2014 Jabberwocky Rd	Southlake	Texas	United States of America
1500	2011 Interiors Blvd	South San Francisco	California	United States of America
1700	2004 Charade Rd	Seattle	Washington	United States of America
1800	460 Bloor St. W.	Toronto	Ontario	Canada
2500	Magdalen Centre, The Oxford Science Park	Oxford	Oxford	United Kingdom

2. The HR department needs a report of all employees. Write a query to display the last name, department number, and department name for all employees.

LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping
Rajs	50	Shipping
Davies	50	Shipping
Vargas	50	Shipping
...		
De Haan	90	Executive
Higgins	110	Accounting
Gietz	110	Accounting

19 rows selected.

Practice 5 (continued)

3. The HR department needs a report of employees in Toronto. Display the last name, job, department number, and department name for all employees who work in Toronto.

LAST_NAME	JOB_ID	DEPARTMENT_ID	DEPARTMENT_NAME
Hartstein	MK_MAN	20	Marketing
Fay	MK_REP	20	Marketing

4. Create a report to display employees' last name and employee number along with their manager's last name and manager number. Label the columns `Employee`, `Emp#`, `Manager`, and `Mgr#`, respectively. Place your SQL statement in a text file named `lab_05_04.sql`.

Employee	EMP#	Manager	Mgr#
Kochhar	101	King	100
De Haan	102	King	100
Mourgos	124	King	100
Zlotkey	149	King	100
Hartstein	201	King	100
Whalen	200	Kochhar	101
Higgins	205	Kochhar	101
Hunold	103	De Haan	102
Ernst	104	Hunold	103
Lorentz	107	Hunold	103
Rajs	141	Mourgos	124
Davies	142	Mourgos	124
Matos	143	Mourgos	124
Vargas	144	Mourgos	124
Employee	EMP#	Manager	Mgr#
Abel	174	Zlotkey	149
Taylor	176	Zlotkey	149
Grant	178	Zlotkey	149
Fay	202	Hartstein	201
Gietz	206	Higgins	205

19 rows selected.

Practice 5 (continued)

5. Modify `lab_05_04.sql` to display all employees including King, who has no manager. Order the results by the employee number. Place your SQL statement in a text file named `lab_05_05.sql`. Run the query in `lab_05_05.sql`.

Employee	EMP#	Manager	Mgr#
King	100		
Kochhar	101	King	100
De Haan	102	King	100
Hunold	103	De Haan	102
Ernst	104	Hunold	103
Lorentz	107	Hunold	103
Mourgos	124	King	100

...

20 rows selected.

6. Create a report for the HR department that displays employee last names, department numbers, and all the employees who work in the same department as a given employee. Give each column an appropriate label. Save the script to a file named `lab_05_06.sql`.

DEPARTMENT	EMPLOYEE	COLLEAGUE
20	Fay	Hartstein
20	Hartstein	Fay
50	Davies	Matos
50	Davies	Mourgos
50	Davies	Rajs
50	Davies	Vargas
50	Matos	Davies
50	Matos	Mourgos
50	Matos	Rajs
50	Matos	Vargas
50	Mourgos	Davies
50	Mourgos	Matos
50	Mourgos	Rajs
50	Mourgos	Vargas

...

42 rows selected.

Practice 5 (continued)

7. The HR department needs a report on job grades and salaries. To familiarize yourself with the `JOB_GRADES` table, first show the structure of the `JOB_GRADES` table. Then create a query that displays the name, job, department name, salary, and grade for all employees.

Name	Null?	Type
GRADE_LEVEL		VARCHAR2(3)
LOWEST_SAL		NUMBER
HIGHEST_SAL		NUMBER

LAST_NAME	JOB_ID	DEPARTMENT_NAME	SALARY	GRA
Matos	ST_CLERK	Shipping	2600	A
Vargas	ST_CLERK	Shipping	2500	A
Lorentz	IT_PROG	IT	4200	B
Mourgos	ST_MAN	Shipping	5800	B
Rajs	ST_CLERK	Shipping	3500	B
Davies	ST_CLERK	Shipping	3100	B
Whalen	AD_ASST	Administration	4400	B

...

19 rows selected.

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

8. The HR department wants to determine the names of all employees who were hired after Davies. Create a query to display the name and hire date of any employee hired after employee Davies.

LAST_NAME	HIRE_DATE
Lorentz	07-FEB-99
Mourgos	16-NOV-99
Matos	15-MAR-98
Vargas	09-JUL-98
Zlotkey	29-JAN-00
Taylor	24-MAR-98
Grant	24-MAY-99
Fay	17-AUG-97

8 rows selected.

Practice 5 (continued)

9. The HR department needs to find the names and hire dates for all employees who were hired before their managers, along with their managers' names and hire dates. Save the script to a file named `lab5_09.sql`.

LAST_NAME	HIRE_DATE	LAST_NAME	HIRE_DATE
Whalen	17-SEP-87	Kochhar	21-SEP-89
Hunold	03-JAN-90	De Haan	13-JAN-93
Rajs	17-OCT-95	Mourgos	16-NOV-99
Davies	29-JAN-97	Mourgos	16-NOV-99
Matos	15-MAR-98	Mourgos	16-NOV-99
Vargas	09-JUL-98	Mourgos	16-NOV-99
Abel	11-MAY-96	Zlotkey	29-JAN-00
Taylor	24-MAR-98	Zlotkey	29-JAN-00
Grant	24-MAY-99	Zlotkey	29-JAN-00

9 rows selected.

Using Subqueries to Solve Queries

6

Objectives

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

- Define subqueries
- Describe the types of problems that subqueries can solve
- List the types of subqueries
- Write single-row and multiple-row subqueries

Objectives

In this lesson, you 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 Abel's?

Main query:

Which employees have salaries greater than Abel's salary?

Subquery:

What is Abel's salary?

Using a Subquery to Solve a Problem

Suppose you want to write a query to find out who earns a salary greater than Abel's salary.

To solve this problem, you need *two* queries: one to find how much Abel earns, and a second query to find who earns more than that amount.

You can solve this problem by combining the two queries, placing one query *inside* the other query.

The 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.

Subquery Syntax

```
SELECT   select_list
FROM     table
WHERE    expr operator
                   (SELECT      select_list
                    FROM        table);
```

- The subquery (inner query) executes once before the main query (outer query).
- The result of the subquery is used by the main query.

Subquery Syntax

A subquery is a SELECT statement that is embedded in a clause of another SELECT statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

You can place the subquery in a number of SQL clauses, including the following:

- WHERE clause
- HAVING clause
- FROM clause

In the syntax:

operator includes a comparison condition such as >, =, or IN

Note: Comparison conditions fall into two classes: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators (IN, ANY, ALL).

The subquery is often referred to as 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.

Using a Subquery

```
SELECT last_name, salary
FROM   employees      11000
WHERE  salary >
                (SELECT salary
                 FROM   employees
                 WHERE  last_name = 'Abel');
```

LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hartstein	13000
Higgins	12000

ORACLE

Using a Subquery

In the slide, the inner query determines the salary of employee Abel. The outer query takes the result of the inner query and uses this result to display all the employees who earn more than this amount.

Guidelines for Using Subqueries

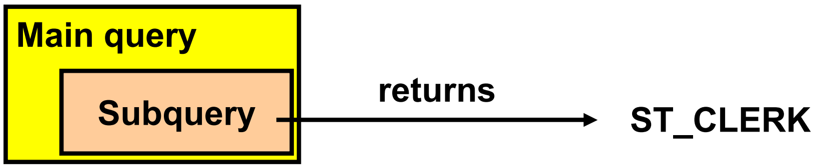
- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition.
- The `ORDER BY` clause in the subquery is not needed unless you are performing Top-N analysis.
- Use single-row operators with single-row subqueries, and use multiple-row operators with multiple-row subqueries.

Guidelines for Using Subqueries

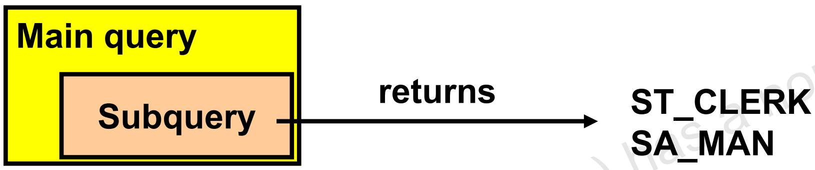
- A subquery must be enclosed in parentheses.
- Place the subquery on the right side of the comparison condition for readability.
- With Oracle8*i* and later releases, an `ORDER BY` clause can be used and is required in the subquery to perform Top-N analysis.
  - Before Oracle8*i*, however, subqueries could not contain an `ORDER BY` clause. Only one `ORDER BY` clause could be used for a `SELECT` statement; if specified, it had to be the last clause in the main `SELECT` statement.
- Two classes of comparison conditions are used in subqueries: single-row operators and multiple-row operators.

Types of Subqueries

- Single-row subquery



- Multiple-row subquery



Types of Subqueries

- **Single-row subqueries:** Queries that return only one row from the inner `SELECT` statement
- **Multiple-row subqueries:** Queries that return more than one row from the inner `SELECT` statement

Note: There are also multiple-column subqueries, which are queries that return more than one column from the inner `SELECT` statement. These are covered in the *Oracle Database 10g: SQL Fundamentals II* course.

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

Single-Row Subqueries

A single-row subquery is one that returns one row from the inner `SELECT` statement. This type of subquery uses a single-row operator. The slide gives a list of single-row operators.

Example

Display the employees whose job ID is the same as that of employee 141:

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id =
                (SELECT job_id
                 FROM   employees
                 WHERE  employee_id = 141);
```

LAST_NAME	JOB_ID
Rajs	ST_CLERK
Davies	ST_CLERK
Matos	ST_CLERK
Vargas	ST_CLERK

Executing Single-Row Subqueries

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id =                 ST_CLERK
                (SELECT job_id
                 FROM   employees
                 WHERE  employee_id = 141)
AND    salary >                    2600
                (SELECT salary
                 FROM   employees
                 WHERE  employee_id = 143);
```

LAST_NAME	JOB_ID	SALARY
Rajs	ST_CLERK	3500
Davies	ST_CLERK	3100

ORACLE

Executing Single-Row Subqueries

A SELECT statement can be considered as a query block. The example in the slide displays employees whose job ID is the same as that of employee 141 and whose salary is greater than that of employee 143.

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 ST_CLERK and 2600, respectively. The outer query block is then processed and uses the values that were returned by the inner queries to complete its search conditions.

Both inner queries return single values (ST_CLERK and 2600, respectively), so this SQL statement is called a single-row subquery.

Note: The outer and inner queries can get data from different tables.

Using Group Functions in a Subquery

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  salary =
                (SELECT MIN(salary)
                 FROM   employees);
```

2500

LAST_NAME	JOB_ID	SALARY
Vargas	ST_CLERK	2500

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. The subquery is in parentheses and is placed after the comparison condition.

The example in the slide displays the employee 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 HAVING Clause with Subqueries

- The Oracle server executes subqueries first.
- The Oracle server returns results into the HAVING clause of the main query.

```
SELECT    department_id, MIN(salary)
FROM      employees
GROUP BY  department_id
HAVING    MIN(salary) >           2500
                        (SELECT MIN(salary)
                         FROM   employees
                         WHERE  department_id = 50);
```


The HAVING Clause with Subqueries

You can use subqueries not only in the WHERE clause but also in the HAVING clause. The Oracle server executes the subquery, and the results are returned into the HAVING clause of the main query.

The SQL statement in the slide displays all the departments that have a minimum salary greater than that of department 50.

DEPARTMENT_ID	MIN(SALARY)
10	4400
20	6000
...	
	7000

7 rows selected.

Example

Find the job with the lowest average salary.

```
SELECT    job_id, AVG(salary)
FROM      employees
GROUP BY  job_id
HAVING    AVG(salary) = (SELECT    MIN(AVG(salary))
                         FROM      employees
                         GROUP BY  job_id);
```

What Is Wrong with This Statement?

```
SELECT employee_id, last_name
FROM   employees
WHERE  salary =
                (SELECT   MIN(salary)
                 FROM     employees
                 GROUP BY department_id);
```

```
ERROR at line 4:
ORA-01427: single-row subquery returns more than
one row
```

Single-row operator with multiple-row subquery

Errors with Subqueries

One common error with subqueries occurs when more than one row is returned for a single-row subquery.

In the SQL statement in the slide, the subquery contains a `GROUP BY` clause, which implies that the subquery will return multiple rows, one for each group that it finds. In this case, the result of the subquery are `4400`, `6000`, `2500`, `4200`, `7000`, `17000`, and `8300`.

The outer query takes those results and uses them in its `WHERE` clause. The `WHERE` clause contains an equal (=) operator, a single-row comparison operator that expects only one value. The = operator cannot accept more than one value from the subquery and, therefore, generates the error.

To correct this error, change the = operator to `IN`.

Will This Statement Return Rows?

```
SELECT last_name, job_id
FROM   employees
WHERE  job_id =
                 (SELECT job_id
                  FROM   employees
                  WHERE  last_name = 'Haas');
```

```
no rows selected
```

Subquery returns no values.

Problems with Subqueries

A common problem with subqueries occurs when no rows are returned by the inner query.

In the SQL statement in the slide, the subquery contains a `WHERE` clause. Presumably, the intention is to find the employee whose name is Haas. The statement is correct but selects no rows when executed.

There is no employee named Haas. So the subquery returns no rows. The outer query takes the results of the subquery (null) and uses these results in its `WHERE` clause. The outer query finds no employee with a job ID equal to null, and so returns no rows. If a job existed with a value of null, the row is not returned because comparison of two null values yields a null; therefore, the `WHERE` condition is not true.

Multiple-Row Subqueries

- Return more than one row
- Use multiple-row comparison operators

Operator	Meaning
`IN`	Equal to any member in the list
`ANY`	Compare value to each value returned by the subquery
`ALL`	Compare value to every value returned by the subquery

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:

```
SELECT last_name, salary, department_id
FROM   employees
WHERE  salary IN (SELECT   MIN(salary)
                  FROM     employees
                  GROUP BY department_id);
```

Example

Find the employees who earn the same salary as the minimum salary for each department.

The inner query is executed first, producing a query result. The main query block is then processed and uses the values that were returned by the inner query to complete its search condition. In fact, the main query appears to the Oracle server as follows:

```
SELECT last_name, salary, department_id
FROM   employees
WHERE  salary IN (2500, 4200, 4400, 6000, 7000, 8300,
 8600, 17000);
```

Using the ANY Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM   employees        9000, 6000, 4200
WHERE  salary < ANY
                    (SELECT salary
                     FROM   employees
                     WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
124	Mourgos	ST_MAN	5800
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

...

10 rows selected.

ORACLE

Multiple-Row Subqueries (continued)

The ANY operator (and its synonym, the SOME operator) compares a value to *each* value returned by a subquery. The slide example displays employees who are not IT programmers and whose salary is less than that of any IT programmer. The maximum salary that a programmer earns is $9,000.

<ANY means less than the maximum. >ANY means more than the minimum. =ANY is equivalent to IN.

Using the `ALL` Operator in Multiple-Row Subqueries

```
SELECT employee_id, last_name, job_id, salary
FROM   employees        9000, 6000, 4200
WHERE  salary < ALL
                    (SELECT salary
                     FROM   employees
                     WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

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

Multiple-Row Subqueries (continued)

The `ALL` operator compares a value to *every* value returned by a subquery. The slide example displays employees whose salary is less than the salary of all employees with a job ID of `IT_PROG` and whose job is not `IT_PROG`.

`>ALL` means more than the maximum, and `<ALL` means less than the minimum.

The `NOT` operator can be used with `IN`, `ANY`, and `ALL` operators.

Null Values in a Subquery

```
SELECT emp.last_name
FROM   employees emp
WHERE  emp.employee_id NOT IN
                              (SELECT mgr.manager_id
                               FROM   employees mgr);

no rows selected
```


Returning Nulls in the Resulting Set of a Subquery

The SQL statement in the slide attempts to display all the employees who do not have any subordinates. Logically, this SQL statement should have returned 12 rows. However, the SQL statement does not return any rows. One of the values returned by the inner query is a null value, and, therefore, the entire query returns no rows.

The reason is that all conditions that compare a null value result in a null. So whenever null values are likely to be part of the results set of a subquery, do not use the `NOT IN` operator. The `NOT IN` operator is equivalent to `<> ALL`.

Notice that the null value as part of the results set of a subquery is not a problem if you use the `IN` operator. The `IN` operator is equivalent to `=ANY`. For example, to display the employees who have subordinates, use the following SQL statement:

```
SELECT emp.last_name
FROM   employees emp
WHERE  emp.employee_id  IN
                          (SELECT mgr.manager_id
                           FROM   employees mgr);
```

Returning Nulls in the Resulting Set of a Subquery (continued)

Alternatively, a WHERE clause can be included in the subquery to display all employees who do not have any subordinates:

```
SELECT last_name FROM employees
WHERE  employee_id NOT IN
                        (SELECT manager_id
                         FROM   employees
                         WHERE  manager_id IS NOT NULL);
```

Summary

In this lesson, you should have learned how to:

- Identify when a subquery can help solve a question
- Write subqueries when a query is based on unknown values

```
SELECT    select_list
FROM      table
WHERE     expr operator
                     (SELECT  select_list
                      FROM    table);
```

ORACLE

Summary

In this lesson, you should have learned how to use subqueries. 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 a search criterion with unknown intermediate values.

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, after which the `WHERE` or `HAVING` clause uses the results
- Can contain group functions

Practice 6: Overview

This practice covers the following topics:

- Creating subqueries to query values based on unknown criteria
- Using subqueries to find out which values exist in one set of data and not in another

Practice 6: Overview

In this practice, you write complex queries to retrieve data from the table using single-row and multiple-row subqueries.

Paper-Based Questions

You may want to create the inner query first for these questions. Make sure that it runs and produces the data that you anticipate before you code the outer query.

Practice 6

1. The HR department needs a query that prompts the user for an employee last name. The query then displays the last name and hire date of any employee in the same department as the employee whose name they supply (excluding that employee). For example, if the user enters `Zlotkey`, find all employees who work with Zlotkey (excluding Zlotkey).

LAST_NAME	HIRE_DATE
Abel	11-MAY-96
Taylor	24-MAR-98

2. Create a report that displays the employee number, last name, and salary of all employees who earn more than the average salary. Sort the results in order of ascending salary.

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000
149	Zlotkey	10500
174	Abel	11000
205	Higgins	12000
201	Hartstein	13000
101	Kochhar	17000
102	De Haan	17000
100	King	24000

8 rows selected.

3. Write a query that displays the employee number and last name of all employees who work in a department with any employee whose last name contains a *u*. Place your SQL statement in a text file named `lab_06_03.sql`. Run your query.

EMPLOYEE_ID	LAST_NAME
124	Mourgos
141	Rajs
142	Davies
143	Matos
144	Vargas
103	Hunold
104	Ernst
107	Lorentz

8 rows selected.

Practice 6 (continued)

4. The HR department needs a report that displays the last name, department number, and job ID of all employees whose department location ID is 1700.

LAST_NAME	DEPARTMENT_ID	JOB_ID
Whalen	10	AD_ASST
King	90	AD_PRES
Kochhar	90	AD_VP
De Haan	90	AD_VP
Higgins	110	AC_MGR
Gietz	110	AC_ACCOUNT

6 rows selected.

Modify the query so that the user is prompted for a location ID. Save this to a file named `lab_06_04.sql`.

5. Create a report for HR that displays the last name and salary of every employee who reports to King.

LAST_NAME	SALARY
Kochhar	17000
De Haan	17000
Mourgos	5800
Zlotkey	10500
Hartstein	13000

6. Create a report for HR that displays the department number, last name, and job ID for every employee in the Executive department.

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

If you have time, complete the following exercise:

7. Modify the query in `lab_06_03.sql` to display the employee number, last name, and salary of all employees who earn more than the average salary and who work in a department with any employee whose last name contains a *u*. Resave `lab_06_03.sql` as `lab_06_07.sql`. Run the statement in `lab_06_07.sql`.

EMPLOYEE_ID	LAST_NAME	SALARY
103	Hunold	9000

Using the Set Operators

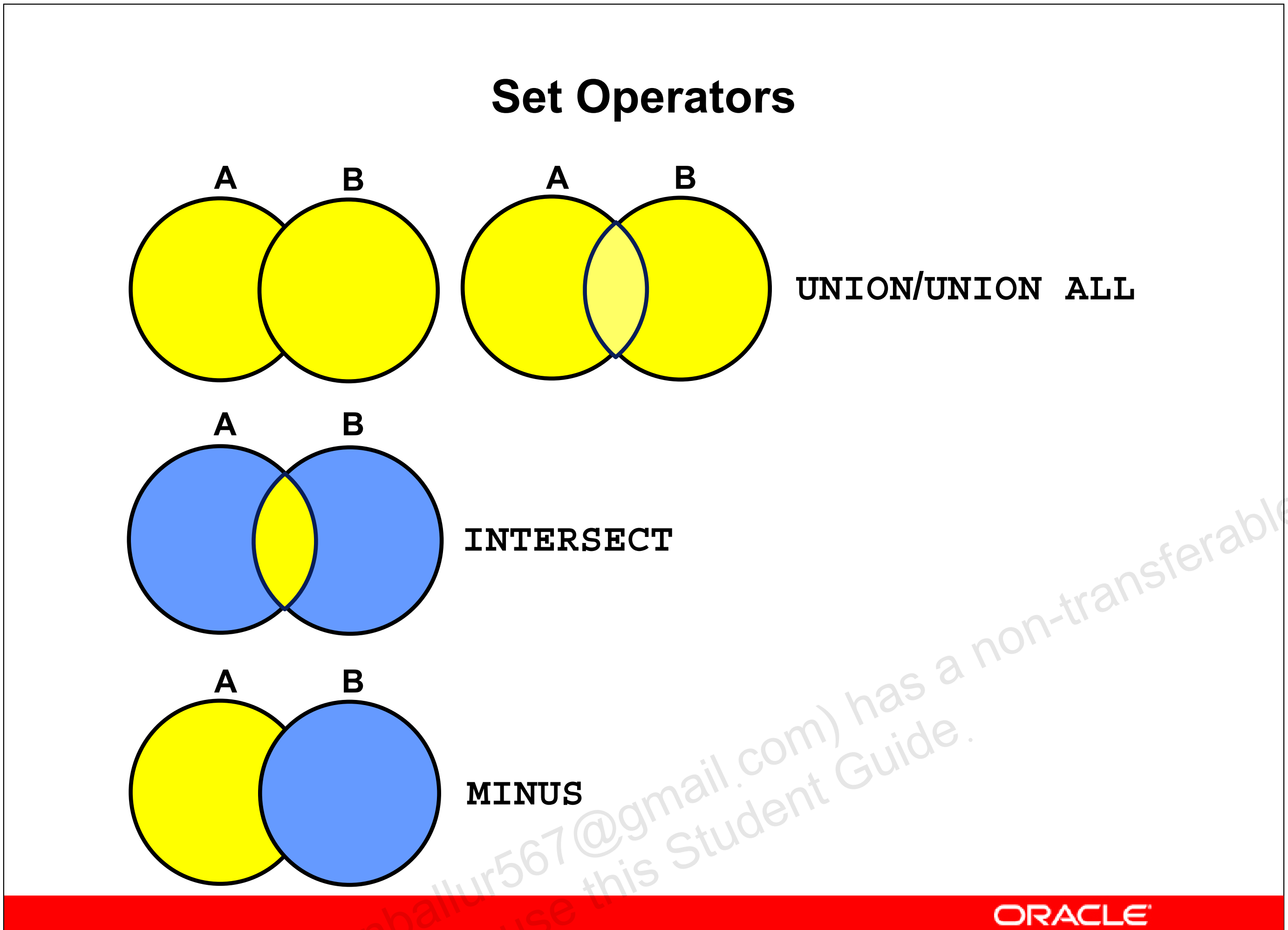
Objectives

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

- Describe set operators
- Use a set operator to combine multiple queries into a single query
- Control the order of rows returned

Objectives

In this lesson, you learn how to write queries by using set operators.



Set Operators

Set operators combine the results of two or more component queries into one result. Queries containing set operators are called *compound queries*.

Operator	Returns
`UNION`	All distinct rows selected by either query
`UNION ALL`	All rows selected by either query, including all duplicates
`INTERSECT`	All distinct rows selected by both queries
`MINUS`	All distinct rows that are selected by the first `SELECT` statement and not selected in the second `SELECT` statement

All set operators have equal precedence. If a SQL statement contains multiple set operators, the Oracle server evaluates them from left (top) to right (bottom) if no parentheses explicitly specify another order. You should use parentheses to specify the order of evaluation explicitly in queries that use the `INTERSECT` operator with other set operators.

Tables Used in This Lesson

The tables used in this lesson are:

- EMPLOYEES: Provides details regarding all current employees
- JOB_HISTORY: Records the details of the start date and end date of the former job, and the job identification number and department when an employee switches jobs

Tables Used in This Lesson

Two tables are used in this lesson. They are the EMPLOYEES table and the JOB_HISTORY table.

The EMPLOYEES table stores the employee details. For the human resource records, this table stores a unique identification number and e-mail address for each employee. The details of the employee's job identification number, salary, and manager are also stored. Some of the employees earn a commission in addition to their salary; this information is tracked, too. The company organizes the roles of employees into jobs. Some of the employees have been with the company for a long time and have switched to different jobs. This is monitored using the JOB_HISTORY table. When an employee switches jobs, the details of the start date and end date of the former job, the job identification number, and the department are recorded in the JOB_HISTORY table.

The structure and data from the EMPLOYEES and JOB_HISTORY tables are shown on the following pages.

Tables Used in This Lesson (continued)

There have been instances in the company of people who have held the same position more than once during their tenure with the company. For example, consider the employee Taylor, who joined the company on 24-MAR-1998. Taylor held the job title `SA_REP` for the period 24-MAR-98 to 31-DEC-98 and the job title `SA_MAN` for the period 01-JAN-99 to 31-DEC-99. Taylor moved back into the job title of `SA_REP`, which is his current job title.

Similarly, consider the employee Whalen, who joined the company on 17-SEP-1987. Whalen held the job title `AD_ASST` for the period 17-SEP-87 to 17-JUN-93 and the job title `AC_ACCOUNT` for the period 01-JUL-94 to 31-DEC-98. Whalen moved back into the job title of `AD_ASST`, which is his current job title.

```
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)

Tables Used in This Lesson (continued)

```
SELECT employee_id, last_name, job_id, hire_date, department_id
FROM employees;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	DEPARTMENT_ID
100	King	AD_PRES	17-JUN-87	90
101	Kochhar	AD_VP	21-SEP-89	90
102	De Haan	AD_VP	13-JAN-93	90
103	Hunold	IT_PROG	03-JAN-90	60
104	Ernst	IT_PROG	21-MAY-91	60
107	Lorentz	IT_PROG	07-FEB-99	60
124	Mourgos	ST_MAN	16-NOV-99	50
141	Rajs	ST_CLERK	17-OCT-95	50
142	Davies	ST_CLERK	29-JAN-97	50
143	Matos	ST_CLERK	15-MAR-98	50
144	Vargas	ST_CLERK	09-JUL-98	50
149	Zlotkey	SA_MAN	29-JAN-00	80
174	Abel	SA_REP	11-MAY-96	80
176	Taylor	SA_REP	24-MAR-98	80
EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	DEPARTMENT_ID
178	Grant	SA_REP	24-MAY-99	
200	Whalen	AD_ASST	17-SEP-87	10
201	Hartstein	MK_MAN	17-FEB-96	20

...

20 rows selected.

```
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)

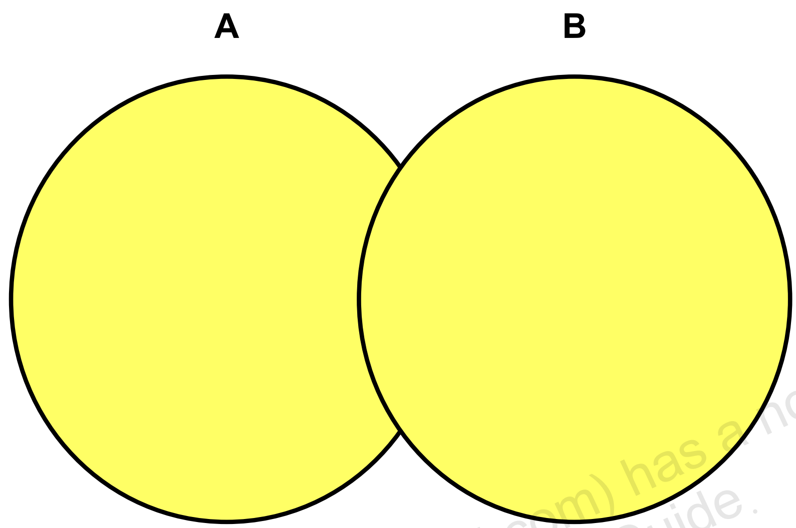
Tables Used in This Lesson (continued)

```
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.

UNION Operator



The UNION operator returns results from both queries after eliminating duplications.

UNION Operator

The UNION operator returns all rows that are selected by either query. Use the UNION operator to return all rows from multiple tables and eliminate any duplicate rows.

Guidelines

- The number of columns and the data types of the columns being selected must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.
- UNION operates over all of the columns being selected.
- NULL values are not ignored during duplicate checking.
- The IN operator has a higher precedence than the UNION operator.
- By default, the output is sorted in ascending order of the first column of the SELECT clause.

Using the UNION Operator

Display the current and previous job details of all employees. Display each employee only once.

```
SELECT employee_id, job_id
FROM   employees
UNION
SELECT employee_id, job_id
FROM   job_history;
```

EMPLOYEE_ID	JOB_ID
100	AD_PRES
101	AC_ACCOUNT
...	
200	AC_ACCOUNT
200	AD_ASST
...	
205	AC_MGR
206	AC_ACCOUNT

ORACLE

Using the UNION Operator

The UNION operator eliminates any duplicate records. If records that occur in both the EMPLOYEES and the JOB_HISTORY tables are identical, the records are displayed only once. Observe in the output shown in the slide that the record for the employee with the EMPLOYEE_ID 200 appears twice because the JOB_ID is different in each row.

Consider the following example:

```
SELECT   employee_id, job_id, department_id
FROM     employees
UNION
SELECT   employee_id, job_id, department_id
FROM     job_history;
```

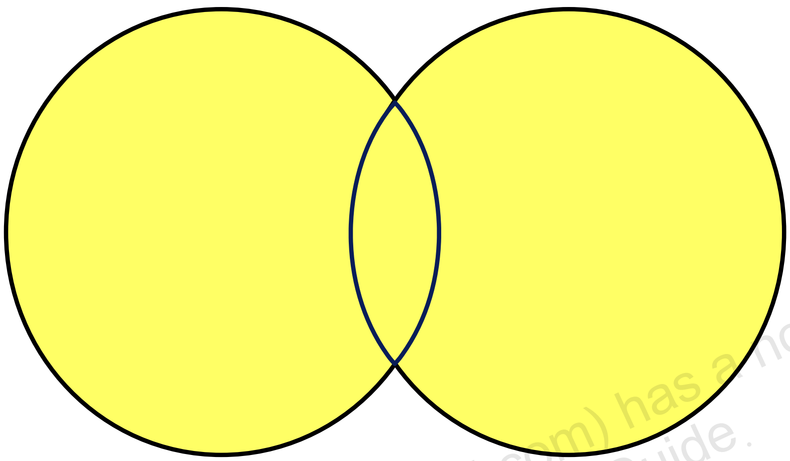
EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
...		
200	AC_ACCOUNT	90
200	AD_ASST	10
200	AD_ASST	90
...		

29 rows selected.

Using the `UNION` Operator (continued)

In the preceding output, employee 200 appears three times. Why? Notice the DEPARTMENT_ID values for employee 200. One row has a DEPARTMENT_ID of 90, another 10, and the third 90. Because of these unique combinations of job IDs and department IDs, each row for employee 200 is unique and therefore not considered to be a duplicate. Observe that the output is sorted in ascending order of the first column of the SELECT clause (in this case, EMPLOYEE_ID).

UNION ALL Operator



The UNION ALL operator returns results from both queries, including all duplications.

UNION ALL Operator

Use the UNION ALL operator to return all rows from multiple queries.

Guidelines

The guidelines for UNION and UNION ALL are the same, with the following two exceptions that pertain to UNION ALL:

- Unlike UNION, duplicate rows are not eliminated and the output is not sorted by default.
- The DISTINCT keyword cannot be used.

Using the UNION ALL Operator

Display the current and previous departments of all employees.

```
SELECT employee_id, job_id, department_id
FROM   employees
UNION ALL
SELECT employee_id, job_id, department_id
FROM   job_history
ORDER BY  employee_id;
```

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
100	AD_PRES	90
101	AD_VP	90
...		
200	AD_ASST	10
200	AD_ASST	90
200	AC_ACCOUNT	90
...		
205	AC_MGR	110
206	AC_ACCOUNT	110

30 rows selected.

ORACLE

UNION ALL Operator (continued)

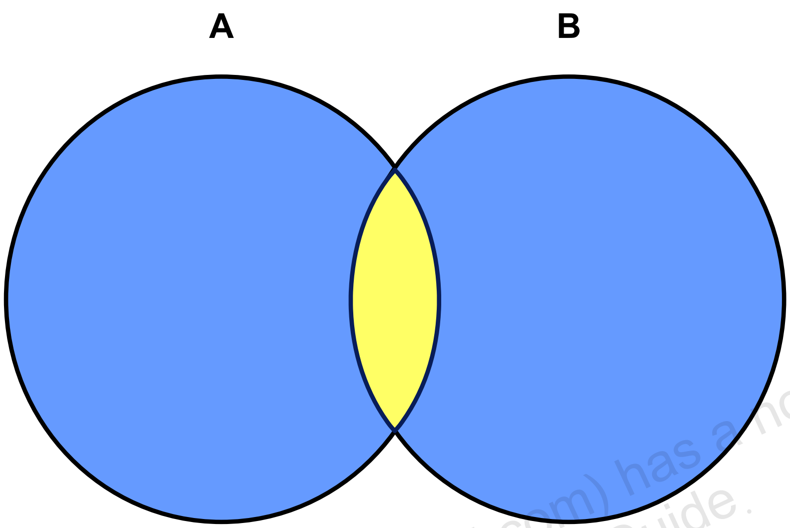
In the example, 30 rows are selected. The combination of the two tables totals to 30 rows. The UNION ALL operator does not eliminate duplicate rows. UNION returns all distinct rows selected by either query. UNION ALL returns all rows selected by either query, including all duplicates. Consider the query on the slide, now written with the UNION clause:

```
SELECT   employee_id, job_id,department_id
FROM     employees
UNION
SELECT   employee_id, job_id,department_id
FROM     job_history
ORDER BY employee_id;
```

The preceding query returns 29 rows. This is because it eliminates the following row (because it is a duplicate):

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
176	SA_REP	80

INTERSECT Operator



The INTERSECT operator returns rows that are common to both queries.

INTERSECT Operator

Use the `INTERSECT` operator to return all rows that are common to multiple queries.

Guidelines

- The number of columns and the data types of the columns being selected by the `SELECT` statements in the queries must be identical in all the `SELECT` statements used in the query. The names of the columns need not be identical.
- Reversing the order of the intersected tables does not alter the result.
- `INTERSECT` does not ignore `NULL` values.

Using the INTERSECT Operator

Display the employee IDs and job IDs of those employees who currently have a job title that is the same as their job title when they were initially hired (that is, they changed jobs but have now gone back to doing their original job).

```
SELECT employee_id, job_id
FROM   employees
INTERSECT
SELECT employee_id, job_id
FROM   job_history;
```

EMPLOYEE_ID	JOB_ID
176	SA_REP
200	AD_ASST

INTERSECT Operator (continued)

In the example in this slide, the query returns only the records that have the same values in the selected columns in both tables.

What will be the results if you add the DEPARTMENT_ID column to the SELECT statement from the EMPLOYEES table and add the DEPARTMENT_ID column to the SELECT statement from the JOB_HISTORY table and run this query? The results may be different because of the introduction of another column whose values may or may not be duplicates.

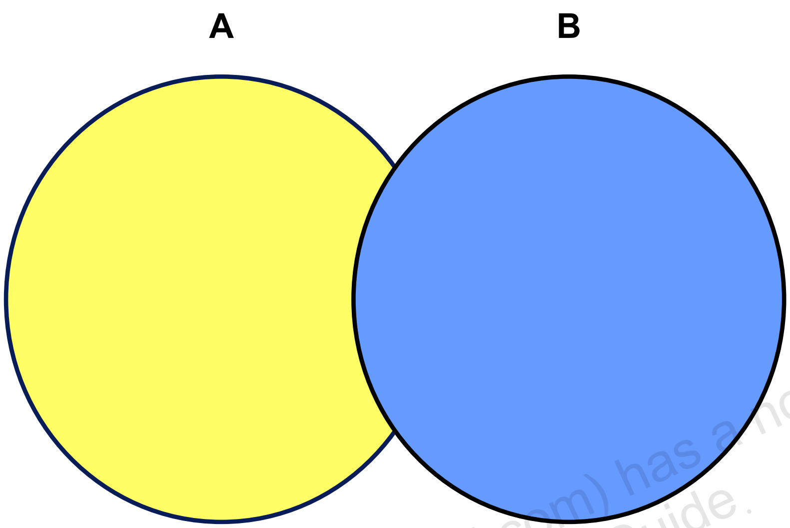
Example

```
SELECT employee_id, job_id, department_id
FROM   employees
INTERSECT
SELECT employee_id, job_id, department_id
FROM   job_history;
```

EMPLOYEE_ID	JOB_ID	DEPARTMENT_ID
176	SA_REP	80

Employee 200 is no longer part of the results because the EMPLOYEES.DEPARTMENT_ID value is different from the JOB_HISTORY.DEPARTMENT_ID value.

MINUS Operator



The MINUS operator returns rows in the first query that are not present in the second query.

MINUS Operator

Use the MINUS operator to return rows returned by the first query that are not present in the second query (the first SELECT statement MINUS the second SELECT statement).

Note: The number of columns and the data types of the columns being selected by the SELECT statements in the queries must be identical in all the SELECT statements used in the query. The names of the columns need not be identical.

MINUS Operator

Display the employee IDs of those employees who have not changed their jobs even once.

```
SELECT employee_id
FROM   employees
MINUS
SELECT employee_id
FROM   job_history;
```

EMPLOYEE_ID
100
103
104
107
...
205
206

15 rows selected.

ORACLE

MINUS Operator (continued)

In the example in the slide, the employee IDs in the JOB_HISTORY table are subtracted from those in the EMPLOYEES table. The results set displays the employees remaining after the subtraction; they are represented by rows that exist in the EMPLOYEES table but do not exist in the JOB_HISTORY table. These are the records of the employees who have not changed their jobs even once.

Set Operator Guidelines

- The expressions in the SELECT lists must match in number and data type.
- Parentheses can be used to alter the sequence of execution.
- The ORDER BY clause:
  - Can appear only at the very end of the statement
  - Will accept the column name, aliases from the first SELECT statement, or the positional notation

Set Operator Guidelines

- The expressions in the select lists of the queries must match in number and data type. Queries that use UNION, UNION ALL, INTERSECT, and MINUS operators in their WHERE clause must have the same number and type of columns in their SELECT list. For example:

```
SELECT employee_id, department_id
FROM   employees
WHERE  (employee_id, department_id)
       IN (SELECT  employee_id, department_id
           FROM    employees
           UNION
           SELECT  employee_id, department_id
           FROM    job_history);
```

- The ORDER BY clause:
  - Can appear only at the very end of the statement
  - Will accept the column name, an alias, or the positional notation
- The column name or alias, if used in an ORDER BY clause, must be from the first SELECT list.
- Set operators can be used in subqueries.

The Oracle Server and Set Operators

- Duplicate rows are automatically eliminated except in `UNION ALL`.
- Column names from the first query appear in the result.
- The output is sorted in ascending order by default except in `UNION ALL`.

The Oracle Server and Set Operators

When a query uses set operators, the Oracle server eliminates duplicate rows automatically except in the case of the `UNION ALL` operator. The column names in the output are decided by the column list in the first `SELECT` statement. By default, the output is sorted in ascending order of the first column of the `SELECT` clause.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and data type. If component queries select character data, the data type of the return values is determined as follows:

- If both queries select values of data type `CHAR`, the returned values have data type `CHAR`.
- If either or both of the queries select values of data type `VARCHAR2`, the returned values have data type `VARCHAR2`.

Matching the SELECT Statements

Using the UNION operator, display the department ID, location, and hire date for all employees.

```
SELECT department_id, TO_NUMBER(null)
       location, hire_date
FROM   employees
UNION
SELECT department_id, location_id,  TO_DATE(null)
FROM   departments;
```

DEPARTMENT_ID	LOCATION	HIRE_DATE
10	1700	
10		17-SEP-87
20	1800	
20		17-FEB-96
...		
110	1700	
110		07-JUN-94
190	1700	
		24-MAY-99

27 rows selected.

Matching the SELECT Statements

Because the expressions in the select lists of the queries must match in number, you can use dummy columns and the data type conversion functions to comply with this rule. In the slide, the name `location` is given as the dummy column heading. The `TO_NUMBER` function is used in the first query to match the `NUMBER` data type of the `LOCATION_ID` column retrieved by the second query. Similarly, the `TO_DATE` function in the second query is used to match the `DATE` data type of the `HIRE_DATE` column retrieved by the first query.

Matching the SELECT Statement: Example

Using the UNION operator, display the employee ID, job ID, and salary of all employees.

```
SELECT employee_id, job_id,salary
FROM   employees
UNION
SELECT employee_id, job_id,0
FROM   job_history;
```

EMPLOYEE_ID	JOB_ID	SALARY
100	AD_PRES	24000
101	AC_ACCOUNT	0
101	AC_MGR	0
...		
205	AC_MGR	12000
206	AC_ACCOUNT	8300

30 rows selected.

Matching the SELECT Statement: Example

The EMPLOYEES and JOB_HISTORY tables have several columns in common (for example, EMPLOYEE_ID, JOB_ID, and DEPARTMENT_ID). But what if you want the query to display the employee ID, job ID, and salary using the UNION operator, knowing that the salary exists only in the EMPLOYEES table?

The code example in the slide matches the EMPLOYEE_ID and JOB_ID columns in the EMPLOYEES and JOB_HISTORY tables. A literal value of 0 is added to the JOB_HISTORY SELECT statement to match the numeric SALARY column in the EMPLOYEES SELECT statement.

In the preceding results, each row in the output that corresponds to a record from the JOB_HISTORY table contains a 0 in the SALARY column.

Controlling the Order of Rows

Produce an English sentence using two `UNION` operators.

```
COLUMN a_dummy NOPRINT
SELECT 'sing' AS "My dream", 3 a_dummy
FROM dual
UNION
SELECT 'I''d like to teach', 1 a_dummy
FROM dual
UNION
SELECT 'the world to', 2 a_dummy
FROM dual
ORDER BY a_dummy;
```

My dream
I'd like to teach
the world to
sing

Controlling the Order of Rows

By default, the output is sorted in ascending order on the first column. You can use the `ORDER BY` clause to change this.

The `ORDER BY` clause can be used only once in a compound query. If used, the `ORDER BY` clause must be placed at the end of the query. The `ORDER BY` clause accepts the column name or an alias. Without the `ORDER BY` clause, the code example in the slide produces the following output in the alphabetical order of the first column:

My dream
I'd like to teach
sing
the world to

Note: Consider a compound query where the `UNION` set operator is used more than once. In this case, the `ORDER BY` clause can use only positions rather than explicit expressions.

The *i*SQL*Plus COLUMN Command

You can use the *i*SQL*Plus COLUMN command to customize column headings.

Syntax:

```
COL[UMN] [{column|alias} [option]]
```

Where OPTION is:

CLE[AR]: Clears any column formats

HEA[DING] *text*: Sets the column heading

FOR[MAT] *format*: Changes the display of the column using a format model

NOPRINT | PRINT: Suppresses or displays the column heading and data

NULL

The following statement suppresses the column data and title heading for the column named A_DUMMY. Notice that the first SELECT clause in the previous slide creates a dummy column named A_DUMMY.

```
COLUMN a_dummy NOPRINT
```

Summary

In this lesson, you should have learned how to:

- Use `UNION` to return all distinct rows
- Use `UNION ALL` to return all rows, including duplicates
- Use `INTERSECT` to return all rows that are shared by both queries
- Use `MINUS` to return all distinct rows that are selected by the first query but not by the second
- Use `ORDER BY` only at the very end of the statement

Summary

- The `UNION` operator returns all rows selected by either query. Use the `UNION` operator to return all rows from multiple tables and eliminate any duplicate rows.
- Use the `UNION ALL` operator to return all rows from multiple queries. Unlike the case with the `UNION` operator, duplicate rows are not eliminated and the output is not sorted by default.
- Use the `INTERSECT` operator to return all rows that are common to multiple queries.
- Use the `MINUS` operator to return rows returned by the first query that are not present in the second query.
- Remember to use the `ORDER BY` clause only at the very end of the compound statement.
- Make sure that the corresponding expressions in the `SELECT` lists match in number and data type.

Practice 7: Overview

In this practice, you use the set operators to create reports:

- Using the `UNION` operator
- Using the `INTERSECTION` operator
- Using the `MINUS` operator

Practice 7: Overview

In this practice, you write queries using the set operators.

Practice 7

1. The HR department needs a list of department IDs for departments that do not contain the job ID `ST_CLERK`. Use set operators to create this report.

DEPARTMENT_ID
10
20
60
80
90
110
190

7 rows selected.

2. The HR department needs a list of countries that have no departments located in them. Display the country ID and the name of the countries. Use set operators to create this report.

CO	COUNTRY_NAME
DE	Germany

3. Produce a list of jobs for departments 10, 50, and 20, in that order. Display job ID and department ID using set operators.

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

4. Create a report that lists the employee IDs and job IDs of those employees who currently have a job title that is the same as their job title when they were initially hired by the company (that is, they changed jobs but have now gone back to doing their original job).

EMPLOYEE_ID	JOB_ID
176	SA_REP
200	AD_ASST

Practice 7 (continued)

5. The HR department needs a report with the following specifications:
   - Last name and department ID of all the employees from the `EMPLOYEES` table, regardless of whether or not they belong to a department
   - Department ID and department name of all the departments from the `DEPARTMENTS` table, regardless of whether or not they have employees working in them

   Write a compound query to accomplish this.

LAST_NAME	DEPARTMENT_ID	TO_CHAR(NULL)
Abel	80	
Davies	50	
De Haan	90	
Ernst	60	
Fay	20	
Gietz	110	
Grant		
Hartstein	20	
Higgins	110	
Hunold	60	
King	90	
Kochhar	90	
Lorentz	60	
Matos	50	
Mourgos	50	
Rajs	50	
Taylor	80	
Vargas	50	
Whalen	10	
Zlotkey	80	
	10	Administration
	20	Marketing
	50	Shipping
	60	IT
	80	Sales
	90	Executive
	110	Accounting
	190	Contracting

28 rows selected.

8

Manipulating Data

Objectives

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

- Describe each data manipulation language (DML) statement
- Insert rows into a table
- Update rows in a table
- Delete rows from a table
- Control transactions

Objective

In this lesson, you learn how to use DML statements to insert rows into a table, update existing rows in a table, and delete existing rows from a table. You 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.

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.

Adding a New Row to a Table

New row

70	Public Relations	100	1700

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

Insert new row into 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
70	Public Relations	100	1700

ORACLE

Adding a New Row to a Table

The graphic in the slide illustrates adding a new department to the `DEPARTMENTS` table.

INSERT Statement Syntax

- Add new rows to a table by using the `INSERT` statement:

```
INSERT INTO   table [(column [, column...])]
VALUES        (value [, value...]);
```

- With this syntax, only one row is inserted at a time.

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 in single quotation marks.

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.

```
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)

For clarity, use the column list in the `INSERT` clause.

Enclose character and date values in single quotation marks; it is not recommended that you enclose numeric values in single quotation marks.

Number values should not be enclosed in single quotation marks, because implicit conversion may take place for numeric values that are assigned to `NUMBER` data type columns if single quotation marks 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.
```


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 you can use null values in the targeted column 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.
```


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;
```

EMPLOYEE_ID	LAST_NAME	JOB_ID	HIRE_DATE	COMMISSION_PCT
113	Popp	AC_ACCOUNT	27-SEP-01	

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.
```

- Verify your addition.

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_P
114	Den	Raphealy	DRAPHEAL	515.127.4561	03-FEB-99	AC_ACCOUNT	11000	

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 in the slide records information for employee Raphealy in the EMPLOYEES table. It sets the HIRE_DATE column to be February 3, 1999. If you use the following statement instead of the one shown in 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

- Use `&` substitution in a SQL statement to prompt for values.
- `&` is a placeholder for the variable value.

```
INSERT INTO departments
            (department_id, department_name, location_id)
VALUES      (&department_id, '&department_name',&location);
```

Input Required

Enter value for department_id:	40	Cancel	Continue
Enter value for department_name:	Human Resources	Cancel	Continue
Enter value for location:	2500	Cancel	Continue

```
1 row created.
```

ORACLE

Creating a Script to Manipulate Data

You can save commands with substitution variables to a file and execute the commands in the file. The slide example records information for a department in the `DEPARTMENTS` table.

Run the script file and you are prompted for input for each of the `&` substitution variables. After entering a value for the substitution variable, click the Continue button. The values that you input are then substituted into the statement. This enables you to run the same script file over and over but supply a different set of values each time you run it.

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 to 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 of the rows of a table, use `SELECT *` in the subquery:

```
INSERT INTO copy_emp
  SELECT *
  FROM   employees;
```

For more information, see "`SELECT`" ("subqueries" section) in the *Oracle Database SQL Reference*.

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.
```


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.

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 if the `INSERT` statement is to work successfully. For example, you could not put in a duplicate employee ID or omit a value for a mandatory not-null column.

Using a Subquery in an `INSERT` Statement

Verify the results:

```
SELECT employee_id, last_name, email, hire_date,
       job_id, salary, department_id
FROM   employees
WHERE  department_id = 50;
```

EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
124	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50
141	Rajs	TRAJS	17-OCT-95	ST_CLERK	3500	50
142	Davies	CDAVIES	29-JAN-97	ST_CLERK	3100	50
143	Matos	RMATOS	15-MAR-98	ST_CLERK	2600	50
144	Vargas	PVARGAS	09-JUL-98	ST_CLERK	2500	50
99999	Taylor	DTAYLOR	07-JUN-99	ST_CLERK	5000	50

6 rows selected.

Using a Subquery in an `INSERT` Statement (continued)

The example shows the results of the subquery that was used to identify the table for the `INSERT` statement.

Changing Data in a Table

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSION_P
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	60	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	60	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	60	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

Update rows in the EMPLOYEES table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID	COMMISSIO
100	Steven	King	SKING	17-JUN-87	AD_PRES	24000	90	
101	Neena	Kochhar	NKOCHHAR	21-SEP-89	AD_VP	17000	90	
102	Lex	De Haan	LDEHAAN	13-JAN-93	AD_VP	17000	90	
103	Alexander	Hunold	AHUNOLD	03-JAN-90	IT_PROG	9000	30	
104	Bruce	Ernst	BERNST	21-MAY-91	IT_PROG	6000	30	
107	Diana	Lorentz	DLORENTZ	07-FEB-99	IT_PROG	4200	30	
124	Kevin	Mourgos	KMOURGOS	16-NOV-99	ST_MAN	5800	50	

ORACLE

Changing Data in a Table

The slide illustrates changing the department number for employees in department 60 to department 30.

`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).

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 “`UPDATE`” in the *Oracle Database SQL Reference*.

Note: In general, use the primary key to identify a single row. Using other columns can 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;
22 rows updated.
```


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.

```
SELECT last_name, department_id
FROM   copy_emp;
```

LAST_NAME	DEPARTMENT_ID
King	110
Kochhar	110
De Haan	110
Hunold	110
Ernst	110
Lorentz	110

...

22 rows selected.

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 salary 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, the 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.
```

ORACLE

Updating Rows Based on Another Table

You can use subqueries in `UPDATE` statements to update rows in a table. The example in the slide updates the `COPY_EMP` table based on the values from the `EMPLOYEES` table. It changes the department number of all employees with employee 200's job ID to employee 100's current department number.

Removing a Row from a Table

DEPARTMENTS

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing		
100	Finance		
50	Shipping	124	1500
60	IT	103	1400

Delete a row from the `DEPARTMENTS` table:

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing		
50	Shipping	124	1500
60	IT	103	1400

ORACLE

Removing a Row from a Table

The graphic in the slide removes the Finance department from the `DEPARTMENTS` table (assuming that there are no constraints defined on the `DEPARTMENTS` table).

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, the message "`0 rows deleted`" is returned.

For more information, see "`DELETE`" in the *Oracle Database SQL Reference*.

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;
22 rows deleted.
```


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 in 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 in 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.

`TRUNCATE` Statement

- Removes all rows from a table, leaving the table empty and the table structure intact
- Is a data definition language (DDL) statement rather than a DML statement; cannot easily be undone
- Syntax:

```
TRUNCATE TABLE table_name;
```

- Example:

```
TRUNCATE TABLE copy_emp;
```


`TRUNCATE` Statement

A more efficient method of emptying a table is with the `TRUNCATE` statement.

You can use the `TRUNCATE` statement to quickly remove all rows from a table or cluster. Removing rows with the `TRUNCATE` statement is faster than removing them with the `DELETE` statement for the following reasons:

- The `TRUNCATE` statement is a data definition language (DDL) statement and generates no rollback information. Rollback information is covered later in this lesson.
- Truncating a table does not fire the delete triggers of the table.
- If the table is the parent of a referential integrity constraint, you cannot truncate the table. You need to disable the constraint before issuing the `TRUNCATE` statement. Disabling constraints is covered in a subsequent lesson.

Database Transactions

A database transaction consists of one of the following:

- DML statements that constitute one consistent change to the data
- One DDL statement
- One data control language (DCL) statement

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.

Transaction Types

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

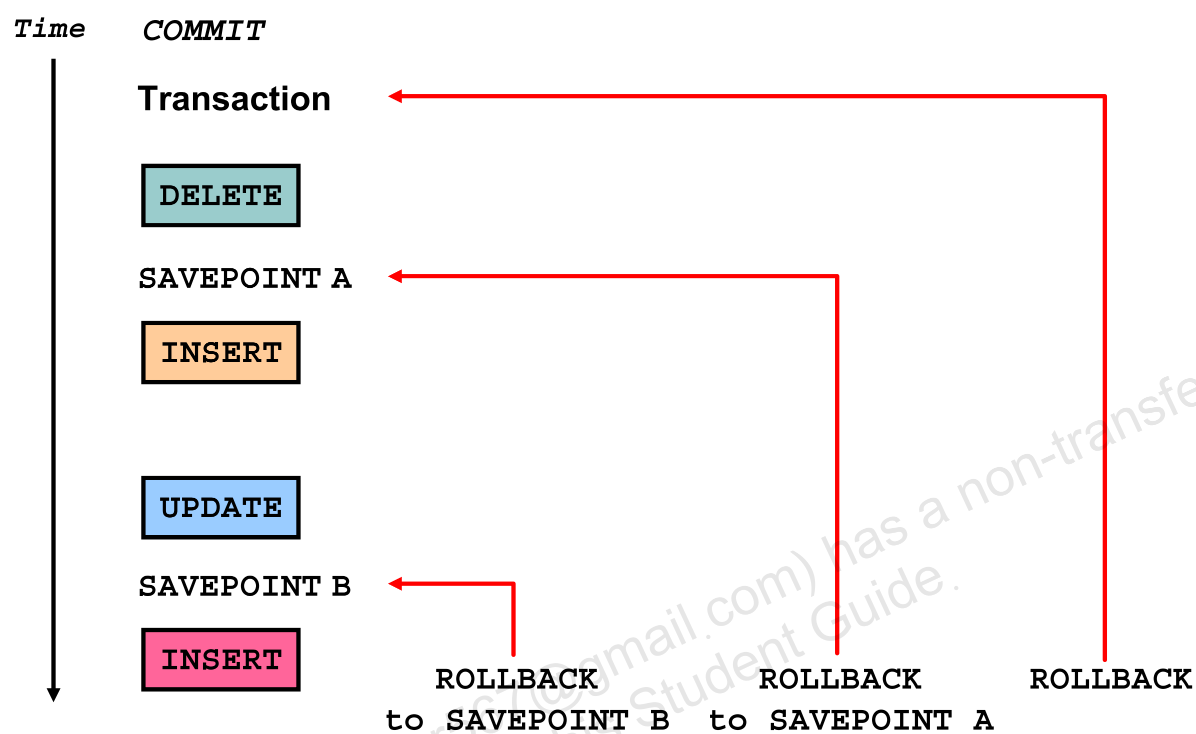
With COMMIT and ROLLBACK statements, you can:

- Ensure data consistency
- Preview data changes before making changes permanent
- Group logically related operations

Advantages of COMMIT and ROLLBACK

With the COMMIT and ROLLBACK statements, you have control over making changes to the data permanent.

Controlling Transactions



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 that were 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. Because savepoints are logical, there is no way to list the savepoints that you have created.

Note: SAVEPOINT is not ANSI standard SQL.

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 Marker

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.

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

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 *i*SQL*Plus.

Implicit Transaction Processing (continued)

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 the Exit button. With 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 in the affected rows.

Committing Changes

Every data change made during the transaction is temporary until the transaction is committed.

The state of the data before `COMMIT` or `ROLLBACK` statements are issued can be described as follows:

- 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.

Committing Changes (continued)

Make all pending changes permanent by using the `COMMIT` statement. Here is what happens after a `COMMIT` statement:

- Data changes are written to the database.
- The previous state of the data is no longer available with normal SQL queries.
- 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 created.
```

- Commit the changes:

```
COMMIT;
Commit complete.
```

ORACLE

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, and update a row in the EMPLOYEES table. Make the data change permanent.

```
DELETE FROM departments
WHERE  department_id IN (290, 300);
1 row deleted.

 UPDATE  employees
   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;
20 rows deleted.
ROLLBACK ;
Rollback complete.
```


Rolling Back Changes

Discard all pending changes by using the ROLLBACK statement, which results in the following:

- Data changes are undone.
- The previous state of the data is restored.
- Locks on the affected rows are released.

State of the Data After `ROLLBACK`

```
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.
```


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.

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.

Statement-Level Rollback

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.

The Oracle server issues an implicit commit before and after any 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:

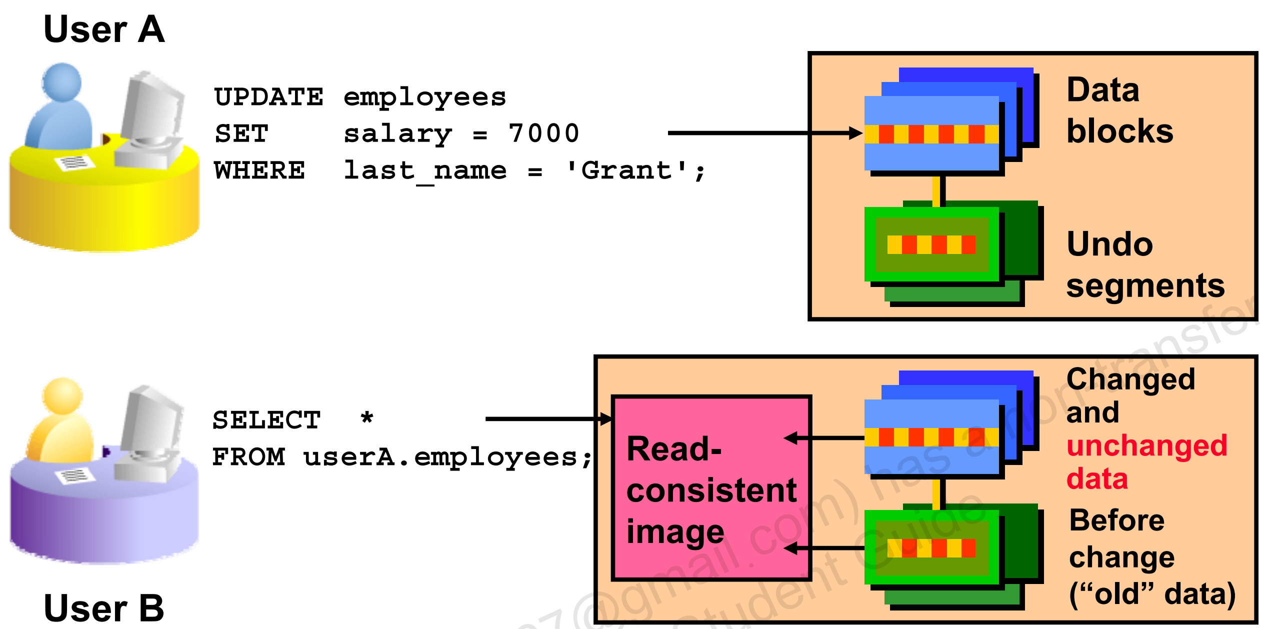
- Read operations (`SELECT` statement)
- Write operations (`INSERT`, `UPDATE`, `DELETE` statements)

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 that 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.

Implementation of Read Consistency



Implementation of Read Consistency

Read consistency is an automatic implementation. It keeps a partial copy of the database in undo segments. The read-consistent image is constructed from committed data from the table and old data being changed and not yet committed from the undo segment.

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 an *undo segment*.

All readers, except the one who issued the change, still see the database as it existed before the changes started; they view the undo 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 undo 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 issuing a select statement *after* the commit is done. The space occupied by the *old* data in the undo 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 undo segment is written back to the table.
- All users see the database as it existed before the transaction began.

Summary

In this lesson, you should have learned how to use the following statements:

Function	Description
INSERT	Adds a new row to the table
UPDATE	Modifies existing rows in the table
DELETE	Removes existing rows from the table
COMMIT	Makes all pending changes permanent
SAVEPOINT	Is used to roll back to the savepoint marker
ROLLBACK	Discards all pending data changes

Summary

In this lesson, you should have learned how to manipulate data in the Oracle database by using the `INSERT`, `UPDATE`, and `DELETE` statements, as well as how to control data changes by using the `COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements.

The Oracle server guarantees a consistent view of data at all times.

Practice 8: Overview

This practice covers the following topics:

- Inserting rows into the tables
- Updating and deleting rows in the table
- Controlling transactions

Practice 8: Overview

In this practice, you add rows to the MY_EMPLOYEE table, update and delete data from the table, and control your transactions.

Practice 8

The HR department wants you to create SQL statements to insert, update, and delete employee data. As a prototype, you use the `MY_EMPLOYEE` table, before giving the statements to the HR department.

Insert data into the `MY_EMPLOYEE` table.

1. Run the statement in the `lab_08_01.sql` script to build the `MY_EMPLOYEE` table to 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. Create an `INSERT` statement to 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. *Do not enter all rows yet.*

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 8 (continued)

6. Write an insert statement in a dynamic reusable script 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 user ID. Save this script to a file named `lab_08_06.sql`.
7. Populate the table with the next two rows of sample data listed in step 3 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

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 $1,000 for all employees who have 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

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

Practice 8 (continued)

15. Commit all pending changes.

Control data transaction to the `MY_EMPLOYEE` table.

16. Populate the table with the last row of sample data listed in step 3 by using the statements in the script that you created in step 6. Run the statements in the script.
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

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 new row is still intact.

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

23. Make the data addition permanent.

Using DDL Statements to Create and Manage Tables

Objectives

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

- Categorize the main database objects
- Review the table structure
- List the data types that are available for columns
- Create a simple table
- Explain how constraints are created at the time of table creation
- Describe how schema objects work

Objectives

In this lesson, you are introduced to the data definition language (DDL) statements. You are taught the basics of how to create simple tables, alter them, and remove them. The data types available in DDL are shown, and schema concepts are introduced. Constraints are tied into this lesson. Exception messages that are generated from violating constraints during DML are shown and explained.

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

Database Objects

An Oracle database can contain multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.

- **Table:** Stores data
- **View:** Subset of data from one or more tables
- **Sequence:** Generates numeric values
- **Index:** Improves the performance of some queries
- **Synonym:** Gives alternative names to objects

Oracle Table Structures

- Tables can be created at any time, even while users are using the database.
- You do not need to specify the size of a table. The size is ultimately defined by the amount of space allocated to the database as a whole. It is important, however, to estimate how much space a table will use over time.
- Table structure can be modified online.

Note: More database objects are available but are not covered in this course.

Naming Rules

Table names and column names:

- Must begin with a letter
- Must be 1–30 characters long
- Must contain only A–Z, a–z, 0–9, _, $, and #
- Must not duplicate the name of another object owned by the same user
- Must not be an Oracle server–reserved word

ORACLE

Naming Rules

You name database tables and columns according to the standard rules for naming any Oracle database object:

- Table names and column names must begin with a letter and be 1–30 characters long.
- Names must contain only the characters A–Z, a–z, 0–9, _ (underscore), $, and # (legal characters, but their use is discouraged).
- Names must not duplicate the name of another object owned by the same Oracle server user.
- Names must not be an Oracle server–reserved word.

Naming Guidelines

Use descriptive names for tables and other database objects.

Note: Names are case-insensitive. For example, `EMPLOYEES` is treated as the same name as `eMPloyees` or `eMpLOYEES`.

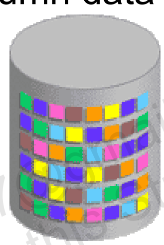
For more information, see “Object Names and Qualifiers” in the *Oracle Database SQL Reference.*

CREATE TABLE Statement

- You must have:
  - CREATE TABLE privilege
  - A storage area

```
CREATE TABLE [schema.]table
          (column datatype [DEFAULT expr][, ...]);
```

- You specify:
  - Table name
  - Column name, column data type, and column size



CREATE TABLE Statement

You create tables to store data by executing the SQL CREATE TABLE statement. This statement is one of the DDL statements, which are a subset of SQL statements used to create, modify, or remove Oracle database structures. These statements have an immediate effect on the database, and they also record information in the data dictionary.

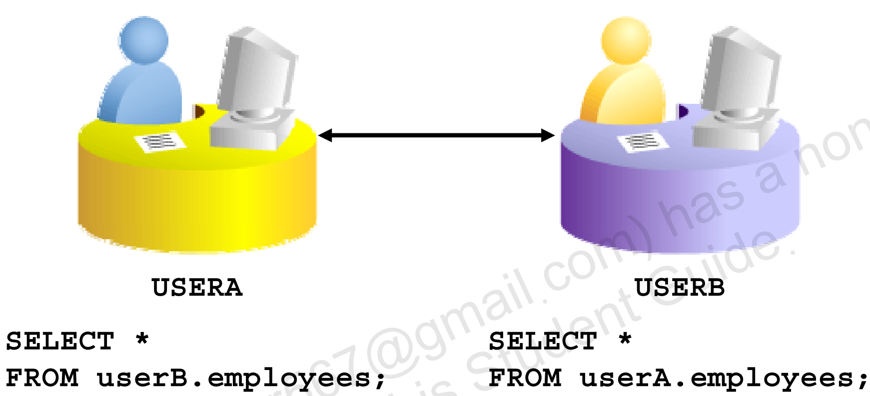
To create a table, a user must have the CREATE TABLE privilege and a storage area in which to create objects. The database administrator uses data control language statements to grant privileges to users (DCL statements are covered in a later lesson).

In the syntax:

schema	Is the same as the owner's name
table	is the name of the table
DEFAULT *expr*	Specifies a default value if a value is omitted in the INSERT statement
column	Is the name of the column
datatype	Is the column's data type and length

Referencing Another User's Tables

- Tables belonging to other users are not in the user's schema.
- You should use the owner's name as a prefix to those tables.



Referencing Another User's Tables

A *schema* is a collection of objects. Schema objects are the logical structures that directly refer to the data in a database. Schema objects include tables, views, synonyms, sequences, stored procedures, indexes, clusters, and database links.

If a table does not belong to the user, the owner's name must be prefixed to the table. For example, if there are schemas named USERA and USERB, and both have an EMPLOYEES table, then if USERA wants to access the EMPLOYEES table that belongs to USERB, he must prefix the table name with the schema name:

```
SELECT *
FROM   userb.employees;
```

If USERB wants to access the EMPLOYEES table that is owned by USERA, he must prefix the table name with the schema name:

```
SELECT *
FROM   usera.employees;
```

DEFAULT Option

- Specify a default value for a column during an insert.

```
... hire_date DATE DEFAULT SYSDATE, ...
```

values.

- Another column's name or a pseudocolumn are illegal values.
- The default data type must match the column data type.

```
CREATE TABLE hire_dates
        (id           NUMBER(8),
         hire_date DATE DEFAULT SYSDATE);
Table created.
```


DEFAULT Option

When you define a table, you can specify that a column be given a default value by using the DEFAULT option. This option prevents null values from entering the columns if a row is inserted without a value for the column. The default value can be a literal, an expression, or a SQL function (such as SYSDATE or USER), but the value cannot be the name of another column or a pseudocolumn (such as NEXTVAL or CURRVAL). The default expression must match the data type of the column.

Note: CURRVAL and NEXTVAL are explained later in this lesson.

Creating Tables

- Create the table.

```
CREATE TABLE dept
        (deptno      NUMBER(2),
         dname       VARCHAR2(14),
         loc         VARCHAR2(13),
         create_date DATE DEFAULT SYSDATE);
Table created.
```

- Confirm table creation.

```
DESCRIBE dept
```

Name	Null?	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)
CREATE_DATE		DATE

Creating Tables

The example in the slide creates the DEPT table, with four columns: DEPTNO, DNAME, LOC, and CREATE_DATE. The CREATE_DATE column has a default value. If a value is not provided for an INSERT statement, the system date is automatically inserted.

It further confirms the creation of the table by issuing the DESCRIBE command.

Because creating a table is a DDL statement, an automatic commit takes place when this statement is executed.

Data Types

Data Type	Description
`VARCHAR2(size)`	Variable-length character data
`CHAR(size)`	Fixed-length character data
`NUMBER(p,s)`	Variable-length numeric data
`DATE`	Date and time values
`LONG`	Variable-length character data (up to 2 GB)
`CLOB`	Character data (up to 4 GB)
`RAW and LONG RAW`	Raw binary data
`BLOB`	Binary data (up to 4 GB)
`BFILE`	Binary data stored in an external file (up to 4 GB)
`ROWID`	A base-64 number system representing the unique address of a row in its table

ORACLE

Data Types

When you identify a column for a table, you need to provide a data type for the column. There are several data types available:

Data Type	Description
`VARCHAR2(`*`size`*`)`	Variable-length character data (A maximum *size* must be specified: minimum *size* is 1; maximum *size* is 4,000.)
`CHAR [(`*`size`*`)]`	Fixed-length character data of length *size* bytes (Default and minimum *size* is 1; maximum *size* is 2,000.)
`NUMBER [(`*`p,s`*`)]`	Number having precision *p* and scale *s* (The precision is the total number of decimal digits, and the scale is the number of digits to the right of the decimal point; the precision can range from 1 to 38, and the scale can range from –84 to 127.)
`DATE`	Date and time values to the nearest second between January 1, 4712 B.C., and December 31, 9999 A.D.
`LONG`	Variable-length character data (up to 2 GB)
`CLOB`	Character data (up to 4 GB)

Data Types (continued)

Data Type	Description
`RAW(size)`	Raw binary data of length *size* (A maximum *size* must be specified: maximum *size* is 2,000.)
`LONG RAW`	Raw binary data of variable length (up to 2 GB)
`BLOB`	Binary data (up to 4 GB)
`BFILE`	Binary data stored in an external file (up to 4 GB)
`ROWID`	A base-64 number system representing the unique address of a row in its table

Guidelines

- A `LONG` column is not copied when a table is created using a subquery.
- A `LONG` column cannot be included in a `GROUP BY` or an `ORDER BY` clause.
- Only one `LONG` column can be used per table.
- No constraints can be defined on a `LONG` column.
- You might want to use a `CLOB` column rather than a `LONG` column.

Datetime Data Types

You can use several datetime data types:

Data Type	Description
`TIMESTAMP`	Date with fractional seconds
`INTERVAL YEAR TO MONTH`	Stored as an interval of years and months
`INTERVAL DAY TO SECOND`	Stored as an interval of days, hours, minutes, and seconds

Other Datetime Data Types

Data Type	**Description**
`TIMESTAMP`	Enables the time to be stored as a date with fractional seconds. There are several variations of this data type.
`INTERVAL YEAR TO MONTH`	Enables time to be stored as an interval of years and months. Used to represent the difference between two datetime values in which the only significant portions are the year and month.
`INTERVAL DAY TO SECOND`	Enables time to be stored as an interval of days, hours, minutes, and seconds. Used to represent the precise difference between two datetime values.

Note: These datetime data types are available with Oracle9*i* and later releases. For detailed information about the datetime data types, see the topics "`TIMESTAMP` Datatype," "`INTERVAL YEAR TO MONTH` Datatype," and "`INTERVAL DAY TO SECOND` Datatype" in the *Oracle SQL Reference*.

Datetime Data Types

- The TIMESTAMP data type is an extension of the DATE data type.
- It stores the year, month, and day of the DATE data type plus hour, minute, and second values as well as the fractional second value.
- You can optionally specify the time zone.

```
TIMESTAMP[(fractional_seconds_precision)]
```

```
TIMESTAMP[(fractional_seconds_precision)]
WITH TIME ZONE
```

```
TIMESTAMP[(fractional_seconds_precision)]
WITH LOCAL TIME ZONE
```

ORACLE

TIMESTAMP Data Type

The TIMESTAMP data type is an extension of the DATE data type. It stores the year, month, and day of the DATE data type plus hour, minute, and second values. This data type is used for storing precise time values.

The fractional_seconds_precision optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

Example

In this example, a table is created named NEW_EMPLOYEES, with a column START_DATE that has a data type of TIMESTAMP:

```
CREATE TABLE new_employees
  (employee_id NUMBER,
   first_name VARCHAR2(15),
   last_name VARCHAR2(15),
   ...
   start_date TIMESTAMP(7),
   ...);
```

Suppose that two rows are inserted in the NEW_EMPLOYEES table. The displayed output shows the differences. (A DATE data type defaults to display the DD-MON-RR format.):

`TIMESTAMP` Data Type (continued)

```
SELECT start_date
FROM   new_employees;

17-JUN-03 12.00.00.000000 AM
21-SEP-03 12.00.00.000000 AM
```

`TIMESTAMP WITH TIME ZONE` Data Type

TIMESTAMP WITH TIME ZONE is a variant of TIMESTAMP that includes a time-zone displacement in its value. The time-zone displacement is the difference (in hours and minutes) between local time and UTC (Universal Time Coordinate, formerly known as Greenwich Mean Time). This data type is used for collecting and evaluating date information across geographic regions.

For example,

```
TIMESTAMP '2003-04-15 8:00:00 -8:00'
```

is the same as

```
TIMESTAMP '2003-04-15 11:00:00 -5:00'
```

That is, 8:00 a.m. Pacific Standard Time is the same as 11:00 a.m. Eastern Standard Time.

This can also be specified as follows:

```
TIMESTAMP '2003-04-15 8:00:00 US/Pacific'
```

`TIMESTAMP WITH LOCAL TIME ZONE` Data Type

TIMESTAMP WITH LOCAL TIME ZONE is another variant of TIMESTAMP that includes a time-zone displacement in its value. It differs from TIMESTAMP WITH TIME ZONE in that data stored in the database is normalized to the database time zone, and the time-zone displacement is not stored as part of the column data. When users retrieve the data, it is returned in the users' local session time zone. The time-zone displacement is the difference (in hours and minutes) between local time and UTC.

Unlike TIMESTAMP WITH TIME ZONE, you can specify columns of type TIMESTAMP WITH LOCAL TIME ZONE as part of a primary or unique key, as in the following example:

```
CREATE TABLE time_example
    (order_date TIMESTAMP WITH LOCAL TIME ZONE);

   INSERT INTO time_example VALUES('15-JAN-04 09:34:28 AM');

   SELECT *
   FROM   time_example;

   ORDER_DATE
   ----------------------------
   15-JAN-04 09.34.28.000000 AM
```

The TIMESTAMP WITH LOCAL TIME ZONE type is appropriate for two-tier applications in which you want to display dates and times using the time zone of the client system.

Datetime Data Types

- The `INTERVAL YEAR TO MONTH` data type stores a period of time using the `YEAR` and `MONTH` datetime fields:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

- The `INTERVAL DAY TO SECOND` data type stores a period of time in terms of days, hours, minutes, and seconds:

```
INTERVAL DAY [(day_precision)]
   TO SECOND [(fractional_seconds_precision)]
```


`INTERVAL YEAR TO MONTH` Data Type

`INTERVAL YEAR TO MONTH` stores a period of time using the `YEAR` and `MONTH` datetime fields.

Use `INTERVAL YEAR TO MONTH` to represent the difference between two datetime values, where the only significant portions are the year and month. For example, you might use this value to set a reminder for a date that is 120 months in the future, or check whether 6 months have elapsed since a particular date.

In the syntax:

`year_precision`	is the number of digits in the `YEAR` datetime field. The default value of `year_precision` is 2.

Examples

- `INTERVAL '123-2' YEAR(3) TO MONTH`
  Indicates an interval of 123 years, 2 months
- `INTERVAL '123' YEAR(3)`
  Indicates an interval of 123 years 0 months
- `INTERVAL '300' MONTH(3)`
  Indicates an interval of 300 months
- `INTERVAL '123' YEAR`
  Returns an error because the default precision is 2, and `123` has 3 digits

INTERVAL YEAR TO MONTH Data Type (continued)

```
CREATE TABLE time_example2
(loan_duration INTERVAL YEAR (3) TO MONTH);

INSERT INTO time_example2 (loan_duration)
  VALUES (INTERVAL '120' MONTH(3));

SELECT TO_CHAR( sysdate+loan_duration, 'dd-mon-yyyy')
FROM   time_example2;        --today's date is 26-Sep-2001
```

TO_CHAR(SYS
26-sep-2011

INTERVAL DAY TO SECOND Data Type

INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds.

Use INTERVAL DAY TO SECOND to represent the precise difference between two datetime values. For example, you might use this value to set a reminder for a time that is 36 hours in the future, or to record the time between the start and end of a race. To represent long spans of time, including multiple years, with high precision, you can use a large value for the days portion.

In the syntax:

day_precision	Is the number of digits in the DAY datetime field. Accepted values are 0 to 9. The default is 2.
fractional_seconds_precision	Is the number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.

Examples

- INTERVAL '4 5:12:10.222' DAY TO SECOND(3)
  Indicates 4 days, 5 hours, 12 minutes, 10 seconds, and 222 thousandths of a second.
- INTERVAL '180' DAY(3)
  Indicates 180 days.
- INTERVAL '4 5:12:10.222' DAY TO SECOND(3)
  Indicates 4 days, 5 hours, 12 minutes, 10 seconds, and 222 thousandths of a second
- INTERVAL '4 5:12' DAY TO MINUTE
  Indicates 4 days, 5 hours, and 12 minutes
- INTERVAL '400 5' DAY(3) TO HOUR
  Indicates 400 days and 5 hours.
- INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)
  Indicates 11 hours, 12 minutes, and 10.2222222 seconds.

INTERVAL DAY TO SECOND Data Type (continued)

Example

```
CREATE TABLE time_example3
(day_duration INTERVAL DAY (3) TO SECOND);

INSERT INTO time_example3 (day_duration)
VALUES (INTERVAL '180' DAY(3));

SELECT sysdate + day_duration "Half Year"
FROM   time_example3;        --today's date is 26-Sep-2001
```

Half Year
25-MAR-02

Including Constraints

- Constraints enforce rules at the table level.
- Constraints prevent the deletion of a table if there are dependencies.
- The following constraint types are valid:
  - `NOT NULL`
  - `UNIQUE`
  - `PRIMARY KEY`
  - `FOREIGN KEY`
  - `CHECK`

Constraints

The Oracle server uses constraints to prevent invalid data entry into tables.

You can use constraints to do the following:

- Enforce rules on the data in a table whenever a row is inserted, updated, or deleted from that table. The constraint must be satisfied for the operation to succeed.
- Prevent the deletion of a table if there are dependencies from other tables
- Provide rules for Oracle tools, such as Oracle Developer

Data Integrity Constraints

Constraint	Description
`NOT NULL`	Specifies that the column cannot contain a null value
`UNIQUE`	Specifies a column or combination of columns whose values must be unique for all rows in the table
`PRIMARY KEY`	Uniquely identifies each row of the table
`FOREIGN KEY`	Establishes and enforces a foreign key relationship between the column and a column of the referenced table
`CHECK`	Specifies a condition that must be true

Constraint Guidelines

- You can name a constraint, or the Oracle server generates a name by using the `SYS_Cn` format.
- Create a constraint at either of the following times:
  - At the same time as the table is created
  - After the table has been created
- Define a constraint at the column or table level.
- View a constraint in the data dictionary.

Constraint Guidelines

All constraints are stored in the data dictionary. Constraints are easy to reference if you give them a meaningful name. Constraint names must follow the standard object-naming rules. If you do not name your constraint, the Oracle server generates a name with the format `SYS_Cn`, where *n* is an integer so that the constraint name is unique.

Constraints can be defined at the time of table creation or after the table has been created.

For more information, see "Constraints" in *Oracle Database SQL Reference*.

Defining Constraints

- Syntax:

```
CREATE TABLE [schema.]table
      (column datatype [DEFAULT expr]
      [column_constraint],
      ...
      [table_constraint][,...]);
```

- Column-level constraint:

```
column [CONSTRAINT constraint_name] constraint_type,
```

- Table-level constraint:

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


Defining Constraints

The slide gives the syntax for defining constraints when creating a table. You can create the constraints at either the column level or table level. Constraints defined at the column level are included when the column is defined. Table-level constraints are defined at the end of the table definition and must refer to the column or columns on which the constraint pertains in a set of parentheses.

`NOT NULL` constraints must be defined at the column level.

Constraints that apply to more than one column must be defined at the table level.

In the syntax:

`schema`	Is the same as the owner's name
`table`	Is the name of the table
`DEFAULT expr`	Specifies a default value to use if a value is omitted in the `INSERT` statement
`column`	Is the name of the column
`datatype`	Is the column's data type and length
`column_constraint`	Is an integrity constraint as part of the column definition
`table_constraint`	Is an integrity constraint as part of the table definition

Defining Constraints

- Column-level constraint:

```
CREATE TABLE employees(
  employee_id  NUMBER(6)
    CONSTRAINT emp_emp_id_pk PRIMARY KEY,
  first_name   VARCHAR2(20),
  ...);
```

1

- Table-level constraint:

```
CREATE TABLE employees(
  employee_id  NUMBER(6),
  first_name   VARCHAR2(20),
  ...
  job_id       VARCHAR2(10) NOT NULL,
  CONSTRAINT emp_emp_id_pk
    PRIMARY KEY (EMPLOYEE_ID));
```

2

Defining Constraints (continued)

Constraints are usually created at the same time as the table. Constraints can be added to a table after its creation and also temporarily disabled.

Both slide examples create a primary key constraint on the `EMPLOYEE_ID` column of the `EMPLOYEES` table.

1. The first example uses the column-level syntax to define the constraint.
2. The second example uses the table-level syntax to define the constraint.

More details about the primary key constraint are provided later in this lesson.

NOT NULL Constraint

Ensures that null values are not permitted for the column:

EMPLOYEE_ID	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
100	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	90
101	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000	90
102	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000	90
103	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000	60
104	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000	60
178	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	
200	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400	10

...

20 rows selected.

NOT NULL constraint (No row can contain a null value for this column.)

NOT NULL constraint

Absence of NOT NULL constraint (Any row can contain a null value for this column.)

NOT NULL Constraint

The NOT NULL constraint ensures that the column contains no null values. Columns without the NOT NULL constraint can contain null values by default. NOT NULL constraints must be defined at the column level.

UNIQUE Constraint

UNIQUE constraint

EMPLOYEES

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

...

INSERT INTO

208	Smith	JSMITH	Allowed
209	Smith	JSMITH	Not allowed: already exists

ORACLE

UNIQUE Constraint

A UNIQUE key integrity constraint requires that every value in a column or set of columns (key) be unique—that is, no two rows of a table can have duplicate values in a specified column or set of columns. The column (or set of columns) included in the definition of the UNIQUE key constraint is called the *unique key*. If the UNIQUE constraint comprises more than one column, that group of columns is called a *composite unique key*.

UNIQUE constraints enable the input of nulls unless you also define NOT NULL constraints for the same columns. In fact, any number of rows can include nulls for columns without NOT NULL constraints because nulls are not considered equal to anything. A null in a column (or in all columns of a composite UNIQUE key) always satisfies a UNIQUE constraint.

Note: Because of the search mechanism for UNIQUE constraints on more than one column, you cannot have identical values in the non-null columns of a partially null composite UNIQUE key constraint.

UNIQUE Constraint

Defined at either the table level or the column level:

```
CREATE TABLE employees(
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
...
    CONSTRAINT emp_email_uk UNIQUE(email));
```


UNIQUE Constraint (continued)

UNIQUE constraints can be defined at the column level or table level. A composite unique key is created by using the table-level definition.

The example in the slide applies the UNIQUE constraint to the EMAIL column of the EMPLOYEES table. The name of the constraint is EMP_EMAIL_UK.

Note: The Oracle server enforces the UNIQUE constraint by implicitly creating a unique index on the unique key column or columns.

PRIMARY KEY Constraint

DEPARTMENTS

PRIMARY KEY

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

...

Not allowed (null value)

INSERT INTO

	Public Accounting		1400
50	Finance	124	1500

Not allowed (50 already exists)

ORACLE

PRIMARY KEY Constraint

A PRIMARY KEY constraint creates a primary key for the table. Only one primary key can be created for each table. The PRIMARY KEY constraint is a column or set of columns that uniquely identifies each row in a table. This constraint enforces uniqueness of the column or column combination and ensures that no column that is part of the primary key can contain a null value.

Note: Because uniqueness is part of the primary key constraint definition, the Oracle server enforces the uniqueness by implicitly creating a unique index on the primary key column or columns.

FOREIGN KEY Constraint

DEPARTMENTS

PRIMARY KEY

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

...

EMPLOYEES

FOREIGN KEY

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

...

INSERT INTO

200	Ford	9	Not allowed (9 does not exist)
201	Ford	60	Allowed

ORACLE

FOREIGN KEY Constraint

The FOREIGN KEY (or referential integrity) constraint designates a column or combination of columns as a foreign key and establishes a relationship between a primary key or a unique key in the same table or a different table.

In the example in the slide, DEPARTMENT_ID has been defined as the foreign key in the EMPLOYEES table (dependent or child table); it references the DEPARTMENT_ID column of the DEPARTMENTS table (the referenced or parent table).

Guidelines

- A foreign key value must match an existing value in the parent table or be NULL.
- Foreign keys are based on data values and are purely logical, rather than physical, pointers.

FOREIGN KEY Constraint

Defined at either the table level or the column level:

```
CREATE TABLE employees(
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
...
    department_id    NUMBER(4),
    CONSTRAINT emp_dept_fk FOREIGN KEY (department_id)
      REFERENCES departments(department_id),
    CONSTRAINT emp_email_uk UNIQUE(email));
```


FOREIGN KEY Constraint (continued)

FOREIGN KEY constraints can be defined at the column or table constraint level. A composite foreign key must be created by using the table-level definition.

The example in the slide defines a FOREIGN KEY constraint on the DEPARTMENT_ID column of the EMPLOYEES table, using table-level syntax. The name of the constraint is EMP_DEPTID_FK.

The foreign key can also be defined at the column level, provided the constraint is based on a single column. The syntax differs in that the keywords FOREIGN KEY do not appear. For example:

```
CREATE TABLE employees
(...
department_id NUMBER(4) CONSTRAINT emp_deptid_fk
REFERENCES departments(department_id),
...
)
```

FOREIGN KEY Constraint: Keywords

- FOREIGN KEY: Defines the column in the child table at the table-constraint level
- REFERENCES: Identifies the table and column in the parent table
- ON DELETE CASCADE: Deletes the dependent rows in the child table when a row in the parent table is deleted
- ON DELETE SET NULL: Converts dependent foreign key values to null

FOREIGN KEY Constraint: Keywords

The foreign key is defined in the child table, and the table containing the referenced column is the parent table. The foreign key is defined using a combination of the following keywords:

- FOREIGN KEY is used to define the column in the child table at the table-constraint level.
- REFERENCES identifies the table and column in the parent table.
- ON DELETE CASCADE indicates that when the row in the parent table is deleted, the dependent rows in the child table are also deleted.
- ON DELETE SET NULL converts foreign key values to null when the parent value is removed.

The default behavior is called the *restrict rule*, which disallows the update or deletion of referenced data.

Without the ON DELETE CASCADE or the ON DELETE SET NULL options, the row in the parent table cannot be deleted if it is referenced in the child table.

`CHECK` Constraint

- Defines a condition that each row must satisfy
- The following expressions are not allowed:
  - References to `CURRVAL`, `NEXTVAL`, `LEVEL`, and `ROWNUM` pseudocolumns
  - Calls to `SYSDATE`, `UID`, `USER`, and `USERENV` functions
  - Queries that refer to other values in other rows

```
..., salary   NUMBER(2)
     CONSTRAINT emp_salary_min
            CHECK (salary > 0),...
```


`CHECK` Constraint

The `CHECK` constraint defines a condition that each row must satisfy. The condition can use the same constructs as query conditions, with the following exceptions:

- References to the `CURRVAL`, `NEXTVAL`, `LEVEL`, and `ROWNUM` pseudocolumns
- Calls to `SYSDATE`, `UID`, `USER`, and `USERENV` functions
- Queries that refer to other values in other rows

A single column can have multiple `CHECK` constraints that refer to the column in its definition. There is no limit to the number of `CHECK` constraints that you can define on a column.

`CHECK` constraints can be defined at the column level or table level.

```
CREATE TABLE employees
   (...
    salary NUMBER(8,2) CONSTRAINT emp_salary_min
                       CHECK (salary > 0),
   ...
```

CREATE TABLE: Example

```
CREATE TABLE employees
    ( employee_id    NUMBER(6)
        CONSTRAINT     emp_employee_id   PRIMARY KEY
    , first_name     VARCHAR2(20)
    , last_name      VARCHAR2(25)
        CONSTRAINT     emp_last_name_nn  NOT NULL
    , email          VARCHAR2(25)
        CONSTRAINT     emp_email_nn      NOT NULL
        CONSTRAINT     emp_email_uk      UNIQUE
    , phone_number   VARCHAR2(20)
    , hire_date      DATE
        CONSTRAINT     emp_hire_date_nn  NOT NULL
    , job_id         VARCHAR2(10)
        CONSTRAINT     emp_job_nn        NOT NULL
    , salary         NUMBER(8,2)
        CONSTRAINT     emp_salary_ck     CHECK (salary>0)
    , commission_pct NUMBER(2,2)
    , manager_id     NUMBER(6)
    , department_id  NUMBER(4)
        CONSTRAINT     emp_dept_fk       REFERENCES
           departments (department_id));
```

ORACLE

The CREATE TABLE Example

The example shows the statement used to create the EMPLOYEES table in the HR schema.

Violating Constraints

```
UPDATE employees
SET    department_id = 55
WHERE  department_id = 110;
```

```
UPDATE employees
       *
ERROR at line 1:
ORA-02291: integrity constraint (HR.EMP_DEPT_FK)
violated - parent key not found
```

Department 55 does not exist.

Integrity Constraint Error

When you have constraints in place on columns, an error is returned to you if you try to violate the constraint rule.

For example, if you attempt to update a record with a value that is tied to an integrity constraint, an error is returned.

In the example in the slide, department 55 does not exist in the parent table, `DEPARTMENTS`, and so you receive the *parent key* violation `ORA-02291`.

Violating Constraints

You cannot delete a row that contains a primary key that is used as a foreign key in another table.

```
DELETE FROM departments
WHERE       department_id = 60;
```

```
DELETE FROM departments
            *
ERROR at line 1:
ORA-02292: integrity constraint (HR.EMP_DEPT_FK)
violated - child record found
```


Integrity Constraint Error (continued)

If you attempt to delete a record with a value that is tied to an integrity constraint, an error is returned.

The example in the slide tries to delete department 60 from the `DEPARTMENTS` table, but it results in an error because that 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.
```

Creating a Table by Using a Subquery

- Create a table and insert rows by combining the `CREATE TABLE` statement and the `AS` *`subquery`* option.

```
CREATE TABLE table
          [(column, column...)]
AS subquery;
```

- Match the number of specified columns to the number of subquery columns.
- Define columns with column names and default values.

Creating a Table from Rows in Another Table

A second method for creating a table is to apply the `AS` *`subquery`* clause, which both creates the table and inserts rows returned from the subquery.

In the syntax:

`table`	is the name of the table
`column`	is the name of the column, default value, and integrity constraint
`subquery`	is the `SELECT` statement that defines the set of rows to be inserted into the new table

Guidelines

- The table is created with the specified column names, and the rows retrieved by the `SELECT` statement are inserted into the table.
- The column definition can contain only the column name and default value.
- If column specifications are given, the number of columns must equal the number of columns in the subquery `SELECT` list.
- If no column specifications are given, the column names of the table are the same as the column names in the subquery.
- The column data type definitions and the NOT NULL constraint are passed to the new table. The other constraint rules are not passed to the new table. However, you can add constraints in the column definition.

Creating a Table by Using a Subquery

```
CREATE TABLE  dept80
  AS
    SELECT   employee_id, last_name,
             salary*12 ANNSAL,
             hire_date
    FROM     employees
    WHERE    department_id = 80;
Table created.
```

```
DESCRIBE dept80
```

Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
ANNSAL		NUMBER
HIRE_DATE	NOT NULL	DATE

Creating a Table from Rows in Another Table (continued)

The slide example creates a table named DEPT80, which contains details of all the employees working in department 80. Notice that the data for the DEPT80 table comes from the EMPLOYEES table.

You can verify the existence of a database table and check column definitions by using the *i*SQL*Plus DESCRIBE command.

Be sure to provide a column alias when selecting an expression. The expression SALARY*12 is given the alias ANNSAL. Without the alias, the following error is generated:

```
ERROR at line 3:
ORA-00998: must name this expression with a column alias
```

ALTER TABLE Statement

Use the ALTER TABLE statement to:

- Add a new column
- Modify an existing column
- Define a default value for the new column
- Drop a column

ALTER TABLE Statement

After you create a table, you may need to change the table structure for any of the following reasons:

- You omitted a column.
- Your column definition needs to be changed.
- You need to remove columns.

You can do this by using the ALTER TABLE statement. For information about the ALTER TABLE statement, see the *Oracle Database 10g SQL Fundamentals II* course.

Dropping a Table

- All data and structure in the table are deleted.
- Any pending transactions are committed.
- All indexes are dropped.
- All constraints are dropped.
- You *cannot* roll back the `DROP TABLE` statement.

```
DROP TABLE dept80;
Table dropped.
```


Dropping a Table

The `DROP TABLE` statement removes the definition of an Oracle table. When you drop a table, the database loses all the data in the table and all the indexes associated with it.

Syntax

```
DROP TABLE table
```

In the syntax, *`table`* is the name of the table.

Guidelines

- All data is deleted from the table.
- Any views and synonyms remain but are invalid.
- Any pending transactions are committed.
- Only the creator of the table or a user with the `DROP ANY TABLE` privilege can remove a table.

Note: The `DROP TABLE` statement, once executed, is irreversible. The Oracle server does not question the action when you issue the `DROP TABLE` statement. If you own that table or have a high-level privilege, then the table is immediately removed. As with all DDL statements, `DROP TABLE` is committed automatically.

Summary

In this lesson, you should have learned how to use the `CREATE TABLE` statement to create a table and include constraints.

- Categorize the main database objects
- Review the table structure
- List the data types that are available for columns
- Create a simple table
- Explain how constraints are created at the time of table creation
- Describe how schema objects work

Summary

In this lesson, you should have learned how to do the following:

CREATE TABLE

- Use the `CREATE TABLE` statement to create a table and include constraints.
- Create a table based on another table by using a subquery.

DROP TABLE

- Remove rows and a table structure.
- Once executed, this statement cannot be rolled back.

Practice 9: Overview

This practice covers the following topics:

- Creating new tables
- Creating a new table by using the `CREATE TABLE AS` syntax
- Verifying that tables exist
- Dropping tables

Practice 9: Overview

Create new tables by using the `CREATE TABLE` statement. Confirm that the new table was added to the database. Create the syntax in the command file, and then execute the command file to create the table.

Practice 9

1. Create the DEPT table based on the following table instance chart. Place the syntax in a script called lab_09_01.sql, and then execute the statement in the script to create the table. Confirm that the table is created.

Column Name	ID	NAME
Key Type	Primary key	
Nulls/Unique		
FK Table		
FK Column		
Data type	NUMBER	VARCHAR2
Length	7	25

Name	Null?	Type
ID		NUMBER(7)
NAME		VARCHAR2(25)

2. Populate the DEPT table with data from the DEPARTMENTS table. Include only columns that you need.

3. Create the EMP table based on the following table instance chart. Place the syntax in a script called lab_09_03.sql, and then execute the statement in the script to create the table. Confirm that the table is created.

Column Name	ID	LAST_NAME	FIRST_NAME	DEPT_ID
Key Type				
Nulls/Unique				
FK Table				DEPT
FK Column				ID
Data type	NUMBER	VARCHAR2	VARCHAR2	NUMBER
Length	7	25	25	7

Name	Null?	Type
ID		NUMBER(7)
LAST_NAME		VARCHAR2(25)
FIRST_NAME		VARCHAR2(25)
DEPT_ID		NUMBER(7)

Practice 9 (continued)

4. Create the EMPLOYEES2 table based on the structure of the EMPLOYEES table. Include only the EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY, and DEPARTMENT_ID columns. Name the columns in your new table ID, FIRST_NAME, LAST_NAME, SALARY, and DEPT_ID, respectively.
5. Drop the EMP table.

10

Creating Other Schema Objects

Objectives

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

- Create simple and complex views
- Retrieve data from views
- Create, maintain, and use sequences
- Create and maintain indexes
- Create private and public synonyms

Objectives

In this lesson, you are introduced to the view, sequence, synonym, and index objects. You are taught the basics of creating and using views, sequences, and indexes.

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

Database Objects

There are several other objects in a database in addition to tables. In this lesson, you learn about views, sequences, indexes, and synonyms.

With views, you can present and hide data from tables.

Many applications require the use of unique numbers as primary key values. You can either build code into the application to handle this requirement or use a sequence to generate unique numbers.

If you want to improve the performance of some queries, you should consider creating an index. You can also use indexes to enforce uniqueness on a column or a collection of columns.

You can provide alternative names for objects by using synonyms.

What Is a View?

EMPLOYEES table

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
					JUL-98	ST_CLERK	25
					JAN-00	SA_MAN	105
					MAY-96	SA_REP	110
					MAR-98	SA_REP	86
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	70
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	44
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	130
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	60
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	120
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	83

20 rows selected.

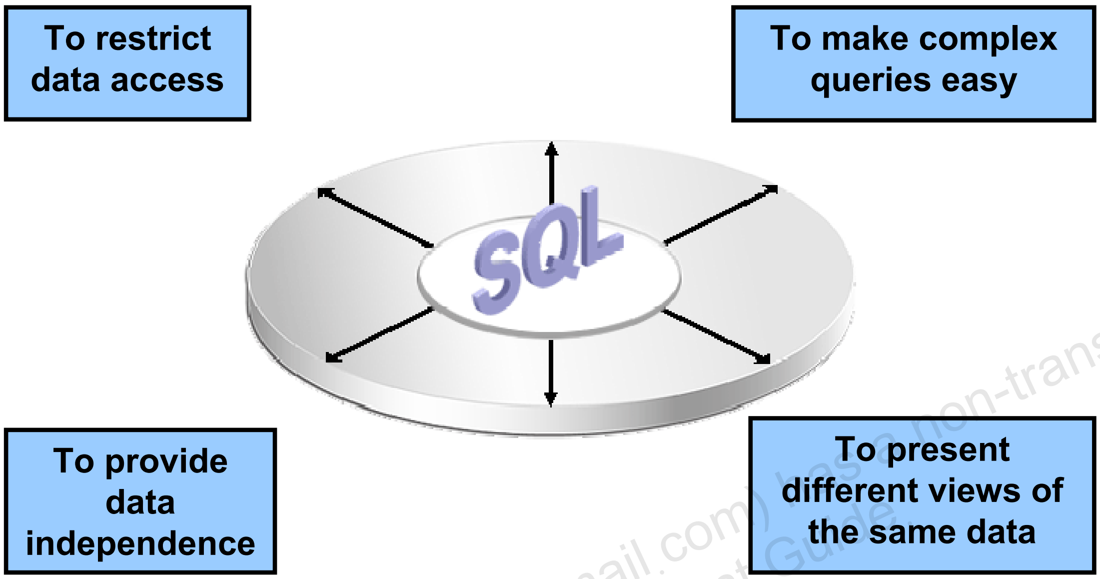
EMPLOYEE_ID	LAST_NAME	SALARY
149	Zlotkey	10500
174	Abel	11000
176	Taylor	8600

ORACLE

What Is a View?

You can present logical subsets or combinations of data by creating views of tables. A view is a logical table based on a table or another view. A view contains no data of its own but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called *base tables*. The view is stored as a SELECT statement in the data dictionary.

Advantages of Views



ORACLE

Advantages of Views

- Views restrict access to the data because the view can display selected columns from the table.
- Views can be used to make simple queries to retrieve the results of complicated queries. For example, views can be used to query information from multiple tables without the user knowing how to write a join statement.
- Views provide data independence for ad hoc users and application programs. One view can be used to retrieve data from several tables.
- Views provide groups of users access to data according to their particular criteria.

For more information, see “`CREATE VIEW`” in the *Oracle SQL Reference*.

Simple Views and Complex Views

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operations through a view	Yes	Not always

Simple Views and Complex Views

There are two classifications for views: simple and complex. The basic difference is related to the DML (`INSERT`, `UPDATE`, and `DELETE`) operations.

- A simple view is one that:
  - Derives data from only one table
  - Contains no functions or groups of data
  - Can perform DML operations through the view
- A complex view is one that:
  - Derives data from many tables
  - Contains functions or groups of data
  - Does not always allow DML operations through the view

Creating a View

- You embed a subquery in the CREATE VIEW statement:

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
  [(alias[, alias]...)]
 AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY [CONSTRAINT constraint]];
```

- The subquery can contain complex SELECT syntax.

Creating a View

You can create a view by embedding a subquery in the `CREATE VIEW` statement.

In the syntax:

`OR REPLACE`	Re-creates the view if it already exists
`FORCE`	Creates the view regardless of whether or not the base tables exist
`NOFORCE`	Creates the view only if the base tables exist (This is the default.)
`view`	Is the name of the view
`alias`	Specifies names for the expressions selected by the view's query (The number of aliases must match the number of expressions selected by the view.)
`subquery`	Is a complete `SELECT` statement (You can use aliases for the columns in the `SELECT` list.)
`WITH CHECK OPTION`	Specifies that only those rows that are accessible to the view can be inserted or updated
`constraint`	Is the name assigned to the `CHECK OPTION` constraint
`WITH READ ONLY`	ensures that no DML operations can be performed on this view

Creating a View

- Create the `EMPVU80` view, which contains details of employees in department 80:

```
CREATE VIEW   empvu80
 AS SELECT  employee_id, last_name, salary
    FROM    employees
    WHERE   department_id = 80;
View created.
```

- Describe the structure of the view by using the *i*SQL*Plus `DESCRIBE` command:

```
DESCRIBE empvu80
```


Creating a View (continued)

The example in the slide creates a view that contains the employee number, last name, and salary for each employee in department 80.

You can display the structure of the view by using the *i*SQL*Plus `DESCRIBE` command.

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
SALARY		NUMBER(8,2)

Guidelines for Creating a View

- The subquery that defines a view can contain complex `SELECT` syntax, including joins, groups, and subqueries.
- If you do not specify a constraint name for a view created with the `WITH CHECK OPTION`, the system assigns a default name in the format `SYS_C`*n*.
- You can use the `OR REPLACE` option to change the definition of the view without dropping and re-creating it or regranting object privileges previously granted on it.

Creating a View

- Create a view by using column aliases in the subquery:

```
CREATE VIEW   salvu50
 AS SELECT   employee_id ID_NUMBER, last_name NAME,
             salary*12 ANN_SALARY
    FROM     employees
    WHERE    department_id = 50;
View created.
```

- Select the columns from this view by the given alias names:

Creating a View (continued)

You can control the column names by including column aliases in the subquery.

The example in the slide creates a view containing the employee number (EMPLOYEE_ID) with the alias ID_NUMBER, name (LAST_NAME) with the alias NAME, and annual salary (SALARY) with the alias ANN_SALARY for every employee in department 50.

As an alternative, you can use an alias after the CREATE statement and before the SELECT subquery. The number of aliases listed must match the number of expressions selected in the subquery.

```
CREATE OR REPLACE VIEW   salvu50 (ID_NUMBER, NAME, ANN_SALARY)
  AS SELECT  employee_id, last_name, salary*12
     FROM    employees
     WHERE   department_id = 50;
View created.
```

Retrieving Data from a View

```
SELECT *
FROM   salvu50;
```

ID_NUMBER	NAME	ANN_SALARY
124	Mourgos	69600
141	Rajs	42000
142	Davies	37200
143	Matos	31200
144	Vargas	30000

Retrieving Data from a View

You can retrieve data from a view as you would from any table. You can display either the contents of the entire view or just specific rows and columns.

Modifying a View

- Modify the `EMPVU80` view by using a `CREATE OR REPLACE VIEW` clause. Add an alias for each column name:

```
CREATE OR REPLACE VIEW empvu80
  (id_number, name, sal, department_id)
AS SELECT  employee_id, first_name || ' '
           || last_name, salary, department_id
   FROM    employees
   WHERE   department_id = 80;
View created.
```

- Column aliases in the `CREATE OR REPLACE VIEW` clause are listed in the same order as the columns in the subquery.

Modifying a View

With the `OR REPLACE` option, a view can be created even if one exists with this name already, thus replacing the old version of the view for its owner. This means that the view can be altered without dropping, re-creating, and regranting object privileges.

Note: When assigning column aliases in the `CREATE OR REPLACE VIEW` clause, remember that the aliases are listed in the same order as the columns in the subquery.

Creating a Complex View

Create a complex view that contains group functions to display values from two tables:

```
CREATE OR REPLACE VIEW dept_sum_vu
  (name, minsal, maxsal, avgsal)
AS SELECT   d.department_name, MIN(e.salary),
            MAX(e.salary),AVG(e.salary)
   FROM     employees e JOIN departments d
   ON       (e.department_id = d.department_id)
   GROUP BY d.department_name;
View created.
```

ORACLE

Creating a Complex View

The example in the slide creates a complex view of department names, minimum salaries, maximum salaries, and average salaries by department. Note that alternative names have been specified for the view. This is a requirement if any column of the view is derived from a function or an expression.

You can view the structure of the view by using the *i*SQL*Plus `DESCRIBE` command. Display the contents of the view by issuing a `SELECT` statement.

```
SELECT  *
FROM    dept_sum_vu;
```

NAME	MINSAL	MAXSAL	AVGSAL
Accounting	8300	12000	10150
Administration	4400	4400	4400
Executive	17000	24000	19333.3333
IT	4200	9000	6400
Marketing	6000	13000	9500
Sales	8600	11000	10033.3333
Shipping	2500	5800	3500

7 rows selected.

Rules for Performing DML Operations on a View

- You can usually perform DML operations on simple views.
- You cannot remove a row if the view contains the following:
  - Group functions
  - A `GROUP BY` clause
  - The `DISTINCT` keyword
  - The pseudocolumn `ROWNUM` keyword

ORACLE

Performing DML Operations on a View

You can perform DML operations on data through a view if those operations follow certain rules.

You can remove a row from a view unless it contains any of the following:

- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- The pseudocolumn `ROWNUM` keyword

Rules for Performing DML Operations on a View

You cannot modify data in a view if it contains:

- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- The pseudocolumn `ROWNUM` keyword
- Columns defined by expressions

Performing DML Operations on a View (continued)

You can modify data through a view unless it contains any of the conditions mentioned in the previous slide or columns defined by expressions (for example, `SALARY * 12`).

Rules for Performing DML Operations on a View

You cannot add data through a view if the view includes:

- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- The pseudocolumn `ROWNUM` keyword
- Columns defined by expressions
- `NOT NULL` columns in the base tables that are not selected by the view

Performing DML Operations on a View (continued)

You can add data through a view unless it contains any of the items listed in the slide. You cannot add data to a view if the view contains `NOT NULL` columns without default values in the base table. All required values must be present in the view. Remember that you are adding values directly to the underlying table *through* the view.

For more information, see "`CREATE VIEW`" in the *Oracle SQL Reference*.

Using the `WITH CHECK OPTION` Clause

- You can ensure that DML operations performed on the view stay in the domain of the view by using the `WITH CHECK OPTION` clause:

```
CREATE OR REPLACE VIEW empvu20
AS SELECT     *
   FROM      employees
   WHERE     department_id = 20
   WITH CHECK OPTION CONSTRAINT empvu20_ck ;
View created.
```

- Any attempt to change the department number for any row in the view fails because it violates the `WITH CHECK OPTION` constraint.

Using the `WITH CHECK OPTION` Clause

It is possible to perform referential integrity checks through views. You can also enforce constraints at the database level. The view can be used to protect data integrity, but the use is very limited.

The `WITH CHECK OPTION` clause specifies that `INSERT`s and `UPDATE`s performed through the view cannot create rows that the view cannot select, and therefore it enables integrity constraints and data validation checks to be enforced on data being inserted or updated. If there is an attempt to perform DML operations on rows that the view has not selected, an error is displayed, along with the constraint name if that has been specified.

```
UPDATE empvu20
SET    department_id = 10
WHERE  employee_id = 201;
```

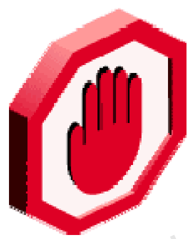
causes:

```
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Note: No rows are updated because if the department number were to change to 10, the view would no longer be able to see that employee. With the `WITH CHECK OPTION` clause, therefore, the view can see only employees in department 20 and does not allow the department number for those employees to be changed through the view.

Denying DML Operations

- You can ensure that no DML operations occur by adding the `WITH READ ONLY` option to your view definition.
- Any attempt to perform a DML operation on any row in the view results in an Oracle server error.



Denying DML Operations

You can ensure that no DML operations occur on your view by creating it with the `WITH READ ONLY` option. The example in the next slide modifies the `EMPVU10` view to prevent any DML operations on the view.

Denying DML Operations

```
CREATE OR REPLACE VIEW empvu10
    (employee_number, employee_name, job_title)
AS SELECT     employee_id, last_name, job_id
   FROM       employees
   WHERE      department_id = 10
   WITH READ ONLY ;
View created.
```


Denying DML Operations (continued)

Any attempt to remove a row from a view with a read-only constraint results in an error:

```
DELETE FROM empvu10
WHERE  employee_number = 200;
DELETE FROM empvu10
         *
ERROR at line 1:
ORA-01752: cannot delete from view without exactly one key-
preserved table
```

Any attempt to insert a row or modify a row using the view with a read-only constraint results in an Oracle server error:

```
01733: virtual column not allowed here.
```

Removing a View

You can remove a view without losing data because a view is based on underlying tables in the database.

```
DROP VIEW view;
```

```
DROP VIEW empvu80;
View dropped.
```


Removing a View

You use the DROP VIEW statement to remove a view. The statement removes the view definition from the database. Dropping views has no effect on the tables on which the view was based. Views or other applications based on deleted views become invalid. Only the creator or a user with the DROP ANY VIEW privilege can remove a view.

In the syntax:

view is the name of the view

Practice 10: Overview of Part 1

This practice covers the following topics:

- Creating a simple view
- Creating a complex view
- Creating a view with a check constraint
- Attempting to modify data in the view
- Removing views

Practice 10: Overview of Part 1

Part 1 of this lesson's practice provides you with a variety of exercises in creating, using, and removing views.

Complete questions 1–6 at the end of this lesson.

Sequences

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

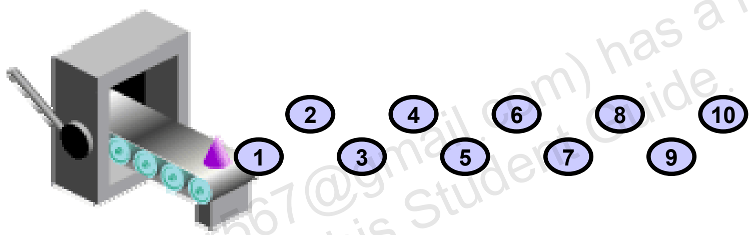
Sequences

A sequence is a database object that creates integer values. You can create sequences and then use them to generate numbers.

Sequences

A sequence:

- Can automatically generate unique numbers
- Is a sharable object
- Can be used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory



Sequences (continued)

A sequence is a user-created database object that can be shared by multiple users to generate integers.

You can define a sequence to generate unique values or to recycle and use the same numbers again.

A typical usage for sequences is to create a primary key value, which must be unique for each row. The sequence is generated and incremented (or decremented) by an internal Oracle routine. This can be a time-saving object because it can reduce the amount of application code needed to write a sequence-generating routine.

Sequence numbers are stored and generated independently of tables. Therefore, the same sequence can be used for multiple tables.

CREATE SEQUENCE Statement: Syntax

Define a sequence to generate sequential numbers automatically:

```
CREATE SEQUENCE sequence
       [INCREMENT BY n]
       [START WITH n]
       [{MAXVALUE n | NOMAXVALUE}]
       [{MINVALUE n | NOMINVALUE}]
       [{CYCLE | NOCYCLE}]
       [{CACHE n | NOCACHE}];
```

ORACLE

Creating a Sequence

Automatically generate sequential numbers by using the CREATE SEQUENCE statement.

In the syntax:

sequence	Is the name of the sequence generator
INCREMENT BY *n*	Specifies the interval between sequence numbers, where *n* is an integer (If this clause is omitted, the sequence increments by 1.)
START WITH *n*	Specifies the first sequence number to be generated (If this clause is omitted, the sequence starts with 1.)
MAXVALUE *n*	Specifies the maximum value the sequence can generate
NOMAXVALUE	Specifies a maximum value of 10^27 for an ascending sequence and –1 for a descending sequence (This is the default option.)
MINVALUE *n*	Specifies the minimum sequence value
NOMINVALUE	Specifies a minimum value of 1 for an ascending sequence and –(10^26) for a descending sequence (This is the default option.)

Creating a Sequence

- Create a sequence named DEPT_DEPTID_SEQ to be used for the primary key of the DEPARTMENTS table.
- Do not use the CYCLE option.

```
CREATE SEQUENCE dept_deptid_seq
                INCREMENT BY 10
                START WITH 120
                MAXVALUE 9999
                NOCACHE
                NOCYCLE;
Sequence created.
```


Creating a Sequence (continued)

CYCLE | NOCYCLE	Specifies whether the sequence continues to generate values after reaching its maximum or minimum value (NOCYCLE is the default option.)
CACHE *n* | NOCACHE	Specifies how many values the Oracle server preallocates and keeps in memory (By default, the Oracle server caches 20 values.)

The example in the slide creates a sequence named DEPT_DEPTID_SEQ to be used for the DEPARTMENT_ID column of the DEPARTMENTS table. The sequence starts at 120, does not allow caching, and does not cycle.

Do not use the CYCLE option if the sequence is used to generate primary key values, unless you have a reliable mechanism that purges old rows faster than the sequence cycles.

For more information, see “CREATE SEQUENCE” in the *Oracle SQL Reference*.

Note: The sequence is not tied to a table. Generally, you should name the sequence after its intended use. However, the sequence can be used anywhere, regardless of its name.

NEXTVAL and CURRVAL Pseudocolumns

- NEXTVAL returns the next available sequence value. It returns a unique value every time it is referenced, even for different users.
- CURRVAL obtains the current sequence value.
- NEXTVAL must be issued for that sequence before CURRVAL contains a value.

NEXTVAL and CURRVAL Pseudocolumns

After you create your sequence, it generates sequential numbers for use in your tables. Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.

The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify NEXTVAL with the sequence name. When you reference *sequence*.NEXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL.

The CURRVAL pseudocolumn is used to refer to a sequence number that the current user has just generated. NEXTVAL must be used to generate a sequence number in the current user's session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. When you reference *sequence*.CURRVAL, the last value returned to that user's process is displayed.

`NEXTVAL` and `CURRVAL` Pseudocolumns (continued)

Rules for Using `NEXTVAL` and `CURRVAL`

You can use NEXTVAL and CURRVAL in the following contexts:

- The SELECT list of a SELECT statement that is not part of a subquery
- The SELECT list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

You cannot use NEXTVAL and CURRVAL in the following contexts:

- The SELECT list of a view
- A SELECT statement with the DISTINCT keyword
- A SELECT statement with GROUP BY, HAVING, or ORDER BY clauses
- A subquery in a SELECT, DELETE, or UPDATE statement
- The DEFAULT expression in a CREATE TABLE or ALTER TABLE statement

For more information, see "Pseudocolumns" and "CREATE SEQUENCE" in *Oracle SQL Reference*.

Using a Sequence

- Insert a new department named “Support” in location ID 2500:

```
INSERT INTO departments(department_id,
            department_name, location_id)
VALUES      (dept_deptid_seq.NEXTVAL,
            'Support', 2500);
1 row created.
```

- View the current value for the `DEPT_DEPTID_SEQ` sequence:

```
SELECT   dept_deptid_seq.CURRVAL
FROM     dual;
```

ORACLE

Using a Sequence

The example in the slide inserts a new department in the `DEPARTMENTS` table. It uses the `DEPT_DEPTID_SEQ` sequence to generate a new department number as follows.

You can view the current value of the sequence:

```
SELECT dept_deptid_seq.CURRVAL
FROM   dual;
```

CURRVAL
120

Suppose that you now want to hire employees to staff the new department. The `INSERT` statement to be executed for all new employees can include the following code:

```
INSERT INTO employees (employee_id, department_id, ...)
VALUES (employees_seq.NEXTVAL, dept_deptid_seq .CURRVAL, ...);
```

Note: The preceding example assumes that a sequence called `EMPLOYEE_SEQ` has already been created to generate new employee numbers.

Caching Sequence Values

- Caching sequence values in memory gives faster access to those values.
- Gaps in sequence values can occur when:
  - A rollback occurs
  - The system crashes
  - A sequence is used in another table

Caching Sequence Values

You can cache sequences in memory to provide faster access to those sequence values. The cache is populated the first time you refer to the sequence. Each request for the next sequence value is retrieved from the cached sequence. After the last sequence value is used, the next request for the sequence pulls another cache of sequences into memory.

Gaps in the Sequence

Although sequence generators issue sequential numbers without gaps, this action occurs independent of a commit or rollback. Therefore, if you roll back a statement containing a sequence, the number is lost.

Another event that can cause gaps in the sequence is a system crash. If the sequence caches values in memory, then those values are lost if the system crashes.

Because sequences are not tied directly to tables, the same sequence can be used for multiple tables. If you do so, each table can contain gaps in the sequential numbers.

Modifying a Sequence

Change the increment value, maximum value, minimum value, cycle option, or cache option:

```
ALTER SEQUENCE dept_deptid_seq
               INCREMENT BY 20
               MAXVALUE 999999
               NOCACHE
               NOCYCLE;
Sequence altered.
```


Modifying a Sequence

If you reach the MAXVALUE limit for your sequence, no additional values from the sequence are allocated and you will receive an error indicating that the sequence exceeds the MAXVALUE. To continue to use the sequence, you can modify it by using the ALTER SEQUENCE statement.

Syntax

```
ALTER  SEQUENCE   sequence
      [INCREMENT BY n]
      [{MAXVALUE n | NOMAXVALUE}]
      [{MINVALUE n | NOMINVALUE}]
      [{CYCLE | NOCYCLE}]
      [{CACHE n | NOCACHE}];
```

In the syntax, *sequence* is the name of the sequence generator.

For more information, see "ALTER SEQUENCE" in the *Oracle SQL Reference*.

Guidelines for Modifying a Sequence

- You must be the owner or have the `ALTER` privilege for the sequence.
- Only future sequence numbers are affected.
- The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed.
- To remove a sequence, use the `DROP` statement:

```
DROP SEQUENCE dept_deptid_seq;
Sequence dropped.
```


Guidelines for Modifying a Sequence

- You must be the owner or have the `ALTER` privilege for the sequence to modify it. You must be the owner or have the `DROP ANY SEQUENCE` privilege to remove it.
- Only future sequence numbers are affected by the `ALTER SEQUENCE` statement.
- The `START WITH` option cannot be changed using `ALTER SEQUENCE`. The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed. For example, a new `MAXVALUE` that is less than the current sequence number cannot be imposed.

```
ALTER SEQUENCE dept_deptid_seq
       INCREMENT BY 20
       MAXVALUE 90
       NOCACHE
       NOCYCLE;
ALTER SEQUENCE dept_deptid_seq
*
ERROR at line 1:
ORA-04009: MAXVALUE cannot be made to be less than the
 current value
```

Indexes

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

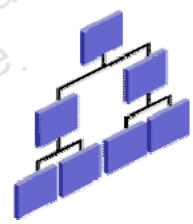
Indexes

Indexes are database objects that you can create to improve the performance of some queries. Indexes can also be created automatically by the server when you create a primary key or unique constraint.

Indexes

An index:

- Is a schema object
- Can be used by the Oracle server to speed up the retrieval of rows by using a pointer
- Can reduce disk I/O by using a rapid path access method to locate data quickly
- Is independent of the table that it indexes
- Is used and maintained automatically by the Oracle server



ORACLE

Indexes (continued)

An Oracle server index is a schema object that can speed up the retrieval of rows by using a pointer. Indexes can be created explicitly or automatically. If you do not have an index on the column, then a full table scan occurs.

An index provides direct and fast access to rows in a table. Its purpose is to reduce the necessity of disk I/O by using an indexed path to locate data quickly. The index is used and maintained automatically by the Oracle server. After an index is created, no direct activity is required by the user.

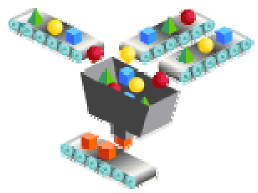
Indexes are logically and physically independent of the table that they index. This means that they can be created or dropped at any time and have no effect on the base tables or other indexes.

Note: When you drop a table, corresponding indexes are also dropped.

For more information, see "Schema Objects: Indexes" in *Database Concepts*.

How Are Indexes Created?

- Automatically: A unique index is created automatically when you define a `PRIMARY KEY` or `UNIQUE` constraint in a table definition.



- Manually: Users can create nonunique indexes on columns to speed up access to the rows.

Types of Indexes

Two types of indexes can be created.

Unique index: The Oracle server automatically creates this index when you define a column in a table to have a `PRIMARY KEY` or a `UNIQUE` key constraint. The name of the index is the name that is given to the constraint.

Nonunique index: This is an index that a user can create. For example, you can create a `FOREIGN KEY` column index for a join in a query to improve retrieval speed.

Note: You can manually create a unique index, but it is recommended that you create a unique constraint, which implicitly creates a unique index.

Creating an Index

- Create an index on one or more columns:

```
CREATE INDEX index
ON table (column[, column]...);
```

- Improve the speed of query access to the LAST_NAME column in the EMPLOYEES table:

```
CREATE INDEX  emp_last_name_idx
ON            employees(last_name);
Index created.
```


Creating an Index

Create an index on one or more columns by issuing the CREATE INDEX statement.

In the syntax:

index	Is the name of the index
table	Is the name of the table
column	Is the name of the column in the table to be indexed

For more information, see "CREATE INDEX" in *Oracle SQL Reference*.

Index Creation Guidelines

Create an index when:	
✓	A column contains a wide range of values
✓	A column contains a large number of null values
✓	One or more columns are frequently used together in a `WHERE` clause or a join condition
✓	The table is large and most queries are expected to retrieve less than 2% to 4% of the rows in the table
Do not create an index when:	
✗	The columns are not often used as a condition in the query
✗	The table is small or most queries are expected to retrieve more than 2% to 4% of the rows in the table
✗	The table is updated frequently
✗	The indexed columns are referenced as part of an expression

ORACLE

More Is Not Always Better

Having more indexes on a table does not produce faster queries. Each DML operation that is committed on a table with indexes means that the indexes must be updated. The more indexes that you have associated with a table, the more effort the Oracle server must make to update all the indexes after a DML operation.

When to Create an Index

Therefore, you should create indexes only if:

- The column contains a wide range of values
- The column contains a large number of null values
- One or more columns are frequently used together in a `WHERE` clause or join condition
- The table is large and most queries are expected to retrieve less than 2% to 4% of the rows

Remember that if you want to enforce uniqueness, you should define a unique constraint in the table definition. A unique index is then created automatically.

Removing an Index

- Remove an index from the data dictionary by using the `DROP INDEX` command:

```
DROP INDEX index;
```

- Remove the `UPPER_LAST_NAME_IDX` index from the data dictionary:

```
DROP INDEX emp_last_name_idx;
Index dropped.
```

- To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

Removing an Index

You cannot modify indexes. To change an index, you must drop it and then re-create it.

Remove an index definition from the data dictionary by issuing the `DROP INDEX` statement. To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

In the syntax, *`index`* is the name of the index.

Note: If you drop a table, indexes and constraints are automatically dropped but views and sequences remain.

Synonyms

Object	Description
Table	Basic unit of storage; composed of rows
View	Logically represents subsets of data from one or more tables
Sequence	Generates numeric values
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

Synonyms

Synonyms are database objects that enable you to call a table by another name. You can create synonyms to give an alternative name to a table.

Synonyms

Simplify access to objects by creating a synonym (another name for an object). With synonyms, you can:

- Create an easier reference to a table that is owned by another user
- Shorten lengthy object names

```
CREATE [PUBLIC] SYNONYM synonym
FOR    object;
```


Creating a Synonym for an Object

To refer to a table that is owned by another user, you need to prefix the table name with the name of the user who created it, followed by a period. Creating a synonym eliminates the need to qualify the object name with the schema and provides you with an alternative name for a table, view, sequence, procedure, or other objects. This method can be especially useful with lengthy object names, such as views.

In the syntax:

`PUBLIC`	Creates a synonym that is accessible to all users
`synonym`	Is the name of the synonym to be created
`object`	Identifies the object for which the synonym is created

Guidelines

- The object cannot be contained in a package.
- A private synonym name must be distinct from all other objects that are owned by the same user.

For more information, see “`CREATE SYNONYM`” in the *Oracle SQL Reference*.

Creating and Removing Synonyms

- Create a shortened name for the `DEPT_SUM_VU` view:

```
CREATE SYNONYM  d_sum
FOR  dept_sum_vu;
Synonym Created.
```

- Drop a synonym:

```
DROP  SYNONYM d_sum;
Synonym dropped.
```


Creating a Synonym

The slide example creates a synonym for the `DEPT_SUM_VU` view for quicker reference.

The database administrator can create a public synonym that is accessible to all users. The following example creates a public synonym named `DEPT` for Alice's `DEPARTMENTS` table:

```
CREATE PUBLIC SYNONYM  dept
FOR    alice.departments;
Synonym created.
```

Removing a Synonym

To remove a synonym, use the `DROP SYNONYM` statement. Only the database administrator can drop a public synonym.

```
DROP PUBLIC SYNONYM  dept;
Synonym dropped.
```

For more information, see "`DROP SYNONYM`" in the *Oracle SQL Reference*.

Summary

In this lesson, you should have learned how to:

- Create, use, and remove views
- Automatically generate sequence numbers by using a sequence generator
- Create indexes to improve query retrieval speed
- Use synonyms to provide alternative names for objects

Summary

In this lesson, you should have learned about database objects such as views, sequences, indexes, and synonyms.

Practice 10: Overview of Part 2

This practice covers the following topics:

- Creating sequences
- Using sequences
- Creating nonunique indexes
- Creating synonyms

Practice 10: Overview of Part 2

Part 2 of this lesson's practice provides you with a variety of exercises in creating and using a sequence, an index, and a synonym.

Complete questions 7–10 at the end of this lesson.

Practice 10

Part 1

1. The staff in the HR department want to hide some of the data in the EMPLOYEES table. They want a view called EMPLOYEES_VU based on the employee numbers, employee names, and department numbers from the EMPLOYEES table. They want the heading for the employee name to be EMPLOYEE.
2. Confirm that the view works. Display the contents of the EMPLOYEES_VU view.

EMPLOYEE_ID	EMPLOYEE	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90
103	Hunold	60
104	Ernst	60
107	Lorentz	60
...		
206	Gietz	110

20 rows selected.

3. Using your EMPLOYEES_VU view, write a query for the HR department to display all employee names and department numbers.

EMPLOYEE	DEPARTMENT_ID
King	90
Kochhar	90
...	
Gietz	110

20 rows selected.

Practice 10 (continued)

4. Department 50 needs access to its employee data. Create a view named DEPT50 that contains the employee numbers, employee last names, and department numbers for all employees in department 50. You have been asked to label the view columns EMPNO, EMPLOYEE, and DEPTNO. For security purposes, do not allow an employee to be reassigned to another department through the view.
5. Display the structure and contents of the DEPT50 view.

Name	Null?	Type
EMPNO	NOT NULL	NUMBER(6)
EMPLOYEE	NOT NULL	VARCHAR2(25)
DEPTNO		NUMBER(4)

EMPNO	EMPLOYEE	DEPTNO
124	Mourgos	50
141	Rajs	50
142	Davies	50
143	Matos	50
144	Vargas	50

6. Test your view. Attempt to reassign Matos to department 80.

Practice 10 (continued)

Part 2

7. You need a sequence that can be used with the primary key column of the DEPT table. The sequence should start at 200 and have a maximum value of 1,000. Have your sequence increment by 10. Name the sequence DEPT_ID_SEQ.
8. To test your sequence, write a script to insert two rows in the DEPT table. Name your script lab_10_08.sql. Be sure to use the sequence that you created for the ID column. Add two departments: Education and Administration. Confirm your additions. Run the commands in your script.
9. Create a nonunique index on the NAME column in the DEPT table.
10. Create a synonym for your EMPLOYEES table. Call it EMP.

11

Managing Objects with Data Dictionary Views

Objectives

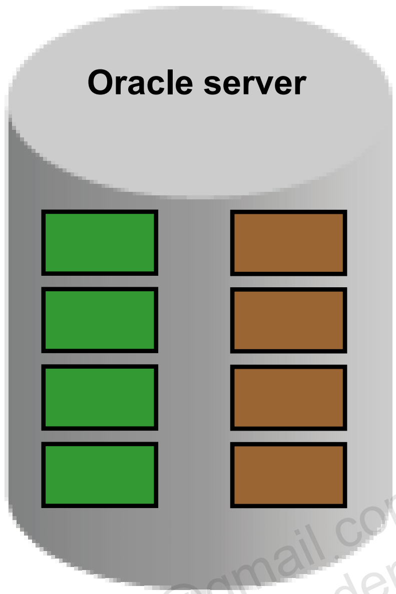
After completing this lesson, you should be able to do the following:

- Use the data dictionary views to research data on your objects
- Query various data dictionary views

Objectives

In this lesson, you are introduced to the data dictionary views. You learn that the dictionary views can be used to retrieve metadata and create reports about your schema objects.

The Data Dictionary



Tables containing business data:
EMPLOYEES
DEPARTMENTS
LOCATIONS
JOB_HISTORY
...

Data dictionary views:
DICTIONARY
USER_OBJECTS
USER_TABLES
USER_TAB_COLUMNS
...

ORACLE

The Data Dictionary

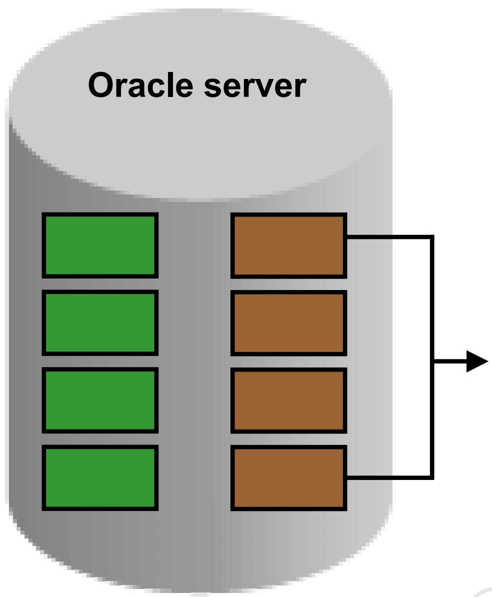
User tables are tables created by the user and contain business data, such as EMPLOYEES. There is another collection of tables and views in the Oracle database known as the data dictionary. This collection is created and maintained by the Oracle server and contains information about the database. The *data dictionary* is structured in tables and views, just like other database data. Not only is the data dictionary central to every Oracle database, but it is an important tool for all users, from end users to application designers and database administrators.

You use SQL statements to access the data dictionary. Because the data dictionary is read-only, you can issue only queries against its tables and views.

You can query the dictionary views that are based on the dictionary tables to find information such as:

- Definitions of all schema objects in the database (tables, views, indexes, synonyms, sequences, procedures, functions, packages, triggers, and so on)
- Default values for columns
- Integrity constraint information
- Names of Oracle users
- Privileges and roles that each user has been granted
- Other general database information

Data Dictionary Structure



Consists of:

– **Base tables**
– **User-accessible views**

Data Dictionary Structure

Underlying base tables store information about the associated database. Only the Oracle server should write to and read these tables. You rarely access them directly.

There are several views that summarize and display the information stored in the base tables of the data dictionary. These views decode the base table data into useful information (such as user or table names) using joins and `WHERE` clauses to simplify the information. Most users are given access to the views rather than the base tables.

The Oracle user `SYS` owns all base tables and user-accessible views of the data dictionary. No Oracle user should *ever* alter (`UPDATE`, `DELETE`, or `INSERT`) any rows or schema objects contained in the `SYS` schema, because such activity can compromise data integrity.

Data Dictionary Structure

View naming convention:

View Prefix	Purpose
USER	User's view (what is in your schema; what you own)
ALL	Expanded user's view (what you can access)
DBA	Database administrator's view (what is in everyone's schemas)
V$	Performance-related data

Data Dictionary Structure (continued)

The data dictionary consists of sets of views. In many cases, a set consists of three views containing similar information and distinguished from each other by their prefixes. For example, there is a view named `USER_OBJECTS`, another named `ALL_OBJECTS`, and a third named `DBA_OBJECTS`.

These three views contain similar information about objects in the database, except that the scope is different. `USER_OBJECTS` contains information about objects that you own or created. `ALL_OBJECTS` contains information about all objects to which you have access. `DBA_OBJECTS` contains information on all objects that are owned by all users. For views that are prefixed with `ALL` or `DBA`, there is usually an additional column in the view named `OWNER` to identify who owns the object.

There is also a set of views that is prefixed with `v$`. These views are dynamic in nature and hold information about performance. Dynamic performance tables are not true tables, and they should not be accessed by most users. However, database administrators can query and create views on the tables and grant access to those views to other users. This course does not go into details about these views.

How to Use the Dictionary Views

Start with `DICTIONARY`. It contains the names and descriptions of the dictionary tables and views.

```
DESCRIBE DICTIONARY
```

Name	Null?	Type
TABLE_NAME		VARCHAR2(30)
COMMENTS		VARCHAR2(4000)

```
SELECT *
FROM   dictionary
WHERE  table_name = 'USER_OBJECTS';
```

TABLE_NAME	COMMENTS
USER_OBJECTS	Objects owned by the user

How to Use the Dictionary Views

To familiarize yourself with the dictionary views, you can use the dictionary view named `DICTIONARY`. It contains the name and short description of each dictionary view to which you have access.

You can write queries to search for information on a particular view name, or you can search the `COMMENTS` column for a word or phrase. In the example shown, the `DICTIONARY` view is described. It has two columns. The `SELECT` statement retrieves information about the dictionary view named `USER_OBJECTS`. The `USER_OBJECTS` view contains information about all the objects that you own.

You can write queries to search the `COMMENTS` column for a word or phrase. For example, the following query returns the names of all views that you are permitted to access in which the `COMMENTS` column contains the word *columns*:

```
SELECT table_name
FROM dictionary
WHERE LOWER(comments) LIKE '%columns';
```

Note: The names in the data dictionary are uppercase.

USER_OBJECTS and ALL_OBJECTS Views

USER_OBJECTS:

- Query USER_OBJECTS to see all of the objects that are owned by you
- Is a useful way to obtain a listing of all object names and types in your schema, plus the following information:
  - Date created
  - Date of last modification
  - Status (valid or invalid)

ALL_OBJECTS:

- Query ALL_OBJECTS to see all objects to which you have access

ORACLE

USER_OBJECTS View

You can query the USER_OBJECTS view to see the names and types of all the objects in your schema. There are several columns in this view:

- **OBJECT_NAME:** Name of the object
- **OBJECT_ID:** Dictionary object number of the object
- **OBJECT_TYPE:** Type of object (such as TABLE, VIEW, INDEX, SEQUENCE)
- **CREATED:** Timestamp for the creation of the object
- **LAST_DDL_TIME:** Timestamp for the last modification of the object resulting from a DDL command
- **STATUS:** Status of the object (VALID, INVALID, or N/A)
- **GENERATED:** Was the name of this object system-generated? (Y|N)

Note: This is not a complete listing of the columns. For a complete listing, see "USER_OBJECTS" in the *Oracle Database Reference*.

You can also query the ALL_OBJECTS view to see a listing of all objects to which you have access.

USER_OBJECTS View

```
SELECT object_name, object_type, created, status
FROM   user_objects
ORDER BY object_type;
```

OBJECT_NAME	OBJECT_TYPE	CREATED	STATUS
REG_ID_PK	INDEX	10-DEC-03	VALID
...			
DEPARTMENTS_SEQ	SEQUENCE	10-DEC-03	VALID
REGIONS	TABLE	10-DEC-03	VALID
LOCATIONS	TABLE	10-DEC-03	VALID
DEPARTMENTS	TABLE	10-DEC-03	VALID
JOB_HISTORY	TABLE	10-DEC-03	VALID
JOB_GRADES	TABLE	10-DEC-03	VALID
EMPLOYEES	TABLE	10-DEC-03	VALID
JOBS	TABLE	10-DEC-03	VALID
COUNTRIES	TABLE	10-DEC-03	VALID
EMP_DETAILS_VIEW	VIEW	10-DEC-03	VALID

ORACLE

USER_OBJECTS View (continued)

The example shows the names, types, dates of creation, and status of all objects that are owned by this user.

The OBJECT_TYPE column holds the values of either TABLE, VIEW, SEQUENCE, INDEX, PROCEDURE, FUNCTION, PACKAGE, or TRIGGER.

The STATUS column holds a value of VALID, INVALID, or N/A. While tables are always valid, the views, procedures, functions, packages, and triggers may be invalid.

The CAT View

For a simplified query and output, you can query the CAT view. This view contains only two columns: TABLE_NAME and TABLE_TYPE. It provides the names of all your INDEX, TABLE, CLUSTER, VIEW, SYNONYM, SEQUENCE, or UNDEFINED objects.

Table Information

USER_TABLES:

```
DESCRIBE user_tables
```

Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
TABLESPACE_NAME		VARCHAR2(30)
CLUSTER_NAME		VARCHAR2(30)
IOT_NAME		VARCHAR2(30)

```
SELECT table_name
FROM   user_tables;
```

TABLE_NAME
JOB_GRADES
REGIONS
COUNTRIES
LOCATIONS
DEPARTMENTS

...

ORACLE

USER_TABLES View

You can use the USER_TABLES view to obtain the names of all of your tables. The USER_TABLES view contains information about your tables. In addition to providing the table name, it contains detailed information on the storage.

The TABS view is a synonym of the USER_TABLES view. You can query it to see a listing of tables that you own:

```
SELECT table_name
FROM  tabs;
```

Note: For a complete listing of the columns in the USER_TABLES view, see "USER_TABLES" in the *Oracle Database Reference*.

You can also query the ALL_TABLES view to see a listing of all tables to which you have access.

Column Information

USER_TAB_COLUMNS:

```
DESCRIBE user_tab_columns
```

Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME	NOT NULL	VARCHAR2(30)
DATA_TYPE		VARCHAR2(106)
DATA_TYPE_MOD		VARCHAR2(3)
DATA_TYPE_OWNER		VARCHAR2(30)
DATA_LENGTH	NOT NULL	NUMBER
DATA_PRECISION		NUMBER
DATA_SCALE		NUMBER
NULLABLE		VARCHAR2(1)
COLUMN_ID		NUMBER
DEFAULT_LENGTH		NUMBER
DATA_DEFAULT		LONG

...

Column Information

You can query the USER_TAB_COLUMNS view to find detailed information about the columns in your tables. While the USER_TABLES view provides information on your table names and storage, detailed column information is found in the USER_TAB_COLUMNS view.

This view contains information such as:

- Column names
- Column data types
- Length of data types
- Precision and scale for NUMBER columns
- Whether nulls are allowed (Is there a NOT NULL constraint on the column?)
- Default value

Note: For a complete listing and description of the columns in the USER_TAB_COLUMNS view, see "USER_TAB_COLUMNS" in the *Oracle Database Reference*.

Column Information

```
SELECT column_name, data_type, data_length,
       data_precision, data_scale, nullable
FROM   user_tab_columns
WHERE  table_name = 'EMPLOYEES';
```

COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION	DATA_SCALE	NUL
EMPLOYEE_ID	NUMBER	22	6	0	N
FIRST_NAME	VARCHAR2	20			Y
LAST_NAME	VARCHAR2	25			N
EMAIL	VARCHAR2	25			N
PHONE_NUMBER	VARCHAR2	20			Y
HIRE_DATE	DATE	7			N
JOB_ID	VARCHAR2	10			N
SALARY	NUMBER	22	8	2	Y
COMMISSION_PCT	NUMBER	22	2	2	Y
MANAGER_ID	NUMBER	22	6	0	Y
DEPARTMENT_ID	NUMBER	22	4	0	Y

Column Information (continued)

By querying the `USER_TAB_COLUMNS` table, you can find details about your columns such as the names, data types, data type lengths, null constraints, and default value for a column.

The example shown displays the columns, data types, data lengths, and null constraints for the `EMPLOYEES` table. Note that this information is similar to the output from the *i*SQL*Plus `DESCRIBE` command.

Constraint Information

- `USER_CONSTRAINTS` describes the constraint definitions on your tables.
- `USER_CONS_COLUMNS` describes columns that are owned by you and that are specified in constraints.

```
DESCRIBE user_constraints
```

Name	Null?	Type
OWNER	NOT NULL	VARCHAR2(30)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)
CONSTRAINT_TYPE		VARCHAR2(1)
TABLE_NAME	NOT NULL	VARCHAR2(30)
SEARCH_CONDITION		LONG
R_OWNER		VARCHAR2(30)
R_CONSTRAINT_NAME		VARCHAR2(30)
DELETE_RULE		VARCHAR2(9)
STATUS		VARCHAR2(8)

...

Constraint Information

You can find out the names of your constraints, the type of constraint, the table name to which the constraint applies, the condition for check constraints, foreign key constraint information, deletion rule for foreign key constraints, the status, and many other types of information about your constraints.

Note: For a complete listing and description of the columns in the `USER_CONSTRAINTS` view, see "`USER_CONSTRAINTS`" in the *Oracle Database Reference*.

Constraint Information

```
SELECT constraint_name, constraint_type,
       search_condition, r_constraint_name,
       delete_rule, status
FROM   user_constraints
WHERE  table_name = 'EMPLOYEES';
```

CONSTRAINT_NAME	CON	SEARCH_CONDITION	R_CONSTRAINT_NAME	DELETE_RULE	STATUS
EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL			ENABLED
EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL			ENABLED
EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL			ENABLED
EMP_JOB_NN	C	"JOB_ID" IS NOT NULL			ENABLED
EMP_SALARY_MIN	C	salary > 0			ENABLED
EMP_EMAIL_UK	U				ENABLED
EMP_EMP_ID_PK	P				ENABLED
EMP_DEPT_FK	R		DEPT_ID_PK	NO ACTION	ENABLED
EMP_JOB_FK	R		JOB_ID_PK	NO ACTION	ENABLED
EMP_MANAGER_FK	R		EMP_EMP_ID_PK	NO ACTION	ENABLED

`USER_CONSTRAINTS`: Example

In the example shown, the `USER_CONSTRAINTS` view is queried to find the names, types, check conditions, name of the unique constraint that the foreign key references, deletion rule for a foreign key, and status for constraints on the `EMPLOYEES` table.

The `CONSTRAINT_TYPE` can be:

- `C` (check constraint on a table)
- `P` (primary key)
- `U` (unique key)
- `R` (referential integrity)
- `V` (with check option, on a view)
- `O` (with read-only, on a view)

The `DELETE_RULE` can be:

- **`CASCADE`:** If the parent record is deleted, the child records are deleted too.
- **`NO ACTION`:** A parent record can be deleted only if no child records exist.

The `STATUS` can be:

- **`ENABLED`:** Constraint is active.
- **`DISABLED`:** Constraint is made not active.

Constraint Information

```
DESCRIBE user_cons_columns
```

Name	Null?	Type
OWNER	NOT NULL	VARCHAR2(30)
CONSTRAINT_NAME	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
POSITION		NUMBER

```
SELECT constraint_name, column_name
FROM   user_cons_columns
WHERE  table_name = 'EMPLOYEES';
```

CONSTRAINT_NAME	COLUMN_NAME
EMP_EMAIL_UK	EMAIL
EMP_SALARY_MIN	SALARY
EMP_JOB_NN	JOB_ID
EMP_HIRE_DATE_NN	HIRE_DATE

...

ORACLE

Querying `USER_CONS_COLUMNS`

To find the names of the columns to which a constraint applies, query the USER_CONS_COLUMNS dictionary view. This view tells you the name of the owner of a constraint, the name of the constraint, the table that the constraint is on, the names of the columns with the constraint, and the original position of column or attribute in the definition of the object.

Note: A constraint may apply to more than one column.

You can also write a join between the USER_CONSTRAINTS and USER_CONS_COLUMNS to create customized output from both tables.

View Information

1

```
DESCRIBE user_views
```

Name	Null?	Type
VIEW_NAME	NOT NULL	VARCHAR2(30)
TEXT_LENGTH		NUMBER
TEXT		LONG

2

```
SELECT DISTINCT view_name FROM user_views;
```

VIEW_NAME
EMP_DETAILS_VIEW

3

```
SELECT text FROM user_views
WHERE view_name = 'EMP_DETAILS_VIEW';
```

TEXT
SELECT e.employee_id, e.job_id, e.manager_id, e.department_id, d.locat ion_id, l.country_id, e.first_name, e.last_name, e.salary, e.commissio n_pct, d.department_name, j.job_title, l.city, l.state_province, c.cou ntry_name, r.region_name FROM employees e, departments d, jobs j, loca tions l, countries c, regions r WHERE e.department_id = d.department_id AN D d.location_id = l.location_id AND l.country_id = c.country_id AND c.region _id = r.region_id AND j.job_id = e.job_id WITH READ ONLY

Views in the Data Dictionary

After your view is created, you can query the data dictionary view called USER_VIEWS to see the name of the view and the view definition. The text of the SELECT statement that constitutes your view is stored in a LONG column. The LENGTH column is the number of characters in the SELECT statement. By default, when you select from a LONG column, only the first 80 characters of the column's value are displayed. To see more than 80 characters, use the *i*SQL*Plus command SET LONG:

```
SET LONG 1000
```

In the examples in the slide:

1. The USER_VIEWS columns are displayed. Note that this is a partial listing.
2. The names of your views are retrieved.
3. The SELECT statement for the EMP_DETAILS_VIEW is displayed from the dictionary.

Data Access Using Views

When you access data using a view, the Oracle server performs the following operations:

- It retrieves the view definition from the data dictionary table USER_VIEWS.
- It checks access privileges for the view base table.
- It converts the view query into an equivalent operation on the underlying base table or tables. In other words, data is retrieved from, or an update is made to, the base tables.

Sequence Information

```
DESCRIBE user_sequences
```

Name	Null?	Type
SEQUENCE_NAME	NOT NULL	VARCHAR2(30)
MIN_VALUE		NUMBER
MAX_VALUE		NUMBER
INCREMENT_BY	NOT NULL	NUMBER
CYCLE_FLAG		VARCHAR2(1)
ORDER_FLAG		VARCHAR2(1)
CACHE_SIZE	NOT NULL	NUMBER
LAST_NUMBER	NOT NULL	NUMBER

`USER_SEQUENCES` View

The `USER_SEQUENCES` view describes all sequences that are owned by you. When you create the sequence, you specify criteria that are stored in the `USER_SEQUENCES` view. The columns in this view are:

- **`SEQUENCE_NAME`:** Name of the sequence
- **`MIN_VALUE`:** Minimum value of the sequence
- **`MAX_VALUE`:** Maximum value of the sequence
- **`INCREMENT_BY`:** Value by which sequence is incremented
- **`CYCLE_FLAG`:** Does sequence wrap around on reaching limit?
- **`ORDER_FLAG`:** Are sequence numbers generated in order?
- **`CACHE_SIZE`:** Number of sequence numbers to cache
- **`LAST_NUMBER`:** Last sequence number written to disk. If a sequence uses caching, the number written to disk is the last number placed in the sequence cache. This number is likely to be greater than the last sequence number that was used.

Sequence Information

- Verify your sequence values in the USER_SEQUENCES data dictionary table.

```
SELECT   sequence_name, min_value, max_value,
         increment_by, last_number
FROM     user_sequences;
```

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
LOCATIONS_SEQ	1	9900	100	3300
DEPARTMENTS_SEQ	1	9990	10	280
EMPLOYEES_SEQ	1	1.0000E+27	1	207

- The LAST_NUMBER column displays the next available sequence number if NOCACHE is specified.

Confirming Sequences

After creating your sequence, it is documented in the data dictionary. Because a sequence is a database object, you can identify it in the USER_OBJECTS data dictionary table.

You can also confirm the settings of the sequence by selecting from the USER_SEQUENCES data dictionary view.

Viewing the Next Available Sequence Value Without Incrementing It

If the sequence was created with NOCACHE, it is possible to view the next available sequence value without incrementing it by querying the USER_SEQUENCES table.

Synonym Information

```
DESCRIBE user_synonyms
```

Name	Null?	Type
SYNONYM_NAME	NOT NULL	VARCHAR2(30)
TABLE_OWNER		VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
DB_LINK		VARCHAR2(128)

```
SELECT *
FROM   user_synonyms;
```

SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK
EMP	ORA1	EMPLOYEES	

`USER_SYNONYMS` View

The `USER_SYNONYMS` dictionary view describes private synonyms (synonyms that are owned by you).

You can query this view to find your synonyms. You can query `ALL_SYNONYMS` to find out the name of all of the synonyms that are available to you and the objects on which these synonyms apply.

The columns in this view are:

- **`SYNONYM_NAME`:** Name of the synonym
- **`TABLE_OWNER`:** Owner of the object that is referenced by the synonym
- **`TABLE_NAME`:** Name of the table or view that is referenced by the synonym
- **`DB_LINK`:** Name of the database link reference (if any)

Adding Comments to a Table

- You can add comments to a table or column by using the COMMENT statement:

```
COMMENT ON TABLE employees
IS 'Employee Information';
Comment created.
```

- Comments can be viewed through the data dictionary views:
  – ALL_COL_COMMENTS
  – USER_COL_COMMENTS
  – ALL_TAB_COMMENTS
  – USER_TAB_COMMENTS

Adding Comments to a Table

The Comment command is used to document objects. You can add a comment of up to 4,000 bytes about a column, table, view, or snapshot by using the COMMENT statement. The comment is stored in the data dictionary and can be viewed in one of the following data dictionary views in the COMMENTS column:

- ALL_COL_COMMENTS
- USER_COL_COMMENTS
- ALL_TAB_COMMENTS
- USER_TAB_COMMENTS

Syntax

```
COMMENT ON TABLE table | COLUMN table.column
    IS 'text';
```

In the syntax:

table	Is the name of the table
column	Is the name of the column in a table
text	Is the text of the comment

You can drop a comment from the database by setting it to empty string (''):

```
COMMENT ON TABLE  employees IS ' ';
```

Summary

In this lesson, you should have learned how to find information about your objects through the following dictionary views:

- DICTIONARY
- USER_OBJECTS
- USER_TABLES
- USER_TAB_COLUMNS
- USER_CONSTRAINTS
- USER_CONS_COLUMNS
- USER_VIEWS
- USER_SEQUENCES
- USER_TAB_SYNONYMS

Summary

In this lesson, you learned about some of the dictionary views that are available to you. You can use these dictionary views to find information about your tables, constraints, views, sequences, and synonyms.

Practice 11: Overview

This practice covers the following topics:

- Querying the dictionary views for table and column information
- Querying the dictionary views for constraint information
- Querying the dictionary views for view information
- Querying the dictionary views for sequence information
- Querying the dictionary views for synonym information
- Adding a comment to a table and querying the dictionary views for comment information

Practice 11: Overview

In this practice, you query the dictionary views to find information about objects in your schema.

Practice 11

1. For a specified table, create a script that reports the column names, data types, and data types' lengths, as well as whether nulls are allowed. Prompt the user to enter the table name. Give appropriate aliases to the columns `DATA_PRECISION` and `DATA_SCALE`. Save this script in a file named `lab_11_01.sql`.

   For example, if the user enters `DEPARTMENTS`, the following output results:

COLUMN_NAME	DATA_TYPE	DATA_LENGTH	PRECISION	SCALE	NUL
DEPARTMENT_ID	NUMBER	22	4	0	N
DEPARTMENT_NAME	VARCHAR2	30			N
MANAGER_ID	NUMBER	22	6	0	Y
LOCATION_ID	NUMBER	22	4	0	Y

2. Create a script that reports the column name, constraint name, constraint type, search condition, and status for a specified table. You must join the `USER_CONSTRAINTS` and `USER_CONS_COLUMNS` tables to obtain all of this information. Prompt the user to enter the table name. Save the script in a file named `lab_11_02.sql`.

   For example, if the user enters `DEPARTMENTS`, the following output results:

COLUMN_NAME	CONSTRAINT_NAME	CON	SEARCH_CONDITION	STATUS
DEPARTMENT_NAME	DEPT_NAME_NN	C	"DEPARTMENT_NAME" IS NOT NULL	ENABLED
DEPARTMENT_ID	DEPT_ID_PK	P		ENABLED
LOCATION_ID	DEPT_LOC_FK	R		ENABLED
MANAGER_ID	DEPT_MGR_FK	R		ENABLED

3. Add a comment to the `DEPARTMENTS` table. Then query the `USER_TAB_COMMENTS` view to verify that the comment is present.

COMMENTS
Company department information including name, code, and location.

4. Find the names of all synonyms that are in your schema.

SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK
EMP	ORA1	EMPLOYEES	

Practice 11 (continued)

5. You need to determine the names and definitions of all of the views in your schema. Create a report that retrieves view information: the view name and text from the `USER_VIEWS` data dictionary view.
Note: Another view already exists. The `EMP_DETAILS_VIEW` was created as part of your schema. Also, if you completed practice 10, you will see the `DEPT50` view.
Note: To see more contents of a `LONG` column, use the *i*SQL*Plus command `SET LONG` *n*, where *n* is the value of the number of characters of the `LONG` column that you want to see.

VIEW_NAME	TEXT
EMPLOYEES_VU	SELECT employee_id, last_name employee, department_id FROM employees
EMP_DETAILS_VIEW	SELECT e.employee_id, e.job_id, e.manager_id, e.department_id, d.locat ion_id, l.country_id, e.first_name, e.last_name, e.salary, e.commissio n_pct, d.department_name, j.job_title, l.city, l.state_province, c.cou ntry_name, r.region_name FROM employees e, departments d, jobs j, loca tions l, countries c, regions r WHERE e.department_id = d.department_id AN D d.location_id = l.location_id AND l.country_id = c.country_id AND c.region_id = r.region_id AND j.job_id = e.job_id WITH READ ONLY

6. Find the names of your sequences. Write a query in a script to display the following information about your sequences: sequence name, maximum value, increment size, and last number. Name the script `lab_11_06.sql`. Run the statement in your script.

SEQUENCE_NAME	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
DEPARTMENTS_SEQ	9990	10	280
DEPT_ID_SEQ	1000	10	200
EMPLOYEES_SEQ	1.0000E+27	1	207
LOCATIONS_SEQ	9900	100	3300