Oracle Database: SQL Fundamentals II

Student Guide

D64260GC11
Edition 1.1
March 2012
D76289

ORACLE

Author

Sharon Sophia Stephen

Technical Contributors and Reviewers

Alan Paulson
Brian Pottle
Bryan Roberts
Claire Bennett
Diganta Choudhury
Gerlinde Frenzen
James Spiller
Jenny Tsai Smith
Joel Goodman
Ken Cooper
Kimseong Loh
Laszlo Czinkoczki
Laura Garza
Lex van der Werff
Lori Tritz
Manish Pawar
Marcie Young
Nancy Greenberg
Richard Green
Srinivas Putrevu
Swarnapriya Shridhar
Timothy McGlue
Yanti Chang

Editors

Aju Kumar
Smita Kommini

Graphic Designer

Seema Bopaiah

Publishers

Giri Venugopal
Nita Brozowski
Durga Ramesh

Contents

1

Introduction

Lesson Objectives

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

- Discuss the goals of the course
- Describe the database schema and tables that are used in the course
- Identify the available environments that can be used in the course
- Review some of the basic concepts of SQL

Lesson Agenda

- Course objectives and course agenda
- The database schema and appendixes used in the course and the available development environment in this course
- Review of some basic concepts of SQL
- Oracle Database 11*g* documentation and additional resources

Course Objectives

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

- Control database access to specific objects
- Add new users with different levels of access privileges
- Manage schema objects
- Manage objects with data dictionary views
- Manipulate large data sets in the Oracle database by using subqueries
- Manage data in different time zones
- Write multiple-column subqueries
- Use scalar and correlated subqueries
- Use the regular expression support in SQL

ORACLE

Course Prerequisites

The *Oracle Database: SQL Fundamentals I* course is a prerequisite for this course.

The *Oracle Database: SQL Fundamentals I* course offers you an introduction to Oracle Database technology. In this course, 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.

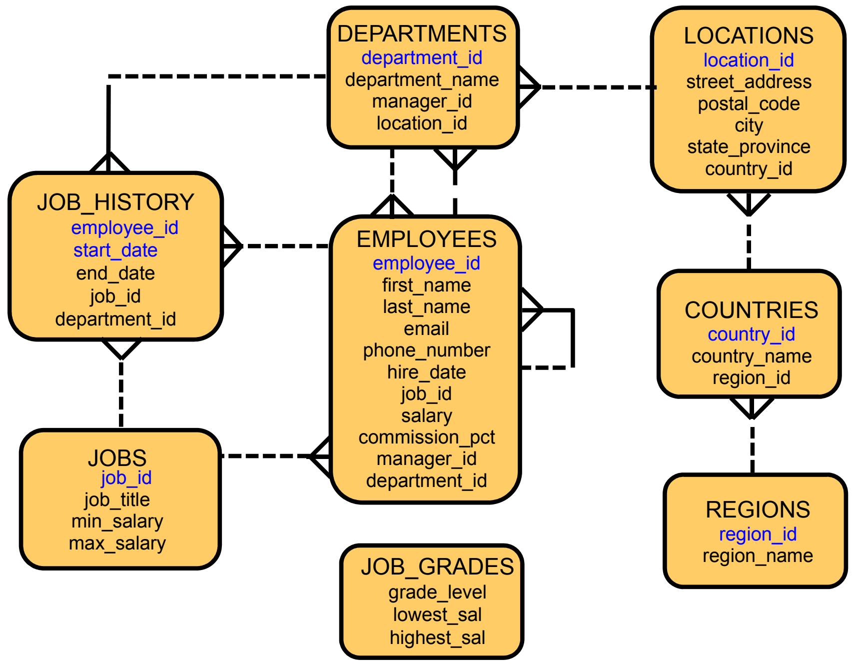
Course Agenda

- Day 1:
  - Introduction
  - Controlling User Access
  - Managing Schema Objects
  - Managing Objects with Data Dictionary Views
- Day 2:
  - Manipulating Large Data Sets
  - Managing Data in Different Time Zones
  - Retrieving Data by Using Subqueries
  - Regular Expression Support

Lesson Agenda

- Course objectives and course agenda
- The database schema and appendixes used in the course and the available development environment in this course
- Review of some basic concepts of SQL
- Oracle Database 11*g* documentation and additional resources

Tables Used in This Course



This course uses data from the following tables:

Table Descriptions

- The EMPLOYEES table contains information about all the employees, such as their first and last names, job IDs, salaries, hire dates, department IDs, and manager IDs. This table is a child of the DEPARTMENTS table.
- The DEPARTMENTS table contains information such as the department ID, department name, manager ID, and location ID. This table is the primary key table to the EMPLOYEES table.
- The LOCATIONS table contains department location information. It contains location ID, street address, city, state province, postal code, and country ID information. It is the primary key table to the DEPARTMENTS table and is a child of the COUNTRIES table.
- The COUNTRIES table contains the country names, country IDs, and region IDs. It is a child of the REGIONS table. This table is the primary key table to the LOCATIONS table.
- The REGIONS table contains region IDs and region names of the various countries. It is a primary key table to the COUNTRIES table.

- The `JOB_GRADES` table identifies a salary range per job grade. The salary ranges do not overlap.
- The `JOB_HISTORY` table stores job history of the employees.
- The `JOBS` table contains job titles and salary ranges.

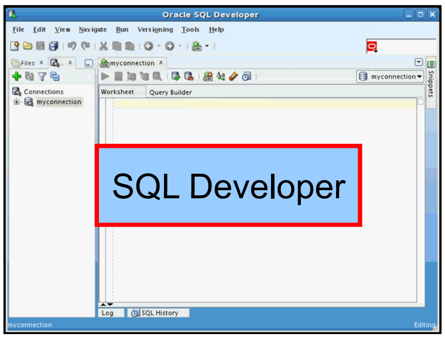
Appendixes and Practices Used in the Course

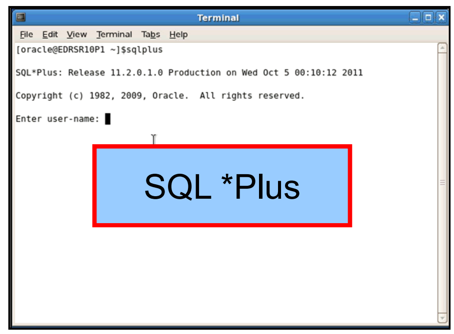
- Appendix A: Table Descriptions
- Appendix B: Using SQL Developer
- Appendix C: Using SQL*Plus
- Appendix D: Using JDeveloper
- Appendix E: Generating Reports by Grouping Related Data
- Appendix F: Hierarchical Retrieval
- Appendix G: Writing Advanced Scripts
- Appendix H: Oracle Database Architectural Components
- Activity Guide
  - Practices and Solutions
  - Additional Practices and Solutions

Development Environments

There are two development environments for this course:

- The primary tool is Oracle SQL Developer.
- You can also use SQL*Plus command-line interface.





ORACLE

SQL Developer

This course has been developed using Oracle SQL Developer as the tool for running the SQL statements discussed in the examples in the slide and the practices.

- SQL Developer version 1.5.4 is shipped with Oracle Database 11*g* Release 2, and is the default tool for this class.
- In addition, SQL Developer version 3.1 is also available on the classroom machine, and may be installed for use. At the time of publication of this course, version 3.0 was the latest release of SQL Developer.

SQL*Plus

The SQL*Plus environment may also be used to run all SQL commands covered in this course.

Note

- See Appendix B titled “Using SQL Developer” for information about using SQL Developer, including simple instructions on installing version 3.0.
- See Appendix C titled “Using SQL*Plus” for information about using SQL*Plus.

Lesson Agenda

- Course objectives and course agenda
- The database schema and appendixes used in the course and the available development environment in this course
- Review of some basic concepts of SQL
- Oracle Database 11*g* documentation and additional resources

The next few slides provide a brief overview of some of the basic concepts that you learned in the course titled *Oracle Database: SQL Fundamentals I*.

Review of Restricting Data

- Restrict the rows that are returned by using the `WHERE` clause.
- Use comparison conditions to compare one expression with another value or expression.

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

- Use logical conditions to combine the result of two component conditions and produce a single result based on those conditions.

ORACLE

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.

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

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

You use comparison conditions in the `WHERE` clause in the following format:

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

Apart from those mentioned in the slide, you use other comparison conditions such as `=`, `<`, `>`, `<>`, `<=`, and `>=`.

Three logical operators are available in SQL:

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

Review of Sorting Data

- 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
1	King	AD_PRES	90	17-JUN-87
2	Whalen	AD_ASST	10	17-SEP-87
3	Kochhar	AD_VP	90	21-SEP-89
4	Hunold	IT_PROG	60	03-JAN-90
5	Ernst	IT_PROG	60	21-MAY-91
6	De Haan	AD_VP	90	13-JAN-93

...

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

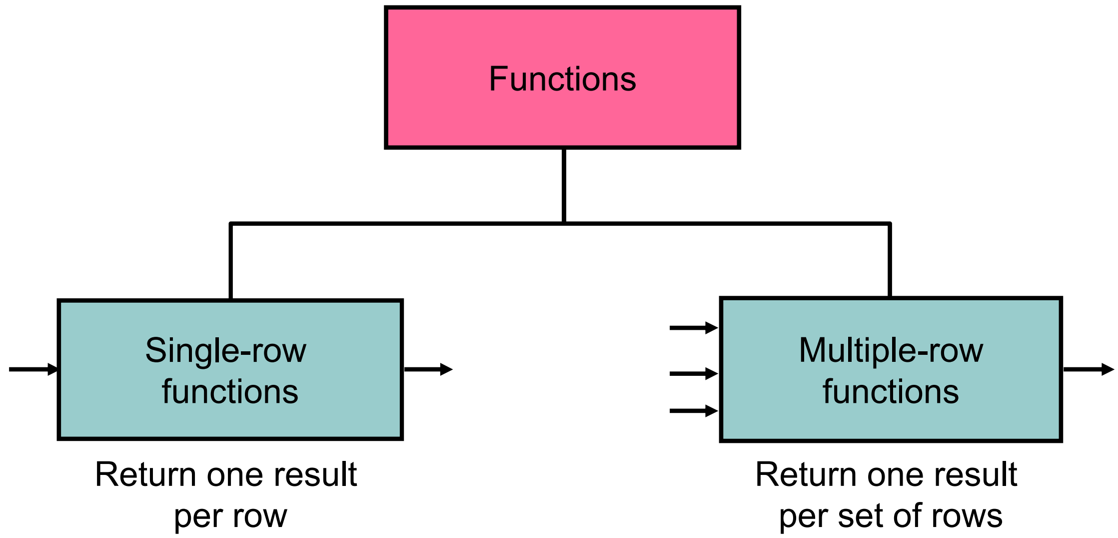
```
SELECT          expr
FROM            table
[WHERE          condition(s)]
[ORDER BY  {column, expr, numeric_position} [ASC|DESC]];
```

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.

Review of SQL Functions



There are two types of functions:

- Single-row functions
- Multiple-row functions

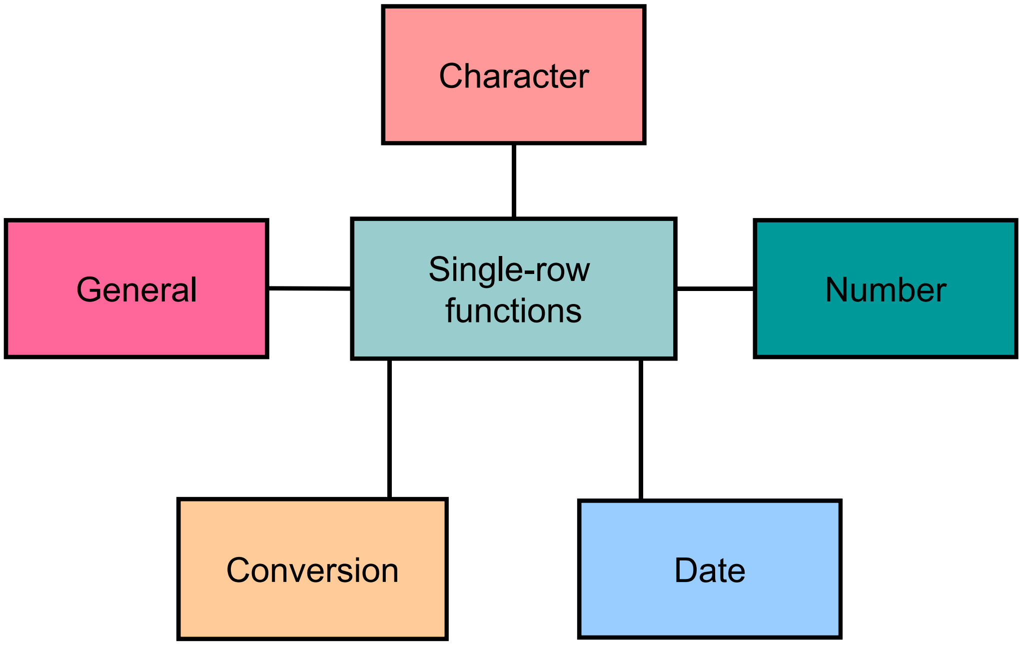
Single-Row Functions

These functions operate on single rows only and return one result per row. There are different types of single-row functions such as character, number, date, conversion, and general functions.

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*.

Review of Single-Row Functions

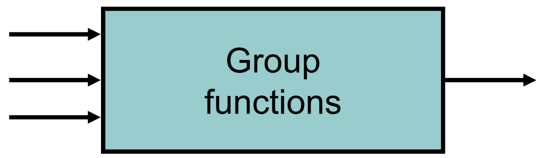


The following are different types of single-row functions:

- **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 the `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`

Review of Types of Group Functions

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



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>AVG([DISTINCT|ALL]n)</td><td>Average value of n, ignoring null values</td></tr>
<tr><td>COUNT({*|[DISTINCT|ALL]expr})</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>MAX([DISTINCT|ALL]expr)</td><td>Maximum value of expr, ignoring null values</td></tr>
<tr><td>MIN([DISTINCT|ALL]expr)</td><td>Minimum value of expr, ignoring null values</td></tr>
<tr><td>STDDEV([DISTINCT|ALL]n)</td><td>Standard deviation of n, ignoring null values</td></tr>
<tr><td>SUM([DISTINCT|ALL]n)</td><td>Sum values of n, ignoring null values</td></tr>
<tr><td>VARIANCE([DISTINCT|ALL]n)</td><td>Variance of n, ignoring null values</td></tr>
</table>

Review of Using Subqueries

- A subquery is a `SELECT` statement nested in a clause of another `SELECT` statement.
- Syntax:

```
SELECT select_list
FROM   table
WHERE  expr operator
                    (SELECT select_list
                     FROM   table   );
```

- Types of subqueries:

Single-row subquery	Multiple-row subquery
Returns only one row	Returns more than one row
Uses single-row comparison operators	Uses multiple-row comparison operators

You can build powerful statements out of simple ones by using subqueries. Subqueries are useful when a query is based on a search criterion with unknown intermediate values.

You can place the subquery in a number of SQL clauses, including the following:

- `WHERE` clause
- `HAVING` clause
- `FROM` clause

The subquery (inner query) executes once before the main query (outer query). The result of the subquery is used by the main query.

A single-row subquery uses a single-row operator such as `=`, `>`, `<`, `>=`, `<=`, or `<>`. With a multiple-row subquery, you use a multiple-row operator such as `IN`, `ANY`, or `ALL`.

Example: Display details of employees whose salary is equal to the minimum salary.

```
SELECT last_name, salary, job_id
FROM   employees
WHERE  salary = (SELECT MIN(salary)
                 FROM   employees );
```

In the example, the `MIN` group function returns a single value to the outer query.

Note: In this course, you learn how to use multiple-column subqueries. Multiple-column subqueries return more than one column from the inner `SELECT` statement.

Review of Manipulating Data

A data manipulation language (DML) statement is executed when you:

- Add new rows to a table
- Modify existing rows in a table
- Remove existing rows from a table

Function	Description
`INSERT`	Adds a new row to the table
`UPDATE`	Modifies existing rows in the table
`DELETE`	Removes existing rows from the table
`MERGE`	Updates, inserts, or deletes a row conditionally into/from a table

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. You can add new rows to a table by using the `INSERT` statement. With the following syntax, only one row is inserted at a time.

```
INSERT INTO table [(column [, column…])]
VALUES      (value[, value...]);
```

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. 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.

The `UPDATE` statement modifies specific rows if you specify the `WHERE` clause.

```
UPDATE table
SET column = value [, column = value, ...]
[WHERE condition];
```

You can remove existing rows by using the `DELETE` statement. You can delete specific rows by specifying the `WHERE` clause in the `DELETE` statement.

```
DELETE [FROM] table
[WHERE condition];
```

You learn about the `MERGE` statement in the lesson titled “Manipulating Large Data Sets.”

Lesson Agenda

- Course objectives and course agenda
- The database schema and appendixes used in the course and the available development environment in this course
- Review of some basic concepts of SQL
- Oracle Database 11*g* documentation and additional resources

Oracle Database SQL Documentation

- *Oracle Database New Features Guide*
- *Oracle Database Reference*
- *Oracle Database SQL Language Reference*
- *Oracle Database Concepts*
- *Oracle Database SQL Developer User's Guide Release 3.1*

Navigate to http://www.oracle.com/pls/db102/homepage to access the Oracle Database 10*g* documentation library.

Navigate to http://www.oracle.com/pls/db112/homepage to access the Oracle Database 11*g* Release 2 documentation library.

Additional Resources

For additional information about the new Oracle 11*g* SQL, refer to the following:

- *Oracle Database 11g: New Features eStudies*
- *Oracle by Example series (OBE): Oracle Database 11g*

Summary

In this lesson, you should have learned the following:

- The course objectives
- The sample tables used in the course

Practice 1: Overview

This practice covers the following topics:

- Running the SQL Developer online tutorial
- Starting SQL Developer and creating a new database connection and browsing the tables
- Executing SQL statements using the SQL Worksheet
- Reviewing the basic concepts of SQL

In this practice, you use SQL Developer to execute SQL statements.

Note: All written practices use SQL Developer as the development environment. Although it is recommended that you use SQL Developer, you can also use the SQL*Plus environment that is available in this course.

2

Controlling User Access

Objectives

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

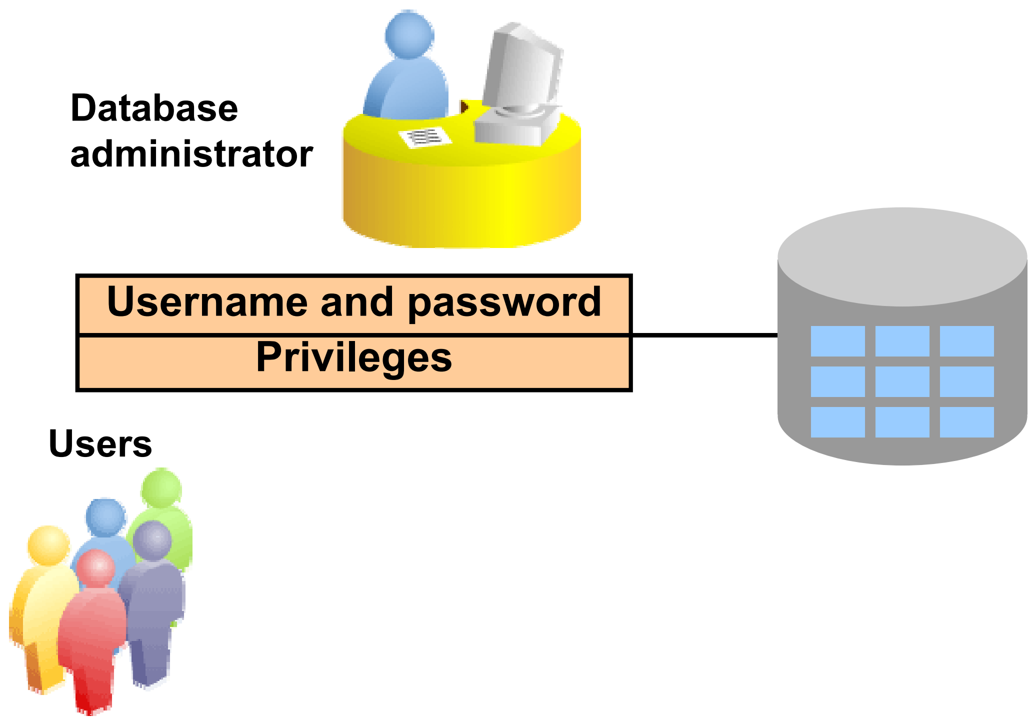
- Differentiate system privileges from object privileges
- Grant privileges on tables
- Grant roles
- Distinguish between privileges and roles

In this lesson, you learn how to control database access to specific objects and add new users with different levels of access privileges.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges

Controlling User Access



In a multiple-user environment, you want to maintain security of database access and use. With Oracle Server database security, you can do the following:

- Control database access.
- Give access to specific objects in the database.
- Confirm given and received privileges with the Oracle data dictionary.

Database security can be classified into two categories: system security and data security. System security covers access and use of the database at the system level, such as the username and password, the disk space allocated to users, and the system operations that users can perform. Database security covers access and use of the database objects and the actions that those users can perform on the objects.

Privileges

- Database security:
  - System security
  - Data security
- System privileges: Performing a particular action within the database
- Object privileges: Manipulating the content of the database objects
- Schemas: Collection of objects such as tables, views, and sequences

A privilege is the right to execute particular SQL statements. The database administrator (DBA) is a high-level user with the ability to create users and grant users access to the database and its objects. Users require *system privileges* to gain access to the database and *object privileges* to manipulate the content of the objects in the database. Users can also be given the privilege to grant additional privileges to other users or to *roles*, which are named groups of related privileges.

Schemas

A *schema* is a collection of objects such as tables, views, and sequences. The schema is owned by a database user and has the same name as that user.

A system privilege is the right to perform a particular action, or to perform an action on any schema objects of a particular type. An object privilege provides the user the ability to perform a particular action on a specific schema object.

For more information, see the reference manual *Oracle Database 2 Day DBA for 10g or 11g database*.

System Privileges

- More than 200 privileges are available.
- The database administrator has high-level system privileges for tasks such as:
  - Creating new users
  - Removing users
  - Removing tables
  - Backing up tables

More than 200 distinct system privileges are available for users and roles. Typically, system privileges are provided by the database administrator (DBA).

The table SYSTEM_PRIVILEGE_MAP contains all the system privileges available, based on the version release. This table is also used to map privilege type numbers to type names.

Typical DBA Privileges

System Privilege	Operations Authorized
CREATE USER	Grantee can create other Oracle users.
DROP USER	Grantee can drop another user.
DROP ANY TABLE	Grantee can drop a table in any schema.
BACKUP ANY TABLE	Grantee can back up any table in any schema with the export utility.
SELECT ANY TABLE	Grantee can query tables, views, or materialized views in any schema.
CREATE ANY TABLE	Grantee can create tables in any schema.

Creating Users

The DBA creates users with the `CREATE USER` statement.

```
CREATE USER user
IDENTIFIED BY   password;
```

```
CREATE USER  demo
IDENTIFIED BY demo;
```


The DBA creates the user by executing the `CREATE USER` statement. The user does not have any privileges at this point. The DBA can then grant privileges to that user. These privileges determine what the user can do at the database level.

The slide gives the abridged syntax for creating a user.

In the syntax:

user	Is the name of the user to be created
Password	Specifies that the user must log in with this password

For more information, see the *Oracle Database SQL Reference for 10g or 11g database*.

Note: Starting with Oracle Database 11*g*, passwords are case-sensitive.

User System Privileges

- After a user is created, the DBA can grant specific system privileges to that user.

```
GRANT privilege [, privilege...]
TO user [, user| role, PUBLIC...];
```

- An application developer, for example, may have the following system privileges:
  - `CREATE SESSION`
  - `CREATE TABLE`
  - `CREATE SEQUENCE`
  - `CREATE VIEW`
  - `CREATE PROCEDURE`

Typical User Privileges

After the DBA creates a user, the DBA can assign privileges to that user.

System Privilege	Operations Authorized
CREATE SESSION	Connect to the database.
CREATE TABLE	Create tables in the user's schema.
CREATE SEQUENCE	Create a sequence in the user's schema.
CREATE VIEW	Create a view in the user's schema.
CREATE PROCEDURE	Create a stored procedure, function, or package in the user's schema.

In the syntax:

privilege	Is the system privilege to be granted
user |role|PUBLIC	Is the name of the user, the name of the role, or PUBLIC (which designates that every user is granted the privilege)

Note: Current system privileges can be found in the SESSION_PRIVS dictionary view. Data dictionary is a collection of tables and views created and maintained by the Oracle Server. They contain information about the database.

Granting System Privileges

The DBA can grant specific system privileges to a user.

```
GRANT  create session, create table,
       create sequence, create view
TO     demo;
```

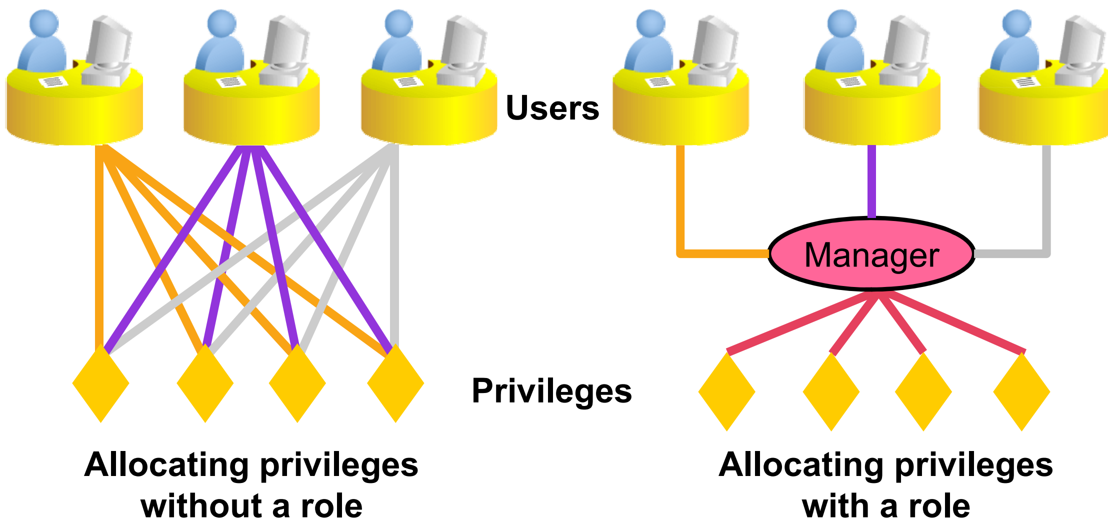

The DBA uses the `GRANT` statement to allocate system privileges to the user. After the user has been granted the privileges, the user can immediately use those privileges.

In the example in the slide, the `demo` user has been assigned the privileges to create sessions, tables, sequences, and views.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges

What Is a Role?



A role is a named group of related privileges that can be granted to the user. This method makes it easier to revoke and maintain privileges.

A user can have access to several roles, and several users can be assigned the same role. Roles are typically created for a database application.

Creating and Assigning a Role

First, the DBA must create the role. Then the DBA can assign privileges to the role and assign the role to users.

Syntax

```
CREATE   ROLE role;
```

In the syntax:

role Is the name of the role to be created

After the role is created, the DBA can use the `GRANT` statement to assign the role to users as well as assign privileges to the role. A role is not a schema object; therefore, any user can add privileges to a role.

Creating and Granting Privileges to a Role

- Create a role:

```
CREATE ROLE manager;
```

- Grant privileges to a role:

```
GRANT create table, create view
TO manager;
```

- Grant a role to users:

```
GRANT manager TO alice;
```


Creating a Role

The example in the slide creates a `manager` role and then enables the manager to create tables and views. It then grants user `alice` the role of a manager. Now `alice` can create tables and views.

If users have multiple roles granted to them, they receive all the privileges associated with all the roles.

Changing Your Password

- The DBA creates your user account and initializes your password.
- You can change your password by using the `ALTER USER` statement.

```
ALTER USER demo
IDENTIFIED BY employ;
```


The DBA creates an account and initializes a password for every user. You can change your password by using the `ALTER USER` statement.

The slide example shows that the `demo` user changes the password by using the `ALTER USER` statement.

Syntax

```
ALTER USER user IDENTIFIED BY password;
```

In the syntax:

user	Is the name of the user
password	Specifies the new password

Although this statement can be used to change your password, there are many other options. You must have the `ALTER USER` privilege to change any other option.

For more information, see the *Oracle Database SQL Reference for 10g or 11g database*.

Note: SQL*Plus has a `PASSWORD` command (`PASSW`) that can be used to change the password of a user when the user is logged in. This command is not available in SQL Developer.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges

Object Privileges

Object privilege	Table	View	Sequence
ALTER	✓		✓
DELETE	✓	✓	
INDEX	✓		
INSERT	✓	✓	
REFERENCES	✓		
SELECT	✓	✓	✓
UPDATE	✓	✓	

An *object privilege* is a privilege or right to perform a particular action on a specific table, view, sequence, or procedure. Each object has a particular set of grantable privileges. The table in the slide lists the privileges for various objects. Note that the only privileges that apply to a sequence are `SELECT` and `ALTER`. `UPDATE`, `REFERENCES`, and `INSERT` can be restricted by specifying a subset of updatable columns.

A `SELECT` privilege can be restricted by creating a view with a subset of columns and granting the `SELECT` privilege only on the view. A privilege granted on a synonym is converted to a privilege on the base table referenced by the synonym.

Note: With the `REFERENCES` privilege, you can ensure that other users can create `FOREIGN KEY` constraints that reference your table.

Object Privileges

- Object privileges vary from object to object.
- An owner has all the privileges on the object.
- An owner can give specific privileges on that owner's object.

```
GRANT        object_priv [(columns)]
ON           object
TO           {user|role|PUBLIC}
[WITH GRANT OPTION];
```


Granting Object Privileges

Different object privileges are available for different types of schema objects. A user automatically has all object privileges for schema objects contained in the user's schema. A user can grant any object privilege on any schema object that the user owns to any other user or role. If the grant includes `WITH GRANT OPTION`, the grantee can further grant the object privilege to other users; otherwise, the grantee can use the privilege but cannot grant it to other users.

In the syntax:

object_priv	Is an object privilege to be granted
ALL	Specifies all object privileges
columns	Specifies the column from a table or view on which privileges are granted
ON *object*	Is the object on which the privileges are granted
TO	Identifies to whom the privilege is granted
PUBLIC	Grants object privileges to all users
WITH GRANT OPTION	Enables the grantee to grant the object privileges to other users and roles

Note: In the syntax, *schema* is the same as the owner's name.

Granting Object Privileges

- Grant query privileges on the `EMPLOYEES` table:

```
GRANT  select
ON     employees
TO     demo;
```

- Grant privileges to update specific columns to users and roles:

```
GRANT  update (department_name, location_id)
ON     departments
TO     demo, manager;
```


Guidelines

- To grant privileges on an object, the object must be in your own schema, or you must have been granted the object privileges `WITH GRANT OPTION`.
- An object owner can grant any object privilege on the object to any other user or role of the database.
- The owner of an object automatically acquires all object privileges on that object.

The first example in the slide grants the `demo` user the privilege to query your `EMPLOYEES` table. The second example grants `UPDATE` privileges on specific columns in the `DEPARTMENTS` table to `demo` and to the `manager` role.

For example, if your schema is `oraxx`, and the `demo` user now wants to use a `SELECT` statement to obtain data from your `EMPLOYEES` table, the syntax he or she must use is:

```
SELECT  * FROM oraxx.employees;
```

Alternatively, the `demo` user can create a synonym for the table and issue a `SELECT` statement from the synonym:

```
CREATE SYNONYM emp FOR oraxx.employees;
SELECT * FROM emp;
```

Note: DBAs generally allocate system privileges; any user who owns an object can grant object privileges.

Passing On Your Privileges

- Give a user authority to pass along privileges:

```
GRANT   select, insert
ON      departments
TO      demo
WITH    GRANT OPTION;
```

- Allow all users on the system to query data from Alice's `DEPARTMENTS` table:

```
GRANT   select
ON      alice.departments
TO      PUBLIC;
```


`WITH GRANT OPTION` Keyword

A privilege that is granted with the `WITH GRANT OPTION` clause can be passed on to other users and roles by the grantee. Object privileges granted with the `WITH GRANT OPTION` clause are revoked when the grantor's privilege is revoked.

The example in the slide gives the `demo` user access to your `DEPARTMENTS` table with the privileges to query the table and add rows to the table. The example also shows that `demo` can give others these privileges.

`PUBLIC` Keyword

An owner of a table can grant access to all users by using the `PUBLIC` keyword.

The second example allows all users on the system to query data from Alice's `DEPARTMENTS` table.

Confirming Granted Privileges

Data Dictionary View	Description
ROLE_SYS_PRIVS	System privileges granted to roles
ROLE_TAB_PRIVS	Table privileges granted to roles
USER_ROLE_PRIVS	Roles accessible by the user
USER_SYS_PRIVS	System privileges granted to the user
USER_TAB_PRIVS_MADE	Object privileges granted on the user's objects
USER_TAB_PRIVS_RECD	Object privileges granted to the user
USER_COL_PRIVS_MADE	Object privileges granted on the columns of the user's objects
USER_COL_PRIVS_RECD	Object privileges granted to the user on specific columns

If you attempt to perform an unauthorized operation, such as deleting a row from a table for which you do not have the `DELETE` privilege, the Oracle server does not permit the operation to take place.

If you receive the Oracle server error message "Table or view does not exist," you have done either of the following:

- Named a table or view that does not exist
- Attempted to perform an operation on a table or view for which you do not have the appropriate privilege

The data dictionary is organized in tables and views and contains information about the database. You can access the data dictionary to view the privileges that you have. The table in the slide describes various data dictionary views.

You learn more about data dictionary views in the lesson titled "Managing Objects with Data Dictionary Views."

Note: The `ALL_TAB_PRIVS_MADE` dictionary view describes all the object grants made by the user or made on the objects owned by the user.

Lesson Agenda

- System privileges
- Creating a role
- Object privileges
- Revoking object privileges

Revoking Object Privileges

- You use the `REVOKE` statement to revoke privileges granted to other users.
- Privileges granted to others through the `WITH GRANT OPTION` clause are also revoked.

```
REVOKE {privilege [, privilege...]|ALL}
ON     object
FROM   {user[, user...]|role|PUBLIC}
[CASCADE CONSTRAINTS];
```


You can remove privileges granted to other users by using the `REVOKE` statement. When you use the `REVOKE` statement, the privileges that you specify are revoked from the users you name and from any other users to whom those privileges were granted by the revoked user.

In the syntax:

`CASCADE`	Is required to remove any referential integrity constraints made to the `CONSTRAINTS` object by means of the `REFERENCES` privilege

For more information, see the *Oracle Database SQL Reference for 10g or 11g database*.

Note: If a user were to leave the company and you revoke his or her privileges, you must regrant any privileges that this user may have granted to other users. If you drop the user account without revoking privileges from it, the system privileges granted by this user to other users are not affected by this action.

Revoking Object Privileges

Revoke the SELECT and INSERT privileges given to the demo user on the DEPARTMENTS table.

```
REVOKE  select, insert
ON      departments
FROM    demo;
```


The example in the slide revokes SELECT and INSERT privileges given to the demo user on the DEPARTMENTS table.

Note: If a user is granted a privilege with the WITH GRANT OPTION clause, that user can also grant the privilege with the WITH GRANT OPTION clause, so that a long chain of grantees is possible, but no circular grants (granting to a grant ancestor) are permitted. If the owner revokes a privilege from a user who granted the privilege to other users, the revoking cascades to all the privileges granted.

For example, if user A grants a SELECT privilege on a table to user B including the WITH GRANT OPTION clause, user B can grant to user C the SELECT privilege with the WITH GRANT OPTION clause as well, and user C can then grant to user D the SELECT privilege. If user A revokes privileges from user B, the privileges granted to users C and D are also revoked.

With the WITH ADMIN OPTION clause, the grantee becomes the administrator of that system privilege because the grantee gets the rights to grant or revoke any system privilege or role in the database. So, be cautious when using this clause.

Quiz

Which of the following statements are true?

a. After a user creates an object, the user can pass along any of the available object privileges to other users by using the `GRANT` statement.

b. A user can create roles by using the `CREATE ROLE` statement to pass along a collection of system or object privileges to other users.

c. Users can change their own passwords.

d. Users can view the privileges granted to them and those that are granted on their objects.

Answer: a, c, d

Summary

In this lesson, you should have learned how to:

- Differentiate system privileges from object privileges
- Grant privileges on tables
- Grant roles
- Distinguish between privileges and roles

DBAs establish initial database security for users by assigning privileges to the users.

- The DBA creates users who must have a password. The DBA is also responsible for establishing the initial system privileges for a user.
- After the user has created an object, the user can pass along any of the available object privileges to other users or to all users by using the `GRANT` statement.
- A DBA can create roles by using the `CREATE ROLE` statement to pass along a collection of system or object privileges to multiple users. Roles make granting and revoking privileges easier to maintain.
- Users can change their passwords by using the `ALTER USER` statement.
- You can remove privileges from users by using the `REVOKE` statement.
- With data dictionary views, users can view the privileges granted to them and those that are granted on their objects.

Practice 2: Overview

This practice covers the following topics:

- Granting other users privileges to your table
- Modifying another user's table through the privileges granted to you

Team up with other students for this exercise about controlling access to database objects.

3

Managing Schema Objects

Objectives

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

- Add constraints
- Create indexes
- Create indexes by using the `CREATE TABLE` statement
- Create function-based indexes
- Drop columns and set columns as `UNUSED`
- Perform `FLASHBACK` operations
- Create and use external tables

This lesson contains information about creating indexes and constraints and altering existing objects. You also learn about external tables and the provision to name the index at the time of creating a `PRIMARY KEY` constraint.

Lesson Agenda

- Using the `ALTER TABLE` statement to add, modify, and drop a column
- Managing constraints:
  - Adding and dropping a constraint
  - Deferring constraints
  - Enabling and disabling a constraint
- Creating indexes:
  - Using the `CREATE TABLE` statement
  - Creating function-based indexes
  - Removing an index
- Performing flashback operations
- Creating and using temporary tables
- Creating and using external tables

`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

After you create a table, you may need to change the table structure because you omitted a column, your column definition needs to be changed, or you need to remove columns. You can do this by using the `ALTER TABLE` statement.

`ALTER TABLE` Statement

Use the `ALTER TABLE` statement to add, modify, or drop columns:

```
ALTER TABLE table
ADD         (column datatype [DEFAULT expr]
            [, column datatype]...);
```

```
ALTER TABLE table
MODIFY      (column datatype [DEFAULT expr]
            [, column datatype]...);
```

```
ALTER TABLE table
DROP (column [, column] …);
```

ORACLE

You can add columns to a table, modify columns, and drop columns from a table by using the `ALTER TABLE` statement.

In the syntax:

`table`	Is the name of the table
`ADD|MODIFY|DROP`	Is the type of modification
`column`	Is the name of the column
`datatype`	Is the data type and length of the column
`DEFAULT` *`expr`*	Specifies the default value for a column

Adding a Column

- You use the `ADD` clause to add columns:

```
ALTER TABLE dept80
ADD         (job_id VARCHAR2(9));
```

```
table DEPT80 altered.
```

- The new column becomes the last column:

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
1	145	Russell	14000	01-OCT-96	(null)
2	146	Partners	13500	05-JAN-97	(null)
3	147	Errazuriz	12000	10-MAR-97	(null)
4	148	Cambrault	11000	15-OCT-99	(null)
5	149	Zlotkey	10500	29-JAN-00	(null)

Guidelines for Adding a Column

- You can add or modify columns.
- You cannot specify where the column is to appear. The new column becomes the last column.

The example in the slide adds a column named `JOB_ID` to the `DEPT80` table. The `JOB_ID` column becomes the last column in the table.

Note: If a table already contains rows when a column is added, the new column is initially null or takes the default value for all the rows. You can add a mandatory `NOT NULL` column to a table that contains data in the other columns only if you specify a default value. You can add a `NOT NULL` column to an empty table without the default value.

Modifying a Column

- You can change a column's data type, size, and default value.

```
ALTER TABLE dept80
MODIFY      (last_name VARCHAR2(30));
```

```
table DEPT80 altered.
```

- A change to the default value affects only subsequent insertions to the table.

You can modify a column definition by using the `ALTER TABLE` statement with the `MODIFY` clause. Column modification can include changes to a column's data type, size, and default value.

Guidelines

- You can increase the width or precision of a numeric column.
- You can increase the width of character columns.
- You can decrease the width of a column if:
  - The column contains only null values
  - The table has no rows
  - The decrease in column width is not less than the existing values in that column
- You can change the data type if the column contains only null values. The exception to this is `CHAR`-to-`VARCHAR2` conversions, which can be done with data in the columns.
- You can convert a `CHAR` column to the `VARCHAR2` data type or convert a `VARCHAR2` column to the `CHAR` data type only if the column contains null values or if you do not change the size.
- A change to the default value of a column affects only subsequent insertions to the table.

Dropping a Column

Use the `DROP COLUMN` clause to drop columns that you no longer need from the table:

```
ALTER TABLE  dept80
DROP  (job_id);
```

```
table DEPT80 altered.
```

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
1	145	Russell	14000	01-OCT-96
2	146	Partners	13500	05-JAN-97
3	147	Errazuriz	12000	10-MAR-97
4	148	Cambrault	11000	15-OCT-99
5	149	Zlotkey	10500	29-JAN-00

You can drop a column from a table by using the `ALTER TABLE` statement with the `DROP COLUMN` clause.

Guidelines

- The column may or may not contain data.
- Using the `ALTER TABLE DROP COLUMN` statement, only one column can be dropped at a time.
- The table must have at least one column remaining in it after it is altered.
- After a column is dropped, it cannot be recovered.
- A primary key that is referenced by another column cannot be dropped, unless the cascade option is added.
- Dropping a column can take a while if the column has a large number of values. In this case, it may be better to set it to be unused and drop it when there are fewer users on the system to avoid extended locks.

Note: Certain columns can never be dropped, such as columns that form part of the partitioning key of a partitioned table or columns that form part of the `PRIMARY KEY` of an index-organized table. For more information about index-organized tables and partitioned tables, refer to *Oracle Database Concepts* and *Oracle Database Administrator's Guide.*

SET UNUSED Option

- You use the SET UNUSED option to mark one or more columns as unused.
- You use the DROP UNUSED COLUMNS option to remove the columns that are marked as unused.

```
ALTER TABLE  <table_name>
SET   UNUSED(<column name> [ , <column_name>]);
OR
ALTER TABLE  <table_name>
SET   UNUSED COLUMN <column name> [,<column_name>];
```

```
ALTER TABLE <table_name>
DROP  UNUSED COLUMNS;
```


The SET UNUSED option marks one or more columns as unused so that they can be dropped when the demand on system resources is lower. Specifying this clause does not actually remove the target columns from each row in the table (that is, it does not restore the disk space used by these columns). Therefore, the response time is faster than if you executed the DROP clause. Unused columns are treated as if they were dropped, even though their column data remains in the table's rows. After a column has been marked as unused, you have no access to that column. A SELECT * query will not retrieve data from unused columns. In addition, the names and types of columns marked as unused will not be displayed during a DESCRIBE statement, and you can add to the table a new column with the same name as an unused column. The SET UNUSED information is stored in the USER_UNUSED_COL_TABS dictionary view.

Note: The guidelines for setting a column to be UNUSED are similar to those for dropping a column.

`DROP UNUSED COLUMNS` Option

`DROP UNUSED COLUMNS` removes from the table all columns currently marked as unused. You can use this statement when you want to reclaim the extra disk space from unused columns in the table. If the table contains no unused columns, the statement returns with no errors.

```
ALTER TABLE  dept80
SET   UNUSED (last_name);

table DEPT80 altered.



ALTER TABLE  dept80
DROP  UNUSED COLUMNS;

table DEPT80 altered.
```

Lesson Agenda

- Using the `ALTER TABLE` statement to add, modify, and drop a column
- Managing constraints:
  - Adding and dropping a constraint
  - Deferring constraints
  - Enabling and disabling a constraint
- Creating indexes:
  - Using the `CREATE TABLE` statement
  - Creating function-based indexes
  - Removing an index
- Performing flashback operations
- Creating and using temporary tables
- Creating and using external tables

Adding a Constraint Syntax

Use the ALTER TABLE statement to:

- Add or drop a constraint, but not modify its structure
- Enable or disable constraints
- Add a NOT NULL constraint by using the MODIFY clause

```
ALTER TABLE  <table_name>
ADD [CONSTRAINT <constraint_name>]
type (<column_name>);
```


You can add a constraint for existing tables by using the ALTER TABLE statement with the ADD clause.

In the syntax:

table	Is the name of the table
constraint	Is the name of the constraint
type	Is the constraint type
column	Is the name of the column affected by the constraint

The constraint name syntax is optional, although recommended. If you do not name your constraints, the system generates constraint names.

Guidelines

- You can add, drop, enable, or disable a constraint, but you cannot modify its structure.
- You can add a NOT NULL constraint to an existing column by using the MODIFY clause of the ALTER TABLE statement.

Note: You can define a NOT NULL column only if the table is empty or if the column has a value for every row.

Adding a Constraint

Add a `FOREIGN KEY` constraint to the `EMP2` table indicating that a manager must already exist as a valid employee in the `EMP2` table.

```
ALTER TABLE emp2
MODIFY employee_id PRIMARY KEY;
```

```
table EMP2 altered.
```

```
ALTER TABLE emp2
ADD CONSTRAINT emp_mgr_fk
  FOREIGN KEY(manager_id)
  REFERENCES emp2(employee_id);
```

```
table EMP2 altered.
```


The first example in the slide modifies the `EMP2` table to add a `PRIMARY KEY` constraint on the `EMPLOYEE_ID` column. Note that because no constraint name is provided, the constraint is automatically named by the Oracle Server. The second example in the slide creates a `FOREIGN KEY` constraint on the `EMP2` table. The constraint ensures that a manager exists as a valid employee in the `EMP2` table.

ON DELETE Clause

- Use the ON DELETE CASCADE clause to delete child rows when a parent key is deleted:

```
ALTER TABLE emp2 ADD CONSTRAINT emp_dt_fk
FOREIGN KEY (Department_id)
REFERENCES departments(department_id) ON DELETE CASCADE;
```

```
table EMP2 altered.
```

- Use the ON DELETE SET NULL clause to set the child rows value to null when a parent key is deleted:

```
ALTER TABLE emp2 ADD CONSTRAINT emp_dt_fk
FOREIGN KEY (Department_id)
REFERENCES departments(department_id) ON DELETE SET NULL;
```

```
table EMP2 altered.
```


ON DELETE

By using the ON DELETE clause, you can determine how Oracle Database handles referential integrity if you remove a referenced primary or unique key value.

ON DELETE CASCADE

The ON DELETE CASCADE action allows parent key data that is referenced from the child table to be deleted, but not updated. When data in the parent key is deleted, all the rows in the child table that depend on the deleted parent key values are also deleted. To specify this referential action, include the ON DELETE CASCADE option in the definition of the FOREIGN KEY constraint.

ON DELETE SET NULL

When data in the parent key is deleted, the ON DELETE SET NULL action causes all the rows in the child table that depend on the deleted parent key value to be converted to null.

If you omit this clause, Oracle does not allow you to delete referenced key values in the parent table that have dependent rows in the child table.

Deferring Constraints

Constraints can have the following attributes:

- `DEFERRABLE` or `NOT DEFERRABLE`
- `INITIALLY DEFERRED` or `INITIALLY IMMEDIATE`

```
ALTER TABLE dept2
ADD CONSTRAINT dept2_id_pk
PRIMARY KEY (department_id)
DEFERRABLE INITIALLY DEFERRED
```

Deferring constraint on creation

```
SET CONSTRAINTS dept2_id_pk IMMEDIATE
```

Changing a specific constraint attribute

```
ALTER SESSION
SET CONSTRAINTS= IMMEDIATE
```

Changing all constraints for a session

ORACLE

You can defer checking constraints for validity until the end of the transaction. A constraint is deferred if the system does not check whether the constraint is satisfied, until a `COMMIT` statement is submitted. If a deferred constraint is violated, the database returns an error and the transaction is not committed and it is rolled back. If a constraint is immediate (not deferred), it is checked at the end of each statement. If it is violated, the statement is rolled back immediately. If a constraint causes an action (for example, `DELETE CASCADE`), that action is always taken as part of the statement that caused it, whether the constraint is deferred or immediate. Use the `SET CONSTRAINTS` statement to specify, for a particular transaction, whether a deferrable constraint is checked following each data manipulation language (DML) statement or when the transaction is committed. To create deferrable constraints, you must create a nonunique index for that constraint.

You can define constraints as either deferrable or `NOT DEFERRABLE` (default), and either initially deferred or `INITIALLY IMMEDIATE` (default). These attributes can be different for each constraint.

Usage scenario: Company policy dictates that department number 40 should be changed to 45. Changing the `DEPARTMENT_ID` column affects employees assigned to this department. Therefore, you make the `PRIMARY KEY` and `FOREIGN KEYs` deferrable and initially deferred. You update both department and employee information, and at the time of commit, all the rows are validated.

Difference Between `INITIALLY DEFERRED` and `INITIALLY IMMEDIATE`

`INITIALLY DEFERRED`	Waits to check the constraint until the transaction ends
`INITIALLY IMMEDIATE`	Checks the constraint at the end of the statement execution

```
CREATE TABLE emp_new_sal (salary NUMBER
        CONSTRAINT sal_ck
        CHECK (salary > 100)
        DEFERRABLE INITIALLY IMMEDIATE,
        bonus NUMBER
        CONSTRAINT bonus_ck
        CHECK (bonus > 0 )
        DEFERRABLE INITIALLY DEFERRED );
```

```
table EMP_NEW_SAL created.
```


A constraint that is defined as deferrable can be specified as either `INITIALLY DEFERRED` or `INITIALLY IMMEDIATE`. The `INITIALLY IMMEDIATE` clause is the default.

In the slide example:

- The `sal_ck` constraint is created as `DEFERRABLE INITIALLY IMMEDIATE`
- The `bonus_ck` constraint is created as `DEFERRABLE INITIALLY DEFERRED`

After creating the `emp_new_sal` table, as shown in the slide, you attempt to insert values into the table and observe the results. When both the `sal_ck` and `bonus_ck` constraints are satisfied, the rows are inserted without an error.

Example 1: Insert a row that violates `sal_ck`. In the `CREATE TABLE` statement, `sal_ck` is specified as an initially immediate constraint. This means that the constraint is verified immediately after the `INSERT` statement and you observe an error.

```
INSERT INTO emp_new_sal VALUES(90,5);
```

```
SQL Error: ORA-02290: check constraint (ORA21.SAL_CK) violated
02290. 00000 -  "check constraint (%s.%s) violated"
```

Example 2: Insert a row that violates `bonus_ck`. In the `CREATE TABLE` statement, `bonus_ck` is specified as deferrable and also initially deferred. Therefore, the constraint is not verified until you `COMMIT` or set the constraint state back to immediate.

```
INSERT INTO emp_new_sal VALUES(110, -1);
```

```
1 rows inserted
```

The row insertion is successful. But you observe an error when you commit the transaction.

```
COMMIT;
```

```
SQL Error: ORA-02091: transaction rolled back
ORA-02290: check constraint (ORA21.BONUS_CK) violated
02091. 00000 -  "transaction rolled back"
```

The commit failed due to constraint violation. Therefore, at this point, the transaction is rolled back by the database.

Example 3: Set the DEFERRED status to all constraints that can be deferred. Note that you can also set the DEFERRED status to a single constraint if required.

```
SET CONSTRAINTS ALL DEFERRED;
```

```
constraints ALL succeeded.
```

Now, if you attempt to insert a row that violates the sal_ck constraint, the statement is executed successfully.

```
INSERT INTO emp_new_sal VALUES(90,5);
```

```
1 rows inserted
```

However, you observe an error when you commit the transaction. The transaction fails and is rolled back. This is because both the constraints are checked upon COMMIT.

```
COMMIT;
```

```
SQL Error: ORA-02091: transaction rolled back
ORA-02290: check constraint (ORA21.SAL_CK) violated
02091. 00000 -  "transaction rolled back"
```

Example 4: Set the IMMEDIATE status to both the constraints that were set as DEFERRED in the previous example.

```
SET CONSTRAINTS ALL IMMEDIATE;
```

```
constraints ALL succeeded.
```

You observe an error if you attempt to insert a row that violates either sal_ck or bonus_ck.

```
INSERT INTO emp_new_sal VALUES(110, -1);
```

```
SQL Error: ORA-02290: check constraint (ORA21.BONUS_CK) violated
02290. 00000 -  "check constraint (%s.%s) violated"
```

Note: If you create a table without specifying constraint deferability, the constraint is checked immediately at the end of each statement. For example, with the CREATE TABLE statement of the newemp_details table, if you do not specify the newemp_det_pk constraint deferability, the constraint is checked immediately.

```
CREATE TABLE newemp_details(emp_id NUMBER, emp_name
VARCHAR2(20),
CONSTRAINT newemp_det_pk PRIMARY KEY(emp_id));
```

When you attempt to defer the newemp_det_pk constraint that is not deferrable, you observe the following error:

```
SET CONSTRAINT newemp_det_pk DEFERRED;
```

```
SQL Error: ORA-02447: cannot defer a constraint that is not deferrable
```

Dropping a Constraint

- Remove the manager constraint from the EMP2 table:

```
ALTER TABLE emp2
DROP CONSTRAINT emp_mgr_fk;
```

```
table EMP2 altered.
```

- Remove the PRIMARY KEY constraint on the DEPT2 table and drop the associated FOREIGN KEY constraint on the EMP2.DEPARTMENT_ID column:

```
ALTER TABLE dept2
DROP PRIMARY KEY CASCADE;
```

```
table DEPT2 altered.
```


To drop a constraint, you can identify the constraint name from the USER_CONSTRAINTS and USER_CONS_COLUMNS data dictionary views. Then use the ALTER TABLE statement with the DROP clause. The CASCADE option of the DROP clause causes any dependent constraints also to be dropped.

Syntax

```
ALTER TABLE    table
 DROP  PRIMARY KEY | UNIQUE (column) |
       CONSTRAINT   constraint   [CASCADE];
```

In the syntax:

table	Is the name of the table
column	Is the name of the column affected by the constraint
constraint	Is the name of the constraint

When you drop an integrity constraint, that constraint is no longer enforced by the Oracle Server and is no longer available in the data dictionary.

Disabling Constraints

- Execute the `DISABLE` clause of the `ALTER TABLE` statement to deactivate an integrity constraint.
- Apply the `CASCADE` option to disable dependent integrity constraints.

```
ALTER TABLE emp2
DISABLE CONSTRAINT emp_dt_fk;
```

```
table EMP2 altered.
```


You can disable a constraint, without dropping it or re-creating it, by using the `ALTER TABLE` statement with the `DISABLE` clause.

Syntax

```
ALTER   TABLE   table
DISABLE CONSTRAINT constraint [CASCADE];
```

In the syntax:

table	Is the name of the table
constraint	Is the name of the constraint

Guidelines

- You can use the `DISABLE` clause in both the `CREATE TABLE` statement and the `ALTER TABLE` statement.
- The `CASCADE` clause disables dependent integrity constraints.
- Disabling a `UNIQUE` or `PRIMARY KEY` constraint removes the unique index.

Enabling Constraints

- Activate an integrity constraint that is currently disabled in the table definition by using the `ENABLE` clause.

```
ALTER TABLE        emp2
ENABLE CONSTRAINT emp_dt_fk;
```

```
table EMP2 altered.
```

- A `UNIQUE` index is automatically created if you enable a `UNIQUE` key or a `PRIMARY KEY` constraint.

You can enable a constraint without dropping it or re-creating it by using the `ALTER TABLE` statement with the `ENABLE` clause.

Syntax

```
ALTER   TABLE      table
ENABLE  CONSTRAINT constraint;
```

In the syntax:

table	Is the name of the table
constraint	Is the name of the constraint

Guidelines

- If you enable a constraint, that constraint applies to all the data in the table. All the data in the table must comply with the constraint.
- If you enable a `UNIQUE` key or a `PRIMARY KEY` constraint, a `UNIQUE` or `PRIMARY KEY` index is created automatically. If an index already exists, it can be used by these keys.
- You can use the `ENABLE` clause in both the `CREATE TABLE` statement and the `ALTER TABLE` statement.

- Enabling a `PRIMARY KEY` constraint that was disabled with the `CASCADE` option does not enable any `FOREIGN KEYs` that are dependent on the `PRIMARY KEY`.
- To enable a `UNIQUE` or `PRIMARY KEY` constraint, you must have the privileges necessary to create an index on the table.

Cascading Constraints

- The CASCADE CONSTRAINTS clause is used along with the DROP COLUMN clause.
- The CASCADE CONSTRAINTS clause drops all referential integrity constraints that refer to the PRIMARY and UNIQUE keys defined on the dropped columns.
- The CASCADE CONSTRAINTS clause also drops all multicolumn constraints defined on the dropped columns.

This statement illustrates the usage of the CASCADE CONSTRAINTS clause. Assume that the TEST1 table is created as follows:

```
CREATE TABLE test1 (
  col1_pk NUMBER PRIMARY KEY,
  col2_fk NUMBER,
  col1 NUMBER,
  col2 NUMBER,
  CONSTRAINT fk_constraint FOREIGN KEY (col2_fk) REFERENCES
    test1,
  CONSTRAINT ck1 CHECK (col1_pk > 0 and col1 > 0),
  CONSTRAINT ck2 CHECK (col2_fk > 0));
```

An error is returned for the following statements:

ALTER TABLE test1 DROP (col1_pk); —col1_pk is a parent key.

ALTER TABLE test1 DROP (col1); —col1 is referenced by the multicolumn constraint, ck1.

Cascading Constraints

Example:

```
ALTER TABLE emp2
DROP COLUMN employee_id CASCADE CONSTRAINTS;
```

```
table EMP2 altered.
```

```
ALTER TABLE test1
DROP (col1_pk, col2_fk, col1) CASCADE CONSTRAINTS;
```

```
table TEST1 altered.
```


Submitting the following statement drops the EMPLOYEE_ID column, the PRIMARY KEY constraint, and any FOREIGN KEY constraints referencing the PRIMARY KEY constraint for the EMP2 table:

```
ALTER TABLE emp2 DROP COLUMN employee_id CASCADE CONSTRAINTS;
```

If all columns referenced by the constraints defined on the dropped columns are also dropped, CASCADE CONSTRAINTS is not required. For example, assuming that no other referential constraints from other tables refer to the COL1_PK column, it is valid to submit the following statement without the CASCADE CONSTRAINTS clause for the TEST1 table created on the previous page:

```
ALTER TABLE test1 DROP (col1_pk, col2_fk, col1);
```

Renaming Table Columns and Constraints

Use the RENAME COLUMN clause of the ALTER TABLE statement to rename table columns.

a

```
ALTER TABLE marketing RENAME COLUMN team_id
TO id;
```

```
table MARKETING altered.
```

Use the RENAME CONSTRAINT clause of the ALTER TABLE statement to rename any existing constraint for a table.

b

```
ALTER TABLE marketing RENAME CONSTRAINT mktg_pk
TO new_mktg_pk;
```

```
table MARKETING altered.
```


When you rename a table column, the new name must not conflict with the name of any existing column in the table. You cannot use any other clauses in conjunction with the RENAME COLUMN clause.

The slide examples use the marketing table with the PRIMARY KEY mktg_pk defined on the id column.

```
CREATE TABLE marketing (team_id   NUMBER(10),
                        target   VARCHAR2(50),
CONSTRAINT mktg_pk PRIMARY KEY(team_id));
```

```
CREATE TABLE succeeded.
```

Example **a** shows that the id column of the marketing table is renamed mktg_id. Example **b** shows that mktg_pk is renamed new_mktg_pk.

When you rename any existing constraint for a table, the new name must not conflict with any of your existing constraint names. You can use the RENAME CONSTRAINT clause to rename system-generated constraint names.

Lesson Agenda

- Using the `ALTER TABLE` statement to add, modify, and drop a column
- Managing constraints:
  - Adding and dropping a constraint
  - Deferring constraints
  - Enabling and disabling a constraint
- Creating indexes:
  - Using the `CREATE TABLE` statement
  - Creating function-based indexes
  - Removing an index
- Performing flashback operations
- Creating and using temporary tables
- Creating and using external tables

Overview of Indexes

Indexes are created:

- Automatically
  - PRIMARY KEY creation
  - UNIQUE KEY creation
- Manually
  - The CREATE INDEX statement
  - The CREATE TABLE statement

Two types of indexes can be created.

- One type is a unique index. The Oracle Server automatically creates a unique index when you define a column or group of columns in a table to have a PRIMARY KEY or a UNIQUE key constraint. The name of the index is the name given to the constraint.
- The other type of index is a nonunique index, which a user can create. For example, you can create an index for a FOREIGN KEY column to be used in joins to improve retrieval speed.

You can create an index on one or more columns by issuing the CREATE INDEX statement.

For more information, see *Oracle Database SQL Reference for 10g or 11g database*.

Note: You can manually create a unique index, but it is recommended that you create a UNIQUE constraint, which implicitly creates a unique index.

CREATE INDEX with the CREATE TABLE Statement

```
CREATE TABLE NEW_EMP
(employee_id NUMBER(6)
             PRIMARY KEY USING INDEX
            (CREATE INDEX emp_id_idx ON
            NEW_EMP(employee_id)),
first_name  VARCHAR2(20),
last_name   VARCHAR2(25));
```

```
table NEW_EMP created.
```

```
SELECT INDEX_NAME, TABLE_NAME
FROM   USER_INDEXES
WHERE  TABLE_NAME = 'NEW_EMP';
```

	INDEX_NAME	TABLE_NAME
1	EMP_ID_IDX	NEW_EMP

In the example in the slide, the CREATE INDEX clause is used with the CREATE TABLE statement to create a PRIMARY KEY index explicitly. You can name your indexes at the time of PRIMARY KEY creation to be different from the name of the PRIMARY KEY constraint.

You can query the USER_INDEXES data dictionary view for information about your indexes.

Note: You learn more about USER_INDEXES in the lesson titled "Managing Objects with Data Dictionary Views."

The following example illustrates the database behavior if the index is not explicitly named:

```
CREATE TABLE EMP_UNNAMED_INDEX
   (employee_id NUMBER(6) PRIMARY KEY ,
    first_name VARCHAR2(20),
    last_name VARCHAR2(25));
```

```
table EMP_UNNAMED_INDEX created.
```

```
SELECT INDEX_NAME, TABLE_NAME
FROM   USER_INDEXES
WHERE  TABLE_NAME = 'EMP_UNNAMED_INDEX';
```

	INDEX_NAME	TABLE_NAME
1	SYS_C0017294	EMP_UNNAMED_INDEX

Observe that the Oracle Server gives a generic name to the index that is created for the PRIMARY KEY column.

You can also use an existing index for your PRIMARY KEY column—for example, when you are expecting a large data load and want to speed up the operation. You may want to disable the constraints while performing the load and then enable them, in which case having a unique index on the PRIMARY KEY will still cause the data to be verified during the load. Therefore, you can first create a nonunique index on the column designated as PRIMARY KEY, and then create the PRIMARY KEY column and specify that it should use the existing index. The following examples illustrate this process:

Step 1: Create the table:

```
CREATE TABLE NEW_EMP2
 (employee_id NUMBER(6),
 first_name  VARCHAR2(20),
 last_name   VARCHAR2(25)
 );
```

Step 2: Create the index:

```
CREATE INDEX emp_id_idx2 ON
  new_emp2(employee_id);
```

Step 3: Create the PRIMARY KEY:

```
ALTER TABLE new_emp2 ADD PRIMARY KEY (employee_id) USING INDEX
 emp_id_idx2;
```

Function-Based Indexes

- A function-based index is based on expressions.
- The index expression is built from table columns, constants, SQL functions, and user-defined functions.

```
CREATE INDEX upper_dept_name_idx
ON dept2(UPPER(department_name));
```

```
index UPPER_DEPT_NAME_IDX created.
```

```
SELECT *
FROM   dept2
WHERE  UPPER(department_name) = 'SALES';
```


Function-based indexes defined with the `UPPER(`*`column_name`*`)` or `LOWER(`*`column_name`*`)` keywords allow non-case-sensitive searches. For example, consider the following index:

```
CREATE INDEX upper_last_name_idx ON emp2 (UPPER(last_name));
```

This facilitates processing queries such as:

```
SELECT * FROM emp2 WHERE UPPER(last_name) = 'KING';
```

The Oracle Server uses the index only when that particular function is used in a query. For example, the following statement may use the index, but without the `WHERE` clause, the Oracle Server may perform a full table scan:

```
SELECT    *
FROM      employees
WHERE     UPPER (last_name) IS NOT NULL
ORDER BY UPPER (last_name);
```

Note: The `QUERY_REWRITE_ENABLED` initialization parameter must be set to `TRUE` for a function-based index to be used.

The Oracle Server treats indexes with columns marked `DESC` as function-based indexes. The columns marked `DESC` are sorted in descending order.

Removing an Index

- Remove an index from the data dictionary by using the `DROP INDEX` command:

```
DROP INDEX index;
```

- Remove the `UPPER_DEPT_NAME_IDX` index from the data dictionary:

```
DROP INDEX upper_dept_name_idx;
```

```
index UPPER_DEPT_NAME_IDX dropped.
```

- To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

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, then indexes, constraints, and triggers are automatically dropped, but views and sequences remain.

DROP TABLE … PURGE

```
DROP TABLE dept80 PURGE;
```

```
table DEPT80 dropped.
```


Oracle Database provides a feature for dropping tables. When you drop a table, the database does not immediately release the space associated with the table. Rather, the database renames the table and places it in a recycle bin, where it can later be recovered with the `FLASHBACK TABLE` statement if you find that you dropped the table in error. If you want to immediately release the space associated with the table at the time you issue the `DROP TABLE` statement, include the `PURGE` clause as shown in the statement in the slide.

Specify `PURGE` only if you want to drop the table and release the space associated with it in a single step. If you specify `PURGE`, the database does not place the table and its dependent objects into the recycle bin.

Using this clause is equivalent to first dropping the table and then purging it from the recycle bin. This clause saves you one step in the process. It also provides enhanced security if you want to prevent sensitive material from appearing in the recycle bin.

Lesson Agenda

- Using the `ALTER TABLE` statement to add, modify, and drop a column
- Managing constraints:
  - Adding and dropping a constraint
  - Deferring constraints
  - Enabling and disabling a constraint
- Creating indexes:
  - Using the `CREATE TABLE` statement
  - Creating function-based indexes
  - Removing an index
- Performing flashback operations
- Creating and using temporary tables
- Creating and using external tables

`FLASHBACK TABLE` Statement

- Enables you to recover tables to a specified point in time with a single statement
- Restores table data along with associated indexes and constraints
- Enables you to revert the table and its contents to a certain point in time or system change number (SCN)

Oracle Flashback Table enables you to recover tables to a specified point in time with a single statement. You can restore table data along with associated indexes and constraints while the database is online, undoing changes to only the specified tables.

The Flashback Table feature is similar to a self-service repair tool. For example, if a user accidentally deletes important rows from a table and then wants to recover the deleted rows, you can use the `FLASHBACK TABLE` statement to restore the table to the time before the deletion and see the missing rows in the table.

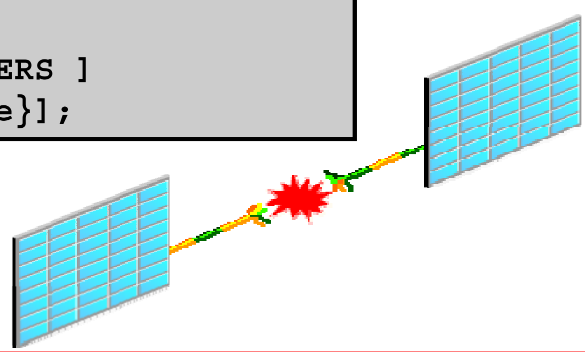
When using the `FLASHBACK TABLE` statement, you can revert the table and its contents to a certain time or to an SCN.

Note: The SCN is an integer value associated with each change to the database. It is a unique incremental number in the database. Every time you commit a transaction, a new SCN is recorded.

`FLASHBACK TABLE` Statement

- Repair tool for accidental table modifications
  - Restores a table to an earlier point in time
  - Offers ease of use, availability, and fast execution
  - Is performed in place
- Syntax:

```
FLASHBACK TABLE[schema.]table[,
[ schema.]table ]...
TO { TIMESTAMP | SCN } expr
[ { ENABLE | DISABLE } TRIGGERS ]
[BEFORE DROP {RENAME TO table}];
```



Self-Service Repair Facility

Oracle Database provides a SQL data definition language (DDL) command, `FLASHBACK TABLE`, to restore the state of a table to an earlier point in time in case it is inadvertently deleted or modified. The `FLASHBACK TABLE` command is a self-service repair tool to restore data in a table along with associated attributes such as indexes or views. This is done, while the database is online, by rolling back only the subsequent changes to the given table. Compared to traditional recovery mechanisms, this feature offers significant benefits such as ease of use, availability, and faster restoration. It also takes the burden off the DBA to find and restore application-specific properties. The flashback table feature does not address physical corruption caused because of a bad disk.

Syntax

You can invoke a `FLASHBACK TABLE` operation on one or more tables, even on tables in different schemas. You specify the point in time to which you want to revert by providing a valid time stamp. By default, database triggers are disabled during the flashback operation for all tables involved. You can override this default behavior by specifying the `ENABLE TRIGGERS` clause.

Note: For more information about recycle bin and flashback semantics, refer to *Oracle Database Administrator's Guide for 10g or 11g database*.

Using the `FLASHBACK TABLE` Statement

```
DROP TABLE emp2;
```

```
table EMP2 dropped.
```

```
SELECT original_name, operation, droptime FROM
recyclebin;
```

ORIGINAL_NAME	OPERATION	DROPTIME
EMP2	DROP	2009-05-20:18:00:39

…

```
FLASHBACK TABLE emp2 TO BEFORE DROP;
```

```
table EMP2 succeeded.
```


Syntax and Examples

The example restores the `EMP2` table to a state before a `DROP` statement.

The recycle bin is actually a data dictionary table containing information about dropped objects. Dropped tables and any associated objects—such as, indexes, constraints, nested tables, and so on—are not removed and still occupy space. They continue to count against user space quotas until specifically purged from the recycle bin, or until they must be purged by the database because of tablespace space constraints.

Each user can be thought of as an owner of a recycle bin because, unless a user has the `SYSDBA` privilege, the only objects that the user has access to in the recycle bin are those that the user owns. A user can view his or her objects in the recycle bin by using the following statement:

```
SELECT * FROM RECYCLEBIN;
```

When you drop a user, any objects belonging to that user are not placed in the recycle bin and any objects in the recycle bin are purged.

You can purge the recycle bin with the following statement:

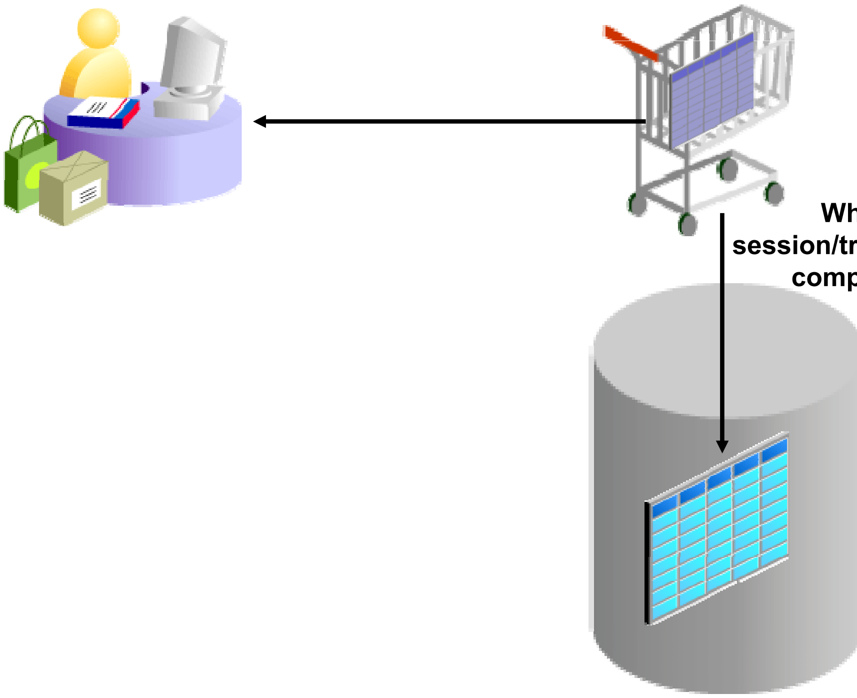
```
PURGE RECYCLEBIN;
```

Lesson Agenda

- Using the `ALTER TABLE` statement to add, modify, and drop a column
- Managing constraints:
  - Adding and dropping a constraint
  - Deferring constraints
  - Enabling and disabling a constraint
- Creating indexes:
  - Using the `CREATE TABLE` statement
  - Creating function-based indexes
  - Removing an index
- Performing flashback operations
- Creating and using temporary tables
- Creating and using external tables

ORACLE

Temporary Tables



A temporary table is a table that holds data that exists only for the duration of a transaction or session. Data in a temporary table is private to the session, which means that each session can see and modify only its own data.

Temporary tables are useful in applications where a result set must be buffered. For example, a shopping cart in an online application can be a temporary table. Each item is represented by a row in the temporary table. While you are shopping in an online store, you can keep on adding or removing items from your cart. During the session, this cart data is private. After you finalize your shopping and make the payments, the application moves the row for the chosen cart to a permanent table. At the end of the session, the data in the temporary table is automatically dropped.

Because temporary tables are statically defined, you can create indexes for them. Indexes created on temporary tables are also temporary. The data in the index has the same session or transaction scope as the data in the temporary table. You can also create a view or trigger on a temporary table.

Creating a Temporary Table

```
CREATE GLOBAL TEMPORARY TABLE cart(n NUMBER,d DATE)
ON COMMIT DELETE ROWS;
```

1

```
CREATE GLOBAL TEMPORARY TABLE today_sales
ON COMMIT PRESERVE ROWS AS
     SELECT * FROM orders
              WHERE order_date = SYSDATE;
```

2

To create a temporary table, you can use the following command:

```
CREATE GLOBAL TEMPORARY TABLE tablename
ON COMMIT [PRESERVE | DELETE] ROWS
```

By associating one of the following settings with the `ON COMMIT` clause, you can decide whether the data in the temporary table is transaction-specific (default) or session-specific.

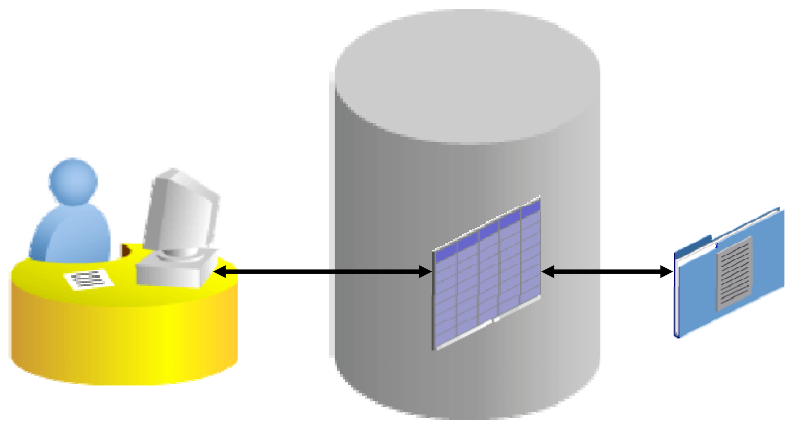
1. `DELETE ROWS`: As shown in example 1 in the slide, the `DELETE ROWS` setting creates a temporary table that is transaction-specific. A session becomes bound to the temporary table with a transaction's first insert into the table. The binding goes away at the end of the transaction. The database truncates the table (delete all rows) after each commit.
2. `PRESERVE ROWS`: As shown in example 2 in the slide, the `PRESERVE ROWS` setting creates a temporary table that is session-specific. Each sales representative session can store its own sales data for the day in the table. When a salesperson performs first insert on the `today_sales` table, his or her session gets bound to the `today_sales` table. This binding goes away at the end of the session or by issuing a `TRUNCATE` of the table in the session. The database truncates the table when you terminate the session.

When you create a temporary table in an Oracle database, you create a static table definition. Like permanent tables, temporary tables are defined in the data dictionary. However, temporary tables and their indexes do not automatically allocate a segment when created. Instead, temporary segments are allocated when data is first inserted. Until data is loaded in a session, the table appears empty.

Lesson Agenda

- Using the `ALTER TABLE` statement to add, modify, and drop a column
- Managing constraints:
  - Adding and dropping a constraint
  - Deferring constraints
  - Enabling and disabling a constraint
- Creating indexes:
  - Using the `CREATE TABLE` statement
  - Creating function-based indexes
  - Removing an index
- Performing flashback operations
- Creating and using temporary tables
- **Creating and using external tables**

External Tables



An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database. This external table definition can be thought of as a view that is used for running any SQL query against external data without requiring that the external data first be loaded into the database. The external table data can be queried and joined directly and in parallel without requiring that the external data first be loaded in the database. You can use SQL, PL/SQL, and Java to query the data in an external table.

The main difference between external tables and regular tables is that externally organized tables are read-only. No data manipulation language (DML) operations are possible, and no indexes can be created on them. However, you can create an external table, and thus unload data, by using the `CREATE TABLE AS SELECT` command.

The Oracle Server provides two major access drivers for external tables. One, the loader access driver (or `ORACLE_LOADER`) is used for reading data from external files whose format can be interpreted by the SQL*Loader utility. Note that not all SQL*Loader functionality is supported with external tables. The `ORACLE_DATAPUMP` access driver can be used to both import and export data by using a platform-independent format. The `ORACLE_DATAPUMP` access driver writes rows from a `SELECT` statement to be loaded into an external table as part of a `CREATE TABLE ...ORGANIZATION EXTERNAL...AS SELECT` statement. You can then use `SELECT` to read data out of that data file. You can also create an external table definition on another system and use that data file. This allows data to be moved between Oracle databases.

Creating a Directory for the External Table

Create a DIRECTORY object that corresponds to the directory on the file system where the external data source resides.

```
CREATE OR REPLACE DIRECTORY emp_dir
AS '/…/emp_dir';

GRANT READ ON DIRECTORY emp_dir TO ora_21;
```


Creating an External Table: Example

Use the CREATE DIRECTORY statement to create a directory object. A directory object specifies an alias for a directory on the server's file system where an external data source resides. You can use directory names when referring to an external data source, rather than hard code the operating system path name, for greater file management flexibility.

You must have CREATE ANY DIRECTORY system privileges to create directories. When you create a directory, you are automatically granted the READ and WRITE object privileges and can grant READ and WRITE privileges to other users and roles. The DBA can also grant these privileges to other users and roles.

A user needs READ privileges for all directories used in external tables to be accessed and WRITE privileges for the log, bad, and discard file locations being used.

In addition, a WRITE privilege is necessary when the external table framework is being used to unload data.

Oracle also provides the ORACLE_DATAPUMP type, with which you can unload data (that is, read data from a table in the database and insert it into an external table) and then reload it into an Oracle database. This is a one-time operation that can be done when the table is created. After the creation and initial population is done, you cannot update, insert, or delete any rows.

Syntax

```
CREATE [OR REPLACE] DIRECTORY AS 'path_name';
```

In the syntax:

`OR REPLACE`	Specify `OR REPLACE` to re-create the directory database object if it already exists. You can use this clause to change the definition of an existing directory without dropping, re-creating, and regranting database object privileges previously granted on the directory. Users who were previously granted privileges on a redefined directory can continue to access the directory without requiring that the privileges be regranted.
`directory`	Specify the name of the directory object to be created. The maximum length of the directory name is 30 bytes. You cannot qualify a directory object with a schema name.
`'path_name'`	Specify the full path name of the operating system directory to be accessed. The path name is case-sensitive.

Creating an External Table

```
CREATE TABLE <table_name>
   ( <col_name> <datatype>, … )
ORGANIZATION EXTERNAL
    (TYPE <access_driver_type>
     DEFAULT DIRECTORY <directory_name>
     ACCESS PARAMETERS
      (… ) )
      LOCATION ('<location_specifier>')
REJECT LIMIT [0 | <number> | UNLIMITED];
```


You create external tables by using the `ORGANIZATION EXTERNAL` clause of the `CREATE TABLE` statement. You are not, in fact, creating a table. Rather, you are creating metadata in the data dictionary that you can use to access external data. You use the `ORGANIZATION` clause to specify the order in which the data rows of the table are stored. By specifying `EXTERNAL` in the `ORGANIZATION` clause, you indicate that the table is a read-only table located outside the database. Note that the external files must already exist outside the database.

`TYPE` *`<access_driver_type>`* indicates the access driver of the external table. The access driver is the application programming interface (API) that interprets the external data for the database. If you do not specify `TYPE`, Oracle uses the default access driver, `ORACLE_LOADER`. The other option is `ORACLE_DATAPUMP`.

You use the `DEFAULT DIRECTORY` clause to specify one or more Oracle database directory objects that correspond to directories on the file system where the external data sources may reside.

The optional `ACCESS PARAMETERS` clause enables you to assign values to the parameters of the specific access driver for this external table.

Use the `LOCATION` clause to specify one external locator for each external data source. Usually, `<location_specifier>` is a file, but it need not be.

The `REJECT LIMIT` clause enables you to specify how many conversion errors can occur during a query of the external data before an Oracle error is returned and the query is aborted. The default value is `0`.

The syntax for using the `ORACLE_DATAPUMP` access driver is as follows:

```
CREATE TABLE extract_emps
ORGANIZATION EXTERNAL (TYPE ORACLE_DATAPUMP
                       DEFAULT DIRECTORY …
                       ACCESS PARAMETERS (… )
                       LOCATION (…)
                       PARALLEL 4
                       REJECT LIMIT UNLIMITED
AS
SELECT * FROM …;
```

Creating an External Table by Using `ORACLE_LOADER`

```
CREATE TABLE oldemp (
  fname char(25), lname CHAR(25))
  ORGANIZATION EXTERNAL
  (TYPE ORACLE_LOADER
  DEFAULT DIRECTORY emp_dir
  ACCESS PARAMETERS
  (RECORDS DELIMITED BY NEWLINE
   NOBADFILE
   NOLOGFILE
  FIELDS TERMINATED BY ','
  (fname POSITION ( 1:20) CHAR,
   lname POSITION (22:41) CHAR))
  LOCATION ('emp.dat'))
  PARALLEL 5
  REJECT LIMIT 200;
```

```
table OLDEMP created.
```


Creating an External Table by Using the `ORACLE_LOADER` Access Driver: Example

Assume that there is a flat file that has records in the following format:

```
10,jones,11-Dec-1934
20,smith,12-Jun-1972
```

Records are delimited by new lines, and the fields are all terminated by a comma (`,`). The name of the file is `/emp_dir/emp.dat`.

To convert this file as the data source for an external table, whose metadata will reside in the database, you must perform the following steps:

1. Create a directory object, `emp_dir`, as follows:
   `CREATE DIRECTORY emp_dir AS '/emp_dir' ;`
2. Run the `CREATE TABLE` command shown in the slide.

The example in the slide illustrates the table specification to create an external table for the file:

```
/emp_dir/emp.dat
```

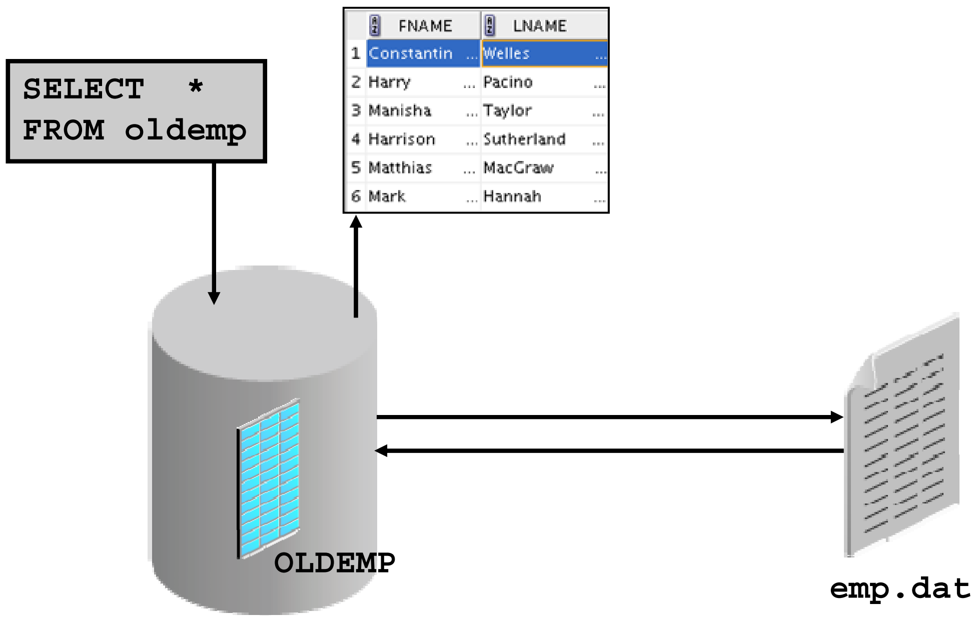
In the example, the TYPE specification is given only to illustrate its use. ORACLE_LOADER is the default access driver if not specified. The ACCESS PARAMETERS option provides values to parameters of the specific access driver, which are interpreted by the access driver, not by the Oracle Server.

The PARALLEL clause enables five parallel execution servers to simultaneously scan the external data sources (files) when executing the INSERT INTO TABLE statement. For example, if PARALLEL=5 were specified, more than one parallel execution server can be working on a data source. Because external tables can be very large, for performance reasons, it is advisable to specify the PARALLEL clause, or a parallel hint for the query.

The REJECT LIMIT clause specifies that if more than 200 conversion errors occur during a query of the external data, the query must be aborted and an error must be returned. These conversion errors can arise when the access driver tries to transform the data in the data file to match the external table definition.

After the CREATE TABLE command executes successfully, the OLDEMP external table can be described and queried in the same way as a relational table.

Querying External Tables



An external table does not describe any data that is stored in the database. It does not describe how data is stored in the external source. Instead, it describes how the external table layer must present the data to the server. It is the responsibility of the access driver and the external table layer to do the necessary transformations required on the data in the data file so that it matches the external table definition.

When the database server accesses data in an external source, it calls the appropriate access driver to get the data from an external source in a form that the database server expects.

It is important to remember that the description of the data in the data source is separate from the definition of the external table. The source file can contain more or fewer fields than there are columns in the table. Also, the data types for fields in the data source can be different from the columns in the table. The access driver takes care of ensuring that the data from the data source is processed so that it matches the definition of the external table.

Creating an External Table by Using `ORACLE_DATAPUMP`: Example

```
CREATE TABLE emp_ext
   (employee_id, first_name, last_name)
    ORGANIZATION EXTERNAL
     (
      TYPE ORACLE_DATAPUMP
      DEFAULT DIRECTORY emp_dir
      LOCATION
       ('emp1.exp','emp2.exp')
     )
    PARALLEL
AS
SELECT employee_id, first_name, last_name
FROM   employees;
```


You can perform the unload and reload operations with external tables by using the `ORACLE_DATAPUMP` access driver.

Note: In the context of external tables, loading data refers to the act of data being read from an external table and loaded into a table in the database. Unloading data refers to the act of reading data from a table and inserting it into an external table.

The example in the slide illustrates the table specification to create an external table by using the `ORACLE_DATAPUMP` access driver. Data is then populated into the two files: `emp1.exp` and `emp2.exp`.

To populate data read from the `EMPLOYEES` table into an external table, you must perform the following steps:

1. Create a directory object, `emp_dir`, as follows:
   ```
   CREATE DIRECTORY emp_dir AS '/emp_dir' ;
   ```
2. Run the `CREATE TABLE` command shown in the slide.

Note: The `emp_dir` directory is the same as created in the previous example of using `ORACLE_LOADER`.

You can query the external table by executing the following code:

```
SELECT * FROM emp_ext;
```

Quiz

A `FOREIGN KEY` constraint enforces the following action: When the data in the parent key is deleted, all the rows in the child table that depend on the deleted parent key values are also deleted.

a. True

b. False

Answer: b

Quiz

In all the cases, when you execute a `DROP TABLE` command, the database renames the table and places it in a recycle bin, from where it can later be recovered by using the `FLASHBACK TABLE` statement.

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned how to:

- Add constraints
- Create indexes
- Create indexes by using the `CREATE TABLE` statement
- Create function-based indexes
- Drop columns and set columns as `UNUSED`
- Perform `FLASHBACK` operations
- Create and use external tables

In this lesson, you learned how to perform the following tasks for schema object management:

- Alter tables to add or modify columns or constraints.
- Create indexes and function-based indexes by using the `CREATE INDEX` statement.
- Drop unused columns.
- Use `FLASHBACK` mechanics to restore tables.
- Use the `ORGANIZATION EXTERNAL` clause of the `CREATE TABLE` statement to create an external table. An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database.
- Use external tables to query data without first loading it into the database.
- Name your `PRIMARY KEY` column indexes when you create the table with the `CREATE TABLE` statement.

Practice 3: Overview

This practice covers the following topics:

- Altering tables
- Adding columns
- Dropping columns
- Creating indexes
- Creating external tables

In this practice, you use the `ALTER TABLE` command to modify columns and add constraints. You use the `CREATE INDEX` command to create indexes when creating a table, along with the `CREATE TABLE` command. You create external tables.

4

Managing Objects with Data Dictionary Views

Objectives

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

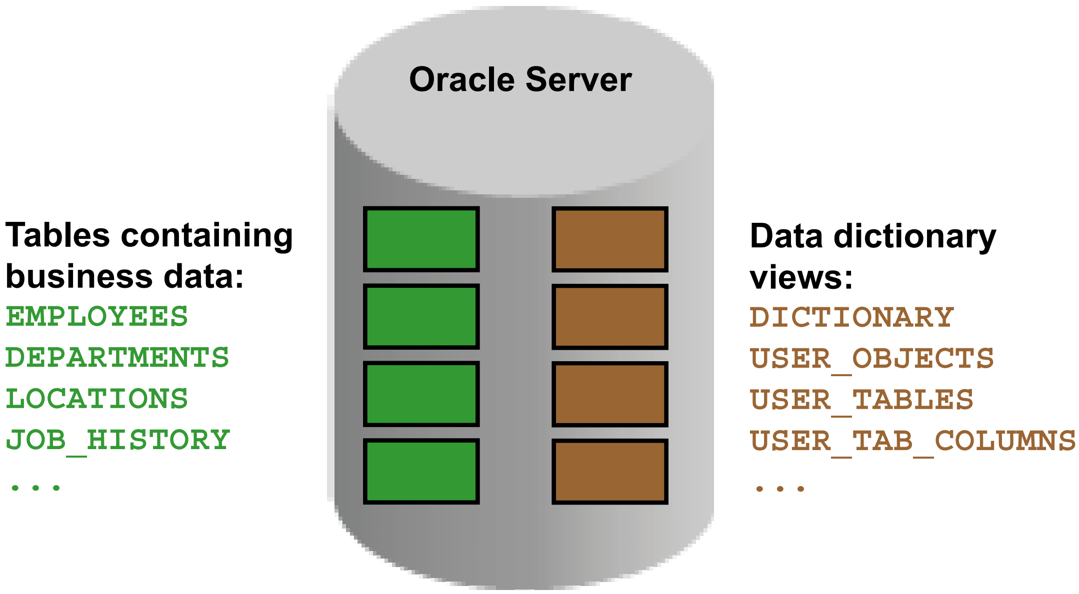
- Use the data dictionary views to research data on your objects
- Query various data dictionary views

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.

Lesson Agenda

- Introduction to data dictionary
- Querying the dictionary views for the following:
  - Table information
  - Column information
  - Constraint information
- Querying the dictionary views for the following:
  - View information
  - Sequence information
  - Synonym information
  - Index information
- Adding a comment to a table and querying the dictionary views for comment information

Data Dictionary



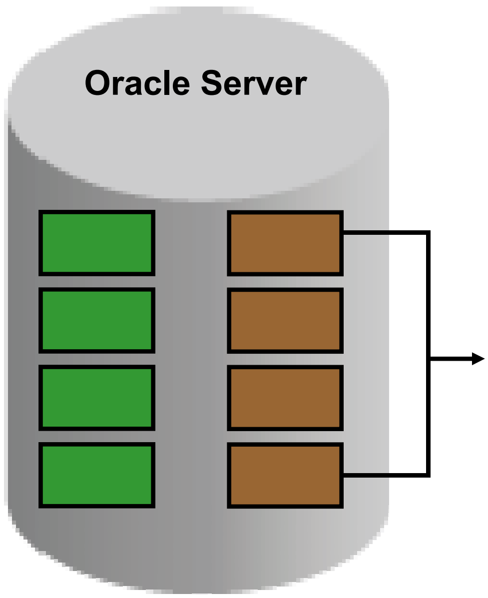
User tables are tables created by the user and contain business data, such as `EMPLOYEES`. There is another collection of tables and views in the Oracle database known as the *data dictionary*. This collection is created and maintained by the Oracle Server and contains information about the database. 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 also 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**

Underlying base tables store information about the associated database. Only the Oracle Server should write to and read from 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

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 you created. `ALL_OBJECTS` contains information about all objects to which you have access. `DBA_OBJECTS` contains information about 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
1	USER_OBJECTS	Objects owned by the user

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 about 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 in uppercase.

USER_OBJECTS and ALL_OBJECTS Views

USER_OBJECTS:

- Query USER_OBJECTS to see all the objects that you own.
- Using USER_OBJECTS, you can 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 the objects to which you have access.

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:** Time stamp for the creation of the object
- **LAST_DDL_TIME:** Time stamp for the last modification of the object resulting from a data definition language (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
1	LOC_COUNTRY_IX	INDEX	19-MAY-09	VALID

...

53	EMPLOYEES2	TABLE	22-MAY-09	VALID
54	SECURE_EMPLOYEES	TRIGGER	19-MAY-09	VALID
55	UPDATE_JOB_HISTORY	TRIGGER	19-MAY-09	VALID
56	EMP_DETAILS_VIEW	VIEW	19-MAY-09	VALID

...

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. Although 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.

Note: CAT is a synonym for USER_CATALOG—a view that lists tables, views, synonyms and sequences owned by the user.

Lesson Agenda

- Introduction to data dictionary
- Querying the dictionary views for the following:
  - Table information
  - Column information
  - Constraint information
- Querying the dictionary views for the following:
  - View information
  - Sequence information
  - Synonym information
  - Index information
- Adding a comment to a table and querying the dictionary views for comment information

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
1	REGIONS
2	LOCATIONS
3	DEPARTMENTS
4	JOBS
5	EMPLOYEES
6	JOB_HISTORY

...

You can use the USER_TABLES view to obtain the names of all your tables. The USER_TABLES view contains information about your tables. In addition to providing the table name, it contains detailed information about 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)
```

...

You can query the `USER_TAB_COLUMNS` view to find detailed information about the columns in your tables. Although the `USER_TABLES` view provides information about 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
1	EMPLOYEE_ID	NUMBER	22	6
2	FIRST_NAME	VARCHAR2	20	(null)
3	LAST_NAME	VARCHAR2	25	(null)
4	EMAIL	VARCHAR2	25	(null)
5	PHONE_NUMBER	VARCHAR2	20	(null)
6	HIRE_DATE	DATE	7	(null)
7	JOB_ID	VARCHAR2	10	(null)
8	SALARY	NUMBER	22	8
9	COMMISSION_PCT	NUMBER	22	2
10	MANAGER_ID	NUMBER	22	6
11	DEPARTMENT_ID	NUMBER	22	4

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 `DESCRIBE` command.

To view information about columns set as unused, you use the `USER_UNUSED_COL_TABS` dictionary view.

Note: Names of the objects in Data Dictionary are in uppercase.

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)
```

...

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*.

USER_CONSTRAINTS: Example

```
SELECT constraint_name, constraint_type,
       search_condition, r_constraint_name,
       delete_rule, status
FROM   user_constraints
WHERE  table_name = 'EMPLOYEES';
```

	CONSTRAINT_NAME	C...	SEARCH_CONDITION	R_CONSTR...	DELET...	STATUS
1	EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL	(null)	(null)	ENABLED
2	EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL	(null)	(null)	ENABLED
3	EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL	(null)	(null)	ENABLED
4	EMP_JOB_NN	C	"JOB_ID" IS NOT NULL	(null)	(null)	ENABLED
5	EMP_SALARY_MIN	C	salary > 0	(null)	(null)	ENABLED
6	EMP_EMAIL_UK	U	(null)	(null)	(null)	ENABLED
7	EMP_EMP_ID_PK	P	(null)	(null)	(null)	ENABLED
8	EMP_DEPT_FK	R	(null)	DEPT_ID_PK	NO ACTION	ENABLED
9	EMP_JOB_FK	R	(null)	JOB_ID_PK	NO ACTION	ENABLED
10	EMP_MANAGER_FK	R	(null)	EMP_EMP_ID_PK	NO ACTION	ENABLED

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, or NOT NULL)
- 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.
- **SET NULL**: If the parent record is deleted, change the respective child record to null.
- **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.

Querying USER_CONS_COLUMNS

```
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
1	EMP_LAST_NAME_NN	LAST_NAME
2	EMP_EMAIL_NN	EMAIL
3	EMP_HIRE_DATE_NN	HIRE_DATE
4	EMP_JOB_NN	JOB_ID
5	EMP_SALARY_MIN	SALARY
6	EMP_EMAIL_UK	EMAIL

...

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 USER_CONSTRAINTS and USER_CONS_COLUMNS to create customized output from both tables.

Lesson Agenda

- Introduction to data dictionary
- Querying the dictionary views for the following:
  - Table information
  - Column information
  - Constraint information
- Querying the dictionary views for the following:
  - View information
  - Sequence information
  - Synonym information
  - Index information
- Adding a comment to a table and querying the dictionary views for comment information

View Information

1

```
DESCRIBE user_views
```

```
Name                           Null     Type
------------------------------ -------- -------------
VIEW_NAME                      NOT NULL VARCHAR2(30)
TEXT_LENGTH                             NUMBER
TEXT                                    LONG()
```

2

```
SELECT view_name FROM user_views;
```

	VIEW_NAME
1	EMP_DETAILS_VIEW

3

```
SELECT text FROM user_views
WHERE view_name = 'EMP_DETAILS_VIEW';
```

	TEXT
1	SELECT e.employee_id, e.job_id, e.manager_id, e.department_id, d.location_id, l.co

...

AND c.region_id = r.region_id AND j.job_id = e.job_idWITH READ ONLY

ORACLE

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 in SQL*Plus, use the SET LONG command:

```
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 by 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. That is, 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
```


The USER_SEQUENCES view describes all sequences that you own. 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 the sequence is incremented
- **CYCLE_FLAG:** Whether sequence wraps around on reaching the limit
- **ORDER_FLAG:** Whether sequence numbers are 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.

Confirming Sequences

- 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
1	DEPARTMENTS_SEQ	1	9990	10	280
2	EMPLOYEES_SEQ	1	99999999999999...	1	207
3	LOCATIONS_SEQ	1	9900	100	3300

- The `LAST_NUMBER` column displays the next available sequence number if `NOCACHE` is specified.

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.

Index Information

- `USER_INDEXES` provides information about your indexes.
- `USER_IND_COLUMNS` describes columns comprising your indexes and columns of indexes on your tables.

```
DESCRIBE user_indexes
```

```
Name                             Null     Type
-------------------------------- -------- -------------
INDEX_NAME                       NOT NULL VARCHAR2(30)
INDEX_TYPE                                VARCHAR2(27)
TABLE_OWNER                      NOT NULL VARCHAR2(30)
TABLE_NAME                       NOT NULL VARCHAR2(30)
TABLE_TYPE                                VARCHAR2(11)
UNIQUENESS                                VARCHAR2(9)
```

...

You query the `USER_INDEXES` view to find out the names of your indexes, the table name on which the index is created, and whether the index is unique.

Note: For a complete listing and description of the columns in the `USER_INDEXES` view, see "`USER_INDEXES`" in the *Oracle Database Reference for 10g or 11g database.*

USER_INDEXES: Examples

a

```
SELECT index_name, table_name, uniqueness
FROM   user_indexes
WHERE  table_name = 'EMPLOYEES';
```

	INDEX_NAME	TABLE_NAME	UNIQUENESS
1	EMP_EMAIL_UK	EMPLOYEES	UNIQUE
2	EMP_EMP_ID_PK	EMPLOYEES	UNIQUE
3	EMP_DEPARTMENT_IX	EMPLOYEES	NONUNIQUE
4	EMP_JOB_IX	EMPLOYEES	NONUNIQUE
5	EMP_MANAGER_IX	EMPLOYEES	NONUNIQUE
6	EMP_NAME_IX	EMPLOYEES	NONUNIQUE

b

```
SELECT index_name, table_name
FROM   user_indexes
WHERE  table_name = 'EMP_LIB';
```

	INDEX_NAME	TABLE_NAME
1	SYS_C0011777	EMP_LIB

In slide example **a**, the `USER_INDEXES` view is queried to find the name of the index, name of the table on which the index is created, and whether the index is unique.

In slide example **b**, observe that the Oracle Server gives a generic name to the index that is created for the `PRIMARY KEY` column. The `EMP_LIB` table is created by using the following code:

```
CREATE TABLE emp_lib
    (book_id NUMBER(6) PRIMARY KEY,
     title VARCHAR2(25),
     category VARCHAR2(20));
```

```
table EMP_LIB created.
```

Querying USER_IND_COLUMNS

```
DESCRIBE user_ind_columns
```

```
Name                           Null     Type
------------------------------ -------- --------------
INDEX_NAME                              VARCHAR2(30)
TABLE_NAME                              VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
COLUMN_POSITION                         NUMBER
COLUMN_LENGTH                           NUMBER
CHAR_LENGTH                             NUMBER
DESCEND                                 VARCHAR2(4)
```

```
SELECT  index_name, column_name,table_name
FROM    user_ind_columns
WHERE   index_name = 'LNAME_IDX';
```

	INDEX_NAME	COLUMN_NAME	TABLE_NAME
1	LNAME_IDX	LAST_NAME	EMP_TEST

The USER_IND_COLUMNS dictionary view provides information such as the name of the index, name of the indexed table, name of a column within the index, and the column's position within the index.

For the slide example, the emp_test table and LNAME_IDX index are created by using the following code:

```
CREATE TABLE emp_test AS SELECT * FROM employees;
CREATE INDEX lname_idx ON emp_test(last_name);
```

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
1	TEAM2	ORA22	DEPARTMENTS	(null)

The `USER_SYNONYMS` dictionary view describes private synonyms (synonyms that you own).

You can query this view to find your synonyms. You can query `ALL_SYNONYMS` to find out the name of all 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)

Lesson Agenda

- Introduction to data dictionary
- Querying the dictionary views for the following:
  - Table information
  - Column information
  - Constraint information
- Querying the dictionary views for the following:
  - View information
  - Sequence information
  - Synonym information
  - Index information
- Adding a comment to a table and querying the dictionary views for comment information

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 ON COLUMN employees.first_name
IS 'First name of the employee';
```

- Comments can be viewed through the data dictionary views:
  - `ALL_COL_COMMENTS`
  - `USER_COL_COMMENTS`
  - `ALL_TAB_COMMENTS`
  - `USER_TAB_COMMENTS`

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 '';
```

Quiz

The dictionary views that are based on the dictionary tables contain information such as:

a. Definitions of all the schema objects in the database
b. Default values for the columns
c. Integrity constraint information
d. Privileges and roles that each user has been granted
e. All of the above

Answer: e

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_INDEXES`
- `USER_SYNONYMS`

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 4: 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
- Querying the dictionary views for index information
- Adding a comment to a table and querying the dictionary views for comment information

In this practice, you query the dictionary views to find information about objects in your schema.

5

Manipulating Large Data Sets

Objectives

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

- Manipulate data by using subqueries
- Specify explicit default values in the `INSERT` and `UPDATE` statements
- Describe the features of multitable `INSERT`s
- Use the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merge rows in a table
- Track the changes made to data over a period of time

In this lesson, you learn how to manipulate data in the Oracle database by using subqueries. You learn how to use the `DEFAULT` keyword in `INSERT` and `UPDATE` statements to identify a default column value. You also learn about multitable `INSERT` statements, the `MERGE` statement, and tracking changes in the database.

Lesson Agenda

- Manipulating data by using subqueries
- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Tracking the changes to data over a period of time

Using Subqueries to Manipulate Data

You can use subqueries in data manipulation language (DML) statements to:

- Retrieve data by using an inline view
- Copy data from one table to another
- Update data in one table based on the values of another table
- Delete rows from one table based on rows in another table

Subqueries can be used to retrieve data from a table that you can use as input to an `INSERT` into a different table. In this way, you can easily copy large volumes of data from one table to another with one single `SELECT` statement. Similarly, you can use subqueries to do mass updates and deletes by using them in the `WHERE` clause of the `UPDATE` and `DELETE` statements. You can also use subqueries in the `FROM` clause of a `SELECT` statement. This is called an inline view.

Note: You learned how to update and delete rows based on another table in the course titled *Oracle Database: SQL Fundamentals I*.

Retrieving Data by Using a Subquery as Source

```
SELECT department_name, city
FROM   departments
NATURAL JOIN (SELECT l.location_id, l.city, l.country_id
              FROM   loc l
              JOIN   countries c
              ON(l.country_id = c.country_id)
              JOIN regions USING(region_id)
              WHERE region_name = 'Europe');
```

	DEPARTMENT_NAME	CITY
1	Human Resources	London
2	Sales	Oxford
3	Public Relations	Munich

You can use a subquery in the `FROM` clause of a `SELECT` statement, which is very similar to how views are used. A subquery in the `FROM` clause of a `SELECT` statement is also called an *inline* view. A subquery in the `FROM` clause of a `SELECT` statement defines a data source for that particular `SELECT` statement, and only that `SELECT` statement. As with a database view, the `SELECT` statement in the subquery can be as simple or as complex as you like.

When a database view is created, the associated `SELECT` statement is stored in the data dictionary. In situations where you do not have the necessary privileges to create database views, or when you would like to test the suitability of a `SELECT` statement to become a view, you can use an inline view.

With inline views, you can have all the code needed to support the query in one place. This means that you can avoid the complexity of creating a separate database view. The example in the slide shows how to use an inline view to display the department name and the city in Europe. The subquery in the `FROM` clause fetches the location ID, city name, and the country by joining three different tables. The output of the inner query is considered as a table for the outer query. The inner query is similar to that of a database view but does not have any physical name.

For the example in the slide, the `loc` table is created by running the following statement:

```
CREATE TABLE loc AS SELECT * FROM locations;
```

You can display the same output as in the example in the slide by performing the following two steps:

1. Create a database view:
```
CREATE OR REPLACE VIEW european_cities
AS
SELECT l.location_id, l.city, l.country_id
FROM   loc l
JOIN   countries c
ON(l.country_id = c.country_id)
JOIN regions USING(region_id)
WHERE region_name = 'Europe';
```
2. Join the `EUROPEAN_CITIES` view with the `DEPARTMENTS` table:
```
SELECT department_name, city
FROM   departments
NATURAL JOIN european_cities;
```

Note: You learned how to create database views in the course titled *Oracle Database: SQL Fundamentals I*.

Inserting by Using a Subquery as a Target

```
INSERT INTO (SELECT l.location_id, l.city, l.country_id
             FROM   loc l
             JOIN   countries c
             ON(l.country_id = c.country_id)
             JOIN regions USING(region_id)
             WHERE region_name = 'Europe')
VALUES (3300, 'Cardiff', 'UK');
```

```
1 rows inserted.
```


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 in order for the `INSERT` statement to work successfully. For example, you cannot put in a duplicate location ID or leave out a value for a mandatory `NOT NULL` column.

This use of subqueries helps you avoid having to create a view just for performing an `INSERT`.

The example in the slide uses a subquery in the place of `LOC` to create a record for a new European city.

Note: You can also perform the `INSERT` operation on the `EUROPEAN_CITIES` view by using the following code:

```
INSERT INTO european_cities
VALUES (3300,'Cardiff','UK');
```

Inserting by Using a Subquery as a Target

Verify the results.

```
SELECT location_id, city, country_id
FROM   loc
```

	LOCATION_ID	CITY	COUNTRY_ID
20	2900	Geneva	CH
21	3000	Bern	CH
22	3100	Utrecht	NL
23	3200	Mexico City	MX
24	3300	Cardiff	UK

The example in the slide shows that the insert via the inline view created a new record in the base table LOC.

The following example shows the results of the subquery that was used to identify the table for the INSERT statement.

```
SELECT l.location_id, l.city, l.country_id
FROM   loc l
JOIN   countries c
ON(l.country_id = c.country_id)
JOIN regions USING(region_id)
WHERE region_name = 'Europe'
```

	LOCATION_ID	CITY	COUNTRY_ID
6	2700	Munich	DE
7	2900	Geneva	CH
8	3000	Bern	CH
9	3100	Utrecht	NL
10	3300	Cardiff	UK

Using the `WITH CHECK OPTION` Keyword on DML Statements

The `WITH CHECK OPTION` keyword prohibits you from changing rows that are not in the subquery.

```
INSERT INTO ( SELECT location_id, city, country_id
              FROM   loc
              WHERE  country_id IN
              (SELECT country_id
               FROM countries
               NATURAL JOIN regions
               WHERE region_name = 'Europe')
              WITH CHECK OPTION )
VALUES (3600, 'Washington', 'US');
```

```
Error starting at line 1 in command:
INSERT INTO ( SELECT location_id, city, country_id
              FROM   loc
              WHERE  country_id IN
              (SELECT country_id
               FROM countries
               NATURAL JOIN regions
               WHERE region_name = 'Europe')
               WITH CHECK OPTION )
VALUES (3600, 'Washington', 'US')
Error report:
SQL Error: ORA-01402: view WITH CHECK OPTION where-clause violation
01402. 00000 -  "view WITH CHECK OPTION where-clause violation"
*Cause:
*Action:
```


Specify the `WITH CHECK OPTION` keyword to indicate that if the subquery is used in place of a table in an `INSERT`, `UPDATE`, or `DELETE` statement, no changes that will produce rows that are not included in the subquery are permitted to that table.

The example in the slide shows how to use an inline view with `WITH CHECK OPTION`. The `INSERT` statement prevents the creation of records in the `LOC` table for a city that is not in Europe.

The following example executes successfully because of the changes in the `VALUES` list.

```
INSERT INTO (SELECT location_id, city, country_id
             FROM   loc
             WHERE  country_id IN
             (SELECT country_id
              FROM countries
              NATURAL JOIN regions
              WHERE region_name = 'Europe')
              WITH CHECK OPTION)
VALUES (3500, 'Berlin', 'DE');
```

The use of an inline view with the `WITH CHECK OPTION` provides an easy method to prevent changes to the table.

To prevent the creation of a non-European city, you can also use a database view by performing the following steps:

1. Create a database view:

```
CREATE OR REPLACE VIEW european_cities
AS
SELECT location_id, city, country_id
FROM   locations
WHERE  country_id in
  (SELECT country_id
   FROM countries
   NATURAL JOIN regions
   WHERE region_name = 'Europe')
WITH CHECK OPTION;
```

2. Verify the results by inserting data:

```
INSERT INTO european_cities
VALUES (3400,'New York','US');
```

The second step produces the same error as shown in the slide.

Lesson Agenda

- Manipulating data by using subqueries
- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Tracking the changes to data over a period of time

Explicit Default Feature: Overview

- Use the `DEFAULT` keyword as a column value where the default column value is desired.
- This allows the user to control where and when the default value should be applied to data.
- Explicit defaults can be used in `INSERT` and `UPDATE` statements.

The `DEFAULT` keyword can be used in `INSERT` and `UPDATE` statements to identify a default column value. If no default value exists, a null value is used.

The `DEFAULT` option saves you from having to hard-code the default value in your programs or query the dictionary to find it, as was done before this feature was introduced. Hard-coding the default is a problem if the default changes, because the code consequently needs changing. Accessing the dictionary is not usually done in an application; therefore, this is a very important feature.

Using Explicit Default Values

- `DEFAULT` with `INSERT`:

```
INSERT INTO deptm3
  (department_id, department_name, manager_id)
VALUES (300, 'Engineering', DEFAULT);
```

- `DEFAULT` with `UPDATE`:

```
UPDATE deptm3
SET manager_id = DEFAULT
WHERE department_id = 10;
```


Specify `DEFAULT` to set the column to the value previously specified as the default value for the column. If no default value for the corresponding column has been specified, the Oracle server sets the column to null.

In the first example in the slide, the `INSERT` statement uses a default value for the `MANAGER_ID` column. If there is no default value defined for the column, a null value is inserted instead.

The second example uses the `UPDATE` statement to set the `MANAGER_ID` column to a default value for department 10. If no default value is defined for the column, it changes the value to null.

Note: When creating a table, you can specify a default value for a column. This is discussed in *SQL Fundamentals I*.

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%';
```

```
33 rows inserted.
```

- Do not use the VALUES clause.
- Match the number of columns in the INSERT clause with that in the subquery.

ORACLE

You can use the INSERT statement to add rows to a table where the values are derived from existing tables. In place of the VALUES clause, you use a subquery.

Syntax

```
INSERT INTO table [ column (, column) ] subquery;
```

In the syntax:

table	Is the table name
column	Is the name of the column in the table to populate
subquery	Is the subquery that returns rows into the table

The number of columns and their data types in the column list of the INSERT clause must match the number of values and their data types in the subquery. To create a copy of the rows of a table, use SELECT * in the subquery.

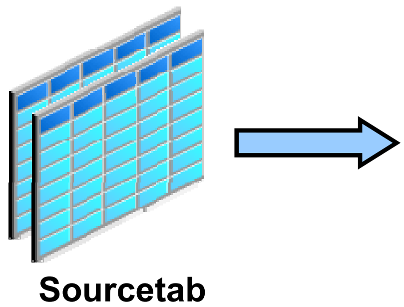
```
INSERT INTO EMPL3
   SELECT *
   FROM   employees;
```

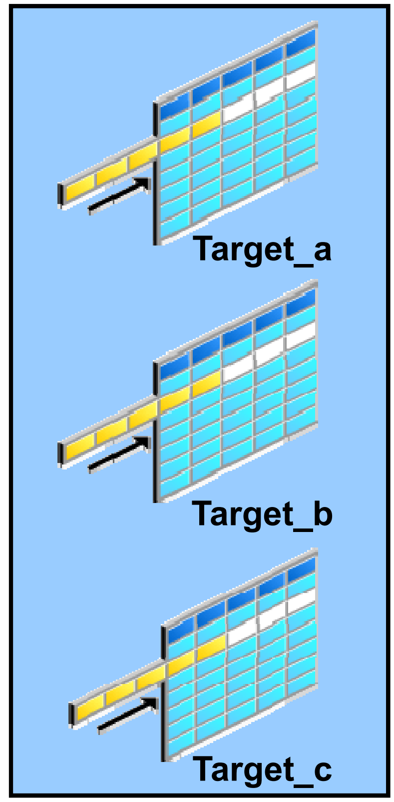
Note: You use the LOG ERRORS clause in your DML statement to enable the DML operation to complete regardless of errors. Oracle writes the details of the error message to an error-logging table that you have created. For more information, see the *Oracle Database SQL Reference for 10g or 11g database*.

Lesson Agenda

- Manipulating data by using subqueries
- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERT`s:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Tracking the changes to data over a period of time

Multitable INSERT Statements: Overview






```
INSERT  ALL
   INTO target_a VALUES(…,…,…)
   INTO target_b VALUES(…,…,…)
   INTO target_c VALUES(…,…,…)
   SELECT …
   FROM  sourcetab
   WHERE …;
```

Subquery

In a multitable `INSERT` statement, you insert computed rows derived from the rows returned from the evaluation of a subquery into one or more tables.

Multitable `INSERT` statements are useful in a data warehouse scenario. You need to load your data warehouse regularly so that it can serve its purpose of facilitating business analysis. To do this, data from one or more operational systems must be extracted and copied into the warehouse. The process of extracting data from the source system and bringing it into the data warehouse is commonly called ETL, which stands for extraction, transformation, and loading.

During extraction, the desired data must be identified and extracted from many different sources, such as database systems and applications. After extraction, the data must be physically transported to the target system or an intermediate system for further processing. Depending on the chosen means of transportation, some transformations can be done during this process. For example, a SQL statement that directly accesses a remote target through a gateway can concatenate two columns as part of the `SELECT` statement.

After data is loaded into the Oracle database, data transformations can be executed using SQL operations. A multitable `INSERT` statement is one of the techniques for implementing SQL data transformations.

Multitable INSERT Statements: Overview

- Use the INSERT…SELECT statement to insert rows into multiple tables as part of a single DML statement.
- Multitable INSERT statements are used in data warehousing systems to transfer data from one or more operational sources to a set of target tables.
- They provide significant performance improvement over:
  - Single DML versus multiple INSERT…SELECT statements
  - Single DML versus a procedure to perform multiple inserts by using the IF...THEN syntax

Multitable INSERT statements offer the benefits of the INSERT ... SELECT statement when multiple tables are involved as targets. Without multitable INSERT, you had to deal with *n* independent INSERT ... SELECT statements, thus processing the same source data *n* times and increasing the transformation workload *n* times.

As with the existing INSERT ... SELECT statement, the new statement can be parallelized and used with the direct-load mechanism for faster performance.

Each record from any input stream, such as a nonrelational database table, can now be converted into multiple records for a more relational database table environment. To alternatively implement this functionality, you were required to write multiple INSERT statements.

Types of Multitable `INSERT` Statements

The different types of multitable `INSERT` statements are:

- Unconditional `INSERT`
- Conditional `INSERT ALL`
- Conditional `INSERT FIRST`
- Pivoting `INSERT`

You use different clauses to indicate the type of `INSERT` to be executed. The types of multitable `INSERT` statements are:

- **Unconditional `INSERT`:** For each row returned by the subquery, a row is inserted into each of the target tables.
- **Conditional `INSERT ALL`:** For each row returned by the subquery, a row is inserted into each target table if the specified condition is met.
- **Pivoting `INSERT`:** This is a special case of the unconditional `INSERT ALL`.
- **Conditional `INSERT FIRST`:** For each row returned by the subquery, a row is inserted into the very first target table in which the condition is met.

Multitable INSERT Statements

- Syntax for multitable INSERT:

```
INSERT [conditional_insert_clause]
[insert_into_clause values_clause] (subquery)
```

- conditional_insert_clause:

```
[ALL|FIRST]
[WHEN condition THEN] [insert_into_clause values_clause]
[ELSE] [insert_into_clause values_clause]
```


The slide displays the generic format for multitable INSERT statements.

Unconditional INSERT: ALL into_clause

Specify ALL followed by multiple insert_into_clauses to perform an unconditional multitable INSERT. The Oracle Server executes each insert_into_clause once for each row returned by the subquery.

Conditional INSERT: conditional_insert_clause

Specify the conditional_insert_clause to perform a conditional multitable INSERT. The Oracle Server filters each insert_into_clause through the corresponding WHEN condition, which determines whether that insert_into_clause is executed. A single multitable INSERT statement can contain up to 127 WHEN clauses.

Conditional INSERT: ALL

If you specify ALL, the Oracle Server evaluates each WHEN clause regardless of the results of the evaluation of any other WHEN clause. For each WHEN clause whose condition evaluates to true, the Oracle Server executes the corresponding INTO clause list.

Conditional INSERT: FIRST

If you specify FIRST, the Oracle Server evaluates each WHEN clause in the order in which it appears in the statement. If the first WHEN clause evaluates to true, the Oracle Server executes the corresponding INTO clause and skips subsequent WHEN clauses for the given row.

Conditional INSERT: ELSE Clause

For a given row, if no WHEN clause evaluates to true:

- If you have specified an ELSE clause, the Oracle Server executes the INTO clause list associated with the ELSE clause
- If you did not specify an ELSE clause, the Oracle Server takes no action for that row

Restrictions on Multitable INSERT Statements

- You can perform multitable INSERT statements only on tables, and not on views or materialized views.
- You cannot perform a multitable INSERT on a remote table.
- You cannot specify a table collection expression when performing a multitable INSERT.
- In a multitable INSERT, all insert_into_clauses cannot combine to specify more than 999 target columns.

Unconditional INSERT ALL

- Select the EMPLOYEE_ID, HIRE_DATE, SALARY, and MANAGER_ID values from the EMPLOYEES table for those employees whose EMPLOYEE_ID is greater than 200.
- Insert these values into the SAL_HISTORY and MGR_HISTORY tables by using a multitable INSERT.

```
INSERT  ALL
   INTO sal_history VALUES(EMPID,HIREDATE,SAL)
   INTO mgr_history VALUES(EMPID,MGR,SAL)
   SELECT employee_id EMPID, hire_date HIREDATE,
          salary SAL, manager_id MGR
   FROM  employees
   WHERE employee_id > 200;
```

```
12 rows inserted
```


The example in the slide inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables.

The SELECT statement retrieves the details of employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The details of the employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The details of employee ID, manager ID, and salary are inserted into the MGR_HISTORY table.

This INSERT statement is referred to as an unconditional INSERT because no further restriction is applied to the rows that are retrieved by the SELECT statement. All the rows retrieved by the SELECT statement are inserted into the two tables: SAL_HISTORY and MGR_HISTORY. The VALUES clause in the INSERT statements specifies the columns from the SELECT statement that must be inserted into each of the tables. Each row returned by the SELECT statement results in two insertions: one for the SAL_HISTORY table and one for the MGR_HISTORY table.

A total of 12 rows were selected:

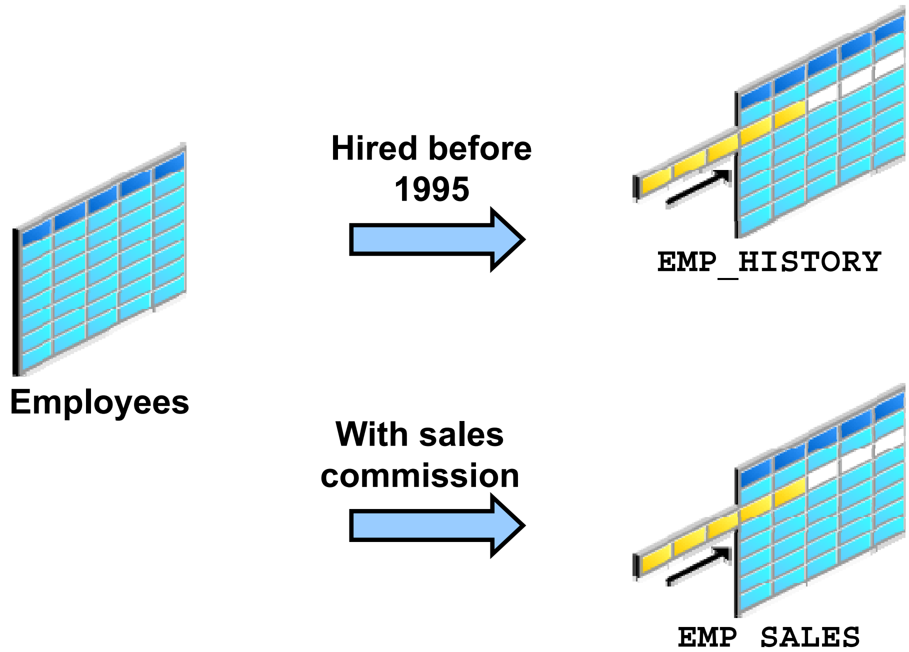
```
SELECT COUNT(*) total_in_sal FROM sal_history;
```

	TOTAL_IN_SAL
1	6

```
SELECT COUNT(*) total_in_mgr FROM mgr_history;
```

	TOTAL_IN_MGR
1	6

Conditional `INSERT ALL`: Example



For all employees in the employees tables, if the employee was hired before 1995, insert that employee record into the employee history. If the employee earns a sales commission, insert the record information into the `EMP_SALES` table. The SQL statement is shown on the next page.

Conditional INSERT ALL

```
INSERT  ALL
 WHEN HIREDATE < '01-JAN-95' THEN
   INTO emp_history VALUES(EMPID,HIREDATE,SAL)
 WHEN COMM IS NOT NULL THEN
   INTO emp_sales VALUES(EMPID,COMM,SAL)
   SELECT employee_id EMPID, hire_date HIREDATE,
          salary SAL, commission_pct COMM
   FROM   employees
```

```
48 rows inserted
```


The example in the slide is similar to the example in the previous slide because it inserts rows into both the EMP_HISTORY and the EMP_SALES tables. The SELECT statement retrieves details such as employee ID, hire date, salary, and commission percentage for all employees from the EMPLOYEES table. Details such as employee ID, hire date, and salary are inserted into the EMP_HISTORY table. Details such as employee ID, commission percentage, and salary are inserted into the EMP_SALES table.

This INSERT statement is referred to as a conditional INSERT ALL because a further restriction is applied to the rows that are retrieved by the SELECT statement. From the rows that are retrieved by the SELECT statement, only those rows in which the hire date was prior to 1995 are inserted in the EMP_HISTORY table. Similarly, only those rows where the value of commission percentage is not null are inserted in the EMP_SALES table.

```
SELECT count(*) FROM emp_history;
```

	COUNT(*)
1	13

```
SELECT count(*) FROM emp_sales;
```

	COUNT(*)
1	35

You can also optionally use the `ELSE` clause with the `INSERT ALL` statement.

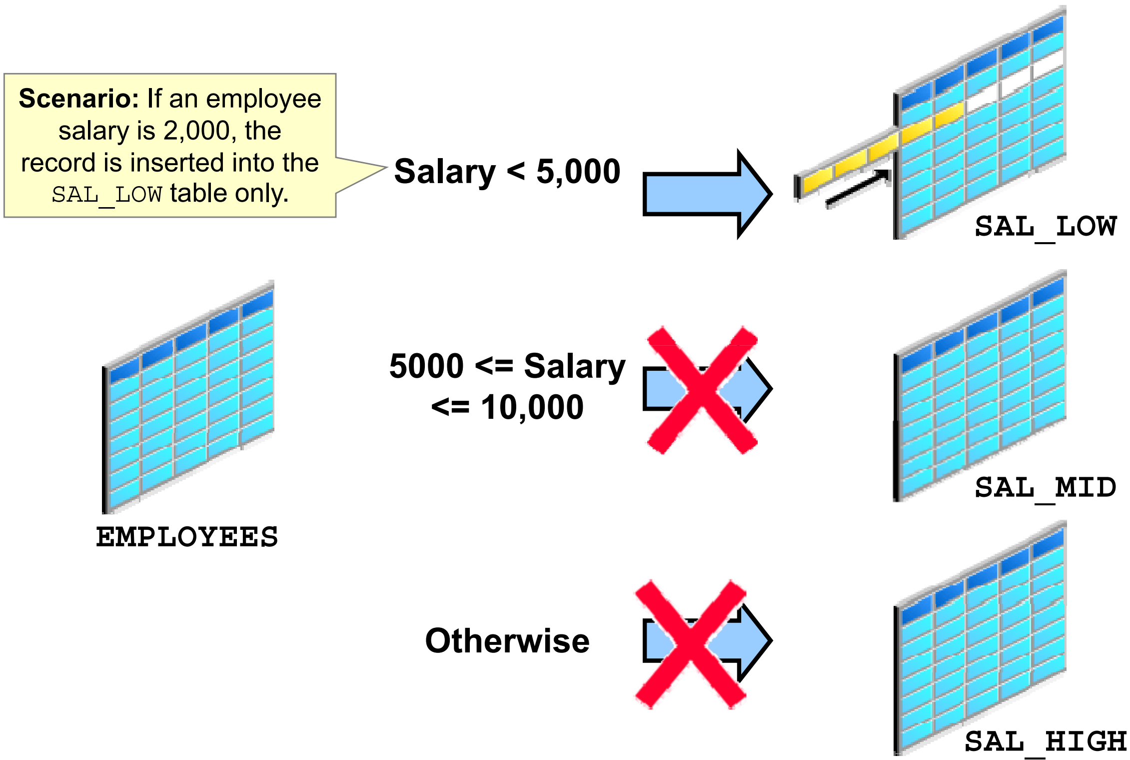
Example:

```
INSERT ALL
WHEN job_id IN
(select job_id FROM jobs WHERE job_title LIKE '%Manager%') THEN
INTO managers2(last_name,job_id,SALARY)
VALUES (last_name,job_id,SALARY)
WHEN SALARY>10000 THEN
INTO richpeople(last_name,job_id,SALARY)
VALUES (last_name,job_id,SALARY)
ELSE
INTO poorpeople VALUES (last_name,job_id,SALARY)
SELECT * FROM employees;
```

Result:

```
116 rows inserted
```

Conditional `INSERT FIRST`: Example



For all employees in the `EMPLOYEES` table, insert the employee information into the first target table that meets the condition. In the example, if an employee has a salary of 2,000, the record is inserted into the `SAL_LOW` table only. The SQL statement is shown on the next page.

Conditional INSERT FIRST

```
INSERT FIRST
WHEN salary < 5000 THEN
  INTO sal_low VALUES (employee_id, last_name, salary)
WHEN salary between 5000 and 10000 THEN
  INTO sal_mid VALUES (employee_id, last_name, salary)
ELSE
  INTO sal_high VALUES (employee_id, last_name, salary)
SELECT employee_id, last_name, salary
FROM employees
```

```
107 rows inserted
```


The `SELECT` statement retrieves details such as employee ID, last name, and salary for every employee in the `EMPLOYEES` table. For each employee record, it is inserted into the very first target table that meets the condition.

This `INSERT` statement is referred to as a conditional `INSERT FIRST`. The `WHEN salary < 5000` condition is evaluated first. If this first `WHEN` clause evaluates to true, the Oracle Server executes the corresponding `INTO` clause and inserts the record into the `SAL_LOW` table. It skips subsequent `WHEN` clauses for this row.

If the row does not satisfy the first `WHEN` condition (`WHEN salary < 5000`), the next condition (`WHEN salary between 5000 and 10000`) is evaluated. If this condition evaluates to true, the record is inserted into the `SAL_MID` table, and the last condition is skipped.

If neither the first condition (`WHEN salary < 5000`) nor the second condition (`WHEN salary between 5000 and 10000`) is evaluated to true, the Oracle Server executes the corresponding `INTO` clause for the `ELSE` clause.

A total of 20 rows were inserted:

```
SELECT count(*) low FROM sal_low;
```

	LOW
1	49

```
SELECT count(*) mid FROM sal_mid;
```

	MID
1	43

```
SELECT count(*) high FROM sal_high;
```

	HIGH
1	15

Pivoting `INSERT`

Convert the set of sales records from the nonrelational database table to relational format.

Emp_ID	Week_ID	MON	TUES	WED	THUR	FRI
176	6	2000	3000	4000	5000	6000

Employee_ID	WEEK	SALES
176	6	2000
176	6	3000
176	6	4000
176	6	5000
176	6	6000

Pivoting is an operation in which you must build a transformation such that each record from any input stream, such as a nonrelational database table, must be converted into multiple records for a more relational database table environment.

Suppose you receive a set of sales records from a nonrelational database table:

`SALES_SOURCE_DATA`, in the following format:

```
EMPLOYEE_ID, WEEK_ID, SALES_MON, SALES_TUE, SALES_WED,
SALES_THUR, SALES_FRI
```

You want to store these records in the `SALES_INFO` table in a more typical relational format:

```
EMPLOYEE_ID, WEEK, SALES
```

To solve this problem, you must build a transformation such that each record from the original nonrelational database table, `SALES_SOURCE_DATA`, is converted into five records for the data warehouse's `SALES_INFO` table. This operation is commonly referred to as *pivoting*.

The solution to this problem is shown on the next page.

Pivoting INSERT

```
INSERT ALL
  INTO sales_info VALUES (employee_id,week_id,sales_MON)
  INTO sales_info VALUES (employee_id,week_id,sales_TUE)
  INTO sales_info VALUES (employee_id,week_id,sales_WED)
  INTO sales_info VALUES (employee_id,week_id,sales_THUR)
  INTO sales_info VALUES (employee_id,week_id, sales_FRI)
  SELECT EMPLOYEE_ID, week_id, sales_MON, sales_TUE,
         sales_WED, sales_THUR,sales_FRI
  FROM sales_source_data;
```

```
5 rows inserted
```


In the example in the slide, the sales data is received from the nonrelational database table `SALES_SOURCE_DATA`, which is the details of the sales performed by a sales representative on each day of a week, for a week with a particular week ID.

`DESC SALES_SOURCE_DATA`

```
Name                           Null     Type
------------------------------ -------- -------------
EMPLOYEE_ID                             NUMBER(6)
WEEK_ID                                 NUMBER(2)
SALES_MON                               NUMBER(8,2)
SALES_TUE                               NUMBER(8,2)
SALES_WED                               NUMBER(8,2)
SALES_THUR                              NUMBER(8,2)
SALES_FRI                               NUMBER(8,2)
```

```
SELECT * FROM SALES_SOURCE_DATA;
```

	EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
1	178	6	1750	2200	1500	1500	3000

```
DESC SALES_INFO
```

```
Name                           Null     Type
------------------------------ -------- ------------
EMPLOYEE_ID                             NUMBER(6)
WEEK                                    NUMBER(2)
SALES                                   NUMBER(8,2)
```

```
SELECT * FROM sales_info;
```

	EMPLOYEE_ID	WEEK	SALES
1	178	6	1750
2	178	6	2200
3	178	6	1500
4	178	6	1500
5	178	6	3000

Observe in the preceding example that by using a pivoting `INSERT`, one row from the `SALES_SOURCE_DATA` table is converted into five records for the relational table, `SALES_INFO`.

Lesson Agenda

- Manipulating data by using subqueries
- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERTs`:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Tracking the changes to data over a period of time

MERGE Statement

- Provides the ability to conditionally update, insert, or delete data into a database table
- Performs an UPDATE if the row exists, and an INSERT if it is a new row:
  - Avoids separate updates
  - Increases performance and ease of use
  - Is useful in data warehousing applications

The Oracle Server supports the MERGE statement for INSERT, UPDATE, and DELETE operations. Using this statement, you can update, insert, or delete a row conditionally into a table, thus avoiding multiple DML statements. The decision whether to update, insert, or delete into the target table is based on a condition in the ON clause.

You must have the INSERT and UPDATE object privileges on the target table and the SELECT object privilege on the source table. To specify the DELETE clause of merge_update_clause, you must also have the DELETE object privilege on the target table.

The MERGE statement is deterministic. You cannot update the same row of the target table multiple times in the same MERGE statement.

An alternative approach is to use PL/SQL loops and multiple DML statements. The MERGE statement, however, is easy to use and more simply expressed as a single SQL statement.

The MERGE statement is suitable in a number of data warehousing applications. For example, in a data warehousing application, you may need to work with data coming from multiple sources, some of which may be duplicates. With the MERGE statement, you can conditionally add or modify rows.

MERGE Statement Syntax

You can conditionally insert, update, or delete rows in a table by using the MERGE statement.

```
MERGE INTO table_name table_alias
  USING (table|view|sub_query) alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
    col1 = col1_val,
    col2 = col2_val
 WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```


Merging Rows

You can update existing rows, and insert new rows conditionally by using the MERGE statement. Using the MERGE statement, you can delete obsolete rows at the same time as you update rows in a table. To do this, you include a DELETE clause with its own WHERE clause in the syntax of the MERGE statement.

In the syntax:

INTO clause	Specifies the target table you are updating or inserting into
USING clause	Identifies the source of the data to be updated or inserted; can be a table, view, or subquery
ON clause	The condition on which the MERGE operation either updates or inserts
WHEN MATCHED | WHEN NOT MATCHED	Instructs the server how to respond to the results of the join condition

Note: For more information, see *Oracle Database SQL Reference for 10g or 11g database*.

Merging Rows: Example

Insert or update rows in the `COPY_EMP3` table to match the `EMPLOYEES` table.

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
...
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
e.department_id);
```



```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
c.email = e.email,
c.phone_number = e.phone_number,
c.hire_date = e.hire_date,
c.job_id = e.job_id,
c.salary = e.salary*2,
c.commission_pct = e.commission_pct,
c.manager_id = e.manager_id,
c.department_id = e.department_id
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
```

```
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
e.department_id);
```

The `COPY_EMP3` table is created by using the following code:

```
CREATE TABLE COPY_EMP3 AS SELECT * FROM EMPLOYEES
WHERE SALARY<10000;
```

Then query the `COPY_EMP3` table.

```
SELECT employee_id, salary, commission_pct FROM COPY_EMP3;
```

	EMPLOYEE_ID	SALARY	COMMISSION_PCT
1	198	5200	(null)
2	199	5200	(null)
3	200	8800	(null)
4	202	12000	(null)
5	203	13000	(null)

...

64	197	6000	(null)
65	162	10500	0.25
66	146	13500	0.3
67	150	10000	0.3

...

Observe that there are some employees with `SALARY < 10000` and there are two employees with `COMMISSION_PCT`.

The example in the slide matches the `EMPLOYEE_ID` in the `COPY_EMP3` table to the `EMPLOYEE_ID` in the `EMPLOYEES` table. If a match is found, the row in the `COPY_EMP3` table is updated to match the row in the `EMPLOYEES` table and the salary of the employee is doubled. The records of the two employees with values in the `COMMISSION_PCT` column are deleted. If the match is not found, rows are inserted into the `COPY_EMP3` table.

Merging Rows: Example

```
TRUNCATE TABLE copy_emp3;
SELECT * FROM copy_emp3;
0 rows selected
```

```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
...
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, ...
```

```
SELECT * FROM copy_emp3;
107 rows selected.
```


The examples in the slide show that the `COPY_EMP3` table is empty. The `c.employee_id = e.employee_id` condition is evaluated. The condition returns false—there are no matches. The logic falls into the `WHEN NOT MATCHED` clause, and the `MERGE` command inserts the rows of the `EMPLOYEES` table into the `COPY_EMP3` table. This means that the `COPY_EMP3` table now has exactly the same data as in the `EMPLOYEES` table.

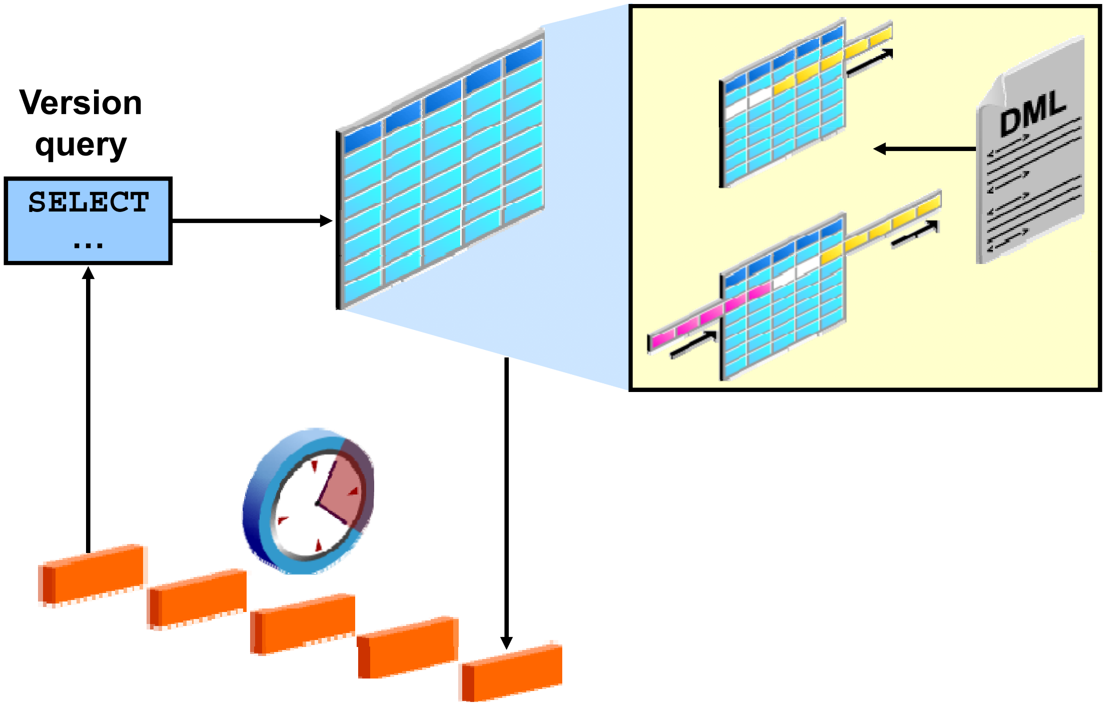
```
SELECT employee_id, salary, commission_pct from copy_emp3;
```

	EMPLOYEE_ID	SALARY	COMMISSION_PCT
1	144	2500	(null)
2	143	2600	(null)
3	202	6000	(null)
4	141	3500	(null)
5	174	11000	0.3
...			
15	149	10500	0.2
16	206	8300	(null)
17	176	8600	0.2
18	124	5800	(null)
19	205	12000	(null)
20	178	7000	0.15

Lesson Agenda

- Manipulating data by using subqueries
- Specifying explicit default values in the `INSERT` and `UPDATE` statements
- Using the following types of multitable `INSERTs`:
  - Unconditional `INSERT`
  - Conditional `INSERT ALL`
  - Conditional `INSERT FIRST`
  - Pivoting `INSERT`
- Merging rows in a table
- Tracking the changes to data over a period of time

Tracking Changes in Data



Versions of retrieved rows

You may discover that, somehow, data in a table has been inappropriately changed. To research this, you can use multiple flashback queries to view row data at specific points in time. More efficiently, you can use the Flashback Version Query feature to view all changes to a row over a period of time. This feature enables you to append a `VERSIONS` clause to a `SELECT` statement that specifies a system change number (SCN) or the time stamp range within which you want to view changes to row values. The query also can return associated metadata, such as the transaction responsible for the change.

Further, after you identify an erroneous transaction, you can use the Flashback Transaction Query feature to identify other changes that were done by the transaction. You then have the option of using the Flashback Table feature to restore the table to a state before the changes were made.

You can use a query on a table with a `VERSIONS` clause to produce all the versions of all the rows that exist, or ever existed, between the time the query was issued and the `undo_retention` seconds before the current time. `undo_retention` is an initialization parameter, which is an autotuned parameter. A query that includes a `VERSIONS` clause is referred to as a version query. The results of a version query behaves as though the `WHERE` clause were applied to the versions of the rows. The version query returns versions of the rows only across transactions.

System change number (SCN): The Oracle server assigns an SCN to identify the redo records for each committed transaction.

Flashback Version Query: Example

```
SELECT salary FROM employees3
WHERE  employee_id = 107;
```

1

```
UPDATE employees3 SET salary = salary * 1.30
WHERE  employee_id = 107;

COMMIT;
```

2

```
SELECT salary FROM employees3
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE  employee_id = 107;
```

3

1

	SALARY
1	4200

3

	SALARY
1	5460
2	4200

In the example in the slide, the salary for employee 107 is retrieved (1). The salary for employee 107 is increased by 30 percent and this change is committed (2). The different versions of salary are displayed (3).

The `VERSIONS` clause does not change the plan of the query. For example, if you run a query on a table that uses the index access method, the same query on the same table with a `VERSIONS` clause continues to use the index access method. The versions of the rows returned by the version query are versions of the rows across transactions. The `VERSIONS` clause has no effect on the transactional behavior of a query. This means that a query on a table with a `VERSIONS` clause still inherits the query environment of the ongoing transaction.

The default `VERSIONS` clause can be specified as `VERSIONS BETWEEN {SCN|TIMESTAMP} MINVALUE AND MAXVALUE`.

The `VERSIONS` clause is a SQL extension only for queries. You can have DML and DDL operations that use a `VERSIONS` clause within subqueries. The row version query retrieves all the committed versions of the selected rows. Changes made by the current active transaction are not returned. The version query retrieves all incarnations of the rows. This essentially means that versions returned include deleted and subsequent reinserted versions of the rows.

The row access for a version query can be defined in one of the following two categories:

- **ROWID-based row access:** In case of ROWID-based access, all versions of the specified ROWID are returned irrespective of the row content. This essentially means that all versions of the slot in the block indicated by the ROWID are returned.
- **All other row access:** For all other row access, all versions of the rows are returned.

VERSIONS BETWEEN Clause

```
SELECT versions_starttime "START_DATE",
       versions_endtime   "END_DATE",
       salary
FROM   employees
    VERSIONS BETWEEN SCN MINVALUE
    AND MAXVALUE
WHERE  last_name = 'Lorentz';
```

	START_DATE	END_DATE	SALARY
1	18-JUN-09 05.07.10.000000000 PM	(null)	5460
2	(null)	18-JUN-09 05.07.10.000000000 PM	4200

You can use the VERSIONS BETWEEN clause to retrieve all the versions of the rows that exist or have ever existed between the time the query was issued and a point back in time.

If the undo retention time is less than the lower bound time or the SCN of the BETWEEN clause, the query retrieves versions up to the undo retention time only. The time interval of the BETWEEN clause can be specified as an SCN interval or a wall-clock interval. This time interval is closed at both the lower and the upper bounds.

In the example, Lorentz's salary changes are retrieved. The NULL value for END_DATE for the first version indicates that this was the existing version at the time of the query. The NULL value for START_DATE for the last version indicates that this version was created at a time before the undo retention time.

Quiz

When you use the `INSERT` or `UPDATE` command, the `DEFAULT` keyword saves you from hard-coding the default value in your programs or querying the dictionary to find it.

a. True

b. False

Answer: a

Summary

In this lesson, you should have learned how to:

- Use DML statements and control transactions
- Describe the features of multitable INSERTs
- Use the following types of multitable INSERTs:
  - Unconditional INSERT
  - Conditional INSERT ALL
  - Conditional INSERT FIRST
  - Pivoting INSERT
- Merge rows in a table
- Manipulate data by using subqueries
- Track the changes to data over a period of time

In this lesson, you should have learned how to manipulate data in the Oracle database by using subqueries. You also should have learned about multitable INSERT statements, the MERGE statement, and tracking changes in the database.

Practice 5: Overview

This practice covers the following topics:

- Performing multitable `INSERT`s
- Performing `MERGE` operations
- Tracking row versions

6

Managing Data in Different Time Zones

Objectives

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

- Use data types similar to DATE that store fractional seconds and track time zones
- Use data types that store the difference between two datetime values
- Use the following datetime functions:
  - CURRENT_DATE
  - CURRENT_TIMESTAMP
  - LOCALTIMESTAMP
  - DBTIMEZONE
  - SESSIONTIMEZONE
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

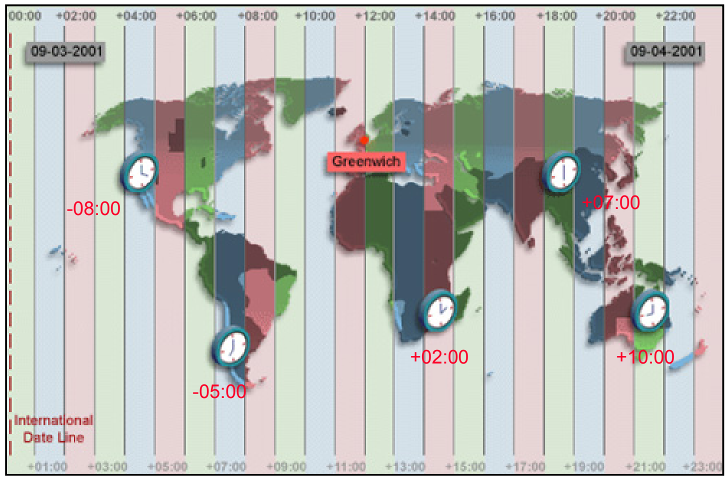
ORACLE

In this lesson, you learn how to use data types similar to DATE that store fractional seconds and track time zones. This lesson addresses some of the datetime functions available in the Oracle database.

Lesson Agenda

- CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP
- INTERVAL data types
- Using the following functions:
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

Time Zones



The image represents the time for each time zone when Greenwich time is 12:00.

The hours of the day are measured by the turning of the earth. The time of day at any particular moment depends on where you are. When it is noon in Greenwich, England, it is midnight along the International Date Line. The earth is divided into 24 time zones, one for each hour of the day. The time along the prime meridian in Greenwich, England, is known as Greenwich Mean Time (GMT). GMT is now known as Coordinated Universal Time (UTC). UTC is the time standard against which all other time zones in the world are referenced. It is the same all year round and is not affected by summer time or daylight saving time. The meridian line is an imaginary line that runs from the North Pole to the South Pole. It is known as zero longitude and it is the line from which all other lines of longitude are measured. All time is measured relative to UTC and all places have a latitude (their distance north or south of the equator) and a longitude (their distance east or west of the Greenwich meridian).

TIME_ZONE Session Parameter

TIME_ZONE may be set to:

- An absolute offset
- Database time zone
- OS local time zone
- A named region

```
ALTER SESSION SET TIME_ZONE = '-05:00';
ALTER SESSION SET TIME_ZONE = dbtimezone;
ALTER SESSION SET TIME_ZONE = local;
ALTER SESSION SET TIME_ZONE = 'America/New_York';
```


The Oracle database supports storing the time zone in your date and time data, as well as fractional seconds. The ALTER SESSION command can be used to change time zone values in a user's session. The time zone values can be set to an absolute offset, a named time zone, a database time zone, or the local time zone.

CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP

- CURRENT_DATE:
  - Returns the current date from the user session
  - Has a data type of DATE
- CURRENT_TIMESTAMP:
  - Returns the current date and time from the user session
  - Has a data type of TIMESTAMP WITH TIME ZONE
- LOCALTIMESTAMP:
  - Returns the current date and time from the user session
  - Has a data type of TIMESTAMP

The CURRENT_DATE and CURRENT_TIMESTAMP functions return the current date and current time stamp, respectively. The data type of CURRENT_DATE is DATE. The data type of CURRENT_TIMESTAMP is TIMESTAMP WITH TIME ZONE. The values returned display the time zone displacement of the SQL session executing the functions. The time zone displacement is the difference (in hours and minutes) between local time and UTC. The TIMESTAMP WITH TIME ZONE data type has the format:

```
TIMESTAMP [ (fractional_seconds_precision) ] WITH TIME ZONE
```

where 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 through 9. The default is 6.

The LOCALTIMESTAMP function returns the current date and time in the session time zone. The difference between LOCALTIMESTAMP and CURRENT_TIMESTAMP is that LOCALTIMESTAMP returns a TIMESTAMP value, whereas CURRENT_TIMESTAMP returns a TIMESTAMP WITH TIME ZONE value.

These functions are national language support (NLS)–sensitive—that is, the results will be in the current NLS calendar and datetime formats.

Note: The SYSDATE function returns the current date and time as a DATE data type. You learned how to use the SYSDATE function in the course titled *Oracle Database: SQL Fundamentals I*.

Comparing Date and Time in a Session's Time Zone

The TIME_ZONE parameter is set to –5:00 and then SELECT statements for each date and time are executed to compare differences.

```
ALTER SESSION
SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
ALTER SESSION SET TIME_ZONE = '-5:00';

SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;          1


SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;     2


SELECT SESSIONTIMEZONE, LOCALTIMESTAMP FROM DUAL;        3
```


The ALTER SESSION command sets the date format of the session to 'DD-MON-YYYY HH24:MI:SS'–that is, day of month (1–31)-abbreviated name of month-4-digit year hour of day (0–23):minute (0–59):second (0–59).

The example in the slide illustrates that the session is altered to set the TIME_ZONE parameter to –5:00. Then the SELECT statement for CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP is executed to observe the differences in format.

Note: The TIME_ZONE parameter specifies the default local time zone displacement for the current SQL session. TIME_ZONE is a session parameter only, not an initialization parameter. The TIME_ZONE parameter is set as follows:

```
TIME_ZONE = '[+ | -] hh:mm'
```

The format mask ([+ | -] hh:mm) indicates the hours and minutes before or after UTC.

Comparing Date and Time in a Session's Time Zone

Results of queries:

```
session SET altered.
```

1

	SESSIONTIMEZONE	CURRENT_DATE
1	-05:00	23-JUN-2009 01:34:52

2

	SESSIONTIMEZONE	CURRENT_TIMESTAMP
1	-05:00	23-JUN-09 01.35.26.239882000 AM -05:00

3

	SESSIONTIMEZONE	LOCALTIMESTAMP
1	-05:00	23-JUN-09 01.36.21.811798000 AM

In this case, the `CURRENT_DATE` function returns the current date in the session's time zone, the `CURRENT_TIMESTAMP` function returns the current date and time in the session's time zone as a value of the data type `TIMESTAMP WITH TIME ZONE`, and the `LOCALTIMESTAMP` function returns the current date and time in the session's time zone.

DBTIMEZONE and SESSIONTIMEZONE

- Display the value of the database time zone:

```
SELECT DBTIMEZONE FROM DUAL;
```

	DBTIMEZONE
1	+00:00

- Display the value of the session's time zone:

```
SELECT SESSIONTIMEZONE FROM DUAL;
```

	SESSIONTIMEZONE
1	-05:00

The DBA sets the database's default time zone by specifying the `SET TIME_ZONE` clause of the `CREATE DATABASE` statement. If omitted, the default database time zone is the operating system time zone. The database time zone cannot be changed for a session with an `ALTER SESSION` statement.

The `DBTIMEZONE` function returns the value of the database time zone. The return type is a time zone offset (a character type in the format: `'[+|-]TZH:TZM'`) or a time zone region name, depending on how the user specified the database time zone value in the most recent `CREATE DATABASE` or `ALTER DATABASE` statement. The example in the slide shows that the database time zone is set to "–05:00," as the `TIME_ZONE` parameter is in the format:

```
TIME_ZONE = '[+ | -] hh:mm'
```

The `SESSIONTIMEZONE` function returns the value of the current session's time zone. The return type is a time zone offset (a character type in the format `'[+|-]TZH:TZM'`) or a time zone region name, depending on how the user specified the session time zone value in the most recent `ALTER SESSION` statement. The example in the slide shows that the session time zone is offset to UTC by –8 hours. Observe that the database time zone is different from the current session's time zone.

TIMESTAMP Data Types

Data Type	Fields
TIMESTAMP	Year, Month, Day, Hour, Minute, Second with fractional seconds
TIMESTAMP WITH TIME ZONE	Same as the TIMESTAMP data type; also includes: TIMEZONE_HOUR, and TIMEZONE_MINUTE or TIMEZONE_REGION
TIMESTAMP WITH LOCAL TIME ZONE	Same as the TIMESTAMP data type; also includes a time zone offset in its value

The TIMESTAMP data type is an extension of the DATE data type.

```
TIMESTAMP (fractional_seconds_ precision)
```

This data type contains the year, month, and day values of date, as well as hour, minute, and second values of time, where significant fractional seconds precision is the number of digits in the fractional part of the SECOND datetime field. The accepted values of significant fractional_seconds_precision are 0 through 9. The default is 6.

```
TIMESTAMP (fractional_seconds_precision) WITH TIME ZONE
```

This data type contains all values of TIMESTAMP as well as time zone displacement value.

```
TIMESTAMP (fractional_seconds_precision) WITH LOCAL TIME ZONE
```

This data type contains all values of TIMESTAMP, with the following exceptions:

- Data is normalized to the database time zone when it is stored in the database.
- When the data is retrieved, users see the data in the session time zone.

`TIMESTAMP` Fields

Datetime Field	Valid Values
`YEAR`	–4712 to 9999 (excluding year 0)
`MONTH`	01 to 12
`DAY`	01 to 31
`HOUR`	00 to 23
`MINUTE`	00 to 59
`SECOND`	00 to 59.9(N) where 9(N) is precision
`TIMEZONE_HOUR`	–12 to 14
`TIMEZONE_MINUTE`	00 to 59

Each datetime data type is composed of several of these fields. Datetimes are mutually comparable and assignable only if they have the same datetime fields.

Difference Between `DATE` and `TIMESTAMP`

A

```
-- when hire_date is
of type DATE

SELECT hire_date
FROM employees;
```

	HIRE_DATE
1	21-JUN-99
2	13-JAN-00
3	17-SEP-87
4	17-FEB-96
5	17-AUG-97
6	07-JUN-94
7	07-JUN-94
8	07-JUN-94

B

```
ALTER TABLE employees
MODIFY hire_date
TIMESTAMP(7);
SELECT hire_date
FROM employees;
```

	HIRE_DATE
1	21-JUN-99 12.00.00.000000000 AM
2	13-JAN-00 12.00.00.000000000 AM
3	17-SEP-87 12.00.00.000000000 AM
4	17-FEB-96 12.00.00.000000000 AM
5	17-AUG-97 12.00.00.000000000 AM
6	07-JUN-94 12.00.00.000000000 AM
7	07-JUN-94 12.00.00.000000000 AM
8	07-JUN-94 12.00.00.000000000 AM

...

ORACLE

`TIMESTAMP` Data Type: Example

In the slide, example A shows the data from the `hire_date` column of the `EMPLOYEES` table when the data type of the column is `DATE`. In example B, the table is altered and the data type of the `hire_date` column is made into `TIMESTAMP`. The output shows the differences in display. You can convert from `DATE` to `TIMESTAMP` when the column has data, but you cannot convert from `DATE` or `TIMESTAMP` to `TIMESTAMP WITH TIME ZONE` unless the column is empty.

You can specify the fractional seconds precision for time stamp. If none is specified, as in this example, it defaults to 6.

For example, the following statement sets the fractional seconds precision as 7:

```
ALTER TABLE employees
MODIFY hire_date TIMESTAMP(7);
```

Note: The Oracle date data type by default appears as shown in this example. However, the date data type also contains additional information such as hours, minutes, seconds, AM, and PM. To obtain the date in this format, you can apply a format mask or a function to the date value.

Comparing TIMESTAMP Data Types

```
CREATE TABLE web_orders
(order_date TIMESTAMP WITH TIME ZONE,
 delivery_time TIMESTAMP WITH LOCAL TIME ZONE);
```

```
INSERT INTO web_orders values
(current_date, current_timestamp + 2);
```

```
SELECT * FROM web_orders;
```

	ORDER_DATE	DELIVERY_TIME
1	23-JUN-09 01.56.39.000000000 AM -05:00	25-JUN-09 01.56.39.000000000 AM

In the example in the slide, a new table `web_orders` is created with a column of data type `TIMESTAMP WITH TIME ZONE` and a column of data type `TIMESTAMP WITH LOCAL TIME ZONE`. This table is populated whenever a `web_order` is placed. The time stamp and time zone for the user placing the order are inserted based on the `CURRENT_DATE value`. The local time stamp and time zone are populated by inserting two days from the `CURRENT_TIMESTAMP` value into it every time an order is placed. When a web-based company guarantees shipping, they can estimate their delivery time based on the time zone of the person placing the order.

Lesson Agenda

- CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP
- INTERVAL data types
- Using the following functions:
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

`INTERVAL` Data Types

- `INTERVAL` data types are used to store the difference between two datetime values.
- There are two classes of intervals:
  - Year-month
  - Day-time
- The precision of the interval is:
  - The actual subset of fields that constitutes an interval
  - Specified in the interval qualifier

Data Type	Fields
`INTERVAL YEAR TO MONTH`	Year, Month
`INTERVAL DAY TO SECOND`	Days, Hour, Minute, Second with fractional seconds

`INTERVAL` data types are used to store the difference between two datetime values. There are two classes of intervals: year-month intervals and day-time intervals. A year-month interval is made up of a contiguous subset of fields of `YEAR` and `MONTH`, whereas a day-time interval is made up of a contiguous subset of fields consisting of `DAY`, `HOUR`, `MINUTE`, and `SECOND`. The actual subset of fields that constitute an interval is called the precision of the interval and is specified in the interval qualifier. Because the number of days in a year is calendar-dependent, the year-month interval is NLS-dependent, whereas day-time interval is NLS-independent.

The interval qualifier may also specify the leading field precision, which is the number of digits in the leading or only field, and in case the trailing field is `SECOND`, it may also specify the fractional seconds precision, which is the number of digits in the fractional part of the `SECOND` value. If not specified, the default value for leading field precision is 2 digits, and the default value for fractional seconds precision is 6 digits.

INTERVAL YEAR (year_precision) TO MONTH

This data type stores a period of time in years and months, where `year_precision` is the number of digits in the `YEAR` datetime field. The accepted values are 0 through 9. The default is 6.

INTERVAL DAY (day_precision) TO SECOND (fractional_seconds_precision)

This data type stores a period of time in days, hours, minutes, and seconds, where `day_precision` is the maximum number of digits in the `DAY` datetime field (accepted values are 0 through 9; the default is 2), and `fractional_seconds_precision` is the number of digits in the fractional part of the `SECOND` field. The accepted values are 0 through 9. The default is 6.

INTERVAL Fields

INTERVAL Field	Valid Values for Interval
YEAR	Any positive or negative integer
MONTH	00 to 11
DAY	Any positive or negative integer
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 59.9(N) where 9(N) is precision

INTERVAL YEAR TO MONTH can have fields of YEAR and MONTH.

INTERVAL DAY TO SECOND can have fields of DAY, HOUR, MINUTE, and SECOND.

The actual subset of fields that constitute an item of either type of interval is defined by an interval qualifier, and this subset is known as the precision of the item.

Year-month intervals are mutually comparable and assignable only with other year-month intervals, and day-time intervals are mutually comparable and assignable only with other day-time intervals.

INTERVAL YEAR TO MONTH: Example

```
CREATE TABLE warranty
(prod_id number,  warranty_time INTERVAL YEAR(3) TO
MONTH);
INSERT INTO warranty VALUES (123, INTERVAL '8' MONTH);
INSERT INTO warranty VALUES (155, INTERVAL '200'
YEAR(3));
INSERT INTO warranty VALUES (678, '200-11');
SELECT * FROM warranty;
```

	PROD_ID	WARRANTY_TIME
1	123	0-8
2	155	200-0
3	678	200-11

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. Specify INTERVAL YEAR TO MONTH as follows:

`INTERVAL YEAR [(year_precision)] TO MONTH`

where year_precision is the number of digits in the YEAR datetime field. The default value of year_precision is 2.

Restriction: The leading field must be more significant than the trailing field. For example, INTERVAL '0-1' MONTH TO YEAR is not valid.

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 three 18

The Oracle database supports two interval data types: `INTERVAL YEAR TO MONTH` and `INTERVAL DAY TO SECOND`; the column type, PL/SQL argument, variable, and return type must be one of the two. However, for interval literals, the system recognizes other American National Standards Institute (ANSI) interval types such as `INTERVAL '2' YEAR` or `INTERVAL '10' HOUR`. In these cases, each interval is converted to one of the two supported types.

In the example in the slide, a `WARRANTY` table is created, which contains a `warranty_time` column that takes the `INTERVAL YEAR(3) TO MONTH` data type. Different values are inserted into it to indicate years and months for various products. When these rows are retrieved from the table, you see a year value separated from the month value by a (`-`).

INTERVAL DAY TO SECOND Data Type: Example

```
CREATE TABLE lab
( exp_id number, test_time INTERVAL DAY(2) TO SECOND);

INSERT INTO lab VALUES (100012, '90 00:00:00');
INSERT INTO lab VALUES (56098,
  INTERVAL '6 03:30:16' DAY  TO SECOND);
```

```
SELECT * FROM lab;
```

	EXP_ID	TEST_TIME
1	100012	90 0:0:0.0
2	56098	6 3:30:16.0

In the example in the slide, you create the lab table with a `test_time` column of the `INTERVAL DAY TO SECOND` data type. You then insert into it the value `‘90 00:00:00’` to indicate 90 days and 0 hours, 0 minutes, and 0 seconds, and `INTERVAL '6 03:30:16' DAY TO SECOND` to indicate 6 days, 3 hours, 30 minutes, and 16 seconds. The `SELECT` statement shows how this data is displayed in the database.

Lesson Agenda

- CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP
- INTERVAL data types
- Using the following functions:
  - EXTRACT
  - TZ_OFFSET
  - FROM_TZ
  - TO_TIMESTAMP
  - TO_YMINTERVAL
  - TO_DSINTERVAL

EXTRACT

- Display the YEAR component from the SYSDATE.

```
SELECT EXTRACT (YEAR FROM SYSDATE) FROM DUAL;
```

	EXTRACT(YEARFROMSYSDATE)
1	2009

- Display the MONTH component from the HIRE_DATE for those employees whose MANAGER_ID is 100.

```
SELECT last_name, hire_date,
       EXTRACT (MONTH FROM HIRE_DATE)
FROM employees
WHERE manager_id = 100;
```

	LAST_NAME	HIRE_DATE	EXTRACT(MONTHFROMHIRE_DATE)
1	Hartstein	17-FEB-1996 00:00:00	2
2	Kochhar	21-SEP-1989 00:00:00	9
3	De Haan	13-JAN-1993 00:00:00	1
4	Raphaely	07-DEC-1994 00:00:00	12
5	Weiss	18-JUL-1996 00:00:00	7

The EXTRACT expression extracts and returns the value of a specified datetime field from a datetime or interval value expression. You can extract any of the components mentioned in the following syntax using the EXTRACT function. The syntax of the EXTRACT function is:

```
SELECT  EXTRACT ([YEAR]  [MONTH] [DAY]  [HOUR]  [MINUTE] [SECOND]
                 [TIMEZONE_HOUR]  [TIMEZONE_MINUTE]
                 [TIMEZONE_REGION]  [TIMEZONE_ABBR]
FROM [datetime_value_expression]  [interval_value_expression]);
```

When you extract a TIMEZONE_REGION or TIMEZONE_ABBR (abbreviation), the value returned is a string containing the appropriate time zone name or abbreviation. When you extract any of the other values, the value returned is a date in the Gregorian calendar. When extracting from a datetime with a time zone value, the value returned is in UTC.

In the first example in the slide, the EXTRACT function is used to extract the YEAR from SYSDATE. In the second example in the slide, the EXTRACT function is used to extract the MONTH from the HIRE_DATE column of the EMPLOYEES table for those employees who report to the manager whose EMPLOYEE_ID is 100.

TZ_OFFSET

Display the time zone offset for the 'US/Eastern', 'Canada/Yukon' and 'Europe/London' time zones:

```
SELECT TZ_OFFSET('US/Eastern'),
       TZ_OFFSET('Canada/Yukon'),
       TZ_OFFSET('Europe/London')
 FROM DUAL;
```

	TZ_OFFSET('US/EASTERN')	TZ_OFFSET('CANADA/YUKON')	TZ_OFFSET('EUROPE/LONDON')
1	-04:00	-07:00	+01:00

The TZ_OFFSET function returns the time zone offset corresponding to the value entered. The return value is dependent on the date when the statement is executed. For example, if the TZ_OFFSET function returns a value –08:00, this value indicates that the time zone where the command was executed is eight hours behind UTC. You can enter a valid time zone name, a time zone offset from UTC (which simply returns itself), or the keyword SESSIONTIMEZONE or DBTIMEZONE. The syntax of the TZ_OFFSET function is:

```
TZ_OFFSET ( ['time_zone_name'] '[+ | -] hh:mm' ]
                  [ SESSIONTIMEZONE] [DBTIMEZONE]
```

The Fold Motor Company has its headquarters in Michigan, USA, which is in the US/Eastern time zone. The company president, Mr. Fold, wants to conduct a conference call with the vice president of the Canadian operations and the vice president of European operations, who are in the Canada/Yukon and Europe/London time zones, respectively. Mr. Fold wants to find out the time in each of these places to make sure that his senior management will be available to attend the meeting. His secretary, Mr. Scott, helps by issuing the queries shown in the example and gets the following results:

- The 'US/Eastern' time zone is four hours behind UTC.
- The 'Canada/Yukon' time zone is seven hours behind UTC.
- The 'Europe/London' time zone is one hour ahead of UTC.

For a listing of valid time zone name values, you can query the V$TIMEZONE_NAMES dynamic performance view.

```
SELECT * FROM V$TIMEZONE_NAMES;
```

	TZNAME	TZABBREV
1	Africa/Abidjan	LMT
2	Africa/Abidjan	GMT
3	Africa/Accra	LMT
4	Africa/Accra	GMT
5	Africa/Accra	GHST

...

FROM_TZ

Display the TIMESTAMP value '2000-03-28 08:00:00' as a TIMESTAMP WITH TIME ZONE value for the 'Australia/North' time zone region.

```
SELECT FROM_TZ(TIMESTAMP
       '2000-07-12 08:00:00', 'Australia/North')
FROM DUAL;
```

	FROM_TZ(TIMESTAMP'2000-07-1208:00:00','AUSTRALIA/NORTH')
1	12-JUL-00 08.00.00.000000000 AM AUSTRALIA/NORTH

The FROM_TZ function converts a TIMESTAMP value to a TIMESTAMP WITH TIME ZONE value.

The syntax of the FROM_TZ function is as follows:

```
FROM_TZ(TIMESTAMP timestamp_value, time_zone_value)
```

where time_zone_value is a character string in the format 'TZH:TZM' or a character expression that returns a string in TZR (time zone region) with an optional TZD format. TZD is an abbreviated time zone string with daylight saving information. TZR represents the time zone region in datetime input strings. Examples are 'Australia/North', 'PST' for US/Pacific standard time, 'PDT' for US/Pacific daylight time, and so on.

The example in the slide converts a TIMESTAMP value to TIMESTAMP WITH TIME ZONE.

Note: To see a listing of valid values for the TZR and TZD format elements, query the V$TIMEZONE_NAMES dynamic performance view.

TO_TIMESTAMP

Display the character string '2007-03-06 11:00:00' as a TIMESTAMP value:

```
SELECT TO_TIMESTAMP ('2007-03-06 11:00:00',
                     'YYYY-MM-DD HH:MI:SS')
FROM DUAL;
```

TO_TIMESTAMP('2007-03-0611:00:00','YYYY-MM-DDHH:MI:SS')
06-MAR-07 11.00.00.000000000

The TO_TIMESTAMP function converts a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of the TIMESTAMP data type. The syntax of the TO_TIMESTAMP function is:

```
TO_TIMESTAMP (char,[fmt],['nlsparam'])
```

The optional fmt specifies the format of char. If you omit fmt, the string must be in the default format of the TIMESTAMP data type. The optional nlsparam specifies the language in which month and day names, and abbreviations, are returned. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

If you omit nlsparams, this function uses the default date language for your session.

The example in the slide converts a character string to a value of TIMESTAMP.

Note: You use the TO_TIMESTAMP_TZ function to convert a string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to a value of the TIMESTAMP WITH TIME ZONE data type. For more information about this function, see *Oracle Database SQL Language Reference for 10g or 11g database*.

TO_YMINTERVAL

Display a date that is one year and two months after the hire date for the employees working in the department with the DEPARTMENT_ID 20.

```
SELECT hire_date,
       hire_date + TO_YMINTERVAL('01-02') AS
       HIRE_DATE_YMININTERVAL
FROM   employees
WHERE department_id = 20;
```

	HIRE_DATE	HIRE_DATE_YMININTERVAL
1	17-FEB-1996 00:00:00	17-APR-1997 00:00:00
2	17-AUG-1997 00:00:00	17-OCT-1998 00:00:00

The TO_YMINTERVAL function converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to an INTERVAL YEAR TO MONTH data type. The INTERVAL YEAR TO MONTH data type stores a period of time using the YEAR and MONTH datetime fields. The format of INTERVAL YEAR TO MONTH is as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

where year_precision is the number of digits in the YEAR datetime field. The default value of year_precision is 2.

The syntax of the TO_YMINTERVAL function is:

```
TO_YMINTERVAL (char)
```

where char is the character string to be converted.

The example in the slide calculates a date that is one year and two months after the hire date for the employees working in the department 20 of the EMPLOYEES table.

TO_DSINTERVAL

Display a date that is 100 days and 10 hours after the hire date for all the employees.

```
SELECT last_name,
 TO_CHAR(hire_date, 'mm-dd-yy:hh:mi:ss') hire_date,
  TO_CHAR(hire_date +
   TO_DSINTERVAL('100 10:00:00'),
     'mm-dd-yy:hh:mi:ss') hiredate2
FROM employees;
```

	LAST_NAME	HIRE_DATE	HIREDATE2
1	OConnell	06-21-99:12:00:00	09-29-99:10:00:00
2	Grant	01-13-00:12:00:00	04-22-00:10:00:00
3	Whalen	09-17-87:12:00:00	12-26-87:10:00:00
4	Hartstein	02-17-96:12:00:00	05-27-96:10:00:00
5	Fay	08-17-97:12:00:00	11-25-97:10:00:00
6	Mavris	06-07-94:12:00:00	09-15-94:10:00:00
7	Baer	06-07-94:12:00:00	09-15-94:10:00:00
8	Higgins	06-07-94:12:00:00	09-15-94:10:00:00

...

TO_DSINTERVAL converts a character string of the CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to an INTERVAL DAY TO SECOND data type.

In the example in the slide, the date 100 days and 10 hours after the hire date is obtained.

Daylight Saving Time (DST)

- Start of Daylight Saving:
  - Time jumps from 01:59:59 AM to 03:00:00 AM.
  - Values from 02:00:00 AM to 02:59:59 AM are not valid.
- End of Daylight Saving:
  - Time jumps from 02:00:00 AM to 01:00:01 AM.
  - Values from 01:00:01 AM to 02:00:00 AM are ambiguous because they are visited twice.

Most western nations advance the clock ahead one hour during the summer months. This period is called daylight saving time. Daylight saving time lasts from the second Sunday in March to the first Sunday in November in the most of the United States, Mexico, and Canada. The nations of the European Union observe daylight saving time, but they call it the summer time period. Europe's summer time period begins a week earlier than its North American counterpart, but ends at the same time.

The Oracle database automatically determines, for any given time zone region, whether daylight saving time is in effect and returns local time values accordingly. The datetime value is sufficient for the Oracle database to determine whether daylight saving time is in effect for a given region in all cases except boundary cases. A boundary case occurs during the period when daylight saving time goes into or out of effect. For example, in the US/Eastern region, when daylight saving time goes into effect, the time changes from 01:59:59 AM to 03:00:00 AM. The one-hour interval between 02:00:00 AM and 02:59:59 AM. does not exist. When daylight saving time goes out of effect, the time changes from 02:00:00 AM back to 01:00:01 AM, and the one-hour interval between 01:00:01 AM and 02:00:00 AM is repeated.

ERROR_ON_OVERLAP_TIME

The ERROR_ON_OVERLAP_TIME is a session parameter to notify the system to issue an error when it encounters a datetime that occurs in the overlapped period and no time zone abbreviation was specified to distinguish the period.

For example, daylight saving time ends on October 31, at 02:00:01 AM. The overlapped periods are:

- 10/31/2004 01:00:01 AM to 10/31/2004 02:00:00 AM (EDT)
- 10/31/2004 01:00:01 AM to 10/31/2004 02:00:00 AM (EST)

If you input a datetime string that occurs in one of these two periods, you need to specify the time zone abbreviation (for example, EDT or EST) in the input string for the system to determine the period. Without this time zone abbreviation, the system does the following:

If the ERROR_ON_OVERLAP_TIME parameter is FALSE, it assumes that the input time is standard time (for example, EST). Otherwise, an error is raised.

Quiz

The TIME_ZONE session parameter may be set to:

a. A relative offset
b. Database time zone
c. OS local time zone
d. A named region

Answer: b, c, d

Summary

In this lesson, you should have learned how to use the following functions:

- `CURRENT_DATE`
- `CURRENT_TIMESTAMP`
- `LOCALTIMESTAMP`
- `DBTIMEZONE`
- `SESSIONTIMEZONE`
- `EXTRACT`
- `TZ_OFFSET`
- `FROM_TZ`
- `TO_TIMESTAMP`
- `TO_YMINTERVAL`
- `TO_DSINTERVAL`

This lesson addressed some of the datetime functions available in the Oracle database.

Practice 6: Overview

This practice covers using the datetime functions.

In this practice, you display time zone offsets, `CURRENT_DATE`, `CURRENT_TIMESTAMP`, and `LOCALTIMESTAMP`. You also set time zones and use the `EXTRACT` function.

7

Retrieving Data by Using Subqueries

Objectives

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

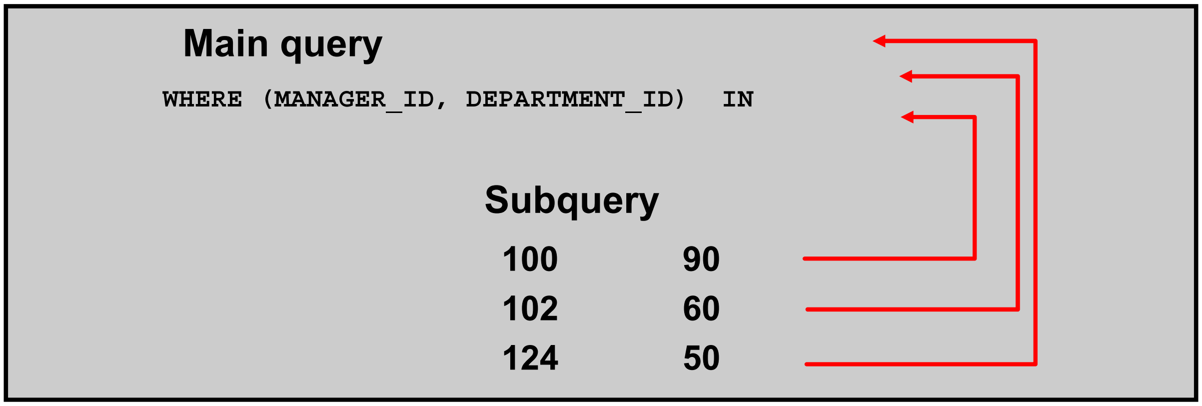
- Write a multiple-column subquery
- Use scalar subqueries in SQL
- Solve problems with correlated subqueries
- Update and delete rows by using correlated subqueries
- Use the `EXISTS` and `NOT EXISTS` operators
- Use the `WITH` clause

In this lesson, you learn how to write multiple-column subqueries and subqueries in the `FROM` clause of a `SELECT` statement. You also learn how to solve problems by using scalar, correlated subqueries and the `WITH` clause.

Lesson Agenda

- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Multiple-Column Subqueries



Each row of the main query is compared to values from a multiple-row and multiple-column subquery.

So far, you have written single-row subqueries and multiple-row subqueries where only one column is returned by the inner `SELECT` statement and this is used to evaluate the expression in the parent `SELECT` statement. If you want to compare two or more columns, you must write a compound `WHERE` clause using logical operators. Using multiple-column subqueries, you can combine duplicate `WHERE` conditions into a single `WHERE` clause.

Syntax

```
SELECT      column, column, ...
FROM  table
WHERE (column, column, ...) IN
                 (SELECT column, column, ...
                                     FROM    table
                                     WHERE   condition);
```

The graphic in the slide illustrates that the values of `MANAGER_ID` and `DEPARTMENT_ID` from the main query are being compared with the `MANAGER_ID` and `DEPARTMENT_ID` values retrieved by the subquery. Because the number of columns that are being compared is more than one, the example qualifies as a multiple-column subquery.

Note: Before you run the examples in the next few slides, you need to create the `empl_demo` table and populate data into it by using the `lab_07_insert_empdata.sql` file.

Column Comparisons

Multiple-column comparisons involving subqueries can be:

- Nonpairwise comparisons
- Pairwise comparisons

ORACLE

Pairwise Versus Nonpairwise Comparisons

Multiple-column comparisons involving subqueries can be nonpairwise comparisons or pairwise comparisons. If you consider the example "Display the details of the employees who work in the same department, and have the same manager, as 'Daniel'? ," you get the correct result with the following statement:

```
SELECT first_name, last_name, manager_id, department_id
FROM empl_demo
WHERE manager_id IN (SELECT manager_id
                     FROM empl_demo
                     WHERE first_name = 'Daniel')
AND department_id IN (SELECT department_id
                      FROM empl_demo
                      WHERE first_name = 'Daniel');
```

There is only one "Daniel" in the `EMPL_DEMO` table (Daniel Faviet, who is managed by employee 108 and works in department 100). However, if the subqueries return more than one row, the result might not be correct. For example, if you run the same query but substitute "John" for "Daniel," you get an incorrect result. This is because the combination of `department_id` and `manager_id` is important. To get the correct result for this query, you need a pairwise comparison.

Pairwise Comparison Subquery

Display the details of the employees who are managed by the same manager and work in the same department as employees with the first name of "John."

```
SELECT  employee_id, manager_id, department_id
FROM    empl_demo
WHERE  (manager_id, department_id) IN
                      (SELECT manager_id, department_id
                       FROM empl_demo
                       WHERE first_name = 'John')
AND first_name <> 'John';
```


The example in the slide compares the combination of values in the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPL_DEMO table with the values in the MANAGER_ID column and the DEPARTMENT_ID column for the employees with the FIRST_NAME of "John." First, the subquery to retrieve the MANAGER_ID and DEPARTMENT_ID values for the employees with the FIRST_NAME of "John" is executed. This subquery returns the following:

	MANAGER_ID	DEPARTMENT_ID
1	108	100
2	123	50
3	100	80

These values are compared with the MANAGER_ID column and the DEPARTMENT_ID column of each row in the EMPL_DEMO table. If the combination matches, the row is displayed. In the output, the records of the employees with the FIRST_NAME of "John" will not be displayed. The following is the output of the query in the slide:

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	113	108	100
2	112	108	100
3	111	108	100
4	109	108	100
5	195	123	50
6	194	123	50
7	193	123	50
8	192	123	50
9	140	123	50
10	138	123	50
11	137	123	50
12	149	100	80
13	148	100	80
14	147	100	80
15	146	100	80

Nonpairwise Comparison Subquery

Display the details of the employees who are managed by the same manager as the employees with the first name of “John” and work in the same department as the employees with the first name of “John.”

```
SELECT  employee_id, manager_id, department_id
FROM    empl_demo
WHERE   manager_id IN
                  (SELECT manager_id
                   FROM empl_demo
                   WHERE first_name = 'John')
AND department_id IN
                  (SELECT department_id
                   FROM empl_demo
                   WHERE first_name = 'John')
AND first_name <> 'John';
```


The example shows a nonpairwise comparison of the columns. First, the subquery to retrieve the `MANAGER_ID` values for the employees with the `FIRST_NAME` of “John” is executed. Similarly, the second subquery to retrieve the `DEPARTMENT_ID` values for the employees with the `FIRST_NAME` of “John” is executed. The retrieved values of the `MANAGER_ID` and `DEPARTMENT_ID` columns are compared with the `MANAGER_ID` and `DEPARTMENT_ID` columns for each row in the `EMPL_DEMO` table. If the `MANAGER_ID` column of the row in the `EMPL_DEMO` table matches with any of the values of `MANAGER_ID` retrieved by the inner subquery, and if the `DEPARTMENT_ID` column of the row in the `EMPL_DEMO` table matches with any of the values of `DEPARTMENT_ID` retrieved by the second subquery, the record is displayed.

The following is the output of the query in the previous slide:

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	109	108	100
2	111	108	100
3	112	108	100
4	113	108	100
5	120	100	50
6	121	100	50
7	122	100	50
8	123	100	50
9	124	100	50
10	137	123	50
11	138	123	50
12	140	123	50
13	192	123	50
14	193	123	50
15	194	123	50
16	195	123	50
17	146	100	80
18	147	100	80
19	148	100	80
20	149	100	80

This query retrieves additional rows than the pairwise comparison (those with the combination of `manager_id=100` and `department_id=50` or `80`, although no employee named “John” has such a combination).

Lesson Agenda

- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Scalar Subquery Expressions

- A scalar subquery expression is a subquery that returns exactly one column value from one row.
- Scalar subqueries can be used in:
  - The condition and expression part of `DECODE` and `CASE`
  - All clauses of `SELECT` except `GROUP BY`
  - The `SET` clause and `WHERE` clause of an `UPDATE` statement

A subquery that returns exactly one column value from one row is also referred to as a scalar subquery. Multiple-column subqueries that are written to compare two or more columns, using a compound `WHERE` clause and logical operators, do not qualify as scalar subqueries.

The value of the scalar subquery expression is the value of the select list item of the subquery. If the subquery returns 0 rows, the value of the scalar subquery expression is `NULL`. If the subquery returns more than one row, the Oracle Server returns an error. The Oracle Server has always supported the usage of a scalar subquery in a `SELECT` statement. You can use scalar subqueries in:

- The condition and expression part of `DECODE` and `CASE`
- All clauses of `SELECT` except `GROUP BY`
- The `SET` clause and `WHERE` clause of an `UPDATE` statement

However, scalar subqueries are not valid expressions in the following places:

- As default values for columns and hash expressions for clusters
- In the `RETURNING` clause of data manipulation language (DML) statements
- As the basis of a function-based index
- In `GROUP BY` clauses, `CHECK` constraints, and `WHEN` conditions
- In `CONNECT BY` clauses
- In statements that are unrelated to queries, such as `CREATE PROFILE`

Scalar Subqueries: Examples

- Scalar subqueries in `CASE` expressions:

```
SELECT employee_id, last_name,
       (CASE
        WHEN department_id =          20
             (SELECT department_id
              FROM departments
                  WHERE location_id = 1800)
          THEN 'Canada' ELSE 'USA' END) location
FROM   employees;
```

- Scalar subqueries in the `ORDER BY` clause:

```
SELECT    employee_id, last_name
FROM      employees e
ORDER BY (SELECT department_name
            FROM departments d
             WHERE e.department_id = d.department_id);
```

ORACLE

The first example in the slide demonstrates that scalar subqueries can be used in `CASE` expressions. The inner query returns the value 20, which is the department ID of the department whose location ID is 1800. The `CASE` expression in the outer query uses the result of the inner query to display the employee ID, last names, and a value of Canada or USA, depending on whether the department ID of the record retrieved by the outer query is 20 or not.

The following is the result of the first example in the slide 12 12

...

	EMPLOYEE_ID	LAST_NAME	LOCATION
1	198	OConnell	USA
2	199	Grant	USA
3	200	Whalen	USA
4	201	Hartstein	Canada
5	202	Fay	Canada
6	203	Mavris	USA

The second example in the slide demonstrates that scalar subqueries can be used in the ORDER BY clause. The example orders the output based on the DEPARTMENT_NAME by matching the DEPARTMENT_ID from the EMPLOYEES table with the DEPARTMENT_ID from the DEPARTMENTS table. This comparison is done in a scalar subquery in the ORDER BY clause. The following is the result of the second example:

	EMPLOYEE_ID	LAST_NAME
1	205	Higgins
2	206	Gietz
3	200	Whalen
4	100	King
5	101	Kochhar
6	102	De Haan
7	112	Urman
8	108	Greenberg
9	109	Faviet

...

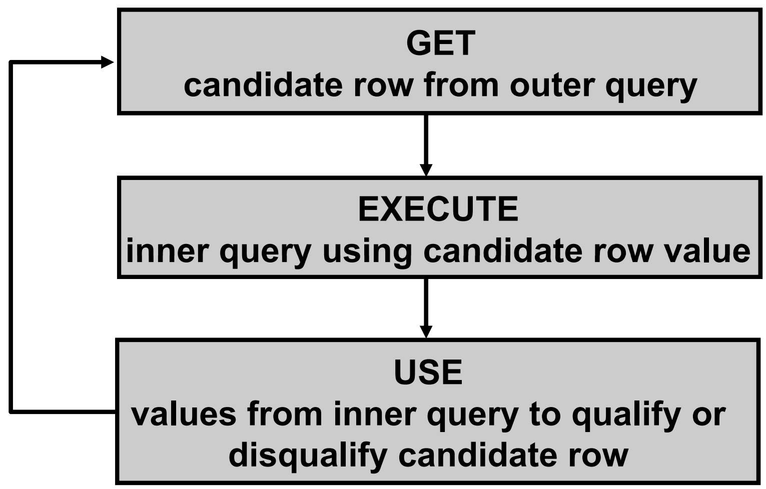
The second example uses a correlated subquery. In a correlated subquery, the subquery references a column from a table referred to in the parent statement. Correlated subqueries are explained later in this lesson.

Lesson Agenda

- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Correlated Subqueries

Correlated subqueries are used for row-by-row processing. Each subquery is executed once for every row of the outer query.



The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a `SELECT`, `UPDATE`, or `DELETE` statement.

Nested Subqueries Versus Correlated Subqueries

With a normal nested subquery, the inner `SELECT` query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. That is, the inner query is driven by the outer query.

Nested Subquery Execution

- The inner query executes first and finds a value.
- The outer query executes once, using the value from the inner query.

Correlated Subquery Execution

- Get a candidate row (fetched by the outer query).
- Execute the inner query by using the value of the candidate row.
- Use the values resulting from the inner query to qualify or disqualify the candidate.
- Repeat until no candidate row remains.

Correlated Subqueries

The subquery references a column from a table in the parent query.

```
SELECT column1, column2, ...
FROM   table1   Outer_table
WHERE  column1 operator
                      (SELECT  column1, column2
                       FROM    table2
                       WHERE   expr1 =
                               Outer_table.expr2);
```


A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result or set of results for each candidate row considered by the main query. That is, you use a correlated subquery to answer a multipart question whose answer depends on the value in each row processed by the parent statement.

The Oracle Server performs a correlated subquery when the subquery references a column from a table in the parent query.

Note: You can use the `ANY` and `ALL` operators in a correlated subquery.

Using Correlated Subqueries

Find all employees who earn more than the average salary in their department.

```
SELECT last_name, salary, department_id
FROM   employees outer_table
WHERE  salary >
                (SELECT AVG(salary)
                 FROM   employees inner_table
                 WHERE inner_table.department_id =
                 outer_table.department_id);
```

Each time a row from the outer query is processed, the inner query is evaluated.

The example in the slide determines which employees earn more than the average salary of their department. In this case, the correlated subquery specifically computes the average salary for each department.

Because both the outer query and inner query use the `EMPLOYEES` table in the `FROM` clause, an alias is given to `EMPLOYEES` in the outer `SELECT` statement for clarity. The alias makes the entire `SELECT` statement more readable. Without the alias, the query would not work properly because the inner statement would not be able to distinguish the inner table column from the outer table column.

Using Correlated Subqueries

Display details of those employees who have changed jobs at least twice.

```
SELECT e.employee_id, last_name,e.job_id
FROM   employees e
WHERE  2 <= (SELECT COUNT(*)
             FROM   job_history
             WHERE  employee_id = e.employee_id);
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID
1	200	Whalen	AD_ASST
2	101	Kochhar	AD_VP
3	176	Taylor	SA_REP

ORACLE

The example in the slide displays the details of those employees who have changed jobs at least twice. The Oracle Server evaluates a correlated subquery as follows:

1. Select a row from the table specified in the outer query. This will be the current candidate row.
2. Store the value of the column referenced in the subquery from this candidate row. (In the example in the slide, the column referenced in the subquery is `E.EMPLOYEE_ID`.)
3. Perform the subquery with its condition referencing the value from the outer query's candidate row. (In the example in the slide, the `COUNT(*)` group function is evaluated based on the value of the `E.EMPLOYEE_ID` column obtained in step 2.)
4. Evaluate the `WHERE` clause of the outer query on the basis of results of the subquery performed in step 3. This determines whether the candidate row is selected for output. (In the example, the number of times an employee has changed jobs, evaluated by the subquery, is compared with 2 in the `WHERE` clause of the outer query. If the condition is satisfied, that employee record is displayed.)
5. Repeat the procedure for the next candidate row of the table, and so on, until all the rows in the table have been processed.

The correlation is established by using an element from the outer query in the subquery. In this example, you compare `EMPLOYEE_ID` from the table in the subquery with `EMPLOYEE_ID` from the table in the outer query.

Lesson Agenda

- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

Using the `EXISTS` Operator

- The `EXISTS` operator tests for existence of rows in the results set of the subquery.
- If a subquery row value is found:
  - The search does not continue in the inner query
  - The condition is flagged `TRUE`
- If a subquery row value is not found:
  - The condition is flagged `FALSE`
  - The search continues in the inner query

With nesting `SELECT` statements, all logical operators are valid. In addition, you can use the `EXISTS` operator. This operator is frequently used with correlated subqueries to test whether a value retrieved by the outer query exists in the results set of the values retrieved by the inner query. If the subquery returns at least one row, the operator returns `TRUE`. If the value does not exist, it returns `FALSE`. Accordingly, `NOT EXISTS` tests whether a value retrieved by the outer query is not a part of the results set of the values retrieved by the inner query.

Using the EXISTS Operator

```
SELECT employee_id, last_name, job_id, department_id
FROM   employees outer
WHERE  EXISTS ( SELECT 'X'
                 FROM   employees
                 WHERE  manager_id =
                        outer.employee_id);
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	201	Hartstein	MK_MAN	20
2	205	Higgins	AC_MGR	110
3	100	King	AD_PRES	90
4	101	Kochhar	AD_VP	90
5	102	De Haan	AD_VP	90
6	103	Hunold	IT_PROG	60
7	108	Greenberg	FI_MGR	100
8	114	Raphaely	PU_MAN	30

The EXISTS operator ensures that the search in the inner query does not continue when at least one match is found for the manager and employee number by the condition:

```
WHERE manager_id = outer.employee_id.
```

Note that the inner SELECT query does not need to return a specific value, so a constant can be selected.

Find All Departments That Do Not Have Any Employees

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT 'X'
                  FROM   employees
                  WHERE  department_id = d.department_id);
```

	DEPARTMENT_ID	DEPARTMENT_NAME
1	120	Treasury
2	130	Corporate Tax
3	140	Control And Credit
4	150	Shareholder Services
5	160	Benefits
6	170	Manufacturing
7	180	Construction

All Rows Fetched: 16

...

ORACLE

Using the `NOT EXISTS` Operator

Alternative Solution

A `NOT IN` construct can be used as an alternative for a `NOT EXISTS` operator, as shown in the following example:

```
SELECT department_id, department_name
FROM   departments
WHERE  department_id NOT IN (SELECT department_id
                             FROM   employees);
```

All Rows Fetched: 0

However, `NOT IN` evaluates to `FALSE` if any member of the set is a `NULL` value. Therefore, your query will not return any rows even if there are rows in the `departments` table that satisfy the `WHERE` condition.

Correlated `UPDATE`

Use a correlated subquery to update rows in one table based on rows from another table.

```
UPDATE table1 alias1
SET    column = (SELECT expression
                 FROM   table2 alias2
                 WHERE  alias1.column =
                        alias2.column);
```


In the case of the `UPDATE` statement, you can use a correlated subquery to update rows in one table based on rows from another table.

Using Correlated UPDATE

- Denormalize the EMPL6 table by adding a column to store the department name.
- Populate the table by using a correlated update.

```
ALTER TABLE empl6
ADD(department_name VARCHAR2(25));
```

```
UPDATE empl6 e
SET    department_name =
              (SELECT department_name
               FROM   departments d
               WHERE  e.department_id = d.department_id);
```


The example in the slide denormalizes the EMPL6 table by adding a column to store the department name and then populates the table by using a correlated update.

Following is another example for a correlated update.

Problem Statement

The REWARDS table has a list of employees who have exceeded expectations in their performance. Use a correlated subquery to update rows in the EMPL6 table based on rows from the REWARDS table:

```
UPDATE empl6
SET    salary = (SELECT empl6.salary + rewards.pay_raise
                 FROM   rewards
                 WHERE  employee_id  =
                        empl6.employee_id
                 AND  payraise_date =
                      (SELECT MAX(payraise_date)
                      FROM   rewards
                      WHERE  employee_id = empl6.employee_id))
WHERE  empl6.employee_id
IN     (SELECT employee_id FROM rewards);
```

This example uses the REWARDS table. The REWARDS table has the following columns: EMPLOYEE_ID, PAY_RAISE, and PAYRAISE_DATE. Every time an employee gets a pay raise, a record with details such as the employee ID, the amount of the pay raise, and the date of receipt of the pay raise is inserted into the REWARDS table. The REWARDS table can contain more than one record for an employee. The PAYRAISE _DATE column is used to identify the most recent pay raise received by an employee.

In the example, the SALARY column in the EMPL6 table is updated to reflect the latest pay raise received by the employee. This is done by adding the current salary of the employee with the corresponding pay raise from the REWARDS table.

Correlated DELETE

Use a correlated subquery to delete rows in one table based on rows from another table.

```
DELETE FROM table1 alias1
WHERE  column operator
              (SELECT expression
               FROM   table2 alias2
               WHERE  alias1.column = alias2.column);
```


In the case of a DELETE statement, you can use a correlated subquery to delete only those rows that also exist in another table. If you decide that you will maintain only the last four job history records in the JOB_HISTORY table, when an employee transfers to a fifth job, you delete the oldest JOB_HISTORY row by looking up the JOB_HISTORY table for the MIN(START_DATE) for the employee. The following code illustrates how the preceding operation can be performed using a correlated DELETE:

```
DELETE FROM job_history JH
WHERE employee_id =
      (SELECT employee_id
       FROM employees E
       WHERE JH.employee_id = E.employee_id
       AND START_DATE =
             (SELECT MIN(start_date)
              FROM job_history JH
              WHERE JH.employee_id = E.employee_id)
              AND 5 >  (SELECT COUNT(*)
                        FROM job_history JH
                        WHERE JH.employee_id = E.employee_id
                        GROUP BY EMPLOYEE_ID
                        HAVING COUNT(*) >= 4));
```

Using Correlated DELETE

Use a correlated subquery to delete only those rows from the EMPL6 table that also exist in the EMP_HISTORY table.

```
DELETE FROM empl6 E
WHERE employee_id =
           (SELECT employee_id
            FROM   emp_history
            WHERE  employee_id = E.employee_id);
```


Example

Two tables are used in this example. They are:

- The EMPL6 table, which provides details of all the current employees
- The EMP_HISTORY table, which provides details of previous employees

EMP_HISTORY contains data regarding previous employees, so it would be erroneous if the same employee's record existed in both the EMPL6 and EMP_HISTORY tables. You can delete such erroneous records by using the correlated subquery shown in the slide.

Lesson Agenda

- Writing a multiple-column subquery
- Using scalar subqueries in SQL
- Solving problems with correlated subqueries
- Using the `EXISTS` and `NOT EXISTS` operators
- Using the `WITH` clause

`WITH` Clause

- Using the `WITH` clause, you can use the same query block in a `SELECT` statement when it occurs more than once within a complex query.
- The `WITH` clause retrieves the results of a query block and stores it in the user's temporary tablespace.
- The `WITH` clause may improve performance.

Using the `WITH` clause, you can define a query block before using it in a query. The `WITH` clause (formally known as `subquery_factoring_clause`) enables you to reuse the same query block in a `SELECT` statement when it occurs more than once within a complex query. This is particularly useful when a query has many references to the same query block and there are joins and aggregations.

Using the `WITH` clause, you can reuse the same query when it is costly to evaluate the query block and it occurs more than once within a complex query. Using the `WITH` clause, the Oracle Server retrieves the results of a query block and stores it in the user's temporary tablespace. This can improve performance.

`WITH` Clause Benefits

- Makes the query easy to read
- Evaluates a clause only once, even if it appears multiple times in the query
- In most cases, may improve performance for large queries

WITH Clause: Example

Using the `WITH` clause, write a query to display the department name and total salaries for those departments whose total salary is greater than the average salary across departments.

The problem in the slide would require the following intermediate calculations:

1. Calculate the total salary for every department, and store the result using a `WITH` clause.
2. Calculate the average salary across departments, and store the result using a `WITH` clause.
3. Compare the total salary calculated in the first step with the average salary calculated in the second step. If the total salary for a particular department is greater than the average salary across departments, display the department name and the total salary for that department.

The solution for this problem is provided on the next page.

WITH Clause: Example

```
WITH
dept_costs  AS (
   SELECT d.department_name, SUM(e.salary) AS dept_total
   FROM   employees e JOIN departments d
   ON     e.department_id = d.department_id
   GROUP BY d.department_name),
avg_cost    AS (
   SELECT SUM(dept_total)/COUNT(*) AS dept_avg
   FROM   dept_costs)
SELECT *
FROM   dept_costs
WHERE  dept_total >
        (SELECT dept_avg
         FROM avg_cost)
ORDER BY department_name;
```


The SQL code in the slide is an example of a situation in which you can improve performance and write SQL more simply by using the WITH clause. The query creates the query names DEPT_COSTS and AVG_COST and then uses them in the body of the main query. Internally, the WITH clause is resolved either as an inline view or a temporary table. The optimizer chooses the appropriate resolution depending on the cost or benefit of temporarily storing the results of the WITH clause.

The output generated by the SQL code in the slide is as follows:

	DEPARTMENT_NAME	DEPT_TOTAL
1	Sales	304500
2	Shipping	156400

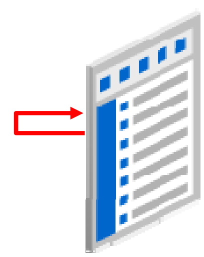
WITH Clause Usage Notes

- It is used only with SELECT statements.
- A query name is visible to all WITH element query blocks (including their subquery blocks) defined after it and the main query block itself (including its subquery blocks).
- When the query name is the same as an existing table name, the parser searches from the inside out, and the query block name takes precedence over the table name.
- The WITH clause can hold more than one query. Each query is then separated by a comma.

Recursive `WITH` Clause

The Recursive `WITH` clause:

- Enables formulation of recursive queries
- Creates a query with a name, called the Recursive `WITH` element name
- Contains two types of query block members: an anchor and a recursive
- Is ANSI-compatible



In Oracle Database 11*g* Release 2, the `WITH` clause has been extended to enable formulation of recursive queries.

Recursive `WITH` defines a recursive query with a name, the *Recursive `WITH` element name*. The Recursive `WITH` element definition must contain at least two query blocks: an anchor member and a recursive member. There can be multiple anchor members, but there can be only a single recursive member.

The recursive `WITH` clause, Oracle Database 11*g* Release 2 *partially* complies with the American National Standards Institute (ANSI). Recursive `WITH` can be used to query hierarchical data such as organization charts.

Recursive `WITH` Clause: Example

FLIGHTS Table

	SOURCE	DESTIN	FLIGHT_TIME
1	San Jose	Los Angeles	1.3
2	New York	Boston	1.1
3	Los Angeles	New York	5.8

1

```
WITH Reachable_From (Source, Destin, TotalFlightTime) AS
(
    SELECT Source, Destin, Flight_time
    FROM Flights
  UNION ALL
    SELECT incoming.Source, outgoing.Destin,
           incoming.TotalFlightTime+outgoing.Flight_time
    FROM Reachable_From incoming, Flights outgoing
    WHERE incoming.Destin = outgoing.Source
)
SELECT Source, Destin, TotalFlightTime
FROM Reachable_From;
```

2

3

	SOURCE	DESTIN	TOTALFLIGHTTIME
1	San Jose	Los Angeles	1.3
2	New York	Boston	1.1
3	Los Angeles	New York	5.8
4	San Jose	New York	7.1
5	Los Angeles	Boston	6.9
6	San Jose	Boston	8.2

ORACLE

Example 1 in the slide displays records from a `FLIGHTS` table describing flights between two cities. This example is specific to 11*g* R2 version of Oracle Database.

Using the query in example 2, you query the `FLIGHTS` table to display the total flight time between any source and destination. The `WITH` clause in the query, which is named `Reachable From`, has a `UNION ALL` query with two branches. The first branch is the *anchor* branch, which selects all the rows from the `Flights` table. The second branch is the recursive branch. It joins the contents of `Reachable From` to the `Flights` table to find other cities that can be reached, and adds these to the content of `Reachable From`. The operation will finish when no more rows are found by the recursive branch.

Example 3 displays the result of the query that selects everything from the `WITH` clause element `Reachable From`.

For details, see:

- *Oracle Database SQL Language Reference 11g Release 2.0*
- *Oracle Database Data Warehousing Guide 11g Release 2.0*

Quiz

With a correlated subquery, the inner `SELECT` statement drives the outer `SELECT` statement.

a. True
b. False

Answer: b

Summary

In this lesson, you should have learned that:

- A multiple-column subquery returns more than one column
- Multiple-column comparisons can be pairwise or nonpairwise
- A multiple-column subquery can also be used in the `FROM` clause of a `SELECT` statement

You can use multiple-column subqueries to combine multiple `WHERE` conditions in a single `WHERE` clause. Column comparisons in a multiple-column subquery can be pairwise comparisons or nonpairwise comparisons.

You can use a subquery to define a table to be operated on by a containing query.

Scalar subqueries can be used in:

- The condition and expression part of `DECODE` and `CASE`
- All clauses of `SELECT` except `GROUP BY`
- A `SET` clause and `WHERE` clause of the `UPDATE` statement

Summary

- Correlated subqueries are useful whenever a subquery must return a different result for each candidate row.
- The `EXISTS` operator is a Boolean operator that tests the presence of a value.
- Correlated subqueries can be used with `SELECT`, `UPDATE`, and `DELETE` statements.
- You can use the `WITH` clause to use the same query block in a `SELECT` statement when it occurs more than once.

The Oracle Server performs a correlated subquery when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a `SELECT`, `UPDATE`, or `DELETE` statement. Using the `WITH` clause, you can reuse the same query when it is costly to re-evaluate the query block and it occurs more than once within a complex query.

Practice 7: Overview

This practice covers the following topics:

- Creating multiple-column subqueries
- Writing correlated subqueries
- Using the `EXISTS` operator
- Using scalar subqueries
- Using the `WITH` clause

In this practice, you write multiple-column subqueries, and correlated and scalar subqueries. You also solve problems by writing the `WITH` clause.

8

Regular Expression Support

Objectives

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

- List the benefits of using regular expressions
- Use regular expressions to search for, match, and replace strings

In this lesson, you learn to use the regular expression support feature. Regular expression support is available in both SQL and PL/SQL.

Lesson Agenda

- Introduction to regular expressions
- Using metacharacters with regular expressions
- Using the regular expressions functions:
  - `REGEXP_LIKE`
  - `REGEXP_REPLACE`
  - `REGEXP_INSTR`
  - `REGEXP_SUBSTR`
- Accessing subexpressions
- Using the `REGEXP_COUNT` function
- Regular expressions and check constraints

What Are Regular Expressions?

- You use regular expressions to search for (and manipulate) simple and complex patterns in string data by using standard syntax conventions.
- You use a set of SQL functions and conditions to search for and manipulate strings in SQL and PL/SQL.
- You specify a regular expression by using:
  - Metacharacters, which are operators that specify the search algorithms
  - Literals, which are the characters for which you are searching

Oracle Database provides support for regular expressions. The implementation complies with the Portable Operating System for UNIX (POSIX) standard, controlled by the Institute of Electrical and Electronics Engineers (IEEE), for ASCII data-matching semantics and syntax. Oracle's multilingual capabilities extend the matching capabilities of the operators beyond the POSIX standard. Regular expressions are a method of describing both simple and complex patterns for searching and manipulating.

String manipulation and searching contribute to a large percentage of the logic within a web-based application. Usage ranges from the simple, such as finding the words "San Francisco" in a specified text, to the complex task of extracting all URLs from the text and the more complex task of finding all words whose every second character is a vowel.

When coupled with native SQL, the use of regular expressions allows for very powerful search and manipulation operations on any data stored in an Oracle database. You can use this feature to easily solve problems that would otherwise involve complex programming.

Benefits of Using Regular Expressions

Regular expressions enable you to implement complex match logic in the database with the following benefits:

- By centralizing match logic in Oracle Database, you avoid intensive string processing of SQL result sets by middle-tier applications.
- Using server-side regular expressions to enforce constraints, you eliminate the need to code data validation logic on the client.
- The built-in SQL and PL/SQL regular expression functions and conditions make string manipulations more powerful and easier than in previous releases of Oracle Database 11*g*.

Regular expressions are a powerful text-processing component of programming languages such as PERL and Java. For example, a PERL script can process each HTML file in a directory, read its contents into a scalar variable as a single string, and then use regular expressions to search for URLs in the string. One reason for many developers writing in PERL is that it has a robust pattern-matching functionality. Oracle's support of regular expressions enables developers to implement complex match logic in the database. Regular expressions were introduced in Oracle Database 10*g*.

Using the Regular Expressions Functions and Conditions in SQL and PL/SQL

Function or Condition Name	Description
REGEXP_LIKE	Is similar to the LIKE operator, but performs regular expression matching instead of simple pattern matching (condition)
REGEXP_REPLACE	Searches for a regular expression pattern and replaces it with a replacement string
REGEXP_INSTR	Searches a string for a regular expression pattern and returns the position where the match is found
REGEXP_SUBSTR	Searches for a regular expression pattern within a given string and extracts the matched substring
REGEXP_COUNT	Returns the number of times a pattern match is found in an input sting

Oracle Database provides a set of SQL functions that you use to search and manipulate strings by using regular expressions. You use these functions on a text literal, bind variable, or any column that holds character data such as CHAR, NCHAR, CLOB, NCLOB, NVARCHAR2, and VARCHAR2 (but not LONG). A regular expression must be enclosed within single quotation marks. This ensures that the entire expression is interpreted by the SQL function and can improve the readability of your code.

- REGEXP_LIKE: This condition searches a character column for a pattern. Use this condition in the WHERE clause of a query to return rows matching the regular expression that you specify.
- REGEXP_REPLACE: This function searches for a pattern in a character column and replaces each occurrence of that pattern with the pattern that you specify.
- REGEXP_INSTR: This function searches a string for a given occurrence of a regular expression pattern. You specify which occurrence you want to find and the start position to search from. This function returns an integer indicating the position in the string where the match is found.
- REGEXP_SUBSTR: This function returns the actual substring matching the regular expression pattern that you specify.
- REGEXP_COUNT: This function, introduced with 11*g* Release 2, returns the number of times a pattern match is found in the input string.

Lesson Agenda

- Introduction to regular expressions
- Using metacharacters with regular expressions
- Using the regular expressions functions:
  - `REGEXP_LIKE`
  - `REGEXP_REPLACE`
  - `REGEXP_INSTR`
  - `REGEXP_SUBSTR`
- Accessing subexpressions
- Using the `REGEXP_COUNT` function

What Are Metacharacters?

- Metacharacters are special characters that have a special meaning such as a wildcard, a repeating character, a nonmatching character, or a range of characters.
- You can use several predefined metacharacter symbols in the pattern matching.
- For example, the `^(f|ht)tps?:$` regular expression searches for the following from the beginning of the string:
  - The literals `f` or `ht`
  - The `t` literal
  - The `p` literal, optionally followed by the `s` literal
  - The colon "`:`" literal at the end of the string

ORACLE

The regular expression in the slide matches the `http:`, `https:`, `ftp:`, and `ftps:` strings.

Note: For a complete list of the regular expressions' metacharacters, see the *Oracle Database Advanced Application Developer's Guide for 10g or 11g.*

Using Metacharacters with Regular Expressions

Syntax	Description
`.`	Matches any character in the supported character set, except NULL
`+`	Matches one or more occurrences
`?`	Matches zero or one occurrence
`*`	Matches zero or more occurrences of the preceding subexpression
`{m}`	Matches exactly *m* occurrences of the preceding expression
`{m, }`	Matches at least *m* occurrences of the preceding subexpression
`{m,n}`	Matches at least *m*, but not more than *n*, occurrences of the preceding subexpression
`[…]`	Matches any single character in the list within the brackets
`|`	Matches one of the alternatives
`( ... )`	Treats the enclosed expression within the parentheses as a unit. The subexpression can be a string of literals or a complex expression containing operators.

Any character, " . " : `a.b` matches the strings `abb`, `acb`, and `adb`, but not `acc`.

One or more, " + " : `a+` matches the strings `a`, `aa`, and `aaa`, but does not match `bbb`.

Zero or one, " ? ": `ab?c` matches the strings `abc` and `ac`, but does not match `abbc`.

Zero or more, " * ": `ab*c` matches the strings `ac`, `abc`, and `abbc`, but does not match `abb`.

Exact count, " {m} ": `a{3}` matches the strings `aaa`, but does not match `aa`.

At least count, " {m,} ": `a{3,}` matches the strings `aaa` and `aaaa`, but not `aa`.

Between count, " {m,n} ": `a{3,5}` matches the strings `aaa`, `aaaa`, and `aaaaa`, but not `aa`.

Matching character list, " […] ": `[abc]` matches the first character in the strings `all`, `bill`, and `cold`, but does not match any characters in `doll`.

Or, " | ": `a|b` matches character `a` or character `b`.

Subexpression, " (…) ": `(abc)?def` matches the optional string `abc`, followed by `def`. The expression matches `abcdefghi` and `def`, but does not match `ghi`. The subexpression can be a string of literals or a complex expression containing operators.

Using Metacharacters with Regular Expressions

Syntax	Description
`^`	Matches the beginning of a string
`$`	Matches the end of a string
`\`	Treats the subsequent metacharacter in the expression as a literal
`\n`	Matches the *n*th (1–9) preceding subexpression of whatever is grouped within parentheses. The parentheses cause an expression to be remembered; a backreference refers to it.
`\d`	A digit character
`[:class:]`	Matches any character belonging to the specified POSIX character class
`[^:class:]`	Matches any single character *not* in the list within the brackets

Beginning/end of line anchor, " ^ " and "$": `^def` matches `def` in the string `defghi` but does not match `def` in `abcdef`. `def$` matches `def` in the string `abcdef` but does not match `def` in the string `defghi`.

Escape character " \ ": `\+` searches for a `+`. It matches the plus character in the string `abc+def`, but does not match `Abcdef`.

Backreference, " \n ": `(abc|def)xy\1` matches the strings `abcxyabc` and `defxydef`, but does not match `abcxydef` or `abcxy`. A backreference enables you to search for a repeated string without knowing the actual string ahead of time. For example, the expression `^(.*)\1$` matches a line consisting of two adjacent instances of the same string.

Digit character, "\d": The expression `^\[\d{3}\] \d{3}-\d{4}$` matches [650] 555-1212 but does not match 650-555-1212.

Character class, " [:class:] ": `[[:upper:]]+` searches for one or more consecutive uppercase characters. This matches `DEF` in the string `abcDEFghi` but does not match the string `abcdefghi`.

Nonmatching character list (or class), " [^...] ": `[^abc]` matches the character `d` in the string `abcdef`, but not `a`, `b`, or `c`.

Lesson Agenda

- Introduction to regular expressions
- Using metacharacters with regular expressions
- Using the regular expressions functions:
  - `REGEXP_LIKE`
  - `REGEXP_REPLACE`
  - `REGEXP_INSTR`
  - `REGEXP_SUBSTR`
- Accessing subexpressions
- Using the `REGEXP_COUNT` function

Regular Expressions Functions and Conditions: Syntax

```
REGEXP_LIKE   (source_char, pattern [,match_option]
```

```
REGEXP_INSTR  (source_char, pattern [, position
               [, occurrence [, return_option
               [, match_option [, subexpr]]]]])
```

```
REGEXP_SUBSTR (source_char, pattern [, position
               [, occurrence [, match_option
               [, subexpr]]]])
```

```
REGEXP_REPLACE(source_char, pattern [,replacestr
                [, position [, occurrence
                [, match_option]]]])
```

```
REGEXP_COUNT (source_char, pattern [, position

              [, occurrence [, match_option]]])
```


The syntax for the regular expressions functions and conditions is as follows:

- `source_char`: A character expression that serves as the search value
- `pattern`: A regular expression, a text literal
- `occurrence`: A positive integer indicating which occurrence of pattern in `source_char` Oracle Server should search for. The default is 1.
- `position`: A positive integer indicating the character of `source_char` where Oracle Server should begin the search. The default is 1.
- `return_option`:
  - 0: Returns the position of the first character of the occurrence (default)
  - 1: Returns the position of the character following the occurrence
- `Replacestr`: Character string replacing pattern
- `match_parameter`:
  - " `c` ": Uses case-sensitive matching (default)
  - " `i` ": Uses non-case-sensitive matching
  - " `n` ": Allows match-any-character operator
  - " `m` ": Treats source string as multiple lines
- `subexpr`: Fragment of pattern enclosed in parentheses. You learn more about subexpressions later in this lesson.

Performing a Basic Search by Using the REGEXP_LIKE Condition

```
REGEXP_LIKE(source_char, pattern [, match_parameter ])
```

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$');
```

	FIRST_NAME	LAST_NAME
1	Steven	King
2	Steven	Markle
3	Stephen	Stiles

REGEXP_LIKE is similar to the LIKE condition, except that REGEXP_LIKE performs regular-expression matching instead of the simple pattern matching performed by LIKE. This condition evaluates strings by using characters as defined by the input character set.

Example of REGEXP_LIKE

In this query, against the EMPLOYEES table, all employees with first names containing either Steven or Stephen are displayed. In the expression used '^Ste(v|ph)en$ ':

- ^ indicates the beginning of the expression
- $ indicates the end of the expression
- | indicates either/or

Replacing Patterns by Using the REGEXP_REPLACE Function

```
REGEXP_REPLACE(source_char, pattern [,replacestr
[, position [, occurrence [, match_option]]]])
```

```
SELECT last_name, REGEXP_REPLACE(phone_number, '\.','-')
   AS phone
FROM employees;
```

Original

	LAST_NAME	PHONE
1	OConnell	650.507.9833
2	Grant	650.507.9844
3	Whalen	515.123.4444
4	Hartstein	515.123.5555

Partial results

	LAST_NAME	PHONE
1	OConnell	650-507-9833
2	Grant	650-507-9844
3	Whalen	515-123-4444
4	Hartstein	515-123-5555

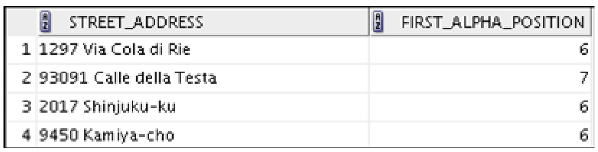
Using the `REGEXP_REPLACE` function, you reformat the phone number to replace the period (`.`) delimiter with a dash (`-`) delimiter. Here is an explanation of each of the elements used in the regular expression example:

- `phone_number` is the source column.
- `'\.'` is the search pattern.
  - Use single quotation marks (`' '`) to search for the literal character period (`.`).
  - Use a backslash (`\`) to search for a character that is normally treated as a metacharacter.
- `'-'` is the replace string.

Finding Patterns by Using the `REGEXP_INSTR` Function

```
REGEXP_INSTR  (source_char, pattern [, position [,
   occurrence [, return_option [, match_option]]]])
```

```
SELECT street_address,
REGEXP_INSTR(street_address,'[[:alpha:]]') AS
   First_Alpha_Position
FROM  locations;
```



	STREET_ADDRESS	FIRST_ALPHA_POSITION
1	1297 Via Cola di Rie	6
2	93091 Calle della Testa	7
3	2017 Shinjuku-ku	6
4	9450 Kamiya-cho	6

In this example, the `REGEXP_INSTR` function is used to search the street address to find the location of the first alphabetic character, regardless of whether it is in uppercase or lowercase. Note that `[:<class>:]` implies a character class and matches any character from within that class; `[:alpha:]` matches with any alphabetic character. The partial results are displayed.

In the expression used in the query `'[[:alpha:]]'`:

- `[` starts the expression
- `[:alpha:]` indicates alphabetic character class
- `]` ends the expression

Note: The POSIX character class operator enables you to search for an expression within a character list that is a member of a specific POSIX character class. You can use this operator to search for specific formatting, such as uppercase characters, or you can search for special characters such as digits or punctuation characters. The full set of POSIX character classes is supported. Use the syntax `[:class:]`, where `class` is the name of the POSIX character class to search for. The following regular expression searches for one or more consecutive uppercase characters : `[[:upper:]]+` .

Extracting Substrings by Using the `REGEXP_SUBSTR` Function

```
REGEXP_SUBSTR (source_char, pattern [, position
               [, occurrence [, match_option]]])
```

```
SELECT REGEXP_SUBSTR(street_address , ' [^ ]+ ') AS Road
FROM locations;
```

	ROAD
1	Via
2	Calle
3	(null)
4	(null)
5	Jabberwocky

In this example, the road names are extracted from the `LOCATIONS` table. To do this, the contents in the `STREET_ADDRESS` column that are after the first space are returned by using the `REGEXP_SUBSTR` function. In the expression used in the query `' [^ ]+ '`:

- `[` starts the expression
- ^ indicates NOT
- ' ' indicates space
- `]` ends the expression
- `+` indicates 1 or more

Lesson Agenda

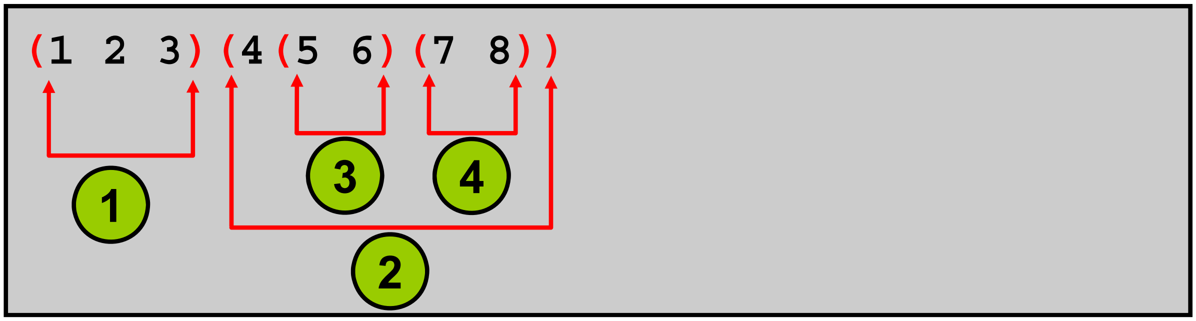
- Introduction to regular expressions
- Using metacharacters with regular expressions
- Using the regular expressions functions:
  - `REGEXP_LIKE`
  - `REGEXP_REPLACE`
  - `REGEXP_INSTR`
  - `REGEXP_SUBSTR`
- Accessing subexpressions
- Using the `REGEXP_COUNT` function

Subexpressions

Examine this expression:

```
(1 2 3)(4(5 6)(7 8))
```

The subexpressions are:



Oracle Database 11*g* provides regular expression support parameters to access a subexpression. In the slide example, a string of digits is shown. The parentheses identify the subexpressions within the string of digits. Reading from left to right, and from outer parentheses to the inner parentheses, the subexpressions in the string of digits are:

1. 123
2. 45678
3. 56
4. 78

You can search for any of those subexpressions with the `REGEXP_INSTR` and `REGEXP_SUBSTR` functions.

Using Subexpressions with Regular Expression Support

```
SELECT
  REGEXP_INSTR
① ('0123456789',       -- source char or search value
② '(123)(4(56)(78))', -- regular expression patterns
③ 1,                   -- position to start searching
④ 1,                   -- occurrence
⑤ 0,                   -- return option
⑥ 'i',                 -- match option (case insensitive)
⑦ 1)               -- sub-expression on which to search
   "Position"
FROM dual;
```

	Position
1	2

`REGEXP_INSTR` and `REGEXP_SUBSTR` have an optional `SUBEXPR` parameter that lets you target a particular substring of the regular expression being evaluated.

In the example shown in the slide, you may want to search for the first subexpression pattern in your list of subexpressions. The example shown identifies several parameters for the `REGEXP_INSTR` function. This example is specific to 11*g* R2 version of Oracle Database.

1. The string you are searching is identified.
2. The subexpressions are identified. The first subexpression is `123`. The second subexpression is `45678`, the third is `56`, and the fourth is `78`.
3. The third parameter identifies from which position to start searching.
4. The fourth parameter identifies the occurrence of the pattern you want to find. `1` means find the first occurrence.
5. The fifth parameter is the return option. This is the position of the first character of the occurrence. (If you specify `1`, the position of the character following the occurrence is returned.)
6. The sixth parameter identifies whether your search should be case-sensitive or not.
7. The last parameter is the parameter added in Oracle Database 11*g*. This parameter specifies which subexpression you want to find. In the example shown, you are searching for the first subexpression, which is `123`.

Why Access the *n*th Subexpression?

- A more realistic use: DNA sequencing
- You may need to find a specific subpattern that identifies a protein needed for immunity in mouse DNA.

```
SELECT
    REGEXP_INSTR('ccacctttccctccactcctcacgttctcacctgtaaagcgtccctc
    cctcatccccatgcccccttaccctgcagggtagagtaggctagaaaccagagagctccaagc
    tccatctgtggagaggtgccatccttgggctgcagagagaggagaatttgccccaaagctgcc
    tgcagagcttcaccacccttagtctcacaaagccttgagttcatagcatttcttgagttttca
    ccctgcccagcaggacactgcagcacccaaagggcttcccaggagtagggttgccctcaagag
    gctcttgggtctgatggccacatcctggaattgttttcaagttgatggtcacagccctgaggc
    atgtaggggcgtggggatgcgctctgctctgctctcctctcctgaacccctgaaccctctggc
    taccccagagcacttagagccag',
                     '(gtc(tcac)(aaag))',
                      1, 1, 0, 'i',
                      1) "Position"
FROM dual;
```

	Position
1	195

In life sciences, you may need to extract the offsets of subexpression matches from a DNA sequence for further processing. For example, you may need to find a specific protein sequence, such as the begin offset for the DNA sequence preceded by `gtc` and followed by `tcac` followed by `aaag`. To accomplish this goal, you can use the `REGEXP_INSTR` function, which returns the position where a match is found.

In the slide example, the position of the first subexpression `(gtc)` is returned. `gtc` appears starting in position 195 of the DNA string. This example is specific to 11*g* R2 version of Oracle Database.

If you modify the slide example to search for the second subexpression `(tcac)`, the query results in the following output. `tcac` appears starting in position 198 of the DNA string.

	Position
1	198

If you modify the slide example to search for the third subexpression `(aaag)`, the query results in the following output. `aaag` appears starting in position 202 of the DNA string.

	Position
1	202

REGEXP_SUBSTR: Example

```
SELECT
  REGEXP_SUBSTR
① ('acgctgcactgca', -- source char or search value
②  'acg(.*)gca',    -- regular expression pattern
③  1,               -- position to start searching
④  1,               -- occurrence
⑤  'i',             -- match option (case insensitive)
⑥  1)               -- sub-expression
 "Value"
FROM dual;
```

	Value
1	ctgcact

In the example shown in the slide:

1. `acgctgcactgca` is the source to be searched.
2. `acg(.*)gca` is the pattern to be searched. Find `acg` followed by `gca` with potential characters between the `acg` and the `gca`.
3. Start searching at the first character of the source.
4. Search for the first occurrence of the pattern.
5. Use non-case-sensitive matching on the source.
6. Use a nonnegative integer value that identifies the *n*th subexpression to be targeted. This is the subexpression parameter. In this example, `1` indicates the first subexpression. You can use a value from 0–9. A zero means that no subexpression is targeted. The default value for this parameter is 0.

This example is specific to 11*g* R2 version of Oracle Database.

Lesson Agenda

- Introduction to regular expressions
- Using metacharacters with regular expressions
- Using the regular expressions functions:
  - REGEXP_LIKE
  - REGEXP_REPLACE
  - REGEXP_INSTR
  - REGEXP_SUBSTR
- Accessing subexpressions
- Using the REGEXP_COUNT function

Using the `REGEXP_COUNT` Function

```
REGEXP_COUNT (source_char, pattern [, position
                 [, occurrence [, match_option]]])
```

```
SELECT REGEXP_COUNT(
    'ccacctttccctccactcctcacgttctcacctgtaaagcgtccctccctcatccccatgcccccttaccctgcag
    ggtagagtaggctagaaaccagagagctccaagctccatctgtggagaggtgccatccttgggctgcagagagaggag
    aatttgccccaaagctgcctgcagagcttcaccacccttagtctcacaaagccttgagttcatagcatttcttgagtt
    ttcaccctgcccagcaggacactgcagcacccaaagggcttcccaggagtagggttgccctcaagaggctcttgggtc
    tgatggccacatcctggaattgttttcaagttgatggtcacagccctgaggcatgtaggggcgtggggatgcgctctg
    ctctgctctcctctcctgaacccctgaaccctctggctaccccagagcacttagagccag',
    'gtc') AS Count
FROM dual;
```

	COUNT
1	4

The `REGEXP_COUNT` function evaluates strings by using characters as defined by the input character set. It returns an integer indicating the number of occurrences of the pattern. If no match is found, the function returns 0.

In the slide example, the number of occurrences for a DNA substring is determined by using the `REGEXP_COUNT` function. This example is specific to the 11*g* R2 version of Oracle Database.

The following example shows that the number of times the pattern `123` occurs in the string `123123123123` is three times. The search starts from the second position of the string.

```
SELECT REGEXP_COUNT
   ('123123123123', -- source char or search value
   '123',          -- regular expression pattern
    2,             -- position where the search should start
   'i')            -- match option (case insensitive)
   As Count
FROM dual;
```

	COUNT
1	3

Regular Expressions and Check Constraints: Examples

```
ALTER TABLE emp8
  ADD CONSTRAINT email_addr
  CHECK(REGEXP_LIKE(email,'@')) NOVALIDATE;
```

```
INSERT INTO emp8 VALUES
   (500,'Christian','Patel','ChrisP2creme.com',
    1234567890,'12-Jan-2004','HR_REP',2000,null,102,40);
```

```
Error starting at line 1 in command:
INSERT INTO emp8 VALUES
  (500,'Christian','Patel',
  'ChrisP2creme.com', 1234567890,
   '12-Jan-2004', 'HR_REP', 2000, null, 102, 40)
Error report:
SQL Error: ORA-02290: check constraint (TEACH_B.EMAIL_ADDR) violated
02290. 00000 -  "check constraint (%s.%s) violated"
*Cause:    The values being inserted do not satisfy the named check

*Action:   do not insert values that violate the constraint.
```

ORACLE

Regular expressions can also be used in CHECK constraints. In this example, a CHECK constraint is added on the EMAIL column of the EMPLOYEES table. This ensures that only strings containing an "@" symbol are accepted. The constraint is tested. The CHECK constraint is violated because the email address does not contain the required symbol. The NOVALIDATE clause ensures that existing data is not checked.

For the slide example, the emp8 table is created by using the following code:

```
CREATE TABLE emp8 AS SELECT * FROM employees;
```

Note: The example in the slide is executed by using the "Execute Statement" option in SQL Developer. The output format differs if you use the "Run Script" option.

Quiz

With the use of regular expressions in SQL and PL/SQL, you can:

a. Avoid intensive string processing of SQL result sets by middle-tier applications
b. Avoid data validation logic on the client
c. Enforce constraints on the server

Answer: a, b, c

Summary

In this lesson, you should have learned how to use regular expressions to search for, match, and replace strings.

In this lesson, you have learned to use the regular expression support features. Regular expression support is available in both SQL and PL/SQL.

Practice 8: Overview

This practice covers using regular expressions functions to do the following:

- Searching for, replacing, and manipulating data
- Creating a new `CONTACTS` table and adding a `CHECK` constraint to the `p_number` column to ensure that phone numbers are entered into the database in a specific standard format
- Testing the adding of some phone numbers into the `p_number` column by using various formats

In this practice, you use regular expressions functions to search for, replace, and manipulate data. You also create a new `CONTACTS` table and add a `CHECK` constraint to the `p_number` column to ensure that phone numbers are entered into the database in a specific standard format.

A
Table Descriptions

Schema Description

Overall Description

The Oracle database sample schemas portray a sample company that operates worldwide to fill orders for several different products. The company has three divisions:

- Human Resources: Tracks information about the employees and facilities
- Order Entry: Tracks product inventories and sales through various channels
- Sales History: Tracks business statistics to facilitate business decisions

Each of these divisions is represented by a schema. In this course, you have access to the objects in all the schemas. However, the emphasis of the examples, demonstrations, and practices is on the Human Resources (HR) schema.

All scripts necessary to create the sample schemas reside in the `$ORACLE_HOME/demo/schema/` folder.

Human Resources (HR)

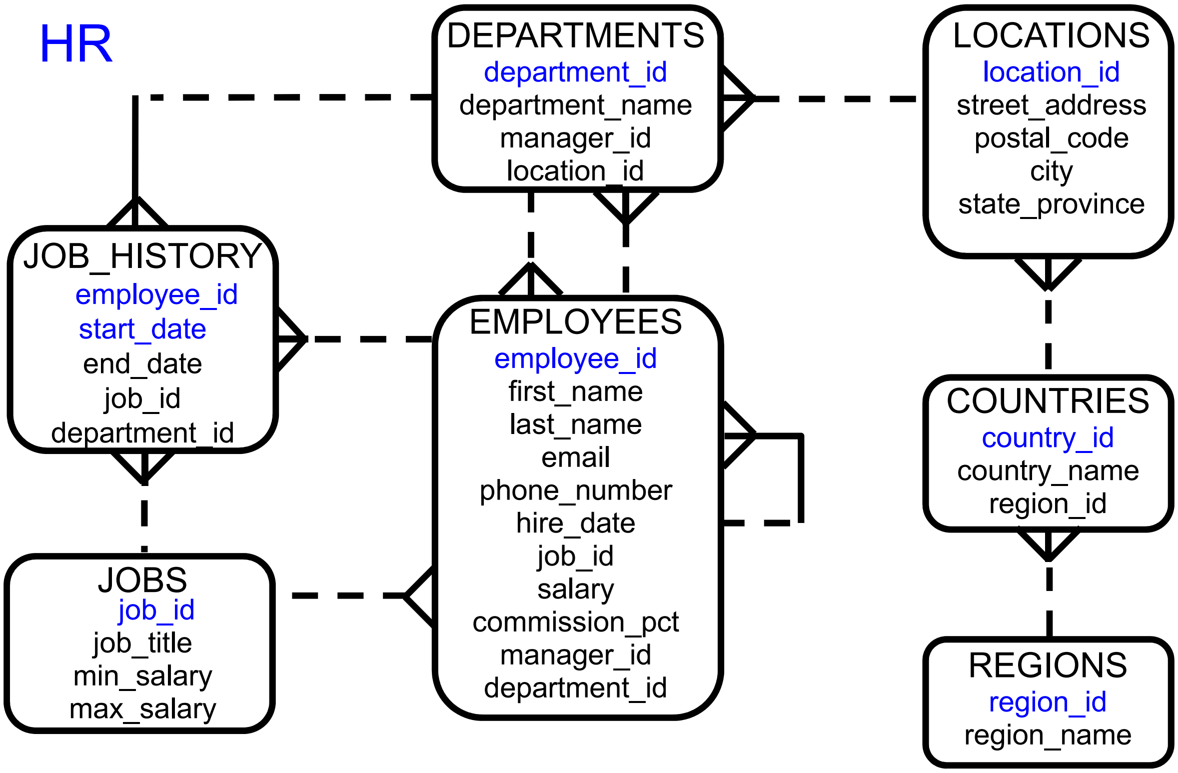
This is the schema that is used in this course. In the Human Resource (HR) records, each employee has an identification number, email address, job identification code, salary, and manager. Some employees earn commissions in addition to their salary.

The company also tracks information about jobs within the organization. Each job has an identification code, job title, and a minimum and maximum salary range for the job. Some employees have been with the company for a long time and have held different positions within the company. When an employee resigns, the duration the employee was working, the job identification number, and the department are recorded.

The sample company is regionally diverse, so it tracks the locations of its warehouses and departments. Each employee is assigned to a department, and each department is identified either by a unique department number or a short name. Each department is associated with one location, and each location has a full address that includes the street name, postal code, city, state or province, and the country code.

In places where the departments and warehouses are located, the company records details such as the country name, currency symbol, currency name, and the region where the country is located geographically.

The HR Entity Relationship Diagram



The Human Resources (HR) Table Descriptions

```
DESCRIBE countries

Name                                      Null?    Type
----------------------------------------- -------- ------------------------
COUNTRY_ID                                NOT NULL CHAR(2)
COUNTRY_NAME                                       VARCHAR2(40)
REGION_ID                                          NUMBER
```

```
SELECT * FROM countries
```

	COUNTRY_ID	COUNTRY_NAME	REGION_ID
1	AR	Argentina	2
2	AU	Australia	3
3	BE	Belgium	1
4	BR	Brazil	2
5	CA	Canada	2
6	CH	Switzerland	1
7	CN	China	3
8	DE	Germany	1
9	DK	Denmark	1
10	EG	Egypt	4
11	FR	France	1
12	HK	HongKong	3
13	IL	Israel	4
14	IN	India	3
15	IT	Italy	1
16	JP	Japan	3
17	KW	Kuwait	4
18	LB	Lebanon	4
19	MX	Mexico	2
20	NG	Nigeria	4
21	NL	Netherlands	1
22	SG	Singapore	3
23	UK	United Kingdom	1
24	US	United States of America	2
25	ZM	Zambia	4
26	ZW	Zimbabwe	4

```
DESCRIBE departments
```

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

```
SELECT * FROM departments
```

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	280	TRAINING	(null)	2500
2	290	Unknown	(null)	1700
3	300	ADVERTISING	(null)	1200
4	310	Unknown	(null)	1200
5	980	Education	(null)	2500
6	460	Training	(null)	2400
7	10	Administration	200	1700
8	20	Marketing	201	1800
9	30	Purchasing	114	1700
10	40	Human Resources	203	2400
11	50	Shipping	121	1500
12	60	IT	103	1400
13	70	Public Relations	204	2700
14	80	Sales	145	2500
15	90	Executive	100	1700
16	100	Finance	108	1700
17	110	Accounting	205	1700
18	120	Treasury	(null)	1700
19	130	Corporate Tax	(null)	1700
20	140	Control And Credit	(null)	1700
21	150	Shareholder Serv...	(null)	1700
22	160	Benefits	(null)	1700
23	170	Manufacturing	(null)	1700
24	180	Construction	(null)	1700
25	190	Contracting	(null)	1700
26	200	Operations	(null)	1700
27	210	IT Support	(null)	1700
28	220	NOC	(null)	1700
29	230	IT Helpdesk	(null)	1700
30	240	Government Sales	(null)	1700
31	250	Retail Sales	(null)	1700
32	260	Recruiting	(null)	1700
33	270	Payroll	(null)	1700

```
DESCRIBE employees
```

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

```
SELECT * FROM employees
```

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	198	Donald	OConnell	DOCONNEL	650.507.9833	21-JUN-99	SH_CLERK	2600	(null)	124	50
2	199	Douglas	Grant	DGRANT	650.507.9844	13-JAN-00	SH_CLERK	2600	(null)	124	50
3	200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400	(null)	101	10
4	201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	13000	(null)	100	20
5	202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	6000	(null)	201	20
6	203	Susan	Mavris	SMAVRIS	515.123.7777	07-JUN-94	HR_REP	6500	(null)	101	40
7	204	Hermann	Baer	HBAER	515.123.8888	07-JUN-94	PR_REP	10000	(null)	101	70
8	205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000	(null)	101	110
9	206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	8300	(null)	205	110
10	207	Joe	Harris	JAHARRIS	(null)	27-SEP-11	SA_REP	1000	0	145	80
11	208	Samuel	Joplin	SJOPLIN	(null)	28-SEP-11	SA_REP	1000	0	145	30
12	100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	(null)	(null)	90
13	101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000	(null)	100	90
14	102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000	(null)	100	90
15	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000	(null)	102	60
16	104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000	(null)	103	60
17	105	David	Austin	DAUSTIN	590.423.4569	25-JUN-97	IT_PROG	4800	(null)	103	60
18	106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-98	IT_PROG	4800	(null)	103	60
19	107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200	(null)	103	60
20	108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-94	FI_MGR	12000	(null)	101	100

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
21	109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG-94	FI_ACCOUNT	9000	(null)	108	100
22	110	John	Chen	JCHEN	515.124.4269	28-SEP-97	FI_ACCOUNT	8200	(null)	108	100
23	111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-97	FI_ACCOUNT	7700	(null)	108	100
24	112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-98	FI_ACCOUNT	7800	(null)	108	100
25	113	Luis	Popp	LPOPP	515.124.4567	07-DEC-99	FI_ACCOUNT	6900	(null)	108	100
26	114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-94	PU_MAN	11000	(null)	100	30
27	115	Alexander	Khoo	AKHOO	515.127.4562	18-MAY-95	PU_CLERK	3100	(null)	114	30
28	116	Shelli	Baida	SBAIDA	515.127.4563	24-DEC-97	PU_CLERK	2900	(null)	114	30
29	117	Sigal	Tobias	STOBIAS	515.127.4564	24-JUL-97	PU_CLERK	2800	(null)	114	30
30	118	Guy	Himuro	GHIMURO	515.127.4565	15-NOV-98	PU_CLERK	2600	(null)	114	30
31	119	Karen	Colmenares	KCOLMENA	515.127.4566	10-AUG-99	PU_CLERK	2500	(null)	114	30
32	120	Matthew	Weiss	MWEISS	650.123.1234	18-JUL-96	ST_MAN	8000	(null)	100	50
33	121	Adam	Fripp	AFRIPP	650.123.2234	10-APR-97	ST_MAN	8200	(null)	100	50
34	122	Payam	Kaufling	PKAUFLIN	650.123.3234	01-MAY-95	ST_MAN	7900	(null)	100	50
35	123	Shanta	Vollman	SVOLLMAN	650.123.4234	10-OCT-97	ST_MAN	6500	(null)	100	50
36	124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800	(null)	100	50
37	125	Julia	Nayer	JNAYER	650.124.1214	16-JUL-97	ST_CLERK	3200	(null)	120	50
38	126	Irene	Mikkilineni	IMIKKILI	650.124.1224	28-SEP-98	ST_CLERK	2700	(null)	120	50
39	127	James	Landry	JLANDRY	650.124.1334	14-JAN-99	ST_CLERK	2400	(null)	120	50
40	128	Steven	Markle	SMARKLE	650.124.1434	08-MAR-00	ST_CLERK	2200	(null)	120	50

Employees

. . .

40	128	Steven	Markle	SMARKLE	650.124.1434	08-MAR-00	ST_CLERK	2200	(null)	120	50
41	129	Laura	Bissot	LBISSOT	650.124.5234	20-AUG-97	ST_CLERK	3300	(null)	121	50
42	130	Mozhe	Atkinson	MATKINSO	650.124.6234	30-OCT-97	ST_CLERK	2800	(null)	121	50
43	131	James	Marlow	JAMRLOW	650.124.7234	16-FEB-97	ST_CLERK	2500	(null)	121	50
44	132	TJ	Olson	TJOLSON	650.124.8234	10-APR-99	ST_CLERK	2100	(null)	121	50
45	133	Jason	Mallin	JMALLIN	650.127.1934	14-JUN-96	ST_CLERK	3300	(null)	122	50
46	134	Michael	Rogers	MROGERS	650.127.1834	26-AUG-98	ST_CLERK	2900	(null)	122	50
47	135	Ki	Gee	KGEE	650.127.1734	12-DEC-99	ST_CLERK	2400	(null)	122	50
48	136	Hazel	Philtanker	HPHILTAN	650.127.1634	06-FEB-00	ST_CLERK	2200	(null)	122	50
49	137	Renske	Ladwig	RLADWIG	650.121.1234	14-JUL-95	ST_CLERK	3600	(null)	123	50
50	138	Stephen	Stiles	SSTILES	650.121.2034	26-OCT-97	ST_CLERK	3200	(null)	123	50
51	139	John	Seo	JSEO	650.121.2019	12-FEB-98	ST_CLERK	2700	(null)	123	50
52	140	Joshua	Patel	JPATEL	650.121.1834	06-APR-98	ST_CLERK	2500	(null)	123	50
53	141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500	(null)	124	50
54	142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	3100	(null)	124	50
55	143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600	(null)	124	50
56	144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-98	ST_CLERK	2500	(null)	124	50
57	145	John	Russell	JRUSSEL	011.44.1344....	01-OCT-96	SA_MAN	14000	0.4	100	80
58	146	Karen	Partners	KPARTNER	011.44.1344....	05-JAN-97	SA_MAN	13500	0.3	100	80
59	147	Alberto	Errazuriz	AERRAZUR	011.44.1344....	10-MAR-97	SA_MAN	12000	0.3	100	80
60	148	Gerald	Cambrault	GCAMBRAU	011.44.1344....	15-OCT-99	SA_MAN	11000	0.3	100	80
61	149	Eleni	Zlotkey	EZLOTKEY	011.44.1344....	29-JAN-00	SA_MAN	10500	0.2	100	80
62	150	Peter	Tucker	PTUCKER	011.44.1344....	30-JAN-97	SA_REP	10000	0.3	145	80
63	151	David	Bernstein	DBERNSTE	011.44.1344....	24-MAR-97	SA_REP	9500	0.25	145	80
64	152	Peter	Hall	PHALL	011.44.1344....	20-AUG-97	SA_REP	9000	0.25	145	80
65	153	Christopher	Olsen	COLSEN	011.44.1344....	30-MAR-98	SA_REP	8000	0.2	145	80
66	154	Nanette	Cambrault	NCAMBRAU	011.44.1344....	09-DEC-98	SA_REP	7500	0.2	145	80
67	155	Oliver	Tuvault	OTUVAULT	011.44.1344....	23-NOV-99	SA_REP	7000	0.15	145	80
68	156	Janette	King	JKING	011.44.1345....	30-JAN-96	SA_REP	10000	0.35	146	80
69	157	Patrick	Sully	PSULLY	011.44.1345....	04-MAR-96	SA_REP	9500	0.35	146	80
70	158	Allan	McEwen	AMCEWEN	011.44.1345....	01-AUG-96	SA_REP	9000	0.35	146	80
71	159	Lindsey	Smith	LSMITH	011.44.1345....	10-MAR-97	SA_REP	8000	0.3	146	80
72	160	Louise	Doran	LDORAN	011.44.1345....	15-DEC-97	SA_REP	7500	0.3	146	80
73	161	Sarath	Sewall	SSEWALL	011.44.1345....	03-NOV-98	SA_REP	7000	0.25	146	80
74	162	Clara	Vishney	CVISHNEY	011.44.1346....	11-NOV-97	SA_REP	10500	0.25	147	80
75	163	Danielle	Greene	DGREENE	011.44.1346....	19-MAR-99	SA_REP	9500	0.15	147	80
76	164	Mattea	Marvins	MMARVINS	011.44.1346....	24-JAN-00	SA_REP	7200	0.1	147	80
77	165	David	Lee	DLEE	011.44.1346....	23-FEB-00	SA_REP	6800	0.1	147	80
78	166	Sundar	Ande	SANDE	011.44.1346....	24-MAR-00	SA_REP	6400	0.1	147	80
79	167	Amit	Banda	ABANDA	011.44.1346....	21-APR-00	SA_REP	6200	0.1	147	80
80	168	Lisa	Ozer	LOZER	011.44.1343....	11-MAR-97	SA_REP	11500	0.25	148	80

. . .

Employees

. . .

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
80	168	Lisa	Ozer	LOZER	011.44.1343....	11-MAR-97	SA_REP	11500	0.25	148	80
81	169	Harrison	Bloom	HBLOOM	011.44.1343....	23-MAR-98	SA_REP	10000	0.2	148	80
82	170	Tayler	Fox	TFOX	011.44.1343....	24-JAN-98	SA_REP	9600	0.2	148	80
83	171	William	Smith	WSMITH	011.44.1343....	23-FEB-99	SA_REP	7400	0.15	148	80
84	172	Elizabeth	Bates	EBATES	011.44.1343....	24-MAR-99	SA_REP	7300	0.15	148	80
86	174	Ellen	Abel	EABEL	011.44.1644....	11-MAY-96	SA_REP	11000	0.3	149	80
87	175	Alyssa	Hutton	AHUTTON	011.44.1644....	19-MAR-97	SA_REP	8800	0.25	149	80
88	176	Jonathon	Taylor	JTAYLOR	011.44.1644....	24-MAR-98	SA_REP	11446.6	0.2	149	80
89	177	Jack	Livingston	JLIVINGS	011.44.1644....	23-APR-98	SA_REP	8400	0.2	149	80
90	178	Kimberely	Grant	KGRANT	011.44.1644....	24-MAY-99	SA_REP	7000	0.15	149	(null)
91	179	Charles	Johnson	CJOHNSON	011.44.1644....	04-JAN-00	SA_REP	6200	0.1	149	80
92	180	Winston	Taylor	WTAYLOR	650.507.9876	24-JAN-98	SH_CLERK	3200	(null)	120	50
93	181	Jean	Fleaur	JFLEAUR	650.507.9877	23-FEB-98	SH_CLERK	3100	(null)	120	50
94	182	Martha	Sullivan	MSULLIVA	650.507.9878	21-JUN-99	SH_CLERK	2500	(null)	120	50
95	183	Girard	Geoni	GGEONI	650.507.9879	03-FEB-00	SH_CLERK	2800	(null)	120	50
96	184	Nandita	Sarchand	NSARCHAN	650.509.1876	27-JAN-96	SH_CLERK	4200	(null)	121	50
97	185	Alexis	Bull	ABULL	650.509.2876	20-FEB-97	SH_CLERK	4100	(null)	121	50
98	186	Julia	Dellinger	JDELLING	650.509.3876	24-JUN-98	SH_CLERK	3400	(null)	121	50
99	187	Anthony	Cabrio	ACABRIO	650.509.4876	07-FEB-99	SH_CLERK	3000	(null)	121	50
100	188	Kelly	Chung	KCHUNG	650.505.1876	14-JUN-97	SH_CLERK	3800	(null)	122	50
101	189	Jennifer	Dilly	JDILLY	650.505.2876	13-AUG-97	SH_CLERK	3600	(null)	122	50
102	190	Timothy	Gates	TGATES	650.505.3876	11-JUL-98	SH_CLERK	2900	(null)	122	50
103	191	Randall	Perkins	RPERKINS	650.505.4876	19-DEC-99	SH_CLERK	2500	(null)	122	50
104	192	Sarah	Bell	SBELL	650.501.1876	04-FEB-96	SH_CLERK	4000	(null)	123	50
105	193	Britney	Everett	BEVERETT	650.501.2876	03-MAR-97	SH_CLERK	3900	(null)	123	50
106	194	Samuel	McCain	SMCCAIN	650.501.3876	01-JUL-98	SH_CLERK	3200	(null)	123	50
107	195	Vance	Jones	VJONES	650.501.4876	17-MAR-99	SH_CLERK	2800	(null)	123	50
108	196	Alana	Walsh	AWALSH	650.507.9811	24-APR-98	SH_CLERK	3100	(null)	124	50
109	197	Kevin	Feeney	KFEENEY	650.507.9822	23-MAY-98	SH_CLERK	3000	(null)	124	50

. . .

```
DESCRIBE job_history
```

```
 Name                                      Null?    Type
 ----------------------------------------- -------- -------------------------
 EMPLOYEE_ID                               NOT NULL NUMBER(6)
 START_DATE                                NOT NULL DATE
 END_DATE                                  NOT NULL DATE
 JOB_ID                                    NOT NULL VARCHAR2(10)
 DEPARTMENT_ID                                      NUMBER(4)
```

```
SELECT * FROM job_history
```

	EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
1	102	13-JAN-93	24-JUL-98	IT_PROG	60
2	101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
3	101	28-OCT-93	15-MAR-97	AC_MGR	110
4	201	17-FEB-96	19-DEC-99	MK_REP	20
5	114	24-MAR-98	31-DEC-99	ST_CLERK	50
6	122	01-JAN-99	31-DEC-99	ST_CLERK	50
7	200	17-SEP-87	17-JUN-93	AD_ASST	90
8	176	24-MAR-98	31-DEC-98	SA_REP	80
9	176	01-JAN-99	31-DEC-99	SA_MAN	80
10	200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

```
DESCRIBE jobs
```

```
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------
 JOB_ID                                    NOT NULL VARCHAR2(10)
 JOB_TITLE                                 NOT NULL VARCHAR2(35)
 MIN_SALARY                                         NUMBER(6)
 MAX_SALARY                                         NUMBER(6)
```

```
SELECT * FROM jobs
```

	JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
1	AD_PRES	President	20000	40000
2	AD_VP	Administration Vice President	15000	30000
3	AD_ASST	Administration Assistant	3000	6000
4	FI_MGR	Finance Manager	8200	16000
5	FI_ACCOUNT	Accountant	4200	9000
6	AC_MGR	Accounting Manager	8200	16000
7	AC_ACCOUNT	Public Accountant	4200	9000
8	SA_MAN	Sales Manager	10000	20000
9	SA_REP	Sales Representative	6000	12000
10	PU_MAN	Purchasing Manager	8000	15000
11	PU_CLERK	Purchasing Clerk	2500	5500
12	ST_MAN	Stock Manager	5500	8500
13	ST_CLERK	Stock Clerk	2000	5000
14	SH_CLERK	Shipping Clerk	2500	5500
15	IT_PROG	Programmer	4000	10000
16	MK_MAN	Marketing Manager	9000	15000
17	MK_REP	Marketing Representative	4000	9000
18	HR_REP	Human Resources Representative	4000	9000
19	PR_REP	Public Relations Representative	4500	10500

DESCRIBE locations

```
Name                                      Null?    Type
----------------------------------------- -------- ------------------
JOB_ID                                    NOT NULL VARCHAR2(10)
JOB_TITLE                                 NOT NULL VARCHAR2(35)
MIN_SALARY                                         NUMBER(6)
MAX_SALARY                                         NUMBER(6)
```

SELECT * FROM locations

	LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	COUNTRY_ID
1	1000	1297 Via Cola di Rie	00989	Roma	(null)	IT
2	1100	93091 Calle della Testa	10934	Venice	(null)	IT
3	1200	2017 Shinjuku-ku	1689	Tokyo	Tokyo Prefecture	JP
4	1300	9450 Kamiya-cho	6823	Hiroshima	(null)	JP
5	1400	2014 Jabberwocky Rd	26192	Southlake	Texas	US
6	1500	2011 Interiors Blvd	99236	South San Francisco	California	US
7	1600	2007 Zagora St	50090	South Brunswick	New Jersey	US
8	1700	2004 Charade Rd	98199	Seattle	Washington	US
9	1800	147 Spadina Ave	M5V 2L7	Toronto	Ontario	CA
10	1900	6092 Boxwood St	YSW 9T2	Whitehorse	Yukon	CA
11	2000	40-5-12 Laogianggen	190518	Beijing	(null)	CN
12	2100	1298 Vileparle (E)	490231	Bombay	Maharashtra	IN
13	2200	12-98 Victoria Street	2901	Sydney	New South Wales	AU
14	2300	198 Clementi North	540198	Singapore	(null)	SG
15	2400	8204 Arthur St	(null)	London	(null)	UK
16	2500	Magdalen Centre, The Oxford Science Park	OX9 9ZB	Oxford	Oxford	UK
17	2600	9702 Chester Road	09629850293	Stretford	Manchester	UK
18	2700	Schwanthalerstr. 7031	80925	Munich	Bavaria	DE
19	2800	Rua Frei Caneca 1360	01307-002	Sao Paulo	Sao Paulo	BR
20	2900	20 Rue des Corps-Saints	1730	Geneva	Geneve	CH
21	3000	Murtenstrasse 921	3095	Bern	BE	CH
22	3100	Pieter Breughelstraat 837	3029SK	Utrecht	Utrecht	NL
23	3200	Mariano Escobedo 9991	11932	Mexico City	Distrito Federal,	MX

```
DESCRIBE regions
```

```
Name                                      Null?    Type
----------------------------------------- -------- ------------------
REGION_ID                                 NOT NULL NUMBER
REGION_NAME                                        VARCHAR2(25)
```

```
SELECT * FROM regions
```

	REGION_ID	REGION_NAME
1	1	Europe
2	2	Americas
3	3	Asia
4	4	Middle East and Africa

B

Using SQL Developer

Objectives

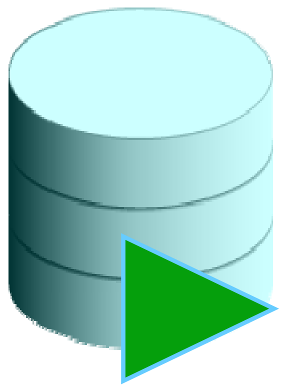
After completing this appendix, you should be able to do the following:

- List the key features of Oracle SQL Developer
- Identify menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use SQL Worksheet
- Save and run SQL scripts
- Create and save reports

In this appendix, you are introduced to the graphical tool called SQL Developer. You learn how to use SQL Developer for your database development tasks. You learn how to use SQL Worksheet to execute SQL statements and SQL scripts.

What Is Oracle SQL Developer?

- Oracle SQL Developer is a graphical tool that enhances productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema by using standard Oracle database authentication.



SQL Developer

ORACLE

Oracle SQL Developer is a free graphical tool designed to improve your productivity and simplify the development of everyday database tasks. With just a few clicks, you can easily create and debug stored procedures, test SQL statements, and view optimizer plans.

SQL Developer, the visual tool for database development, simplifies the following tasks:

- Browsing and managing database objects
- Executing SQL statements and scripts
- Editing and debugging PL/SQL statements
- Creating reports

You can connect to any target Oracle database schema by using standard Oracle database authentication. When connected, you can perform operations on objects in the database.

The SQL Developer 1.2 release tightly integrates with *Developer Migration Workbench* that provides users with a single point to browse database objects and data in third-party databases, and to migrate from these databases to Oracle. You can also connect to schemas for selected third-party (non-Oracle) databases such as MySQL, Microsoft SQL Server, and Microsoft Access, and you can view metadata and data in these databases.

Additionally, SQL Developer includes support for Oracle Application Express 3.0.1 (Oracle APEX).

Specifications of SQL Developer

- Is shipped along with Oracle Database 11*g* Release 2
- Is developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Provides default connectivity by using the Java Database Connectivity (JDBC) thin driver
- Connects to Oracle Database version 9.2.0.1 and later
- Is freely downloadable from the following link:

  http://www.oracle.com/technology/products/database/sql_developer/index.html

Oracle SQL Developer 1.5 is shipped along with Oracle Database 11*g* Release 2. SQL Developer is developed in Java, leveraging the Oracle JDeveloper integrated development environment (IDE). Therefore, it is a cross-platform tool. The tool runs on Windows, Linux, and Mac operating system (OS) X platforms.

Default connectivity to the database is through the JDBC thin driver, and therefore, no Oracle Home is required. SQL Developer does not require an installer and you need to simply unzip the downloaded file. With SQL Developer, users can connect to Oracle Databases 9.2.0.1 and later, and all Oracle Database editions including Express Edition.

Note

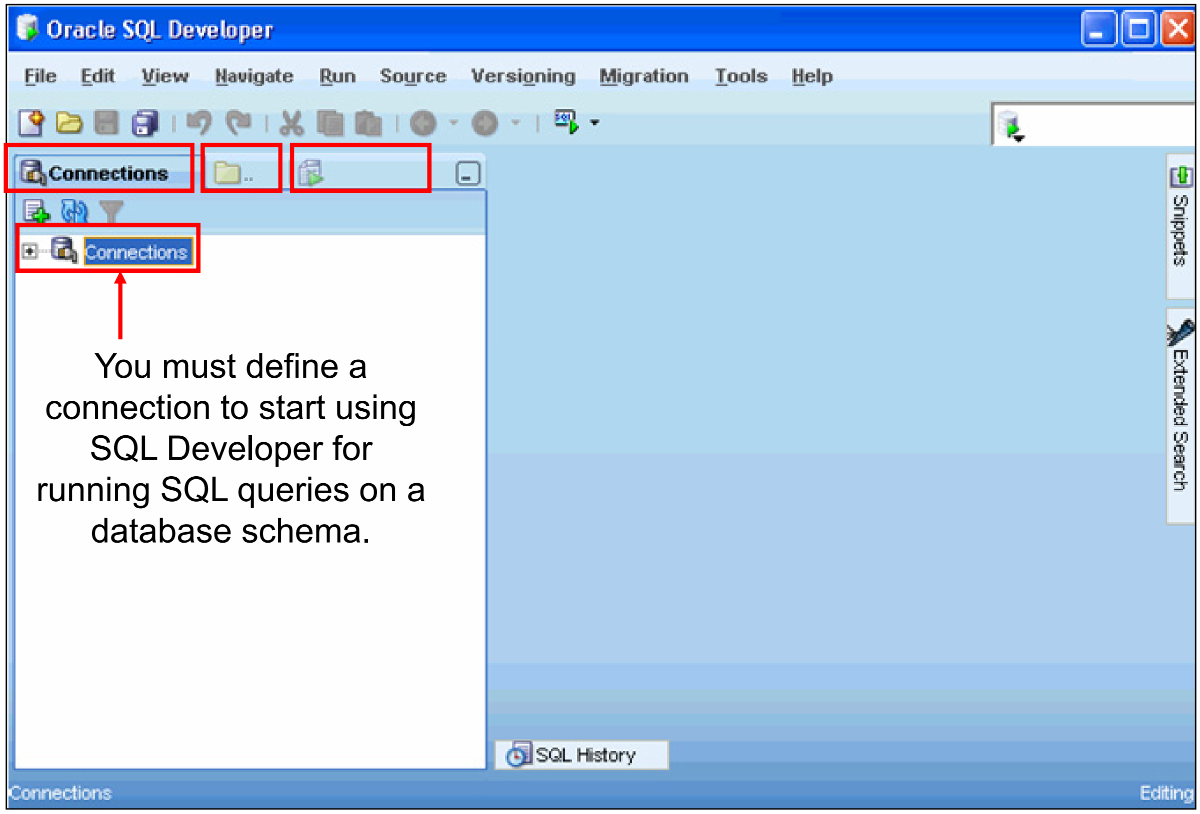
For Oracle Database versions earlier than Oracle Database 11*g* Release 2, you will have to download and install SQL Developer. SQL Developer 1.5 is freely downloadable from the following link:

http://www.oracle.com/technology/products/database/sql_developer/index.html

For instructions on how to install SQL Developer, you can visit the following link:

http://download.oracle.com/docs/cd/E12151_01/index.htm

SQL Developer 1.5 Interface



The SQL Developer 1.5 interface contains three main navigation tabs, from left to right:

- **Connections tab:** By using this tab, you can browse database objects and users to which you have access.
- **Files tab:** Identified by the Files folder icon, this tab enables you to access files from your local machine without having to use the File > Open menu.
- **Reports tab:** Identified by the Reports icon, this tab enables you to run predefined reports or create and add your own reports.

General Navigation and Use

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about selected objects. You can customize many aspects of the appearance and behavior of SQL Developer by setting preferences.

Note: You need to define at least one connection to be able to connect to a database schema and issue SQL queries or run procedures/functions.

Menus

The following menus contain standard entries, plus entries for features specific to SQL Developer:

- **View:** Contains options that affect what is displayed in the SQL Developer interface
- **Navigate:** Contains options for navigating to various panes and for executing subprograms
- **Run:** Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected, and also debugging options
- **Source:** Contains options for use when you edit functions and procedures
- **Versioning:** Provides integrated support for the following versioning and source control systems: Concurrent Versions System (CVS) and Subversion
- **Migration:** Contains options related to migrating third-party databases to Oracle
- **Tools:** Invokes SQL Developer tools such as SQL*Plus, Preferences, and SQL Worksheet

Note: The Run menu also contains options that are relevant when a function or procedure is selected for debugging. These are the same options that are found in the Debug menu in version 1.2.

Creating a Database Connection

- You must have at least one database connection to use SQL Developer.
- You can create and test connections for multiple:
  - Databases
  - Schemas
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an Extensible Markup Language (XML) file.
- Each additional database connection created is listed in the Connections Navigator hierarchy.

A connection is a SQL Developer object that specifies the necessary information for connecting to a specific database as a specific user of that database. To use SQL Developer, you must have at least one database connection, which may be existing, created, or imported.

You can create and test connections for multiple databases and for multiple schemas.

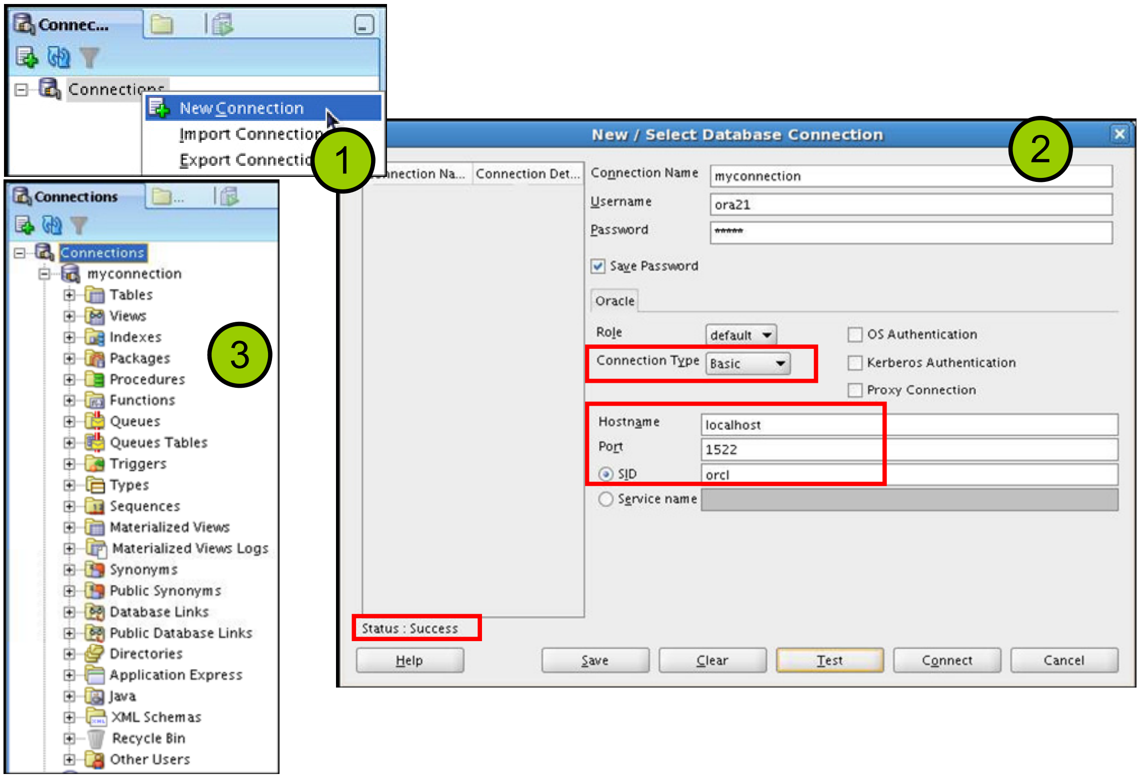
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory, but it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and display the Database Connections dialog box, SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.

Note: On Windows, if the `tnsnames.ora` file exists but its connections are not being used by SQL Developer, define `TNS_ADMIN` as a system environment variable.

You can export connections to an XML file so that you can reuse it later.

You can create additional connections as different users to the same database or to connect to the different databases.

Creating a Database Connection



To create a database connection, perform the following steps:

1. On the Connections tabbed page, right-click Connections and select New Connection.
2. In the New/Select Database Connection window, enter the connection name. Enter the username and password of the schema that you want to connect to.
   a) From the Role drop-down box, you can select either default or SYSDBA (you choose SYSDBA for the `sys` user or any user with database administrator privileges).
   b) You can select the connection type as:

      Basic: In this type, enter host name and SID for the database that you want to connect to. Port is already set to 1521. Or you can also choose to enter the Service name directly if you use a remote database connection.

      TNS: You can select any one of the database aliases imported from the `tnsnames.ora` file.

      LDAP: You can look up database services in Oracle Internet Directory, which is a component of Oracle Identity Management.

      Advanced: You can define a custom JDBC URL to connect to the database.
   c) Click Test to ensure that the connection has been set correctly.
   d) Click Connect.

If you select the Save Password check box, the password is saved to an XML file. So, after you close the SQL Developer connection and open it again, you are not prompted for the password.

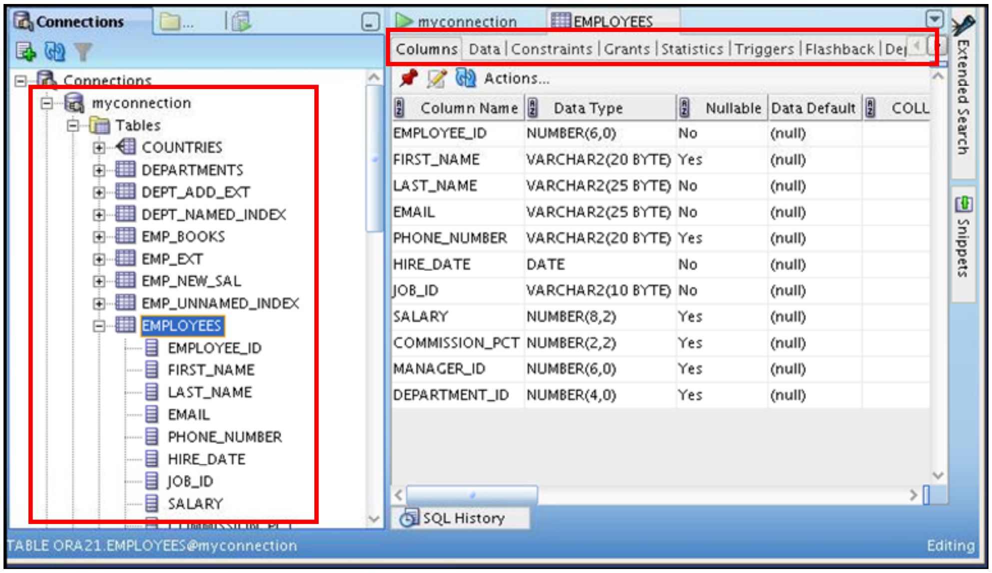
3. The connection gets added in the Connections Navigator. You can expand the connection to view the database objects and view object definitions, for example, dependencies, details, statistics, and so on.

Note: From the same New/Select Database Connection window, you can define connections to non-Oracle data sources using the Access, MySQL, and SQL Server tabs. However, these connections are read-only connections that enable you to browse objects and data in that data source.

Browsing Database Objects

Use the Connections Navigator to:

- Browse through many objects in a database schema
- Review the definitions of objects at a glance



After you create a database connection, you can use the Connections Navigator to browse through many objects in a database schema including Tables, Views, Indexes, Packages, Procedures, Triggers, and Types.

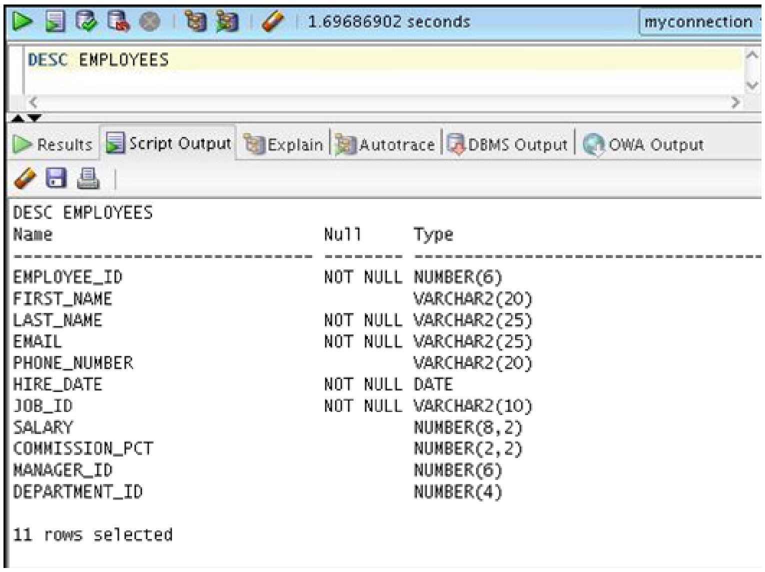
You can see the definition of the objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, the details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

If you want to see the definition of the `EMPLOYEES` table as shown in the slide, perform the following steps:

1. Expand the Connections node in the Connections Navigator.
2. Expand Tables.
3. Click `EMPLOYEES`. By default, the Columns tab is selected. It shows the column description of the table. Using the Data tab, you can view the table data and also enter new rows, update data, and commit these changes to the database.

Displaying the Table Structure

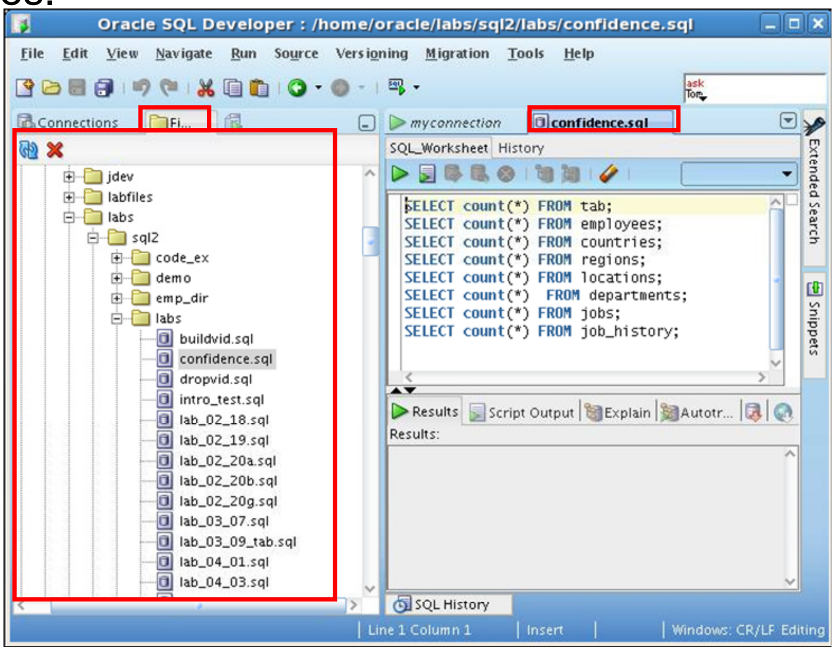
Use the `DESCRIBE` command to display the structure of a table:



In SQL Developer, you can also display the structure of a table using the `DESCRIBE` command. The result of the command is a display of column names and data types as well as an indication of whether a column must contain data.

Browsing Files

Use the File Navigator to explore the file system and open system files.



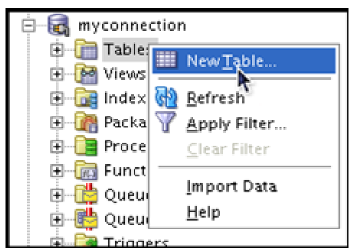
Browsing Database Objects

You can use the File Navigator to browse and open system files.

- To view the files navigator, click the Files tab, or click View > Files.
- To view the contents of a file, double-click a file name to display its contents in the SQL worksheet area.

Creating a Schema Object

- SQL Developer supports the creation of any schema object by:
  - Executing a SQL statement in SQL Worksheet
  - Using the context menu
- Edit the objects by using an edit dialog box or one of the many context-sensitive menus.
- View the data definition language (DDL) for adjustments such as creating a new object or editing an existing schema object.

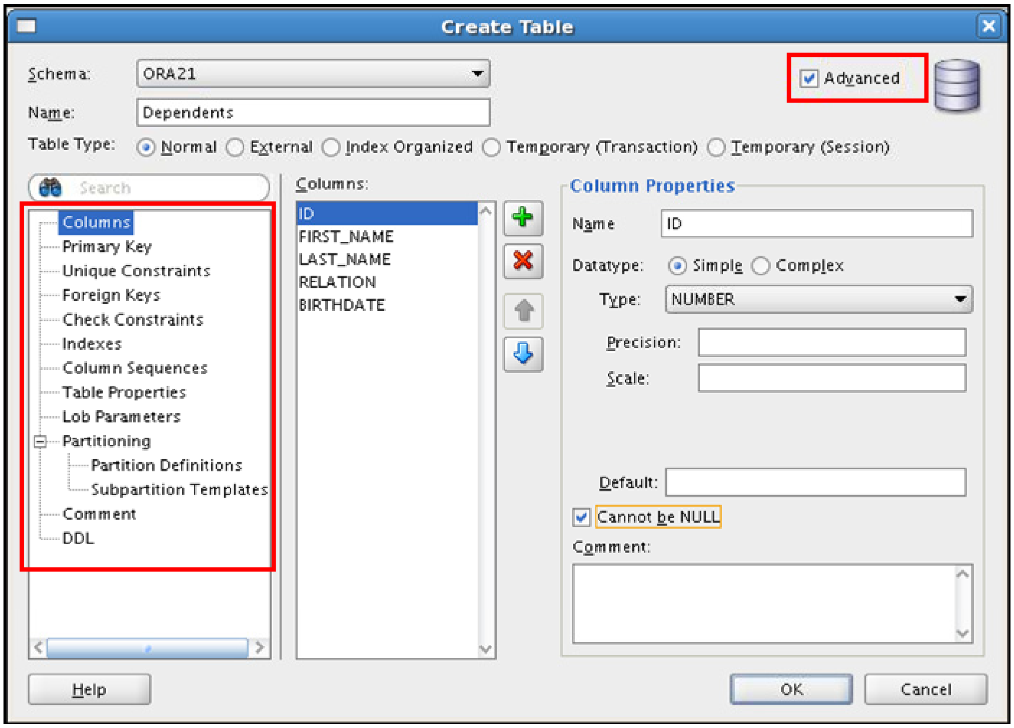


SQL Developer supports the creation of any schema object by executing a SQL statement in SQL Worksheet. Alternatively, you can create objects by using the context menus. When created, you can edit the objects by using an edit dialog box or one of the many context-sensitive menus.

As new objects are created or existing objects are edited, the DDL for those adjustments is available for review. An Export DDL option is available if you want to create the full DDL for one or more objects in the schema.

The slide shows how to create a table byb using the context menu. To open a dialog box for creating a new table, right-click Tables and select New Table. The dialog boxes to create and edit database objects have multiple tabs, each reflecting a logical grouping of properties for that type of object.

Creating a New Table: Example



In the Create Table dialog box, if you do not select the Advanced check box, you can create a table quickly by specifying columns and some frequently used features.

If you select the Advanced check box, the Create Table dialog box changes to one with multiple options, in which you can specify an extended set of features while you create the table.

The example in the slide shows how to create the `DEPENDENTS` table by selecting the Advanced check box.

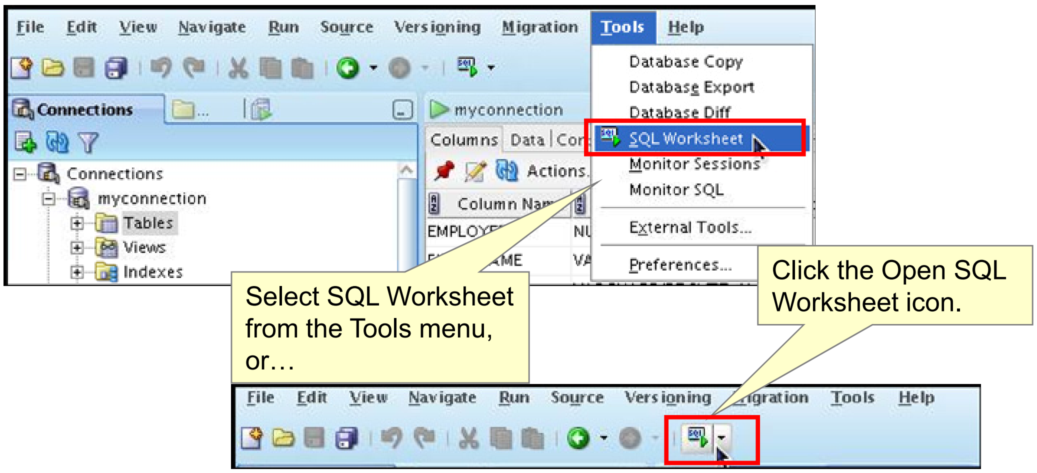
To create a new table, perform the following steps:

1. In the Connections Navigator, right-click Tables.
2. Select Create TABLE.
3. In the Create Table dialog box, select Advanced.
4. Specify column information.
5. Click OK.

Although it is not required, you should also specify a primary key by using the Primary Key tab in the dialog box. Sometimes, you may want to edit the table that you have created; to do so, right-click the table in the Connections Navigator and select Edit.

Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL *Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.



When you connect to a database, a SQL Worksheet window for that connection automatically opens. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. The SQL Worksheet supports SQL*Plus statements to a certain extent. SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database.

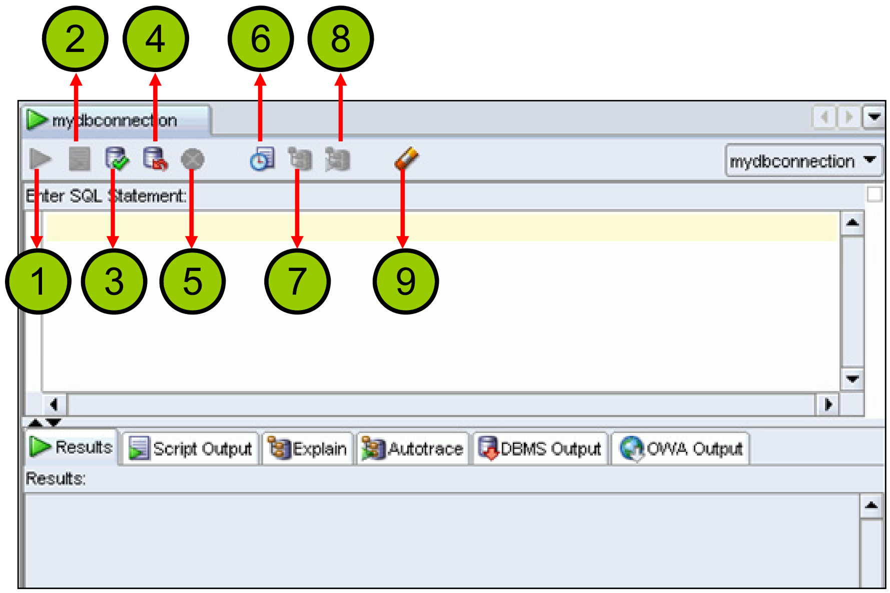
You can specify actions that can be processed by the database connection associated with the worksheet, such as:

- Creating a table
- Inserting data
- Creating and editing a trigger
- Selecting data from a table
- Saving the selected data to a file

You can display a SQL Worksheet by using one of the following:

- Select Tools > SQL Worksheet.
- Click the Open SQL Worksheet icon.

Using the SQL Worksheet



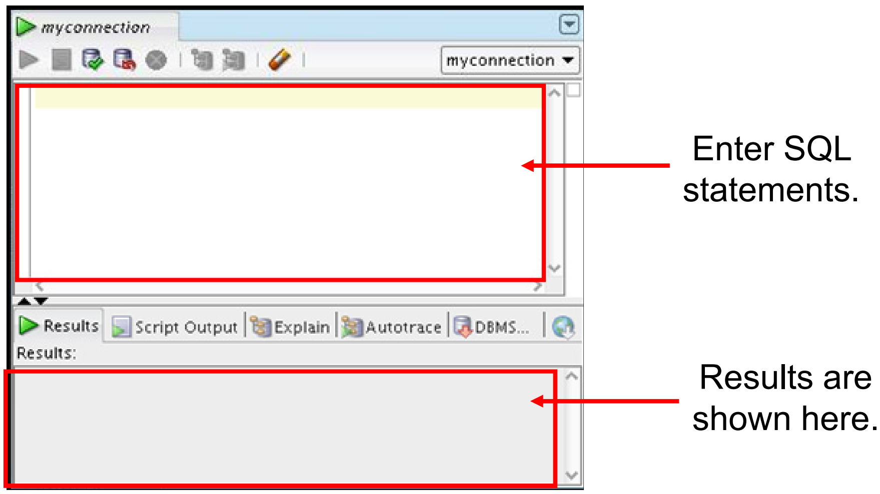
You may want to use the shortcut keys or icons to perform certain tasks such as executing a SQL statement, running a script, and viewing the history of SQL statements that you have executed. You can use the SQL Worksheet toolbar that contains icons to perform the following tasks:

1. **Execute Statement:** Executes the statement where the cursor is located in the Enter SQL Statement box. You can use bind variables in the SQL statements, but not substitution variables.
2. **Run Script:** Executes all statements in the Enter SQL Statement box by using the Script Runner. You can use substitution variables in the SQL statements, but not bind variables.
3. **Commit:** Writes any changes to the database and ends the transaction
4. **Rollback:** Discards any changes to the database, without writing them to the database, and ends the transaction
5. **Cancel:** Stops the execution of any statements currently being executed
6. **SQL History:** Displays a dialog box with information about SQL statements that you have executed
7. **Execute Explain Plan:** Generates the execution plan, which you can see by clicking the Explain tab

8. **Autotrace:** Generates trace information for the statement
9. **Clear:** Erases the statement or statements in the Enter SQL Statement box

Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.



When you connect to a database, a SQL Worksheet window for that connection automatically opens. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. All SQL and PL/SQL commands are supported as they are passed directly from the SQL Worksheet to the Oracle database. SQL*Plus commands used in SQL Developer have to be interpreted by the SQL Worksheet before being passed to the database.

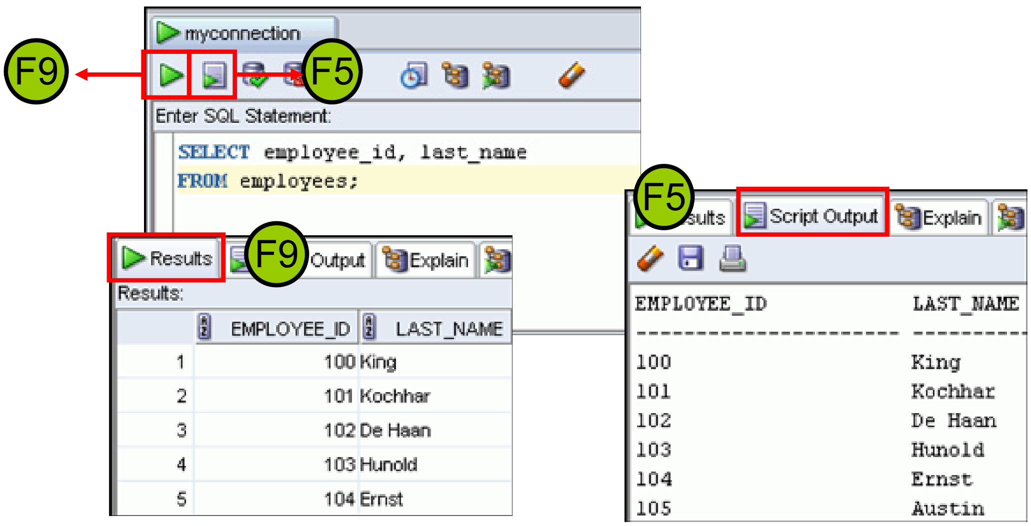
The SQL Worksheet currently supports a number of SQL*Plus commands. Commands not supported by the SQL Worksheet are ignored and are not sent to the Oracle database. Through the SQL Worksheet, you can execute SQL statements and some of the SQL*Plus commands.

You can display a SQL Worksheet by using any of the following two options:

- Select Tools > SQL Worksheet.
- Click the Open SQL Worksheet icon.

Executing SQL Statements

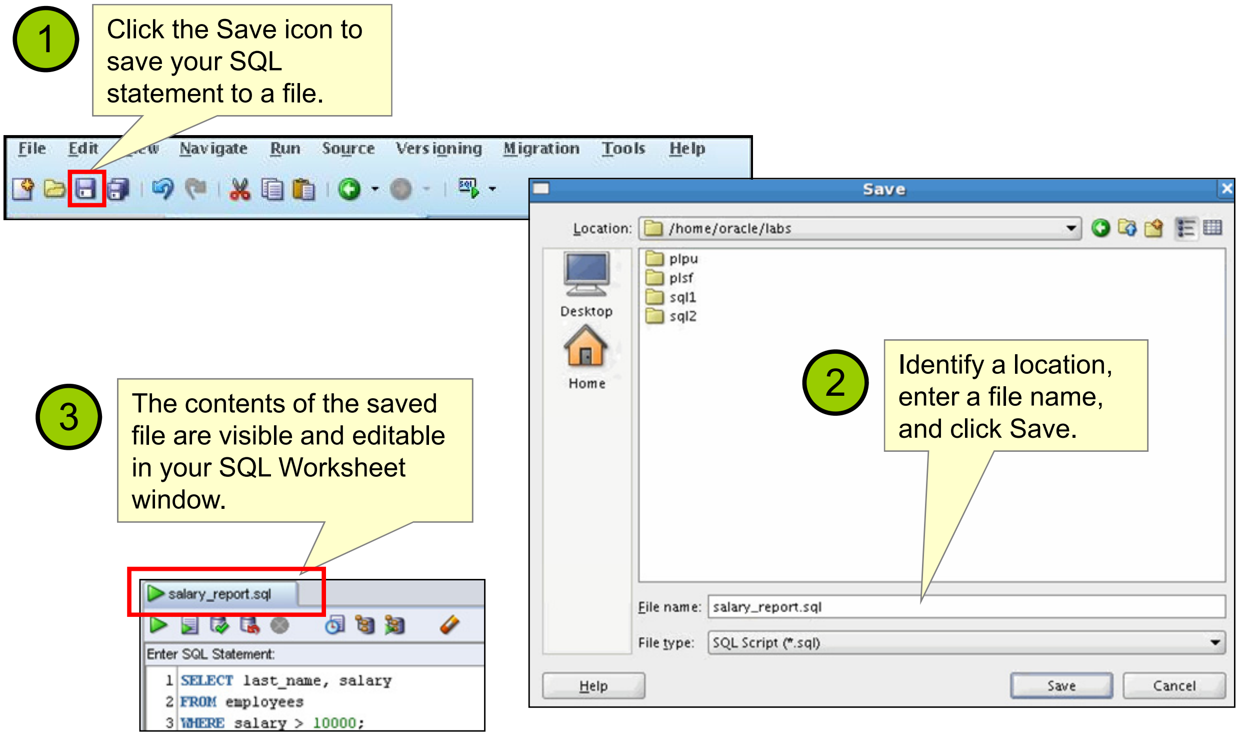
Use the Enter SQL Statement box to enter single or multiple SQL statements.



ORACLE

The example in the slide shows the difference in output for the same query when the F9 key or Execute Statement is used, versus the output when F5 or Run Script is used.

Saving SQL Scripts



You can save your SQL statements from the SQL Worksheet into a text file. To save the contents of the Enter SQL Statement box, follow these steps:

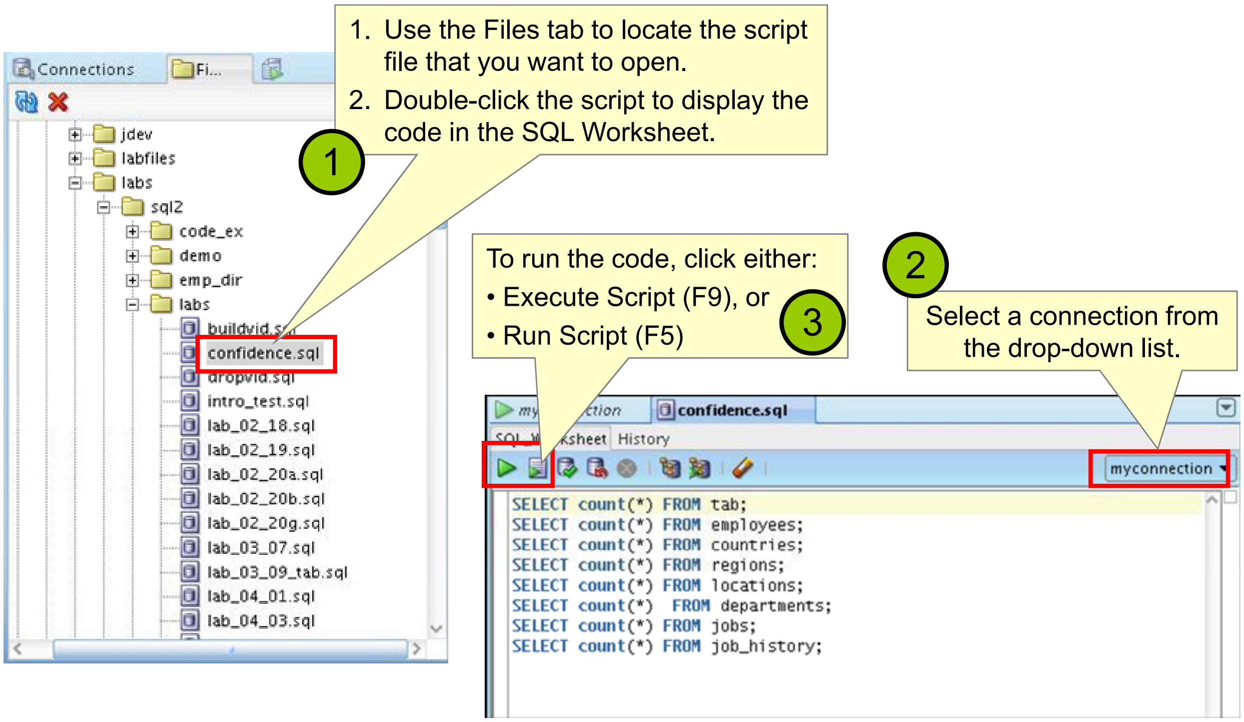
1. Click the Save icon or use the File > Save menu item.
2. In the Windows Save dialog box, enter a file name and the location where you want the file saved.
3. Click Save.

After you save the contents to a file, the Enter SQL Statement window displays a tabbed page of your file contents. You can have multiple files open at the same time. Each file is displayed as a tabbed page.

Script Pathing

You can select a default path to look for scripts and to save scripts. Under Tools > Preferences > Database > Worksheet Parameters, enter a value in the “Select default path to look for scripts” field.

Executing Saved Script Files: Method 1



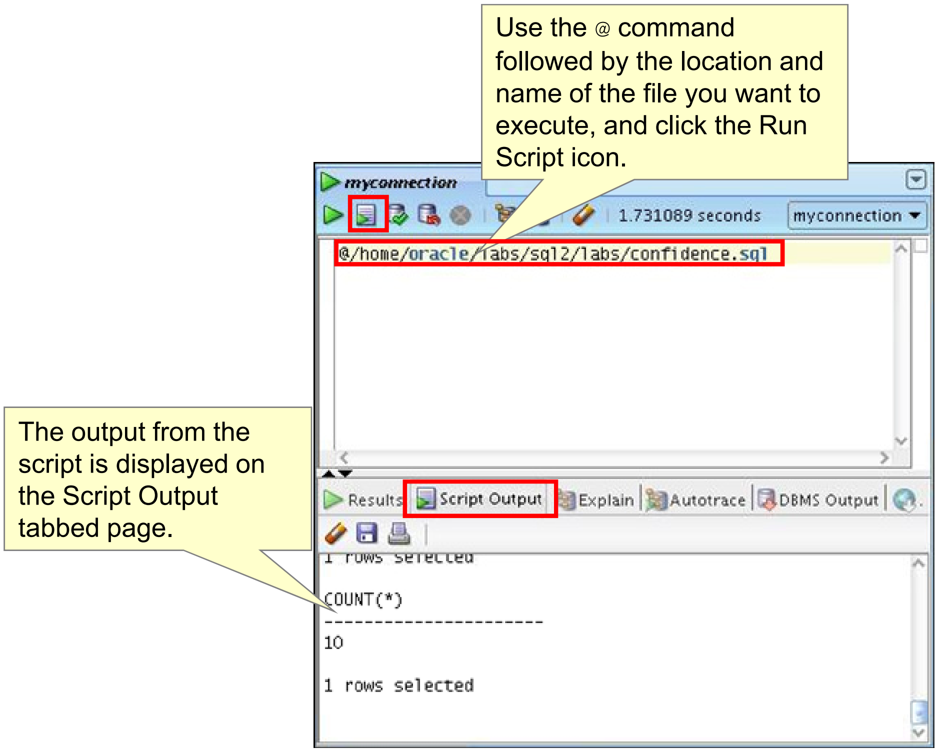
To open a script file and display the code in the SQL Worksheet area, perform the following:

1. In the files navigator select (or navigate to) the script file that you want to open.
2. Double-click to open. The code of the script file is displayed in the SQL Worksheet area.
3. Select a connection from the connection drop-down list.
4. To run the code, click the Run Script (F5) icon on the SQL Worksheet toolbar. If you have not selected a connection from the connection drop-down list, a connection dialog box will appear. Select the connection you want to use for the script execution.

Alternatively, you can also:

1. Select File > Open. The Open dialog box is displayed.
2. In the Open dialog box, select (or navigate to) the script file that you want to open.
3. Click Open. The code of the script file is displayed in the SQL Worksheet area.
4. Select a connection from the connection drop-down list.
5. To run the code, click the Run Script (F5) icon on the SQL Worksheet toolbar. If you have not selected a connection from the connection drop-down list, a connection dialog box will appear. Select the connection you want to use for the script execution.

Executing Saved Script Files: Method 2



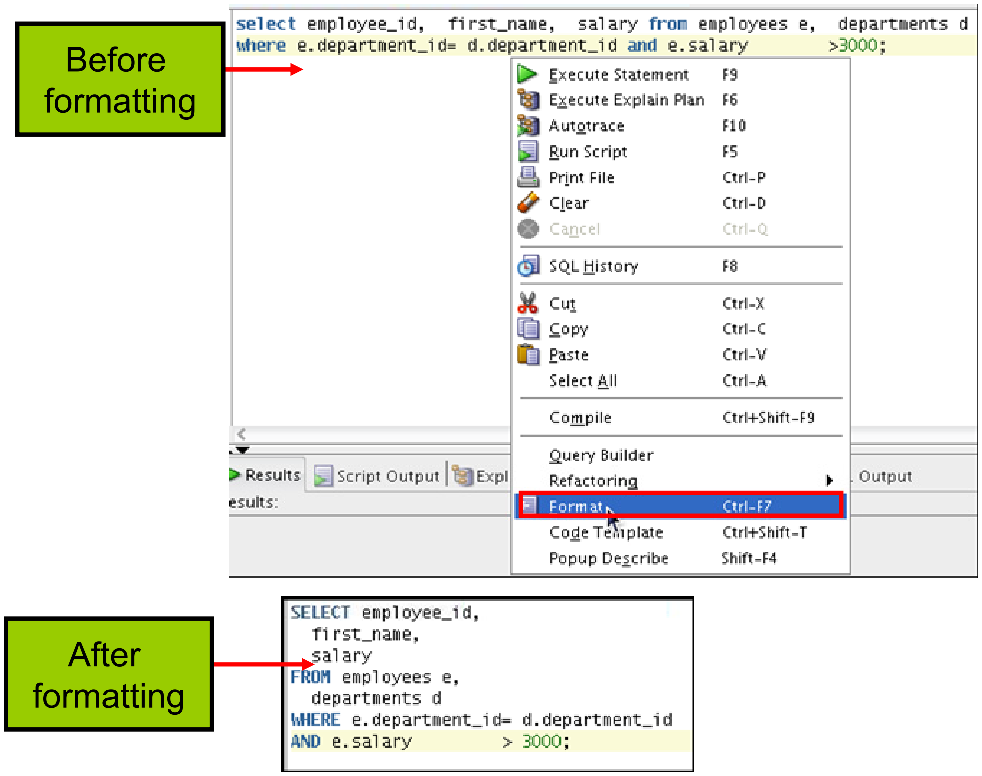
ORACLE

To run a saved SQL script, perform the following:

1. Use the **@** command, followed by the location, and name of the file you want to run, in the Enter SQL Statement window.
2. Click the Run Script icon.

The results from running the file are displayed on the Script Output tabbed page. You can also save the script output by clicking the Save icon on the Script Output tabbed page. The Windows Save dialog box appears and you can identify a name and location for your file.

Formatting the SQL Code



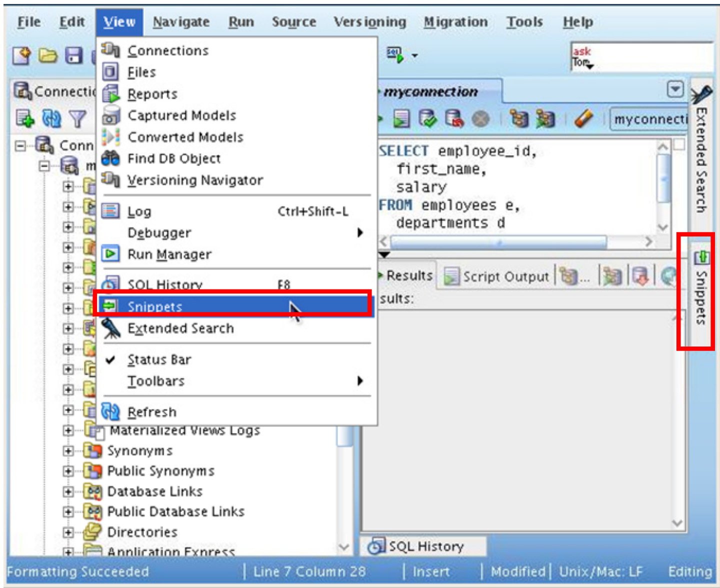
You may want to beautify the indentation, spacing, capitalization, and line separation of the SQL code. SQL Developer has a feature for formatting SQL code.

To format the SQL code, right-click in the statement area and select Format SQL.

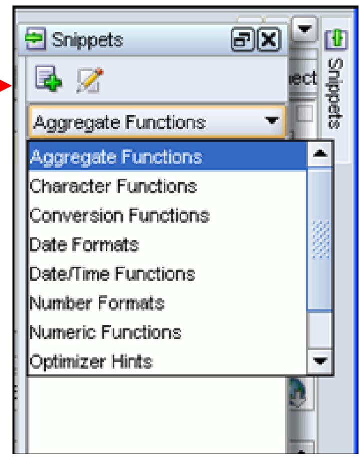
In the example in the slide, before formatting, the SQL code has the keywords not capitalized and the statement not properly indented. After formatting, the SQL code is beautified with the keywords capitalized and the statement properly indented.

Using Snippets

Snippets are code fragments that may be just syntax or examples.



When you place your cursor here, it shows the Snippets window. From the drop-down list, you can select the functions category that you want.

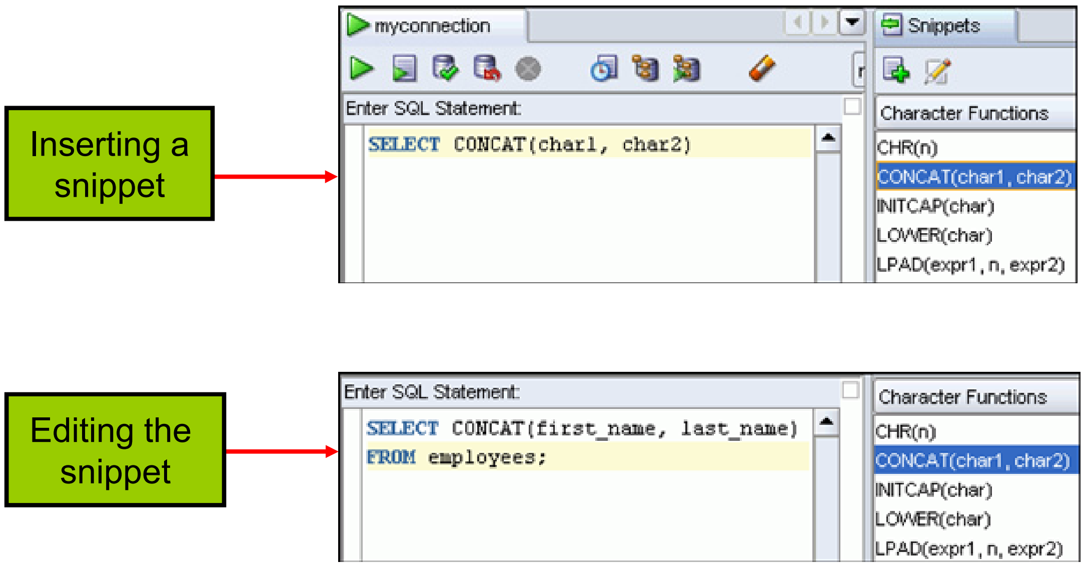


You may want to use certain code fragments when you use the SQL Worksheet or create or edit a PL/SQL function or procedure. SQL Developer has a feature called Snippets. Snippets are code fragments such as SQL functions, Optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag snippets into the Editor window.

To display Snippets, select View > Snippets.

The Snippets window is displayed at the right side. You can use the drop-down list to select a group. A Snippets button is placed in the right window margin, so that you can display the Snippets window if it becomes hidden.

Using Snippets: Example



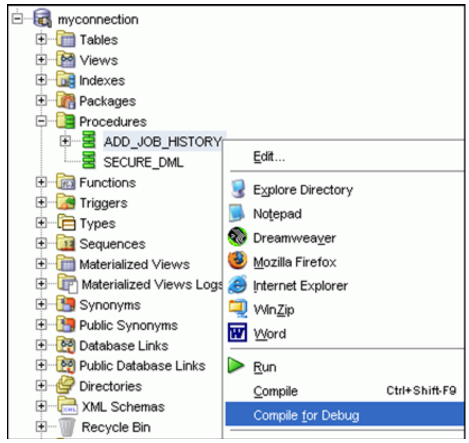
To insert a Snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window into the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function in a tool tip, place the cursor over the function name.

The example in the slide shows that `CONCAT(char1, char2)` is dragged from the Character Functions group in the Snippets window. Then the `CONCAT` function syntax is edited and the rest of the statement is added as in the following:

```
SELECT CONCAT(first_name, last_name)
FROM employees;
```

Debugging Procedures and Functions

- Use SQL Developer to debug PL/SQL functions and procedures.
- Use the "Compile for Debug" option to perform a PL/SQL compilation so that the procedure can be debugged.
- Use Debug menu options to set breakpoints, and to perform step-into and step-over tasks.



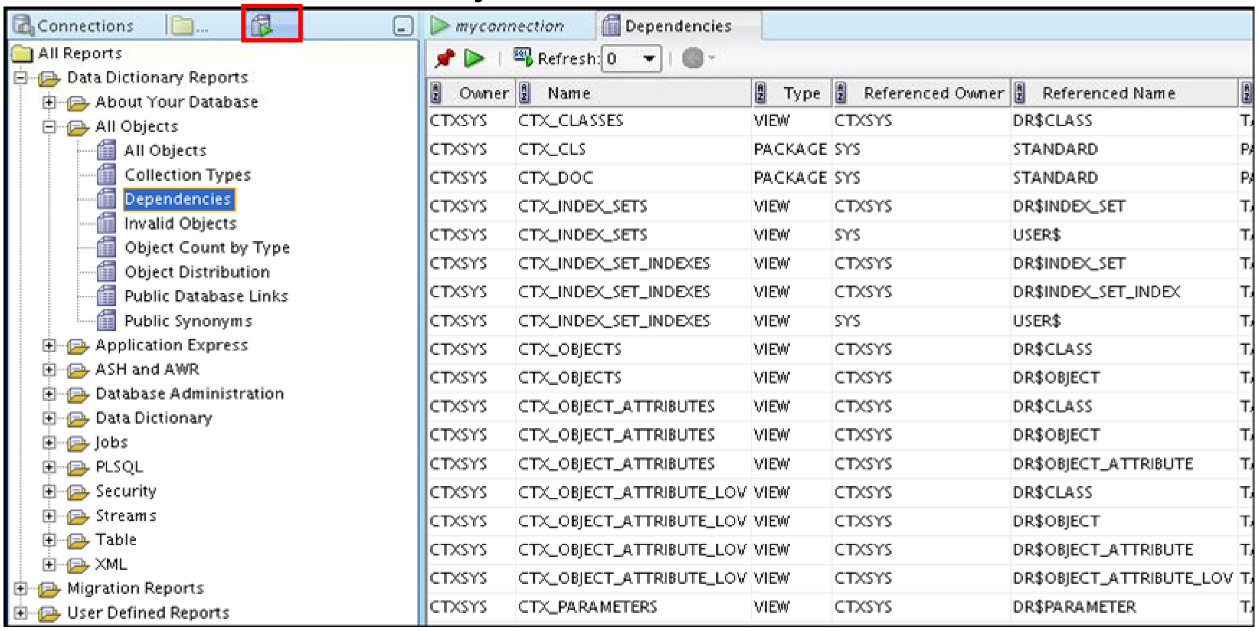
In SQL Developer, you can debug PL/SQL procedures and functions. Using the Debug menu options, you can perform the following debugging tasks:

- **Find Execution Point** goes to the next execution point.
- **Resume** continues execution.
- **Step Over** bypasses the next method and goes to the next statement after the method.
- **Step Into** goes to the first statement in the next method.
- **Step Out** leaves the current method and goes to the next statement.
- **Step to End of Method** goes to the last statement of the current method.
- **Pause** halts execution but does not exit, thus allowing you to resume execution.
- **Terminate** halts and exits the execution. You cannot resume execution from this point; instead, to start running or debugging from the beginning of the function or procedure, click the Run or Debug icon on the Source tab toolbar.
- **Garbage Collection** removes invalid objects from the cache in favor of more frequently accessed and more valid objects.

These options are also available as icons on the debugging toolbar.

Database Reporting

SQL Developer provides a number of predefined reports about the database and its objects.



ORACLE

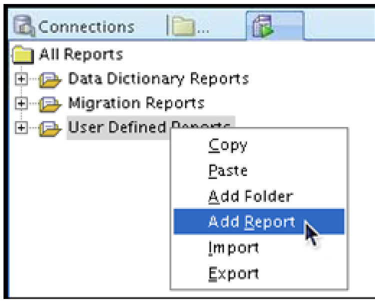
SQL Developer provides many reports about the database and its objects. These reports can be grouped into the following categories:

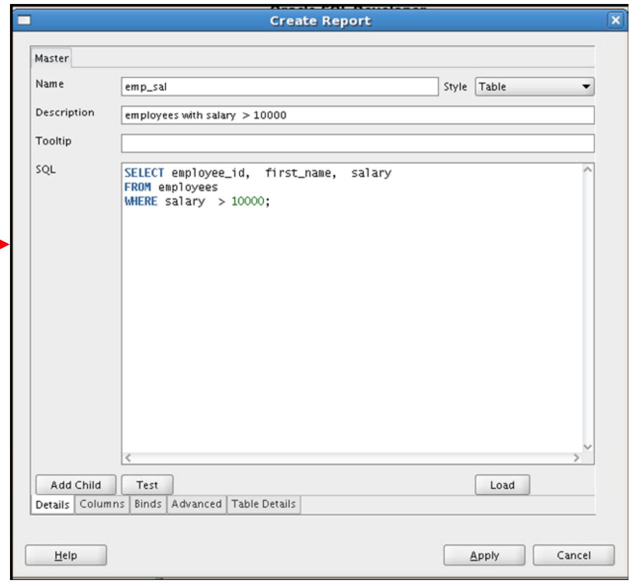
- About Your Database reports
- Database Administration reports
- Table reports
- PL/SQL reports
- Security reports
- XML reports
- Jobs reports
- Streams reports
- All Objects reports
- Data Dictionary reports
- User-Defined reports

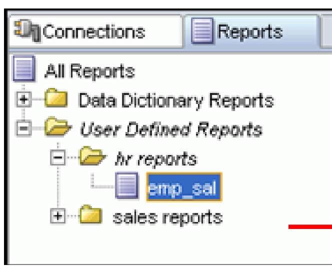
To display reports, click the Reports tab at the left side of the window. Individual reports are displayed in tabbed panes at the right side of the window; and for each report, you can select (using a drop-down list) the database connection for which to display the report. For reports about objects, the objects shown are only those visible to the database user associated with the selected database connection, and the rows are usually ordered by Owner. You can also create your own user-defined reports.

Creating a User-Defined Report

Create and save user-defined reports for repeated use.







Organize reports in folders.

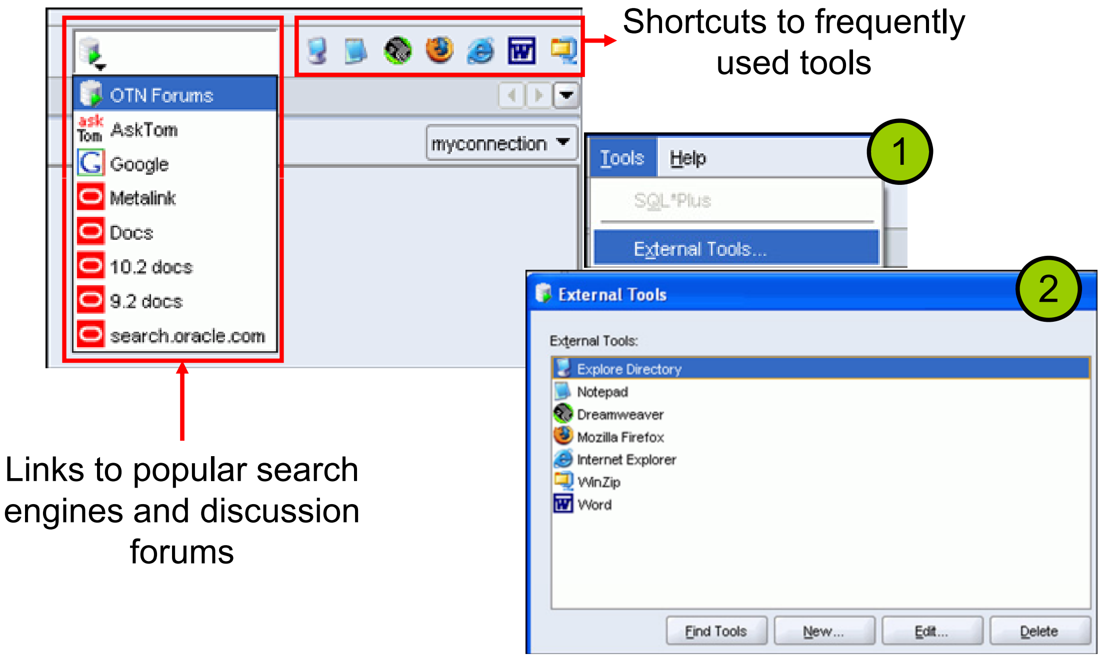
User-defined reports are reports created by SQL Developer users. To create a user-defined report, perform the following steps:

1. Right-click the User Defined Reports node under Reports, and select Add Report.
2. In the Create Report Dialog box, specify the report name and the SQL query to retrieve information for the report. Then, click Apply.

In the example in the slide, the report name is specified as `emp_sal`. An optional description is provided indicating that the report contains details of employees with `salary >= 10000`. The complete SQL statement for retrieving the information to be displayed in the user-defined report is specified in the SQL box. You can also include an optional tool tip to be displayed when the cursor stays briefly over the report name in the Reports navigator display.

You can organize user-defined reports in folders, and you can create a hierarchy of folders and subfolders. To create a folder for user-defined reports, right-click the User Defined Reports node, or any folder name under that node, and select Add Folder. Information about user-defined reports, including any folders for these reports, is stored in a file named `UserReports.xml` under the directory for user-specific information.

Search Engines and External Tools



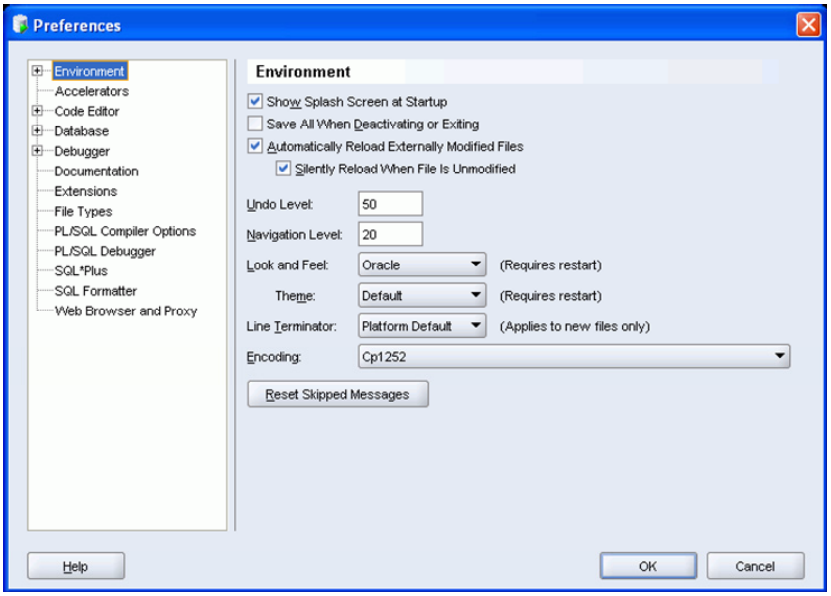
To enhance productivity of the SQL developers, SQL Developer has added quick links to popular search engines and discussion forums such as AskTom, Google, and so on. Also, you have shortcut icons to some of the frequently used tools such as Notepad, Microsoft Word, and Dreamweaver, available to you.

You can add external tools to the existing list or even delete shortcuts to tools that you do not use frequently. To do so, perform the following:

1. From the Tools menu, select External Tools.
2. In the External Tools dialog box, select New to add new tools. Select Delete to remove any tool from the list.

Setting Preferences

- Customize the SQL Developer interface and environment.
- In the Tools menu, select Preferences.

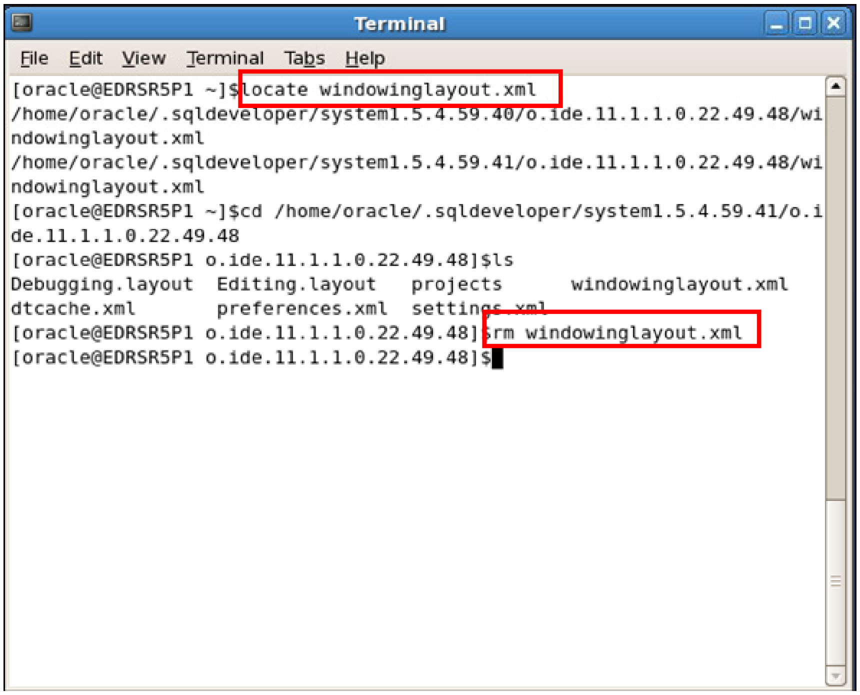


You can customize many aspects of the SQL Developer interface and environment by modifying SQL Developer preferences according to your preferences and needs. To modify SQL Developer preferences, select Tools, then Preferences.

The preferences are grouped into categories, such as the following :

- Environment
- Accelerators (keyboard shortcuts)
- Code Editors
- Database
- Debugger
- Documentation
- Extensions
- File Types
- Migration
- PL/SQL Compilers
- PL/SQL Debugger

Resetting the SQL Developer Layout



While working with SQL Developer, if the Connections Navigator disappears or if you cannot dock the Log window in its original place, perform the following steps to fix the problem:

1. Exit from SQL Developer.
2. Open a terminal window and use the `locate` command to find the location of `windowinglayout.xml`.
3. Go to the directory that has `windowinglayout.xml` and delete it.
4. Restart SQL Developer.

Summary

In this appendix, you should have learned how to use SQL Developer to do the following:

- Browse, create, and edit database objects
- Execute SQL statements and scripts in SQL Worksheet
- Create and save custom reports

SQL Developer is a free graphical tool to simplify database development tasks. Using SQL Developer, you can browse, create, and edit database objects. You can use SQL Worksheet to run SQL statements and scripts. SQL Developer enables you to create and save your own special set of reports for repeated use.

Using SQL*Plus

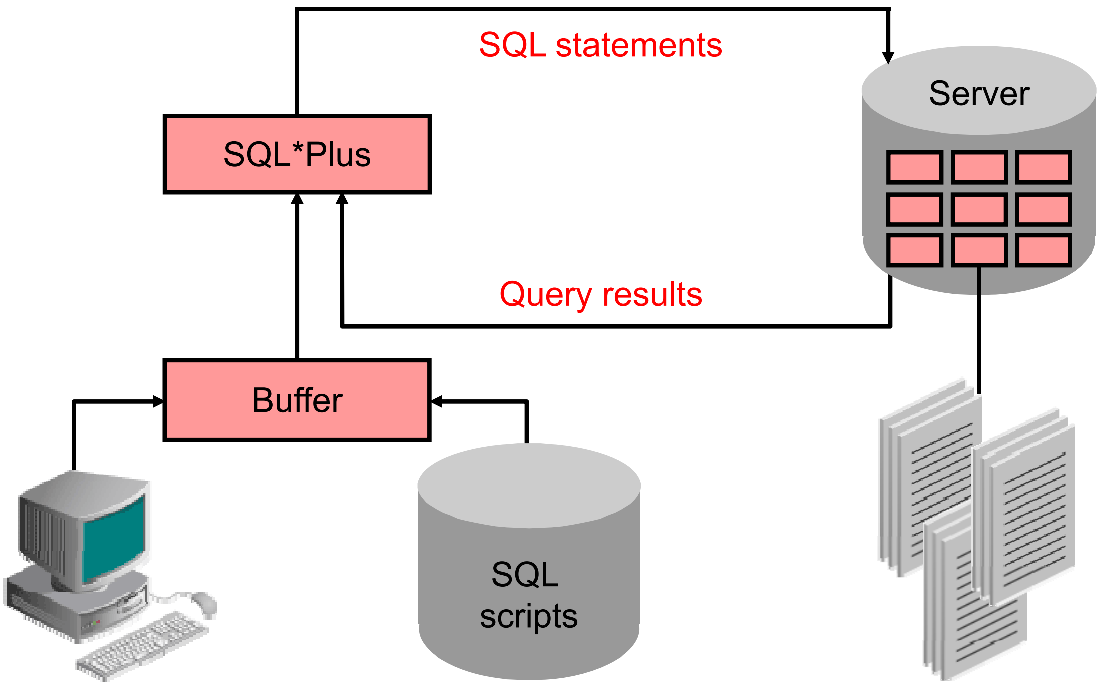
Objectives

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

- Log in to SQL*Plus
- Edit SQL commands
- Format the output using SQL*Plus commands
- Interact with script files

You might want to create `SELECT` statements that can be used again and again. This appendix also covers the use of SQL*Plus commands to execute SQL statements. You learn how to format output by using SQL*Plus commands, edit SQL commands, and save scripts in SQL*Plus.

SQL and SQL*Plus Interaction



SQL and SQL*Plus

SQL is a command language used for communication with the Oracle Server from any tool or application. Oracle SQL contains many extensions. When you enter a SQL statement, it is stored in a part of the memory called the *SQL buffer* and remains there until you enter a new SQL statement. SQL*Plus is an Oracle tool that recognizes and submits SQL statements to the Oracle9*i* Server for execution. It contains its own command language.

Features of SQL

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

Features of SQL*Plus

- Accepts ad hoc entry of statements
- Accepts SQL input from files
- Provides a line editor for modifying SQL statements

- Controls environmental settings
- Formats query results into basic reports
- Accesses local and remote databases

SQL Statements Versus SQL*Plus Commands

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

SQL*Plus
- An environment
- Oracle-proprietary
- Keywords can be abbreviated.
- Commands do not allow manipulation of values in the database.

The following table compares SQL and SQL*Plus:

SQL	SQL*Plus
Is a language for communicating with the Oracle server to access data	Recognizes SQL statements and sends them to the server
Is based on American National Standards Institute (ANSI)–standard SQL	Is the Oracle-proprietary interface for executing SQL statements
Manipulates data and table definitions in the database	Does not allow manipulation of values in the database
Is entered into the SQL buffer on one or more lines	Is entered one line at a time, not stored in the SQL buffer
Does not have a continuation character	Uses a dash (–) as a continuation character if the command is longer than one line
Cannot be abbreviated	Can be abbreviated
Uses a termination character to execute commands immediately	Does not require termination characters; executes commands immediately
Uses functions to perform some formatting	Uses commands to format data

SQL*Plus: Overview

- Log in to SQL*Plus.
- Describe the table structure.
- Edit your SQL statement.
- Execute SQL from SQL*Plus.
- Save SQL statements to files and append SQL statements to files.
- Execute saved files.
- Load commands from the file to the buffer to edit.

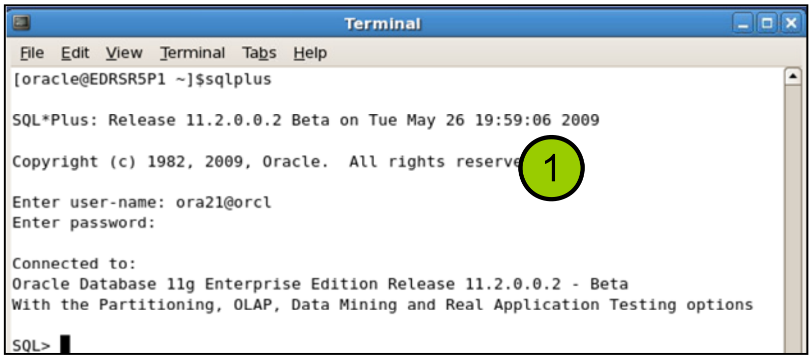
SQL*Plus is an environment in which you can:

- 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

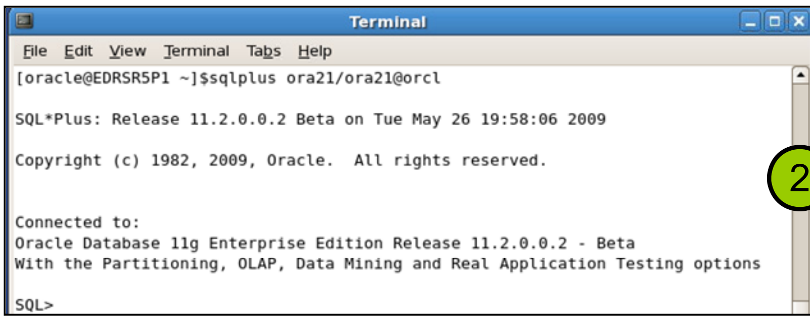
SQL*Plus commands can be divided into the following main categories:

Category	**Purpose**
Environment	Affect the general behavior of SQL statements for the session.
Format	Format query results.
File manipulation	Save, load, and run script files.
Execution	Send SQL statements from the SQL buffer to the Oracle server.
Edit	Modify SQL statements in the buffer.
Interaction	Create and pass variables to SQL statements, print variable values, and print messages to the screen.
Miscellaneous	Connect to the database, manipulate the SQL*Plus environment, and display column definitions.

Logging In to SQL*Plus




```
sqlplus [username[/password[@database]]]
```



ORACLE

How you invoke SQL*Plus depends on which type of operating system you are running Oracle Database in.

To log in from a Linux environment:

1. Right-click your Linux desktop and select terminal.
2. Enter the `sqlplus` command shown in the slide.
3. Enter the username, password, and database name.

In the syntax:

`username`	Your database username
`password`	Your database password (Your password is visible if you enter it here.)
`@database`	The database connect string

Note: To ensure the integrity of your password, do not enter it at the operating system prompt. Instead, enter only your username. Enter your password at the password prompt.

Displaying the Table Structure

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

```
DESC[RIBE] tablename
```


In SQL*Plus, you can display the structure of a table by using the `DESCRIBE` command. The result of the command is a display of column names and data types as well as an indication if a column must contain data.

In the syntax:

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

To describe the `DEPARTMENTS` table, use this command:

```
SQL> 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)
```

Displaying the Table Structure

```
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)
```


The example in the slide displays the information about the structure of the `DEPARTMENTS` table. In the result:

`Null?`: Specifies whether a column must contain data. (`NOT NULL` indicates that a column must contain data.)

`Type`: Displays the data type for a column

SQL*Plus Editing Commands

- `A[PPEND] text`
- `C[HANGE] / old / new`
- `C[HANGE] / text /`
- `CL[EAR] BUFF[ER]`
- `DEL`
- `DEL n`
- `DEL m n`

SQL*Plus commands are entered one line at a time and are not stored in the SQL buffer.

Command	Description
`A[PPEND] text`	Adds text to the end of the current line
`C[HANGE] / old / new`	Changes *old* text to *new* in the current line
`C[HANGE] / text /`	Deletes *text* from the current line
`CL[EAR] BUFF[ER]`	Deletes all lines from the SQL buffer
`DEL`	Deletes current line
`DEL n`	Deletes line *n*
`DEL m n`	Deletes lines *m* to *n* inclusive

Guidelines

- If you press Enter before completing a command, SQL*Plus prompts you with a line number.
- You terminate the SQL buffer either by entering one of the terminator characters (semicolon or slash) or by pressing Enter twice. The SQL prompt then appears.

SQL*Plus Editing Commands

- `I[NPUT]`
- `I[NPUT]` *`text`*
- `L[IST]`
- `L[IST]` *`n`*
- `L[IST]` *`m n`*
- `R[UN]`
- *`n`*
- *`n text`*
- `0` *`text`*

Command	**Description**
`I[NPUT]`	Inserts an indefinite number of lines
`I[NPUT]` *`text`*	Inserts a line consisting of *`text`*
`L[IST]`	Lists all lines in the SQL buffer
`L[IST]` *`n`*	Lists one line (specified by *`n`*)
`L[IST]` *`m n`*	Lists a range of lines (*`m`* to *`n`*) inclusive
`R[UN]`	Displays and runs the current SQL statement in the buffer
`n`	Specifies the line to make the current line
`n text`	Replaces line *`n`* with *`text`*
`0` *`text`*	Inserts a line before line 1

Note: You can enter only one SQL*Plus command for each SQL prompt. SQL*Plus commands are not stored in the buffer. To continue a SQL*Plus command on the next line, end the first line with a hyphen (-).

Using LIST, n, and APPEND

```
LIST
  1  SELECT last_name
  2* FROM   employees
```

```
1
  1* SELECT last_name
```

```
A , job_id
  1* SELECT last_name, job_id
```

```
LIST
  1  SELECT last_name, job_id
  2* FROM   employees
```


- Use the `L[IST]` command to display the contents of the SQL buffer. The asterisk (`*`) beside line 2 in the buffer indicates that line 2 is the current line. Any edits that you made apply to the current line.
- Change the number of the current line by entering the number (`n`) of the line that you want to edit. The new current line is displayed.
- Use the `A[PPEND]` command to add text to the current line. The newly edited line is displayed. Verify the new contents of the buffer by using the `LIST` command.

Note: Many SQL*Plus commands, including `LIST` and `APPEND`, can be abbreviated to just their first letter. `LIST` can be abbreviated to `L`; `APPEND` can be abbreviated to `A`.

Using the CHANGE Command

```
LIST
  1* SELECT * from employees
```

```
c/employees/departments
  1* SELECT * from departments
```

```
LIST
  1* SELECT * from departments
```


- Use `L[IST]` to display the contents of the buffer.
- Use the `C[HANGE]` command to alter the contents of the current line in the SQL buffer. In this case, replace the `employees` table with the `departments` table. The new current line is displayed.
- Use the `L[IST]` command to verify the new contents of the buffer.

SQL*Plus File Commands

- `SAVE` *`filename`*
- `GET` *`filename`*
- `START` *`filename`*
- `@` *`filename`*
- `EDIT` *`filename`*
- `SPOOL` *`filename`*
- `EXIT`

SQL statements communicate with the Oracle Server. SQL*Plus commands control the environment, format query results, and manage files. You can use the commands described in the following table:

Command	Description
`SAV[E]` *`filename`* `[.ext] [REP[LACE]APP[END]]`	Saves current contents of SQL buffer to a file. Use `APPEND` to add to an existing file; use `REPLACE` to overwrite an existing file. The default extension is `.sql`.
`GET` *`filename`* `[.ext]`	Writes the contents of a previously saved file to the SQL buffer. The default extension for the file name is `.sql`.
`STA[RT]` *`filename`* `[.ext]`	Runs a previously saved command file
`@` *`filename`*	Runs a previously saved command file (same as `START`)
`ED[IT]`	Invokes the editor and saves the buffer contents to a file named `afiedt.buf`
`ED[IT] [`*`filename`*`[.ext]]`	Invokes the editor to edit the contents of a saved file
`SPO[OL] [`*`filename`*`[.ext]|OFF|OUT]`	Stores query results in a file. `OFF` closes the spool file. `OUT` closes the spool file and sends the file results to the printer.
`EXIT`	Quits SQL*Plus

Using the SAVE and START Commands

```
LIST
  1  SELECT last_name, manager_id, department_id
  2* FROM employees
```

```
SAVE my_query
  Created file my_query
```

```
START my_query

LAST_NAME                 MANAGER_ID DEPARTMENT_ID
------------------------- ---------- -------------
King                                            90
Kochhar                          100            90
...
107 rows selected.
```


SAVE

Use the SAVE command to store the current contents of the buffer in a file. In this way, you can store frequently used scripts for use in the future.

START

Use the START command to run a script in SQL*Plus. You can also, alternatively, use the symbol @ to run a script.

```
@my_query
```

SERVEROUTPUT Command

- Use the SET SERVEROUT[PUT] command to control whether to display the output of stored procedures or PL/SQL blocks in SQL*Plus.
- The DBMS_OUTPUT line length limit is increased from 255 bytes to 32767 bytes.
- The default size is now unlimited.
- Resources are not preallocated when SERVEROUTPUT is set.
- Because there is no performance penalty, use UNLIMITED unless you want to conserve physical memory.

```
SET SERVEROUT[PUT] {ON | OFF} [SIZE {n | UNL[IMITED]}]
    [FOR[MAT] {WRA[PPED] | WOR[D_WRAPPED] | TRU[NCATED]}]
```


Most of the PL/SQL programs perform input and output through SQL statements to store data in database tables or query those tables. All other PL/SQL input/output is done through APIs that interact with other programs. For example, the DBMS_OUTPUT package has procedures such as PUT_LINE. To see the result outside of PL/SQL, you require another program, such as SQL*Plus, to read and display the data passed to DBMS_OUTPUT.

SQL*Plus does not display DBMS_OUTPUT data unless you first issue the SQL*Plus command SET SERVEROUTPUT ON as follows:

```
SET SERVEROUTPUT ON
```

Note

- SIZE sets the number of bytes of the output that can be buffered within the Oracle Database server. The default is UNLIMITED. *n* cannot be less than 2000 or greater than 1,000,000.
- For additional information about SERVEROUTPUT, see the *Oracle Database PL/SQL User's Guide and Reference 11g*.

Using the SQL*Plus SPOOL Command

```
SPO[OL] [file_name[.ext] [CRE[ATE] | REP[LACE] |
    APP[END]] | OFF | OUT]
```

Option	Description
file_name[.ext]	Spools output to the specified file name
CRE[ATE]	Creates a new file with the name specified
REP[LACE]	Replaces the contents of an existing file. If the file does not exist, REPLACE creates the file.
APP[END]	Adds the contents of the buffer to the end of the file you specify
OFF	Stops spooling
OUT	Stops spooling and sends the file to your computer's standard (default) printer

The SPOOL command stores query results in a file or optionally sends the file to a printer. The SPOOL command has been enhanced. You can now append to, or replace, an existing file, where previously you could use SPOOL only to create (and replace) a file. REPLACE is the default.

To spool output generated by commands in a script without displaying the output on the screen, use SET TERMOUT OFF. SET TERMOUT OFF does not affect output from commands that run interactively.

You must use quotation marks around file names containing white space. To create a valid HTML file using SPOOL APPEND commands, you must use PROMPT or a similar command to create the HTML page header and footer. The SPOOL APPEND command does not parse HTML tags. Set SQLPLUSCOMPAT[IBILITY] to 9.2 or earlier to disable the CREATE, APPEND, and SAVE parameters.

Using the AUTOTRACE Command

- It displays a report after the successful execution of SQL data manipulation statements (DML) statements such as SELECT, INSERT, UPDATE, or DELETE.
- The report can now include execution statistics and the query execution path.

```
SET AUTOT[RACE] {ON | OFF | TRACE[ONLY]} [EXP[LAIN]]
    [STAT[ISTICS]]
```

```
SET AUTOTRACE ON
-- The AUTOTRACE report includes both the optimizer
-- execution path and the SQL statement execution
-- statistics
```


EXPLAIN shows the query execution path by performing an EXPLAIN PLAN. STATISTICS displays SQL statement statistics. The formatting of your AUTOTRACE report may vary depending on the version of the server to which you are connected and the configuration of the server. The DBMS_XPLAN package provides an easy way to display the output of the EXPLAIN PLAN command in several predefined formats.

Note

- For additional information about the package and subprograms, see *Oracle Database PL/SQL Packages and Types Reference 11g*.
- For additional information about the EXPLAIN PLAN, see *Oracle Database SQL Reference 11g*.
- For additional information about Execution Plans and the statistics, see the *Oracle Database Performance Tuning Guide 11g*.

Summary

In this appendix, you should have learned how to use SQL*Plus as an environment to do the following:

- Execute SQL statements
- Edit SQL statements
- Format the output
- Interact with script files

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

D

Using JDeveloper

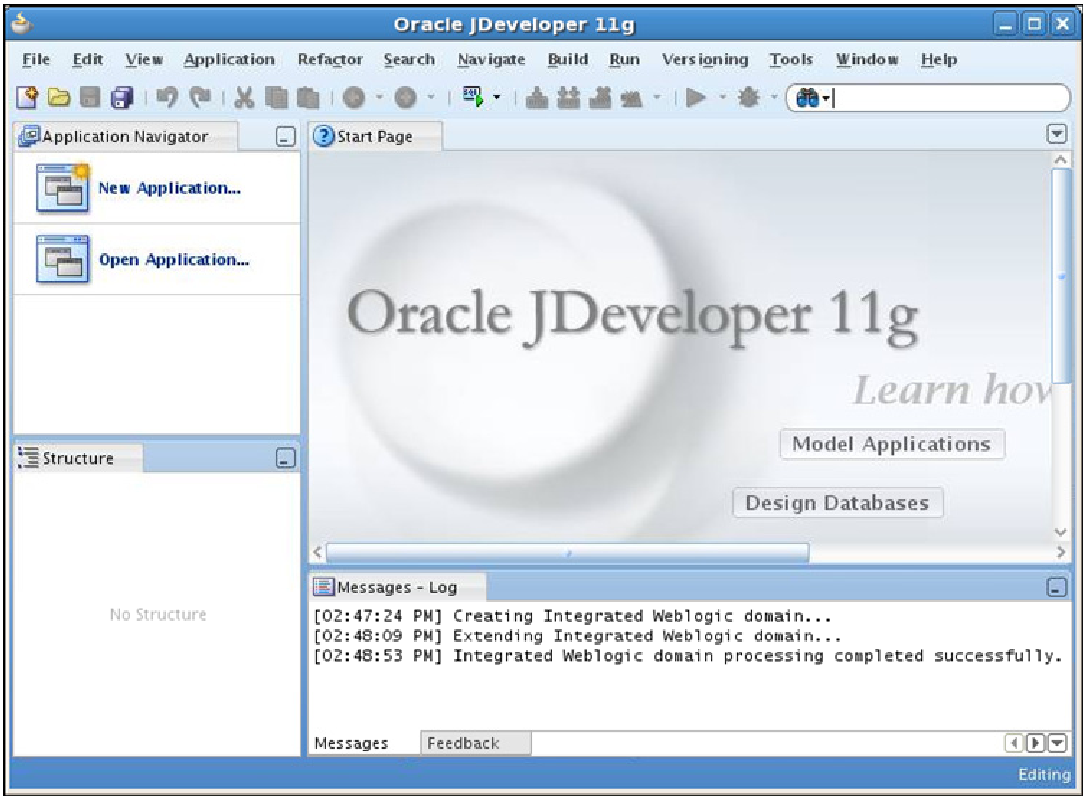
Objectives

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

- List the key features of Oracle JDeveloper
- Create a database connection in JDeveloper
- Manage database objects in JDeveloper
- Use JDeveloper to execute SQL Commands
- Create and run PL/SQL Program Units

In this appendix, you are introduced to the tool JDeveloper. You learn how to use JDeveloper for your database development tasks.

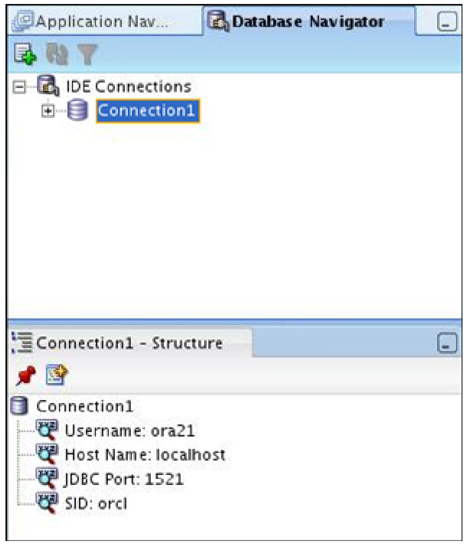
Oracle JDeveloper



Oracle JDeveloper is an integrated development environment (IDE) for developing and deploying Java applications and web services. It supports every stage of the software development life cycle (SDLC) from modeling to deploying. It has the features to use the latest industry standards for Java, XML, and SQL while developing an application.

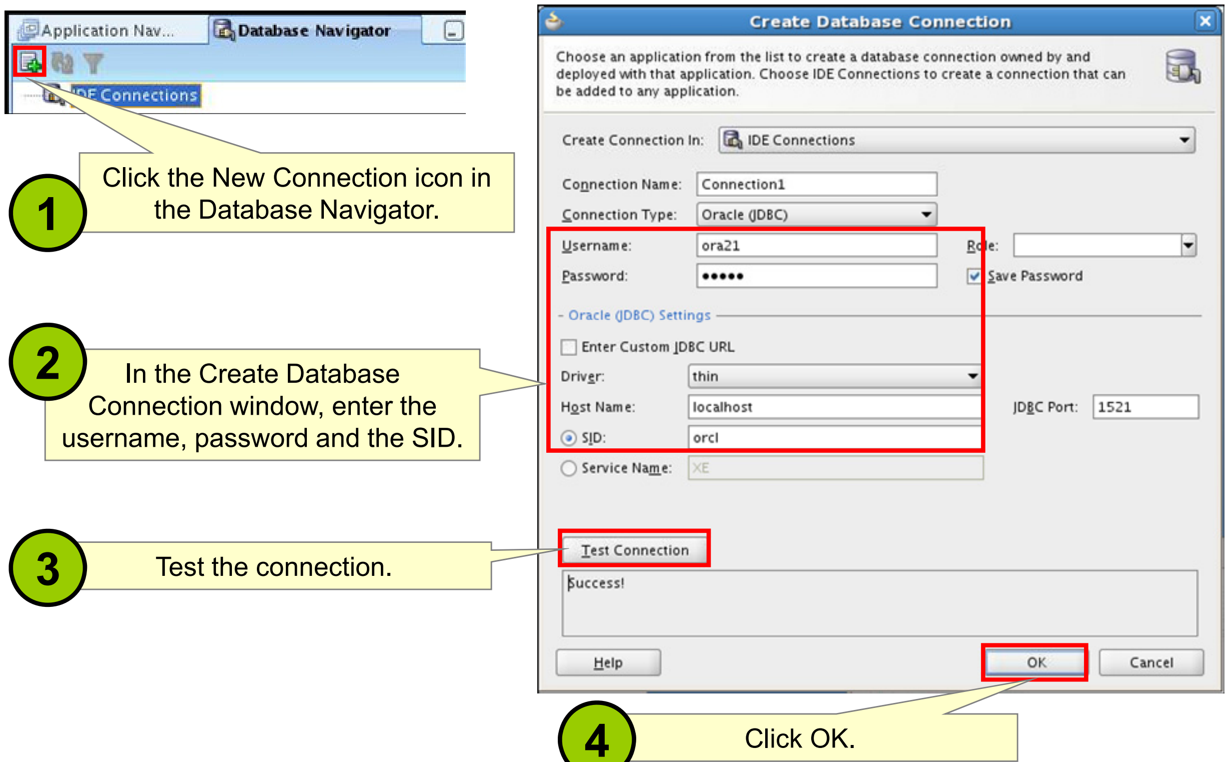
Oracle JDeveloper 11*g* initiates a new approach to J2EE development with features that enable visual and declarative development. This innovative approach makes J2EE development simple and efficient.

Database Navigator



Using Oracle JDeveloper, you can store the information necessary to connect to a database in an object called “connection.” A connection is stored as part of the IDE settings, and can be exported and imported for easy sharing among groups of users. A connection serves several purposes from browsing the database and building applications, all the way through to deployment.

Creating a Connection



A connection is an object that specifies the necessary information for connecting to a specific database as a specific user of that database. You can create and test connections for multiple databases and for multiple schemas.

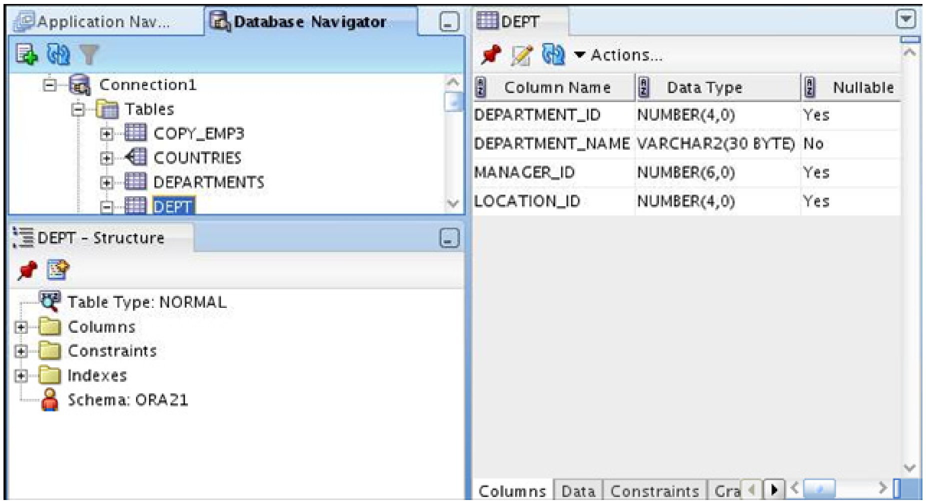
To create a database connection, perform the following steps:

1. Click the New Connection icon in the Database Navigator.
2. In the Create Database Connection window, enter the connection name. Enter the username and password of the schema that you want to connect to. Enter the SID of the database that you want to connect to.
3. Click Test to ensure that the connection has been set correctly.
4. Click OK.

Browsing Database Objects

Use the Database Navigator to:

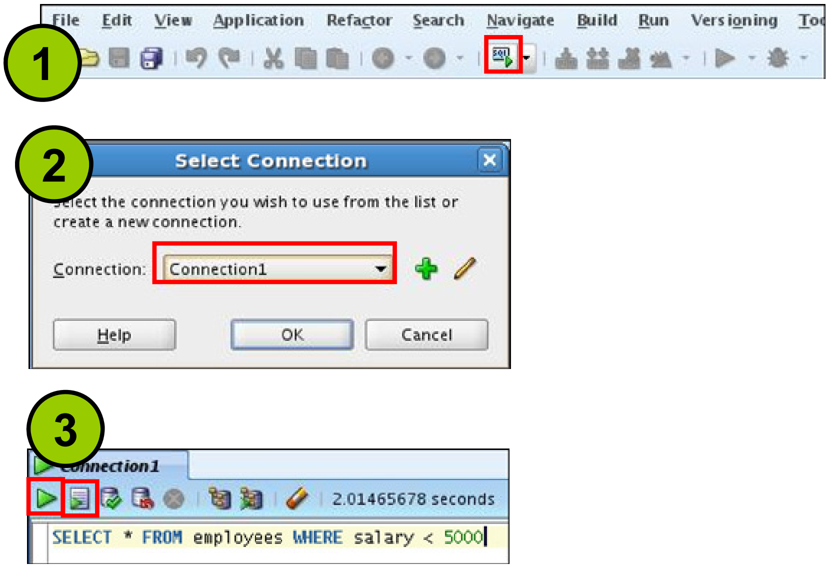
- Browse through many objects in a database schema
- Review the definitions of objects at a glance



After you create a database connection, you can use the Database Navigator to browse through many objects in a database schema including tables, views, indexes, packages, procedures, triggers, and types.

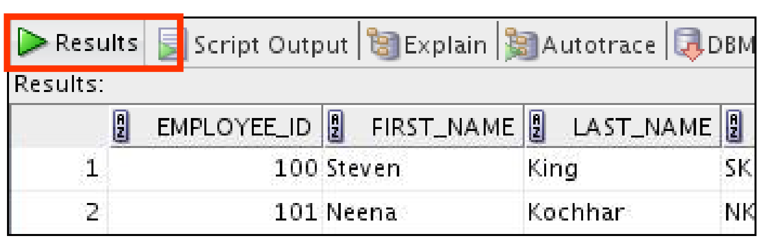
You can see object definitions broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

Executing SQL Statements

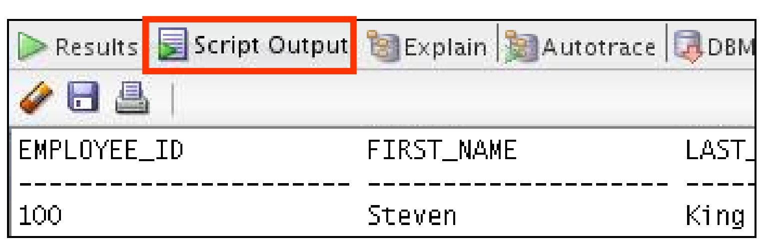


To execute a SQL statement, perform the following steps:

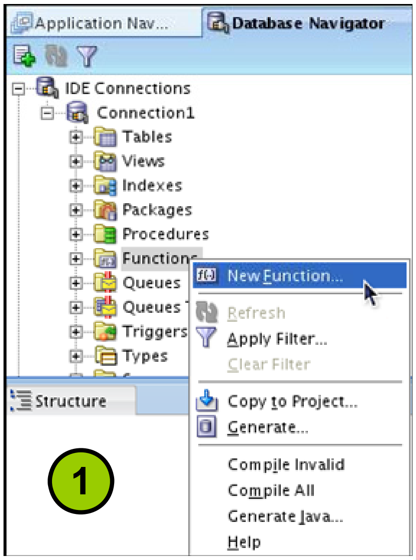
1. Click the Open SQL Worksheet icon.
2. Select the connection.
3. Execute the SQL command by clicking:
   - The **Execute statement** button or by pressing F9. The output is as follows:

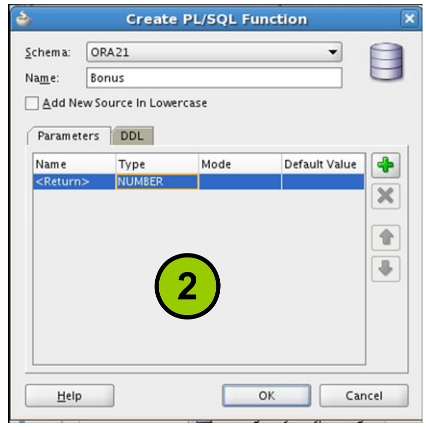


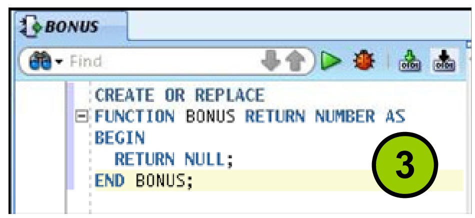
   - The **Run Script** button or by pressing F5. The output is as follows:



Creating Program Units





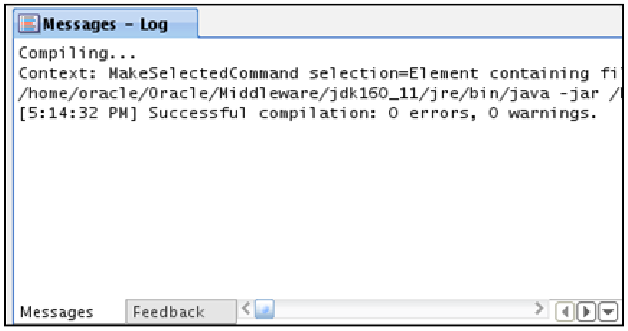


Skeleton of the function

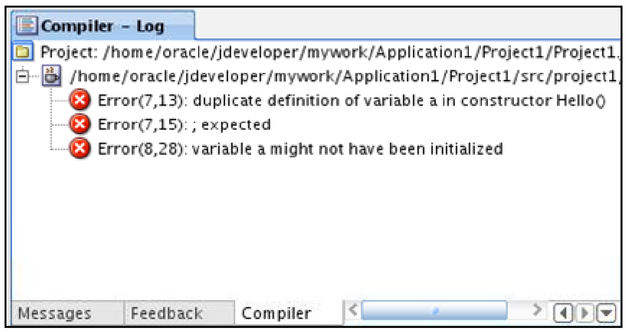
To create a PL/SQL program unit:

1. Select View > Database Navigator. Select and expand a database connection. Right-click a folder corresponding to the object type (Procedures, Packages, Functions). Select “New [Procedures|Packages|Functions]”.
2. Enter a valid name for the function, package, or procedure, and click OK.
3. A skeleton definition is created and opened in the Code Editor. You can then edit the subprogram to suit your need.

Compiling



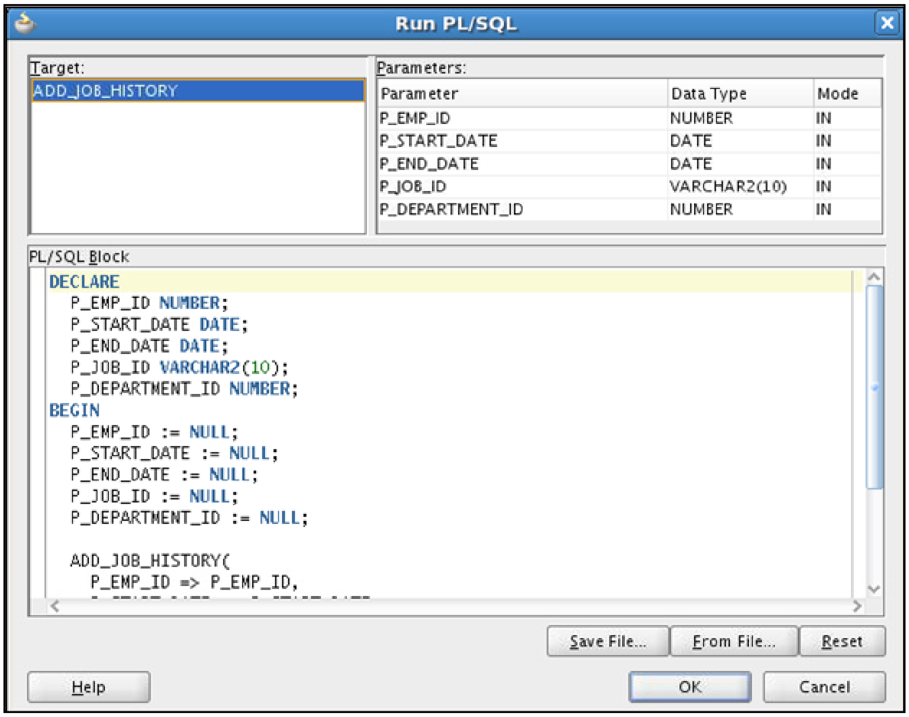
Compilation with errors



Compilation without errors

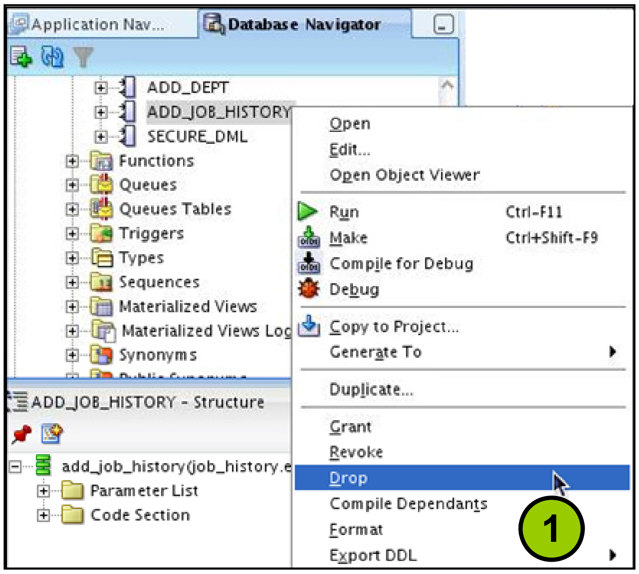
After editing the skeleton definition, you need to compile the program unit. Right-click the PL/SQL object that you need to compile in the Connection Navigator, and then select Compile. Alternatively, you can press Ctrl + Shift + F9 to compile.

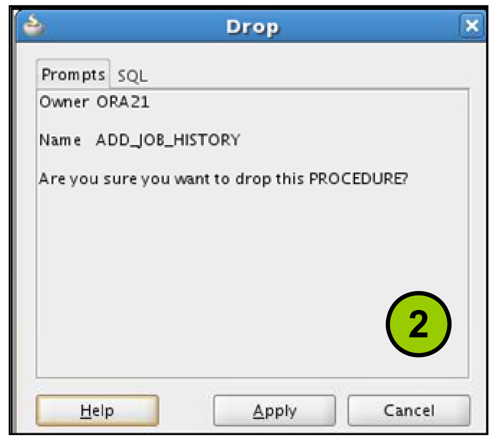
Running a Program Unit



To execute the program unit, right-click the object and select Run. The Run PL/SQL dialog box appears. You may need to change the `NULL` values with reasonable values that are passed into the program unit. After you change the values, click OK. The output is displayed in the Message-Log window.

Dropping a Program Unit

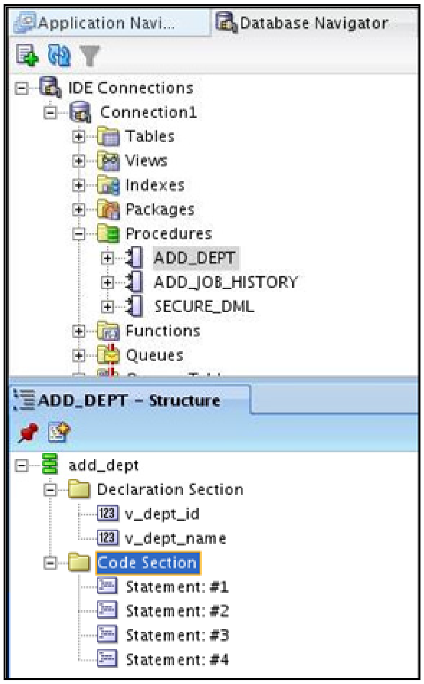


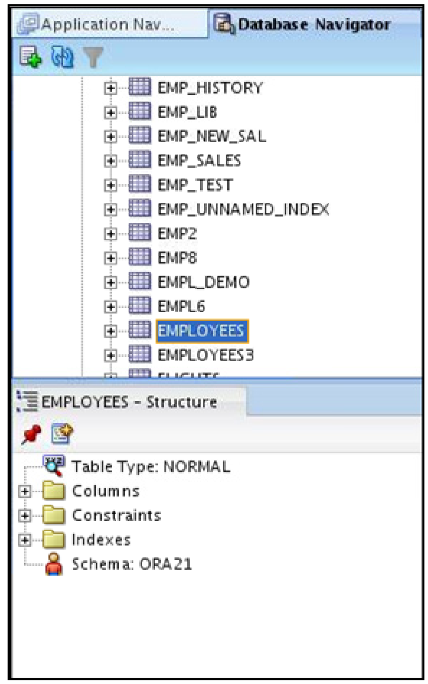


To drop a program unit:

1. Right-click the object and select Drop.
   The Drop Confirmation dialog box appears.
2. Click Apply.
   The object is dropped from the database.

Structure Window



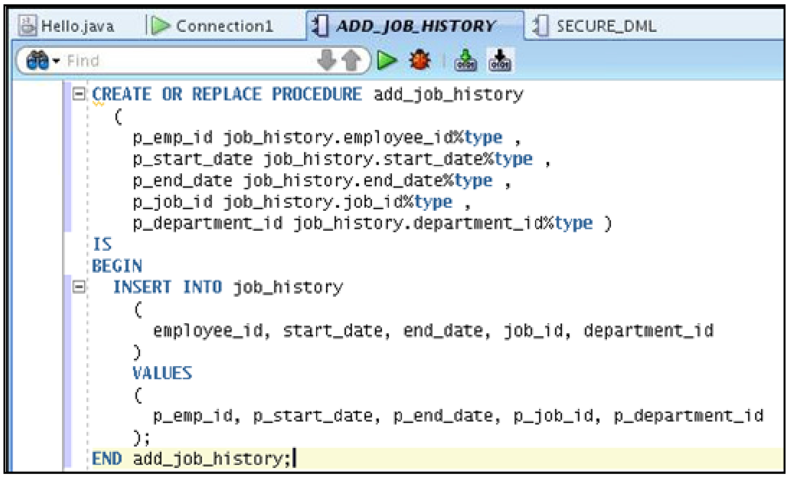


The Structure window offers a structural view of the data in the document that is currently selected in the active window of those windows that participate in providing structure: the navigators, the editors and viewers, and the Property Inspector.

In the Structure window, you can view the document data in a variety of ways. The structures that are available for display are based on document type. For a Java file, you can view code structure, UI structure, or UI model data. For an XML file, you can view XML structure, design structure, or UI model data.

The Structure window is dynamic, tracking always the current selection of the active window (unless you freeze the window's contents on a particular view), as is pertinent to the currently active editor. When the current selection is a node in the navigator, the default editor is assumed. To change the view on the structure for the current selection, select a different structure tab.

Editor Window

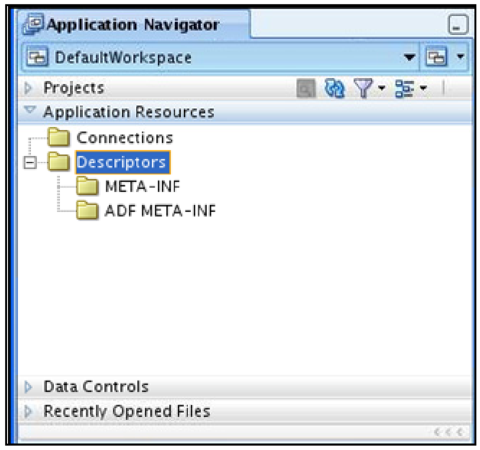


You can view your project files all in one single editor window, you can open multiple views of the same file, or you can open multiple views of different files.

The tabs at the top of the editor window are the document tabs. Selecting a document tab gives that file focus, bringing it to the foreground of the window in the current editor.

The tabs at the bottom of the editor window for a given file are the editor tabs. Selecting an editor tab opens the file in that editor.

Application Navigator

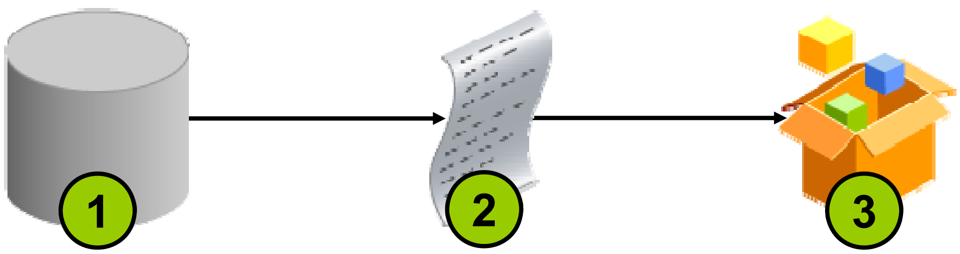


Application Navigator gives you a logical view of your application and the data that it contains. Application Navigator provides an infrastructure that the different extensions can plug in to and use to organize their data and menus in a consistent, abstract manner. While Application Navigator can contain individual files (such as Java source files), it is designed to consolidate complex data. Complex data types such as entity objects, Unified Modeling Language (UML) diagrams, Enterprise JavaBeans (EJB), or web services appear in this navigator as single nodes. The raw files that make up these abstract nodes appear in the Structure window.

Deploying Java Stored Procedures

Before deploying Java stored procedures, perform the following steps:

1. Create a database connection.
2. Create a deployment profile.
3. Deploy the objects.

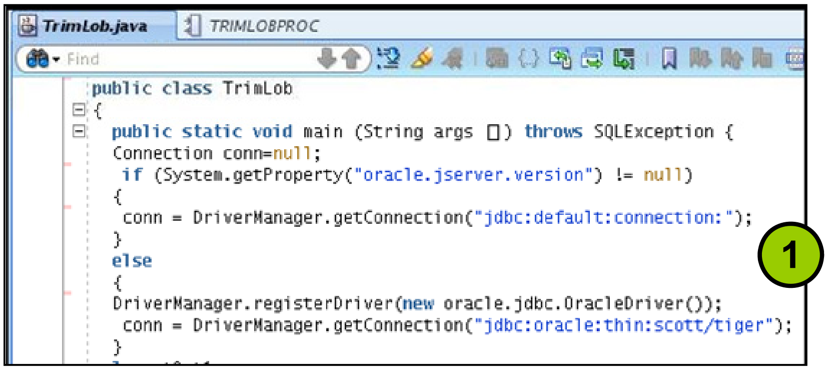


Create a deployment profile for Java stored procedures, and then deploy the classes and, optionally, any public static methods in JDeveloper by using the settings in the profile.

Deploying to the database uses the information provided in the Deployment Profile Wizard and two Oracle Database utilities:

- `loadjava` loads the Java class containing the stored procedures to an Oracle database.
- `publish` generates the PL/SQL call–specific wrappers for the loaded public static methods. Publishing enables the Java methods to be called as PL/SQL functions or procedures.

Publishing Java to PL/SQL



The slide shows the Java code and illustrates how to publish the Java code in a PL/SQL procedure.

How Can I Learn More About JDeveloper 11*g* ?

Topic	Web site
Oracle JDeveloper Product Page	http://www.oracle.com/technology/products/jdev/index.html
Oracle JDeveloper 11*g* Tutorials	http://www.oracle.com/technology/obe/obe11jdev/11/index.html
Oracle JDeveloper 11*g* Product Documentation	http://www.oracle.com/technology/documentation/jdev.html
Oracle JDeveloper 11*g* Discussion Forum	http://forums.oracle.com/forums/forum.jspa?forumID=83

Summary

In this appendix, you should have learned to do the following:

- List the key features of Oracle JDeveloper
- Create a database connection in JDeveloper
- Manage database objects in JDeveloper
- Use JDeveloper to execute SQL Commands
- Create and run PL/SQL Program Units

Objectives

In this appendix, you are introduced to the tool JDeveloper. You learn how to use JDeveloper for your database development tasks.

E

Generating Reports by Grouping Related Data

Objectives

After completing this appendix, you should be able to use the:

- `ROLLUP` operation to produce subtotal values
- `CUBE` operation to produce cross-tabulation values
- `GROUPING` function to identify the row values created by `ROLLUP` or `CUBE`
- `GROUPING SETS` to produce a single result set

In this appendix, you learn how to:

- Group data to obtain the subtotal values by using the `ROLLUP` operator
- Group data to obtain the cross-tabulation values by using the `CUBE` operator
- Use the `GROUPING` function to identify the level of aggregation in the result set produced by a `ROLLUP` or `CUBE` operator
- Use `GROUPING SETS` to produce a single result set that is equivalent to a `UNION ALL 2`

Group Functions: Review

- Group functions operate on sets of rows to give one result per group.

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

- Example:

```
SELECT  AVG(salary), STDDEV(salary),
        COUNT(commission_pct),MAX(hire_date)
FROM    employees
WHERE   job_id LIKE 'SA%';
```


You can use the `GROUP BY` clause to divide the rows in a table into groups. You can then use group functions to return summary information for each group. Group functions can appear in select lists and in `ORDER BY` and `HAVING` clauses. The Oracle server applies the group functions to each group of rows and returns a single result row for each group.

Types of group functions: Each of the group functions—`AVG`, `SUM`, `MAX`, `MIN`, `COUNT`, `STDDEV`, and `VARIANCE`—accepts one argument. The `AVG`, `SUM`, `STDDEV`, and `VARIANCE` functions operate only on numeric values. `MAX` and `MIN` can operate on numeric, character, or date data values. `COUNT` returns the number of non-NULL rows for the given expression. The example in the slide calculates the average salary, standard deviation on the salary, number of employees earning a commission, and the maximum hire date for those employees whose `JOB_ID` begins with SA.

Guidelines for Using Group Functions

- The data types for the arguments can be `CHAR`, `VARCHAR2`, `NUMBER`, or `DATE`.
- All group functions except `COUNT(*)` ignore null values. To substitute a value for null values, use the `NVL` function. `COUNT` returns either a number or zero.
- The Oracle Server implicitly sorts the result set in ascending order of the grouping columns specified, when you use a `GROUP BY` clause. To override this default ordering, you can use `DESC` in an `ORDER BY` clause.

`GROUP BY` Clause: Review

- Syntax:

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

- Example:

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


The example illustrated in the slide is evaluated by the Oracle Server as follows:

- The `SELECT` clause specifies that the following columns be retrieved:
  - Department ID and job ID columns from the `EMPLOYEES` table
  - The sum of all the salaries and the number of employees in each group that you have specified in the `GROUP BY` clause
- The `GROUP BY` clause specifies how the rows should be grouped in the table. The total salary and the number of employees are calculated for each job ID within each department. The rows are grouped by department ID and then grouped by job within each department.

HAVING Clause: Review

- Use the HAVING clause to specify which groups are to be displayed.
- You further restrict the groups on the basis of a limiting condition.

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


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

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

1. It groups rows.
2. It applies the group functions to the groups and displays the groups that match the criteria in the HAVING clause.

GROUP BY with ROLLUP and CUBE Operators

- Use ROLLUP or CUBE with GROUP BY to produce superaggregate rows by cross-referencing columns.
- ROLLUP grouping produces a result set containing the regular grouped rows and the subtotal values.
- CUBE grouping produces a result set containing the rows from ROLLUP and cross-tabulation rows.

You specify ROLLUP and CUBE operators in the GROUP BY clause of a query. ROLLUP grouping produces a result set containing the regular grouped rows and subtotal rows. The ROLLUP operator also calculates a grand total. The CUBE operation in the GROUP BY clause groups the selected rows based on the values of all possible combinations of expressions in the specification and returns a single row of summary information for each group. You can use the CUBE operator to produce cross-tabulation rows.

Note: When working with ROLLUP and CUBE, make sure that the columns following the GROUP BY clause have meaningful, real-life relationships with each other; otherwise, the operators return irrelevant information.

ROLLUP Operator

- ROLLUP is an extension to the GROUP BY clause.
- Use the ROLLUP operation to produce cumulative aggregates, such as subtotals.

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


The ROLLUP operator delivers aggregates and superaggregates for expressions within a GROUP BY statement. The ROLLUP operator can be used by report writers to extract statistics and summary information from result sets. The cumulative aggregates can be used in reports, charts, and graphs.

The ROLLUP operator creates groupings by moving in one direction, from right to left, along the list of columns specified in the GROUP BY clause. It then applies the aggregate function to these groupings.

Note

- To produce subtotals in *n* dimensions (that is, *n* columns in the GROUP BY clause) without a ROLLUP operator, *n*+1 SELECT statements must be linked with UNION ALL. This makes the query execution inefficient because each of the SELECT statements causes table access. The ROLLUP operator gathers its results with just one table access. The ROLLUP operator is useful when there are many columns involved in producing the subtotals.
- Subtotals and totals are produced with ROLLUP. CUBE produces totals as well but effectively rolls up in each possible direction, producing cross-tabular data.

ROLLUP Operator: Example

```
SELECT    department_id, job_id, SUM(salary)
FROM      employees
WHERE     department_id < 60
GROUP BY ROLLUP(department_id, job_id);
```

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	10	AD_ASST	4400
2	10	(null)	4400
3	20	MK_MAN	13000
4	20	MK_REP	6000
5	20	(null)	19000
6	30	PU_MAN	11000
7	30	PU_CLERK	13900
8	30	(null)	24900
9	40	HR_REP	6500
10	40	(null)	6500
11	50	ST_MAN	36400
12	50	SH_CLERK	64300
13	50	ST_CLERK	55700
14	50	(null)	156400
15	(null)	(null)	211200

1

2

3

ORACLE

In the example in the slide:

- Total salaries for every job ID within a department for those departments whose department ID is less than 60 are displayed by the GROUP BY clause
- The ROLLUP operator displays:
  - The total salary for each department whose department ID is less than 60
  - The total salary for all departments whose department ID is less than 60, irrespective of the job IDs

In this example, 1 indicates a group totaled by both DEPARTMENT_ID and JOB_ID, 2 indicates a group totaled only by DEPARTMENT_ID, and 3 indicates the grand total.

The ROLLUP operator creates subtotals that roll up from the most detailed level to a grand total, following the grouping list specified in the GROUP BY clause. First, it calculates the standard aggregate values for the groups specified in the GROUP BY clause (in the example, the sum of salaries grouped on each job within a department). Then it creates progressively higher-level subtotals, moving from right to left through the list of grouping columns. (In the example, the sum of salaries for each department is calculated, followed by the sum of salaries for all departments.)

- Given *n* expressions in the ROLLUP operator of the GROUP BY clause, the operation results in *n* + 1 (in this case, 2 + 1 = 3) groupings.
- Rows based on the values of the first *n* expressions are called rows or regular rows, and the others are called superaggregate rows.

CUBE Operator

- CUBE is an extension to the GROUP BY clause.
- You can use the CUBE operator to produce cross-tabulation values with a single SELECT statement.

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


The CUBE operator is an additional switch in the GROUP BY clause in a SELECT statement. The CUBE operator can be applied to all aggregate functions, including AVG, SUM, MAX, MIN, and COUNT. It is used to produce result sets that are typically used for cross-tabular reports. ROLLUP produces only a fraction of possible subtotal combinations, whereas CUBE produces subtotals for all possible combinations of groupings specified in the GROUP BY clause, and a grand total.

The CUBE operator is used with an aggregate function to generate additional rows in a result set. Columns included in the GROUP BY clause are cross-referenced to produce a superset of groups. The aggregate function specified in the select list is applied to these groups to produce summary values for the additional superaggregate rows. The number of extra groups in the result set is determined by the number of columns included in the GROUP BY clause.

In fact, every possible combination of the columns or expressions in the GROUP BY clause is used to produce superaggregates. If you have *n* columns or expressions in the GROUP BY clause, there will be 2^n possible superaggregate combinations. Mathematically, these combinations form an *n*-dimensional cube, which is how the operator got its name.

By using application or programming tools, these superaggregate values can then be fed into charts and graphs that convey results and relationships visually and effectively.

`CUBE` Operator: Example

```
SELECT    department_id, job_id, SUM(salary)
FROM      employees
WHERE     department_id < 60
GROUP BY CUBE (department_id, job_id) ;
```

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	(null)	(null)	211200
2	(null)	HR_REP	6500
3	(null)	MK_MAN	13000
4	(null)	MK_REP	6000
5	(null)	PU_MAN	11000
6	(null)	ST_MAN	36400
7	(null)	AD_ASST	4400
8	(null)	PU_CLERK	13900
9	(null)	SH_CLERK	64300
10	(null)	ST_CLERK	55700
11	10	(null)	4400
12	10	AD_ASST	4400
13	20	(null)	19000
14	20	MK_MAN	13000
15	20	MK_REP	6000
16	30	(null)	24900

1

2

3

4

The output of the `SELECT` statement in the example can be interpreted as follows:

- The total salary for every job within a department (for those departments whose department ID is less than 60)
- The total salary for each department whose department ID is less than 60
- The total salary for each job irrespective of the department
- The total salary for those departments whose department ID is less than 60, irrespective of the job titles

In this example, 1 indicates the grand total, 2 indicates the rows totaled by `JOB_ID` alone, 3 indicates some of the rows totaled by `DEPARTMENT_ID` and `JOB_ID`, and 4 indicates some of the rows totaled by `DEPARTMENT_ID` alone.

The `CUBE` operator has also performed the `ROLLUP` operation to display the subtotals for those departments whose department ID is less than 60 and the total salary for those departments whose department ID is less than 60, irrespective of the job titles. Further, the `CUBE` operator displays the total salary for every job irrespective of the department.

Note: Similar to the `ROLLUP` operator, producing subtotals in *n* dimensions (that is, *n* columns in the `GROUP BY` clause) without a `CUBE` operator requires that 2^n `SELECT` statements be linked with `UNION ALL`. Thus, a report with three dimensions requires $2^3 = 8$ `SELECT` statements to be linked with `UNION ALL`.

GROUPING Function

The GROUPING function:
- Is used with either the CUBE or ROLLUP operator
- Is used to find the groups forming the subtotal in a row
- Is used to differentiate stored NULL values from NULL values created by ROLLUP or CUBE
- Returns 0 or 1

```
SELECT      [column,] group_function(column) .. ,
            GROUPING(expr)
FROM        table
[WHERE      condition]
[GROUP BY [ROLLUP][CUBE] group_by_expression]
[HAVING     having_expression]
[ORDER BY column];
```


The GROUPING function can be used with either the CUBE or ROLLUP operator to help you understand how a summary value has been obtained.

The GROUPING function uses a single column as its argument. The *expr* in the GROUPING function must match one of the expressions in the GROUP BY clause. The function returns a value of 0 or 1.

The values returned by the GROUPING function are useful to:
- Determine the level of aggregation of a given subtotal (that is, the group or groups on which the subtotal is based)
- Identify whether a NULL value in the expression column of a row of the result set indicates:
  - A NULL value from the base table (stored NULL value)
  - A NULL value created by ROLLUP or CUBE (as a result of a group function on that expression)

A value of 0 returned by the GROUPING function based on an expression indicates one of the following:
- The expression has been used to calculate the aggregate value.
- The NULL value in the expression column is a stored NULL value.

A value of 1 returned by the GROUPING function based on an expression indicates one of the following:
- The expression has not been used to calculate the aggregate value.
- The NULL value in the expression column is created by ROLLUP or CUBE as a result of grouping.

GROUPING Function: Example

```
SELECT    department_id DEPTID, job_id JOB,
          SUM(salary),
          GROUPING(department_id) GRP_DEPT,
          GROUPING(job_id) GRP_JOB
FROM      employees
WHERE     department_id < 50
GROUP BY ROLLUP(department_id, job_id);
```

	DEPTID	JOB	SUM(SALARY)	GRP_DEPT	GRP_JOB
1	10	AD_ASST	4400	0	0
2	10	(null)	4400	0	1
3	20	MK_MAN	13000	0	0
4	20	MK_REP	6000	0	0
5	20	(null)	19000	0	1
6	30	PU_MAN	11000	0	0
7	30	PU_CLERK	13900	0	0
8	30	(null)	24900	0	1
9	40	HR_REP	6500	0	0
10	40	(null)	6500	0	1
11	(null)	(null)	54800	1	1

In the example in the slide, consider the summary value 4400 in the first row (labeled 1). This summary value is the total salary for the job ID of `AD_ASST` within department 10. To calculate this summary value, both the `DEPARTMENT_ID` and `JOB_ID` columns have been taken into account. Thus, a value of `0` is returned for both the `GROUPING(department_id)` and `GROUPING(job_id)` expressions.

Consider the summary value 4400 in the second row (labeled 2). This value is the total salary for department 10 and has been calculated by taking into account the `DEPARTMENT_ID` column; thus, a value of `0` has been returned by `GROUPING(department_id)`. Because the `JOB_ID` column has not been taken into account to calculate this value, a value of `1` has been returned for `GROUPING(job_id)`. You can observe similar output in the fifth row.

In the last row, consider the summary value 54800 (labeled 3). This is the total salary for those departments whose department ID is less than 50 and all job titles. To calculate this summary value, neither of the `DEPARTMENT_ID` and `JOB_ID` columns have been taken into account. Thus, a value of `1` is returned for both the `GROUPING(department_id)` and `GROUPING(job_id)` expressions.

GROUPING SETS

- The GROUPING SETS syntax is used to define multiple groupings in the same query.
- All groupings specified in the GROUPING SETS clause are computed and the results of individual groupings are combined with a UNION ALL operation.
- Grouping set efficiency:
  - Only one pass over the base table is required.
  - There is no need to write complex UNION statements.
  - The more elements GROUPING SETS has, the greater is the performance benefit.

GROUPING SETS is a further extension of the GROUP BY clause, which you can use to specify multiple groupings of data. Doing so facilitates efficient aggregation and, therefore, facilitates analysis of data across multiple dimensions.

A single SELECT statement can now be written using GROUPING SETS to specify various groupings (that can also include ROLLUP or CUBE operators), rather than multiple SELECT statements combined by UNION ALL operators. For example:

```
SELECT department_id, job_id, manager_id, AVG(salary)
 FROM employees
 GROUP BY
 GROUPING SETS
 ((department_id, job_id, manager_id),
 (department_id, manager_id),(job_id, manager_id));
```

This statement calculates aggregates over three groupings:

```
(department_id, job_id, manager_id), (department_id,
  manager_id)and (job_id, manager_id)
```

Without this feature, multiple queries combined together with UNION ALL are required to obtain the output of the preceding SELECT statement. A multiquery approach is inefficient because it requires multiple scans of the same data.

Compare the previous example with the following alternative:

```
SELECT department_id, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY CUBE(department_id, job_id, manager_id);
```

This statement computes all the 8 (2 *2 *2) groupings, though only the `(department_id, job_id, manager_id)`, `(department_id, manager_id)`, and `(job_id, manager_id)` groups are of interest to you.

Another alternative is the following statement:

```
SELECT department_id, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY department_id, job_id, manager_id
UNION ALL
SELECT department_id, NULL, manager_id, AVG(salary)
FROM employees
GROUP BY department_id, manager_id
UNION ALL
SELECT NULL, job_id, manager_id, AVG(salary)
FROM employees
GROUP BY job_id, manager_id;
```

This statement requires three scans of the base table, which makes it inefficient.

`CUBE` and `ROLLUP` can be thought of as grouping sets with very specific semantics and results. The following equivalencies show this fact:

`CUBE(a, b, c)` is equivalent to	`GROUPING SETS ((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), ())`
`ROLLUP(a, b,c)` is equivalent to	`GROUPING SETS ((a, b, c), (a, b),(a), ())`

GROUPING SETS: Example

```
SELECT    department_id, job_id,
          manager_id,AVG(salary)
FROM      employees
GROUP BY GROUPING SETS
((department_id,job_id), (job_id,manager_id));
```

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
1	(null)	SH_CLERK	122	3200
2	(null)	AC_MGR	101	12000
3	(null)	ST_MAN	100	7280
4	(null)	ST_CLERK	121	2675

1

...

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
39	110	AC_MGR	(null)	12000
40	90	AD_PRES	(null)	24000
41	60	IT_PROG	(null)	5760
42	100	FI_MGR	(null)	12000

2

...

ORACLE

The query in the slide calculates aggregates over two groupings. The table is divided into the following groups:

- Department ID, Job ID
- Job ID, Manager ID

The average salaries for each of these groups are calculated. The result set displays the average salary for each of the two groups.

In the output, the group marked as 1 can be interpreted as the following:

- The average salary of all employees with the `SH_CLERK` job ID under manager 122 is 3,200.
- The average salary of all employees with the `AC_MGR` job ID under manager 101 is 12,000, and so on.

The group marked as 2 in the output is interpreted as the following:

- The average salary of all employees with the `AC_MGR` job ID in department 110 is 12,000.
- The average salary of all employees with the `AD_PRES` job ID in department 90 is 24,000, and so on.

The example in the slide can also be written as:

```
SELECT department_id, job_id, NULL as manager_id,
       AVG(salary) as AVGSAL
FROM employees
GROUP BY department_id, job_id
UNION ALL
SELECT NULL, job_id, manager_id, avg(salary) as AVGSAL
FROM employees
GROUP BY job_id, manager_id;
```

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query would need two scans of the base table, EMPLOYEES. This could be very inefficient. Therefore, the usage of the GROUPING SETS statement is recommended.

Composite Columns

- A composite column is a collection of columns that are treated as a unit.

  `ROLLUP (a, (b, c) , d)`
- Use parentheses within the `GROUP BY` clause to group columns, so that they are treated as a unit while computing `ROLLUP` or `CUBE` operations.
- When used with `ROLLUP` or `CUBE`, composite columns would require skipping aggregation across certain levels.

A composite column is a collection of columns that are treated as a unit during the computation of groupings. You specify the columns in parentheses as in the following statement: `ROLLUP (a, (b, c), d)`

Here, `(b, c)` forms a composite column and is treated as a unit. In general, composite columns are useful in `ROLLUP`, `CUBE`, and `GROUPING SETS`. For example, in `CUBE` or `ROLLUP`, composite columns would require skipping aggregation across certain levels.

That is, `GROUP BY ROLLUP(a, (b, c) )` is equivalent to:

```
GROUP BY a, b, c UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

Here, `(b, c)` is treated as a unit and `ROLLUP` is not applied across `(b, c)`. It is as though you have an alias—for example, `z` as an alias for `(b, c)`, and the `GROUP BY` expression reduces to: `GROUP BY ROLLUP(a, z)`.

Note: `GROUP BY( )` is typically a `SELECT` statement with `NULL` values for the columns `a` and `b` and only the aggregate function. It is generally used for generating grand totals.

```
SELECT   NULL, NULL, aggregate_col
FROM     <table_name>
GROUP BY ( );
```

Compare this with the normal `ROLLUP` as in:

```
GROUP BY ROLLUP(a, b, c)
```

This would be:

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY a UNION ALL
GROUP BY ()
```

Similarly:

```
GROUP BY CUBE((a, b), c)
```

This would be equivalent to:

```
GROUP BY a, b, c UNION ALL
GROUP BY a, b UNION ALL
GROUP BY c UNION ALL
GROUP By ()
```

The following table shows the `GROUPING SETS` specification and the equivalent `GROUP BY` specification.

`GROUPING SETS` Statements	Equivalent `GROUP BY` Statements

`GROUP BY GROUPING SETS(a, b, c)`	`GROUP BY a UNION ALL`
	`GROUP BY b UNION ALL`
	`GROUP BY c`

`GROUP BY GROUPING SETS(a, b,(b, c))`
(The GROUPING SETS expression has a composite column.)
	`GROUP BY a UNION ALL`
	`GROUP BY b UNION ALL`
	`GROUP BY b, c`

`GROUP BY GROUPING SETS((a, b, c))`	`GROUP BY a, b, c`

`GROUP BY GROUPING SETS(a, (b), ())`	`GROUP BY a UNION ALL`
	`GROUP BY b UNION ALL`
	`GROUP BY ()`

`GROUP BY GROUPING SETS (a,ROLLUP(b, c))` (The `GROUPING SETS` expression has a composite column.)	`GROUP BY a UNION ALL` `GROUP BY ROLLUP(b, c)`

Composite Columns: Example

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

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
1	(null)	SA_REP	149	7000
2	(null)	(null)	(null)	7000
3	10	AD_ASST	101	4400
4	10	(null)	(null)	4400
5	20	MK_MAN	100	13000
6	20	MK_REP	201	6000
7	20	(null)	(null)	19000

...

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
40	100	FI_MGR	101	12000
41	100	FI_ACCOUNT	108	39600
42	100	(null)	(null)	51600
43	110	AC_MGR	101	12000
44	110	AC_ACCOUNT	205	8300
45	110	(null)	(null)	20300
46	(null)	(null)	(null)	691400

ORACLE

Consider the example:

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

This query results in the Oracle Server computing the following groupings:

- `(job_id, manager_id)`
- `(department_id, job_id, manager_id)`
- `(department_id)`
- Grand total

If you are interested only in specific groups, you cannot limit the calculation to those groupings without using composite columns. With composite columns, this is possible by treating `JOB_ID` and `MANAGER_ID` columns as a single unit while rolling up. Columns enclosed in parentheses are treated as a unit while computing `ROLLUP` and `CUBE`. This is illustrated in the example in the slide. By enclosing the `JOB_ID` and `MANAGER_ID` columns in parentheses, you indicate to the Oracle Server to treat `JOB_ID` and `MANAGER_ID` as a single unit—that is, a composite column.

The example in the slide computes the following groupings:

- `(department_id, job_id, manager_id)`
- `(department_id)`
- `( )`

The example in the slide displays the following:

- Total salary for every job and manager (labeled 1)
- Total salary for every department, job, and manager (labeled 2)
- Total salary for every department (labeled 3)
- Grand total (labeled 4)

The example in the slide can also be written as:

```
SELECT      department_id, job_id, manager_id, SUM(salary)
FROM        employees
GROUP       BY department_id,job_id, manager_id
UNION       ALL
SELECT      department_id,TO_CHAR(NULL),TO_NUMBER(NULL),
      SUM(salary)
FROM        employees
GROUP BY    department_id
UNION ALL
SELECT  TO_NUMBER(NULL), TO_CHAR(NULL),TO_NUMBER(NULL), SUM(salary)
FROM    employees
GROUP BY ();
```

In the absence of an optimizer that looks across query blocks to generate the execution plan, the preceding query would need three scans of the base table, `EMPLOYEES`. This could be very inefficient. Therefore, the use of composite columns is recommended.

Concatenated Groupings

- Concatenated groupings offer a concise way to generate useful combinations of groupings.
- To specify concatenated grouping sets, you separate multiple grouping sets, `ROLLUP` and `CUBE` operations with commas so that the Oracle Server combines them into a single `GROUP BY` clause.
- The result is a cross-product of groupings from each `GROUPING SET`.

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```


Concatenated groupings offer a concise way to generate useful combinations of groupings. The concatenated groupings are specified by listing multiple grouping sets, `CUBEs`, and `ROLLUPs`, and separating them with commas. The following is an example of concatenated grouping sets:

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

This SQL example defines the following groupings:

```
(a, c), (a, d), (b, c), (b, d)
```

Concatenation of grouping sets is very helpful for these reasons:

- **Ease of query development:** You need not manually enumerate all groupings.
- **Use by applications:** SQL generated by online analytical processing (OLAP) applications often involves concatenation of grouping sets, with each `GROUPING SET` defining groupings needed for a dimension.

Concatenated Groupings: Example

```
SELECT    department_id, job_id, manager_id,
          SUM(salary)
FROM      employees
GROUP BY department_id,
          ROLLUP(job_id),
          CUBE(manager_id);
```

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
1	(null)	SA_REP	149	7000
2	10	AD_ASST	101	4400
3	20	MK_MAN	100	13000
4	20	MK_REP	201	6000
...				
	90	AD_VP	100	34000
	90	AD_PRES	(null)	24000
...				
	(null)	SA_REP	(null)	7000
	10	AD_ASST	(null)	4400
...				
	110	(null)	101	12000
	110	(null)	205	8300
	110	(null)	(null)	20300

1

2

3

The example in the slide results in the following groupings:

- `(department_id,job_id,)` (1)
- `(department_id,manager_id)` (2)
- `(department_id)` (3)

The total salary for each of these groups is calculated.

The following is another example of a concatenated grouping.

```
SELECT department_id, job_id, manager_id, SUM(salary) totsal
FROM employees
WHERE department_id<60
GROUP BY GROUPING SETS(department_id),
GROUPING SETS (job_id, manager_id);
```

Summary

In this appendix, you should have learned how to use the:

- `ROLLUP` operation to produce subtotal values
- `CUBE` operation to produce cross-tabulation values
- `GROUPING` function to identify the row values created by `ROLLUP` or `CUBE`
- `GROUPING SETS` syntax to define multiple groupings in the same query
- `GROUP BY` clause to combine expressions in various ways:
  - Composite columns
  - Concatenated grouping sets

- `ROLLUP` and `CUBE` are extensions of the `GROUP BY` clause.
- `ROLLUP` is used to display subtotal and grand total values.
- `CUBE` is used to display cross-tabulation values.
- The `GROUPING` function enables you to determine whether a row is an aggregate produced by a `CUBE` or `ROLLUP` operator.
- With the `GROUPING SETS` syntax, you can define multiple groupings in the same query. `GROUP BY` computes all the groupings specified and combines them with `UNION ALL`.
- Within the `GROUP BY` clause, you can combine expressions in various ways:
  - To specify composite columns, you group columns within parentheses so that the Oracle Server treats them as a unit while computing `ROLLUP` or `CUBE` operations.
  - To specify concatenated grouping sets, you separate multiple grouping sets, `ROLLUP`, and `CUBE` operations with commas so that the Oracle Server combines them into a single `GROUP BY` clause. The result is a cross-product of groupings from each grouping set.

F

Hierarchical Retrieval

Objectives

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

- Interpret the concept of a hierarchical query
- Create a tree-structured report
- Format hierarchical data
- Exclude branches from the tree structure

In this appendix, you learn how to use hierarchical queries to create tree-structured reports.

Sample Data from the EMPLOYEES Table

	EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
1	100	King	AD_PRES	(null)
2	101	Kochhar	AD_VP	100
3	102	De Haan	AD_VP	100
4	103	Hunold	IT_PROG	102
5	104	Ernst	IT_PROG	103
6	107	Lorentz	IT_PROG	103
...				
16	200	Whalen	AD_ASST	101
17	201	Hartstein	MK_MAN	100
18	202	Fay	MK_REP	201
19	205	Higgins	AC_MGR	101
20	206	Gietz	AC_ACCOUNT	205

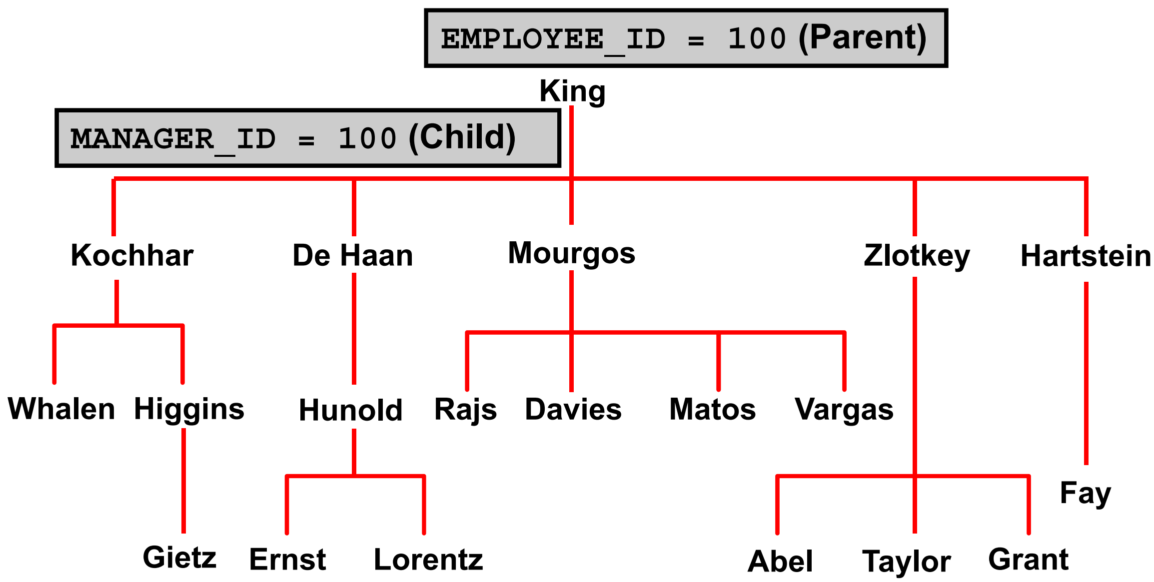
Using hierarchical queries, you can retrieve data based on a natural hierarchical relationship between the rows in a table. A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, a process called *tree walking* enables the hierarchy to be constructed. A hierarchical query is a method of reporting, with the branches of a tree in a specific order.

Imagine a family tree with the oldest members of the family found close to the base or trunk of the tree and the youngest members representing branches of the tree. Branches can have their own branches, and so on.

A hierarchical query is possible when a relationship exists between rows in a table. For example, in the slide, you see that Kochhar, De Haan, and Hartstein report to `MANAGER_ID` 100, which is King's `EMPLOYEE_ID`.

Note: Hierarchical trees are used in various fields such as human genealogy (family trees), livestock (breeding purposes), corporate management (management hierarchies), manufacturing (product assembly), evolutionary research (species development), and scientific research.

Natural Tree Structure



The `EMPLOYEES` table has a tree structure representing the management reporting line. The hierarchy can be created by looking at the relationship between equivalent values in the `EMPLOYEE_ID` and `MANAGER_ID` columns. This relationship can be exploited by joining the table to itself. The `MANAGER_ID` column contains the employee number of the employee's manager.

The parent-child relationship of a tree structure enables you to control:

- The direction in which the hierarchy is walked
- The starting point inside the hierarchy

Note: The slide displays an inverted tree structure of the management hierarchy of the employees in the `EMPLOYEES` table.

Hierarchical Queries

```
SELECT [LEVEL], column, expr...
FROM   table
[WHERE condition(s)]
[START WITH condition(s)]
[CONNECT BY PRIOR condition(s)] ;
```

condition:

```
expr comparison_operator expr
```


Keywords and Clauses

Hierarchical queries can be identified by the presence of the `CONNECT BY` and `START WITH` clauses.

In the syntax:

`SELECT`	Is the standard `SELECT` clause
`LEVEL`	For each row returned by a hierarchical query, the `LEVEL` pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on.
`FROM` *table*	Specifies the table, view, or snapshot containing the columns. You can select from only one table.
`WHERE`	Restricts the rows returned by the query without affecting other rows of the hierarchy
condition	Is a comparison with expressions
`START WITH`	Specifies the root rows of the hierarchy (where to start). This clause is required for a true hierarchical query.
`CONNECT BY PRIOR`	Specifies the columns in which the relationship between parent and child `PRIOR` rows exist. This clause is required for a hierarchical query.

Walking the Tree

Starting Point

- Specifies the condition that must be met
- Accepts any valid condition

```
START WITH column1 = value
```

Using the EMPLOYEES table, start with the employee whose last name is Kochhar.

```
...START WITH last_name = 'Kochhar'
```


The row or rows to be used as the root of the tree are determined by the START WITH clause. The START WITH clause can contain any valid condition.

Examples

Using the EMPLOYEES table, start with King, the president of the company.

```
... START WITH manager_id IS NULL
```

Using the EMPLOYEES table, start with employee Kochhar. A START WITH condition can contain a subquery.

```
... START WITH employee_id = (SELECT employee_id
                               FROM   employees
                               WHERE  last_name = 'Kochhar')
```

If the START WITH clause is omitted, the tree walk is started with all the rows in the table as root rows.

Note: The CONNECT BY and START WITH clauses are not American National Standards Institute (ANSI) SQL standard.

Walking the Tree

```
CONNECT BY PRIOR column1 = column2
```

Walk from the top down, using the EMPLOYEES table.

```
... CONNECT BY PRIOR employee_id = manager_id
```

Direction

Top down	→	Column1 = Parent Key Column2 = Child Key
Bottom up	→	Column1 = Child Key Column2 = Parent Key

The direction of the query is determined by the CONNECT BY PRIOR column placement. For top-down, the PRIOR operator refers to the parent row. For bottom-up, the PRIOR operator refers to the child row. To find the child rows of a parent row, the Oracle Server evaluates the PRIOR expression for the parent row and the other expressions for each row in the table. Rows for which the condition is true are the child rows of the parent. The Oracle Server always selects child rows by evaluating the CONNECT BY condition with respect to a current parent row.

Examples

Walk from the top down using the EMPLOYEES table. Define a hierarchical relationship in which the EMPLOYEE_ID value of the parent row is equal to the MANAGER_ID value of the child row:

```
... CONNECT BY PRIOR employee_id = manager_id
```

Walk from the bottom up using the EMPLOYEES table:

```
... CONNECT BY PRIOR manager_id = employee_id
```

The PRIOR operator does not necessarily need to be coded immediately following CONNECT BY. Thus, the following CONNECT BY PRIOR clause gives the same result as the one in the preceding example:

```
... CONNECT BY employee_id = PRIOR manager_id
```

Note: The CONNECT BY clause cannot contain a subquery.

Walking the Tree: From the Bottom Up

```
SELECT employee_id, last_name, job_id, manager_id
FROM   employees
START  WITH  employee_id = 101
CONNECT BY PRIOR manager_id = employee_id ;
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
1	101	Kochhar	AD_VP	100
2	100	King	AD_PRES	(null)

The example in the slide displays a list of managers starting with the employee whose employee ID is 101.

Walking the Tree: From the Top Down

```
SELECT   last_name||' reports to '||
PRIOR    last_name "Walk Top Down"
FROM     employees
START    WITH last_name = 'King'
CONNECT BY PRIOR employee_id = manager_id ;
```

	Walk Top Down
1	King reports to
2	King reports to
3	Kochhar reports to King
4	Greenberg reports to Kochhar
5	Faviet reports to Greenberg

...

105	Grant reports to Zlotkey
106	Johnson reports to Zlotkey
107	Hartstein reports to King
108	Fay reports to Hartstein

ORACLE

Walking from the top down, display the names of the employees and their manager. Use employee King as the starting point. Print only one column.

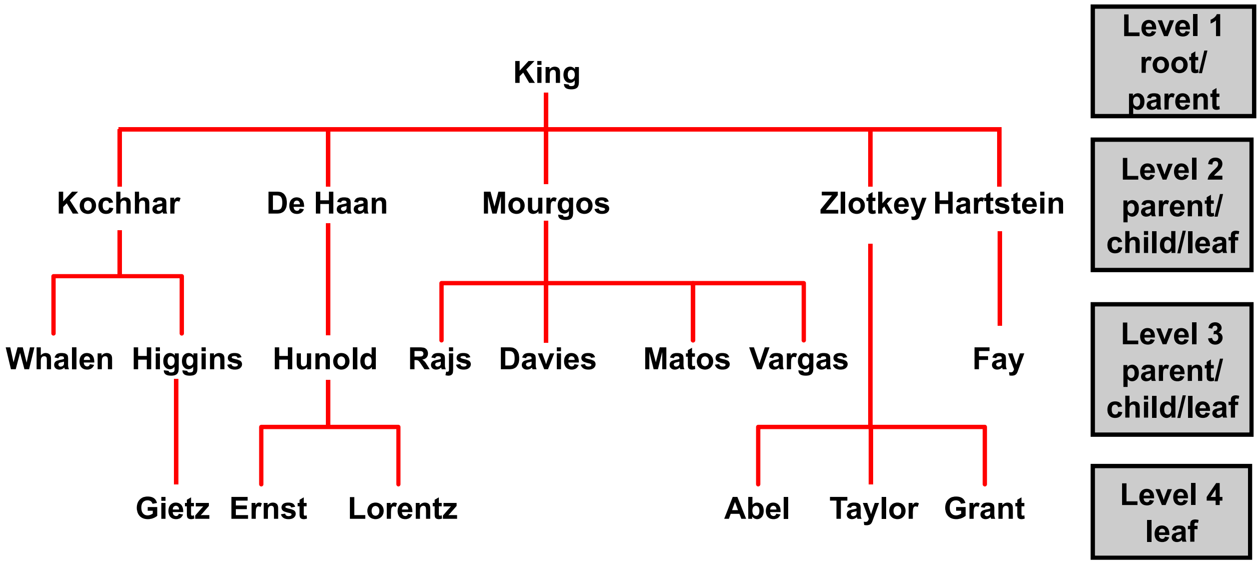
Example

In the following example, EMPLOYEE_ID values are evaluated for the parent row and MANAGER_ID and SALARY values are evaluated for the child rows. The PRIOR operator applies only to the EMPLOYEE_ID value.

```
... CONNECT BY PRIOR employee_id = manager_id
                                  AND salary > 15000;
```

To qualify as a child row, a row must have a MANAGER_ID value equal to the EMPLOYEE_ID value of the parent row and must have a SALARY value greater than $15,000.

Ranking Rows with the LEVEL Pseudocolumn



You can explicitly show the rank or level of a row in the hierarchy by using the `LEVEL` pseudocolumn. This will make your report more readable. The forks where one or more branches split away from a larger branch are called nodes, and the very end of a branch is called a leaf or leaf node. The graphic in the slide shows the nodes of the inverted tree with their `LEVEL` values. For example, employee Higgens is a parent and a child, whereas employee Davies is a child and a leaf.

LEVEL Pseudocolumn

Value	Level for Top Down	Level for Bottom up
1	A root node	A root node
2	A child of a root node	The parent of a root node
3	A child of a child, and so on	A parent of a parent, and so on

In the slide, King is the root or parent (`LEVEL = 1`). Kochhar, De Haan, Mourgos, Zlotkey, Hartstein, Higgens, and Hunold are children and also parents (`LEVEL = 2`). Whalen, Rajs, Davies, Matos, Vargas, Gietz, Ernst, Lorentz, Abel, Taylor, Grant, and Fay are children and leaves (`LEVEL = 3 and LEVEL = 4`).

Note: A *root node* is the highest node within an inverted tree. A *child node* is any nonroot node. A parent node is any node that has children. A leaf node is any node without children. The number of levels returned by a hierarchical query may be limited by available user memory.

Formatting Hierarchical Reports by Using LEVEL and LPAD

Create a report displaying company management levels, beginning with the highest level and indenting each of the following levels.

```
COLUMN org_chart FORMAT A12
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2,'_')
       AS org_chart
FROM   employees
START WITH first_name='Steven' AND last_name='King'
CONNECT BY PRIOR employee_id=manager_id
```


The nodes in a tree are assigned level numbers from the root. Use the `LPAD` function in conjunction with the `LEVEL` pseudocolumn to display a hierarchical report as an indented tree.

In the example in the slide:

- `LPAD(`*`char1`*`,`*`n`* `[,`*`char2`*`])` returns *`char1`*, left-padded to length *`n`* with the sequence of characters in *`char2`*. The argument *`n`* is the total length of the return value as it is displayed on your terminal screen.
- `LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2,'_')` defines the display format.
- *`char1`* is the `LAST_NAME`, *`n`* the total length of the return value, is length of the `LAST_NAME +(LEVEL*2)-2`, and *`char2`* is `'_'`.

That is, this tells SQL to take the `LAST_NAME` and left-pad it with the `'_'` character until the length of the resultant string is equal to the value determined by `LENGTH(last_name)+(LEVEL*2)-2`.

For King, `LEVEL = 1`. Therefore, (2 * 1) – 2 = 2 – 2 = 0. So King does not get padded with any `'_'` character and is displayed in column 1.

For Kochhar, `LEVEL = 2`. Therefore, (2 * 2) – 2 = 4 – 2 = 2. So Kochhar gets padded with 2 `'_'` characters and is displayed indented.

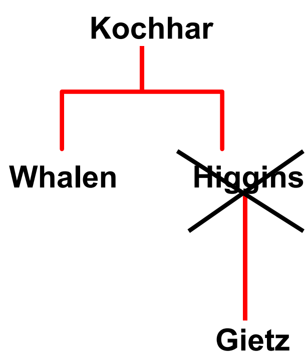
The rest of the records in the `EMPLOYEES` table are displayed similarly.

	ORG_CHART
1	King
2	__Kochhar
3	____Greenberg
4	______Faviet
5	______Chen
6	______Sciarra
7	______Urman
8	______Popp
9	____Whalen
10	____Mavris
11	____Baer
12	____Higgins
13	______Gietz
14	__De Haan
15	____Hunold
16	______Ernst
17	______Austin

Pruning Branches

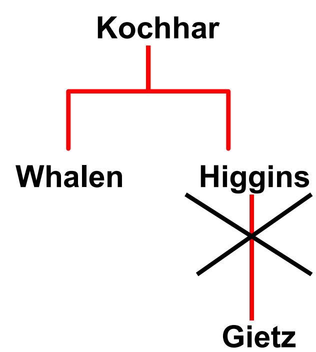
Use the `WHERE` clause to eliminate a node.

```
WHERE last_name != 'Higgins'
```



Use the `CONNECT BY` clause to eliminate a branch.

```
CONNECT BY PRIOR
employee_id = manager_id
AND last_name != 'Higgins'
```



ORACLE

You can use the `WHERE` and `CONNECT BY` clauses to prune the tree (that is, to control which nodes or rows are displayed). The predicate you use acts as a Boolean condition.

Examples

Starting at the root, walk from the top down, and eliminate employee Higgins in the result, but process the child rows.

```
SELECT   department_id, employee_id,last_name, job_id, salary
FROM     employees
WHERE    last_name  != 'Higgins'
START    WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id;
```

Starting at the root, walk from the top down, and eliminate employee Higgins and all child rows.

```
SELECT   department_id, employee_id,last_name, job_id, salary
FROM     employees
START    WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
AND      last_name != 'Higgins';
```

Summary

In this appendix, you should have learned that you can:

- Use hierarchical queries to view a hierarchical relationship between rows in a table
- Specify the direction and starting point of the query
- Eliminate nodes or branches by pruning

You can use hierarchical queries to retrieve data based on a natural hierarchical relationship between rows in a table. The `LEVEL` pseudocolumn counts how far down a hierarchical tree you have traveled. You can specify the direction of the query by using the `CONNECT BY PRIOR` clause. You can specify the starting point using the `START WITH` clause. You can use the `WHERE` and `CONNECT BY` clauses to prune the tree branches.

G

Writing Advanced Scripts

Objectives

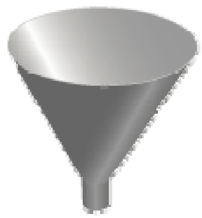
After completing this appendix, you should be able to do the following:

- Describe the type of problems that are solved by using SQL to generate SQL
- Write a script that generates a script of `DROP TABLE` statements
- Write a script that generates a script of `INSERT INTO` statements

In this appendix, you learn how to write a SQL script to generate a SQL script.

Using SQL to Generate SQL

- SQL can be used to generate scripts in SQL.
- The data dictionary is:
  - A collection of tables and views that contain database information
  - Created and maintained by the Oracle Server



SQL can be a powerful tool to generate other SQL statements. In most cases, this involves writing a script file. You can use SQL from SQL to:

- Avoid repetitive coding
- Access information from the data dictionary
- Drop or re-create database objects
- Generate dynamic predicates that contain runtime parameters

The examples used in this appendix involve selecting information from the data dictionary. The data dictionary is a collection of tables and views that contain information about the database. This collection is created and maintained by the Oracle Server. All data dictionary tables are owned by the `SYS` user. Information stored in the data dictionary includes names of Oracle Server users, privileges granted to users, database object names, table constraints, and audit information. There are four categories of data dictionary views. Each category has a distinct prefix that reflects its intended use.

Prefix	Description
`USER_`	Contains details of objects owned by the user
`ALL_`	Contains details of objects to which the user has been granted access rights, in addition to objects owned by the user
`DBA_`	Contains details of users with DBA privileges to access any object in the database
`V$_`	Stores information about database server performance and locking; available only to the DBA

Creating a Basic Script

```
SELECT 'CREATE TABLE ' || table_name ||
       '_test ' || 'AS SELECT * FROM '
       || table_name ||' WHERE 1=2;'
       AS "Create Table Script"
FROM   user_tables;
```

	Create Table Script
1	CREATE TABLE REGIONS_test AS SELECT * FROM REGIONS WHERE 1=2;
2	CREATE TABLE LOCATIONS_test AS SELECT * FROM LOCATIONS WHERE 1=2;
3	CREATE TABLE DEPARTMENTS_test AS SELECT * FROM DEPARTMENTS WHERE 1=2;
4	CREATE TABLE JOBS_test AS SELECT * FROM JOBS WHERE 1=2;
5	CREATE TABLE EMPLOYEES_test AS SELECT * FROM EMPLOYEES WHERE 1=2;
6	CREATE TABLE JOB_HISTORY_test AS SELECT * FROM JOB_HISTORY WHERE 1=2;

The example in the slide produces a report with `CREATE TABLE` statements from every table you own. Each `CREATE TABLE` statement produced in the report includes the syntax to create a table using the table name with a suffix of `_test` and having only the structure of the corresponding existing table. The old table name is obtained from the `TABLE_NAME` column of the data dictionary view `USER_TABLES`.

The next step is to enhance the report to automate the process.

Note: You can query the data dictionary tables to view various database objects that you own. The data dictionary views frequently used include:

- `USER_TABLES`: Displays description of the user's own tables
- `USER_OBJECTS`: Displays all the objects owned by the user
- `USER_TAB_PRIVS_MADE`: Displays all grants on objects owned by the user
- `USER_COL_PRIVS_MADE`: Displays all grants on columns of objects owned by the user

Controlling the Environment

```
SET ECHO OFF
SET FEEDBACK OFF
SET PAGESIZE 0

SQL statement

SET FEEDBACK ON
SET PAGESIZE 24
SET ECHO ON
```

Set system variables to appropriate values.

Set system variables back to the default value.

To execute the SQL statements that are generated, you must capture them in a file that can then be run. You must also plan to clean up the output that is generated and make sure that you suppress elements such as headings, feedback messages, top titles, and so on. In SQL Developer, you can save these statements to a script. To save the contents of the Enter SQL Statement box, click the Save icon or use the **File > Save** menu item. Alternatively, you can right-click in the Enter SQL Statement box and select the Save File option from the drop-down menu.

Note: Some of the SQL*Plus statements are not supported by SQL Worksheet. For the complete list of SQL*Plus statements that are supported, and not supported by SQL Worksheet, refer to the topic titled “SQL*Plus Statements Supported and Not Supported in SQL Worksheet” in the SQL Developer online Help.

The Complete Picture

```
SET ECHO OFF
SET FEEDBACK OFF
SET PAGESIZE 0

SELECT 'DROP TABLE ' || object_name || ';'
FROM   user_objects
WHERE  object_type = 'TABLE'
/

SET FEEDBACK ON
SET PAGESIZE 24
SET ECHO ON
```


The output of the command in the slide is saved into a file called `dropem.sql` in SQL Developer. To save the output into a file in SQL Developer, you use the Save File option under the Script Output pane. The `dropem.sql` file contains the following data. This file can now be started from SQL Developer by locating the script file, loading it, and executing it.

	'DROPTABLE'||OBJECT_NAME||';'
1	DROP TABLE REGIONS;
2	DROP TABLE COUNTRIES;
3	DROP TABLE LOCATIONS;
4	DROP TABLE DEPARTMENTS;
5	DROP TABLE JOBS;
6	DROP TABLE EMPLOYEES;
7	DROP TABLE JOB_HISTORY;
8	DROP TABLE JOB_GRADES;

Dumping the Contents of a Table to a File

```
SET HEADING OFF ECHO OFF FEEDBACK OFF
SET PAGESIZE 0


SELECT
 'INSERT INTO departments_test VALUES
  (' || department_id || ', ''' || department_name ||
   ''', ''' || location_id || ''');'
  AS "Insert Statements Script"
FROM   departments
/


SET PAGESIZE 24
SET HEADING ON ECHO ON FEEDBACK ON
```


Sometimes, it is useful to have the values for the rows of a table in a text file in the format of an `INSERT INTO VALUES` statement. This script can be run to populate the table in case the table has been dropped accidentally.

The example in the slide produces `INSERT` statements for the `DEPARTMENTS_TEST` table, captured in the `data.sql` file using the Save File option in SQL Developer.

The contents of the `data.sql` script file are as follows:

```
INSERT INTO departments_test VALUES
  (10, 'Administration', 1700);
INSERT INTO departments_test VALUES
  (20, 'Marketing', 1800);
INSERT INTO departments_test VALUES
  (50, 'Shipping', 1500);
INSERT INTO departments_test VALUES
  (60, 'IT', 1400);
...
```

Dumping the Contents of a Table to a File

Source	Result
`'''X'''`	`'X'`
`''''`	`'`
`''''‖department_name‖''''`	`'Administration'`
`''', '''`	`','`
`''');'`	`');`

You may have noticed the large number of single quotation marks in the previous slide. A set of four single quotation marks produces one single quotation mark in the final statement. Also remember that character and date values must be enclosed within quotation marks.

Within a string, to display one quotation mark, you need to prefix it with another single quotation mark. For example, in the fifth example in the slide, the surrounding quotation marks are for the entire string. The second quotation mark acts as a prefix to display the third quotation mark. Thus, the result is a single quotation mark followed by the parenthesis, followed by the semicolon.

Generating a Dynamic Predicate

```
COLUMN my_col NEW_VALUE dyn_where_clause

SELECT DECODE('&&deptno', null,
DECODE ('&&hiredate', null, ' ',
'WHERE hire_date=TO_DATE('''||'&&hiredate'',''DD-MON-YYYY'')'),
DECODE ('&&hiredate', null,
'WHERE department_id = ' || '&&deptno',
'WHERE department_id = ' || '&&deptno' ||
' AND hire_date = TO_DATE('''||'&&hiredate'',''DD-MON-YYYY'')'))
AS my_col FROM dual;
```

```
SELECT last_name FROM employees &dyn_where_clause;
```


The example in the slide generates a `SELECT` statement that retrieves data of all employees in a department who were hired on a specific day. The script generates the `WHERE` clause dynamically.

Note: After the user variable is in place, you must use the `UNDEFINE` command to delete it.

The first `SELECT` statement prompts you to enter the department number. If you do not enter any department number, the department number is treated as null by the `DECODE` function, and the user is then prompted for the hire date. If you do not enter any hire date, the hire date is treated as null by the `DECODE` function and the dynamic `WHERE` clause that is generated is also a null, which causes the second `SELECT` statement to retrieve all the rows from the `EMPLOYEES` table.

Note: The `NEW_V[ALUE]` variable specifies a variable to hold a column value. You can reference the variable in `TTITLE` commands. Use `NEW_VALUE` to display column values or the date in the top title. You must include the column in a `BREAK` command with the `SKIP PAGE` action. The variable name cannot contain a pound sign (#). `NEW_VALUE` is useful for master/detail reports in which there is a new master record for each page.

Note: Here, the hire date must be entered in the `DD-MON-YYYY` format.

The `SELECT` statement in the slide can be interpreted as follows:

```
IF   (<<deptno>> is not entered)  THEN
     IF  (<<hiredate>> is not entered)  THEN
           return empty string
     ELSE
           return the string ‘WHERE hire_date =
 TO_DATE('<<hiredate>>', 'DD-MON-YYYY')’
 ELSE
         IF (<<hiredate>> is not entered)  THEN
               return the string ‘WHERE department_id =
 <<deptno>> entered'
         ELSE
               return the string ‘WHERE department_id =
 <<deptno>> entered
                                          AND hire_date =
 TO_DATE(' <<hiredate>>', 'DD-MON-YYYY')’
END IF
```

The returned string becomes the value of the `DYN_WHERE_CLAUSE` variable, which will be used in the second `SELECT` statement.

Note: Use SQL*Plus for these examples.

When the first example in the slide is executed, the user is prompted for the values for `DEPTNO` and `HIREDATE`:

```
Enter value for deptno: 10
```

```
Enter value for hiredate: 17-SEP-1987
```

The following value for `MY_COL` is generated:

```
MY_COL
---------------------------------------------------------------------------
WHERE department_id = 10 AND hire_date = TO_DATE('27-SEP-1987','DD-MON-YYYY')
```

When the second example in the slide is executed, the following output is generated

```
LAST_NAME
-------------------------
Whalen
```

10

Summary

In this appendix, you should have learned that:

- You can write a SQL script to generate another SQL script
- Script files often use the data dictionary
- You can capture the output in a file

SQL can be used to generate SQL scripts. These scripts can be used to avoid repetitive coding, drop or re-create objects, get help from the data dictionary, and generate dynamic predicates that contain runtime parameters.

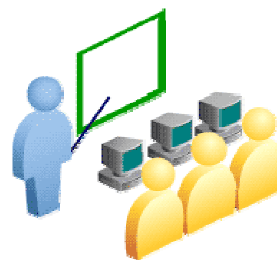
H

Oracle Database Architectural Components

Objectives

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

- List the major database architectural components
- Describe the background processes
- Explain the memory structures
- Correlate the logical and physical storage structures



This appendix provides an overview of the Oracle Database architecture. You learn about the physical and logical structures and various components of Oracle Database and their functions.

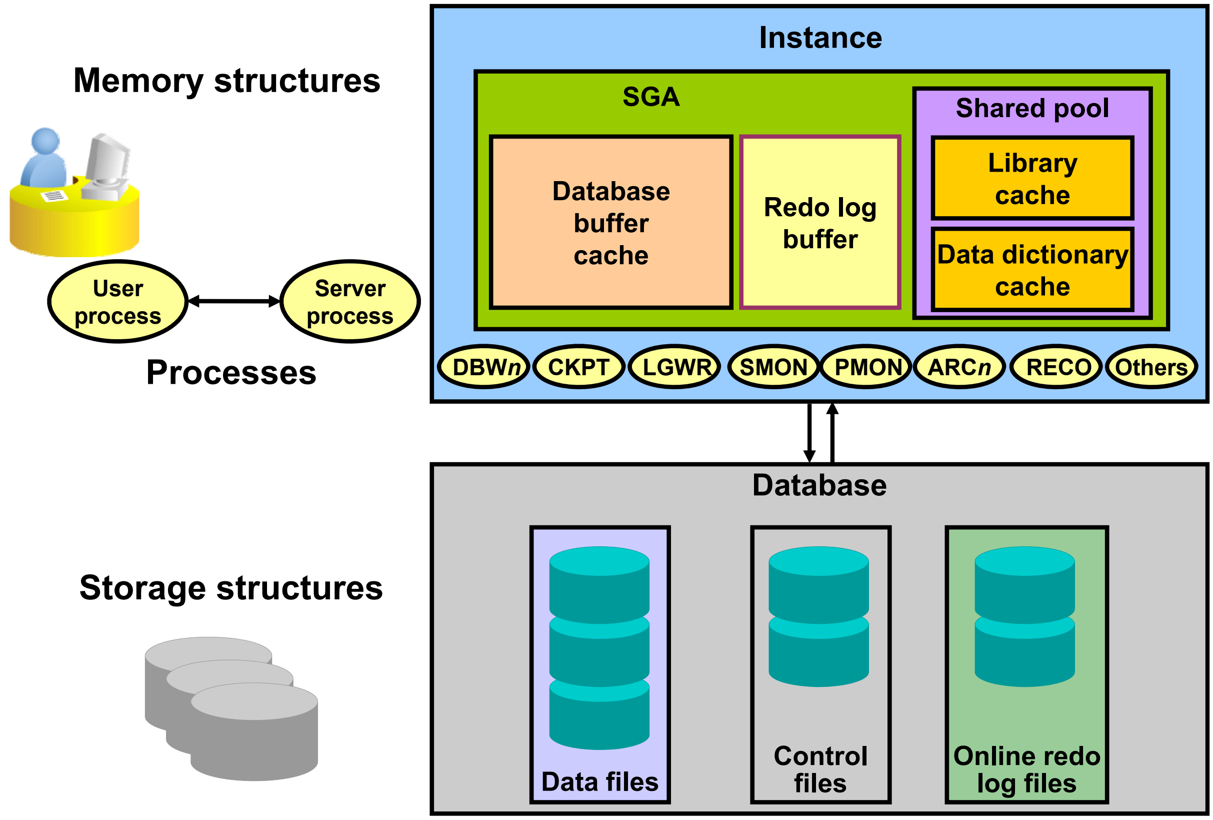
Oracle Database Architecture: Overview

The Oracle Relational Database Management System (RDBMS) is a database management system that provides an open, comprehensive, integrated approach to information management.

A database is a collection of data treated as a unit. The purpose of a database is to store and retrieve related information.

An Oracle Database reliably manages a large amount of data in a multiuser environment so that many users can concurrently access the same data. This is accomplished while delivering high performance. At the same time, it prevents unauthorized access and provides efficient solutions for failure recovery.

Oracle Database Server Structures



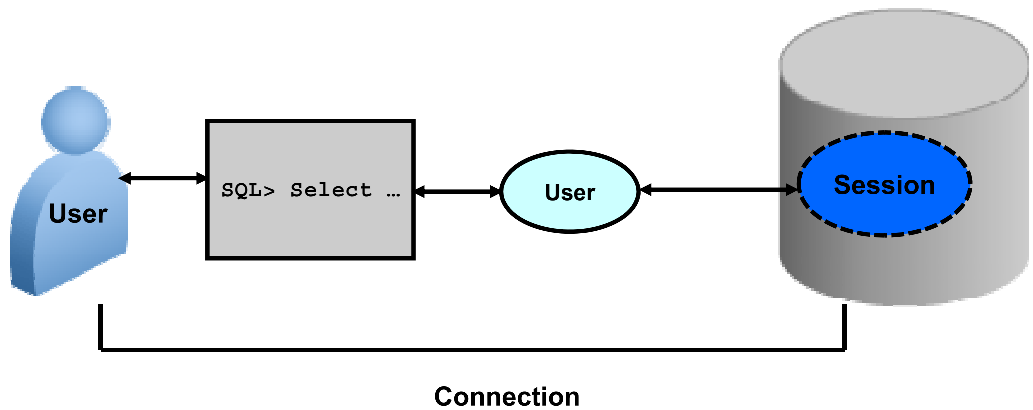
The Oracle Database consists of two main components—the instance and the database.

- The instance consists of the System Global Area (SGA), which is a collection of memory structures, and the background processes that perform tasks within the database. Every time an instance is started, the SGA is allocated and the background processes are started.
- The database consists of both physical structures and logical structures. Because the physical and logical structures are separate, the physical storage of data can be managed without affecting access to logical storage structures. The physical storage structures include:
  - The control files where the database configuration is stored
  - The redo log files that have information required for database recovery
  - The data files where all data is stored

An Oracle instance uses memory structures and processes to manage and access the database storage structures. All memory structures exist in the main memory of the computers that constitute the database server. Processes are jobs that work in the memory of these computers. A process is defined as a "thread of control" or a mechanism in an operating system that can run a series of steps.

Connecting to the Database

- Connection: Communication pathway between a user process and a database instance
- Session: A specific connection of a user to a database instance through a user process



To access information in the database, the user needs to connect to the database using a tool (such as SQL*Plus). After the user establishes connection, a session is created for the user. Connection and session are closely related to user process but are very different in meaning.

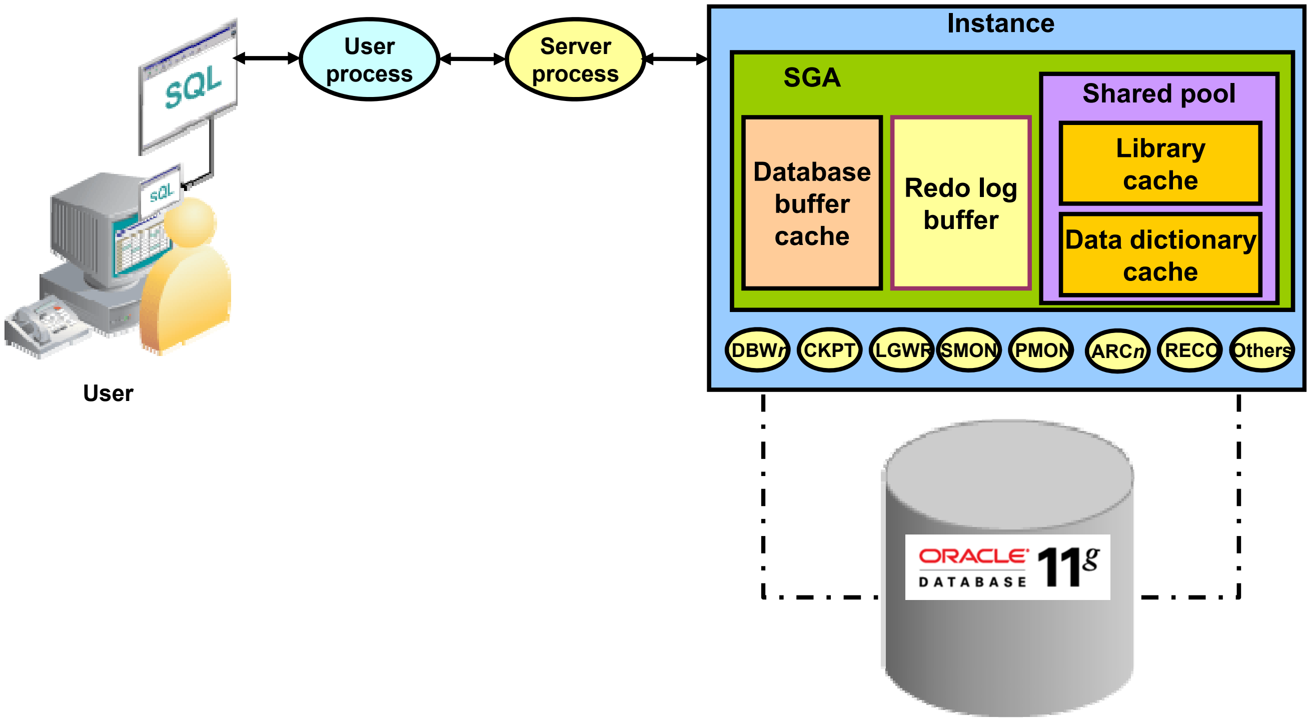
A connection is a communication pathway between a user process and an Oracle Database instance. A communication pathway is established using available interprocess communication mechanisms or network software (when different computers run the database application and Oracle Database, and communicate through a network).

A session represents the state of a current user login to the database instance. For example, when a user starts SQL*Plus, the user must provide a valid username and password, and then a session is established for that user. A session lasts from the time the user connects until the time the user disconnects or exits the database application.

In the case of a dedicated connection, the session is serviced by a permanent dedicated process. In the case of a shared connection, the session is serviced by an available server process selected from a pool, either by the middle tier or by Oracle shared server architecture.

Multiple sessions can be created and exist concurrently for a single Oracle Database user using the same username, but through different applications, or multiple invocations of the same application.

Interacting with an Oracle Database

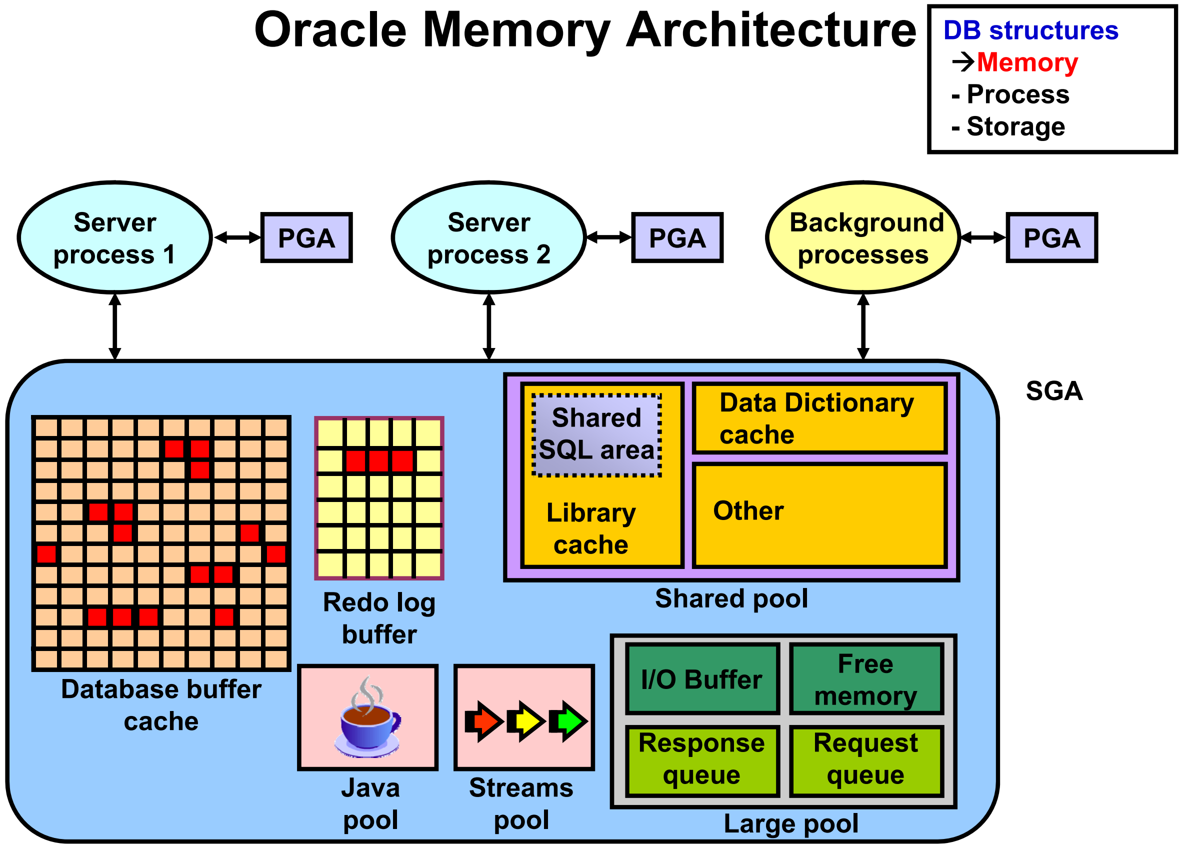


The following example describes Oracle Database operations at the most basic level. It illustrates an Oracle Database configuration where the user and associated server process are on separate computers, connected through a network.

1. An instance has started on a node where Oracle Database is installed, often called the host or database server.
2. A user starts an application spawning a user process. The application attempts to establish a connection to the server. (The connection may be local, client-server, or a three-tier connection from a middle tier.)
3. The server runs a listener that has the appropriate Oracle Net Services handler. The server detects the connection request from the application and creates a dedicated server process on behalf of the user process.
4. The user runs a DML-type SQL statement and commits the transaction. For example, the user changes the address of a customer in a table and commits the change.
5. The server process receives the statement and checks the shared pool (an SGA component) for any shared SQL area that contains a similar SQL statement. If a shared SQL area is found, the server process checks the user's access privileges to the requested data, and the existing shared SQL area is used to process the statement. If not, a new shared SQL area is allocated for the statement, so it can be parsed and processed.

6. The server process retrieves any necessary data values, either from the actual data file (in which the table is stored) or those cached in the SGA.
7. The server process modifies data in the SGA. Because the transaction is committed, the log writer process (LGWR) immediately records the transaction in the redo log file. The database writer process (DBW*n*) writes modified blocks permanently to disk when doing so is efficient.
8. If the transaction is successful, the server process sends a message across the network to the application. If it is not successful, an error message is transmitted.
9. Throughout this entire procedure, the other background processes run, watching for conditions that require intervention. In addition, the database server manages other users' transactions and prevents contention between transactions that request the same data.

Oracle Memory Architecture



Oracle Memory Structures

Oracle Database creates and uses memory structures for various purposes. For example, memory stores program code being run, data shared among users, and private data areas for each connected user.

Two basic memory structures are associated with an instance:

- The System Global Area (SGA) is a group of shared memory structures, known as SGA components, that contain data and control information for one Oracle Database instance. The SGA is shared by all server and background processes. Examples of data stored in the SGA include cached data blocks and shared SQL areas.
- The Program Global Areas (PGA) are memory regions that contain data and control information for a server or background process. A PGA is nonshared memory created by Oracle Database when a server or background process is started. Access to the PGA is exclusive to the server process. Each server process and background process has its own PGA.

The SGA is the memory area that contains data and control information for the instance. The SGA includes the following data structures:

- **Database buffer cache:** Caches blocks of data retrieved from the database
- **Redo Log buffer:** Caches redo information (used for instance recovery) until it can be written to the physical redo log files stored on the disk
- **Shared pool:** Caches various constructs that can be shared among users
- **Large pool:** Is an optional area that provides large memory allocations for certain large processes, such as Oracle backup and recovery operations, and input/output (I/O) server processes
- **Java pool:** Is used for all session-specific Java code and data within the Java Virtual Machine (JVM)
- **Streams pool:** Is used by Oracle Streams to store information required by capture and apply

When you start the instance by using Enterprise Manager or SQL*Plus, the amount of memory allocated for the SGA is displayed.

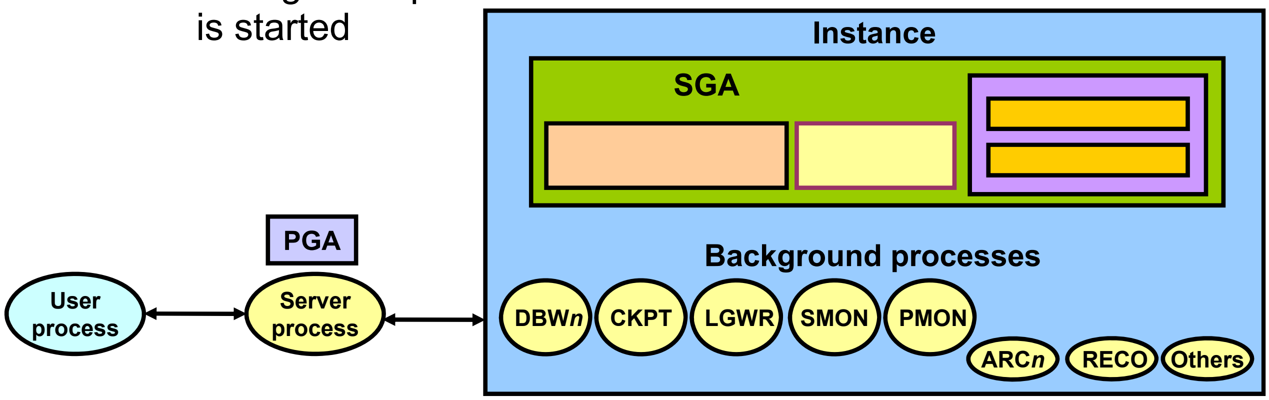
With the dynamic SGA infrastructure, the size of the database buffer cache, the shared pool, the large pool, the Java pool, and the Streams pool changes without shutting down the instance.

Oracle Database uses initialization parameters to create and configure memory structures. For example, the `SGA_TARGET` parameter specifies the total size of the SGA components. If you set `SGA_TARGET` to 0, Automatic Shared Memory Management is disabled.

Process Architecture

DB structures
- Memory
→ Process
- Storage

- User process:
  – Is started when a database user or a batch process connects to the Oracle Database
- Database processes:
  – Server process: Connects to the Oracle instance and is started when a user establishes a session
  – Background processes: Are started when an Oracle instance is started



The processes in an Oracle Database server can be categorized into two major groups:

- User processes that run the application or Oracle tool code
- Oracle Database processes that run the Oracle Database server code. These include server processes and background processes.

When a user runs an application program or an Oracle tool such as SQL*Plus, Oracle Database creates a *user process* to run the user's application. The Oracle Database also creates a *server process* to execute the commands issued by the user process. In addition, the Oracle Server also has a set of *background processes* for an instance that interact with each other and with the operating system to manage the memory structures and asynchronously perform I/O to write data to disk, and perform other required tasks.

The process structure varies for different Oracle Database configurations, depending on the operating system and the choice of Oracle Database options. The code for connected users can be configured as a dedicated server or a shared server.

- With dedicated server, for each user, the database application is run by a user process, which is served by a dedicated server process that executes Oracle Database server code.
- A shared server eliminates the need for a dedicated server process for each connection. A dispatcher directs multiple incoming network session requests to a pool of shared server processes. A shared server process serves any client request.

Server Processes

Oracle Database creates server processes to handle the requests of user processes connected to the instance. In some situations, when the application and Oracle Database operate on the same computer, it is possible to combine the user process and the corresponding server process into a single process to reduce system overhead. However, when the application and Oracle Database operate on different computers, a user process always communicates with Oracle Database through a separate server process.

Server processes created on behalf of each user's application can perform one or more of the following:

- Parse and run SQL statements issued through the application.
- Read necessary data blocks from data files on disk into the shared database buffers of the SGA, if the blocks are not already present in the SGA.
- Return results in such a way that the application can process the information.

Background Processes

To maximize performance and accommodate many users, a multiprocess Oracle Database system uses some additional Oracle Database processes called background processes. An Oracle Database instance can have many background processes.

The following background processes are required for a successful startup of the database instance:

- Database writer (DBW*n*)
- Log writer (LGWR)
- Checkpoint (CKPT)
- System monitor (SMON)
- Process monitor (PMON)

The following background processes are a few examples of optional background processes that can be started if required:

- Recoverer (RECO)
- Job queue
- Archiver (ARC*n*)
- Queue monitor (QMN*n*)

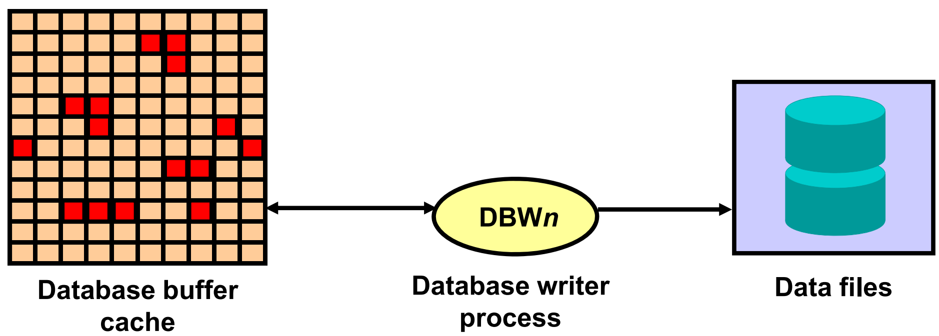
Other background processes may be found in more advanced configurations such as Real Application Clusters (RAC). See the `V$BGPROCESS` view for more information about the background processes.

On many operating systems, background processes are created automatically when an instance is started.

Database Writer Process

Writes modified (dirty) buffers in the database buffer cache to disk:

- Asynchronously while performing other processing
- Periodically to advance the checkpoint

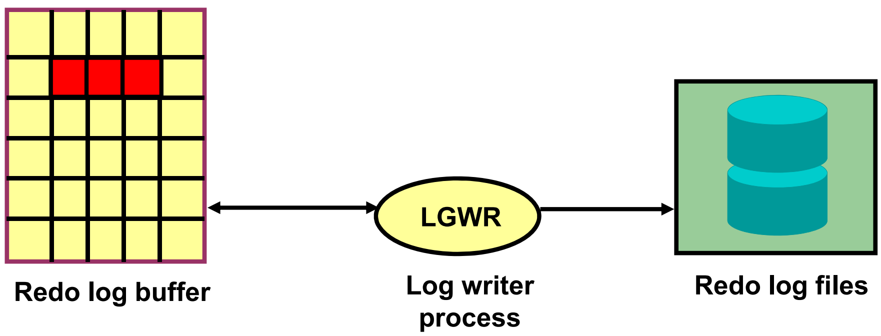


The database writer (DBW*n*) process writes the contents of buffers to data files. The DBW*n* processes are responsible for writing modified (dirty) buffers in the database buffer cache to disk. Although one database writer process (DBW0) is adequate for most systems, you can configure additional processes (DBW1 through DBW9 and DBWa through DBWj) to improve write performance if your system modifies data heavily. These additional DBW*n* processes are not useful on uniprocessor systems.

When a buffer in the database buffer cache is modified, it is marked “dirty” and is added to the LRUW list of dirty buffers that is kept in system change number (SCN) order, thereby matching the order of Redo corresponding to these changed buffers that is written to the Redo logs. When the number of available buffers in the buffer cache falls below an internal threshold such that server processes find it difficult to obtain available buffers, DBW*n* writes dirty buffers to the data files in the order that they were modified by following the order of the LRUW list.

Log Writer Process

- Writes the redo log buffer to a redo log file on disk
- LGWR writes:
  - When a process commits a transaction
  - When the redo log buffer is one-third full
  - Before a DBW*n* process writes modified buffers to disk



The log writer (LGWR) process is responsible for redo log buffer management by writing the redo log buffer entries to a redo log file on disk. LGWR writes all redo entries that have been copied into the buffer since the last time it wrote.

The redo log buffer is a circular buffer. When LGWR writes redo entries from the redo log buffer to a redo log file, server processes can then copy new entries over the entries in the redo log buffer that have been written to disk. LGWR normally writes fast enough to ensure that space is always available in the buffer for new entries, even when access to the redo log is heavy.

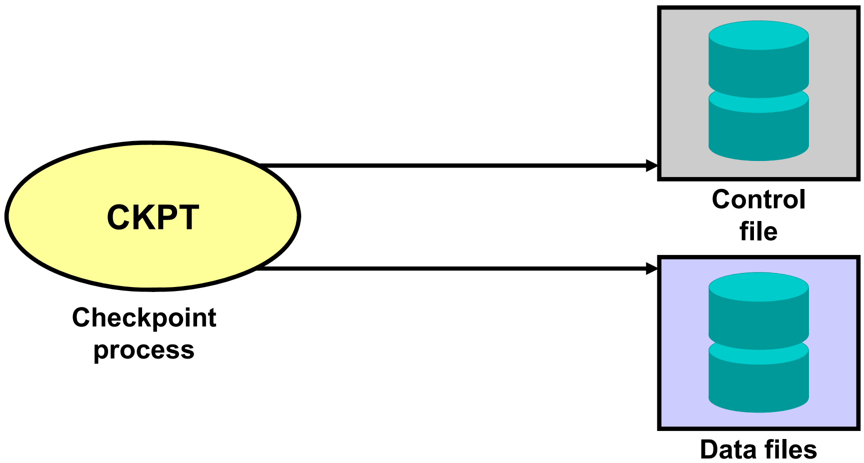
LGWR writes one contiguous portion of the buffer to disk. LGWR writes:

- When a user process commits a transaction
- When the redo log buffer is one-third full
- Before a DBW*n* process writes modified buffers to disk, if necessary

Checkpoint Process

Records checkpoint information in:

- The control file
- Each datafile header



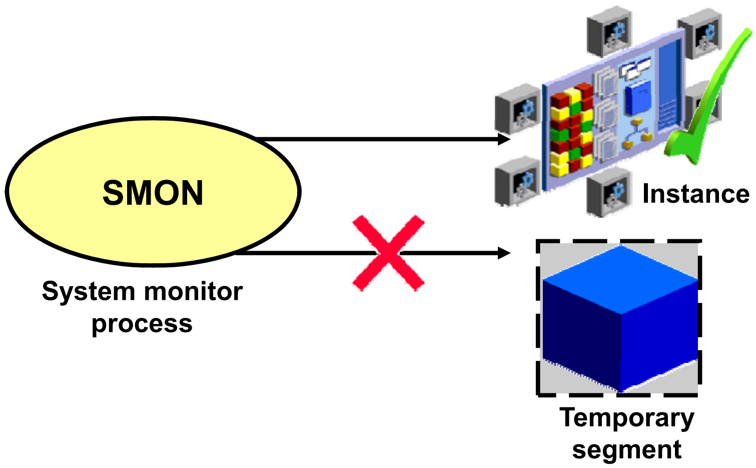
A checkpoint is a data structure that defines an SCN in the redo thread of a database. Checkpoints are recorded in the control file and each data file header, and are a crucial element of recovery.

When a checkpoint occurs, Oracle Database must update the headers of all data files to record the details of the checkpoint. This is done by the CKPT process. The CKPT process does not write blocks to disk; DBW*n* always performs that work. The SCNs recorded in the file headers guarantee that all the changes made to database blocks before that SCN have been written to disk.

The statistic DBWR checkpoints displayed by the `SYSTEM_STATISTICS` monitor in Oracle Enterprise Manager indicate the number of checkpoint requests completed.

System Monitor Process

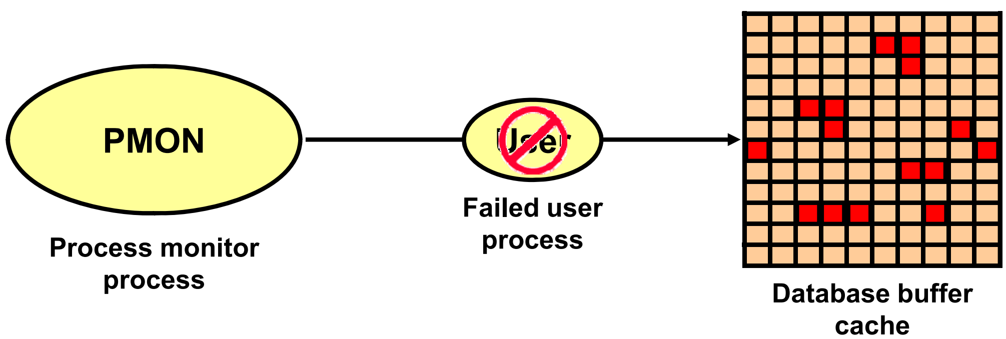
- Performs recovery at instance startup
- Cleans up unused temporary segments



The system monitor (SMON) process performs recovery, if necessary, at instance startup. SMON is also responsible for cleaning up temporary segments that are no longer in use. If any terminated transactions were skipped during instance recovery because of file-read or offline errors, SMON recovers them when the tablespace or file is brought back online. SMON checks regularly to see whether it is needed. Other processes can call SMON if they detect a need for it.

Process Monitor Process

- Performs process recovery when a user process fails:
  - Cleans up the database buffer cache
  - Frees resources used by the user process
- Monitors sessions for idle session timeout
- Dynamically registers database services with listeners

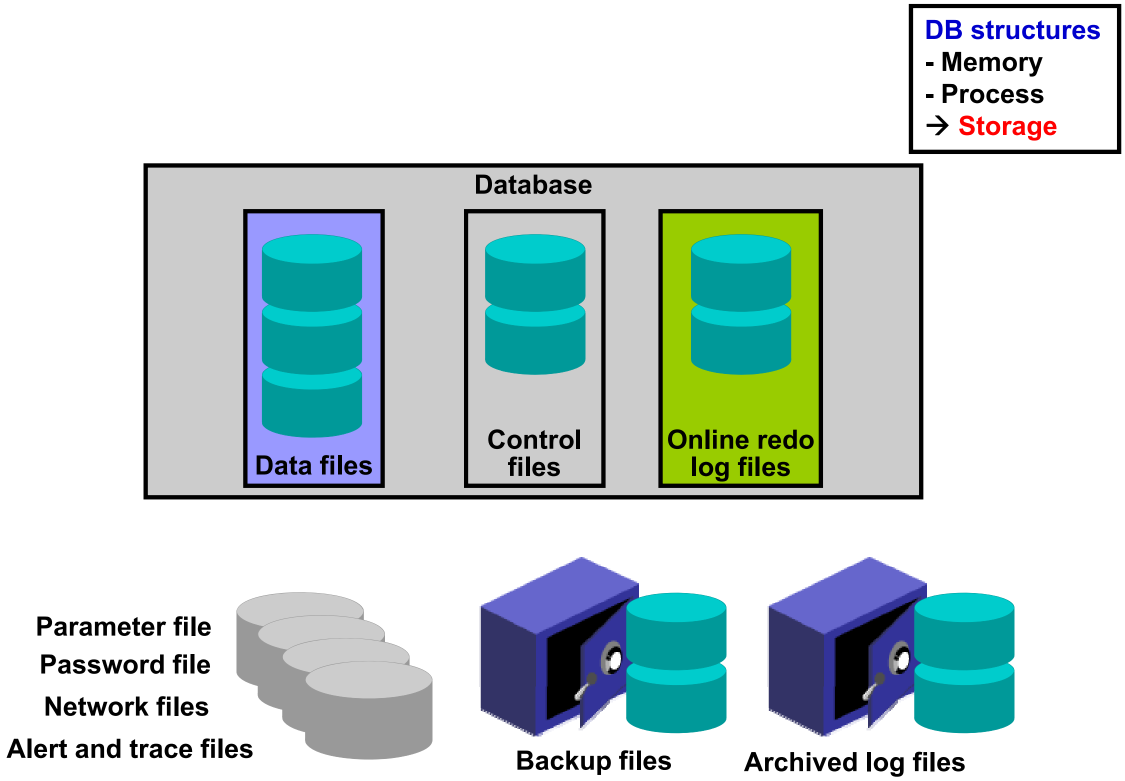


The process monitor (PMON) performs process recovery when a user process fails. PMON is responsible for cleaning up the database buffer cache and freeing resources that the user process was using. For example, it resets the status of the active transaction table, releases locks, and removes the process ID from the list of active processes.

PMON periodically checks the status of dispatcher and server processes, and restarts any that have stopped running (but not any that Oracle Database has terminated intentionally). PMON also registers information about the instance and dispatcher processes with the network listener.

Like SMON, PMON checks regularly to see whether it is needed and can be called if another process detects the need for it.

Oracle Database Storage Architecture



The files that constitute an Oracle Database are organized into the following:

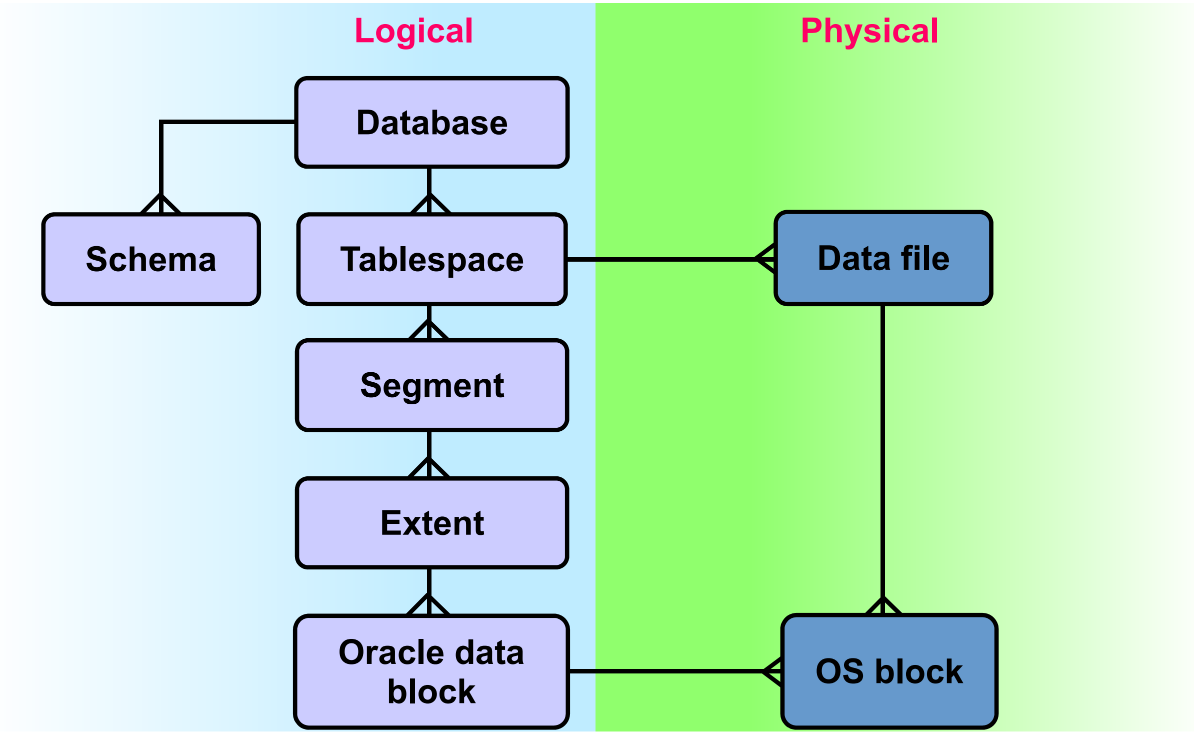
- **Control files:** Contain data about the database itself (that is, physical database structure information). These files are critical to the database. Without them, you cannot open data files to access the data within the database.
- **Data files:** Contain the user or application data of the database, as well as metadata and the data dictionary
- **Online redo log files:** Allow for instance recovery of the database. If the database server crashes and does not lose any data files, the instance can recover the database with the information in these files.

The following additional files are important to the successful running of the database:

- **Backup files:** Are used for database recovery. You typically restore a backup file when a media failure or user error has damaged or deleted the original file.
- **Archived log files:** Contain an ongoing history of the data changes (redo) that are generated by the instance. Using these files and a backup of the database, you can recover a lost data file. That is, archive logs enable the recovery of restored data files.
- **Parameter file:** Is used to define how the instance is configured when it starts up
- **Password file:** Allows `sysdba/sysoper/sysasm` to connect remotely to the database and perform administrative tasks

- **Network files:** Are used for starting the database listener and store information required for user connections
- **Trace files:** Each server and background process can write to an associated trace file. When an internal error is detected by a process, the process dumps information about the error to its trace file. Some of the information written to a trace file is intended for the database administrator, whereas other information is for Oracle Support Services.
- **Alert log files:** These are special trace entries. The alert log of a database is a chronological log of messages and errors. Each instance has one alert log file. Oracle recommends that you review this alert log periodically.

Logical and Physical Database Structures



An Oracle Database has logical and physical storage structures.

Tablespaces

A database is divided into logical storage units called tablespaces, which group related logical structures together. For example, tablespaces commonly group all of an application's objects to simplify some administrative operations. You may have a tablespace for application data and an additional one for application indexes.

Databases, Tablespaces, and Data Files

The relationship among databases, tablespaces, and data files is illustrated in the slide. Each database is logically divided into one or more tablespaces. One or more data files are explicitly created for each tablespace to physically store the data of all logical structures in a tablespace. If it is a `TEMPORARY` tablespace, instead of a data file, the tablespace has a temporary file.

Schemas

A schema is a collection of database objects that are owned by a database user. Schema objects are the logical structures that directly refer to the database's data. Schema objects include such structures as tables, views, sequences, stored procedures, synonyms, indexes, clusters, and database links. In general, schema objects include everything that your application creates in the database.

Data Blocks

At the finest level of granularity, an Oracle Database's data is stored in data blocks. One data block corresponds to a specific number of bytes of physical database space on the disk. A data block size is specified for each tablespace when it is created. A database uses and allocates free database space in Oracle data blocks.

Extents

The next level of logical database space is called an extent. An extent is a specific number of contiguous data blocks (obtained in a single allocation) that are used to store a specific type of information.

Segments

The level of logical database storage above an extent is called a segment. A segment is a set of extents allocated for a certain logical structure. For example, the different types of segments include:

- **Data segments:** Each nonclustered, non-indexed-organized table has a data segment with the exception of external tables, global temporary tables, and partitioned tables, where each table has one or more segments. All of the table's data is stored in the extents of its data segment. For a partitioned table, each partition has a data segment. Each cluster has a data segment. The data of every table in the cluster is stored in the cluster's data segment.
- **Index segments:** Each index has an index segment that stores all of its data. For a partitioned index, each partition has an index segment.
- **Undo segments:** One `UNDO` tablespace is created per database instance that contains numerous undo segments to temporarily store *undo* information. The information in an undo segment is used to generate read-consistent database information and, during database recovery, to roll back uncommitted transactions for users.
- **Temporary segments:** Temporary segments are created by the Oracle Database when a SQL statement needs a temporary work area to complete execution. When the statement finishes execution, the temporary segment's extents are returned to the instance for future use. Specify a default temporary tablespace for every user or a default temporary tablespace, which is used database-wide.

The Oracle Database dynamically allocates space. When the existing extents of a segment are full, additional extents are added. Because extents are allocated as needed, the extents of a segment may or may not be contiguous on the disk.

Processing a SQL Statement

- Connect to an instance using:
  - The user process
  - The server process
- The Oracle Server components that are used depend on the type of SQL statement:
  - Queries return rows.
  - Data manipulation language (DML) statements log changes.
  - Commits ensure transaction recovery.
- Some Oracle Server components do not participate in SQL statement processing.

Not all the components of an Oracle instance are used to process SQL statements. The user and server processes are used to connect a user to an Oracle instance. These processes are not part of the Oracle instance, but are required to process a SQL statement.

Some of the background processes, SGA structures, and database files are used to process SQL statements. Depending on the type of SQL statement, different components are used:

- Queries require additional processing to return rows to the user.
- DML statements require additional processing to log the changes made to the data.
- Commit processing ensures that the modified data in a transaction can be recovered.

Some required background processes do not directly participate in processing a SQL statement, but are used to improve performance and to recover the database. For example, the optional Archiver background process, ARC*n*, is used to ensure that a production database can be recovered.

Processing a Query

- Parse:
  - Search for an identical statement.
  - Check the syntax, object names, and privileges.
  - Lock the objects used during parse.
  - Create and store the execution plan.
- Execute: Identify the rows selected.
- Fetch: Return the rows to the user process.

Queries are different from other types of SQL statements because, if successful, they return data as results. Other statements simply return success or failure, whereas a query can return one row or thousands of rows.

There are three main stages in the processing of a query:

- Parse
- Execute
- Fetch

During the *parse* stage, the SQL statement is passed from the user process to the server process, and a parsed representation of the SQL statement is loaded into a shared SQL area.

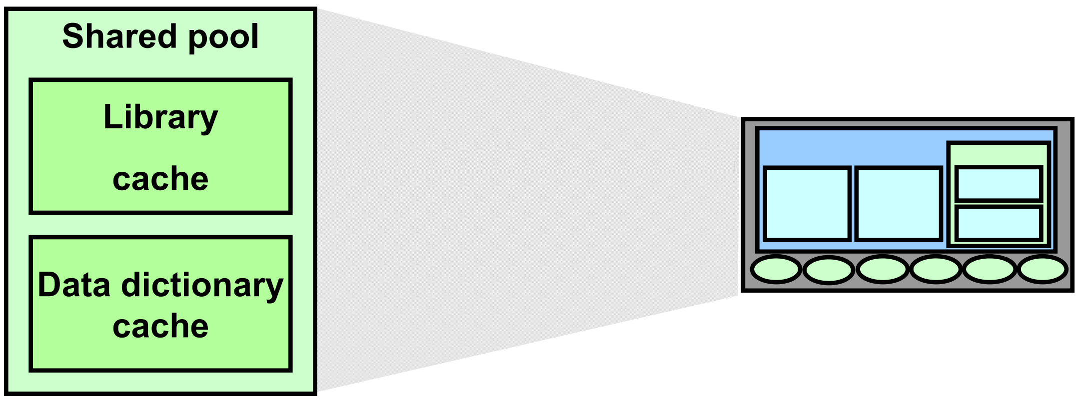
During parse, the server process performs the following functions:

- Searches for an existing copy of the SQL statement in the shared pool
- Validates the SQL statement by checking its syntax
- Performs data dictionary lookups to validate table and column definitions

The execute stage executes the statement using the best optimizer approach and the fetch retrieves the rows back to the user.

Shared Pool

- The library cache contains the SQL statement text, parsed code, and execution plan.
- The data dictionary cache contains table, column, and other object definitions and privileges.
- The shared pool is sized by `SHARED_POOL_SIZE`.



During the parse stage, the server process uses the area in the SGA known as the shared pool to compile the SQL statement. The shared pool has two primary components:

- Library cache
- Data dictionary cache

Library Cache

The library cache stores information about the most recently used SQL statements in a memory structure called a shared SQL area. The shared SQL area contains:

- The text of the SQL statement
- The parse tree, which is a compiled version of the statement
- The execution plan, with steps to be taken when executing the statement

The optimizer is the function in the Oracle Server that determines the optimal execution plan.

If a SQL statement is re-executed and a shared SQL area already contains the execution plan for the statement, the server process does not need to parse the statement. The library cache improves the performance of applications that reuse SQL statements by reducing parse time and memory requirements. If the SQL statement is not reused, it is eventually aged out of the library cache.

Data Dictionary Cache

The data dictionary cache, also known as the dictionary cache or row cache, is a collection of the most recently used definitions in the database. It includes information about database files, tables, indexes, columns, users, privileges, and other database objects.

During the parse phase, the server process looks for the information in the dictionary cache to resolve the object names specified in the SQL statement and to validate the access privileges. If necessary, the server process initiates the loading of this information from the data files.

Sizing the Shared Pool

The size of the shared pool is specified by the `SHARED_POOL_SIZE` initialization parameter.

Database Buffer Cache

- The database buffer cache stores the most recently used blocks.
- The size of a buffer is based on `DB_BLOCK_SIZE`.
- The number of buffers is defined by `DB_BLOCK_BUFFERS`.

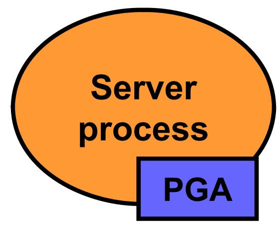
When a query is processed, the server process looks in the database buffer cache for any blocks it needs. If the block is not found in the database buffer cache, the server process reads the block from the data file and places a copy in the buffer cache. Because subsequent requests for the same block may find the block in memory, the requests may not require physical reads. The Oracle Server uses a least-recently-used algorithm to age out buffers that have not been accessed recently to make room for new blocks in the buffer cache.

Sizing the Database Buffer Cache

The size of each buffer in the buffer cache is equal to the size of an Oracle block, and it is specified by the `DB_BLOCK_SIZE` parameter. The number of buffers is equal to the value of the `DB_BLOCK_BUFFERS` parameter.

Program Global Area (PGA)

- Is not shared
- Is writable only by the server process
- Contains:
  - Sort area
  - Session information
  - Cursor state
  - Stack space



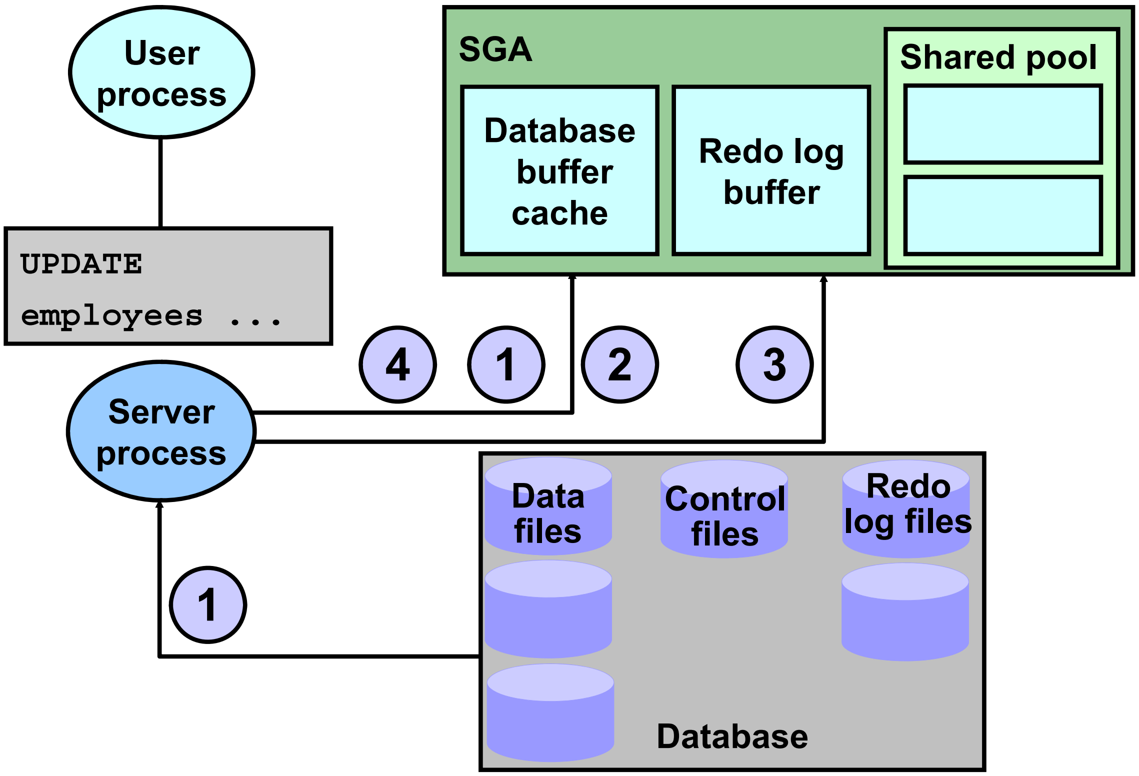
A Program Global Area (PGA) is a memory region that contains data and control information for a server process. It is a nonshared memory created by Oracle when a server process is started. Access to it is exclusive to that server process, and is read and written only by the Oracle Server code acting on behalf of it. The PGA memory allocated by each server process attached to an Oracle instance is referred to as the aggregated PGA memory allocated by the instance.

In a dedicated server configuration, the PGA of the server includes the following components:

- **Sort area:** Is used for any sorts that may be required to process the SQL statement
- **Session information:** Includes user privileges and performance statistics for the session
- **Cursor state:** Indicates the stage in the processing of the SQL statements that are currently used by the session
- **Stack space:** Contains other session variables

The PGA is allocated when a process is created, and deallocated when the process is terminated.

Processing a DML Statement



A data manipulation language (DML) statement requires only two phases of processing:

- Parse is the same as the parse phase used for processing a query.
- Execute requires additional processing to make data changes.

DML Execute Phase

To execute a DML statement:

- If the data and rollback blocks are not already in the buffer cache, the server process reads them from the data files into the buffer cache
- The server process places locks on the rows that are to be modified
- In the redo log buffer, the server process records the changes to be made to the rollback and data blocks
- The rollback block changes record the values of the data before it is modified. The rollback block is used to store the "before image" of the data, so that the DML statements can be rolled back if necessary.
- The data block changes record the new values of the data

The server process records the “before image” to the rollback block and updates the data block. Both of these changes are done in the database buffer cache. Any changed blocks in the buffer cache are marked as dirty buffers (that is, buffers that are not the same as the corresponding blocks on the disk).

The processing of a `DELETE` or `INSERT` command uses similar steps. The “before image” for a `DELETE` contains the column values in the deleted row, and the “before image” of an `INSERT` contains the row location information.

Because the changes made to the blocks are only recorded in memory structures and are not written immediately to disk, a computer failure that causes the loss of the SGA can also lose these changes.

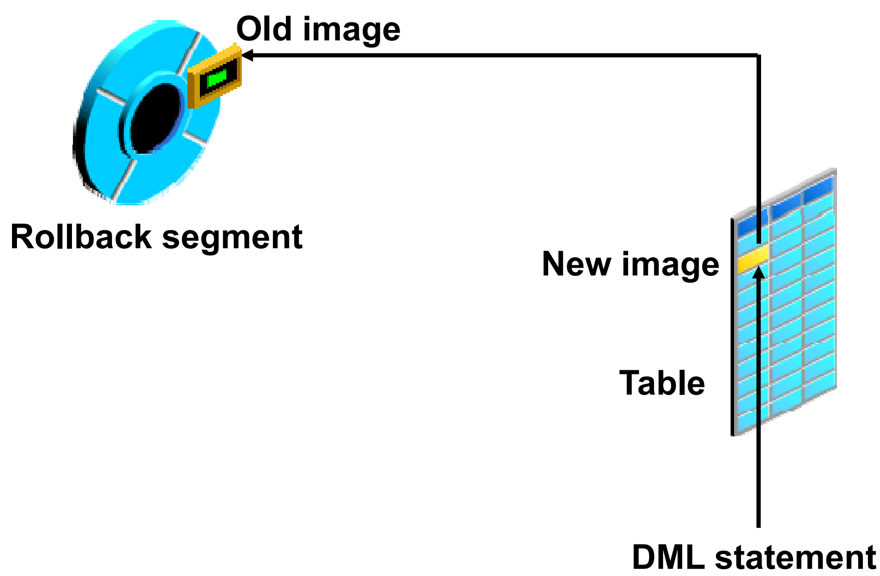
Redo Log Buffer

- Has its size defined by `LOG_BUFFER`
- Records changes made through the instance
- Is used sequentially
- Is a circular buffer

The server process records most of the changes made to data file blocks in the redo log buffer, which is a part of the SGA. The redo log buffer has the following characteristics:

- Its size in bytes is defined by the `LOG_BUFFER` parameter.
- It records the block that is changed, the location of the change, and the new value in a redo entry. A redo entry makes no distinction between the types of blocks that are changed; it only records which bytes are changed in the block.
- The redo log buffer is used sequentially, and changes made by one transaction may be interleaved with changes made by other transactions.
- It is a circular buffer that is reused after it is filled, but only after all the old redo entries are recorded in the redo log files.

Rollback Segment



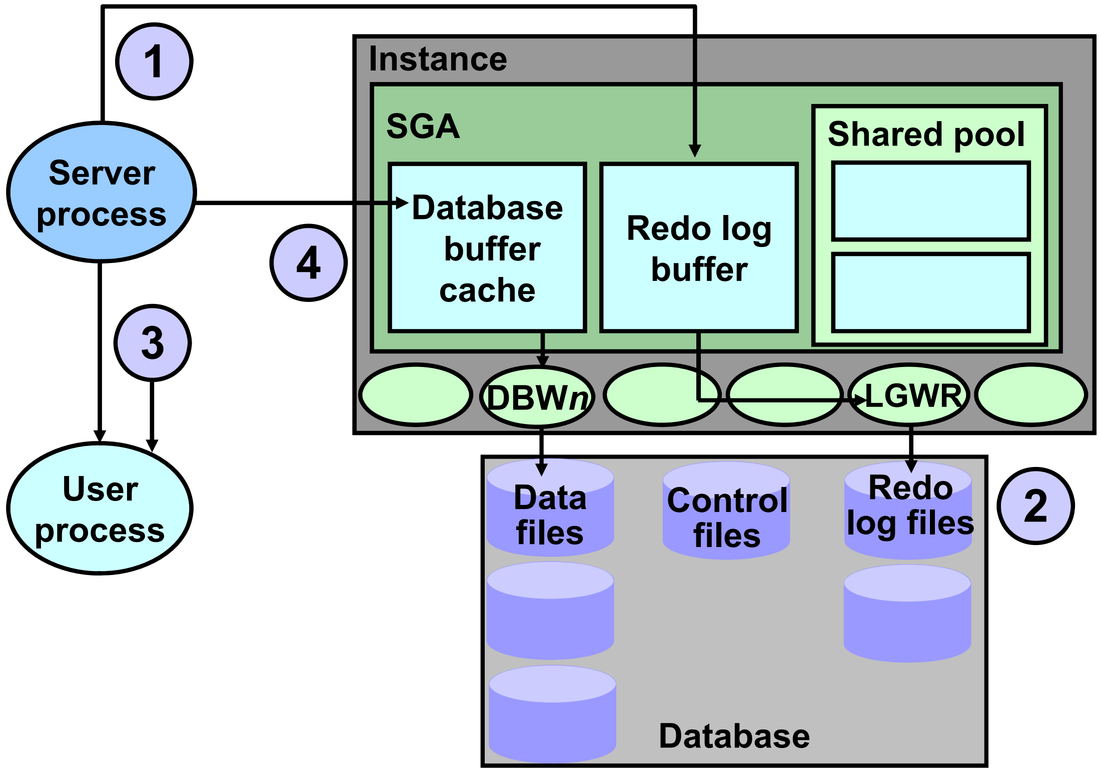
Before making a change, the server process saves the old data value in a rollback segment. This "before image" is used to:

- Undo the changes if the transaction is rolled back
- Provide read consistency by ensuring that other transactions do not see uncommitted changes made by the DML statement
- Recover the database to a consistent state in case of failures

Rollback segments, such as tables and indexes, exist in data files, and rollback blocks are brought into the database buffer cache as required. Rollback segments are created by the DBA.

Changes to rollback segments are recorded in the redo log buffer.

`COMMIT` Processing



The Oracle Server uses a fast `COMMIT` mechanism that guarantees that the committed changes can be recovered in case of instance failure.

System Change Number

Whenever a transaction commits, the Oracle Server assigns a commit SCN to the transaction. The SCN is monotonically incremented and is unique within the database. It is used by the Oracle Server as an internal time stamp to synchronize data and to provide read consistency when data is retrieved from the data files. Using the SCN enables the Oracle Server to perform consistency checks without depending on the date and time of the operating system.

Steps in Processing `COMMIT`s

When a `COMMIT` is issued, the following steps are performed:

1. The server process places a commit record, along with the SCN, in the redo log buffer.
2. LGWR performs a contiguous write of all the redo log buffer entries up to and including the commit record to the redo log files. After this point, the Oracle Server can guarantee that the changes will not be lost even if there is an instance failure.

3. The user is informed that the COMMIT is complete.
4. The server process records information to indicate that the transaction is complete and that resource locks can be released.

Flushing of the dirty buffers to the data file is performed independently by DBW0 and can occur either before or after the commit.

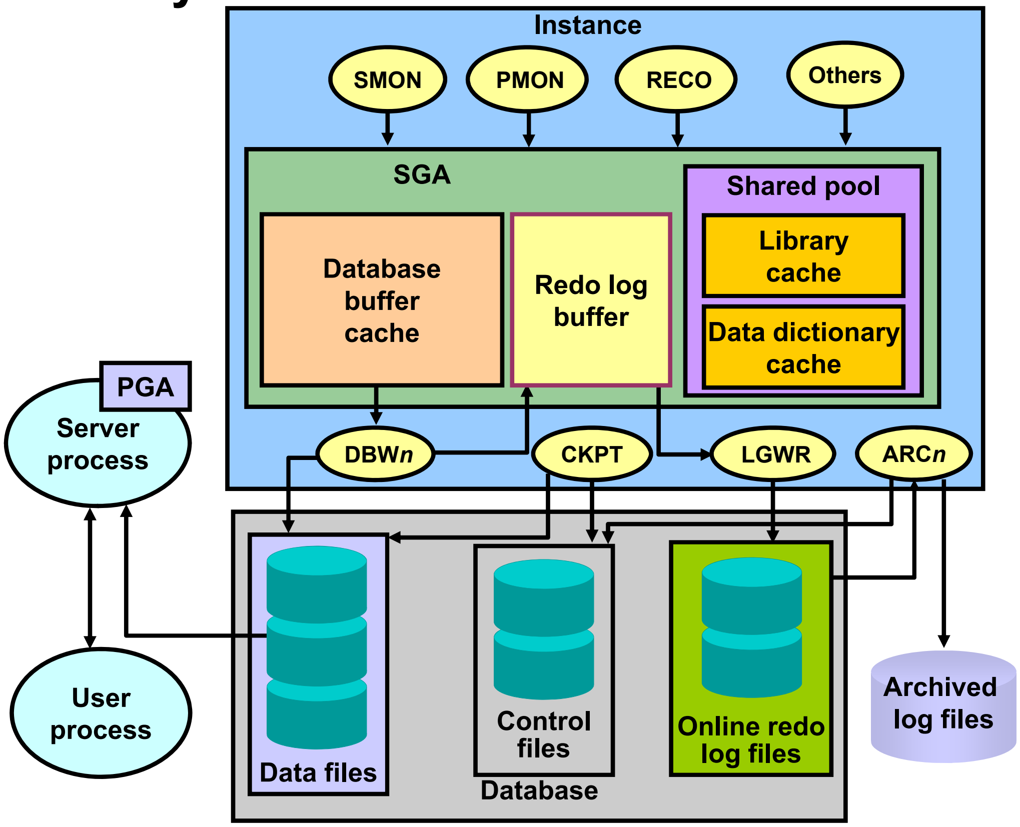
Advantages of the Fast COMMIT

The fast COMMIT mechanism ensures data recovery by writing changes to the redo log buffer instead of the data files. It has the following advantages:

- Sequential writes to the log files are faster than writing to different blocks in the data file.
- Only the minimal information that is necessary to record changes is written to the log files; writing to the data files would require whole blocks of data to be written.
- If multiple transactions request to commit at the same time, the instance piggybacks redo log records into a single write.
- Unless the redo log buffer is particularly full, only one synchronous write is required per transaction. If piggybacking occurs, there can be less than one synchronous write per transaction.
- Because the redo log buffer may be flushed before the COMMIT, the size of the transaction does not affect the amount of time needed for an actual COMMIT operation.

Note: Rolling back a transaction does not trigger LGWR to write to disk. The Oracle Server always rolls back uncommitted changes when recovering from failures. If there is a failure after a rollback, before the rollback entries are recorded on disk, the absence of a commit record is sufficient to ensure that the changes made by the transaction are rolled back.

Summary of the Oracle Database Architecture



An Oracle Database comprises an instance and its associated database:

- An instance comprises the SGA and the background processes.
  - **SGA:** Database buffer cache, redo log buffer, shared pool, and so on
  - **Background processes:** SMON, PMON, DBW*n*, CKPT, LGWR, and so on
- A database comprises storage structures:
  - **Logical:** Tablespaces, schemas, segments, extents, and Oracle block
  - **Physical:** Data files, control files, redo log files

When a user accesses the Oracle Database through an application, a server process communicates with the instance on behalf of the user process.