Oracle Database 10*g*: SQL Fundamentals II

Student Guide • Volume 1

D17111GC30
Edition 3.0
January 2009
D57873

Authors

Salome Clement
Chaitanya Koratamaddi
Priya Vennapusa

Technical Contributors and Reviewers

Claire Bennett
Brian Boxx
Zarko Cesljas
Laurent Dereac
Nancy Greenberg
Yash Jain
Angelika Krupp
Malika Marghadi
Priya Nathan
Narayanan Radhakrishnan
Bryan Roberts
Lata Shivaprasad
Naoko Susuki

Editors

Nita Pavitran
Atanu Raychaudhuri

Graphic Designer

Sanjeev Sharma

Publishers

Jobi Varghese
Giri Venugopal

Contents

Generating Reports by Grouping Related Data

Managing Data in Different Time Zones

Appendix C: Writing Advanced Scripts

Appendix D: Oracle Architectural Components

Appendix E: Using SQL Developer

x

Profile

Before You Begin This Course

Before you begin this course, you should have working experience with SQL.

Prerequisites

Oracle Database 10g: SQL Fundamentals I

How This Course Is Organized

Oracle Database 10g: SQL Fundamentals II is an instructor-led course featuring lectures and hands-on exercises. Online demonstrations and written practice sessions reinforce the concepts and skills that are introduced.

Related Publications

Oracle Publications

Title	Part Number
Oracle® Database Reference 10g Release 2 (10.2)	B14237-02
Oracle® Database SQL Reference 10g Release 2 (10.2)	B14200-02
Oracle® Database Concepts 10*g* Release *2 (10.2)*	B14220-02
Oracle® Database Application Developer's Guide - Fundamentals 10g Release 2 (10.2)	B14251-01
*SQL*Plus® User's Guide and Reference*	B14357-01

Additional Publications

System release bulletins

Installation and user's guides

read.me files

International Oracle User's Group (IOUG) articles

Oracle Magazine

Typographic Conventions

The following two lists explain Oracle University typographical conventions for words that appear within regular text or within code samples.

1. Typographic Conventions for Words Within Regular Text

Convention	Object or Term	Example
Courier New	User input; commands; column, table, and schema names; functions; PL/SQL objects; paths	Use the `SELECT` command to view information stored in the `LAST_NAME` column of the `EMPLOYEES` table. Enter `300`. Log in as `scott`
Initial cap	Triggers; user interface object names, such as button names	Assign a When-Validate-Item trigger to the ORD block. Click the Cancel button.
Italic	Titles of courses and manuals; emphasized words or phrases; placeholders or variables	For more information on the subject see *Oracle SQL Reference Manual* Do *not* save changes to the database. Enter *`hostname`*, where *`hostname`* is the host on which the password is to be changed.
Quotation marks	Lesson or module titles referenced within a course	This subject is covered in Lesson 3, “Working with Objects.”

Typographic Conventions (continued)

2. Typographic Conventions for Words Within Code Samples

Convention	Object or Term	Example
Uppercase	Commands, functions	`SELECT employee_id` `FROM  employees;`
Lowercase, italic	Syntax variables	`CREATE ROLE` *`role`*`;`

Initial cap	Forms triggers	`Form module: ORD`
		`Trigger level: S _ITEM.QUANTITY item`
		`Trigger name: When-Validate-Item`
		`. . .`

Lowercase	Column names, table names, filenames, PL/SQL objects	`. . .`
		`OG_ACTIVATE_LAYER`
		`(OG_GET_LAYER ('prod_pie_layer'))`
		`. . .`
		`SELECT last_name`
		`FROM   employees;`

Bold	Text that must be entered by a user	`CREATE USER` **`scott`** `IDENTIFIED BY` **`tiger`**`;`

Introduction

Objectives

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

- List the course objectives
- Describe the sample tables used in the course

Course Objectives

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

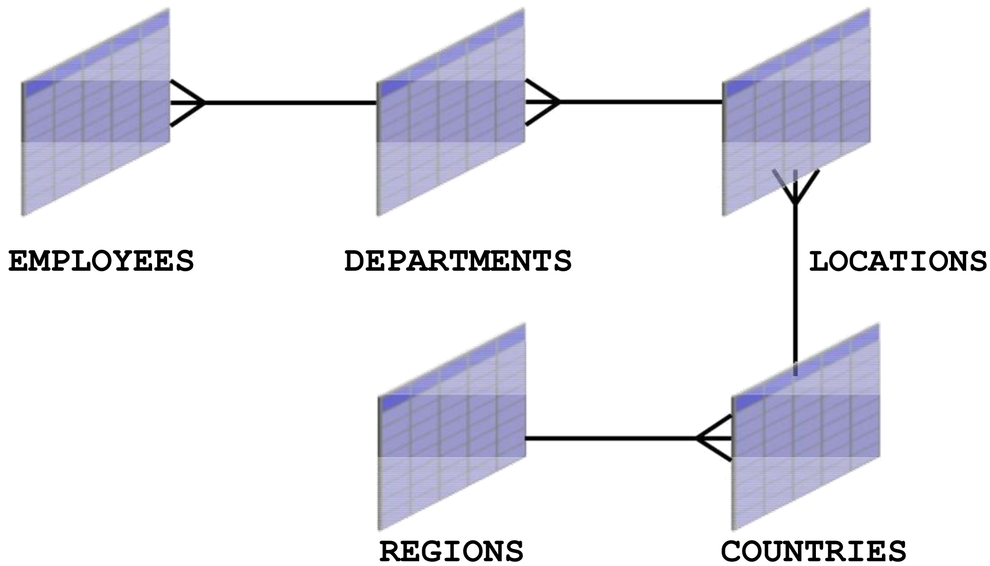
- Use advanced SQL data retrieval techniques to retrieve data from database tables
- Apply advanced techniques in a practice that simulates real life

Course Overview

In this course, you use advanced SQL data retrieval techniques such as:

- Datetime functions
- `ROLLUP`, `CUBE` operators, and `GROUPING SETS`
- Hierarchical queries
- Correlated subqueries
- Multitable inserts
- `Merge` operation
- External tables
- Regular expression usage

Course Application



Tables Used in the Course

The following tables are used in this course:

EMPLOYEES: 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.

DEPARTMENTS: 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.

LOCATIONS: This 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.

COUNTRIES: This 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.

REGIONS: This table contains region IDs and region names of the various countries. It is a primary key table to the COUNTRIES table.

Summary

In this lesson, you should have learned the following:

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

1

Controlling User Access

Objectives

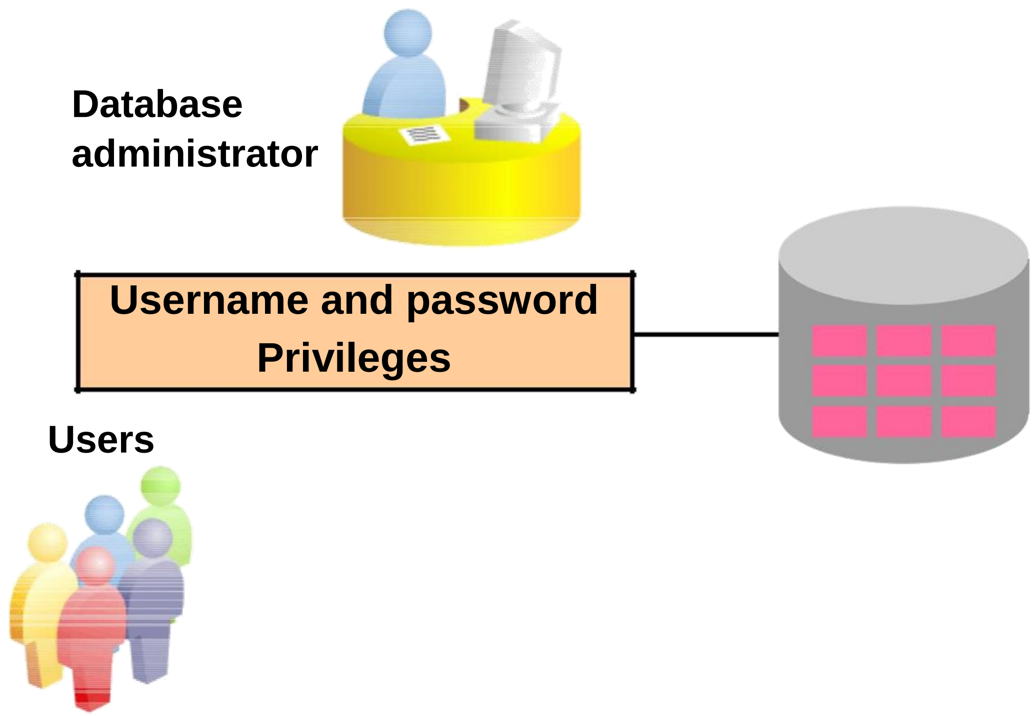
After completing this lesson, you should be able to do the following:

- Differentiate system privileges from object privileges
- Grant privileges on tables
- View privileges in the data dictionary
- Grant roles
- Distinguish between privileges and roles

Objectives

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

Controlling User Access



Controlling User Access

In a multiple-user environment, you want to maintain security of the 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.
- Create synonyms for database objects.

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 have on the objects.

Privileges

- Database security:
  - System security
  - Data security
- System privileges: Gaining access to the database
- Object privileges: Manipulating the content of the database objects
- Schemas: Collection of objects such as tables, views, and sequences

Privileges

Privileges are 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.

For more information, see the *Oracle Database10g Application Developer's Guide–Fundamentals* reference manual.

System Privileges

- More than 100 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

System Privileges

More than 100 distinct system privileges are available for users and roles. System privileges typically are provided by the database administrator.

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  USER1
IDENTIFIED BY   USER1;
CREATE USER succeeded.
```


Creating a User

The DBA creates a 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 *Oracle Database10g SQL Reference*.

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` designates that every user is granted the privilege

Note: Current system privileges can be found in the `SESSION_PRIVS` dictionary view.

Granting System Privileges

The DBA can grant specific system privileges to a user.

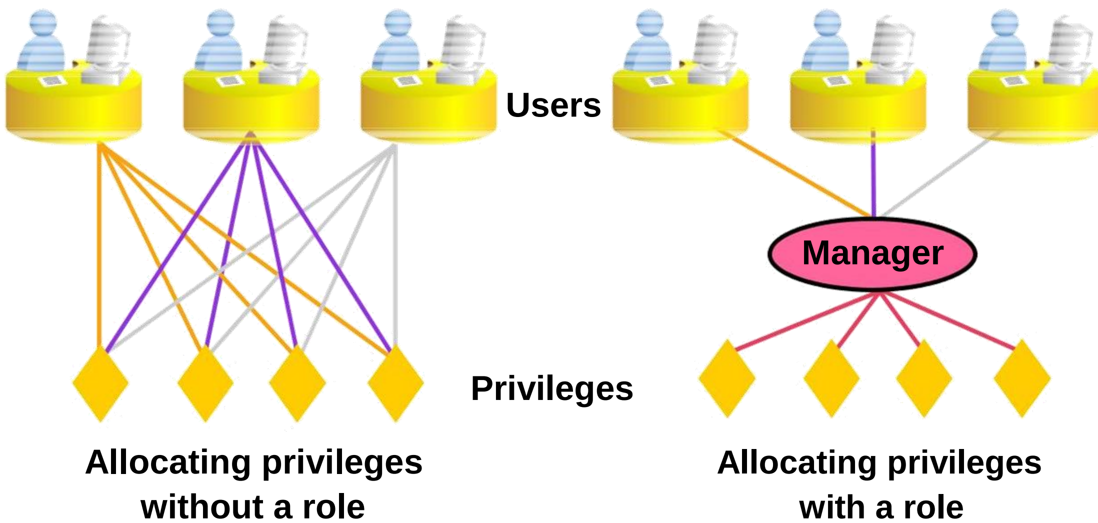
```
GRANT   create session, create table,
        create sequence, create view
TO      scott;
GRANT CREATE succeeded.
```


Granting System Privileges

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, user Scott has been assigned the privileges to create sessions, tables, sequences, and views.

What Is a Role?



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.

Creating and Granting Privileges to a Role

Create a role:

```
CREATE ROLE manager;
CREATE ROLE succeeded.
```

Grant privileges to a role:

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

Grant a role to users:

```
GRANT manager TO BELL, KOCHHAR;
GRANT succeeded.
```

ORACLE

Creating a Role

The example in the slide creates a manager role and then enables managers to create tables and views. It then grants Bell and Kochhar the role of managers. Now Bell and Kochhar 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 HR
IDENTIFIED BY employ;
ALTER USER HR succeeded.
```


Changing Your Password

The DBA creates an account and initializes a password for every user. You can change your 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 Database10g SQL Reference* manual.

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.

Object Privileges

Object Privilege	Table	View	Sequence	Procedure
`ALTER`				
`DELETE`				
`EXECUTE`				
`INDEX`				
`INSERT`				
`REFERENCES`				
`SELECT`				
`UPDATE`				

Object Privileges

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.

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`, then 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

Granting Object Privileges

- Grant query privileges on the EMPLOYEES table:

```
GRANT  select
ON     employees
TO     sue, rich;
GRANT succeeded.
```

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

```
GRANT  update (department_name, location_id)
ON     departments
TO     scott, manager;
GRANT succeeded.
```


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 users Sue and Rich the privilege to query your `EMPLOYEES` table. The second example grants `UPDATE` privileges on specific columns in the `DEPARTMENTS` table to Scott and to the manager role.

If Sue or Rich now want to use a `SELECT` statement to obtain data from the `EMPLOYEES` table, the syntax they must use is:

```
SELECT  * FROM HR.employees;
```

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

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

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

Passing On Your Privileges

Ā À Ā Ā

ive a user authority to pass along privileges:

```
GRANT  select, insert
ON     departments
TO     scott
WITH   GRANT OPTION;
GRANT succeeded.
```

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

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

ORACLE

Passing On Your Privileges

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 user Scott access to your `DEPARTMENTS` table with the privileges to query the table and add rows to the table. The example also shows that Scott 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 Privileges Granted

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_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
`USER_SYS_PRIVS`	System privileges granted to the user

Confirming Granted Privileges

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," then 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

You can access the data dictionary to view the privileges that you have. The chart in the slide describes various data dictionary views.

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];
```


Revoking Object Privileges

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 *Oracle Database10g SQL Reference*.

Note: If a user were to leave the company and you revoke his 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, then the system privileges granted by this user to other users are not affected by this action.

Revoking Object Privileges

As user Alice, revoke the SELECT and INSERT privileges given to user Scott on the DEPARTMENTS table.

```
REVOKE  select, insert
ON      departments
FROM    scott;
REVOKE succeeded.
```


Revoking Object Privileges (continued)

The example in the slide revokes SELECT and INSERT privileges given to user Scott 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, then 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, then the privileges granted to users C and D are also revoked.

Summary

In this lesson, you should have learned about statements that control access to the database and database objects.

Statement	Action
`CREATE USER`	Creates a user (usually performed by a DBA)
`GRANT`	Gives other users privileges to access the objects
`CREATE ROLE`	Creates a collection of privileges (usually performed by a DBA)
`ALTER USER`	Changes a user's password
`REVOKE`	Removes privileges on an object from users

Summary

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 password 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.

With database links, you can access data on remote databases. Privileges cannot be granted on remote objects.

Practice 1: 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
- Creating a synonym
- Querying the data dictionary views related to privileges

Practice 1: Overview

In this exercise, you practice controlling access to the database objects. You use two accounts: `ora21` and `ora22`.

Practice 1

To complete question 6 and the subsequent ones, you need to connect to the database using SQL Developer. To do so, double-click the SQL Developer icon on the desktop.

To create a new database connection in the Connections Navigator, right-click Connections. Select New Connection from the shortcut menu. The New/Select Database Connection dialog box appears.

Create a database connection using the following information:

Connection Name: `ora21`
Username: `ora21`
Password: `ora21`
Hostname: localhost
Port: 1521
SID: ORCL
Ensure that you select the Save Password check box.

Create another database connection using the following information:

Connection Name: `ora22`
Username: `ora22`
Password: `ora22`
Hostname: localhost
Port: 1521
SID: ORCL
Ensure that you select the Save Password check box.

Which privilege should a user be given to log on to the Oracle server? Is this a system or an object privilege?

____

Which privilege should a user be given to create tables?

____

If you create a table, who can pass along privileges to other users on your table?

____

You are the DBA. You are creating many users who require the same system privileges. What should you use to make your job easier?

____

Which command would you use to change your password?

____

Practice 1 (continued)

6. Connect as user `ora21`. Query all the rows in your `DEPARTMENTS` table.

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	40	Human Resources	203	2400
5	50	Shipping	121	1500

…

26	260	Recruiting	(null)	1700
27	270	Payroll	(null)	1700

Add a new row to your `DEPARTMENTS` table. Add Education as department number 500.
Grant user `ora22` access to your `DEPARTMENTS` table.
Connect as user `ora22`. Query user `ora21`'s `DEPARTMENTS` table.

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	500	Education	(null)	(null)
2	10	Administration	200	1700
3	20	Marketing	201	1800
4	30	Purchasing	114	1700

…

27	260	Recruiting	(null)	1700
28	270	Payroll	(null)	1700

Create a synonym for user `ora21`'s `DEPARTMENTS` table. Query all the rows in user `ora21`'s `DEPARTMENTS` table by using your synonym.

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	500	Education	(null)	(null)
2	10	Administration	200	1700
3	20	Marketing	201	1800
4	30	Purchasing	114	1700

…

27	260	Recruiting	(null)	1700
28	270	Payroll	(null)	1700

Practice 1 (continued)

As user `ora22`, query the `USER_TABLES` data dictionary to see information about the tables that you own.

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

As user `ora22`, query the `ALL_TABLES` data dictionary view to see information about all the tables that you can access. Exclude the tables that you own.
Note: Your list may not exactly match the following list:

	TABLE_NAME	OWNER
1	DUAL	SYS
2	SYSTEM_PRIVILEGE_MAP	SYS
…		
91	SDO_GEOR_PLUGIN_REGISTRY	MDSYS
92	DEPARTMENTS	ORA21

Connect as user `ora21` and revoke the `SELECT` privilege from user `ora22`.

Remove the row that you inserted into the `DEPARTMENTS` table in step 7 and save the changes.

2

Managing Schema Objects

ORACLE

Objectives

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

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

Objectives

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.

ALTER TABLE Statement

Use the ALTER TABLE statement to:

- Add a new column
- Modify an existing column
- Define a default value for the new column
- Drop a column

ALTER TABLE Statement

After you create a table, you may need to change the table structure 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);
```


ALTER TABLE Statement (continued)

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 new column
datatype	Is the data type and length of the new column
DEFAULT *expr*	Specifies the default value for a new column

Adding a Column

- You use the ADD clause to add columns:

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

- The new column becomes the last column:

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE	JOB_ID
1	149	Zlotkey	126000	29-JAN-00	(null)
2	174	Abel	132000	11-MAY-96	(null)
3	176	Taylor	103200	24-MAR-98	(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, then the new column is initially null for all the rows. You cannot add a mandatory NOT NULL column to a table that contains data in the other columns. You can only add a NOT NULL column to an empty table.

Modifying a Column

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

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

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

Modifying a Column

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 numeric or 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 you no longer need from the table:

```
ALTER TABLE  dept80
DROP COLUMN  job_id;
ALTER TABLE succeeded.
```

	EMPLOYEE_ID	LAST_NAME	ANNSAL	HIRE_DATE
1	149	Zlotkey	126000	29-JAN-00
2	174	Abel	132000	11-MAY-96
3	176	Taylor	103200	24-MAR-98

ORACLE

Dropping a Column

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` 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 column cannot be dropped if it is part of a constraint or part of an index key 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.

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>);
OR
ALTER TABLE  <table_name>
SET   UNUSED COLUMN <column name>;
```

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


SET UNUSED Option

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 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. 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 of 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);
ALTER TABLE succeeded.

ALTER TABLE  dept80
DROP  UNUSED COLUMNS;
ALTER TABLE succeeded.
```

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>);
```


Adding a Constraint

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; ALTER TABLE succeeded.
```

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


Adding a Constraint (continued)

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 CASCADE

Delete child rows when a parent key is deleted:

```
ALTER TABLE Emp2 ADD CONSTRAINT emp_dt_fk
FOREIGN KEY (Department_id)
REFERENCES departments ON DELETE CASCADE);
ALTER TABLE succeeded.
```


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 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.

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

Deferring Constraints

You can defer checking constraints for validity until the end of the transaction. A constraint is deferred if the system checks that it is satisfied only on commit. If a deferred constraint is violated, then commit causes the transaction to roll back. If a constraint is immediate (not deferred), then 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 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, and either initially deferred or initially immediate. 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 rows are validated.

Dropping a Constraint

Remove the manager constraint from the EMP2 table:

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

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;
ALTER TABLE succeeded.
```


Dropping a Constraint

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;
ALTER TABLE succeeded.
```


Disabling a Constraint

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 currently disabled in the table definition by using the ENABLE clause.

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

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

Enabling a 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, then 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 Constraint (continued)

Guidelines (continued)

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.

Cascading Constraints

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

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

An error is returned for the following statements:

```
ALTER TABLE test1 DROP (pk);      —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; ALTER TABLE succeeded.
```

```
ALTER TABLE test1
DROP (pk, fk, col1);
ALTER TABLE succeeded.
```


Cascading Constraints (continued)

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, then CASCADE CONSTRAINTS is not required. For example, assuming that no other referential constraints from other tables refer to the 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 (pk, fk, col1);
```

Overview of Indexes

Indexes are created:

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

Overview of Indexes

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 10g SQL Reference*.

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));
CREATE TABLE succeeded.
```

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

	INDEX_NAME	TABLE_NAME
1	EMP_ID_IDX	NEW_EMP

ORACLE

CREATE INDEX with the CREATE TABLE Statement

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. 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));
CREATE TABLE succeeded.
   SELECT INDEX_NAME, TABLE_NAME
   FROM   USER_INDEXES
   WHERE  TABLE_NAME = 'EMP_UNNAMED_INDEX';
```

	INDEX_NAME	TABLE_NAME
1	SYS_C006083	EMP_UNNAMED_INDEX

CREATE INDEX with the CREATE TABLE Statement (continued)

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. So 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));

CREATE INDEX succeeded.

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

ORACLE

Function-Based Indexes

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

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

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;
DROP INDEX succeeded.
```

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

Removing an Index

You cannot modify indexes. To change an index, you must drop it and then re-create it. Remove an index definition from the data dictionary by issuing the `DROP INDEX` statement. To drop an index, you must be the owner of the index or have the `DROP ANY INDEX` privilege.

In the syntax:

index	Is the name of the index

Note: If you drop a table, indexes and constraints are automatically dropped, but views and sequences remain.

DROP TABLE … PURGE

```
DROP TABLE dept80 PURGE;
```


DROP TABLE … PURGE

Oracle Database 10*g* introduces a new 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, then 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`, then 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.

Note: You cannot roll back a `DROP TABLE` statement with the `PURGE` clause, and you cannot recover the table if you drop it with the `PURGE` clause. This feature was not available in earlier releases.

FLASHBACK TABLE Statement

Repair tool for accidental table modifications

– Restores a table to an earlier point in time
– Benefits: Ease of use, availability, fast execution
– Performed in place

Syntax:

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


FLASHBACK TABLE Statement

Self-Service Repair Facility

Oracle Database 10*g* provides a new SQL 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 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 10g Release 2 (10.2)*.

FLASHBACK TABLE Statement

```
DROP TABLE emp2;
DROP TABLE succeeded.
```

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

	ORIGINAL_NAME	OPERATION	DROPTIME
1	EMP2	DROP	2008-11-13:08:44:39

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


FLASHBACK TABLE Statement (continued)

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 the unlikely situation where 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:

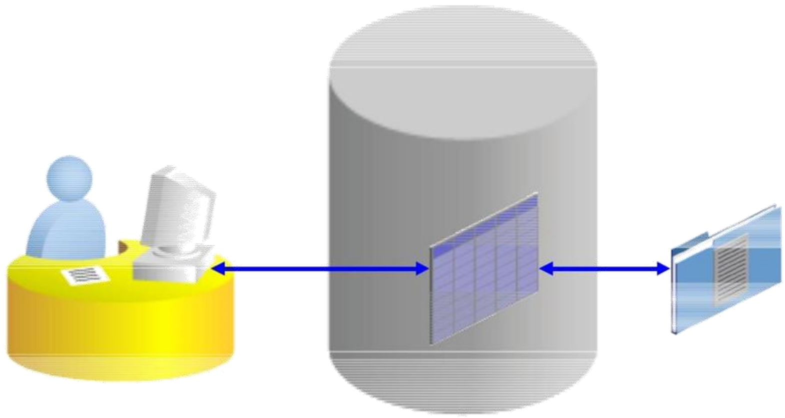
```
SELECT * FROM RECYCLEBIN;
```

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;
```

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 of 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.

External Tables (continued)

The ORACLE_DATAPUMP access driver can be used to both import and export data 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 hr;
```


Example of Creating an External Table

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.

Example of Creating an External Table (continued)

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.

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

```
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];
```


Creating an External Table

You create external tables 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 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.

Creating an External Table (continued)

Use the LOCATION clause to specify one external locator for each external data source. Usually, the <*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.

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;
CREATE TABLE succeeded.
```


Example of Creating an External Table by Using the ORACLE_LOADER Access Driver

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:

Create a directory object, `emp_dir`, as follows:

```
CREATE DIRECTORY emp_dir AS '/emp_dir' ;
```

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
```

Example of Creating an External Table by Using the ORACLE_LOADER Access Driver (continued)

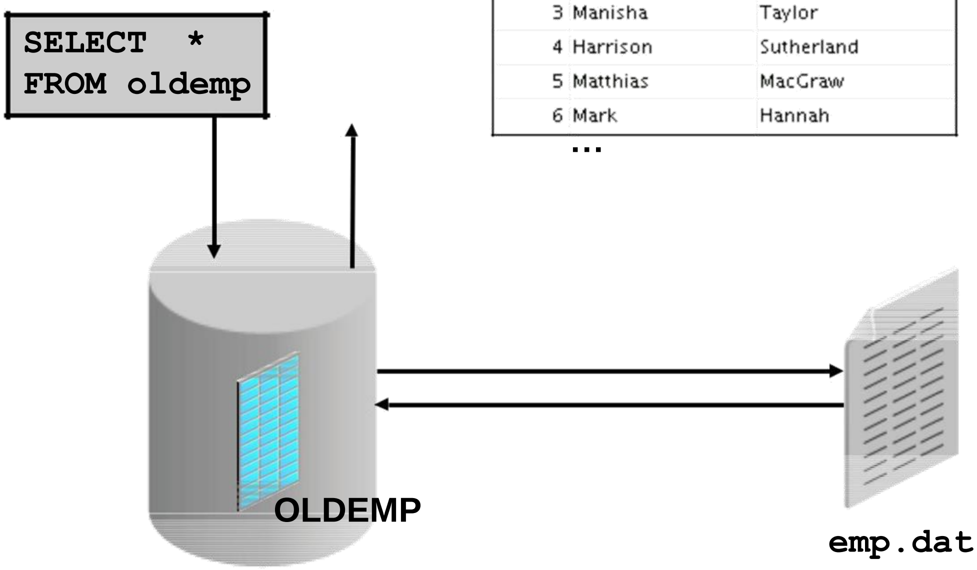
In the example, the TYPE specification is given only to illustrate its use. ORACLE_LOADER is the default access driver if not specified. The ACCESS PARAMETERS option provides values to parameters of the specific access driver, which are interpreted by the access driver, not by the Oracle server.

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, then 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, then the query be aborted and an error 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 like a relational table.

Querying External Tables



	FNAME	LNAME
1	Constantin	Welles
2	Harry	Pacino
3	Manisha	Taylor
4	Harrison	Sutherland
5	Matthias	MacGraw
6	Mark	Hannah

...

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.

Summary

In this lesson, you should have learned how to:

- Add constraints
- Create indexes
- Create a primary key constraint using an index
- Create indexes using the `CREATE TABLE` statement
- Create function-based indexes
- Drop columns and set column `UNUSED`
- Perform `FLASHBACK` operations
- Create and use external tables

Summary

Alter tables to add or modify columns or constraints. Create indexes and function-based indexes using the `CREATE INDEX` statement. Drop unused columns. Use `FLASHBACK` mechanics to restore tables. Use the `external_table` clause to create an external table, which 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 as you create the table with the `CREATE TABLE` statement.

Practice 2: Overview

This practice covers the following topics:

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

Practice 2: Overview

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. You drop columns and use the `FLASHBACK` operation.

Practice 2

Create the `DEPT2` table based on the following table instance chart. Place the syntax in a script called `lab_02_01.sql`, and then execute the statement in the script to create the table. Confirm that the table is created.

Column Name	ID	NAME
Key Type		
Nulls/Unique		
FK Table		
FK Column		
Data type	NUMBER	VARCHAR2
Length	7	25

```
Name                           Null     Type
------------------------------ -------- -------------
ID                                      NUMBER(7)
NAME                                    VARCHAR2(25)
```

Populate the `DEPT2` table with data from the `DEPARTMENTS` table. Include only the columns that you need.

Create the `EMP2` table based on the following table instance chart. Place the syntax in a script called `lab_02_03.sql`, and then execute the statement in the script to create the table. Confirm that the table is created.

Column Name	ID	LAST_NAME	FIRST_NAME	DEPT_ID
Key Type				
Nulls/Unique				
FK Table				
FK Column				
Data type	NUMBER	VARCHAR2	VARCHAR2	NUMBER
Length	7	25	25	7

```
Name                           Null     Type
------------------------------ -------- -------------
ID                                      NUMBER(7)
LAST_NAME                               VARCHAR2(25)
FIRST_NAME                              VARCHAR2(25)
DEPT_ID                                 NUMBER(7)
```

Practice 2 (continued)

Modify the EMP2 table to allow for longer employee last names. Confirm your modification.

```
Name                           Null     Type
------------------------------ -------- -------------
ID                                      NUMBER(7)
LAST_NAME                               VARCHAR2(50)
FIRST_NAME                              VARCHAR2(25)
DEPT_ID                                 NUMBER(7)
```

Confirm that both the DEPT2 and EMP2 tables are stored in the data dictionary.
(**Hint:** USER_TABLES)

	TABLE_NAME
1	DEPT2
2	EMP2

Create the EMPLOYEES2 table based on the structure of the EMPLOYEES table. Include only the EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY, and DEPARTMENT_ID columns. Name the columns in your new table ID, FIRST_NAME, LAST_NAME, SALARY, and DEPT_ID, respectively. Confirm that the table is created.

```
Name                           Null     Type
------------------------------ -------- -------------
ID                                      NUMBER(6)
FIRST_NAME                              VARCHAR2(20)
LAST_NAME                      NOT NULL VARCHAR2(25)
SALARY                                  NUMBER(8,2)
DEPT_ID                                 NUMBER(4)
```

Drop the EMP2 table.
Query the Recycle Bin to see whether the table is present.

	ORIGINAL_NAME	OPERATION	DROPTIME
1	EMP2	DROP	2008-11-14:01:03:45

Undrop the EMP2 table.

```
Name                           Null     Type
------------------------------ -------- -------------
ID                                      NUMBER(7)
LAST_NAME                               VARCHAR2(50)
FIRST_NAME                              VARCHAR2(25)
DEPT_ID                                 NUMBER(7)
```

Drop the FIRST_NAME column from the EMPLOYEES2 table. Confirm your modification by checking the description of the table.

```
Name                           Null     Type
------------------------------ -------- -------------
ID                                      NUMBER(6)
LAST_NAME                      NOT NULL VARCHAR2(25)
SALARY                                  NUMBER(8,2)
DEPT_ID                                 NUMBER(4)
```

Practice 2 (continued)

In the EMPLOYEES2 table, mark the DEPT_ID column as UNUSED. Confirm your modification by checking the description of the table.

```
Name                           Null     Type
------------------------------ -------- -------------
ID                                      NUMBER(6)
LAST_NAME                      NOT NULL VARCHAR2(25)
SALARY                                  NUMBER(8,2)
```

Drop all the UNUSED columns from the EMPLOYEES2 table. Confirm your modification by checking the description of the table.

```
Name                           Null     Type
------------------------------ -------- -------------
ID                                      NUMBER(6)
LAST_NAME                      NOT NULL VARCHAR2(25)
SALARY                                  NUMBER(8,2)
```

Add a table-level PRIMARY KEY constraint to the EMP2 table on the ID column. The constraint should be named at creation. Name the constraint my_emp_id_pk.

Create a PRIMARY KEY constraint to the DEPT2 table using the ID column. The constraint should be named at creation. Name the constraint my_dept_id_pk.

Add a foreign key reference on the EMP2 table that ensures that the employee is not assigned to a nonexistent department. Name the constraint my_emp_dept_id_fk.

Confirm that the constraints were added by querying the USER_CONSTRAINTS view. Note the types and names of the constraints.

	CONSTRAINT_NAME	CONSTRAINT_TYPE
1	MY_DEPT_ID_PK	P
2	MY_EMP_ID_PK	P
3	MY_EMP_DEPT_ID_FK	R

Display the object names and types from the USER_OBJECTS data dictionary view for the EMP2 and DEPT2 tables. Note that the new tables are created and a new index is created.

	OBJECT_NAME	OBJECT_TYPE
1	DEPT2	TABLE
2	DEPT_ID_PK	INDEX
3	DEPT_LOCATION_IX	INDEX
4	EMP2	TABLE
5	EMPLOYEES	TABLE
6	EMPLOYEES2	TABLE
7	EMPLOYEES_SEQ	SEQUENCE
8	EMP_DEPARTMENT_IX	INDEX

...

Practice 2 (continued)

If you have time, complete the following exercise:

Modify the `EMP2` table. Add a `COMMISSION` column of the `NUMBER` data type, precision 2, scale 2. Add a constraint to the `COMMISSION` column that ensures that a commission value is greater than zero.

Drop the `EMP2` and `DEPT2` tables so that they cannot be restored. Check the Recycle Bin.

ORIGINAL_NAME	OPERATION	DROPTIME

```
0 rows selected.
```

Create the `DEPT_NAMED_INDEX` table based on the following table instance chart. Name the index for the `PRIMARY KEY` column as `DEPT_PK_IDX`. Confirm that the table is created.

Column Name	Deptno	Dname
Primary Key	Yes	
Data Type	Number	VARCHAR2
Length	4	30

```
Name                            Null     Type
------------------------------- -------- -------------
DEPTNO                          NOT NULL NUMBER(4)
DNAME                                    VARCHAR2(30)
```

Manipulating Large Data Sets

Objectives

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

- Manipulate data using subqueries
- Describe the features of multitable `INSERT`s
- Use the following types of multitable `INSERT`s
  - Unconditional `INSERT`
  - Pivoting `INSERT`
  - Conditional `ALL INSERT`
  - Conditional `FIRST INSERT`
- Merge rows in a table
- Track the changes to data over a period of time

Objectives

In this lesson, you learn how to manipulate data in the Oracle Database by using subqueries. You also learn about multitable insert statements, the `MERGE` statement, and tracking changes in the database.

Using Subqueries to Manipulate Data

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

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

Using Subqueries to Manipulate Data

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.

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.

Copying Rows from Another Table

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

Syntax

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

In the syntax:

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

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

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

For more information, see *Oracle Database 10g SQL Reference*.

Inserting Using a Subquery as a Target

```
INSERT INTO
        (SELECT employee_id, last_name,
                email, hire_date, job_id, salary,
                department_id
         FROM   empl3
         WHERE department_id = 50)
VALUES (99999, 'Taylor', 'DTAYLOR',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000, 50);

1 row inserted.
```


Inserting Using a Subquery as a Target

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 employee ID or leave out a value for a mandatory `NOT NULL` column.

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

Inserting Using a Subquery as a Target

Verify the results.

```
SELECT employee_id, last_name, email, hire_date,
       job_id, salary, department_id
FROM   empl3
WHERE  department_id = 50;
```

	EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
1	120	Weiss	MWEISS	18-JUL-96	ST_MAN	8000	50
2	121	Fripp	AFRIPP	10-APR-97	ST_MAN	8200	50
3	122	Kaufling	PKAUFLIN	01-MAY-95	ST_MAN	7900	50
...							
45	199	Grant	DGRANT	13-JAN-00	SH_CLERK	2600	50
46	99999	Taylor	DTAYLOR	07-JUN-99	ST_CLERK	5000	50

ORACLE

Inserting Using a Subquery as a Target (continued)

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

Retrieving Data with a Subquery as Source

```
SELECT  a.last_name, a.salary,
        a.department_id, b.salavg
FROM    employees a JOIN (SELECT   department_id,
                          AVG(salary) salavg
                          FROM     employees
                          GROUP BY department_id) b
ON a.department_id = b.department_id
AND     a.salary > b.salavg;
```

	LAST_NAME	SALARY	DEPARTMENT_ID	SALAVG
1	King	24000	90	19333.3333333333333333333333333333333333
2	Hunold	9000	60	5760
3	Ernst	6000	60	5760
4	Greenberg	12000	100	8600
5	Faviet	9000	100	8600
6	Raphaely	11000	30	4150

…

ORACLE

Retrieving Data Using a Subquery as Source

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. The example in the slide displays employee last names, salaries, department numbers, and average salaries for all the employees who earn more than the average salary in their department. The subquery in the `FROM` clause is named `b`, and the outer query references the `SALAVG` column using this alias.

Updating Two Columns with a Subquery

Update the job and salary of employee 114 to match the job of employee 205 and the salary of employee 168:

```
UPDATE   empl3
SET      job_id    = (SELECT  job_id
                      FROM    employees
                      WHERE   employee_id = 205),
         salary    = (SELECT  salary
                      FROM    employees
                      WHERE   employee_id = 168)
WHERE    employee_id    =  114;
1 rows updated.
```

ORACLE

Updating Two Columns with a Subquery

You can update multiple columns in the SET clause of an UPDATE statement by writing multiple subqueries.

Syntax

```
UPDATE table
SET     column   =
                  (SELECT     column
                   FROM table
                   WHERE  condition)
   [ ,
        column   =
                  (SELECT     column
                  FROM table
                   WHERE  condition)]
[WHERE  condition ]    ;
```

Note: If no rows are updated, a message "0 rows updated." is returned.

Updating Rows Based on Another Table

Use subqueries in `UPDATE` statements to update rows in a table based on values from another table:

```
UPDATE   empl3
SET      department_id  = (SELECT department_id
                           FROM employees
                           WHERE  employee_id = 100)
WHERE    job_id         = (SELECT job_id
                           FROM employees
                           WHERE  employee_id = 200);
1 rows updated.
```

ORACLE

Updating Rows Based on Another Table

You can use subqueries in `UPDATE` statements to update rows in a table. The example in the slide updates the `EMPL3` table based on the values from the `EMPLOYEES` table. It changes the department number of all employees with employee 200's job ID to employee 100's current department number.

Deleting Rows Based on Another Table

Use subqueries in `DELETE` statements to remove rows from a table based on values from another table:

```
DELETE FROM empl3
WHERE  department_id =
                    (SELECT department_id
                     FROM   departments
                     WHERE  department_name
                            LIKE '%Public%');
1 rows deleted.
```


Deleting Rows Based on Another Table

You can use subqueries to delete rows from a table based on values from another table. The example in the slide deletes all the employees who are in a department where the department name contains the string "Public." The subquery searches the `DEPARTMENTS` table to find the department number based on the department name containing the string "Public." The subquery then feeds the department number to the main query, which deletes rows of data from the `EMPLOYEES` table based on this department number.

Using the WITH `CHECK OPTION` Keyword on DML Statements

- A subquery is used to identify the table and columns of the DML statement.
- The `WITH CHECK OPTION` keyword prohibits you from changing rows that are not in the subquery.

```
INSERT INTO       (SELECT employee_id, last_name, email,
                  hire_date, job_id, salary
                   FROM    empl3
                   WHERE   department_id = 50
                       WITH CHECK OPTION)
   VALUES (99998, 'Smith', 'JSMITH',
          TO_DATE('07-JUN-99', 'DD-MON-RR'),
          'ST_CLERK', 5000);
```

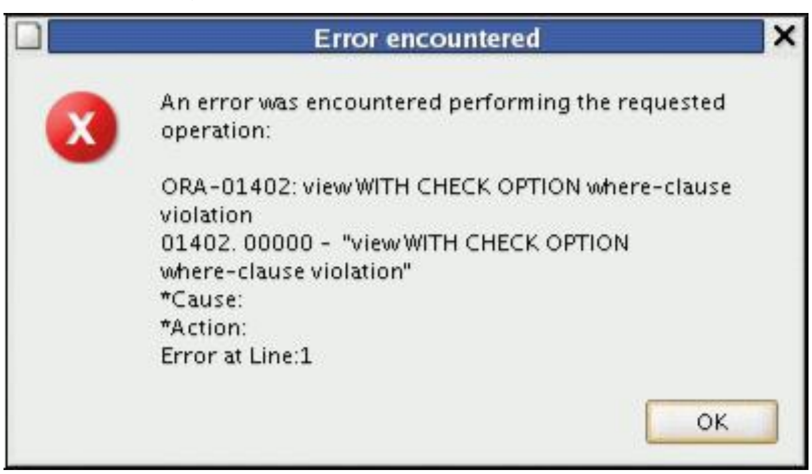
ERROR:

```
ORA-01402: view WITH CHECK OPTION where-
   clause violation
```


`WITH CHECK OPTION` Keyword

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

In the example shown, the `WITH CHECK OPTION` keyword is used. The subquery identifies rows that are in department 50, but the department ID is not in the `SELECT` list, and a value is not provided for it in the `VALUES` list. Inserting this row results in a department ID of null, which is not in the subquery.



Overview of the Explicit Default Feature

- With the explicit default feature, you can use the `DEFAULT` keyword as a column value where the column default is desired.
- The addition of this feature is for compliance with the SQL:1999 standard.
- This allows the user to control where and when the default value should be applied to data.
- Explicit defaults can be used in `INSERT` and `UPDATE` statements.

Explicit Defaults

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

The `DEFAULT` option saves you from hard coding the default value in your programs or querying 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 program, so 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;
```


Using Explicit Default Values

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

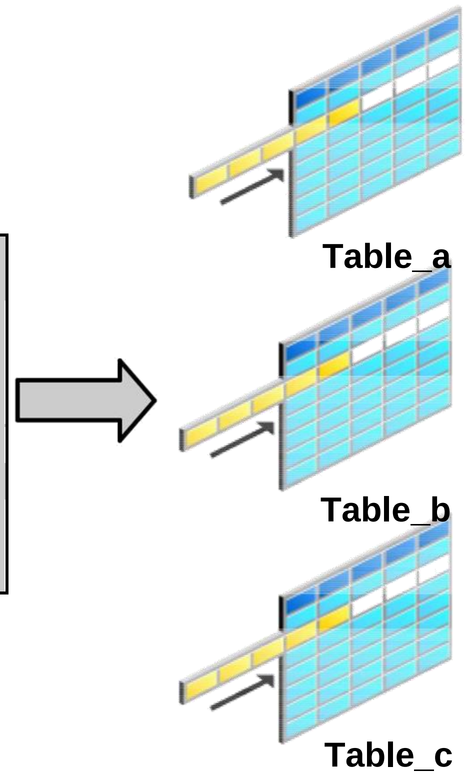
In the first example in the slide, the INSERT statement uses a default value for the MANAGER_ID column. If there is no default value defined for the column, a null value is inserted instead.

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, then 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.*

Overview of Multitable INSERT Statements

```
INSERT  ALL
   INTO table_a VALUES(…,…,…)
   INTO table_b VALUES(…,…,…)
   INTO table_c VALUES(…,…,…)
   SELECT …
   FROM  sourcetab
   WHERE …;
```



Overview of Multitable INSERT Statements

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 can play a very useful role 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.

Overview of Multitable `INSERT` Statements

- The `INSERT…SELECT` statement can be used to insert rows into multiple tables as part of a single DML statement.
- Multitable `INSERT` statements can be 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

Overview of Multitable `INSERT` Statements (continued)

Multitable `INSERT` statements offer the benefits of the `INSERT ... SELECT` statement when multiple tables are involved as targets. Using functionality before Oracle9*i* Database, 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 `ALL INSERT`
- Conditional `FIRST INSERT`
- Pivoting `INSERT`

Types of Multitable `INSERT` Statements

The types of multitable `INSERT` statements are:

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

You use different clauses to indicate the type of `INSERT` to be executed.

Multitable INSERT Statements

Syntax:

```
INSERT [ALL] [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]
```


Multitable INSERT Statements

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.

Multitable INSERT Statements (continued)

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.
```


Unconditional INSERT ALL

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

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

The feedback 12 rows created can be interpreted to mean that a total of eight insertions were performed on the base tables, SAL_HISTORY and MGR_HISTORY.

Conditional 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.
- If the SALARY is greater than $10,000, insert these values into the SAL_HISTORY table using a conditional multitable INSERT statement.
- If the MANAGER_ID is greater than 200, insert these values into the MGR_HISTORY table using a conditional multitable INSERT statement.

ORACLE

Conditional INSERT ALL

The problem statement for a conditional INSERT ALL statement is specified in the slide. The solution to this problem is shown on the next page.

Conditional INSERT ALL

```
INSERT ALL
  WHEN SAL > 10000 THEN
    INTO sal_history VALUES(EMPID,HIREDATE,SAL)
  WHEN MGR > 200   THEN
    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;
4 rows inserted.
```


Conditional INSERT ALL (continued)

The example in the slide is similar to the example in the previous slide because it inserts rows into both the SAL_HISTORY and the MGR_HISTORY tables. The SELECT statement retrieves the employee ID, hire date, salary, and manager ID of those employees whose employee ID is greater than 200 from the EMPLOYEES table. The employee ID, hire date, and salary are inserted into the SAL_HISTORY table. The employee ID, manager ID, and salary are inserted into the MGR_HISTORY table.

This INSERT statement is referred to as a conditional ALL INSERT 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 value of the SAL column is more than 10,000 are inserted in the SAL_HISTORY table, and similarly, only those rows where the value of the MGR column is more than 200 are inserted in the MGR_HISTORY table.

Observe that unlike the previous example (where eight rows were inserted into the tables), in this example, only four rows are inserted.

The feedback 4 rows created can be interpreted to mean that a total of four inserts were performed on the base tables, SAL_HISTORY and MGR_HISTORY.

Conditional **INSERT FIRST**

Select the DEPARTMENT_ID, SUM(SALARY), and MAX(HIRE_DATE) from the EMPLOYEES table.
If the SUM(SALARY) is greater than $25,000, insert these values into the SPECIAL_SAL by using a conditional FIRST multitable INSERT.
If the first WHEN clause evaluates to true, the subsequent WHEN clauses for this row should be skipped.
For the rows that do not satisfy the first WHEN condition, insert into the HIREDATE_HISTORY_00, HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIRE_DATE column using a conditional multitable INSERT.

Conditional **INSERT FIRST**

The problem statement for a conditional FIRST INSERT statement is specified in the slide. The solution to this problem is shown on the next page.

Conditional INSERT FIRST

```
INSERT FIRST
  WHEN SAL  > 25000          THEN
    INTO special_sal VALUES(DEPTID, SAL)
  WHEN HIREDATE like ('%00%') THEN
    INTO hiredate_history_00 VALUES(DEPTID,HIREDATE)
  WHEN HIREDATE like ('%99%') THEN
    INTO hiredate_history_99 VALUES(DEPTID, HIREDATE)
  ELSE
  INTO hiredate_history VALUES(DEPTID, HIREDATE)
  SELECT department_id DEPTID, SUM(salary) SAL,
         MAX(hire_date) HIREDATE
  FROM   employees
  GROUP BY department_id;
12 rows inserted.
```


Conditional INSERT FIRST (continued)

The example in the slide inserts rows into more than one table using a single INSERT statement. The SELECT statement retrieves the department ID, total salary, and maximum hire date for every department in the EMPLOYEES table.

This INSERT statement is referred to as a conditional FIRST INSERT because an exception is made for the departments whose total salary is more than $25,000. The condition WHEN ALL > 25000 is evaluated first. If the total salary for a department is more than $25,000, then the record is inserted into the SPECIAL_SAL table irrespective of the hire date. If this first WHEN clause evaluates to true, the Oracle server executes the corresponding INTO clause and skips subsequent WHEN clauses for this row.

For the rows that do not satisfy the first WHEN condition (WHEN SAL > 25000), the rest of the conditions are evaluated in the same way as a conditional INSERT statement, and the records retrieved by the SELECT statement are inserted into the HIREDATE_HISTORY_00, HIREDATE_HISTORY_99, or HIREDATE_HISTORY tables, based on the value in the HIREDATE column.

The feedback 12 rows created can be interpreted to mean that a total of eight INSERT statements were performed on the base tables, SPECIAL_SAL, HIREDATE_HISTORY_00, HIREDATE_HISTORY_99, and HIREDATE_HISTORY.

Pivoting `INSERT`

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`

Using a pivoting `INSERT`, convert the set of sales records from the nonrelational database table to relational format.

ORACLE

Pivoting `INSERT`

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.

To solve the problem mentioned in the slide, 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 problem statement for a pivoting `INSERT` statement is specified in the slide. 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.
```


Pivoting INSERT (continued)

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)

7 rows selected
```

Pivoting INSERT (continued)

```
SELECT * FROM SALES_SOURCE_DATA;
```

	EMPLOYEE_ID	WEEK_ID	SALES_MON	SALES_TUE	SALES_WED	SALES_THUR	SALES_FRI
1	176	6	2000	3000	4000	5000	6000

```
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	176	6	2000
2	176	6	3000
3	176	6	4000
4	176	6	5000
5	176	6	6000

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`.

MERGE Statement

- Provides the ability to conditionally update or insert 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

MERGE Statement

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 the 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 or update 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 = col_val1,
    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.

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

For more information, see *Oracle Database 10g SQL Reference*.

Merging Rows

Insert or update rows in the EMPL3 table to match the EMPLOYEES table.

```
MERGE INTO empl3  c
  USING employees e
  ON (c.employee_id = e.employee_ id)
WHEN MATCHED THEN
  UPDATE SET
     c.first_name     = e.first_name,
     c.last_name      = e.last_name,
     ...
     c.department_id  = e.department_id
WHEN NOT MATCHED THEN
 INSERT VALUES(e.employee_id, e.first_name, e.last_name,
          e.email, e.phone_number, e.hire_date,
          e.job_id, e.salary, e.commission_pct,
          e.manager_id, e.department_id);
```

ORACLE

Example of Merging Rows

```
MERGE INTO empl3 c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET
    c.first_name     = e.first_name,
    c.last_name      = e.last_name,
    c.email          = e.email,
    c.phone_number   = e.phone_number,
    c.hire_date      = e.hire_date,
    c.job_id         = e.job_id,
    c.salary         = e.salary,
    c.commission_pct = e.commission_pct,
    c.manager_id     = e.manager_id,
    c.department_id  = e.department_id
WHEN NOT MATCHED THEN
  INSERT VALUES(e.employee_id, e.first_name, e.last_name,
       e.email, e.phone_number, e.hire_date, e.job_id,
       e.salary, e.commission_pct, e.manager_id,
       e.department_id);
```

Merging Rows

```
TRUNCATE TABLE empl3;

SELECT *
FROM empl3;
no rows selected
```

```
MERGE INTO empl3 c
  USING employees e
  ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
  UPDATE SET
     ...
WHEN NOT MATCHED THEN
 INSERT VALUES...;
```

```
SELECT *
FROM empl3;

107 rows selected.
```

ORACLE

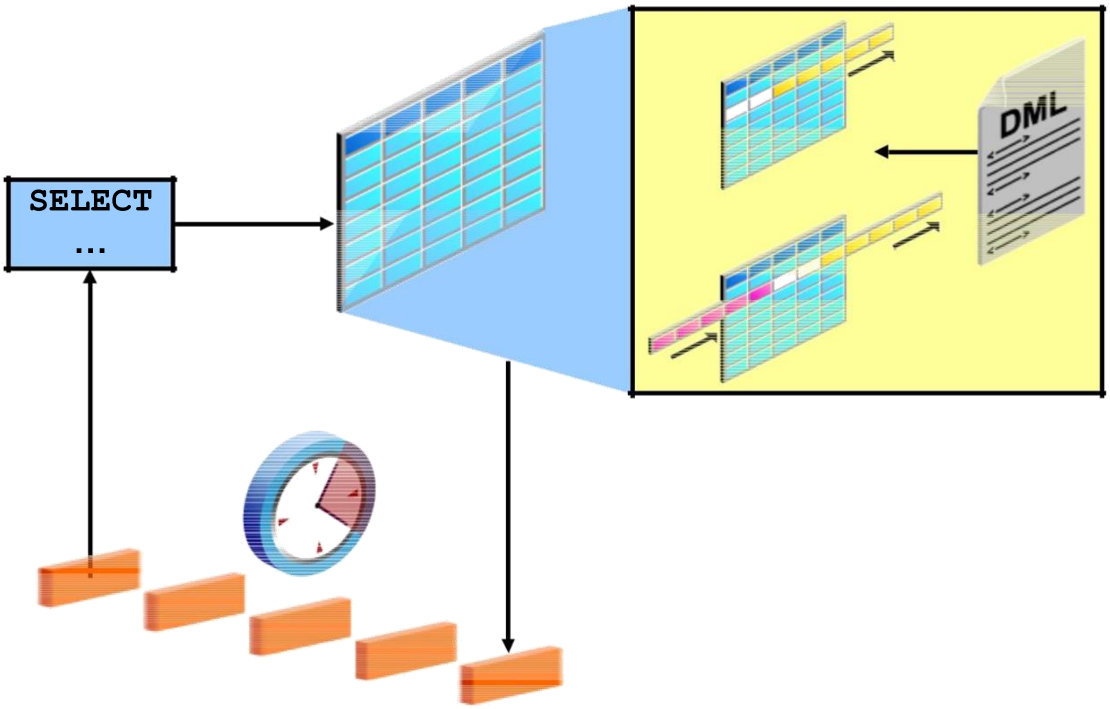
Example of Merging Rows (continued)

The example in the slide matches the EMPLOYEE_ID in the EMPL3 table to the EMPLOYEE_ID in the EMPLOYEES table. If a match is found, the row in the EMPL3 table is updated to match the row in the EMPLOYEES table. If the row is not found, it is inserted into the EMPL3 table.

The condition c.employee_id = e.employee_id is evaluated. Because the EMPL3 table is empty, the condition returns false—there are no matches. The logic falls into the WHEN NOT MATCHED clause, and the MERGE command inserts the rows of the EMPLOYEES table into the EMPL3 table.

If rows existed in the EMPL3 table and employee IDs matched in both tables (the EMPL3 and EMPLOYEES tables), then the existing rows in the EMPL3 table would be updated to match the EMPLOYEES table.

Tracking Changes in Data



Tracking Changes in Data

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 an SCN or time stamp range between 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 then 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 if 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 a system change number (SCN) to identify the redo records for each committed transaction.

Example of the Flashback Version Query

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

1

	SALARY
1	4200

```
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

	SALARY
1	5460
2	4200

ORACLE

Example of the Flashback Version Query

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, then 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.

Example of the Flashback Version Query (continued)

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	14-NOV-08 02.56.36.000000000 AM	(null)	5460
2	(null)	14-NOV-08 02.56.36.000000000 AM	4200

VERSIONS BETWEEN Clause

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/SCN of the BETWEEN clause, then 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 bound.

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

Summary

In this lesson, you should have learned how to:

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

Summary

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 3: Overview

This practice covers the following topics:

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

Practice 3: Overview

In this practice, you add rows to the `emp_data` table, update and delete data from the table, and track your transactions.

Practice 3

Run the `lab_03_01.sql` script in the lab folder to create the `SAL_HISTORY` table. Display the structure of the `SAL_HISTORY` table.

```
Name                           Null     Type
------------------------------ -------- ------------
EMPLOYEE_ID                             NUMBER(6)
HIRE_DATE                               DATE
SALARY                                  NUMBER(8,2)
```

Run the `lab_03_03.sql` script in the lab folder to create the `MGR_HISTORY` table. Display the structure of the `MGR_HISTORY` table.

```
Name                           Null     Type
------------------------------ -------- ------------
EMPLOYEE_ID                             NUMBER(6)
MANAGER_ID                              NUMBER(6)
SALARY                                  NUMBER(8,2)
```

Run the `lab_03_05.sql` script in the lab folder to create the `SPECIAL_SAL` table. Display the structure of the `SPECIAL_SAL` table.

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

a. Write a query to do the following:

Retrieve the employee ID, hire date, salary, and manager ID of those employees whose employee ID is less than 125 from the `EMPLOYEES` table.
If the salary is more than $20,000, insert the employee ID and salary into the `SPECIAL_SAL` table.
Insert the employee ID, hire date, and salary into the `SAL_HISTORY` table.
Insert the employee ID, manager ID, and salary into the `MGR_HISTORY` table.

b. Display the records from the `SPECIAL_SAL` table.

	EMPLOYEE_ID	SALARY
1	100	24000

Practice 3 (continued)

Display the records from the SAL_HISTORY table.

	EMPLOYEE_ID	HIRE_DATE	SALARY
1	101	21-SEP-89	17000
2	102	13-JAN-93	17000
3	103	03-JAN-90	9000
4	104	21-MAY-91	6000

…

24	124	16-NOV-99	5800

Display the records from the MGR_HISTORY table.

	EMPLOYEE_ID	MANAGER_ID	SALARY
1	101	100	17000
2	102	100	17000
3	103	102	9000
4	104	103	6000

…

24	124	100	5800

8. a. Run the `lab_03_08a.sql` script in the lab folder to create the SALES_SOURCE_DATA table.

   Run the `lab_03_08b.sql` script in the lab folder to insert records into the SALES_SOURCE_DATA table.

   Display the structure of the SALES_SOURCE_DATA table.

```
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)
```

d. Display the records from the SALES_SOURCE_DATA table.

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

Run the `lab_03_08c.sql` script in the lab folder to create the `SALES_INFO` table.

Display the structure of the `SALES_INFO` table.

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

Write a query to do the following:

Retrieve the employee ID, week ID, sales on Monday, sales on Tuesday, sales on Wednesday, sales on Thursday, and sales on Friday from the `SALES_SOURCE_DATA` table.

Build a transformation such that each record retrieved from the `SALES_SOURCE_DATA` table is converted into multiple records for the `SALES_INFO` table.

Hint: Use a pivoting `INSERT` statement.

h. Display the records from the `SALES_INFO` table.

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

You have the data of past employees stored in a flat file called `emp.data`. You want to store the names and email IDs of all the past and present employees in a table. To do this, first create an external table called `EMP_DATA` using the `emp.dat` source file in the `emp_dir` directory. You can use the script in `lab_03_09.sql` to do this.

Next, run the `lab_03_10.sql` script to create the `EMP_HIST` table.

Increase the size of the email column to 45.

Merge the data in the `EMP_DATA` table that was created in step 9 with the data in the `EMP_HIST` table. Assume that the data in the external `EMP_DATA` table is the most up-to-date. If a row in the `EMP_DATA` table matches the `EMP_HIST` table, update the email column of the `EMP_HIST` table to match the `EMP_DATA` table row. If a row in the `EMP_DATA` table does not match, insert it into the `EMP_HIST` table. Rows are considered matching when the employee's first and last names are identical.

Retrieve the rows from `EMP_HIST` after the merge.

	FIRST_NAME	LAST_NAME	EMAIL
1	Steven	King	SKING
2	Neena	Kochhar	nkochh@pipit.com
3	Lex	De Haan	LDEHAAN
4	Alexander	Hunold	AHun@MOORHEN.COM
5	Bruce	Ernst	BERNST
6	David	Austin	DAUSTIN
7	Valli	Pataballa	VPATABAL
8	Diana	Lorentz	DLORENTZ
9	Nancy	Greenberg	NGREENBE
10	Daniel	Faviet	DFAVIET
…			
145	Diana	lorentz	dlor@limpkin.com
146	Stephen	King	sking@merganser.com
147	Hema	Voight	Hema.Voight@PHALA...
148	Nancy	greenberg	ngreenb@plover.com

Create the EMP3 table by using the `lab_03_11.sql` script. In the EMP3 table, change the department for Kochhar to 60 and commit your change. Next, change the department for Kochhar to 50 and commit your change. Track the changes to Kochhar by using the Row Versions feature.

	START_DATE	END_DATE	DEPARTMENT_ID
1	17-NOV-08 03.55.06.000000000 AM	(null)	50
2	17-NOV-08 03.55.06.000000000 AM	17-NOV-08 03.55.06.000000000 AM	60
3	(null)	17-NOV-08 03.55.06.000000000 AM	90

4

Generating Reports by Grouping Related Data

Objectives

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

- Use the ROLLUP operation to produce subtotal values
- Use the CUBE operation to produce cross- tabulation values
- Use the GROUPING function to identify the row values created by ROLLUP or CUBE
- Use GROUPING SETS to produce a single result set

Objectives

In this lesson, you learn how to:

- Group data to obtain the following:
  - Subtotal values by using the ROLLUP operator
  - 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 approach

Review of Group Functions

- 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%';
```


Group Functions

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.

Review of the GROUP BY Clause

- 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 ;
```

ORACLE

Review of the GROUP BY Clause

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.

Review of the HAVING Clause

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];
```


HAVING Clause

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:

- It groups rows.
- 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.

GROUP BY with the ROLLUP and CUBE Operators

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 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];
```


ROLLUP Operator

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);
```

1

2

	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

3

Example of a ROLLUP Operator

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:
  - Total salary for each department whose department ID is less than 60
  - 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];
```


CUBE Operator

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. Whereas ROLLUP produces only a fraction of possible subtotal combinations, 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	1
2	(null)	HR_REP	6500	2
3	(null)	MK_MAN	13000	
4	(null)	MK_REP	6000	
5	(null)	PU_MAN	11000	
21	50	(null)	156400	3
22	50	ST_MAN	36400	
23	50	SH_CLERK	64300	4
24	50	ST_CLERK	55700	

Example of a CUBE Operator

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 those departments whose department ID is less than 60
- The total salary for every 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 alone, and 4 indicates some of the rows totaled by DEPARTMENT_ID and JOB_ID.

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. Additionally, 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];
```

ORACLE

GROUPING Function

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

1

2

3

ORACLE

Example of a GROUPING Function

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

GROUPING SETS is a further extension of the GROUP BY clause that 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 (which 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.

GROUPING SETS (continued)

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 groups (department_id, job_id, manager_id), (department_id, manager_id), and (job_id, manager_id) 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. 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
...				
49	30	PU_MAN	(null)	11000
50	10	AD_ASST	(null)	4400
51	20	MK_REP	(null)	6000
52	40	HR_REP	(null)	6500

1

2

ORACLE

GROUPING SETS: Example

The query in the slide calculates aggregates over two groupings. The table is divided into the following groups:

- Job ID, Manager ID
- Department ID, Job 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 3200.
- 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 `PU_MAN` job ID in department 30 is 11,000.
- The average salary of all employees with the `AD_ASST` job ID in department 10 is 4400, and so on.

`GROUPING SETS`: Example (continued)

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.

Composite Columns

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 if 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. This is generally used for generating grand totals.

```
SELECT   NULL, NULL, aggregate_col
FROM     <table_name>
GROUP BY ( );
```

Composite Columns (continued)

Compare this with the normal ROLLUP as in:

```
GROUP BY ROLLUP(a, b, c)
```

which 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)
```

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
...				
41	100	FI_ACCOUNT	108	39600
42	100	(null)	(null)	5160(
43	110	AC_MGR	101	12000
44	110	AC_ACCOUNT	205	8300
45	110	(null)	(null)	20300
46	(null)	(null)	(null)	69266(

1

2

3

4

ORACLE

Composite Columns: Example

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.

Composite Columns: Example (continued)

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:

```
SELECTdepartment_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 Columns

Concatenated groupings offer a concise way to generate useful combinations of groupings. The concatenated groupings are specified by listing multiple grouping sets, `CUBE`s, and `ROLLUP`s, 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

...

34	(null)	SA_REP	(null)	7000

...

90	100	(null)	(null)	51600
91	110	(null)	101	12000
92	110	(null)	205	8300
93	110	(null)	(null)	20300

ORACLE

Concatenated Groupings: Example

The example in the slide results in the following groupings:

`(job_id, manager_id)` (1)
`(department_id,job_id, manager_id)` (2)
`(job_id)`(3)
`(department_id,manager_id)`(4)
`(department_id)` (5)

The total salary for each of these groups is calculated.

Summary

In this lesson, 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

Summary

- `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.

Practice 4: Overview

This practice covers using the following:

- `ROLLUP` operators
- `CUBE` operators
- `GROUPING` functions
- `GROUPING SETS`

Practice 4: Overview

In this practice, you use the `ROLLUP` and `CUBE` operators as extensions of the `GROUP BY` clause. You also use `GROUPING SETS`.

Practice 4

Write a query to display the following for those employees whose manager ID is less than 120:

Manager ID
Job ID and total salary for every job ID of employees who report to the same manager
Total salary of those managers
Total salary of those managers, irrespective of the job IDs

	MANAGER_ID	JOB_ID	SUM(SALARY)
1	100	AD_VP	34000
2	100	MK_MAN	13000
3	100	PU_MAN	11000
4	100	SA_MAN	61000
5	100	ST_MAN	36400
6	100	(null)	155400
7	101	AC_MGR	12000
8	101	FI_MGR	12000

...

18	108	(null)	39600
19	114	PU_CLERK	13900
20	114	(null)	13900
21	(null)	(null)	283860

Observe the output from question 1. Write a query using the `GROUPING` function to determine whether the `NULL` values in the columns corresponding to the `GROUP BY` expressions are caused by the `ROLLUP` operation.

	MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
1	100	AD_VP	34000	0	0
2	100	MK_MAN	13000	0	0
3	100	PU_MAN	11000	0	0
4	100	SA_MAN	61000	0	0
5	100	ST_MAN	36400	0	0

...

18	108	(null)	39600	0	1
19	114	PU_CLERK	13900	0	0
20	114	(null)	13900	0	1
21	(null)	(null)	283860	1	1

Practice 4 (continued)

Write a query to display the following for those employees whose manager ID is less than 120:

- Manager ID
- Job and total salary for every job for employees who report to the same manager
- Total salary of those managers
- Cross-tabulation values to display the total salary for every job, irrespective of the manager
- Total salary irrespective of all job titles

	MANAGER_ID	JOB_ID	SUM(SALARY)
1	(null)	(null)	283860
2	(null)	AD_VP	34000
3	(null)	AC_MGR	12000
4	(null)	FI_MGR	12000
5	(null)	HR_REP	6500
6	(null)	MK_MAN	13000
7	(null)	PR_REP	10000
8	(null)	PU_MAN	11000
9	(null)	SA_MAN	61000
10	(null)	ST_MAN	36400

...

23	101	FI_MGR	12000
24	101	HR_REP	6500
25	101	PR_REP	10000
26	101	AD_ASST	4400
27	102	(null)	9000
28	102	IT_PROG	9000
29	103	(null)	21060
30	103	IT_PROG	21060
31	108	(null)	39600
32	108	FI_ACCOUNT	39600
33	114	(null)	13900
34	114	PU_CLERK	13900

Practice 4 (continued)

Observe the output from question 3. Write a query using the `GROUPING` function to determine whether the `NULL` values in the columns corresponding to the `GROUP BY` expressions are caused by the `CUBE` operation.

	MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
1	(null)	(null)	283860	1	1
2	(null)	AD_VP	34000	1	0
3	(null)	AC_MGR	12000	1	0
4	(null)	FI_MGR	12000	1	0
5	(null)	HR_REP	6500	1	0
6	(null)	MK_MAN	13000	1	0

...

	MGR	JOB	SUM(SALARY)	GROUPING(MANAGER_ID)	GROUPING(JOB_ID)
20	100	ST_MAN	36400	0	0
21	101	(null)	44900	0	1
22	101	AC_MGR	12000	0	0
23	101	FI_MGR	12000	0	0
24	101	HR_REP	6500	0	0
25	101	PR_REP	10000	0	0
26	101	AD_ASST	4400	0	0
27	102	(null)	9000	0	1
28	102	IT_PROG	9000	0	0
29	103	(null)	21060	0	1
30	103	IT_PROG	21060	0	0
31	108	(null)	39600	0	1
32	108	FI_ACCOUNT	39600	0	0
33	114	(null)	13900	0	1
34	114	PU_CLERK	13900	0	0

Practice 4 (continued)

Using `GROUPING SETS`, write a query to display the following groupings:

```
department_id, manager_id, job_id
department_id, job_id
manager_id, job_id
```

The query should calculate the sum of the salaries for each of these groups.

	DEPARTMENT_ID	MANAGER_ID	JOB_ID	SUM(SALARY)
1	90	(null)	AD_PRES	24000
2	90	100	AD_VP	34000
3	20	100	MK_MAN	13000
4	30	100	PU_MAN	11000
5	80	100	SA_MAN	61000
6	50	100	ST_MAN	36400
7	110	101	AC_MGR	12000
8	100	101	FI_MGR	12000

...

74	20	(null)	MK_MAN	13000
75	100	(null)	FI_MGR	12000
76	60	(null)	IT_PROG	30060
77	30	(null)	PU_CLERK	13900
78	50	(null)	SH_CLERK	64300
79	100	(null)	FI_ACCOUNT	39600
80	70	(null)	PR_REP	10000
81	(null)	(null)	SA_REP	7000
82	30	(null)	PU_MAN	11000
83	40	(null)	HR_REP	6500
84	10	(null)	AD_ASST	4400
85	20	(null)	MK_REP	6000

5

Managing Data in Different Time Zones

Objectives

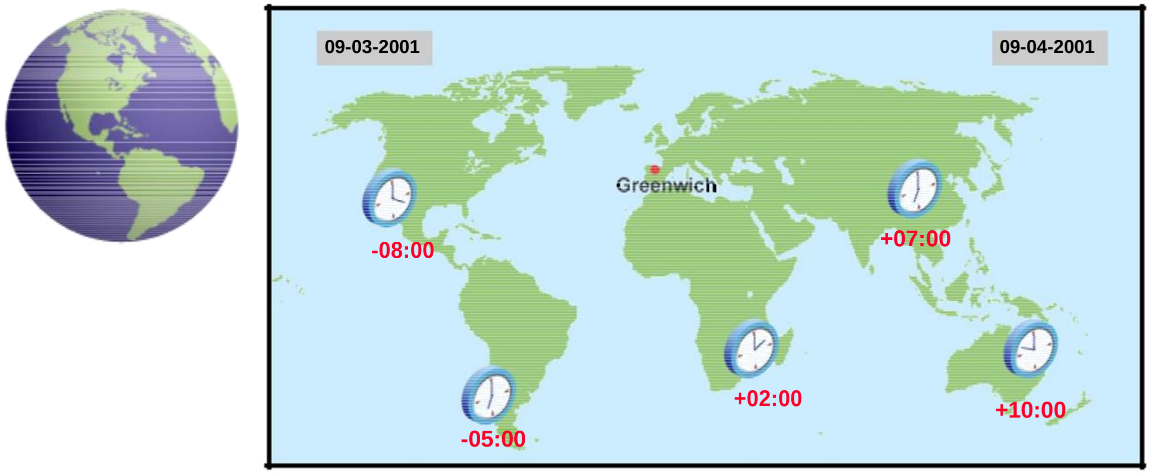
After completing this lesson, you should be able to use the following datetime functions:

- `TZ_OFFSET`
- `FROM_TZ`
- `TO_TIMESTAMP`
- `TO_TIMESTAMP_TZ`
- `TO_YMINTERVAL`
- `TO_DSINTERVAL`
- `CURRENT_DATE`
- `CURRENT_ TIMESTAMP`
- `LOCALTIMESTAMP`
- `DBTIMEZONE`
- `SESSIONTIMEZONE`
- `EXTRACT`

Objectives

This lesson addresses some of the datetime functions available in the Oracle Database.

Time Zones



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

Time Zones

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, or GMT. GMT 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 GMT 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';
```


TIME_ZONE Session Parameter

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 system
  – Has a data type of DATE
- CURRENT_TIMESTAMP:
  – Returns the current time stamp from the system
  – Has a data type of TIMESTAMP WITH TIME ZONE
- LOCALTIMESTAMP:
  – Returns the current time stamp from user session
  – Has a data type of TIMESTAMP

CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP

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 Coordinated Universal Time (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.

CURRENT_DATE

Display the current date and time in the session's time zone.

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

```
ALTER SESSION SET TIME_ZONE = '-5:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

	SESSIONTIMEZONE	CURRENT_DATE
1	-05:00	18-NOV-2008 02:34:51

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;
```

	SESSIONTIMEZONE	CURRENT_DATE
1	-08:00	17-NOV-2008 23:37:03

CURRENT_DATE

The CURRENT_DATE function returns the current date in the session's time zone. The return value is a date in the Gregorian calendar.

The examples in the slide illustrate that CURRENT_DATE is sensitive to the session time zone. In the first example, the session is altered to set the TIME_ZONE parameter to –5:0. 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 (Coordinated Universal Time, formerly known as Greenwich mean time).

Observe in the output that the value of CURRENT_DATE changes when the TIME_ZONE parameter value is changed to –8:0 in the second example.

Note: 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).

CURRENT_TIMESTAMP

Display the current date and fractional time in the session's time zone.

```
ALTER SESSION SET TIME_ZONE = '-5:0';
SELECT SESSIONTIMEZONE,
CURRENT_TIMESTAMP FROM DUAL;
```

	SESSIONTIMEZONE	CURRENT_TIMESTAMP
1	-05:00	18-NOV-08 04.08.46.109757000 AM -05:00

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE,
CURRENT_TIMESTAMP FROM DUAL;
```

	SESSIONTIMEZONE	CURRENT_TIMESTAMP
1	-08:00	18-NOV-08 01.07.04.018151000 AM -08:00

CURRENT_TIMESTAMP

The `CURRENT_TIMESTAMP` function returns the current date and time in the session time zone, as a value of the `TIMESTAMP WITH TIME ZONE` data type. The time zone displacement reflects the current local time of the SQL session. The syntax of the `CURRENT_TIMESTAMP` function is:

```
CURRENT_TIMESTAMP (precision)
```

where *precision* is an optional argument that specifies the fractional second precision of the time value returned. If you omit precision, the default is 6.

The examples in the slide illustrate that `CURRENT_TIMESTAMP` is sensitive to the session time zone. In the first example, the session is altered to set the `TIME_ZONE` parameter to –5:0. Observe in the output that the value of `CURRENT_TIMESTAMP` changes when the `TIME_ZONE` parameter value is changed to –8:0 in the second example.

LOCALTIMESTAMP

Display the current date and time in the session's time zone in a value of the TIMESTAMP data type.

```
ALTER SESSION SET TIME_ZONE = '-5:0';
SELECT CURRENT_TIMESTAMP,
LOCALTIMESTAMP FROM DUAL;
```

	CURRENT_TIMESTAMP	LOCALTIMESTAMP
1	18-NOV-08 04.10.19.678104000 AM -05:00	18-NOV-08 04.10.19.678104000 AM

```
ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT CURRENT_TIMESTAMP,
LOCALTIMESTAMP FROM DUAL;
```

	CURRENT_TIMESTAMP	LOCALTIMESTAMP
1	18-NOV-08 01.11.33.170179000 AM -08:00	18-NOV-08 01.11.33.170179000 AM

LOCALTIMESTAMP returns a TIMESTAMP value, whereas CURRENT_TIMESTAMP returns a TIMESTAMP WITH TIME ZONE value.

ORACLE

LOCALTIMESTAMP

The LOCALTIMESTAMP function returns the current date and time in the session time zone. LOCALTIMESTAMP returns a TIMESTAMP value. The syntax of the LOCALTIMESTAMP function is:

```
LOCALTIMESTAMP (TIMESTAMP_precision)
```

where TIMESTAMP precision is an optional argument that specifies the fractional second precision of the TIMESTAMP value returned.

The examples in the slide illustrate the difference between LOCALTIMESTAMP and CURRENT_TIMESTAMP. Observe that LOCALTIMESTAMP does not display the time zone value, whereas the CURRENT_TIMESTAMP does.

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	-08:00

ORACLE

DBTIMEZONE and SESSIONTIMEZONE

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 Type

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

It stores the year, month, and day of the DATE data type, plus hour, minute, and second values, as well as the fractional second value.

Variations in TIMESTAMP are:

- TIMESTAMP [(fractional_seconds_precision)]_
- TIMESTAMP [(fractional_seconds_precision)]_ WITH TIME ZONE
- TIMESTAMP [(fractional_seconds_precision)]_ WITH LOCAL TIME ZONE

Datetime Data Types

The TIMESTAMP data type contains the datetime fields: YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND and fractional seconds.

The TIMESTAMP WITH TIME ZONE data type contains the datetime fields: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, and TIMEZONE_MINUTE and fractional seconds.

The TIMESTAMP WITH LOCAL TIME ZONE data type contains the same information as the TIMESTAMP data type, except that the data is normalized to the database time zone when stored, and adjusted to match the client's time zone when retrieved.

Note: Fractional second precision 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.

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

TIMESTAMP Data Types

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

`TIMESTAMP` Fields

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 emp5;
```

	HIRE_DATE
1	17-JUN-1987 00:00:00
2	21-SEP-1989 00:00:00
3	13-JAN-1993 00:00:00
4	03-JAN-1990 00:00:00
5	21-MAY-1991 00:00:00
6	25-JUN-1997 00:00:00

…

B

```
ALTER TABLE emp5
MODIFY hire_date TIMESTAMP;

SELECT hire_date
FROM emp5;
```

	HIRE_DATE
1	17-JUN-87 12.00.00.000000000 AM
2	21-SEP-89 12.00.00.000000000 AM
3	13-JAN-93 12.00.00.000000000 AM
4	03-JAN-90 12.00.00.000000000 AM
5	21-MAY-91 12.00.00.000000000 AM
6	25-JUN-97 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 `EMP5` 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, then it defaults to 6.

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

```
ALTER TABLE emp5
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, a.m., and p.m. To obtain the date in this format, you can apply a format mask or a function to the date value.

TIMESTAMP WITH TIME ZONE Data Type

TIMESTAMP WITH TIME ZONE is a variant of TIMESTAMP that includes a time zone displacement in its value.

The time zone displacement is the difference, in hours and minutes, between local time and UTC.

It is specified as:

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


TIMESTAMP WITH TIME ZONE Data Type

UTC stands for Coordinated Universal Time (formerly Greenwich mean time). Two TIMESTAMP WITH TIME ZONE values are considered identical if they represent the same instant in UTC, regardless of the TIME ZONE offsets stored in the data. For example:

```
TIMESTAMP '1999-04-15 8:00:00 -8:00'
```

is the same as

```
TIMESTAMP '1999-04-15 11:00:00 -5:00'.
```

That is, 8:00 a.m. Pacific Standard Time is the same as 11:00 a.m. Eastern Standard Time.

This can also be specified as:

```
TIMESTAMP '1999-04-15 8:00:00 US/Pacific'
```

TIMESTAMP WITH TIMEZONE: Example

```
CREATE TABLE web_orders
(ord_id number primary key,
 order_date TIMESTAMP WITH TIME ZONE);
```

```
INSERT INTO web_orders values
(ord_seq.nextval, current_date);
```

```
SELECT * FROM web_orders;
```

	ORD_ID	ORDER_DATE
1	100	18-NOV-08 01.27.52.000000000 AM -08:00

TIMESTAMP WITH TIME ZONE: Example

In the example in the slide, a new table web_orders is created with a column of data type TIMESTAMP WITH TIME ZONE. This table is populated whenever a web_order is placed. The time stamp and time zone for the user placing the order is inserted based on the CURRENT_DATE value. That way when a Web-based company guarantees shipping, they can estimate their delivery time based on the time zone of the person placing the order.

TIMESTAMP WITH LOCAL TIMEZONE

- TIMESTAMP WITH LOCAL TIME ZONE is another variant of TIMESTAMP that includes a time zone displacement in its value.
- Data stored in the database is normalized to the database time zone.
- The time zone displacement is not stored as part of the column data.
- The Oracle Database returns the data in the user's local session time zone.
- The TIMESTAMP WITH LOCAL TIME ZONE data type is specified as follows:

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


TIMESTAMP WITH LOCAL TIMEZONE

Unlike TIMESTAMP WITH TIME ZONE, you can specify columns of the TIMESTAMP WITH LOCAL TIME ZONE type as part of a primary or unique key. The time zone displacement is the difference (in hours and minutes) between local time and UTC. There is no literal for TIMESTAMP WITH LOCAL TIME ZONE.

TIMESTAMP WITH LOCAL TIMEZONE: Example

```
CREATE TABLE shipping (delivery_time TIMESTAMP WITH
LOCAL TIME ZONE);
INSERT INTO shipping VALUES(current_timestamp + 2);
```

```
SELECT * FROM shipping;
```

	DELIVERY_TIME
1	20-NOV-08 01.30.15.000000000 AM

```
ALTER SESSION SET TIME_ZONE =
'EUROPE/LONDON'; SELECT * FROM shipping;
```

	DELIVERY_TIME
1	20-NOV-08 09.30.15.000000000 AM

TIMESTAMP WITH LOCAL TIME ZONE: Example

In the example in the slide, a new table SHIPPING is created with a column of the TIMESTAMP WITH LOCAL TIME ZONE data type. This table is populated by inserting two days from the CURRENT_TIMESTAMP value into it every time an order is placed. The output from the DATE_TAB table shows that the data is stored without the time zone offset. Then the ALTER SESSION command is issued to change the time zone to the local time zone at the place of delivery. A second query on the same table now reflects the data with the local time zone reflected in the time value, so that the customer can be notified about the expected delivery time.

`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**

ORACLE

`INTERVAL` Data Types

`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` Data Types (continued)

`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 to 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 to 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 to 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 Fields

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` Data Type

`INTERVAL YEAR TO MONTH` stores a period of time using the `YEAR` and `MONTH` datetime fields.

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

For example:

```
'312-2' assigned to INTERVAL YEAR(3) TO MONTH
Indicates an interval of 312 years and 2 months
```

```
'312-0' assigned to INTERVAL YEAR(3) TO
MONTH Indicates 312 years and 0 months
```

```
'0-3' assigned to INTERVAL YEAR TO
MONTH Indicates an interval of 3 months
```


`INTERVAL YEAR TO MONTH` Data Type

`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.

The following `INTERVAL YEAR TO MONTH` literal indicates an interval of 123 years, 3 months:

```
INTERVAL '123-3' YEAR(3) TO MONTH
```

- `INTERVAL '123' YEAR(3)` indicates an interval of 123 years 0 months.
- `INTERVAL '3' MONTH` indicates an interval of 3 months.

`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` Data Type (continued)

`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.

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 displaced by the month value by a (-).

INTERVAL DAY TO SECOND Data Type

INTERVAL DAY TO SECOND (fractional _seconds_precision) stores a period of time in days, hours, minutes, and seconds.

```
INTERVAL DAY[(day_precision)] TO Second
```

For example:

```
INTERVAL '6 03:30:16' DAY TO SECOND

Indicates an interval of 6 days 3 hours 30
minutes and 16 seconds
```

```
INTERVAL '6 00:00:00' DAY TO SECOND

Indicates an interval of 6 days and 0 hours, 0
minutes and 0 seconds
```


INTERVAL DAY TO SECOND Data Type

`INTERVAL DAY (day_precision) TO SECOND` (`fractional_seconds_precision`) 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. Accepted values are 0 through 9. The default is 6.

In the example in the slide, 6 represents the number of days, and 03:30:15 indicates the values for hours, minutes, and seconds.

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

INTERVAL DAY TO SECOND Data Type: Example

In the example in the slide, you are creating 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, minutes, and seconds, and `INTERVAL '6 03:30:16' DAY TO SECOND`. The select statement shows how this data is displayed in the database.

EXTRACT

Display the YEAR component from the SYSDATE.

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

	EXTRACT(YEARFROMSYSDATE)
1	2008

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	Kochhar	21-SEP-1989 00:00:00	9
2	De Haan	13-JAN-1993 00:00:00	1
3	Raphaely	07-DEC-1994 00:00:00	12
4	Weiss	18-JUL-1996 00:00:00	7

...

EXTRACT

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'` time zone:

```
SELECT TZ_OFFSET('US/Eastern') FROM DUAL;
```

	TZ_OFFSET('US/EASTERN')
1	-05:00

Display the time zone offset for the `'Canada/Yukon'` time zone:

```
SELECT TZ_OFFSET('Canada/Yukon') FROM DUAL;
```

	TZ_OFFSET('CANADA/YUKON')
1	-08:00

Display the time zone offset for the `'Europe/London'` time zone:

```
SELECT TZ_OFFSET('Europe/London') FROM DUAL;
```

	TZ_OFFSET('EUROPE/LONDON')
1	+00:00

ORACLE

TZ_OFFSET

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 five hours behind UTC.
The `'Canada/Yukon'` time zone is eight hours behind UTC.
The `'Europe/London'` time zone is same as UTC.

TZ_OFFSET (continued)

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/Algiers	LMT
2	Africa/Algiers	PMT
3	Africa/Algiers	WET
4	Africa/Algiers	WEST
5	Africa/Algiers	CET
6	Africa/Algiers	CEST
7	Africa/Cairo	LMT
8	Africa/Cairo	EET
9	Africa/Cairo	EEST
10	Africa/Casablanca	LMT
11	Africa/Casablanca	WET
12	Africa/Casablanca	WEST
13	Africa/Casablanca	CET
14	Africa/Ceuta	LMT
15	Africa/Ceuta	WET

...

TIMESTAMP Conversion Using FROM_TZ

Display the TIMESTAMP value '2000-03-28 08:00:00' as a TIMESTAMP WITH TIME ZONE value.

```
SELECT FROM_TZ(TIMESTAMP '2000-03-
       28 08:00:00','3:00')
FROM DUAL;
```

	FROM_TZ(TIMESTAMP'2000-03-2808:00:00','3:00')
1	28-MAR-00 08.00.00.000000000 AM +03:00

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
       '2000-03-28 08:00:00', 'Australia/North')
FROM DUAL;
```

	FROM_TZ(TIMESTAMP'2000-03-2808:00:00','AUSTRALIA/NORTH')
1	28-MAR-00 08.00.00.000000000 AM AUSTRALIA/NORTH

TIMESTAMP Conversion Using FROM_TZ

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

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

Converting to **TIMESTAMP** Using TO_TIMESTAMP and **TO_TIMESTAMP_TZ**

Display the character string `'2000-12-01 11:00:00'` as a `TIMESTAMP` value:

```
SELECT TO_TIMESTAMP ('2000-12-01
11:00:00', 'YYYY-MM-DD HH:MI:SS')
FROM DUAL;
```

	TO_TIMESTAMP('2000-12-0111:00:00','YYYY-MM-DDHH:MI:SS')
1	01-DEC-00 11.00.00.000000000 AM

Display the character string `'1999-12-01 11:00:00 -8:00'` as a `TIMESTAMP WITH TIME ZONE` value:

```
SELECT
  TO_TIMESTAMP_TZ('1999-12-01 11:00:00 -8:00',
                  'YYYY-MM-DD HH:MI:SS TZH:TZM')
FROM DUAL;
```

	TO_TIMESTAMP_TZ('1999-12-0111:00:00-8:00','YYYY-MM-DDHH:MI:SSTZH:TZM')
1	01-DEC-99 11.00.00.000000000 AM -08:00

Converting to **TIMESTAMP** Using **TO_TIMESTAMP** and **TO_TIMESTAMP_TZ**

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`, then 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`.

The `TO_TIMESTAMP_TZ` function converts a string of `CHAR`, `VARCHAR2`, `NCHAR`, or `NVARCHAR2` data type to a value of the `TIMESTAMP WITH TIME ZONE` data type. The syntax of the `TO_TIMESTAMP_TZ` function is:

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

The optional `fmt` specifies the format of `char`. If you omit `fmt`, then a string must be in the default format of the `TIMESTAMP WITH TIME ZONE` data type. The example in the slide converts a character string to a value of `TIMESTAMP WITH TIME ZONE`.

Time Interval Conversion with 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

ORACLE

Time Interval Conversion with TO_YMINTERVAL

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.

A reverse calculation can also be done using the TO_YMINTERVAL function. For example:

```
SELECT hire_date, hire_date + TO_YMINTERVAL('-02-04') AS
       HIRE_DATE_YMINTERVAL
FROM EMPLOYEES WHERE department_id = 20;
```

Observe that the character string passed to the TO_YMINTERVAL function has a negative value. The example returns a date that is two years and four months before the hire date for the employees working in the department 20 of the EMPLOYEES table.

Using TO_DSINTERVAL: Example

TO_DSINTERVAL: Converts a character string to an INTERVAL DAY TO SECOND data type

```
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	King	06-17-87:12:00:00	09-25-87:10:00:00
2	Kochhar	09-21-89:12:00:00	12-30-89:10:00:00
3	De Haan	01-13-93:12:00:00	04-23-93:10:00:00
4	Hunold	01-03-90:12:00:00	04-13-90:10:00:00
5	Ernst	05-21-91:12:00:00	08-29-91:10:00:00
6	Austin	06-25-97:12:00:00	10-03-97:10:00:00
7	Pataballa	02-05-98:12:00:00	05-16-98:10:00:00
8	Lorentz	02-07-99:12:00:00	05-18-99:10:00:00

...

ORACLE

TO_DSINTERVAL

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

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

TO_YMINTERVAL

The TO_YMINTERVAL function converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to an INTERVAL YEAR TO MONTH type.

In the following example, the date one year and two months after the hire date is obtained.

```
SELECT hire_date, hire_date + TO_YMINTERVAL('01-02') ytm
FROM employees;
```

	HIRE_DATE	YTM
1	17-JUN-1987 00:00:00	17-AUG-1988 00:00:00
2	21-SEP-1989 00:00:00	21-NOV-1990 00:00:00
3	13-JAN-1993 00:00:00	13-MAR-1994 00:00:00
4	03-JAN-1990 00:00:00	03-MAR-1991 00:00:00
5	21-MAY-1991 00:00:00	21-JUL-1992 00:00:00

...

Daylight Saving Time

First Sunday in April

- Time jumps from 01:59:59 a.m. to 03:00:00 a.m.
- Values from 02:00:00 a.m. to 02:59:59 a.m. are not valid.

Last Sunday in October

- Time jumps from 02:00:00 a.m. to 01:00:01 a.m.
- Values from 01:00:01 a.m. to 02:00:00 a.m. are ambiguous because they are visited twice.

ORACLE

Daylight Saving Time (DST)

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 first Sunday in April to the last Sunday in October 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 a.m. to 03:00:00 a.m. The one-hour interval between 02:00:00 and 02:59:59 a.m. does not exist. When daylight saving time goes out of effect, the time changes from 02:00:00 a.m. back to 01:00:01 a.m., and the one-hour interval between 01:00:01 and 02:00:00 a.m. is repeated.

Daylight Saving Time (DST) (continued)

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 a.m. The overlapped periods are:

10/31/2004 01:00:01 a.m. to 10/31/2004 02:00:00 a.m. (EDT)
10/31/2004 01:00:01 a.m. to 10/31/2004 02:00:00 a.m. (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 parameter ERROR_ON_OVERLAP_TIME is FALSE, then it assumes that the input time is standard time (for example, EST). Otherwise, an error is raised.

Summary

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

- TZ_OFFSET
- FROM_TZ
- TO_TIMESTAMP
- TO_TIMESTAMP_TZ
- TO_YMINTERVAL
- CURRENT_DATE
- CURRENT_TIMESTAMP
- LOCALTIMESTAMP
- DBTIMEZONE
- SESSIONTIMEZONE
- EXTRACT

Summary

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

Practice 5: Overview

This practice covers using the datetime functions.

Practice 5: Overview

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

Practice 5

Alter the session to set the NLS_DATE_FORMAT to DD-MON-YYYY HH24:MI:SS.

a . Write queries to display the time zone offsets (TZ_OFFSET) for the following time zones.

US/Pacific-New

	TZ_OFFSET('US/PACIFIC-NEW')
1	-08:00

Singapore

	TZ_OFFSET('SINGAPORE')
1	+08:00

Egypt

	TZ_OFFSET('EGYPT')
1	+02:00

Alter the session to set the TIME_ZONE parameter value to the time zone offset of US/Pacific-New.

Display the CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP for this session.

	CURRENT_DATE	CURRENT_TIMESTAMP	LOCALTIMESTAMP
1	18-NOV-2008 03:17:31	18-NOV-08 03.17.31.389656000 AM -07:00	18-NOV-08 03.17.31.389656000 AM

Alter the session to set the TIME_ZONE parameter value to the time zone offset of Singapore.

Display the CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP for this session.

Note: The output may be different based on the date when the command is executed.

	CURRENT_DATE	CURRENT_TIMESTAMP	LOCALTIMESTAMP
1	18-NOV-2008 18:19:42	18-NOV-08 06.19.42.486210000 PM +08:00	18-NOV-08 06.19.42.486210000 PM

Note: Observe in the preceding practice that CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP are all sensitive to the session time zone.

Write a query to display DBTIMEZONE and SESSIONTIMEZONE.

	DBTIMEZONE	SESSIONTIMEZONE
1	+00:00	+08:00

Write a query to extract `YEAR` from the `HIRE_DATE` column of the `EMPLOYEES` table for those employees who work in department 80.

	LAST_NAME	EXTRACT(YEARFROMHIRE_DATE)
1	Russell	1996
2	Partners	1997
3	Errazuriz	1997
4	Cambrault	1999
5	Zlotkey	2000
6	Tucker	1997
7	Bernstein	1997

…

30	Abel	1996
31	Hutton	1997
32	Taylor	1998
33	Livingston	1998
34	Johnson	2000

Alter the session to set `NLS_DATE_FORMAT` to `DD-MON-YYYY`.

Examine and run the `lab_05_06.sql` script to create the `SAMPLE_DATES` table and populate it.

Select from the table and view the data.

	DATE_COL
1	18-NOV-2008

Change the data type of the `DATE_COL` column to `TIMESTAMP`. Select from the table to view the data.

	DATE_COL
1	18-NOV-08 05.31.54.000000000 AM

Try to change the data type of the `DATE_COL` column to `TIMESTAMP WITH TIME ZONE`. What happens?

Create a query to retrieve last names from the `EMPLOYEES` table and calculate the review status. If the year hired is 1998, display `Needs Review` for the review status; otherwise, display `not this year!` Name the review status column `Review`. Sort the results by the `HIRE_DATE` column.

Hint: Use a `CASE` expression with the `EXTRACT` function to calculate the review status.

	LAST_NAME	Review
1	King	not this year!
2	Whalen	not this year!
3	Kochhar	not this year!
4	Hunold	not this year!
5	Ernst	not this year!
6	De Haan	not this year!
7	Mavris	not this year!
8	Baer	not this year!

…

65	Olsen	Needs Review
66	Patel	Needs Review
67	Livingston	Needs Review
68	Walsh	Needs Review
69	Feeney	Needs Review
70	Dellinger	Needs Review
71	McCain	Needs Review
72	Vargas	Needs Review
73	Gates	Needs Review
74	Rogers	Needs Review

…

105	Ande	not this year!
106	Banda	not this year!
107	Kumar	not this year!

Practice 5 (continued)

Create a query to print the last names and number of years of service for each employee. If the employee has been employed for five or more years, print `5 years of service`. If the employee has been employed for 10 or more years, print `10 years of service`. If the employee has been employed for 15 or more years, print `15 years of service`. If none of these conditions match, print `maybe next year!` Sort the results by the `HIRE_DATE` column. Use the `EMPLOYEES` table.

Hint: Use `CASE` expressions and `TO_YMINTERVAL`.

	LAST_NAME	HIRE_DATE	SYSDATE	Awards
1	King	17-JUN-1987	18-NOV-2008	15 years of service
2	Kochhar	21-SEP-1989	18-NOV-2008	15 years of service
3	De Haan	13-JAN-1993	18-NOV-2008	15 years of service
4	Hunold	03-JAN-1990	18-NOV-2008	15 years of service
5	Ernst	21-MAY-1991	18-NOV-2008	15 years of service
6	Austin	25-JUN-1997	18-NOV-2008	10 years of service
7	Pataballa	05-FEB-1998	18-NOV-2008	10 years of service
8	Lorentz	07-FEB-1999	18-NOV-2008	5 years of service
9	Greenberg	17-AUG-1994	18-NOV-2008	10 years of service
10	Faviet	16-AUG-1994	18-NOV-2008	10 years of service
11	Chen	28-SEP-1997	18-NOV-2008	10 years of service
12	Sciarra	30-SEP-1997	18-NOV-2008	10 years of service
13	Urman	07-MAR-1998	18-NOV-2008	10 years of service
14	Popp	07-DEC-1999	18-NOV-2008	5 years of service
15	Raphaely	07-DEC-1994	18-NOV-2008	10 years of service

…

94	Everett	03-MAR-1997	18-NOV-2008	10 years of service
95	McCain	01-JUL-1998	18-NOV-2008	10 years of service
96	Jones	17-MAR-1999	18-NOV-2008	5 years of service
97	Walsh	24-APR-1998	18-NOV-2008	10 years of service
98	Feeney	23-MAY-1998	18-NOV-2008	10 years of service
99	OConnell	21-JUN-1999	18-NOV-2008	5 years of service
100	Grant	13-JAN-2000	18-NOV-2008	5 years of service
101	Whalen	17-SEP-1987	18-NOV-2008	15 years of service
102	Hartstein	17-FEB-1996	18-NOV-2008	10 years of service
103	Fay	17-AUG-1997	18-NOV-2008	10 years of service
104	Mavris	07-JUN-1994	18-NOV-2008	10 years of service
105	Baer	07-JUN-1994	18-NOV-2008	10 years of service
106	Higgins	07-JUN-1994	18-NOV-2008	10 years of service
107	Gietz	07-JUN-1994	18-NOV-2008	10 years of service

Retrieving Data Using Subqueries

Objectives

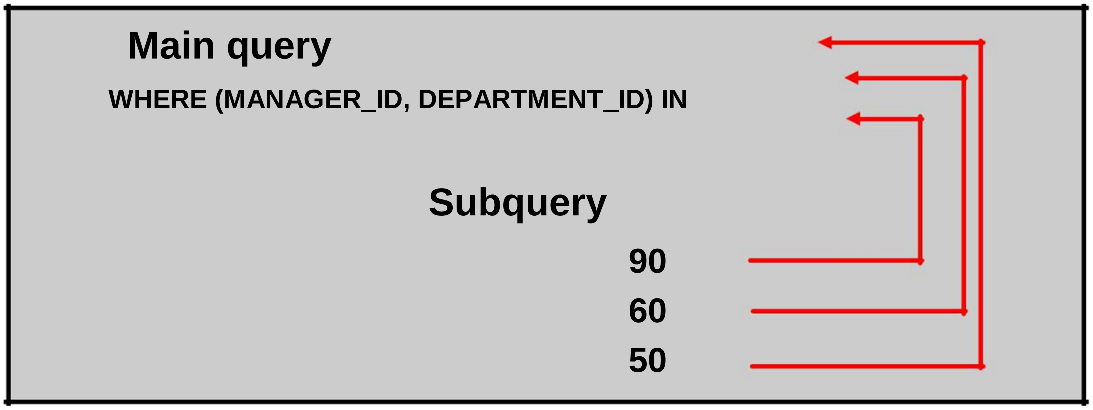
After completing this lesson, you should be able to do the following:

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

Objectives

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.

Multiple-Column Subqueries



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

Multiple-Column Subqueries

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.

Column Comparisons

Multiple-column comparisons involving subqueries can be:

- Nonpairwise comparisons
- Pairwise comparisons

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 employees
WHERE manager_id IN (SELECT manager_id
                     FROM employees
                     WHERE first_name = ‘Daniel’)
AND department_id IN (SELECT department_id
                      FROM employees
                      WHERE first_name = ‘Daniel’);
```

There is only one “Daniel” in the `EMPLOYEES` 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 `manger_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     employees
WHERE   (manager_id, department_id) IN
                         (SELECT manager_id, department_id
                          FROM employees
                          WHERE first_name = 'John')
AND first_name <> 'John';
```

ORACLE

Pairwise Comparison Subquery

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 EMPLOYEES 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 EMPLOYEES 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	137	123	50
2	138	123	50
3	140	123	50

...

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


Nonpairwise Comparison Subquery

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 EMPLOYEES table. If the MANAGER_ID column of the row in the EMPLOYEES 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 EMPLOYEES 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 slide:

	EMPLOYEE_ID	MANAGER_ID	DEPARTMENT_ID
1	120	100	50
2	121	100	50
3	122	100	50
4	123	100	50

...

This query retrieves five rows more than the pairwise comparison (those with the combination of manager_id=100 and department_id=50, although no employee named “John” has such a combination).

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:

- Condition and expression part of DECODE and CASE
- All clauses of SELECT except GROUP BY

Scalar Subqueries in SQL

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, 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
                                                 20
        WHEN department_id =
                    (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);
```


Scalar Subqueries: Examples

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:

	EMPLOYEE_ID	LAST_NAME	LOCATION
1	100	King	USA
2	101	Kochhar	USA
3	102	De Haan	USA
4	103	Hunold	USA
5	104	Ernst	USA
6	105	Austin	USA
7	106	Pataballa	USA

...

107	206	Gietz	USA

Scalar Subqueries: Examples (continued)

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	109	Faviet
8	108	Greenberg
9	112	Urman
10	111	Sciarra
11	110	Chen
12	113	Popp
13	203	Mavris

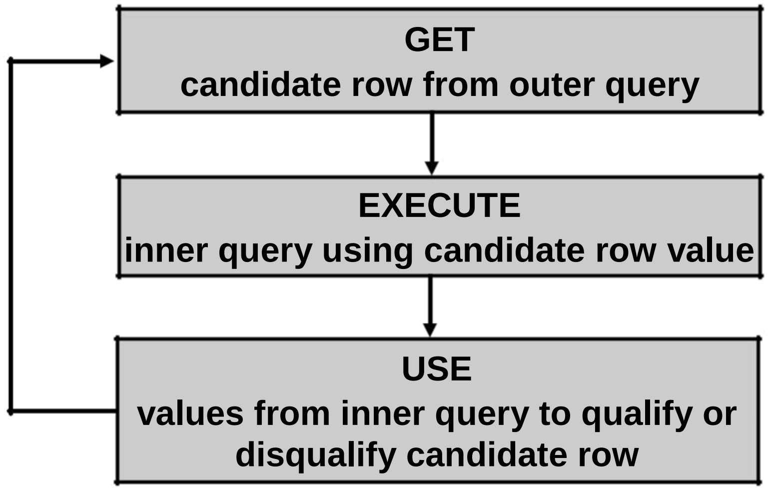
...

96	134	Rogers
97	135	Gee
98	136	Philtanker
99	137	Ladwig
100	138	Stiles
101	139	Seo
102	140	Patel
103	141	Rajs
104	142	Davies
105	143	Matos
106	181	Fleaur
107	178	Grant

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.

Correlated Subqueries

Correlated subqueries are used for row-by -row processing. Each subquery is executed once for every row of the outer query.



Correlated Subqueries

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 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
WHERE  column1 operator
                       (SELECT   column1, column2
                        FROM     table2
                        WHERE    expr1 =
                                   outer.expr2);
```


Correlated Subqueries (continued)

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
WHERE  salary >
                (SELECT AVG(salary)
                 FROM   employees
                 WHERE  department_id =
                 outer.department_id) ;
```

Each time a row from the outer query is processed, the inner query is evaluated.

Using Correlated Subqueries

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	101	Kochhar	AD_VP
2	176	Taylor	SA_REP
3	200	Whalen	AD_ASST

Using Correlated Subqueries (continued)

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:

Select a row from the table specified in the outer query. This will be the current candidate row.
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`.)
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.)
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.)
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 the `EMPLOYEE_ID` from the table in the outer query.

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

EXISTS Operator

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.

Find Employees Who Have At Least One Person Reporting to Them

```
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	100	King	AD_PRES	90
2	101	Kochhar	AD_VP	90
3	102	De Haan	AD_VP	90
4	103	Hunold	IT_PROG	60
5	108	Greenberg	FI_MGR	100
6	114	Raphaely	PU_MAN	30
7	120	Weiss	ST_MAN	50
8	121	Fripp	ST_MAN	50
...				
18	205	Higgins	AC_MGR	110

ORACLE

Using the `EXISTS` Operator

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
...		
15	260	Recruiting
16	270	Payroll

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);
0 rows selected.
```

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);
```


Correlated UPDATE

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);
```

ORACLE

Correlated **UPDATE** (continued)

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);
```

Correlated UPDATE (continued)

This example uses the REWARDS table. The REWARDS table has the columns EMPLOYEE_ID, PAY_RAISE, and PAYRAISE_DATE. Every time an employee gets a pay raise, a record with the details of 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);
```


Correlated DELETE

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, then 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 emp_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);
```

ORACLE

Correlated DELETE (continued)

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.

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 improves performance.

WITH Clause

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.

`WITH` Clause: Example

The problem in the slide would require the following intermediate calculations:

- Calculate the total salary for every department, and store the result using a `WITH` clause.
- Calculate the average salary across departments, and store the result using a `WITH` clause.
- 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, then 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;
```


WITH Clause: Example (continued)

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 in-line 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.

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

Summary

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:

- 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

Summary (continued)

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 reevaluate the query block and it occurs more than once within a complex query.

Practice 6: 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

ORACLE

Practice 6: Overview

In this practice, you write multiple-column subqueries, and correlated and scalar subqueries. You also solve problems by using the `WITH` clause.

Practice 6

Write a query to display the last name, department number, and salary of any employee whose department number and salary both match the department number and salary of any employee who earns a commission.

	LAST_NAME	DEPARTMENT_ID	SALARY
1	Russell	80	14000
2	Partners	80	13500
3	Errazuriz	80	12000
...			
34	Livingston	80	8400

Display the last name, department name, and salary of any employee whose salary and commission match the salary and commission of any employee located in location ID 1700.

	LAST_NAME	DEPARTMENT_NAME	SALARY
1	King	Executive	24000
2	De Haan	Executive	17000
3	Kochhar	Executive	17000
4	Higgins	Accounting	12000
5	Greenberg	Finance	12000
...			
35	Whalen	Administration	4400
36	Gietz	Accounting	8300

Create a query to display the last name, hire date, and salary for all employees who have the same salary and commission as Kochhar.
Note: Do not display Kochhar in the result set.

	LAST_NAME	HIRE_DATE	SALARY
1	De Haan	13-JAN-1993	17000

Create a query to display the employees who earn a salary that is higher than the salary of all the sales managers (`JOB_ID = 'SA_MAN'`). Sort the results on salary from the highest to the lowest.

	LAST_NAME	JOB_ID	SALARY
1	King	AD_PRES	24000
2	De Haan	AD_VP	17000
3	Kochhar	AD_VP	17000

Practice 6 (continued)

Display the employee ID, last name, and department ID of those employees who live in cities whose name begins with *T*.

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	202	Fay	20
2	201	Hartstein	20

Write a query to find all employees who earn more than the average salary in their departments. Display the last name, salary, department ID, and the average salary for the department. Sort by average salary. Use aliases for the columns retrieved by the query as shown in the sample output.

	ENAME	SALARY	DEPTNO	DEPT_AVG
1	Chung	3800	50	3475.555555555555555555555555555555555556
2	Bell	4000	50	3475.555555555555555555555555555555555556
3	Sarchand	4200	50	3475.555555555555555555555555555555555556
4	Bull	4100	50	3475.555555555555555555555555555555555556
5	Vollman	6500	50	3475.555555555555555555555555555555555556
6	Ladwig	3600	50	3475.555555555555555555555555555555555556
7	Rajs	3500	50	3475.555555555555555555555555555555555556
8	Fripp	8200	50	3475.555555555555555555555555555555555556
9	Mourgos	5800	50	3475.555555555555555555555555555555555556

...

34	Hall	9000	80	8955.882352941176470588235294117647058824
35	Hartstein	13000	20	9500
36	Higgins	12000	110	10150
37	King	24000	90	19333.3333333333333333333333333333333333

Find all employees who are not supervisors.

a. First do this using the `NOT EXISTS` operator.

	LAST_NAME
1	Ernst
2	Austin
3	Pataballa
4	Lorentz
5	Faviet
6	Chen

...

88	Baer
89	Gietz

b. Can this be done by using the `NOT IN` operator? How, or why not?

Practice 6 (continued)

Write a query to display the last names of the employees who earn less than the average salary in their departments.

	LAST_NAME
1	Chen
2	Sciarra
3	Urman
4	Popp
5	Khoo
6	Baida
7	Tobias
8	Himuro

...

Write a query to display the last names of the employees who have one or more coworkers in their departments with later hire dates but higher salaries.

	LAST_NAME
1	Pataballa
2	Austin
3	Faviet
4	Sciarra
5	Tobias
6	Bell
7	Sarchand
8	Rajs

...

Practice 6 (continued)

Write a query to display the employee ID, last name, and department name of all employees.
Note: Use a scalar subquery to retrieve the department name in the `SELECT` statement.

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT
1	205	Higgins	Accounting
2	206	Gietz	Accounting
3	200	Whalen	Administration
4	100	King	Executive
5	101	Kochhar	Executive

...

102	140	Patel	Shipping
103	141	Rajs	Shipping
104	142	Davies	Shipping
105	143	Matos	Shipping
106	181	Fleaur	Shipping
107	178	Grant	(null)

Write a query to display the department names of those departments whose total salary cost is above one-eighth (1/8) the total salary cost of the whole company. Use the `WITH` clause to write this query. Name the query `SUMMARY`.

	DEPARTMENT_NAME	DEPT_TOTAL
1	Sales	304500
2	Shipping	156400

Hierarchical Retrieval

Objectives

After completing this lesson, 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

Objectives

In this lesson, 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	105	Austin	IT_PROG	103
7	106	Pataballa	IT_PROG	103
8	107	Lorentz	IT_PROG	103
9	108	Greenberg	FI_MGR	101
10	109	Faviet	FI_ACCOUNT	108
…				
101	200	Whalen	AD_ASST	101
102	201	Hartstein	MK_MAN	100
103	202	Fay	MK_REP	201
104	203	Mavris	HR_REP	101
105	204	Baer	PR_REP	101
106	205	Higgins	AC_MGR	101
107	206	Gietz	AC_ACCOU...	205

ORACLE

Sample Data from the **EMPLOYEES** Table

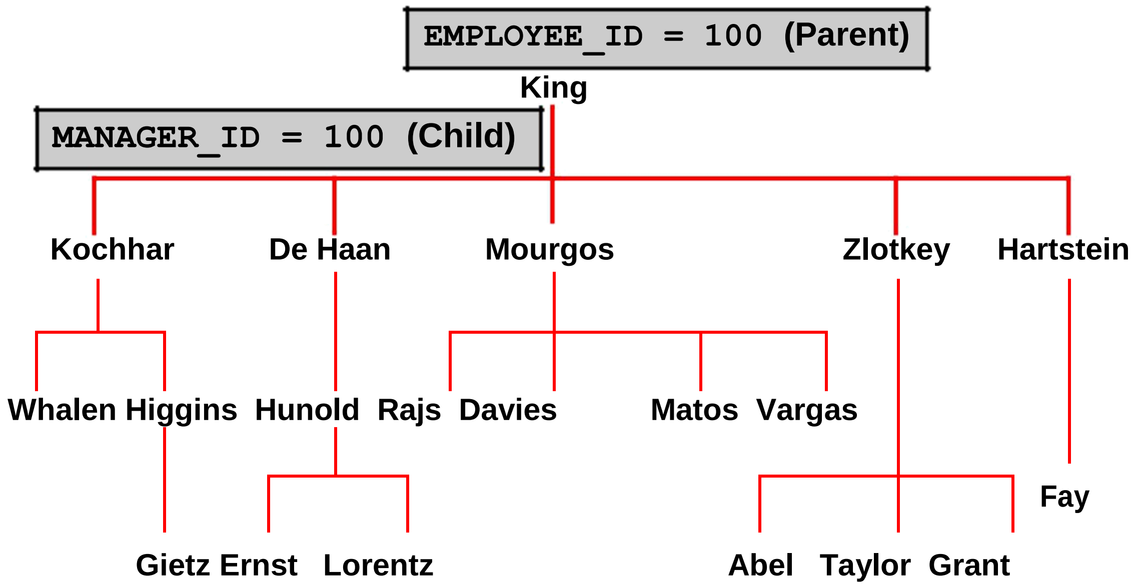
Using hierarchical queries, you can retrieve data based on a natural hierarchical relationship between rows in a table. A relational database does not store records in a hierarchical way. However, where a hierarchical relationship exists between the rows of a single table, a process called *tree walking* 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 eldest 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 employees with the job IDs of `AD_VP`, `ST_MAN`, `SA_MAN`, and `MK_MAN` report directly to the president of the company. You know this because the `MANAGER_ID` column of these records contains the employee ID 100, which belongs to the president (`AD_PRES`).

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



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)] ;
```

WHERE *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` Specifies the columns in which the relationship between parent and child `PRIOR` rows exist. This clause is required for a hierarchical query.

The `SELECT` statement cannot contain a join or query from a view that contains a join.

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'
```


Walking the Tree

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 be used in conjunction with 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. If a WHERE clause is used, the walk is started with all the rows that satisfy the WHERE condition. This no longer reflects a true hierarchy.

Note: The CONNECT BY PRIOR and START WITH clauses are not 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

Walking the Tree (continued)

The direction of the query, whether it is from parent to child or from child to parent, is determined by the CONNECT BY PRIOR column placement. The PRIOR operator refers to the parent 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)

ORACLE

Walking the Tree: From the Bottom Up

The example in the slide displays a list of managers starting with the employee whose employee ID is 101.

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.

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 ;
```

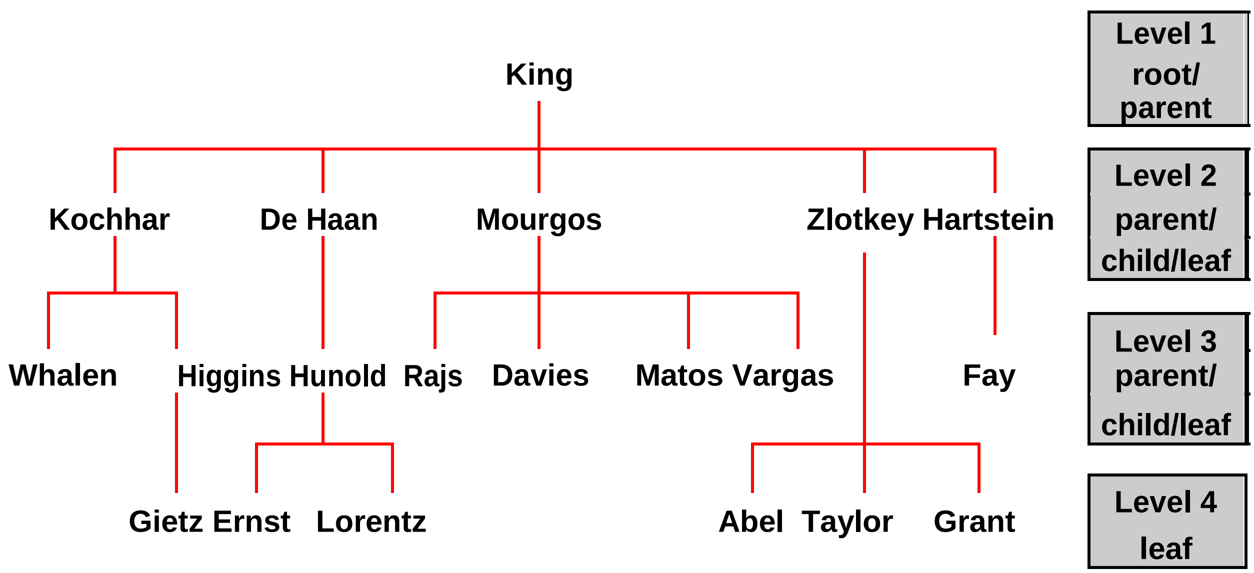
	Walk Top Down
1	King reports to
2	King reports to
3	Kochhar reports to King
4	Greenberg reports to Kochhar
5	Faviet reports to Greenberg
6	Chen reports to Greenberg
7	Sciarra reports to Greenberg
8	Urman reports to Greenberg

...

Walking the Tree: From the Top Down

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.

Ranking Rows with the `LEVEL` Pseudocolumn



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 diagram in the slide shows the nodes of the inverted tree with their `LEVEL` values. For example, employee Higgins is a parent and a child, whereas employee Davies is a child and a leaf.

`LEVEL` Pseudocolumn

Value	Level
1	A root node
2	A child of a root node
3	A child of a child, and so on

In the slide, King is the root or parent (`LEVEL = 1`). Kochhar, De Haan, Mourgos, Zlotkey, Hartstein, Higgins, 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 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
```


Formatting Hierarchical Reports Using LEVEL and LPAD

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.

Formatting Hierarchical Reports Using `LEVEL` and `LPAD` (continued)

	ORG_CHART
1	King
2	__Kochhar
3	____Greenberg
4	______Faviet
5	______Chen
6	______Sciarra
7	______Urman
8	______Popp
9	____Whalen
10	____Mavris

...

100	____Abel
101	____Hutton
102	____Taylor
103	____Livingston
104	____Grant
105	____Johnson
106	__Hartstein
107	____Fay

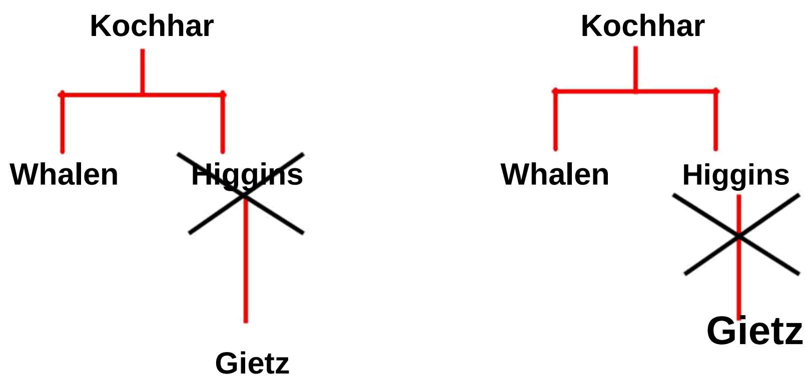
Pruning Branches

Use the `WHERE` clause to eliminate a node.

Use the `CONNECT BY` clause to eliminate a branch.

```
WHERE last_name != 'Higgins'
```

```
CONNECT BY PRIOR
employee_id = manager_id
AND last_name != 'Higgins'
```



Pruning Branches

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 lesson, you should have learned that:

- You can use hierarchical queries to view a hierarchical relationship between rows in a table
- You specify the direction and starting point of the query
- You can eliminate nodes or branches by pruning

ORACLE

Summary

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 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.

Practice 7: Overview

This practice covers the following topics:

- Distinguishing hierarchical queries from nonhierarchical queries
- Walking through a tree
- Producing an indented report by using the `LEVEL` pseudocolumn
- Pruning the tree structure
- Sorting the output

Practice 7: Overview

In this practice, you gain practical experience in producing hierarchical reports.

Note: Question 1 is a paper-based question.

Practice 7

Look at the following output examples. Are they the result of a hierarchical query? Explain why or why not.

Exhibit 1:

	EMPLOYEE_ID	LAST_NAME	MANAGER_ID	SALARY	DEPARTMENT_ID
1	100	King	(null)	24000	90
2	101	Kochhar	100	17000	90
3	102	De Haan	100	17000	90
4	145	Russell	100	14000	80
5	146	Partners	100	13500	80
6	201	Hartstein	100	13000	20
7	205	Higgins	101	12000	110
8	108	Greenberg	101	12000	100
9	147	Errazuriz	100	12000	80
10	168	Ozer	148	11500	80
11	148	Cambrault	100	11000	80
12	174	Abel	149	11000	80
13	114	Raphaely	100	11000	30
14	162	Vishney	147	10500	80
15	149	Zlotkey	100	10500	80

▪ ▪ ▪

Exhibit 2:

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	100	King	90	Executive
2	101	Kochhar	90	Executive
3	102	De Haan	90	Executive
4	103	Hunold	60	IT
5	104	Ernst	60	IT
6	105	Austin	60	IT
7	106	Pataballa	60	IT
8	107	Lorentz	60	IT
9	108	Greenberg	100	Finance
10	109	Faviet	100	Finance

▪ ▪ ▪

Practice 7 (continued)

Exhibit 3:

	RANK	LAST_NAME
1	1	Ernst
2	2	Hunold
3	3	De Haan
4	4	King

Produce a report that shows an organization chart for Mourgos's department. Print the last names, salaries, and department IDs.

	LAST_NAME	SALARY	DEPARTMENT_ID
1	Mourgos	5800	50
2	Rajs	3500	50
3	Davies	3100	50
4	Matos	2600	50
5	Vargas	2500	50
6	Walsh	3100	50
7	Feeney	3000	50
8	OConnell	2600	50
9	Grant	2600	50

Create a report that shows the hierarchy of managers for the employee Lorentz. Display his immediate manager first.

	LAST_NAME
1	Hunold
2	De Haan
3	King

Practice 7 (continued)

Create an indented report that shows the management hierarchy starting from the employee whose `LAST_NAME` is Kochhar. Print the employee's last name, manager ID, and department ID. Give alias names to the columns as shown in the sample output.

	NAME	MGR	DEPTNO
1	Kochhar	100	90
2	__Greenberg	101	100
3	____Faviet	108	100
4	____Chen	108	100
5	____Sciarra	108	100
6	____Urman	108	100
7	____Popp	108	100
8	__Whalen	101	10
9	__Mavris	101	40
10	__Baer	101	70
11	__Higgins	101	110
12	____Gietz	205	110

If you have time, complete the following exercise:

Produce a company organization chart that shows the management hierarchy. Start with the person at the top level, exclude all those with a job ID of `IT_PROG`, and exclude De Haan and those employees who report to De Haan.

	LAST_NAME	EMPLOYEE_ID	MANAGER_ID
1	King	100	(null)
2	Kochhar	101	100
3	Greenberg	108	101
4	Faviet	109	108
5	Chen	110	108
6	Sciarra	111	108
7	Urman	112	108
8	Popp	113	108

...

93	Zlotkey	149	100
94	Abel	174	149
95	Hutton	175	149
96	Taylor	176	149
97	Livingston	177	149
98	Grant	178	149
99	Johnson	179	149
100	Hartstein	201	100
101	Fay	202	201

8

Regular Expression Support

Objectives

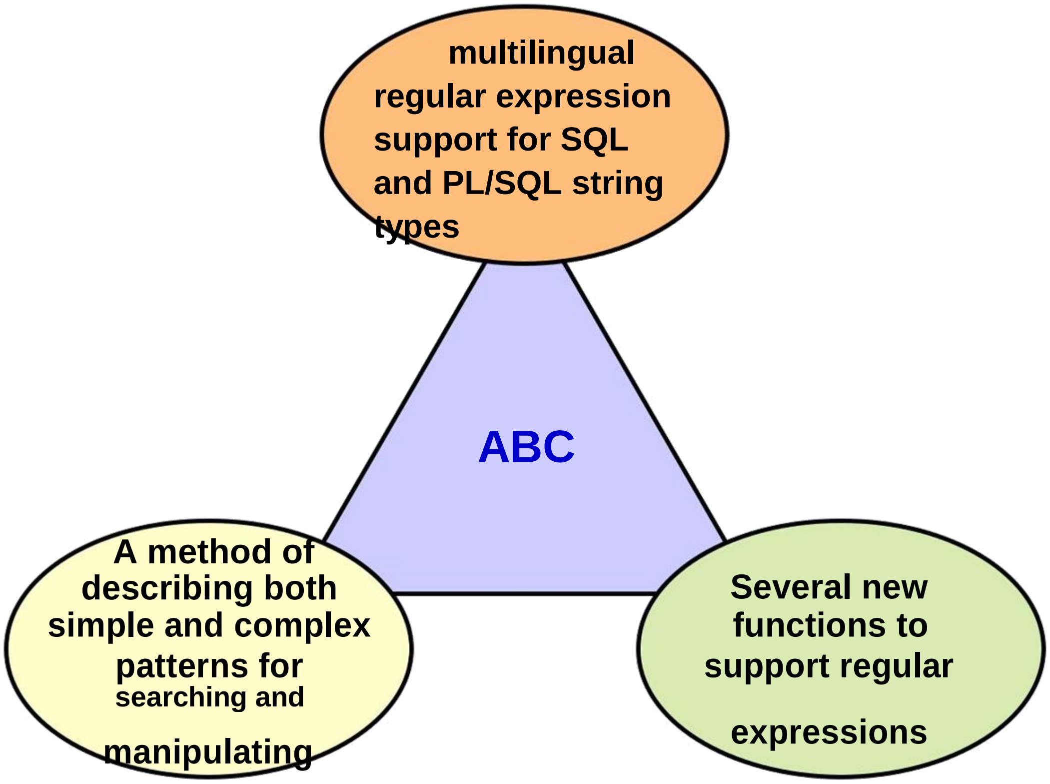
After completing this lesson, you should be able to use regular expression support in SQL to search, match, and replace strings in terms of regular expressions.

ORACLE

Objectives

In this lesson, you learn to use the regular expression support feature that has been introduced in Oracle Database 10*g*.

Regular Expression: Overview



Regular Expression: Overview

Oracle Database 10*g* introduces 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 (for example, find the word "San Francisco" in a specified text) to the complex (for example, extract all URLs from the text) to the more complex (for instance, find 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 be very complex to program.

Meta Characters

Symbol	Description
*	Matches zero or more occurrences
|	Alteration operator for specifying alternative matches
^/$	Matches the start-of-line/end-of-line
[ ]	Bracket expression for a matching list matching any one of the expressions represented in the list
{m}	Matches exactly *m* times
{m,n}	Matches at least *m* times but no more than *n* times
[: :]	Specifies a character class and matches any character in that class
\	Can have 4 different meanings: 1. Stand for itself. 2. Quote the next character. 3. Introduce an operator. 4. Do nothing.
+	Matches one or more occurrences
?	Matches zero or one occurrence
.	Matches any character in the supported character set, except NULL
()	Grouping expression, treated as a single subexpression
[==]	Specifies equivalence classes
\n	Back-reference expression
[..]	Specifies one collation element, such as a multicharacter element

Meta Characters

Meta characters are special characters that have a special meaning, such as a wildcard character, a repeating character, a nonmatching character, or a range of characters. You can use several predefined meta character symbols in the pattern matching.

Using Meta Characters

```
Problem: Find 'abc' within a string:
Solution:             'abc'                              1
Matches:               abc
Does not match:       'def'
```

```
Problem: To find 'a' followed by any character, followed
           by 'c'
Meta Character: any character is defined by '.'
Solution:                'a.c'                          2
Matches:                  abc
Matches:                  adc
Matches:                  alc
Matches:                  a&c
Does not match:           abb
```

```
Problem: To find one or more occurrences of 'a'
Meta Character: Use'+' sign to match one or more of
the previous characters
Solution:               'a+'                            3
Matches:                 a
Matches:                 aa
Does not match:          bbb
```

ORACLE

Using Meta Characters

In the first example, a simple match is performed.

In the second example, the `any` character is defined as a `'.'`. This example searches for the character “a” followed by any character, followed by the character “c”.

The third example searches for one or more occurrences of the letter “a.” The “+” character is used here to indicate a match of one or more of the previous characters.

You can search for nonmatching character lists too. A nonmatching character list allows you to define a set of characters for which a match is invalid. For example, to find anything but the characters “a,” “b,” or “c,” you can define the “^” to indicate a nonmatch.

```
Expression:          [^abc]
  Matches:           abcdef
  Matches:           ghi
  Does not match:    abc
```

To match any letter not between “a” and “i,” you can use:

```
Expression:          [^a-i]
  Matches:           hijk
  Matches:           lmn
  Does not match:    abcdefghi
```

Using Meta Characters (continued)

Meta Character Syntax	Operator Name	Description
.	Any Character – Dot	Match any character.
+	One or More – Plus Quantifier	Match one or more occurrences of the preceding subexpression.
?	Zero or One – Question Mark Quantifier	Match zero or one occurrence of the preceding subexpression.
*****	Zero or More – Star Quantifier	Match zero or more occurrences of the preceding subexpression.
{m} **{m,}** **{m,n}** **[…]**	Interval – Exact Count Matching Character List	Match • exactly *m* occurrences • at least *m* occurrences • at least *m*, but not more than *n* occurrences of the preceding subexpression
[^…]	Non-Matching Character List	Match any character not in list ...
|	Or	'a|b' matches character 'a' or 'b'.
\n **(…)**	Back reference Subexpression or Grouping	Match the nth preceding subexpression, Treat expression ... as a unit.
		where *n* is an integer from 1 to 9.
\	Escape Character	Treat the subsequent meta character in the expression as a literal.
^	Beginning of Line Anchor	Match the subsequent expression when it occurs at the beginning of a line.
$	End of Line Anchor	Match the preceding expression only when it occurs at the end of a line.
[:class:]	POSIX Character Class	Match any character belonging to the specified character *class*.

Regular Expression Functions

Function Name	Description
REGEXP_LIKE	Similar to the LIKE operator, but performs regular expression matching instead of simple pattern matching
REGEXP_REPLACE	Searches for a regular expression pattern and replaces it with a replacement string
REGEXP_INSTR	Searches for a given 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 returns the matched substring

Regular Expression Functions

Oracle Database 10*g* provides a set of SQL functions that you can use to search and manipulate strings using regular expressions. You can use these functions on any data type that holds character data such as CHAR, NCHAR, CLOB, NCLOB, NVARCHAR2, and VARCHAR2. A regular expression must be enclosed or wrapped between single quotation marks. Doing so ensures that the entire expression is interpreted by the SQL function and can improve the readability of your code.

REGEXP_LIKE: This function searches a character column for a pattern. Use this function in the WHERE clause of a query to return rows matching the regular expression you specify.

REGEXP_REPLACE: This function searches for a pattern in a character column and replaces each occurrence of that pattern with the pattern 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 you specify.

REGEXP Function Syntax

```
REGEXP_LIKE   (srcstr, pattern [,match_option])
```

```
REGEXP_INSTR   (srcstr, pattern [, position [, occurrence
               [, return_ option [, match_option]]]])
```

```
REGEXP_SUBSTR (srcstr, pattern [, position
               [, occurrence [, match_option]]])
```

```
REGEXP_REPLACE(srcstr, pattern [,replacestr [, position
               [, occurrence [, match_option]]]])
```


REGEXP Function Syntax

The following table contains descriptions of the terms shown in the syntax in the slide:

`srcstr`	Search value
`pattern`	Regular expression
`occurrence`	Occurrence to search for
`position`	Search starting position
`return_option`	Start or end position of occurrence
`replacestr`	Character string replacing pattern
`match_option`	Option to change default matching; it can include one or more of the following values: "`c`" uses case-sensitive matching (default). "`i`" uses non-case-sensitive matching. "`n`" allows match-any-character operator. "`m`" treats source string as multiple line.

Performing Basic Searches

```
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

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 sentence
$ indicates the end of the sentence
| indicates either/or

Checking the Presence of a Pattern

```
SELECT street_address,
   REGEXP_INSTR(street_address,'[^[:alpha:]]')
FROM   locations
WHERE
   REGEXP_INSTR(street_address,'[^[:alpha:]]')> 1;
```

	STREET_ADDRESS	REGEXP_INSTR(STREET_ADDRESS,'[^[:ALPHA:]]')
1	Magdalen Centre, The Oxford Science Park	9
2	Schwanthalerstr. 7031	16
3	Rua Frei Caneca 1360	4
4	Murtenstrasse 921	14
5	Pieter Breughelstraat 837	7
6	Mariano Escobedo 9991	8

Checking the Presence of a Pattern

In this example, the `REGEXP_INSTR` function is used to search the street address to find the location of the first nonalphabetic character, regardless of whether it is in uppercase or lowercase. The search is performed only on those addresses that do not start with a number. Note that `[:<class>:]` implies a character class and matches any character from within that class, and `[:alpha:]` matches with any alphabetic character. The results are displayed.

In the expression used in the query **`'[^[:alpha:]]'`**:

[starts the expression

^ indicates NOT

[:alpha:] indicates alpha 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:]]+`.

Example of Extracting Substrings

```
SELECT REGEXP_SUBSTR(street_address , ' [^ ]+ ')
"Road" FROM locations;
```

	Road
1	Via
2	Calle
3	(null)
4	(null)
5	Jabberwocky
6	Interiors
7	Zagora
8	Charade
9	Spadina
10	Boxwood

...

ORACLE

Example of Extracting a Substring

In this example, the road names are extracted from the `LOCATIONS` table. To do this, the contents in the `STREET_ADDRESS` column that are before the first space are returned 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
  indicates space

Replacing Patterns

```
SELECT REGEXP_REPLACE( country_name, '(.)',
                       '\1 ') "REGEXP_REPLACE"
FROM countries;
```

	REGEXP_REPLACE
1	A r g e n t i n a
2	A u s t r a l i a
3	B e l g i u m
4	B r a z i l
5	C a n a d a
6	S w i t z e r l a n d
7	C h i n a
8	G e r m a n y
9	D e n m a r k
10	E g y p t

...

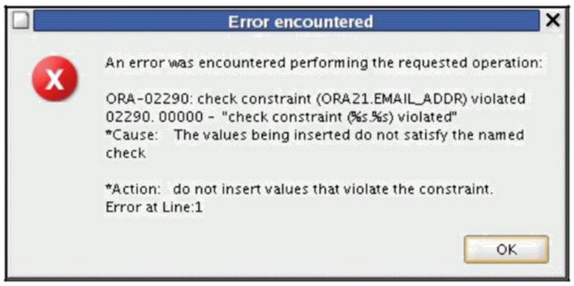
Replacing Patterns

This example examines `COUNTRY_NAME`. The Oracle Database reformats this pattern with a space after each non-null character in the string. The results are shown.

Regular Expressions and Check Constraints

```
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) ;
```



Regular Expressions and Check Constraints

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 e-mail address does not contain the required symbol. The `NOVALIDATE` clause ensures that the existing data is not checked.

Summary

In this lesson, you should have learned how to use regular expression support in SQL and PL/SQL to search, match, and replace strings in terms of regular expressions.

Summary

In this lesson, you have learned to use the regular expression support features that have been introduced in Oracle Database 10*g*.

Practice 8: Overview

This practice covers using regular expressions.

Practice 8: Overview

This practices covers searching and replacing data using regular expressions.

Practice 8

Write a query to search the `EMPLOYEES` table for all employees whose first names start with "Ne" or "Na."

	FIRST_NAME	LAST_NAME
1	Nanette	Cambrault
2	Nancy	Greenberg
3	Neena	Kochhar
4	Nandita	Sarchand

Create a query that removes the spaces in the `STREET _ADDRESS` column of the `LOCATIONS` table in the display.

	REGEXP_REPLACE(STREET_ADDRESS,' ','')
1	1297ViaColadiRie
2	93091CalledellaTesta
3	2017Shinjuku-ku
4	9450Kamiya-cho
5	2014JabberwockyRd
6	2011InteriorsBlvd
7	2007ZagoraSt
8	2004CharadeRd
9	147SpadinaAve
10	6092BoxwoodSt
11	40-5-12Laogianggen
12	1298Vileparle(E)
13	12-98VictoriaStreet
14	198ClementiNorth

...

18	Schwanthalerstr.7031
19	RuaFreiCaneca1360
20	20RuedesCorps-Saints
21	Murtenstrasse921
22	PieterBreughelstraat837
23	MarianoEscobedo9991

Practice 8 (continued)

Create a query that displays “St” replaced by “Street” in the `STREET_ADDRESS` column of the `LOCATIONS` table. Be careful that you do not affect any rows that already have “Street” in them. Display only those rows that are affected.

	REGEXP_REPLACE(STREET_ADDRESS,'ST$','STREET')
1	2007 Zagora Street
2	6092 Boxwood Street
3	12-98 Victoria Street
4	8204 Arthur Street

Appendix A: Practice Solutions

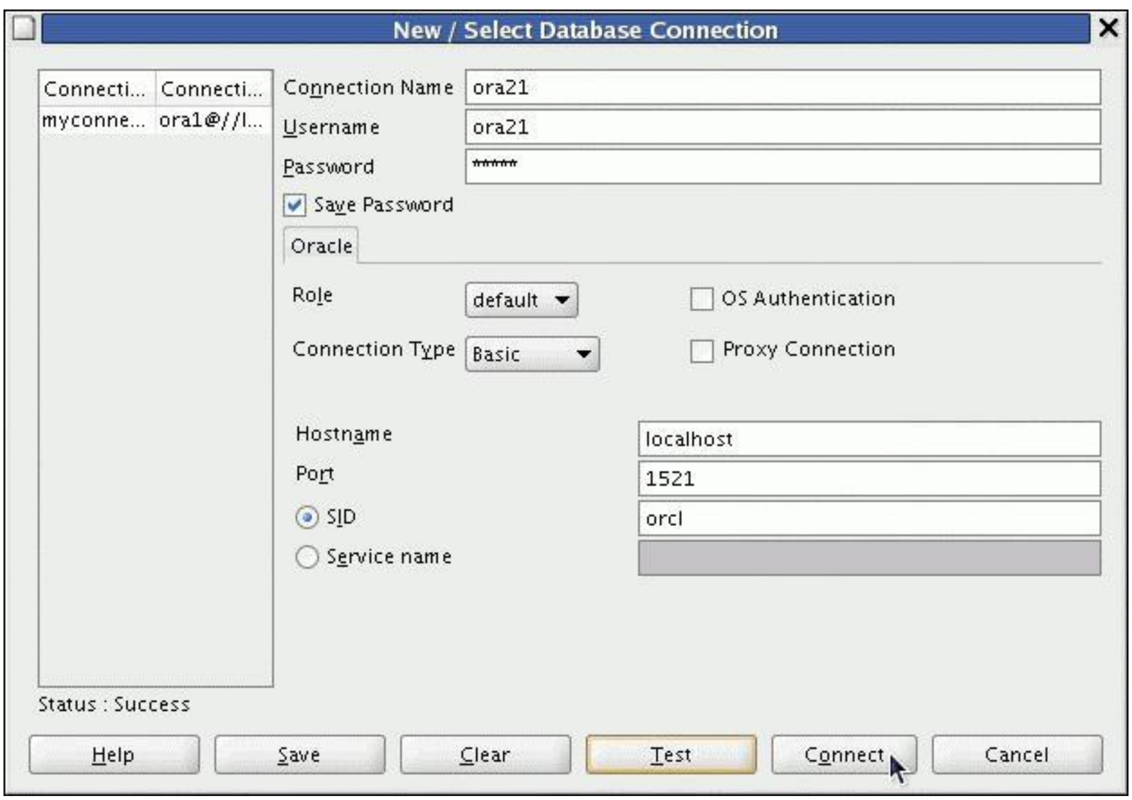
Practice 1: Solutions

To complete question 6 and the subsequent ones, you need to connect to the database using SQL Developer. To do so, double-click the SQL Developer icon on the desktop.

To create a new database connection in the Connections Navigator, right-click Connections. Select New Connection from the menu. The New/Select Database Connection dialog box appears.

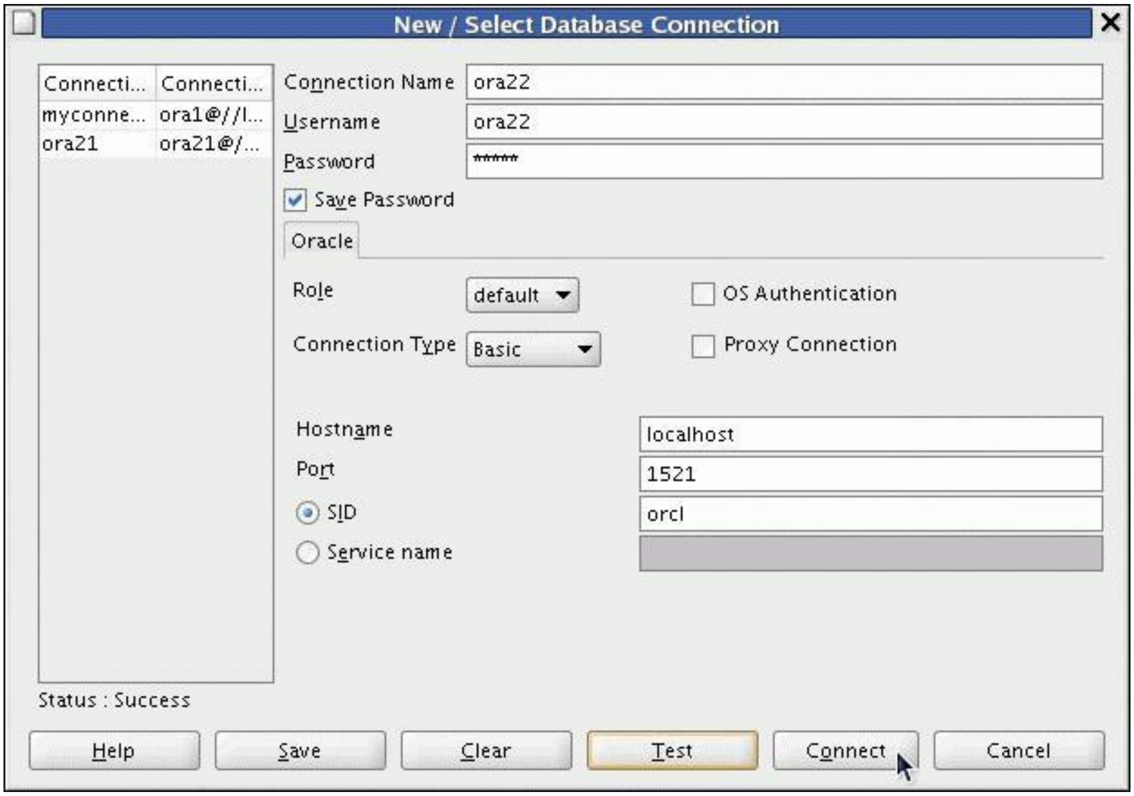
Create a database connection using the following information:

Connection Name: `ora21`
Username: `ora21`
Password: `ora21`
Hostname: localhost
Port: 1521
SID: ORCL
Ensure that you select the Save Password check box.



Create another database connection using the following information:

Connection Name: `ora22`
Username: `ora22`
Password: `ora22`
Hostname: localhost
Port: 1521
SID: ORCL
Ensure that you select the Save Password check box.



Which privilege should a user be given to log on to the Oracle server? Is this a system or an object privilege?

The `CREATE SESSION` system privilege

Which privilege should a user be given to create tables?

The `CREATE TABLE` privilege

If you create a table, who can pass along privileges to other users on your table?

You can, or anyone that you have given those privileges to, by using the `WITH GRANT OPTION`

You are the DBA. You are creating many users who require the same system privileges. What should you use to make your job easier?

Create a role containing the system privileges and grant the role to the users.

Which command would you use to change your password?

The `ALTER USER` statement

Practice 1: Solutions (continued)

6. Connect as user `ora21`. Query all the rows in your `DEPARTMENTS` table.

```
SELECT *
FROM   departments;
```

Add a new row to your `DEPARTMENTS` table. Add Education as department number 500.

```
INSERT INTO departments (department_id, department_name)
VALUES (500, 'Education');
COMMIT;
```

Grant user `ora22` access to your `DEPARTMENTS` table.

```
GRANT SELECT ON departments TO ora22;
```

Connect as user `ora22`. Query user `ora21`’s `DEPARTMENTS` table.

```
SELECT *
FROM ora21.departments;
```

Create a synonym for user `ora21`’s `DEPARTMENTS` table. Query all the rows in user `ora21`’s `DEPARTMENTS` table by using your synonym.

```
CREATE SYNONYM new_dept
FOR ora21.DEPARTMENTS;

SELECT *
FROM new_dept;
```

As user `ora22`, query the `USER_TABLES` data dictionary to see information about the tables that you own.

```
SELECT table_name
FROM   user_tables;
```

Practice 1: Solutions (continued)

As user ora22, query the ALL_TABLES data dictionary view to see information about all the tables that you can access. Exclude the tables that you own.

```
SELECT table_name, owner
FROM   all_tables
WHERE  owner <>'ORA22';
```

13. Connect as user ora21 and revoke the SELECT privilege from user ora22.

```
REVOKE SELECT ON departments
FROM   ora22;
```

Remove the row that you inserted into the DEPARTMENTS table in step 7 and save the changes.

```
DELETE FROM  departments
WHERE department_id = 500
/
COMMIT
/
```

Practice 2: Solutions

Create the DEPT2 table based on the following table instance chart. Place the syntax in a script called lab_02_01.sql, and then execute the statement in the script to create the table. Confirm that the table is created.

Column Name	ID	NAME
Key Type		
Nulls/Unique		
FK Table		
FK Column		
Data type	NUMBER	VARCHAR2
Length	7	25

```
CREATE TABLE dept2
 (id NUMBER(7),
  name VARCHAR2(25));

DESCRIBE dept2
```

Populate the DEPT2 table with data from the DEPARTMENTS table. Include only the columns that you need.

```
INSERT INTO dept2
SELECT  department_id, department_name
FROM    departments;
```

Create the EMP2 table based on the following table instance chart. Place the syntax in a script called lab_02_03.sql, and then execute the statement in the script to create the table. Confirm that the table is created.

```
CREATE TABLE   emp2
  (id           NUMBER(7),
   last_name     VARCHAR2(25),
   first_name    VARCHAR2(25),
   dept_id       NUMBER(7));

DESCRIBE emp2
```

Modify the EMP2 table to allow for longer employee last names. Confirm your modification.

```
ALTER TABLE emp2
MODIFY (last_name   VARCHAR2(50));

DESCRIBE emp2
```

Confirm that both the DEPT2 and EMP2 tables are stored in the data dictionary. (**Hint:** USER_TABLES)

```
SELECT   table_name
FROM     user_tables
WHERE    table_name IN ('DEPT2', 'EMP2');
```

Create the EMPLOYEES2 table based on the structure of the EMPLOYEES table. Include only the EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY, and DEPARTMENT_ID columns. Name the columns in your new table ID, FIRST_NAME, LAST_NAME, SALARY, and DEPT_ID, respectively.

```
CREATE TABLE employees2 AS
SELECT  employee_id id, first_name, last_name, salary,
        department_id dept_id
FROM    employees;
```

Drop the EMP2 table.

```
DROP TABLE emp2;
```

Query the Recycle bin to see whether the table is present.

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

9. Undrop the EMP2 table.

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

Practice 2: Solutions (continued)

Drop the FIRST_NAME column from the EMPLOYEES2 table. Confirm your modification by checking the description of the table.

```
ALTER TABLE employees2
DROP COLUMN first_name;
DESCRIBE employees2
```

In the EMPLOYEES2 table, mark the DEPT_ID column as UNUSED. Confirm your modification by checking the description of the table.

```
ALTER TABLE employees2
SET   UNUSED (dept_id);
DESCRIBE employees2
```

Drop all the UNUSED columns from the EMPLOYEES2 table. Confirm your modification by checking the description of the table.

```
ALTER TABLE employees2
DROP UNUSED COLUMNS;
DESCRIBE employees2
```

Add a table-level PRIMARY KEY constraint to the EMP2 table on the ID column. The constraint should be named at creation. Name the constraint my_emp_id_pk.

```
ALTER TABLE    emp2
ADD CONSTRAINT my_emp_id_pk PRIMARY KEY (id);
```

Create a PRIMARY KEY constraint to the DEPT2 table using the ID column. The constraint should be named at creation. Name the constraint my_dept_id_pk.

```
ALTER TABLE    dept2
ADD CONSTRAINT my_dept_id_pk PRIMARY KEY(id);
```

Add a foreign key reference on the EMP2 table that ensures that the employee is not assigned to a nonexistent department. Name the constraint my_emp_dept_id_fk.

```
ALTER TABLE emp2
ADD CONSTRAINT my_emp_dept_id_fk
FOREIGN KEY (dept_id) REFERENCES dept2(id);
```

Confirm that the constraints were added by querying the USER_CONSTRAINTS view.

Note the types and names of the constraints.

```
SELECT   constraint_name, constraint_type
FROM     user_constraints
WHERE    table_name IN ('EMP2', 'DEPT2');
```

Display the object names and types from the USER_OBJECTS data dictionary view for the EMP2 and DEPT2 tables. Note that the new tables are created and a new index is created.

```
SELECT   object_name, object_type
FROM     user_objects
WHERE    object_name LIKE 'EMP%'
OR       object_name LIKE 'DEPT%';
```

If you have the time, complete the following exercise:

Modify the EMP2 table. Add a COMMISSION column of the NUMBER data type, precision 2, and scale 2. Add a constraint to the COMMISSION column that ensures that a commission value is greater than zero.

```
ALTER TABLE emp2
ADD commission NUMBER(2,2)
CONSTRAINT my_emp_comm_ck CHECK (commission > 0);
```

19. Drop the EMP2 and DEPT2 tables so that they cannot be restored. Verify the Recycle Bin.

```
DROP TABLE emp2 PURGE;
DROP TABLE dept2 PURGE;

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

Create the DEPT_NAMED_INDEX table based on the following table instance chart. Name the index for the PRIMARY KEY column as DEPT_PK_IDX.

Column Name	Deptno	Dname
Primary Key	Yes	
Data Type	Number	VARCHAR2
Length	4	30

Practice 2: Solutions (continued)

```
CREATE TABLE DEPT_NAMED_INDEX
 (deptno NUMBER(4)
 PRIMARY KEY USING INDEX
 (CREATE INDEX dept_pk_idx ON
  DEPT_NAMED_INDEX(deptno)),
  dname VARCHAR2(30));
```

Practice 3: Solutions

Run the `lab_03_01.sql` script in the lab folder to create the `SAL_HISTORY` table.

Display the structure of the `SAL_HISTORY` table.

```
DESC sal_history
```

Run the `lab_03_03.sql` script in the lab folder to create the `MGR_HISTORY` table.

Display the structure of the `MGR_HISTORY` table.

```
DESC mgr_history
```

Run the `lab_03_05.sql` script in the lab folder to create the `SPECIAL_SAL` table.

Display the structure of the `SPECIAL_SAL` table.

```
DESC special_sal
```

a. Write a query to do the following:

Retrieve the employee ID, hire date, salary, and manager ID of those employees whose employee ID is less than 125 from the `EMPLOYEES` table.

If the salary is more than $20,000, insert the employee ID and salary into the `SPECIAL_SAL` table.

Insert the employee ID, hire date, and salary into the `SAL_HISTORY` table.

Insert the employee ID, manager ID, and salary into the `MGR_HISTORY` table.

```
INSERT ALL
WHEN SAL > 20000 THEN
INTO  special_sal VALUES (EMPID, SAL)
ELSE
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 < 125;
```

Display the records from the SPECIAL_SAL table.

```
SELECT * FROM  special_sal;
```

Display the records from the SAL_HISTORY table.

```
SELECT * FROM  sal_history;
```

Display the records from the MGR_HISTORY table.

```
SELECT * FROM mgr_history;
```

a. Run the lab_03_08a.sql script in the lab folder to create the SALES_SOURCE_DATA table.

Run the lab_03_08b.sql script in the lab folder to insert records into the SALES_SOURCE_DATA table.

Display the structure of the SALES_SOURCE_DATA table.

```
DESC sales_source_data
```

Display the records from the SALES_SOURCE_DATA table.

```
SELECT * FROM SALES_SOURCE_DATA;
```

Run the lab_03_08c.sql script in the lab folder to create the SALES_INFO table.

Display the structure of the SALES_INFO table.

```
DESC sales_info
```

Practice 3: Solutions (continued)

Write a query to do the following:

Retrieve the details of the employee ID, week ID, sales on Monday, sales on Tuesday, sales on Wednesday, sales on Thursday, and sales on Friday from the SALES_SOURCE_DATA table.

Build a transformation such that each record retrieved from the SALES_SOURCE_DATA table is converted into multiple records for the SALES_INFO table.

Hint: Use a pivoting INSERT statement.

```
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;
```

h. Display the records from the SALES_INFO table.

```
SELECT * FROM sales_info;
```

You have the data of past employees stored in a flat file called emp.data. You want to store the names and email IDs of all past and present employees in a table. To do this, first create an external table called EMP_DATA using the emp.dat source file in the emp_dir directory. You can use the script in lab_03_09.sql to do this.

```
CREATE TABLE emp_data
  (first_name  VARCHAR2(20)
  ,last_name   VARCHAR2(20)
  , email     VARCHAR2(30)
  )
ORGANIZATION EXTERNAL
(
 TYPE oracle_loader
 DEFAULT DIRECTORY emp_dir
 ACCESS PARAMETERS
 (
  RECORDS DELIMITED BY NEWLINE CHARACTERSET US7ASCII
  NOBADFILE
  NOLOGFILE
  FIELDS
  ( first_name POSITION ( 1:20) CHAR
  , last_name POSITION (22:41) CHAR
  ,  email   POSITION (43:72) CHAR )
 )
 LOCATION ('emp.dat') ) ;
```

Next, run the `lab_03_10.sql` script to create the `EMP_HIST` table.

Increase the size of the email column to 45.

Merge the data in the `EMP_DATA` table that was created in step 9 with the data in the `EMP_HIST` table. Assume that the data in the external `EMP_DATA` table is the most up-to-date. If a row in the `EMP_DATA` table matches the `EMP_HIST` table, update the email column of the `EMP_HIST` table to match the `EMP_DATA` table row. If a row in the `EMP_DATA` table does not match, insert it into the `EMP_HIST` table. Rows are considered matching when the employee's first and last names are identical.

```
MERGE INTO EMP_HIST f USING EMP_DATA h
 ON (f.first_name = h.first_name
 AND f.last_name = h.last_name)
WHEN MATCHED THEN
 UPDATE SET f.email = h.email
WHEN NOT MATCHED THEN
 INSERT (f.first_name
      f.last_name
      f.email)
 VALUES (h.first_name
      h.last_name
      h.email);
```

Retrieve the rows from EMP_HIST after the merge.

```
SELECT * FROM emp_hist;
```

Create the EMP3 table using the lab_03_11.sql script. In the EMP3 table, change the department for Kochhar to 60 and commit your change. Next, change the department for Kochhar to 50 and commit your change. Track the changes to Kochhar using the Row Versions feature.

```
UPDATE emp3 SET department_id = 60
WHERE last_name = 'Kochchar';
COMMIT;
UPDATE emp3 SET department_id = 50
WHERE last_name = 'Kochchar';
COMMIT;
```

```
SELECT VERSIONS_STARTTIME "START_DATE",
   VERSIONS_ENDTIME "END_DATE",  DEPARTMENT_ID
FROM EMP3
   VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE LAST_NAME ='Kochhar';
```

Practice 4: Solutions

Write a query to display the following for those employees whose manager ID is less than 120:

Manager ID

Job ID and total salary for every job ID for employees who report to the same manager

Total salary of those managers

Total salary of those managers, irrespective of the job IDs

```
SELECT manager_id,job_id,sum(salary)
FROM   employees
WHERE manager_id < 120
GROUP BY ROLLUP(manager_id,job_id);
```

Observe the output from question 1. Write a query using the GROUPING function to determine whether the NULL values in the columns corresponding to the GROUP BY expressions are caused by the ROLLUP operation.

```
SELECT manager_id MGR ,job_id JOB,
 sum(salary),GROUPING(manager_id),GROUPING(job_id)
 FROM   employees
 WHERE manager_id < 120
 GROUP BY ROLLUP(manager_id,job_id);
```

Write a query to display the following for those employees whose manager ID is less than 120:

Manager ID

Job and total salary for every job for employees who report to the same manager

Total salary of those managers

Cross-tabulation values to display the total salary for every job, irrespective of the manager

Total salary irrespective of the job titles

```
SELECT manager_id, job_id, sum(salary)
FROM employees
WHERE manager_id < 120
GROUP BY CUBE(manager_id, job_id);
```

Observe the output from question 3. Write a query using the `GROUPING` function to determine whether the `NULL` values in the columns corresponding to the `GROUP BY` expressions are caused by the `CUBE` operation.

```
SELECT manager_id MGR ,job_id JOB,
sum(salary),GROUPING(manager_id),GROUPING(job_id)
FROM employees
WHERE manager_id < 120
GROUP BY CUBE(manager_id,job_id);
```

Using `GROUPING SETS`, write a query to display the following groupings:

`department_id, manager_id, job_id`
`department_id, job_id`
`manager_id, job_id`

The query should calculate the sum of the salaries for each of these groups.

```
SELECT department_id, manager_id, job_id, SUM(salary)
FROM employees
GROUP BY
GROUPING SETS ((department_id, manager_id, job_id),
(department_id, job_id),(manager_id,job_id));
```

Practice 5: Solutions

Alter the session to set the `NLS_DATE_FORMAT` to `DD-MON-YYYY HH24:MI:SS`.

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

a. Write queries to display the time zone offsets (`TZ_OFFSET`) for the following time zones:

US/Pacific-New

```
SELECT TZ_OFFSET ('US/Pacific-New') from dual;
```

Singapore

```
SELECT TZ_OFFSET ('Singapore') from dual;
```

Egypt

```
SELECT TZ_OFFSET ('Egypt') from dual;
```

Alter the session to set the `TIME_ZONE` parameter value to the time zone offset of US/Pacific-New.

```
ALTER SESSION SET TIME_ZONE = '-7:00';
```

Display the `CURRENT_DATE`, `CURRENT_TIMESTAMP`, and `LOCALTIMESTAMP` for this session.

Note: The output may be different based on the date when the command is executed.

```
SELECT CURRENT_DATE, CURRENT_TIMESTAMP,
   LOCALTIMESTAMP FROM DUAL;
```

Alter the session to set the `TIME_ZONE` parameter value to the time zone offset of Singapore.

```
ALTER SESSION SET TIME_ZONE = '+8:00';
```

Display the `CURRENT_DATE`, `CURRENT_TIMESTAMP`, `LOCALTIMESTAMP` for this session.

Note: The output may be different based on the date when the command is executed.

```
SELECT CURRENT_DATE, CURRENT_TIMESTAMP,
LOCALTIMESTAMP FROM DUAL;
```

Note: Observe in the preceding practice that CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP are all sensitive to the session time zone.

Write a query to display DBTIMEZONE and SESSIONTIMEZONE.

```
SELECT DBTIMEZONE,SESSIONTIMEZONE
FROM DUAL;
```

Write a query to extract YEAR from the HIRE_DATE column of the EMPLOYEES table for those employees who work in department 80.

```
SELECT last_name, EXTRACT (YEAR FROM HIRE_DATE)
FROM employees
WHERE department_id = 80;
```

Alter the session to set NLS_DATE_FORMAT to DD-MON-YYYY.

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY';
```

Examine and run the lab_05_06.sql script to create the SAMPLE_DATES table and populate it.

Select from the table and view the data.

```
SELECT * FROM sample_dates;
```

Change the data type of the DATE_COL column to TIMESTAMP. Select from the table to view the data.

```
ALTER TABLE sample_dates MODIFY date_col
TIMESTAMP; SELECT * FROM sample_dates;
```

Try to change the data type of the DATE_COL column to TIMESTAMP WITH TIME ZONE. What happens?

```
ALTER TABLE sample_dates MODIFY
date_col TIMESTAMP WITH TIME ZONE;
```

You are unable to change the data type of the DATE_COL column because the Oracle server does not permit you to convert from TIMESTAMP to TIMESTAMP WITH TIMEZONE by using the ALTER statement.

Practice 5: Solutions (continued)

Create a query to retrieve last names from the `EMPLOYEES` table and calculate the review status. If the year hired is 1998, display `Needs Review` for the review status; otherwise, display `not this year!` Name the review status column `Review`. Sort the results by the `HIRE_DATE` column.
Hint: Use a `CASE` expression with the `EXTRACT` function to calculate the review status.

```
SELECT e.last_name
    (CASE extract(year from e.hire_date)
      WHEN 1998 THEN 'Needs Review'
      ELSE 'not this year!'
     END )           AS "Review "
FROM   employees e
ORDER BY e.hire_date;
```

Create a query to print the last names and the number of years of service for each employee. If the employee has been employed for five or more years, print `5 years of service`. If the employee has been employed for 10 or more years, print `10 years of service`. If the employee has been employed for 15 or more years, print `15 years of service`. If none of these conditions match, print `maybe next year!` Sort the results by the `HIRE_DATE` column. Use the `EMPLOYEES` table.
Hint: Use `CASE` expressions and `TO_YMINTERVAL`.

```
SELECT e.last_name, hire_date, sysdate,
       (CASE
        WHEN  (sysdate -TO_YMINTERVAL('15-0'))>=
            hire_date THEN    '15 years of service'
        WHEN (sysdate -TO_YMINTERVAL('10-
0'))>= hire_date
          THEN '10 years of service'
        WHEN (sysdate - TO_YMINTERVAL('5-
0'))>= hire_date
           THEN '5 years of service'
             ELSE 'maybe next year!'
          END) AS "Awards"
FROM   employees e;
```

Practice 6: Solutions

Write a query to display the last name, department number, and salary of any employee whose department number and salary both match the department number and salary of any employee who earns a commission.

```
SELECT last_name, department_id, salary
FROM   employees
WHERE  (salary, department_id) IN
       (SELECT  salary, department_id
        FROM    employees
            WHERE    commission_pct IS NOT NULL);
```

Display the last name, department name, and salary of any employee whose salary and commission match the salary and commission of any employee located in location ID 1700.

```
SELECT e.last_name, d.department_name, e.salary
FROM   employees e, departments d
WHERE  e.department_id = d.department_id
AND   (salary, NVL(commission_pct,0)) IN
            (SELECT salary, NVL(commission_pct,0)
             FROM   employees e, departments  d
             WHERE  e.department_id = d.department_id
             AND    d.location_id = 1700);
```

Create a query to display the last name, hire date, and salary for all employees who have the same salary and commission as Kochhar.
Note: Do not display Kochhar in the result set.

```
SELECT last_name, hire_date,
salary FROM employees
WHERE  (salary, NVL(commission_pct,0)) IN
              (SELECT salary, NVL(commission_pct,0)
               FROM      employees
               WHERE  last_name = 'Kochhar')
AND last_name != 'Kochhar';
```

Create a query to display the employees who earn a salary that is higher than the salary of all the sales managers (`JOB_ID = 'SA_MAN'`). Sort the results on salary from the highest to the lowest.

```
SELECT last_name, job_id, salary
FROM   employees
WHERE  salary > ALL
             (SELECT salary
              FROM   employees
WHERE  job_id = 'SA_MAN')
ORDER BY salary DESC;
```

Display the details of the employee ID, last name, and department ID of those employees who live in cities whose name begins with *T*.

```
SELECT employee_id, last_name, department_id
FROM employees
WHERE  department_id IN (SELECT department_id
                         FROM departments
                         WHERE location_id IN
                              (SELECT  location_id
                               FROM locations
                               WHERE city LIKE 'T%'));
```

Write a query to find all the employees who earn more than the average salary in their departments. Display the last name, salary, department ID, and the average salary for the department. Sort by average salary. Use aliases for the columns retrieved by the query as shown in the sample output.

```
SELECT e.last_name ename, e.salary salary,
       e.department_id deptno, AVG(a.salary) dept_avg
FROM   employees e, employees a
WHERE  e.department_id = a.department_id
AND    e.salary > (SELECT AVG(salary)
             FROM   employees
             WHERE department_id = e.department_id )
GROUP BY e.last_name, e.salary, e.department_id ORDER
BY AVG(a.salary);
```

Find all employees who are not supervisors.

a. First, do this by using the `NOT EXISTS` operator.

```
SELECT outer.last_name
FROM    employees outer
WHERE  NOT EXISTS (SELECT 'X'
                   FROM employees inner
                    WHERE inner.manager_id =
                          outer.employee_id);
```

b. Can this be done by using the `NOT IN` operator? How, or why not?

```
SELECT outer.last_name
FROM   employees outer
WHERE  outer.employee_id
NOT IN (SELECT inner.manager_id
        FROM   employees inner);
```

This alternative solution is not a good one. The subquery picks up a `NULL` value, so the entire query returns no rows. The reason is that all conditions that compare a `NULL` value result in `NULL`. Whenever `NULL` values are likely to be part of the value set, *do not* use `NOT IN` as a substitute for `NOT EXISTS`.

Write a query to display the last names of the employees who earn less than the average salary in their departments.

```
SELECT last_name
FROM    employees outer
WHERE outer.salary < (SELECT AVG(inner.salary)
                      FROM employees inner
                      WHERE inner.department_id
                            = outer.department_id);
```

Practice 6: Solutions (continued)

Write a query to display the last names of the employees who have one or more coworkers in their departments with later hire dates but higher salaries.

```
SELECT  last_name
FROM    employees outer
WHERE EXISTS (SELECT 'X'
              FROM employees inner
              WHERE inner.department_id =
                    outer.department_id
              AND inner.hire_date > outer.hire_date
              AND inner.salary > outer.salary);
```

Write a query to display the employee ID, last name, and department name of all employees.
Note: Use a scalar subquery to retrieve the department name in the `SELECT` statement.

```
 SELECT employee_id, last_name,
       (SELECT department_name
        FROM departments d
        WHERE   e.department_id =
           d.department_id ) department
FROM employees e
ORDER BY department;
```

Write a query to display the department names of those departments whose total salary cost is above one-eighth (1/8) the total salary cost of the whole company. Use the `WITH` clause to write this query. Name the query `SUMMARY`.

```
WITH
summary AS (
  SELECT d.department_name, SUM(e.salary) AS
  dept_total FROM employees e, departments d
  WHERE e.department_id = d.department_id
  GROUP BY d.department_name)
SELECT department_name, dept_total
FROM summary
WHERE dept_total > ( SELECT SUM(dept_total) * 1/8
                    FROM summary )
ORDER BY dept_total DESC;
```

Practice 7: Solutions

Look at the following output examples. Are these outputs the result of a hierarchical query? Explain why or why not.

Exhibit 1: This is not a hierarchical query; the report simply has a descending sort on SALARY.

Exhibit 2: This is not a hierarchical query; there are two tables involved.

Exhibit 3: Yes, this is most definitely a hierarchical query because it displays the tree structure representing the management reporting line from the EMPLOYEES table.

Produce a report that shows an organization chart for Mourgos's department. Print the last names, salaries, and department IDs.

```
SELECT last_name, salary, department_id
FROM employees
START WITH last_name = 'Mourgos'
CONNECT BY PRIOR employee_id = manager_id;
```

Create a report that shows the hierarchy of managers for employee Lorentz. Display his immediate manager first.

```
SELECT last_name
FROM employees
WHERE last_name != 'Lorentz'
START WITH last_name = 'Lorentz'
CONNECT BY PRIOR manager_id = employee_id;
```

Create an indented report that shows the management hierarchy starting from the employee whose `LAST_NAME` is Kochhar. Print the employee's last name, manager ID, and department ID. Give alias names to the columns as shown in the sample output.

```
COLUMN name FORMAT A20
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-
2,'_')
       name,manager_id mgr, department_id
deptno FROM employees
START WITH last_name = 'Kochhar'
CONNECT BY PRIOR employee_id = manager_id
/
COLUMN name CLEAR
```

Practice 7: Solutions (continued)

If you have time, complete the following exercises:

Produce a company organization chart that shows the management hierarchy. Start with the person at the top level, exclude all people with a job ID of `IT_PROG`, and exclude De Haan and those employees who report to De Haan.

```
SELECT last_name,employee_id, manager_id
FROM   employees
WHERE  job_id != 'IT_PROG'
START WITH manager_id IS NULL
CONNECT BY PRIOR employee_id = manager_id
AND last_name != 'De Haan';
```

Practice 8: Solutions

1. Write a query to search the EMPLOYEES table for all employees whose first names start with “Ne” or “Na.”

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^N(e|a).');
```

Create a query that removes the spaces in the STREET_ADDRESS column of the LOCATIONS table in the display.

```
SELECT regexp_replace (street_address, ' ','')
FROM locations;
```

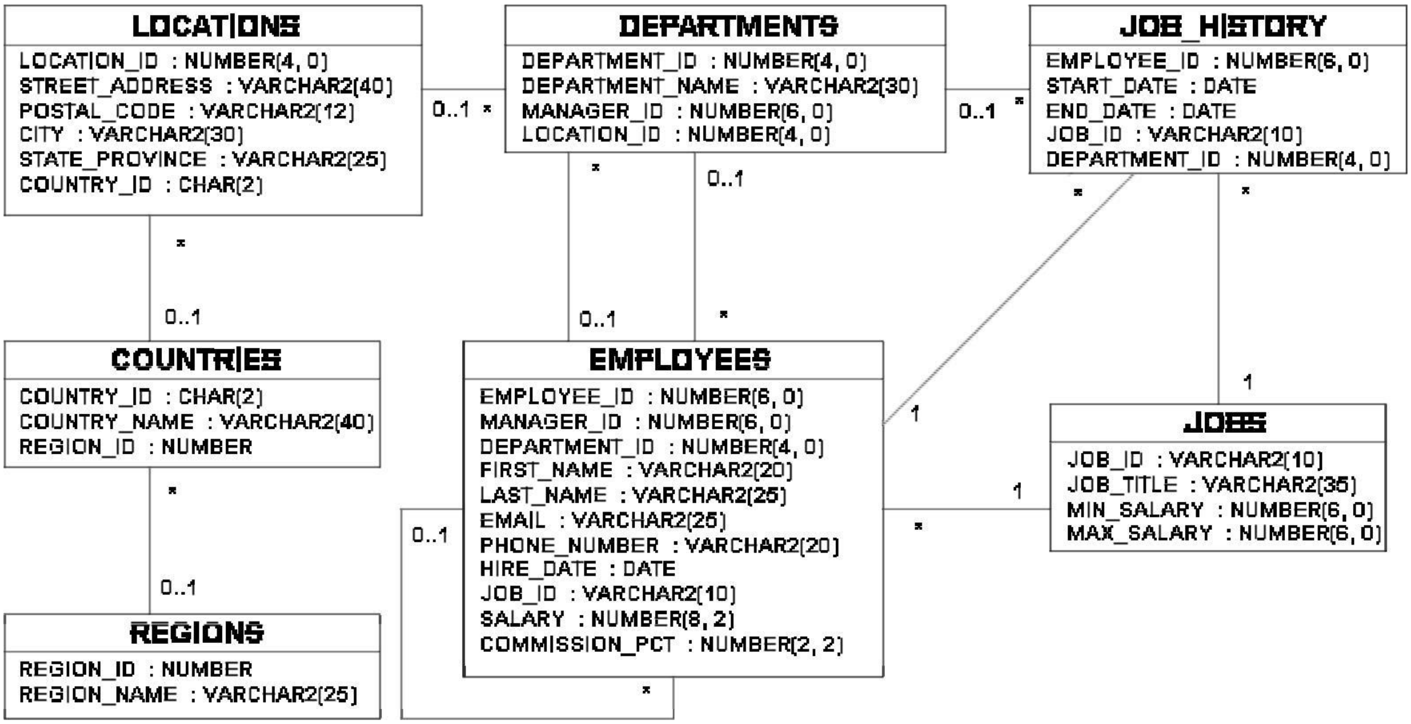
Create a query that displays “St” replaced by “Street” in the STREET_ADDRESS column of the LOCATIONS table. Be careful that you do not affect any rows that already have “Street” in them. Display only those rows, which are affected.

```
SELECT REGEXP_REPLACE (street_address,
'St$', 'Street') FROM locations
WHERE REGEXP_LIKE (street_address, 'St');
```

Appendix B:

Table Descriptions and Data

ENTITY RELATIONSHIP DIAGRAM



Tables in the Schema

```
SELECT * FROM tab;
```

	TNAME	TABTYPE	CLUSTERID
1	REGIONS	TABLE	(null)
2	COUNTRIES	TABLE	(null)
3	LOCATIONS	TABLE	(null)
4	DEPARTMENTS	TABLE	(null)
5	JOBS	TABLE	(null)
6	EMPLOYEES	TABLE	(null)
7	JOB_HISTORY	TABLE	(null)
8	EMP_DETAILS_VIEW	VIEW	(null)

`REGIONS` Table

```
DESCRIBE regions
```

```
Name                           Null     Type
------------------------------ -------- -------------
REGION_ID                      NOT NULL NUMBER
REGION_NAME                             VARCHAR2(25)

2 rows selected
```

```
SELECT * FROM regions;
```

	REGION_ID	REGION_NAME
1	1	Europe
2	2	Americas
3	3	Asia
4	4	Middle East and Africa

COUNTRIES Table

```
DESCRIBE countries
```

```
Name                           Null     Type
------------------------------ -------- -------------
COUNTRY_ID                     NOT NULL CHAR(2)
COUNTRY_NAME                            VARCHAR2(40)
REGION_ID                               NUMBER

3 rows selected
```

```
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	MX	Mexico	2
19	NG	Nigeria	4
20	NL	Netherlands	1
21	SG	Singapore	3
22	UK	United Kingdom	1
23	US	United States of America	2
24	ZM	Zambia	4
25	ZW	Zimbabwe	4

LOCATIONS Table

```
DESCRIBE locations;
```

```
Name                           Null     Type
------------------------------ -------- -------------
LOCATION_ID                    NOT NULL NUMBER(4)
STREET_ADDRESS                          VARCHAR2(40)
POSTAL_CODE                             VARCHAR2(12)
CITY                           NOT NULL VARCHAR2(30)
STATE_PROVINCE                          VARCHAR2(25)
COUNTRY_ID                              CHAR(2)

6 rows selected
```

```
SELECT * FROM locations;
```

	LOC...	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 ...	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

DEPARTMENTS Table

```
DESCRIBE departments
```

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

4 rows selected
```

```
SELECT * FROM departments;
```

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	30	Purchasing	114	1700
4	40	Human Resources	203	2400
5	50	Shipping	121	1500
6	60	IT	103	1400
7	70	Public Relations	204	2700
8	80	Sales	145	2500
9	90	Executive	100	1700
10	100	Finance	108	1700
11	110	Accounting	205	1700
12	120	Treasury	(null)	1700
13	130	Corporate Tax	(null)	1700
14	140	Control And Credit	(null)	1700
15	150	Shareholder Services	(null)	1700
16	160	Benefits	(null)	1700
17	170	Manufacturing	(null)	1700
18	180	Construction	(null)	1700
19	190	Contracting	(null)	1700
20	200	Operations	(null)	1700
21	210	IT Support	(null)	1700
22	220	NOC	(null)	1700
23	230	IT Helpdesk	(null)	1700
24	240	Government Sales	(null)	1700
25	250	Retail Sales	(null)	1700
26	260	Recruiting	(null)	1700
27	270	Payroll	(null)	1700

JOBS Table

```
DESCRIBE jobs
```

```
Name                           Null     Type
------------------------------ -------- --------------
JOB_ID                         NOT NULL VARCHAR2(10)
JOB_TITLE                      NOT NULL VARCHAR2(35)
MIN_SALARY                              NUMBER(6)
MAX_SALARY                              NUMBER(6)

4 rows selected
```

```
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 Represen...	4000	9000
19	PR_REP	Public Relations Representat...	4500	10500

EMPLOYEES Table

```
DESCRIBE employees
```

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

11 rows selected
```

The headings for the `EMPLOYEE_ID`, `FIRST_NAME`, `LAST_NAME`, `SALARY`, `COMMISSION_PCT`, `MANAGER_ID`, and `DEPARTMENT_ID` columns are set as `ID`, `FNAME`, `LNAME`, `SAL`, `COMM`, `MGRID`, and `DEPTID` respectively in the following screenshots to fit the table values across the page.

```
SELECT * FROM employees;
```

	ID	FNAME	LNAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SAL	COMM	MGRID	DEPTID
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	100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	(null)	(null)	90
11	101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000	(null)	100	90
12	102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000	(null)	100	90
13	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000	(null)	102	60
14	104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000	(null)	103	60
15	105	David	Austin	DAUSTIN	590.423.4569	25-JUN-97	IT_PROG	4800	(null)	103	60
16	106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-98	IT_PROG	4800	(null)	103	60
17	107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200	(null)	103	60
18	108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-94	FI_MGR	12000	(null)	101	100
19	109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG-94	FI_ACCOUNT	9000	(null)	108	100
20	110	John	Chen	JCHEN	515.124.4269	28-SEP-97	FI_ACCOUNT	8200	(null)	108	100

EMPLOYEES Table (continued)

	ID	FNAME	LNAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SAL	COMM	MGRID	DEPTID
21	111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-97	FI_ACCOUNT	7700	(null)	108	100
22	112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-98	FI_ACCOUNT	7800	(null)	108	100
23	113	Luis	Popp	LPOPP	515.124.4567	07-DEC-99	FI_ACCOUNT	6900	(null)	108	100
24	114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-94	PU_MAN	11000	(null)	100	30
25	115	Alexander	Khoo	AKHOO	515.127.4562	18-MAY-95	PU_CLERK	3100	(null)	114	30
26	116	Shelli	Baida	SBAIDA	515.127.4563	24-DEC-97	PU_CLERK	2900	(null)	114	30
27	117	Sigal	Tobias	STOBIAS	515.127.4564	24-JUL-97	PU_CLERK	2800	(null)	114	30
28	118	Guy	Himuro	GHIMURO	515.127.4565	15-NOV-98	PU_CLERK	2600	(null)	114	30
29	119	Karen	Colmenares	KCOLMENA	515.127.4566	10-AUG-99	PU_CLERK	2500	(null)	114	30
30	120	Matthew	Weiss	MWEISS	650.123.1234	18-JUL-96	ST_MAN	8000	(null)	100	50
31	121	Adam	Fripp	AFRIPP	650.123.2234	10-APR-97	ST_MAN	8200	(null)	100	50
32	122	Payam	Kaufling	PKAUFLIN	650.123.3234	01-MAY-95	ST_MAN	7900	(null)	100	50
33	123	Shanta	Vollman	SVOLLMAN	650.123.4234	10-OCT-97	ST_MAN	6500	(null)	100	50
34	124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800	(null)	100	50
35	125	Julia	Nayer	JNAYER	650.124.1214	16-JUL-97	ST_CLERK	3200	(null)	120	50
36	126	Irene	Mikkilineni	IMIKKILI	650.124.1224	28-SEP-98	ST_CLERK	2700	(null)	120	50
37	127	James	Landry	JLANDRY	650.124.1334	14-JAN-99	ST_CLERK	2400	(null)	120	50
38	128	Steven	Markle	SMARKLE	650.124.1434	08-MAR-00	ST_CLERK	2200	(null)	120	50
39	129	Laura	Bissot	LBISSOT	650.124.5234	20-AUG-97	ST_CLERK	3300	(null)	121	50
40	130	Mozhe	Atkinson	MATKINSO	650.124.6234	30-OCT-97	ST_CLERK	2800	(null)	121	50
41	131	James	Marlow	JAMRLOW	650.124.7234	16-FEB-97	ST_CLERK	2500	(null)	121	50
42	132	TJ	Olson	TJOLSON	650.124.8234	10-APR-99	ST_CLERK	2100	(null)	121	50
43	133	Jason	Mallin	JMALLIN	650.127.1934	14-JUN-96	ST_CLERK	3300	(null)	122	50
44	134	Michael	Rogers	MROGERS	650.127.1834	26-AUG-98	ST_CLERK	2900	(null)	122	50
45	135	Ki	Gee	KGEE	650.127.1734	12-DEC-99	ST_CLERK	2400	(null)	122	50
46	136	Hazel	Philtanker	HPHILTAN	650.127.1634	06-FEB-00	ST_CLERK	2200	(null)	122	50
47	137	Renske	Ladwig	RLADWIG	650.121.1234	14-JUL-95	ST_CLERK	3600	(null)	123	50
48	138	Stephen	Stiles	SSTILES	650.121.2034	26-OCT-97	ST_CLERK	3200	(null)	123	50
49	139	John	Seo	JSEO	650.121.2019	12-FEB-98	ST_CLERK	2700	(null)	123	50
50	140	Joshua	Patel	JPATEL	650.121.1834	06-APR-98	ST_CLERK	2500	(null)	123	50
51	141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500	(null)	124	50
52	142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	3100	(null)	124	50
53	143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600	(null)	124	50
54	144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-98	ST_CLERK	2500	(null)	124	50
55	145	John	Russell	JRUSSEL	011.44.1344.4292...	01-OCT-96	SA_MAN	14000	0.4	100	80
56	146	Karen	Partners	KPARTNER	011.44.1344.4672...	05-JAN-97	SA_MAN	13500	0.3	100	80
57	147	Alberto	Errazuriz	AERRAZUR	011.44.1344.4292...	10-MAR-97	SA_MAN	12000	0.3	100	80
58	148	Gerald	Cambrault	GCAMBRAU	011.44.1344.6192...	15-OCT-99	SA_MAN	11000	0.3	100	80
59	149	Eleni	Zlotkey	EZLOTKEY	011.44.1344.4290...	29-JAN-00	SA_MAN	10500	0.2	100	80
60	150	Peter	Tucker	PTUCKER	011.44.1344.1292...	30-JAN-97	SA_REP	10000	0.3	145	80
61	151	David	Bernstein	DBERNSTE	011.44.1344.3452...	24-MAR-97	SA_REP	9500	0.25	145	80
62	152	Peter	Hall	PHALL	011.44.1344.4789...	20-AUG-97	SA_REP	9000	0.25	145	80
63	153	Christopher	Olsen	COLSEN	011.44.1344.4987...	30-MAR-98	SA_REP	8000	0.2	145	80
64	154	Nanette	Cambrault	NCAMBRAU	011.44.1344.9876...	09-DEC-98	SA_REP	7500	0.2	145	80
65	155	Oliver	Tuvault	OTUVAULT	011.44.1344.4865...	23-NOV-99	SA_REP	7000	0.15	145	80

EMPLOYEES Table (continued)

	ID	FNAME	LNAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SAL	COMM	MGRID	DEPTID
66	156	Janette	King	JKING	011.44.1345.4292...	30-JAN-96	SA_REP	10000	0.35	146	80
67	157	Patrick	Sully	PSULLY	011.44.1345.9292...	04-MAR-96	SA_REP	9500	0.35	146	80
68	158	Allan	McEwen	AMCEWEN	011.44.1345.8292...	01-AUG-96	SA_REP	9000	0.35	146	80
69	159	Lindsey	Smith	LSMITH	011.44.1345.7292...	10-MAR-97	SA_REP	8000	0.3	146	80
70	160	Louise	Doran	LDORAN	011.44.1345.6292...	15-DEC-97	SA_REP	7500	0.3	146	80
71	161	Sarath	Sewall	SSEWALL	011.44.1345.5292...	03-NOV-98	SA_REP	7000	0.25	146	80
72	162	Clara	Vishney	CVISHNEY	011.44.1346.1292...	11-NOV-97	SA_REP	10500	0.25	147	80
73	163	Danielle	Greene	DGREENE	011.44.1346.2292...	19-MAR-99	SA_REP	9500	0.15	147	80
74	164	Mattea	Marvins	MMARVINS	011.44.1346.3292...	24-JAN-00	SA_REP	7200	0.1	147	80
75	165	David	Lee	DLEE	011.44.1346.5292...	23-FEB-00	SA_REP	6800	0.1	147	80
76	166	Sundar	Ande	SANDE	011.44.1346.6292...	24-MAR-00	SA_REP	6400	0.1	147	80
77	167	Amit	Banda	ABANDA	011.44.1346.7292...	21-APR-00	SA_REP	6200	0.1	147	80
78	168	Lisa	Ozer	LOZER	011.44.1343.9292...	11-MAR-97	SA_REP	11500	0.25	148	80
79	169	Harrison	Bloom	HBLOOM	011.44.1343.8292...	23-MAR-98	SA_REP	10000	0.2	148	80
80	170	Tayler	Fox	TFOX	011.44.1343.7292...	24-JAN-98	SA_REP	9600	0.2	148	80
81	171	William	Smith	WSMITH	011.44.1343.6292...	23-FEB-99	SA_REP	7400	0.15	148	80
82	172	Elizabeth	Bates	EBATES	011.44.1343.5292...	24-MAR-99	SA_REP	7300	0.15	148	80
83	173	Sundita	Kumar	SKUMAR	011.44.1343.3292...	21-APR-00	SA_REP	6100	0.1	148	80
84	174	Ellen	Abel	EABEL	011.44.1644.4292...	11-MAY-96	SA_REP	11000	0.3	149	80
85	175	Alyssa	Hutton	AHUTTON	011.44.1644.4292...	19-MAR-97	SA_REP	8800	0.25	149	80
86	176	Jonathon	Taylor	JTAYLOR	011.44.1644.4292...	24-MAR-98	SA_REP	8600	0.2	149	80
87	177	Jack	Livingston	JLIVINGS	011.44.1644.4292...	23-APR-98	SA_REP	8400	0.2	149	80
88	178	Kimberely	Grant	KGRANT	011.44.1644.4292...	24-MAY-99	SA_REP	7000	0.15	149	(null)
89	179	Charles	Johnson	CJOHNSON	011.44.1644.4292...	04-JAN-00	SA_REP	6200	0.1	149	80
90	180	Winston	Taylor	WTAYLOR	650.507.9876	24-JAN-98	SH_CLERK	3200	(null)	120	50
91	181	Jean	Fleaur	JFLEAUR	650.507.9877	23-FEB-98	SH_CLERK	3100	(null)	120	50
92	182	Martha	Sullivan	MSULLIVA	650.507.9878	21-JUN-99	SH_CLERK	2500	(null)	120	50
93	183	Girard	Geoni	GGEONI	650.507.9879	03-FEB-00	SH_CLERK	2800	(null)	120	50
94	184	Nandita	Sarchand	NSARCHAN	650.509.1876	27-JAN-96	SH_CLERK	4200	(null)	121	50
95	185	Alexis	Bull	ABULL	650.509.2876	20-FEB-97	SH_CLERK	4100	(null)	121	50
96	186	Julia	Dellinger	JDELLING	650.509.3876	24-JUN-98	SH_CLERK	3400	(null)	121	50
97	187	Anthony	Cabrio	ACABRIO	650.509.4876	07-FEB-99	SH_CLERK	3000	(null)	121	50
98	188	Kelly	Chung	KCHUNG	650.505.1876	14-JUN-97	SH_CLERK	3800	(null)	122	50
99	189	Jennifer	Dilly	JDILLY	650.505.2876	13-AUG-97	SH_CLERK	3600	(null)	122	50
100	190	Timothy	Gates	TGATES	650.505.3876	11-JUL-98	SH_CLERK	2900	(null)	122	50
101	191	Randall	Perkins	RPERKINS	650.505.4876	19-DEC-99	SH_CLERK	2500	(null)	122	50
102	192	Sarah	Bell	SBELL	650.501.1876	04-FEB-96	SH_CLERK	4000	(null)	123	50
103	193	Britney	Everett	BEVERETT	650.501.2876	03-MAR-97	SH_CLERK	3900	(null)	123	50
104	194	Samuel	McCain	SMCCAIN	650.501.3876	01-JUL-98	SH_CLERK	3200	(null)	123	50
105	195	Vance	Jones	VJONES	650.501.4876	17-MAR-99	SH_CLERK	2800	(null)	123	50
106	196	Alana	Walsh	AWALSH	650.507.9811	24-APR-98	SH_CLERK	3100	(null)	124	50
107	197	Kevin	Feeney	KFEENEY	650.507.9822	23-MAY-98	SH_CLERK	3000	(null)	124	50

`JOB_HISTORY` Table

```
DESCRIBE job_history
```

```
Name                           Null     Type
------------------------------ -------- -------------
EMPLOYEE_ID                    NOT NULL NUMBER(6)
START_DATE                     NOT NULL DATE
END_DATE                       NOT NULL DATE
JOB_ID                         NOT NULL VARCHAR2(10)
DEPARTMENT_ID                           NUMBER(4)

5 rows selected
```

```
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

C

Writing Advanced Scripts

Objectives

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

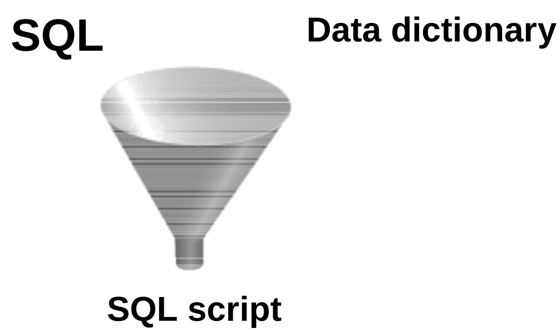
- Describe the type of problems that are solved by using SQL to generate SQL
- Write a script that generates a script of `DROP TABLE` statements
- Write a script that generates a script of `INSERT INTO` statements

Objectives

In this appendix, you learn how to write a SQL script to generates 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
  – Is created and maintained by the Oracle server



Using SQL to Generate SQL

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 run-time parameters

The examples used in this lesson 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;
7	CREATE TABLE COUNTRIES_test AS SELECT * FROM COUNTRIES WHERE 1=2;

A Basic Script

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
```

Set system variables to appropriate values.

SQL STATEMENT

```
SET FEEDBACK ON
SET PAGESIZE 24
SET ECHO ON
```

Set system variables back to the default value.

ORACLE

Controlling the Environment

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.

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 Complete Picture

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;

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
```


Dumping Table Contents to a File

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 output 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'`
`''', '''`	`','`
`''');'`	`');`

Dumping Table Contents to a File (continued)

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 surrounded by quotation marks.

Within a string, to display one single 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 one 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;
```


Generating a Dynamic Predicate

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 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.

Generating a Dynamic Predicate (continued)

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 hire_date: 17-sep-1987
```

The following value for MY_COL is generated:

```
MY_COL
---------------------------------------------------------------------------
where department_id = 10 and hire_date =to_date('17-sep-1987','dd-mon-yyyy')
```

When the second example in the slide is executed, the following output is generated:

```
LAST_NAME
-------------------------
Whalen
```

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

Summary

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 run-time parameters.

D

Oracle Architectural Components

Objectives

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

- Describe the Oracle server architecture and its main components
- List the structures involved in connecting a user to an Oracle instance
- List the stages in processing:
  - Queries
  - Data manipulation language (DML) statements
  - Commits

Objectives

This appendix introduces the Oracle server architecture by describing the files, processes, and memory structures involved in establishing a database connection and executing a SQL command.

Oracle Database Architecture: Overview

The Oracle Database consists of two main components:

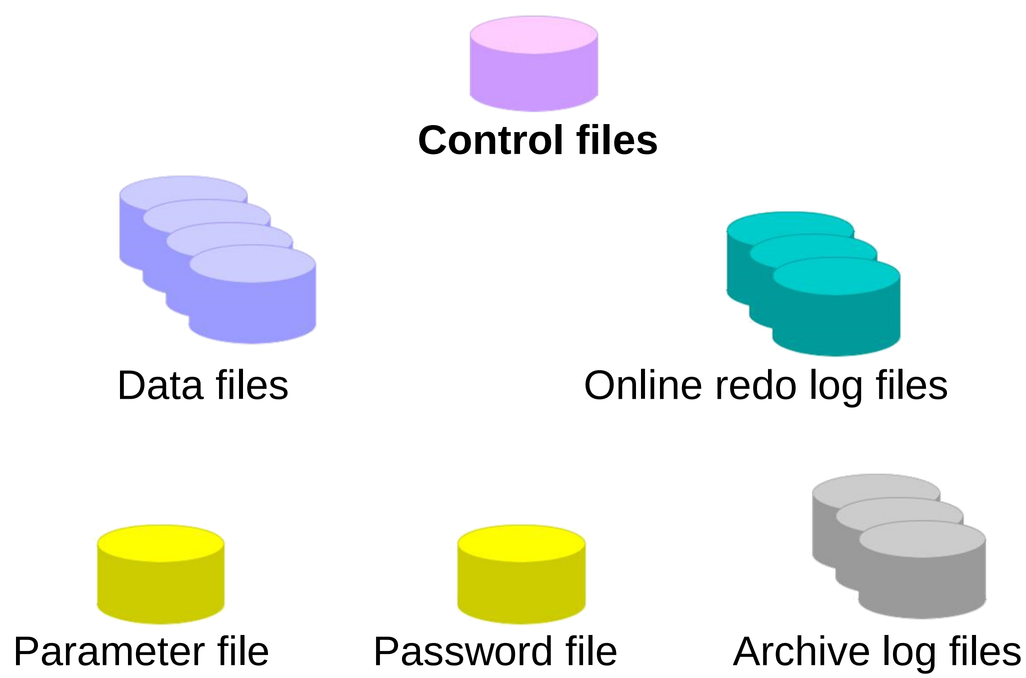
- The database or the physical structures
- The instance or the memory structures

Oracle Database Architecture: Overview

The Oracle Database consists of two main components—the instance and the database itself.

- The database consists of physical files such as:
  - The control file 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
  - The parameter file, which contains the parameters that control the size and properties of an instance
  - The password file, which contains the superuser or SYSDBA password
- The instance consists of the System Global Area (SGA) and the server processes that perform tasks within the database.

Database Physical Architecture



Database Physical Architecture

The files that make up an Oracle Database are organized into the following:

Control files: These files contain data about the database itself, called the metadata. These files are critical to the database. Without them, you cannot open the data files to access the data within the database.

Data files: These files contain the data of the database.

Online redo log files: These files allow for instance recovery of the database. If the database were to crash and not lose any data files, the instance will be able to recover the database with the information in these files.

There are other files that are not officially part of the database but are important to the successful running of the database. These are:

Parameter file: The parameter file is used to define how the instance will be configured when it starts up.

Password file: This file allows users to connect remotely to the database and perform administrative tasks.

Archive log files: These files contain an ongoing history of the redo generated by the instance. These files allow for database recovery. By using these files and a backup of the database, it is possible to recover a lost data file.

Control Files

- Contain physical database structure information
- Should be multiplexed to protect against loss
- Are read at mount stage

Control Files

When you start the instance and mount the database, the control file is read. The entries in the control file specify the physical files that constitute the database.

When you add additional files to your database, the control file is automatically updated.

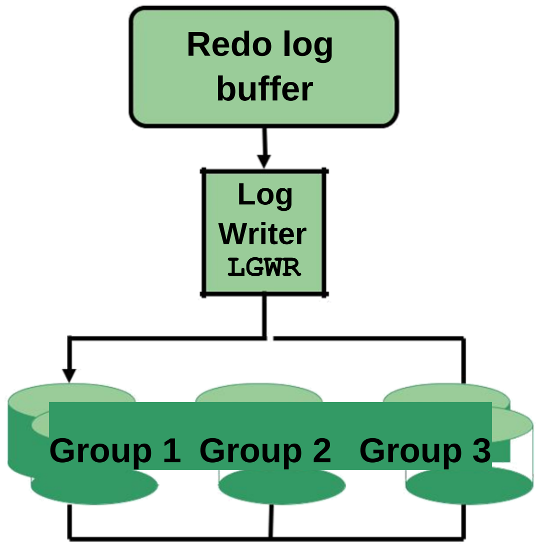
The location of the control files is specified in an initialization parameter.

To protect against failure of the database due to the loss of the control file, you should multiplex the control file on at least three different physical devices. By specifying multiple files through the initialization parameter, you enable the Oracle Database server to maintain multiple copies of the control file.

Redo Log Files

Record changes to the database

Should be multiplexed to protect against loss



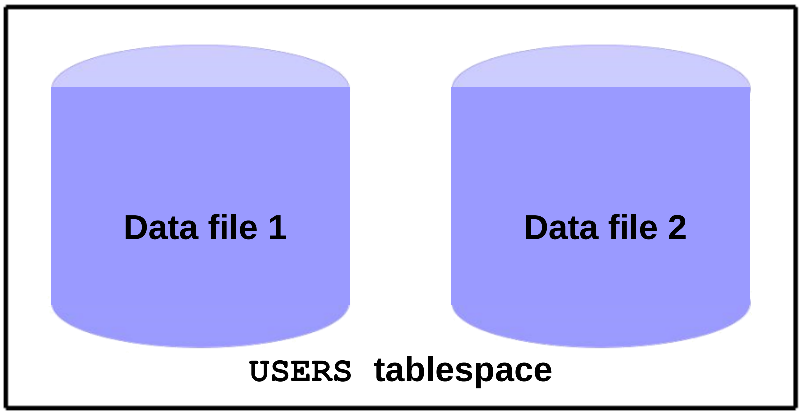
Redo Log Files

Redo log files are used to record changes to the database as a result of transactions and internal Oracle Database server actions. They protect the database from loss of integrity due to system failures caused by power outages, disk failures, and so on. Redo log files should be multiplexed to ensure that the information stored in them is not lost in the event of a disk failure.

The redo log consists of groups of redo log files. A group consists of a redo log file and its multiplexed copies. Each identical copy is said to be a member of that group and each group is identified by a number. The log writer (`LGWR`) process writes redo records from the redo log buffer to a redo log group until the file is filled or a log switch operation is requested. Then it switches and writes to the files in the next group. The redo log groups are used in a circular fashion.

Tablespaces and Data Files

Tablespaces consist of one or more data files.
Data files belong to only one tablespace.



Tablespaces and Data Files

A database is divided into logical storage units called tablespaces, which can be used to group related logical structures. 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.

Note: You can also create `bigfile` tablespaces, which are tablespaces with a single, but very large (up to 4-G blocks) data file. Traditional `smallfile` tablespaces (which are the default) can contain multiple data files, but the files cannot be as large. For more information about `bigfile` tablespaces, refer to the *Database Administrator's Guide*.

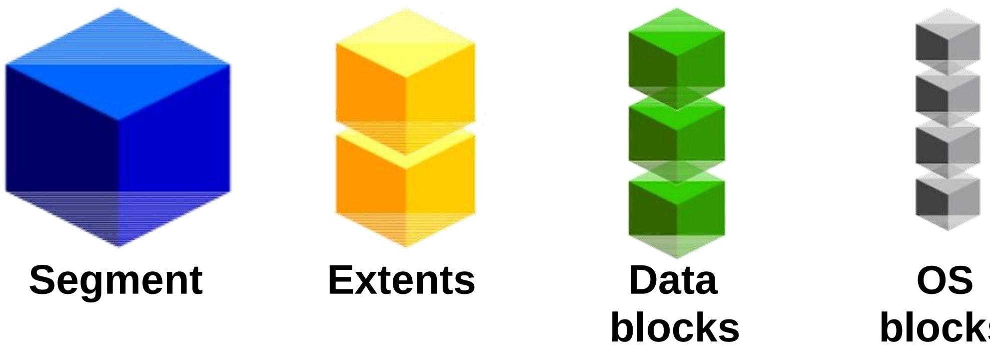
Segments, Extents, and Blocks

Segments exist within a tablespace.

Segments consist of a collection of extents.

Extents are a collection of data blocks.

Data blocks are mapped to OS blocks.



Segments, Extents, and Blocks

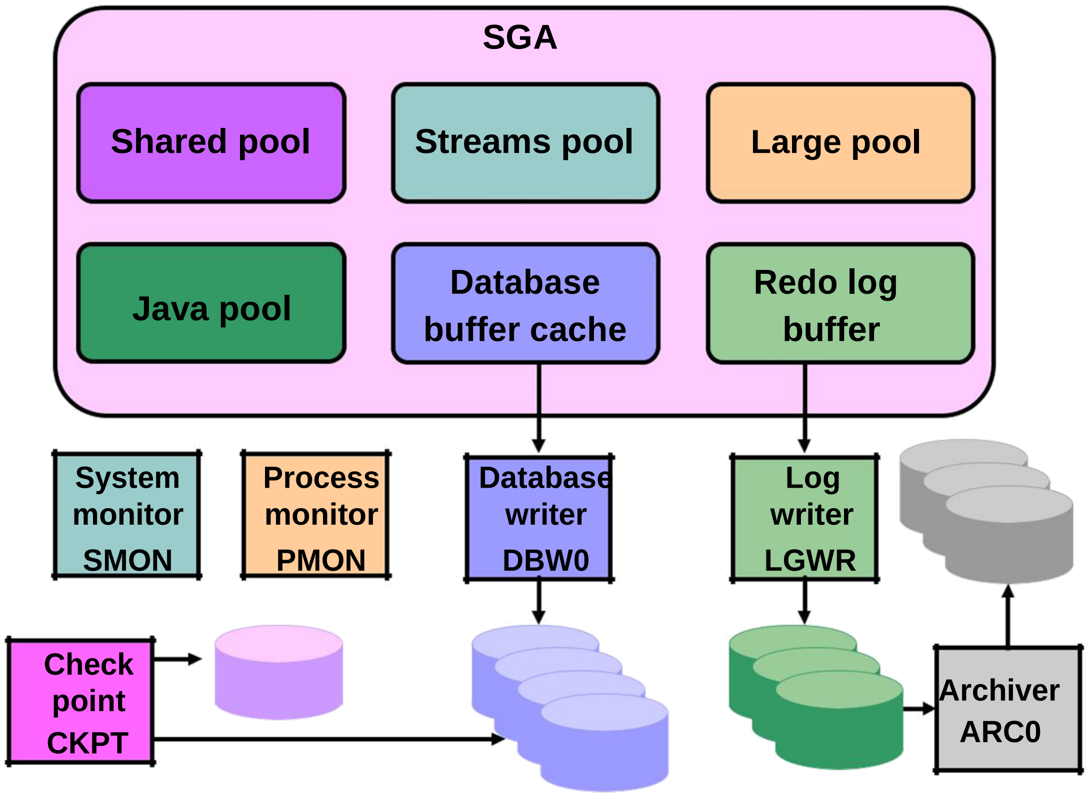
Database objects such as tables and indexes are stored in tablespaces as segments. Each segment contains one or more extents. An extent consists of contiguous data blocks, which means that each extent can exist only in one data file. Data blocks are the smallest unit of I/O in the database.

When the database requests a set of data blocks from the operating system (OS), the OS maps this to the actual OS block on the storage device. Because of this, you need not be aware of the physical address of any of the data in your database. This also means that a data file can be striped and or mirrored on several disks.

The size of the data block can be set at database creation time. The default size of 8 K is adequate for most databases. If your database supports a data warehouse application that has large tables and indexes, then a larger block may be beneficial. If your database supports a transactional application where reads and write are very random, then specifying a smaller block size may be beneficial. The maximum block size is dependent on your OS. The minimum block size is 2 K and should rarely (if ever) be used.

You can have tablespaces with different block sizes. Generally, this should be used only to support transportable tablespaces. For details, refer to the *Database Administrator's Guide*.

Oracle Instance Management



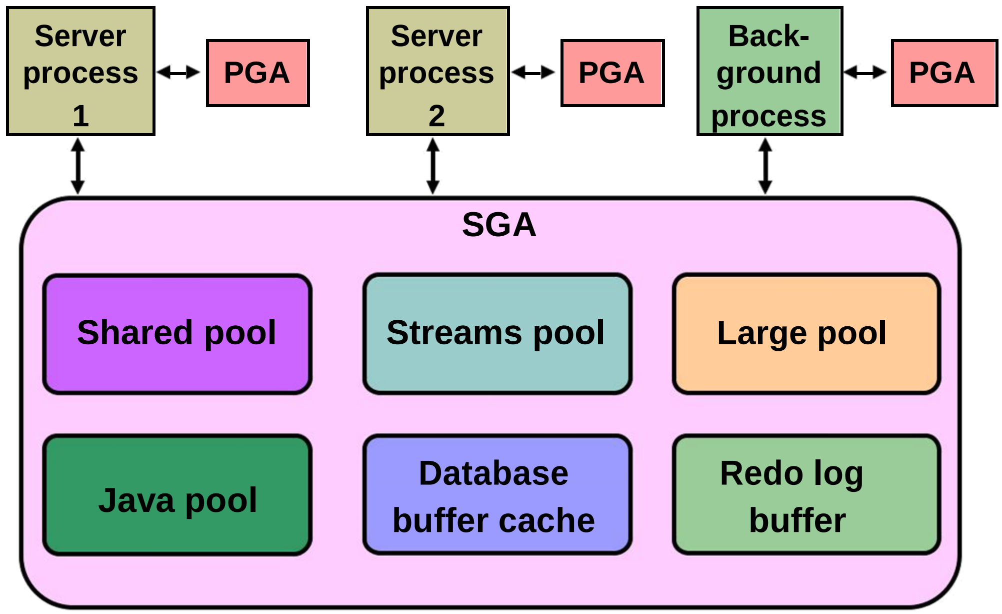
Oracle Instance Management

An Oracle Database server consists of an Oracle Database and an Oracle instance. An Oracle instance consists of memory buffers known as the System Global Area (SGA) and background processes.

The instance is idle (nonexistent) until it is started. When the instance is started, an initialization parameter file is read and the instance is configured accordingly.

After the instance is started and the database is opened, users can access the database.

Oracle Memory Structures



Oracle Memory Structures

The basic memory structures associated with an Oracle instance include:

- **System Global Area (SGA):** Shared by all server and background processes
- **Program Global Area (PGA):** Private to each server and background process; there is one PGA for each process

The System Global Area (SGA) is a shared memory area that contains data and control information for the instance.

The SGA consists of 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 disk
- **Shared pool:** Caches various constructs that can be shared among users
- **Large pool:** Is an optional area used for buffering large I/O requests
- **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

When you start the instance by using Enterprise Manager or SQL*Plus, the memory allocated for the SGA is displayed.

Oracle Memory Structures (continued)

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 can be changed without shutting down the instance.

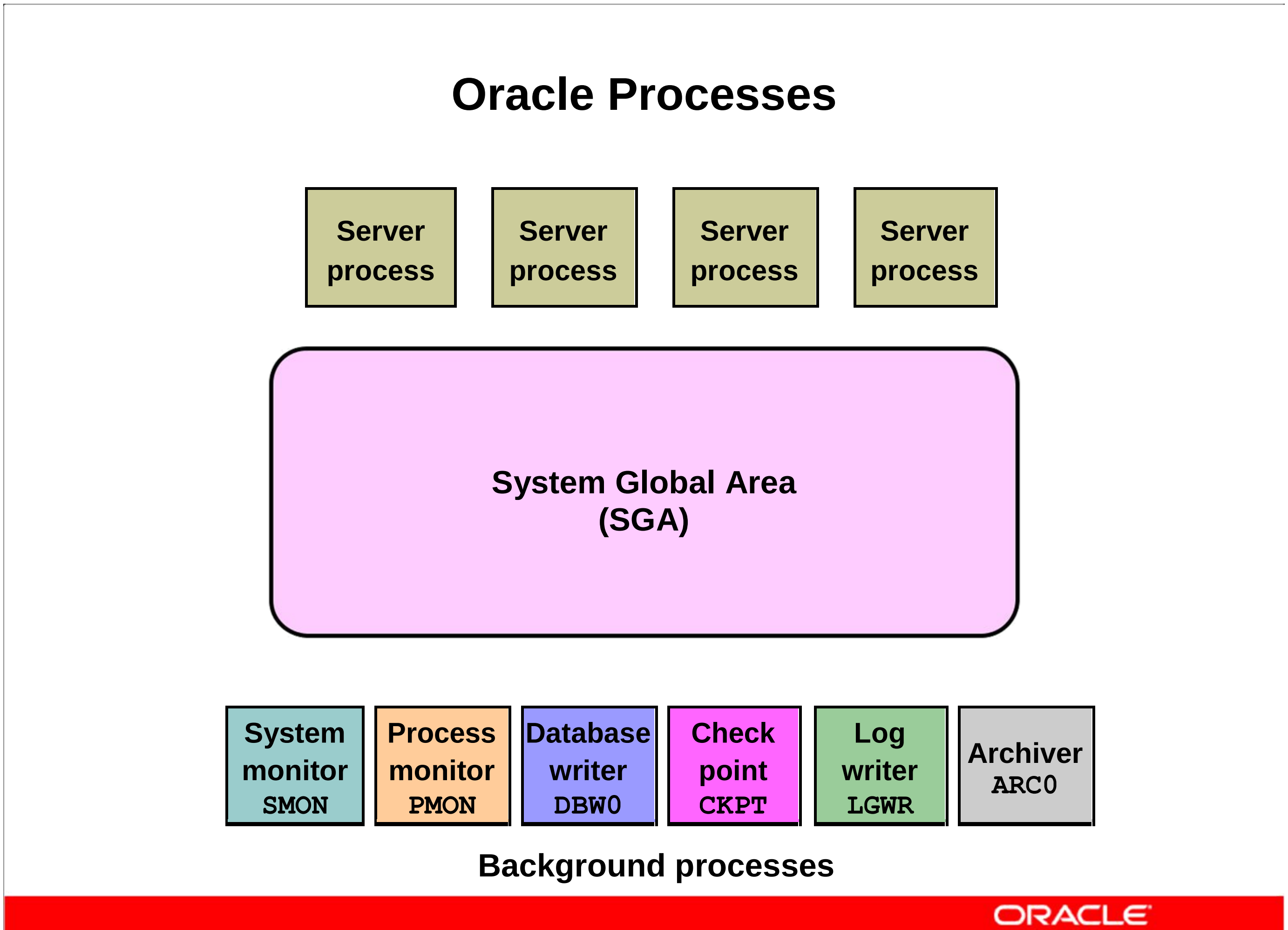
The preconfigured database has been pretuned with adequate settings for the memory parameters. However, as your database usage expands, you may find it necessary to alter the settings of the memory parameters.

Oracle provides alerts and advisors to identify memory-sizing problems and to help you determine appropriate values for memory parameters.

A Program Global Area (PGA) is a memory region that contains data and control information for each server process. A server process services a client's requests. Each server process has its own private PGA area that is created when the server process is started. Access to it is exclusive to that server process, and is read and written only by the Oracle code acting on behalf of it.

The amount of PGA memory used and its content depends on whether the instance is configured in shared server mode. Generally, the PGA contains the following:

- **Private SQL area:** Contains data such as bind information and run-time memory structures. Each session that issues a SQL statement has a private SQL area.
- **Session memory:** Is memory allocated to hold session variables and other information related to the session



Oracle Processes

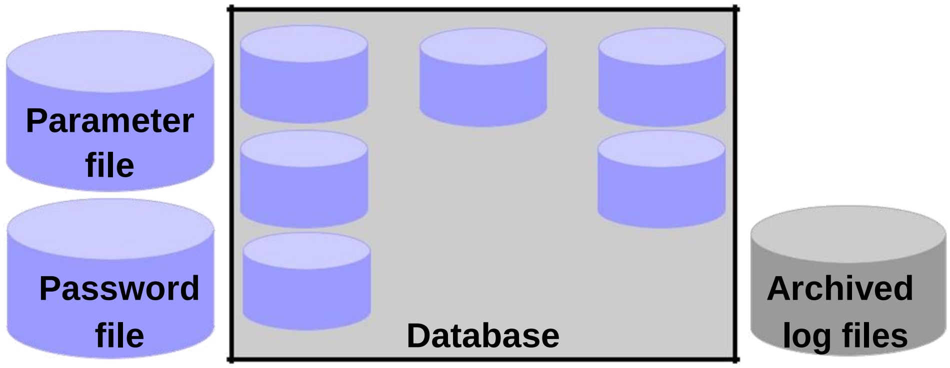
When you invoke an application program or an Oracle tool such as Enterprise Manager, the Oracle server creates a server process to execute commands issued by the application.

Oracle also creates a set of background processes for an instance that interact with each other and with the operating system to manage the memory structures, asynchronously perform I/O to write data to disk, and do general housekeeping.

Which background processes are present depends upon the features that are being used in the database. The most common background processes are the following:

- **System monitor (`SMON`):** Performs crash recovery when the instance is started following a failure
- **Process monitor (`PMON`):** Performs process cleanup when a user process fails
- **Database writer (`DBWn`):** Writes modified blocks from the database buffer cache to the files on disk
- **Checkpoint (`CKPT`):** Signals `DBWn` at checkpoints and updates all the data files and control files of the database to indicate the most recent checkpoint
- **Log writer (`LGWR`):** Writes redo log entries to disk
- **Archiver (`ARCn`):** Copies the redo log files to archival storage when the log files are full or a log switch occurs

Other Key Physical Structures



Other Key Files

The Oracle server also uses other files that are not part of the database:

- The parameter file defines the characteristics of an Oracle instance. For example, it contains parameters that size some of the memory structures in the SGA.
- The password file authenticates which users are permitted to start up and shut down an Oracle instance.
- Archived redo log files are offline copies of the redo log files that may be necessary for recovery from media failures.

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.
- Commit ensures transaction recovery.

Some Oracle server components do not participate in SQL statement processing.

Components Used to Process SQL

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.

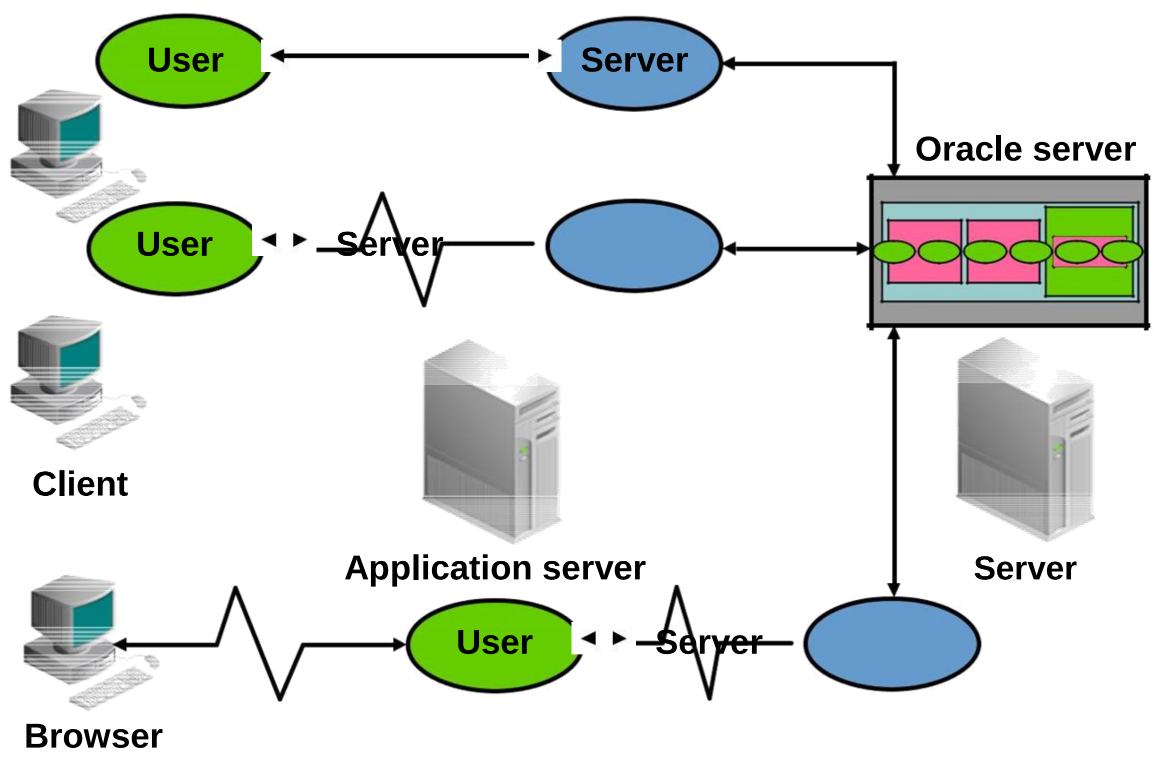
Some of the background processes, SGA structures, and database files are used to process SQL statements. Depending on the type of SQL statement, different components are used:

- Queries require additional processing to return rows to the user.
- Data manipulation language (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.

The optional background process, ARC0, is used to ensure that a production database can be recovered.

Connecting to an Instance



Processes Used to Connect to an Instance

Before users can submit SQL statements to the Oracle server, they must connect to an instance.

The user starts a tool such as SQL Developer or runs an application developed using a tool such as Oracle Forms. This application or tool is executed in a *user process*.

In the most basic configuration, when a user logs on to the Oracle server, a process is created on the computer running the Oracle server. This process is called a server process. The server process communicates with the Oracle instance on behalf of the user process that runs on the client. The server process executes SQL statements on behalf of the user.

Connection

A connection is a communication pathway between a user process and an Oracle server. A database user can connect to an Oracle server in one of three ways:

The user logs on to the operating system running the Oracle instance and starts an application or tool that accesses the database on that system. The communication pathway is established using the interprocess communication mechanisms available on the host operating system.

Processes Used to Connect to an Instance (continued)

Connection (continued)

The user starts the application or tool on a local computer and connects over a network to the computer running the Oracle instance. In this configuration, called client/server, network software is used to communicate between the user and the Oracle server.

In a three-tiered connection, the user's computer communicates over the network to an application or a network server, which is connected through a network to the machine running the Oracle instance. For example, the user runs a browser on a network computer to use an application residing on an NT server that retrieves data from an Oracle Database running on a UNIX host.

Sessions

A session is a specific connection of a user to an Oracle server. The session starts when the user is validated by the Oracle server, and it ends when the user logs out or when there is an abnormal termination. For a given database user, many concurrent sessions are possible if the user logs on from many tools, applications, or terminals at the same time. Except for some specialized database administration tools, starting a database session requires that the Oracle server be available for use.

Note: The type of connection explained here, where there is a one-to-one correspondence between a user and server process, is called a *dedicated server connection*.

Processing a Query

Parse:

- Search for identical statement.
- Check syntax, object names, and privileges.
- Lock objects used during parse.
- Create and store execution plan.

Execute: Identify rows selected.

Fetch: Return rows to user process.

Query Processing Steps

Queries are different from other types of SQL statements because, if successful, they return data as results. Whereas other statements simply return success or failure, 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.

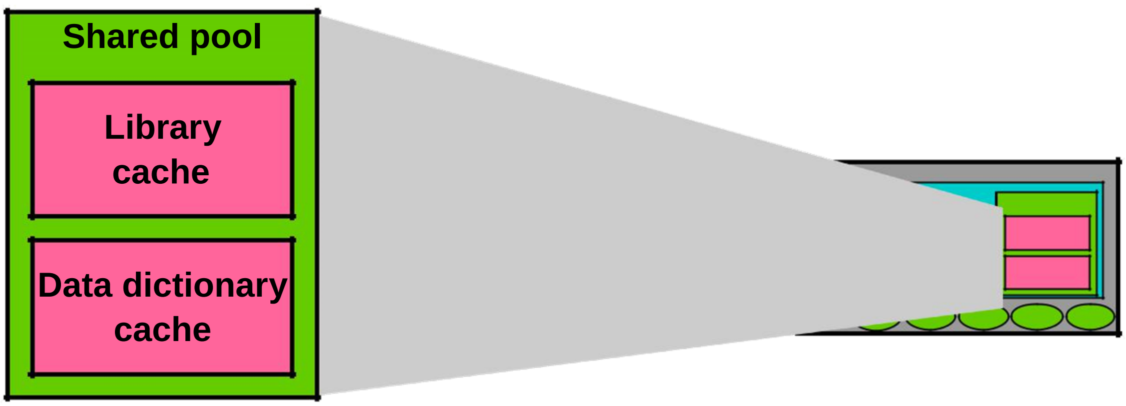
During parse, the server process performs the following functions:

- 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 fetch 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`.



Shared Pool Components

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 reexecuted and a shared SQL area already contains the execution plan for the statement, then 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.

Shared Pool Components (continued)

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

It 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`.

Functions of the Database Buffer Cache

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.

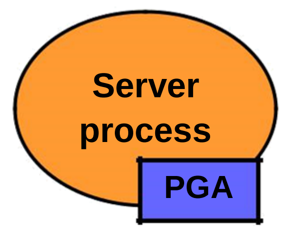
Program Global Area (PGA)

Is not shared

Is writable only by the server process

Contains:

- Sort area
- Session information
- Cursor state
- Stack space



ORACLE

Program Global Area: Components

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

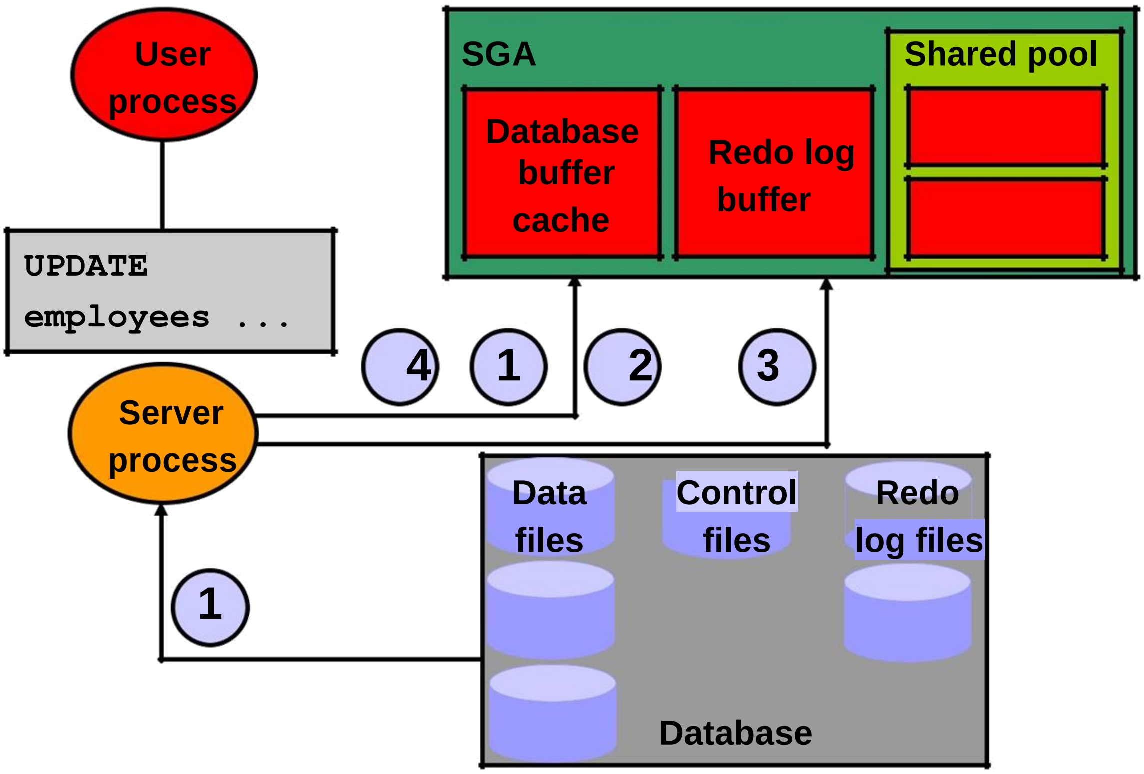
Session information: Includes user privileges and performance statistics for the session

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



DML Processing Steps

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.

DML Processing Steps (continued)

DML Execute Phase (continued)

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

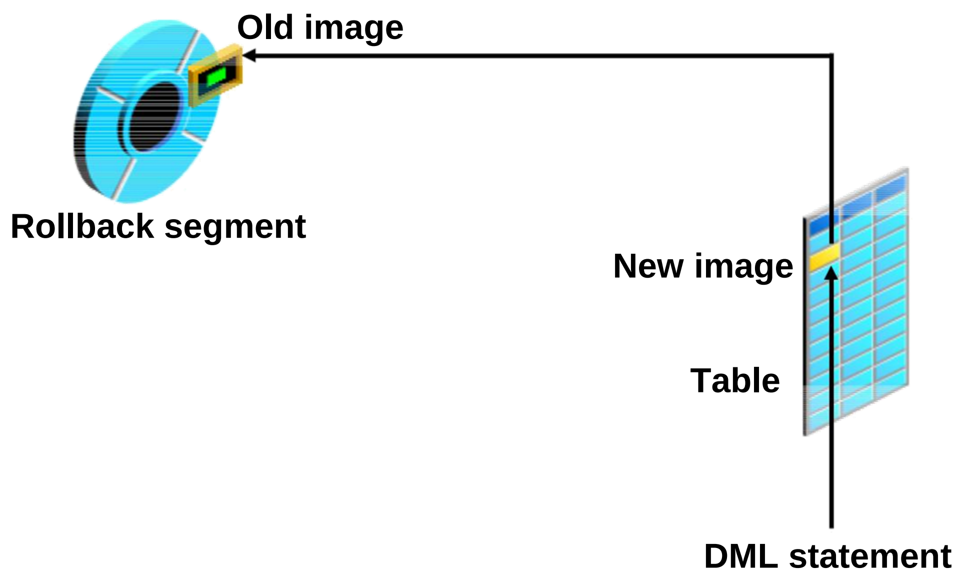
- Has its size defined by LOG_BUFFER
- Records changes made through the instance
- Is used sequentially
- Is a circular buffer

Redo Log Buffer: Characteristics

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 type of block that is 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



Rollback Segment

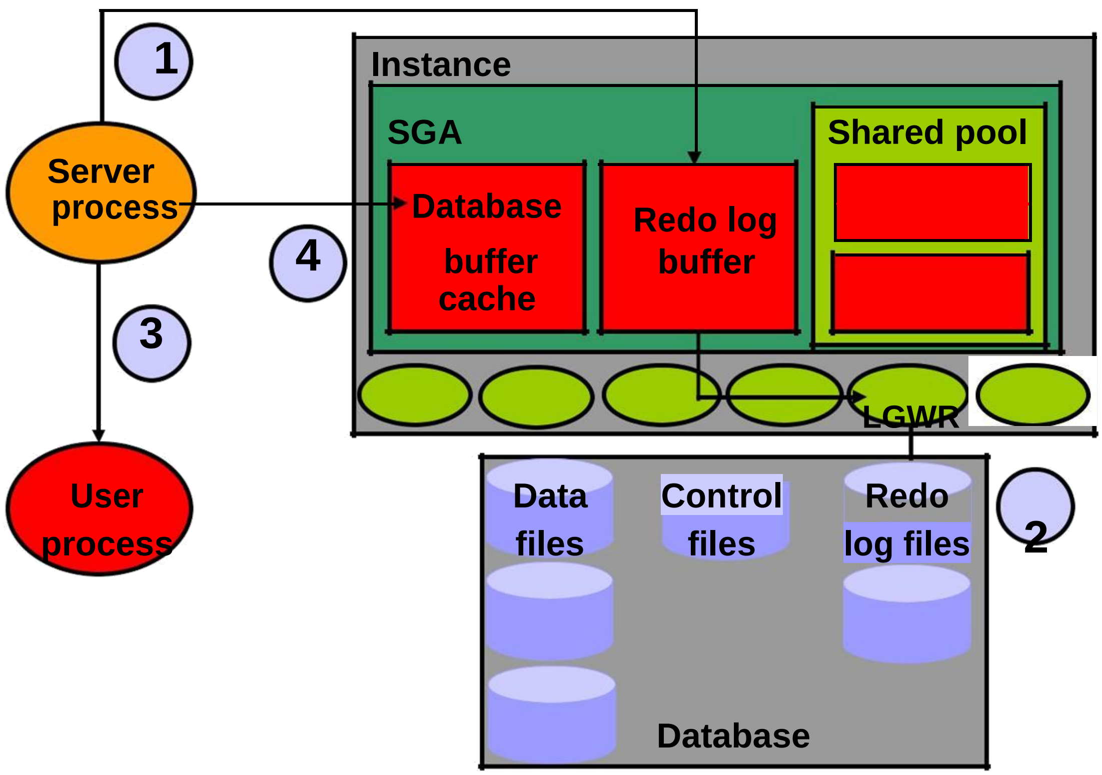
Before making a change, the server process saves the old data value in a rollback segment. This "before image" is used to:

- 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



Fast COMMIT

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 system change number (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 COMMITs

When a COMMIT is issued, the following steps are performed:

The server process places a commit record, along with the SCN, in the redo log buffer.

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.

Fast COMMIT (continued)

Steps in Processing COMMITs (continued)

- The user is informed that the COMMIT is complete.
- 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

In this appendix, you should have learned how to:

- Identify database files: data files, control files, and online redo logs
- Describe SGA memory structures: DB buffer cache, shared SQL pool, and redo log buffer
- Explain primary background processes: `DBW0, LGWR, CKPT, PMON, SMON`, and `ARC0`
- List SQL processing steps: parse, execute, and fetch

Summary

Oracle Database Files

The Oracle Database includes the following files:

- **Control files:** Contain information required to verify the integrity of the database, including the names of the other files in the database (The control files are usually mirrored.)
- **Data files:** Contain the data in the database, including tables, indexes, rollback segments, and temporary segments
- **Online redo logs:** Contain the changes made to the data files (Online redo logs are used for recovery and are usually mirrored.)

Other files commonly used with the database include:

- **Parameter file:** Defines the characteristics of an Oracle instance
- **Password file:** Authenticates privileged database users
- **Archived redo logs:** Are backups of the online redo logs

Summary (continued)

SGA Memory Structures

The System Global Area (SGA) has three primary structures:

- **Shared pool:** Stores the most recently executed SQL statements and the most recently used data from the data dictionary
- **Database buffer cache:** Stores the most recently used data
- **Redo log buffer:** Records changes made to the database using the instance

Background Processes

A production Oracle instance includes the following processes:

- **Database writer (`DBW0`):** Writes changed data to the data files
- **Log writer (`LGWR`):** Records changes to the data files in the online redo log files
- **System monitor (`SMON`):** Checks for consistency and initiates recovery of the database when the database is opened
- **Process monitor (`PMON`):** Cleans up the resources if one of the processes fails
- **Checkpoint process (`CKPT`):** Updates the database status information after a checkpoint
- **Archiver (`ARC0`):** Backs up the online redo log to ensure recovery after a media failure (This process is optional, but is usually included in a production instance.)

Depending on its configuration, the instance may also include other processes.

SQL Statement Processing Steps

The steps used to process a SQL statement include:

- **Parse:** Compiles the SQL statement
- **Execute:** Identifies selected rows or applies DML changes to the data
- **Fetch:** Returns the rows queried by a `SELECT` statement

Using SQL Developer

Objectives

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

- List the key features of Oracle SQL Developer
- Install Oracle SQL Developer
- Identify menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use SQL Worksheet
- Execute SQL statements and SQL scripts
- Create and save reports

Objectives

In this appendix, you are introduced to the graphical tool 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 free graphical tool that enhances productivity and simplifies database development tasks.

You can connect to any target Oracle Database schema using standard Oracle Database authentication.

What Is Oracle SQL Developer?

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 using standard Oracle Database authentication. When connected, you can perform operations on objects in the database.

Key Features

- Developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Default connectivity by using the JDBC Thin driver
- Does not require an installer
- Connects to any Oracle Database version 9.2.0.1 and later
- Bundled with JRE 1.5

Key Features of SQL Developer

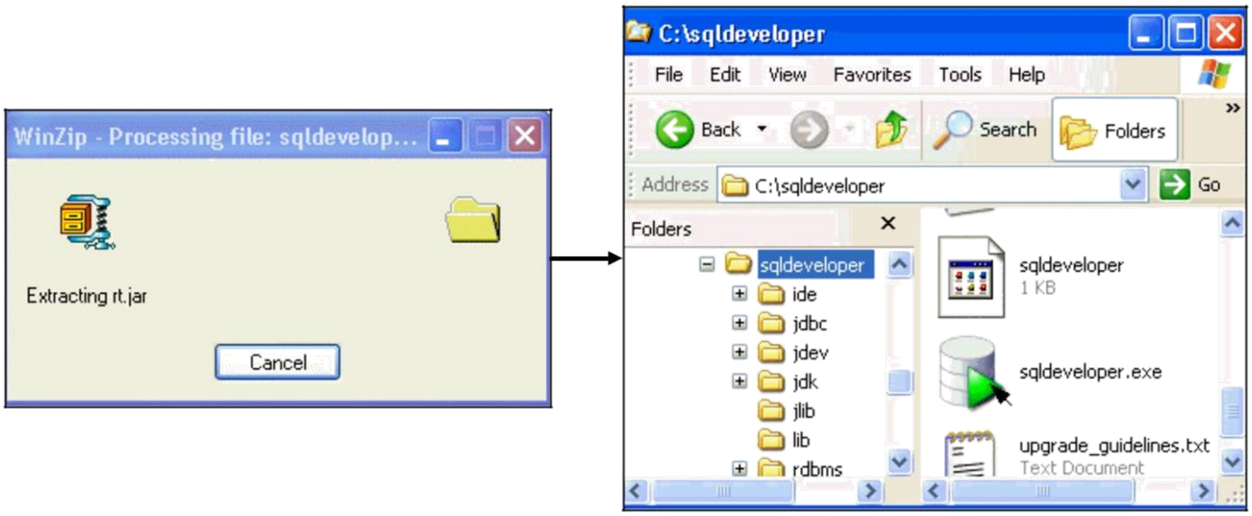
Oracle SQL Developer is developed in Java, leveraging the Oracle JDeveloper IDE. The tool runs on the Windows, Linux, and Mac OS X platforms. You can install SQL Developer on the Database Server and connect remotely from your desktop, thus avoiding client server network traffic.

Default connectivity to the database is through the JDBC Thin driver; so, no Oracle Home is required. SQL Developer does not require an installer and you need to just unzip the downloaded file.

With SQL Developer, users can connect to Oracle Database 9.2.0.1 and later versions, and all Oracle Database editions including Express Edition. SQL Developer is bundled with JRE 1.5, with an additional `tools.jar` to support Windows clients. Non-Windows clients need only JDK 1.5.

Installing SQL Developer

Download the Oracle SQL Developer kit and unzip it into any directory on your machine.



Installing SQL Developer

Oracle SQL Developer does not require an installer. To install SQL Developer, you need an unzip tool.

To install SQL Developer, perform the following steps:

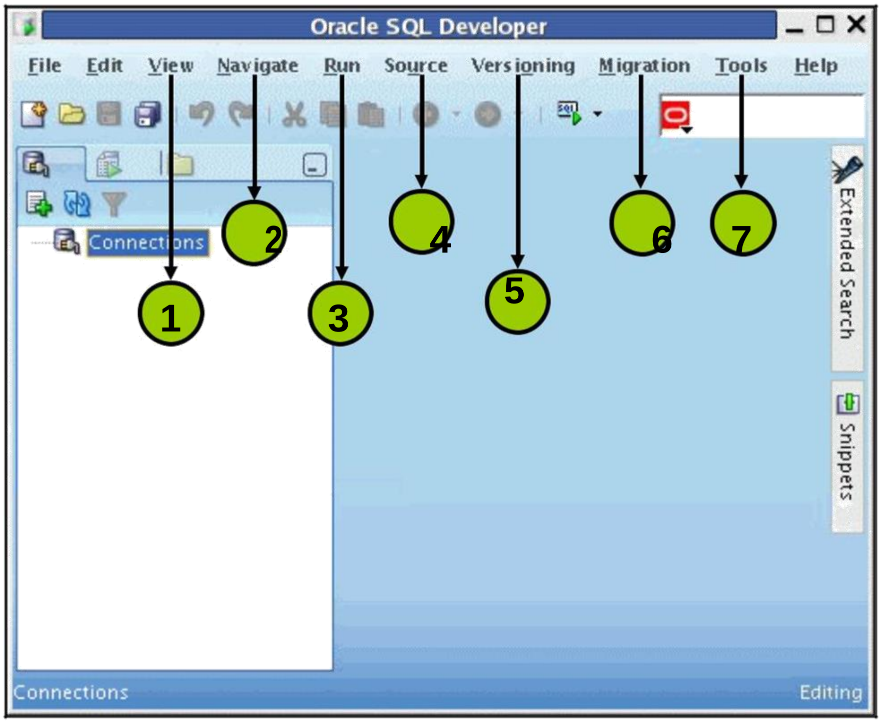
Create a folder as `<local drive>:\SQL Developer`.

Download the SQL Developer kit from http://www.oracle.com/technology/software/products/sql/index.html.

Unzip the downloaded SQL Developer kit into the folder created in step 1.

To start SQL Developer, go to `<local drive>:\SQL Developer`, and double-click `sqldeveloper.exe`.

Menus for SQL Developer



Menus for SQL Developer

SQL Developer has two main navigation tabs.

Connections Navigator: By using this tab, you can browse database objects and users to which you have access.

Reporting Tab: By using this tab, you can run predefined reports or create and add your own reports.

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. The menus at the top 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 panes and for the execution of subprograms
Run: Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected
Source: Contains options for use when editing functions and procedures
Versioning: Enables you to work with the files placed under source code control
Migration: Enables you to migrate from another database, such as Microsoft SQL Server and Microsoft Access, to an Oracle Database
Tools: Invokes SQL Worksheet, Preferences, and any added External Tools

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
  - For multiple schemas
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an XML file.
- Each additional database connection created is listed in the connections navigator hierarchy.

ORACLE

Creating a Database Connection

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.

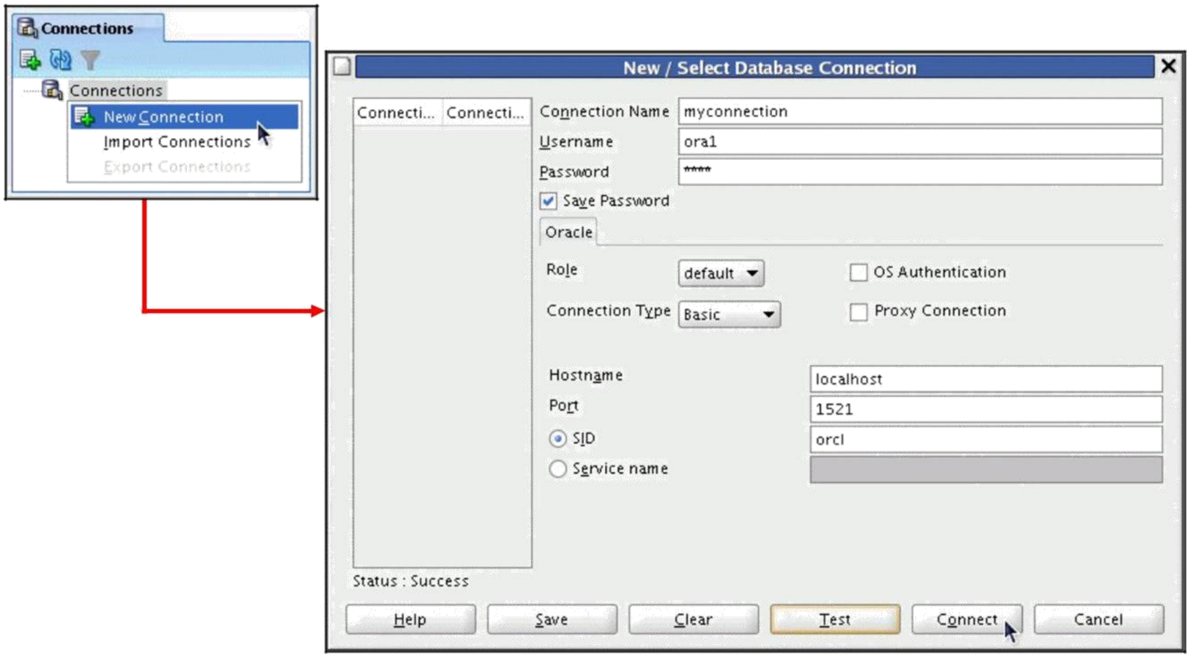
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory. But, it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and display the database connections dialog box, SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.

Note: On Windows systems, 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 to connect to the same database but as different users, or to connect to different databases. Each database connection is listed in the Connections navigator hierarchy.

Creating a Database Connection



Creating a Database Connection (continued)

To create a database connection, perform the following steps:

Double-click <*your_path*>\sqldeveloper\sqldeveloper.exe.

On the **Connections** tabbed page, right-click **Connections** and select **New Connection**.

Enter the connection name, username, password, host name, port number, and SID for the database you want to connect.

Click **Test** to make sure that the connection has been set correctly.

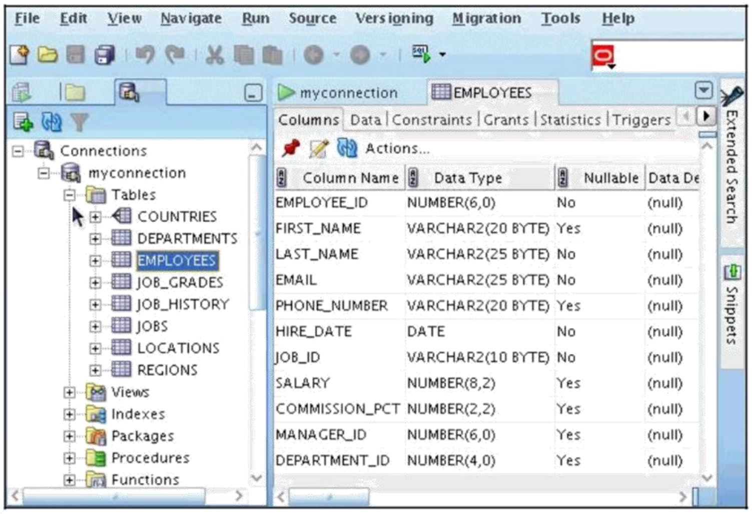
Click **Connect**.

Note: 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 will not be prompted for the password.

Browsing Database Objects

Use the Database Navigator to:

- Browse through many objects in a database schema
- Perform a quick review of the definitions of objects



Browsing Database Objects

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, Types, and so on.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about the selected objects. You can customize many aspects of the appearance of SQL Developer by setting preferences.

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 more are all displayed in an easy-to-read tabbed page.

If you want to see the definition of the `EMPLOYEES` table as shown on the slide, perform the following steps:

1. Expand the connection node in the Connections Navigator.
2. Expand **Tables**.
3. Double-click **EMPLOYEES**.

Using the Data tab, you can enter new rows, update data, and commit these changes to the database.

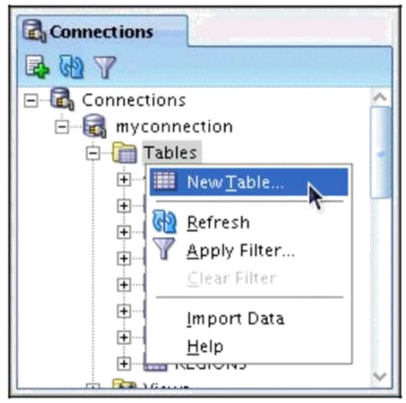
Creating a Schema Object

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 many context-sensitive menus.

View the DDL for adjustments such as creating a new object or editing an existing schema object.



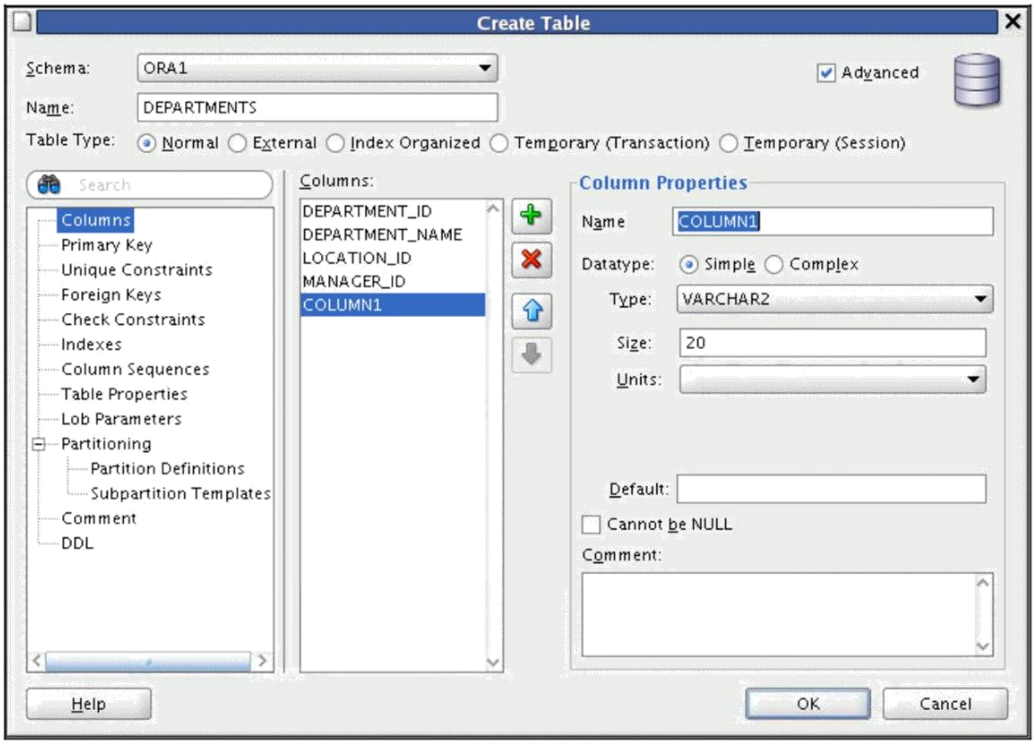
Creating a Schema Object

SQL Developer supports the creation of any schema object by executing a SQL statement in SQL Worksheet. Alternatively, you can create objects using the context menus. Once created, you can edit the objects using an edit dialog or one of 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 creating a table using the context menu. To open a dialog box for creating a new table, right-click **Tables** and select **New Table**. The dialog boxes for creating and editing database objects have multiple tabs, each reflecting a logical grouping of properties for that type of object.

Creating a New Table: Example



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 tabs, in which you can specify an extended set of features while creating the table.

The example in the slide shows creating the `DEPENDENT` table by selecting the **Advanced** check box.

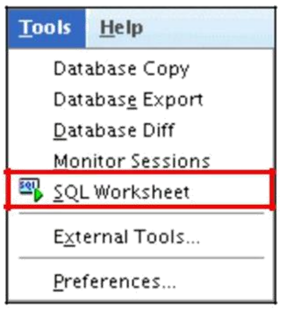
To create a new table, perform the following steps:

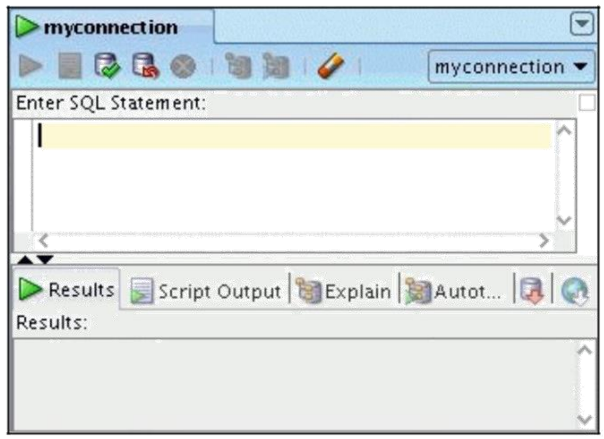
In the Connections Navigator, right-click **Tables**.
Select **New Table**.
In the Create Table dialog box, select **Advanced**.
Specify column information.
Click **OK**.

Although it is not required, you should also specify a primary key by selecting Primary Key in the dialog box. Sometimes, you may want to edit the table that you have created. To edit a table, right-click the table in the Connections Navigator and select **Edit**.

Using SQL Worksheet

Use 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.





Using SQL Worksheet

When you connect to a database, a SQL Worksheet window for that connection is automatically opened. You can use 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.

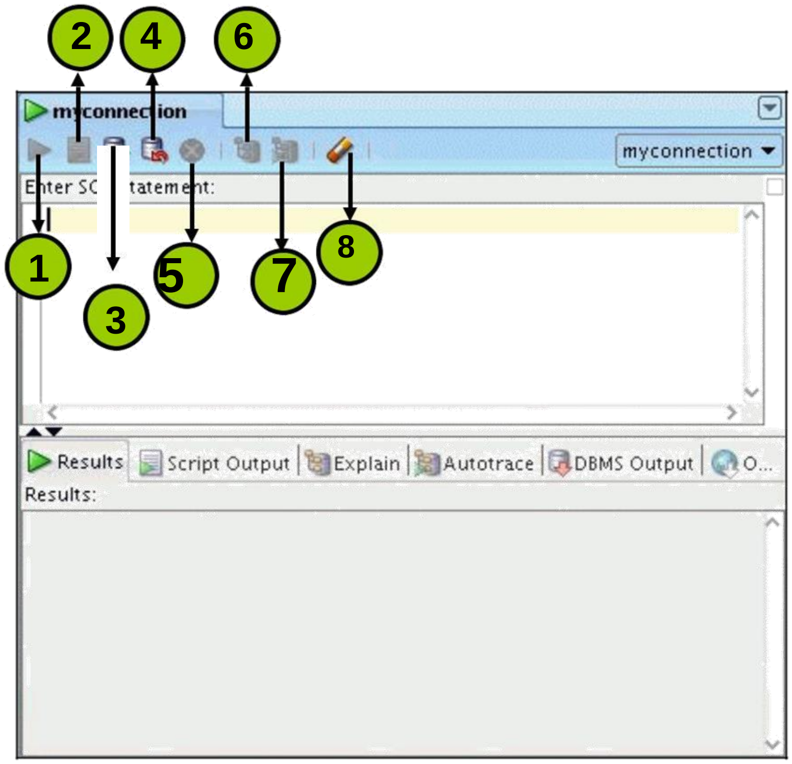
You can specify any actions that can be processed by the database connection associated with the worksheet, such as:

- 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 any of the following two options:

- Select **Tools** > **SQL Worksheet**.
- Click the **Open SQL Worksheet** icon.

Using SQL Worksheet



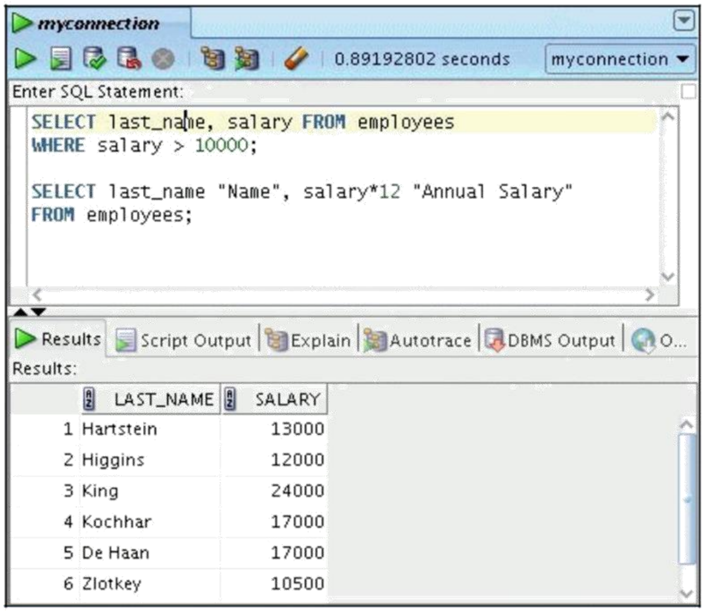
Using SQL Worksheet (continued)

You may want to use 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:

Execute Statement: Executes the statement at the cursor in the Enter SQL Statement box. You can use bind variables in the SQL statements but not substitution variables.
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.
Commit: Writes any changes to the database, and ends the transaction
Rollback: Discards any changes to the database, without writing them to the database, and ends the transaction
Cancel: Stops the execution of any statements currently being executed
Execute Explain Plan: Generates the execution plan, which you can see by clicking the Explain tab
Autotrace: Generates trace information for the statement, which you can see by clicking the Autotrace tab
Clear: Erases the statement or statements in the Enter SQL Statement box

Executing SQL Statements

Use the Enter SQL Statement box to enter single or multiple SQL statements.



Executing SQL Statements

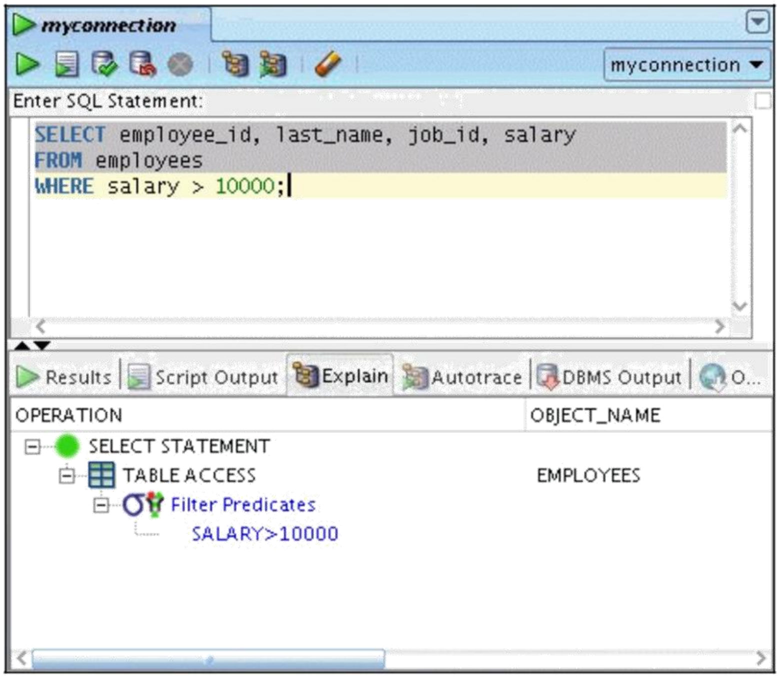
In SQL Worksheet, you can use the Enter SQL Statement box to enter a single or multiple SQL statements. For a single statement, the semicolon at the end is optional.

When you type in the statement, the SQL keywords are automatically highlighted. To execute a SQL statement, ensure that your cursor is within the statement and click the **Execute Statement** icon. Alternatively, you can press the **F9** key.

To execute multiple SQL statements and see the results, click the **Run Script** icon. Alternatively, you can press the **F5** key.

In the example in the slide, as there are multiple SQL statements, the first statement is terminated with a semicolon. The cursor is in the first statement and, therefore, when the statement is executed, results corresponding to the first statement are displayed in the Results box.

Viewing the Execution Plan



Viewing the Execution Plan

You can execute a SQL script and view the execution plan. To execute a SQL script file, perform the following steps:

From the **File** menu, Select **Open**.

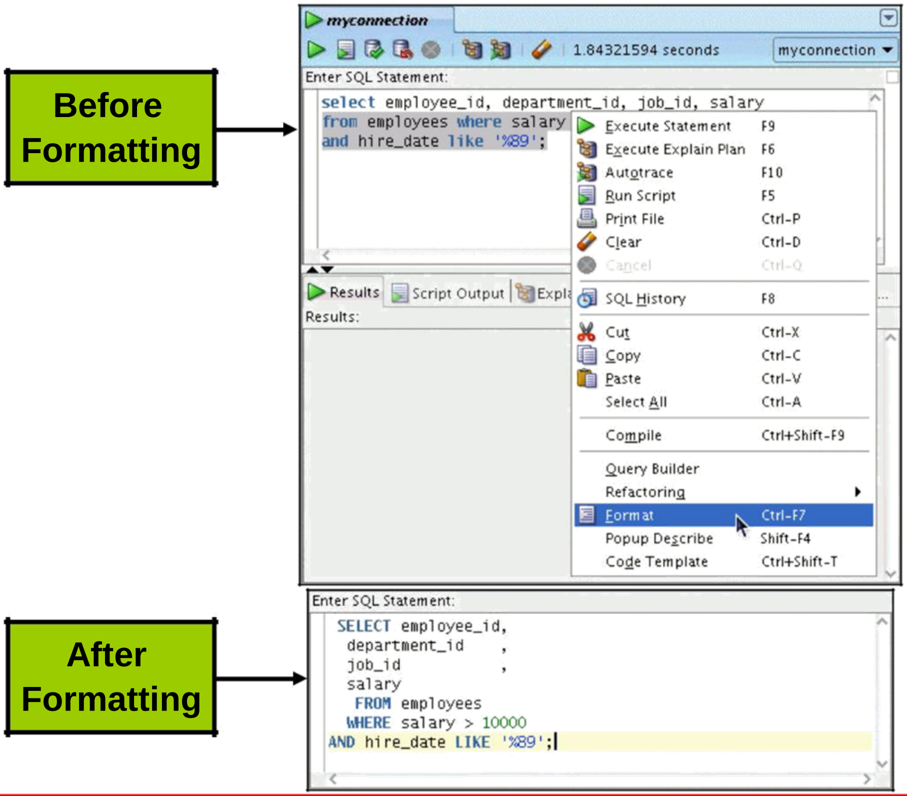
In the Open dialog box, double-click the **`.sql`** file.

Click the **Run Script** icon.

When you double-click the `.sql` file, the SQL statements are loaded into the Enter SQL Statement box. You can execute the script or each line individually. The results are displayed in the Script Output area.

The example in the slide shows the execution plan. The Execute Explain Plan icon generates the execution plan. An execution plan is the sequence of operations that are performed to execute the statement. You can see the execution plan by clicking the **Explain** tab.

Formatting the SQL Code



Formatting the SQL Code

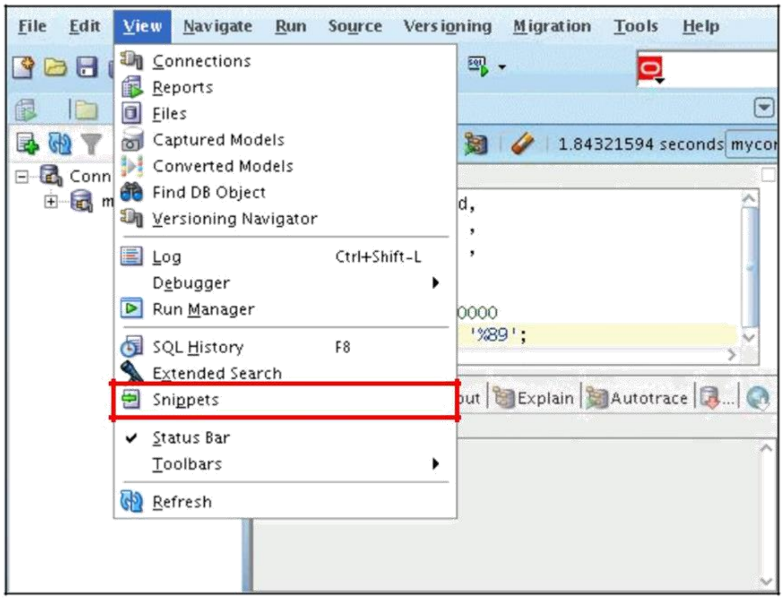
You may want to beautify the indentation, spacing, capitalization, and line separation of the SQL code. SQL Developer has the feature of formatting the SQL code.

To format the SQL code, right-click in the statement area, and select **Format**.

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.



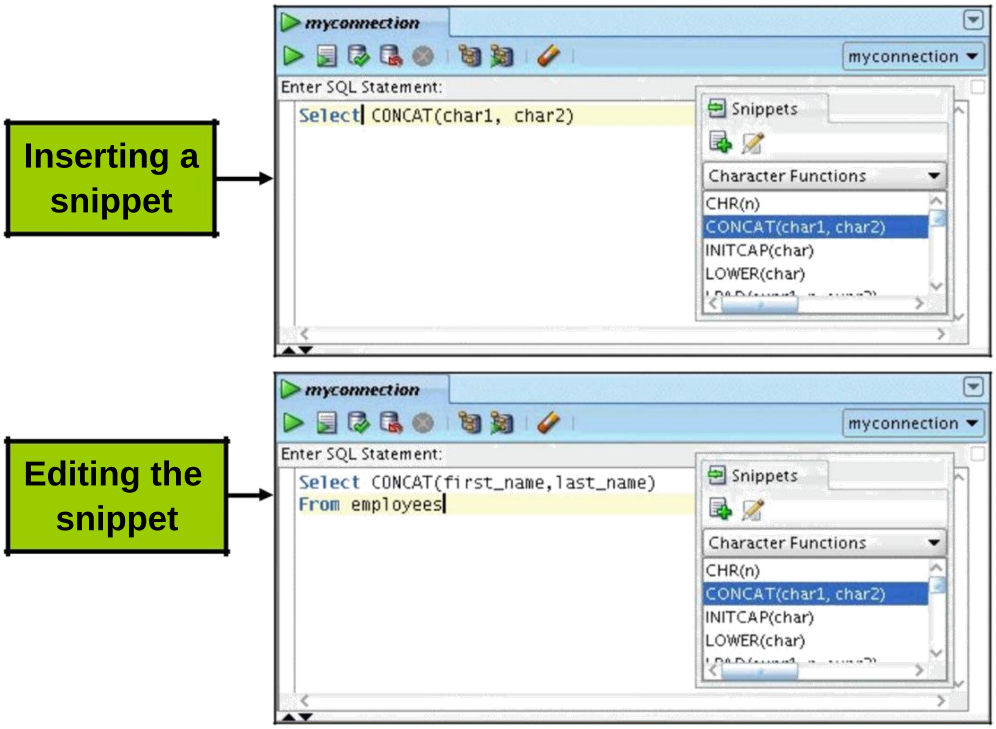
Using Snippets

You may want to use certain code fragments when you are using the SQL Worksheet or creating or editing a PL/SQL function or procedure. SQL Developer has the feature called Snippets. Snippets are code fragments, such as SQL functions, Optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag and drop snippets into the editor window.

To display Snippets, select **View** > **Snippets**.

The Snippets window is displayed on 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



Using Snippets: Example

To insert a snippet into your code in SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window and drop it 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 such as in the following:

```
SELECT CONCAT(first_name, last_name)
FROM employees;
```

Using SQL*Plus

SQL Worksheet does not support all SQL*Plus statements.

SQL*Plus statements that are not supported by SQL Worksheet are:

- append
- archive
- attribute
- break
- change
- clear

Using SQL*Plus

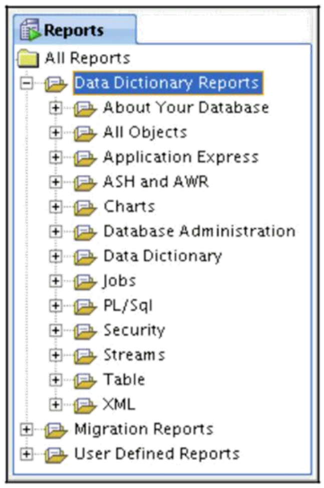
SQL Worksheet supports some SQL*Plus statements. SQL*Plus statements must be interpreted by the SQL Worksheet before being passed to the database; any SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database. For example, some of the SQL*Plus statements that are not supported by SQL Worksheet are listed in the slide. For the complete list of SQL*Plus statements that are supported and not supported by SQL Worksheet, refer to SQL Developer online Help.

Database Reporting

SQL Developer provides you with a number of predefined reports about your database and objects.

The reports are organized into categories.

You can create your own customized reports too.



ORACLE

Database Reporting

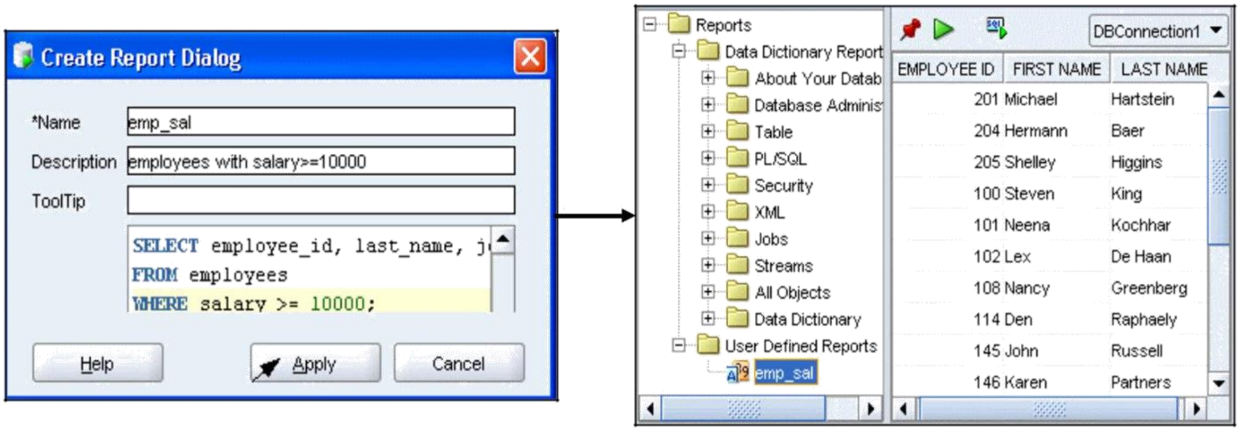
SQL Developer provides many reports about the database and its objects. These reports are grouped into the following categories:

- About Your Database reports
- Object reports
- Application Express reports
- Charts
- Database Administration reports
- Data Dictionary reports
- Jobs reports
- PL/SQL reports
- Security reports
- Streams reports
- Table reports
- XML reports

To display a report, click the Reports tabbed page and then select the report type. You can also create your own user-defined reports.

Creating a User-Defined Report

Create and save user-defined reports for repeated use.



Creating a User-Defined Report

User-defined reports are any reports that are created by SQL Developer users. To create a user-defined report, perform the following steps:

Right-click the **User Defined Reports** node under Reports, and select **Add Report**.

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.

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

Summary

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.

Note: A bolded number or letter refers to an entire lesson or appendix.