Oracle Database 11*g*: SQL Fundamentals II

Electronic Presentation

D49994GC20

Edition 2.0

September 2009

ORACLE®

Authors

Chaitanya Koratamaddi

Brian Pottle

Tulika Srivastava

Technical Contributors and Reviewers

Claire Bennett
Ken Cooper
Yanti Chang
Laszlo Czinkoczki
Burt Demchick
Gerlinde Frenzen
Joel Goodman
Laura Garza
Richard Green
Nancy Greenberg
Akira Kinutani
Wendy Lo
Isabelle Marchand
Timothy Mcglue
Alan Paulson
Srinivas Putrevu
Bryan Roberts
Clinton Shaffer
Abhishek Singh
Jenny Tsai Smith
James Spiller
Lori Tritz
Lex van der Werff
Marcie Young

Editors

Amitha Narayan

Daniel Milne

Graphic Designer

Satish Bettegowda

Publisher

Veena Narasimhan

I

Introduction

Lesson Objectives

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

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

Lesson Agenda

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

Course Objectives

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

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

Course Prerequisites

The *Oracle Database 11g: SQL Fundamentals I* course is a prerequisite for this course.

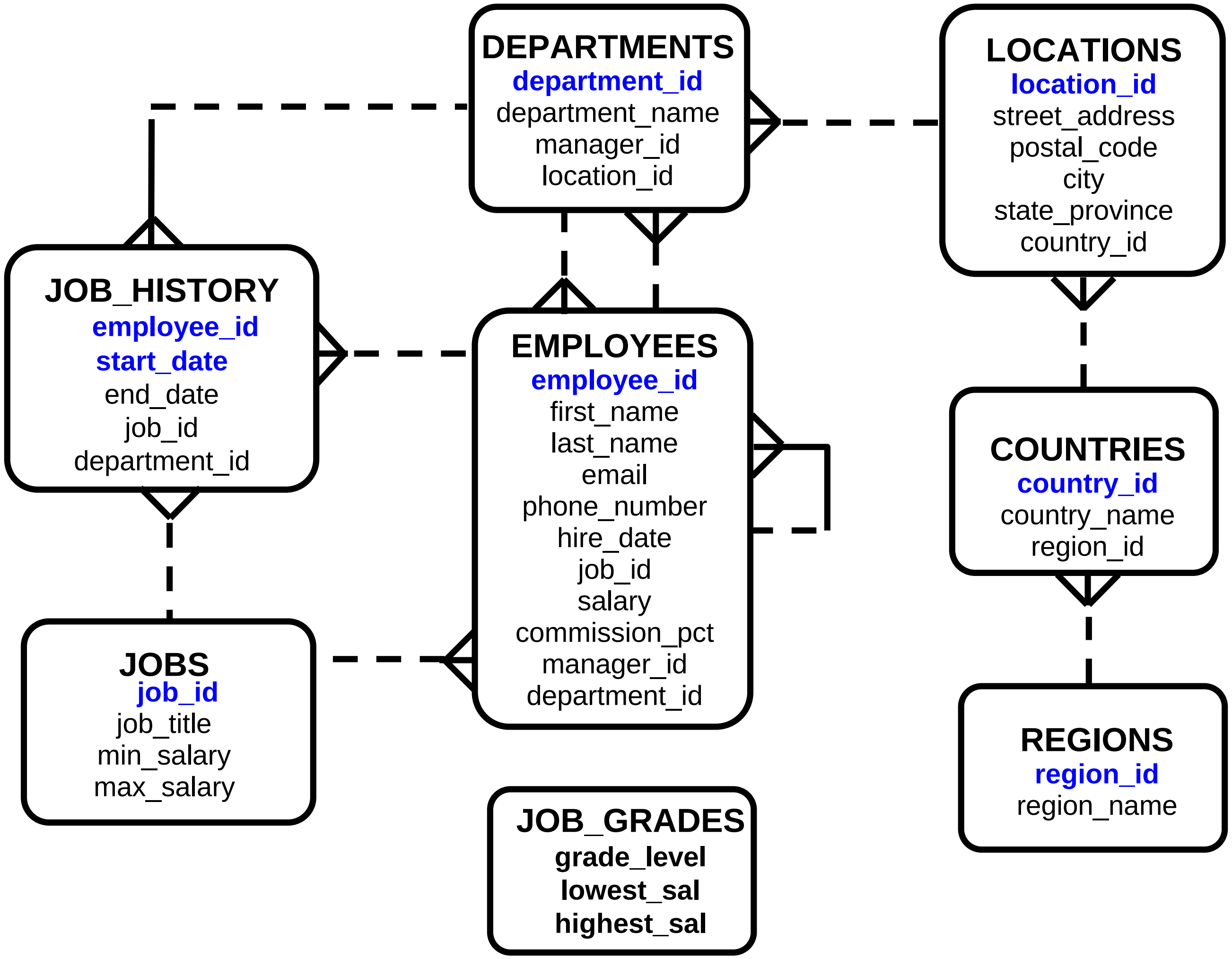
Course Agenda

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

Lesson Agenda

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

Tables Used in This Course



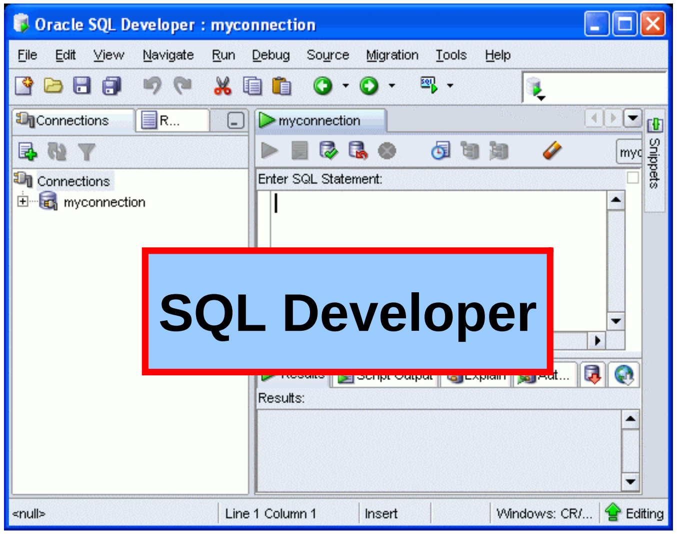
Appendixes Used in This Course

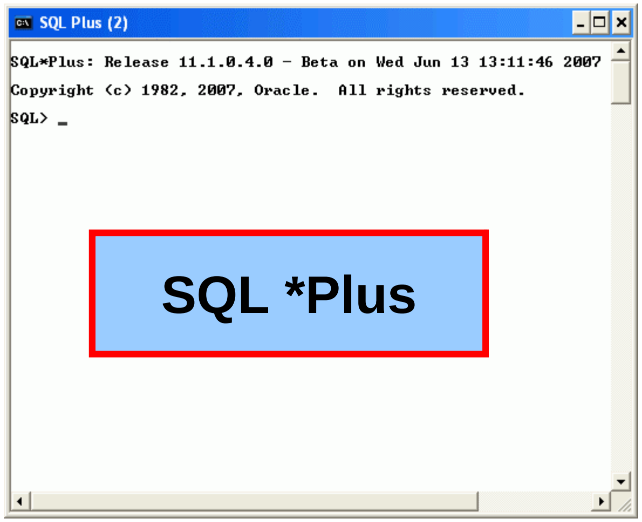
- Appendix A: Practices and Solutions
- Appendix B: Table Descriptions
- Appendix C: Using SQL Developer
- Appendix D: Using SQL*Plus
- Appendix E: Using JDeveloper
- Appendix F: Generating Reports by Grouping Related Data
- Appendix G: Hierarchical Retrieval
- Appendix H: Writing Advanced Scripts
-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

Development Environments

There are two development environments for this course:

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





Lesson Agenda

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

Review of Restricting Data

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

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

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

Review of Sorting Data

- Sort retrieved rows with the `ORDER BY` clause:
  - `ASC`: Ascending order, default
  - `DESC`: Descending order
- The `ORDER BY` clause comes last in the `SELECT` statement:

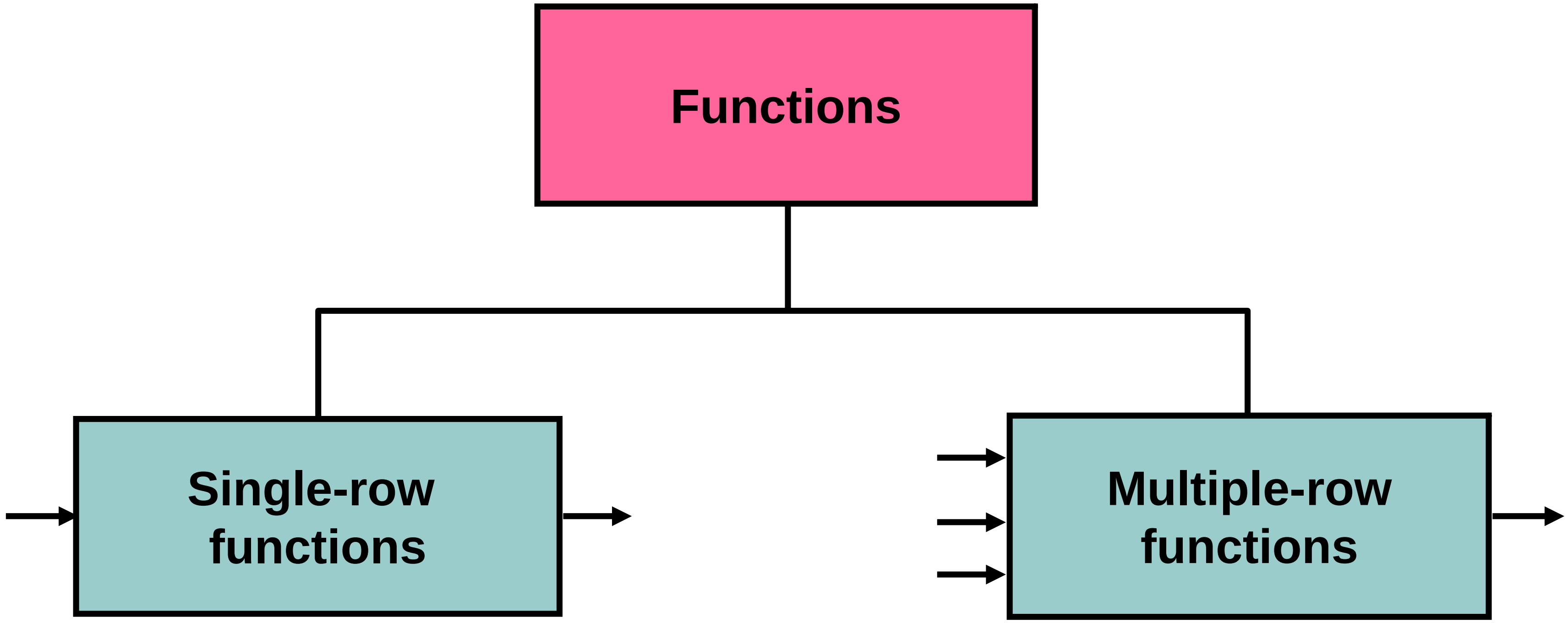
```
SELECT   last_name, job_id, department_id, hire_date
FROM     employees
ORDER BY hire_date ;
```

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

…

ORACLE

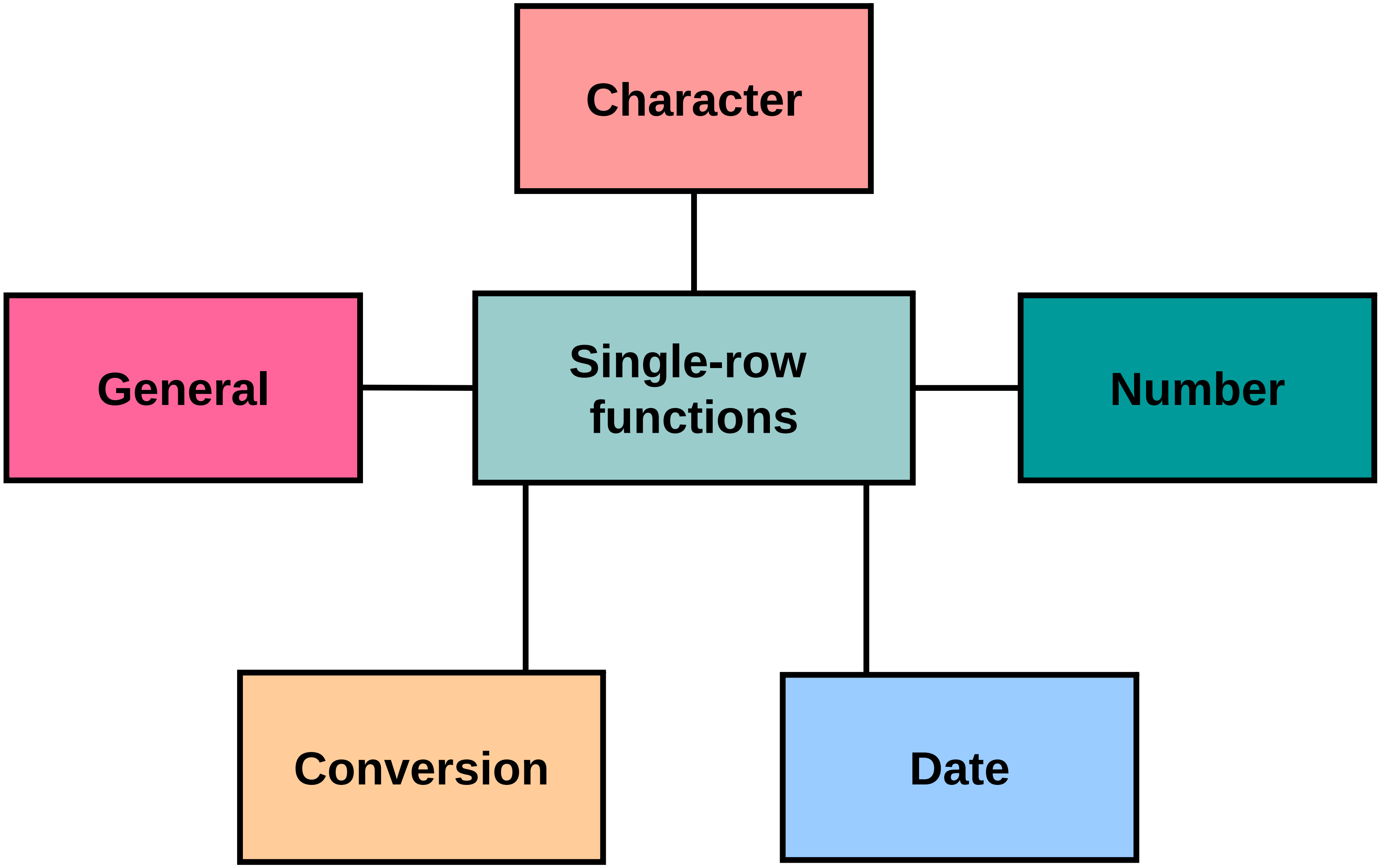
Review of SQL Functions



Return one result per row

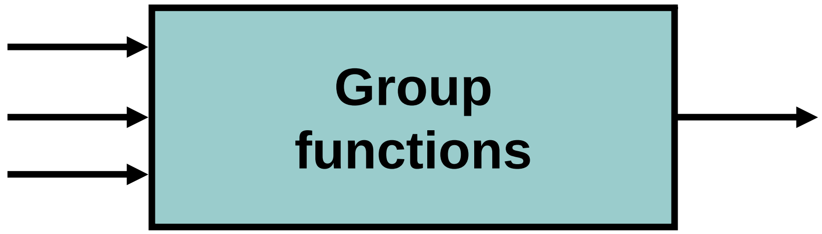
Return one result per set of rows

Review of Single-Row Functions



Review of Types of Group Functions

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



Review of Using Subqueries

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

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

- Types of subqueries:

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

Review of Manipulating Data

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

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

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

Lesson Agenda

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

Oracle Database 11*g* SQL Documentation

- *Oracle Database New Features Guide 11g Release 2 (11.2)*
- *Oracle Database Reference 11g Release 2 (11.2)*
- *Oracle Database SQL Language Reference 11g Release 2 (11.2)*
- *Oracle Database Concepts 11g Release 2 (11.2)*
- *Oracle Database SQL Developer User's Guide Release 1.2*

Additional Resources

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

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

Summary

In this lesson, you should have learned the following:

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

Practice I: Overview

This practice covers the following topics:

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

1

Controlling User Access

Objectives

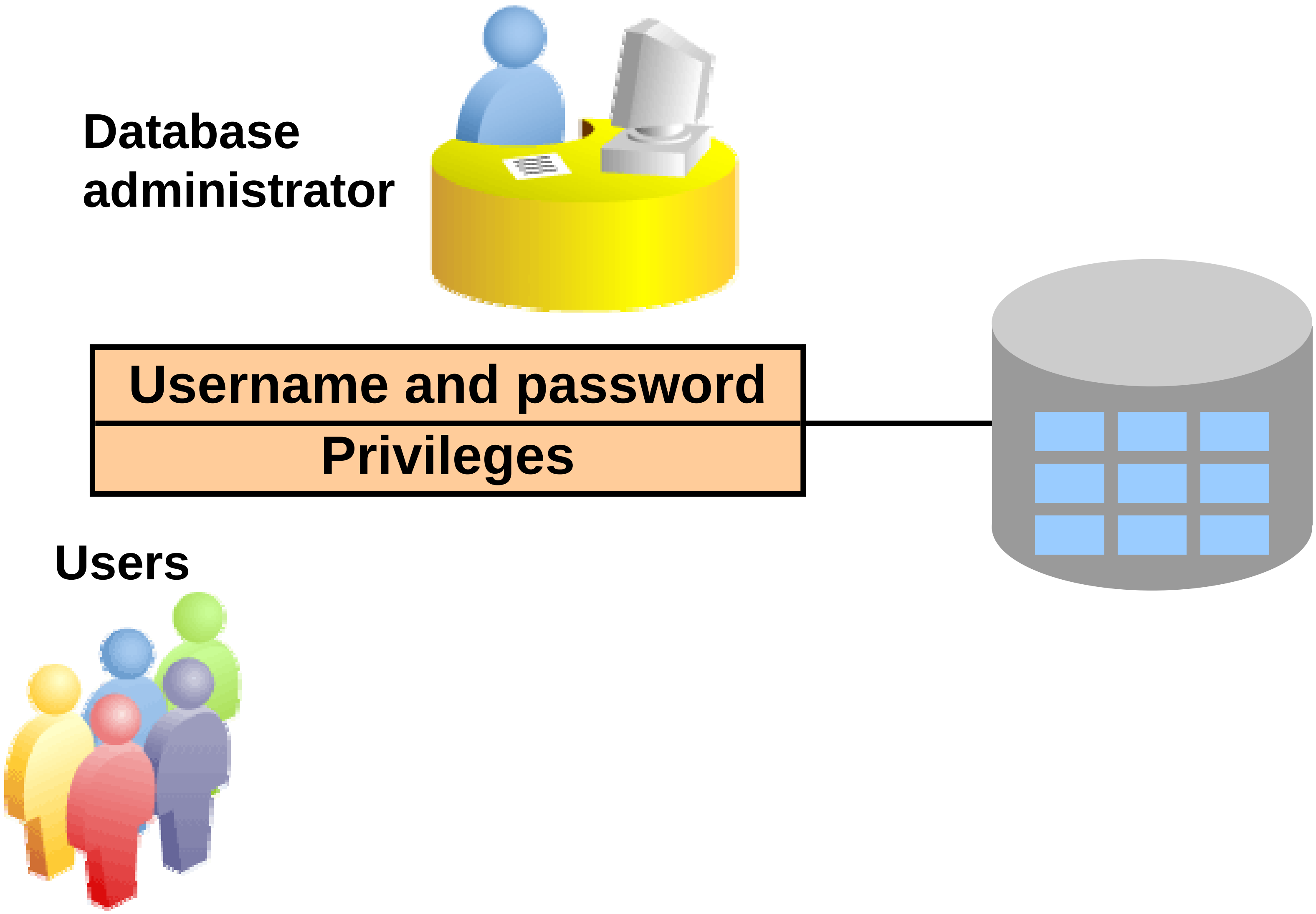
After completing this lesson, you should be able to do the following:

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

Lesson Agenda

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

Controlling User Access



Privileges

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

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

Creating Users

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

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

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

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`

Granting System Privileges

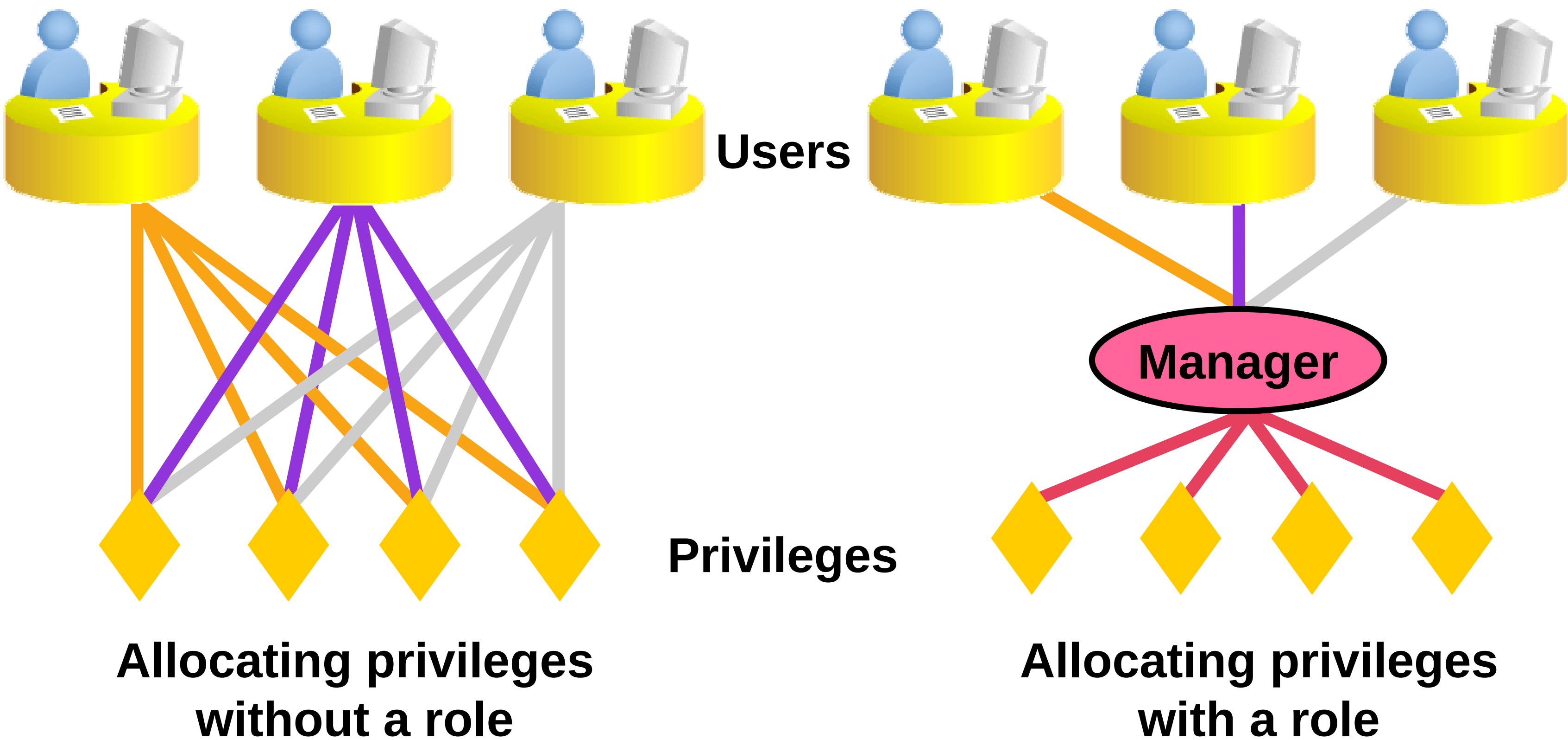
The DBA can grant specific system privileges to a user.

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

Lesson Agenda

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

What Is a Role?



Creating and Granting Privileges to a Role

- Create a role:

```
CREATE ROLE manager;
```

- Grant privileges to a role:

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

- Grant a role to users:

```
GRANT manager TO alice;
```

Changing Your Password

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

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

Lesson Agenda

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

Object Privileges

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

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

- Grant query privileges on the `EMPLOYEES` table:

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

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

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

Passing On Your Privileges

- Give a user authority to pass along privileges:

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

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

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

Confirming Granted Privileges

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

Lesson Agenda

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

Revoking Object Privileges

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

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

Revoking Object Privileges

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

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

Quiz

Which of the following statements are true?

1. After a user creates an object, the user can pass along any of the available object privileges to other users by using the `GRANT` statement.
2. A user can create roles by using the `CREATE ROLE` statement to pass along a collection of system or object privileges to other users.
3. Users can change their own passwords.
4. Users can view the privileges granted to them and those that are granted on their objects.

Summary

In this lesson, you should have learned how to:

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

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

2

Managing Schema Objects

Objectives

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

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

Lesson Agenda

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

ALTER TABLE Statement

Use the ALTER TABLE statement to:

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

ALTER TABLE Statement

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

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

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

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

Adding a Column

- You use the ADD clause to add columns:

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

```
ALTER TABLE dept80 succeeded.
```

- The new column becomes the last column:

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

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 dept80 succeeded.
```

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

Dropping a Column

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

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

```
ALTER TABLE dept80 succeeded.
```

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

SET UNUSED Option

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

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

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

Lesson Agenda

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

Adding a Constraint Syntax

Use the ALTER TABLE statement to:

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

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

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 emp2 succeeded.
```

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

```
ALTER TABLE  succeeded.
```

ON DELETE Clause

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

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

```
ALTER TABLE Emp2 succeeded.
```

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

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

```
ALTER TABLE Emp2 succeeded.
```

Deferring Constraints

Constraints can have the following attributes:

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

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

Deferring constraint on creation

```
SET CONSTRAINTS dept2_id_pk IMMEDIATE
```

Changing a specific constraint attribute

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

Changing all constraints for a session

ORACLE

Difference Between `INITIALLY DEFERRED` and `INITIALLY IMMEDIATE`

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

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

```
create table succeeded.
```

Dropping a Constraint

- Remove the manager constraint from the `EMP2` table:

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

```
ALTER TABLE Emp2 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 dept2 succeeded.
```

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 Emp2 succeeded.
```

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 Emp2 succeeded.
```

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

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.

ORACLE

Cascading Constraints

Example:

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

```
ALTER TABLE Emp2 succeeded.
```

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

```
ALTER TABLE test1 succeeded.
```

Renaming Table Columns and Constraints

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

a

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

```
ALTER TABLE marketing succeeded.
```

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

b

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

```
ALTER TABLE marketing succeeded.
```

Lesson Agenda

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

Overview of Indexes

Indexes are created:

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

`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

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';
```

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 upper_dept_name_idx succeeded.
```

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

DROP TABLE … PURGE

```
DROP TABLE dept80 PURGE;
```

```
DROP TABLE dept80 succeeded.
```

Lesson Agenda

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

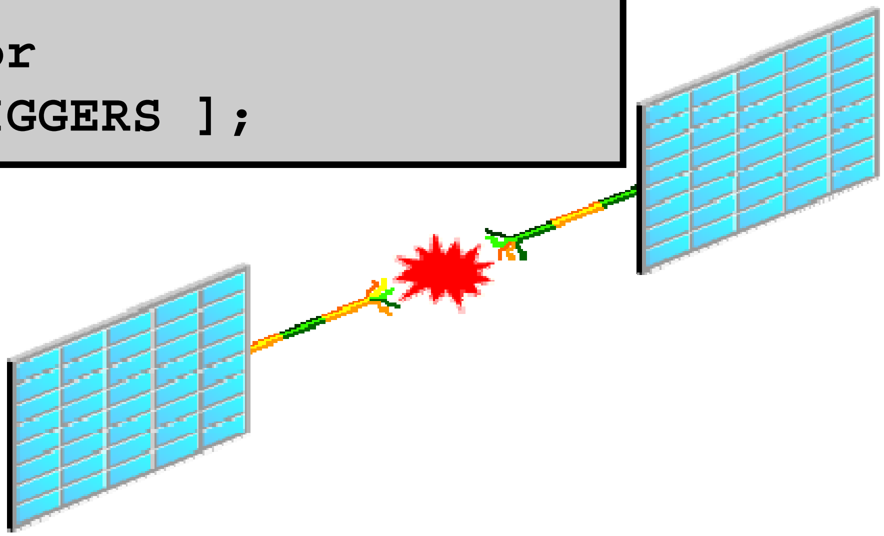
`FLASHBACK TABLE` Statement

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

FLASHBACK TABLE Statement

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

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



Using the `FLASHBACK TABLE` Statement

```
DROP TABLE emp2;
```

```
DROP TABLE emp2 succeeded.
```

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

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

...

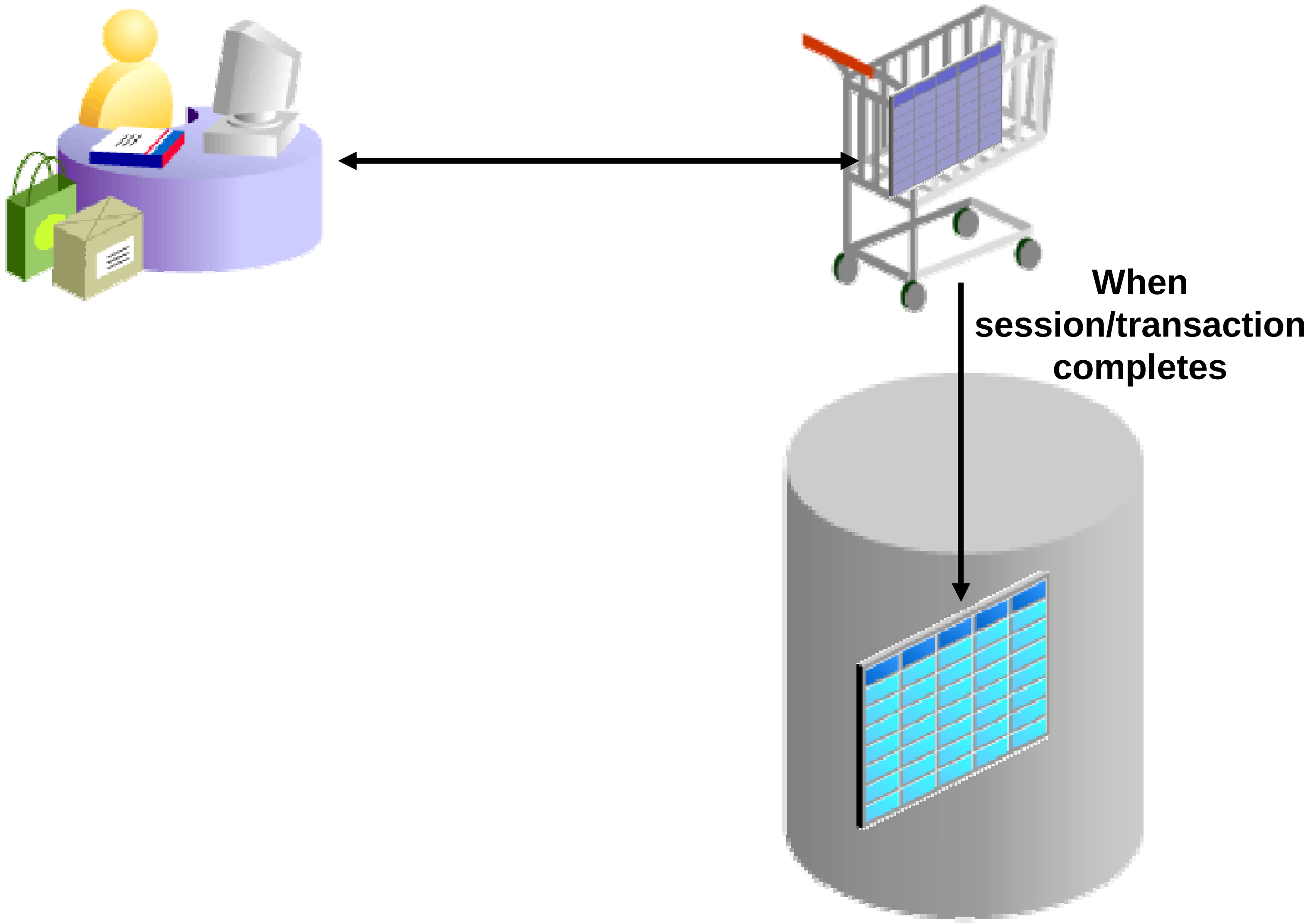
```
FLASHBACK TABLE emp2 TO BEFORE DROP;
```

```
FLASHBACK TABLE succeeded.
```

Lesson Agenda

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

Temporary Tables



Creating a Temporary Table

```
CREATE GLOBAL TEMPORARY TABLE cart
ON COMMIT DELETE ROWS;
```

1

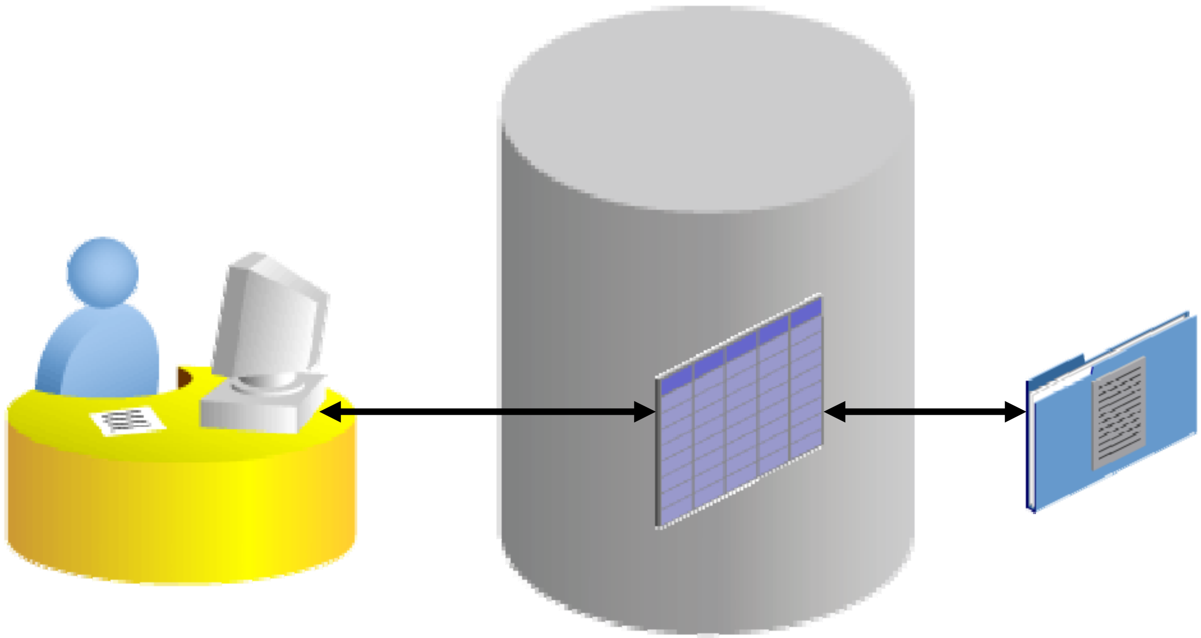
```
CREATE GLOBAL TEMPORARY TABLE today_sales
ON COMMIT PRESERVE ROWS AS
    SELECT * FROM orders
          WHERE order_date = SYSDATE;
```

2

Lesson Agenda

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

External Tables



Creating a Directory for the External Table

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

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

GRANT READ ON DIRECTORY emp_dir TO ora_21;
```

Creating an External Table

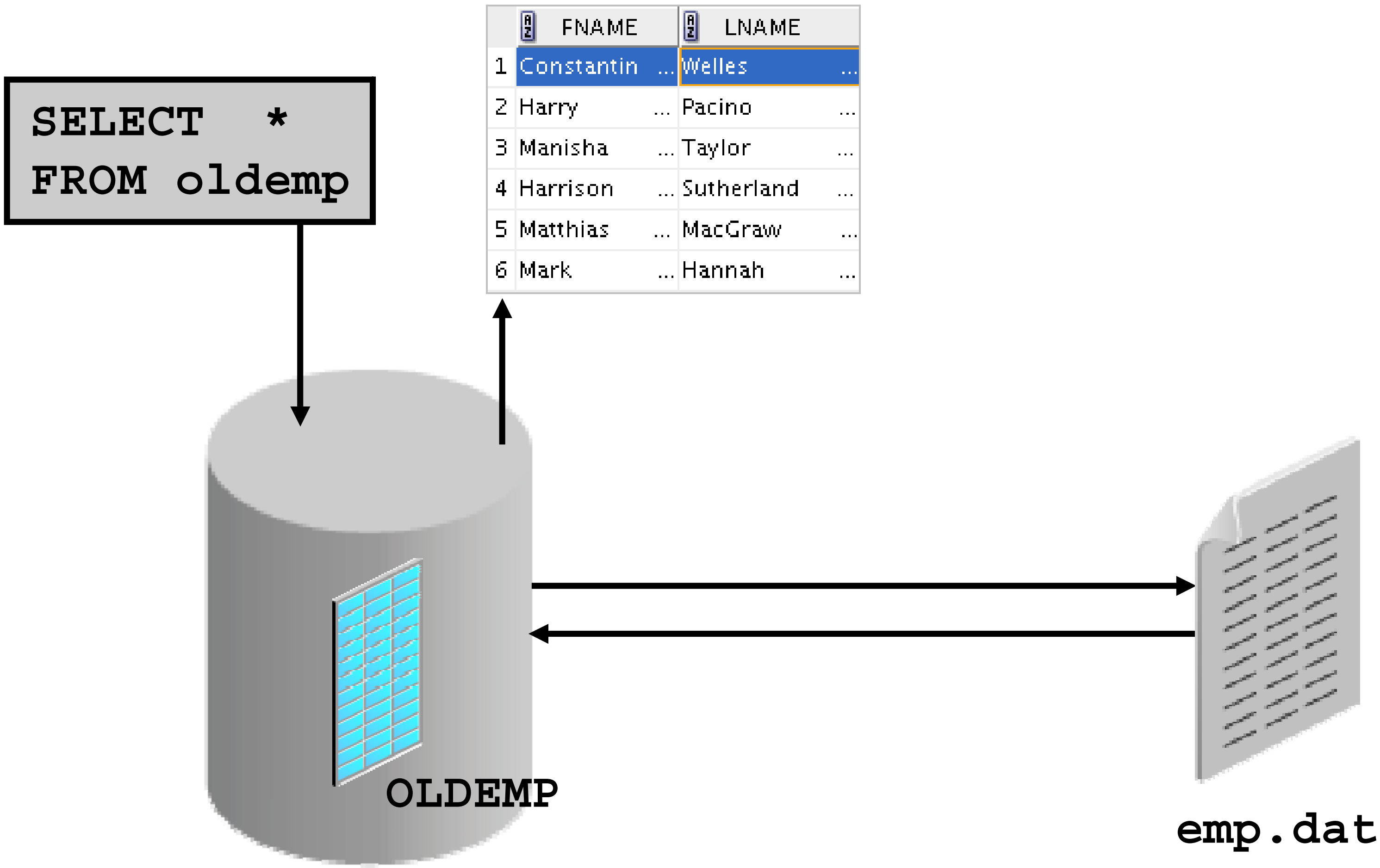
```
CREATE TABLE <table_name>
   ( <col_name> <datatype>, … )
ORGANIZATION EXTERNAL
    (TYPE <access_driver_type>
     DEFAULT DIRECTORY <directory_name>
     ACCESS PARAMETERS
      (… ) )
      LOCATION ('<location_specifier>')
REJECT LIMIT [0 | <number> | UNLIMITED];
```

Creating an External Table 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.
```

Querying External Tables



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

Creating an External Table by Using `ORACLE_DATAPUMP`: Example

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

Quiz

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

1. True
2. False

Quiz

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

1. True
2. False

Summary

In this lesson, you should have learned how to:

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

Practice 2: Overview

This practice covers the following topics:

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

Managing Objects with Data Dictionary Views

Objectives

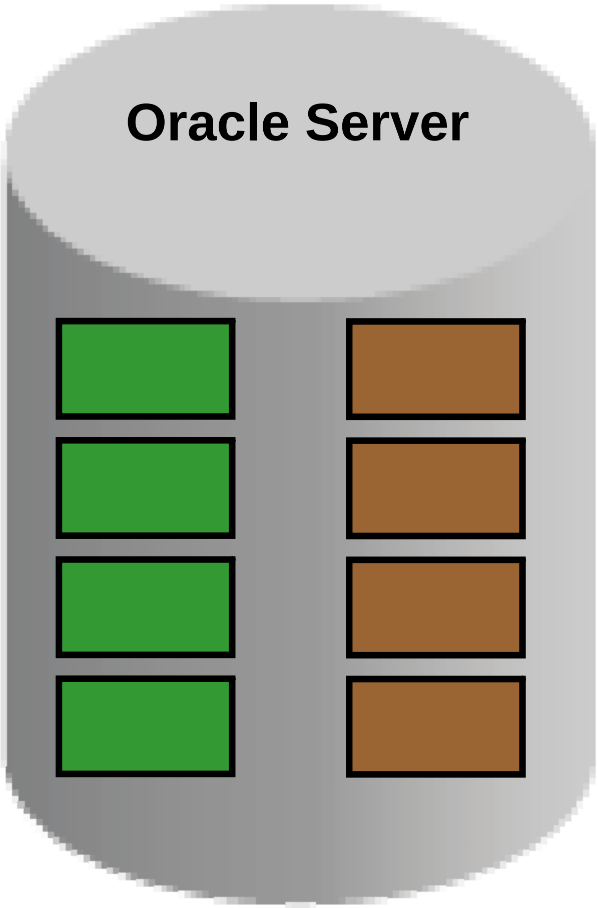
After completing this lesson, you should be able to do the following:

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

Lesson Agenda

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

Data Dictionary



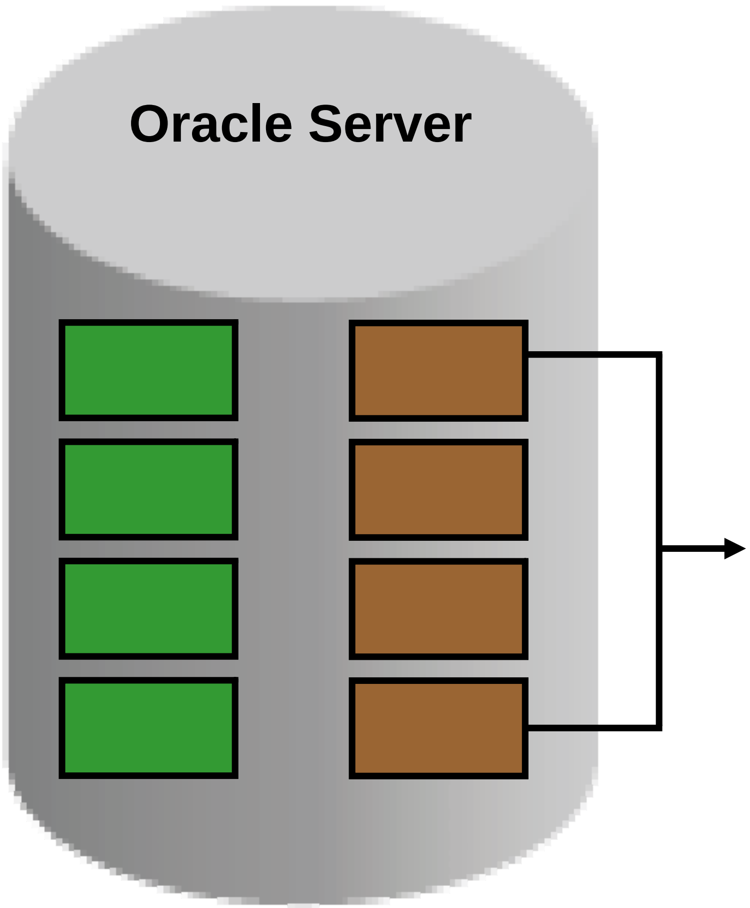
Tables containing business data:

EMPLOYEES
DEPARTMENTS
LOCATIONS
JOB_HISTORY
...

Data dictionary views:

DICTIONARY
USER_OBJECTS
USER_TABLES
USER_TAB_COLUMNS
...

Data Dictionary Structure



Consists of:
- Base tables
- User-accessible views

Data Dictionary Structure

View naming convention:

View Prefix	Purpose
`USER`	User’s view (what is in your schema; what you own)
`ALL`	Expanded user’s view (what you can access)
`DBA`	Database administrator’s view (what is in everyone’s schemas)
`V$`	Performance-related data

How to Use the Dictionary Views

Start with `DICTIONARY`. It contains the names and descriptions of the dictionary tables and views.

```
DESCRIBE DICTIONARY
```

```
Name                           Null     Type
------------------------------ -------- --------------
TABLE_NAME                              VARCHAR2(30)
COMMENTS                                VARCHAR2(4000)

2 rows selected
```

```
SELECT *
FROM   dictionary
WHERE  table_name = 'USER_OBJECTS';
```

	TABLE_NAME	COMMENTS
1	USER_OBJECTS	Objects owned by the user

USER_OBJECTS and ALL_OBJECTS Views

USER_OBJECTS:

- Query USER_OBJECTS to see all the objects that you own.
- Using USER_OBJECTS, you can obtain a listing of all object names and types in your schema, plus the following information:
  – Date created
  – Date of last modification
  – Status (valid or invalid)

ALL_OBJECTS:

- Query ALL_OBJECTS to see all the objects to which you have access.

USER_OBJECTS View

```
SELECT object_name, object_type, created, status
FROM   user_objects
ORDER BY object_type;
```

	OBJECT_NAME	OBJECT_TYPE	CREATED	STATUS
1	LOC_COUNTRY_IX	INDEX	19-MAY-09	VALID

...

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

...

Lesson Agenda

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

Table Information

USER_TABLES:

```
DESCRIBE user_tables
```

```
Name                           Null     Type
------------------------------ -------- ------------
TABLE_NAME                     NOT NULL VARCHAR2(30)
TABLESPACE_NAME                         VARCHAR2(30)
CLUSTER_NAME                            VARCHAR2(30)
IOT_NAME                                VARCHAR2(30)
```

...

```
SELECT table_name
FROM   user_tables;
```

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

...

Column Information

USER_TAB_COLUMNS:

```
DESCRIBE user_tab_columns
```

```
Name                           Null     Type
------------------------------ -------- -------------
TABLE_NAME                     NOT NULL VARCHAR2(30)
COLUMN_NAME                    NOT NULL VARCHAR2(30)
DATA_TYPE                               VARCHAR2(106)
DATA_TYPE_MOD                           VARCHAR2(3)
DATA_TYPE_OWNER                         VARCHAR2(30)
DATA_LENGTH                    NOT NULL NUMBER
DATA_PRECISION                          NUMBER
DATA_SCALE                              NUMBER
NULLABLE                                VARCHAR2(1)
```

...

Column Information

```
SELECT column_name, data_type, data_length,
       data_precision, data_scale, nullable
FROM   user_tab_columns
WHERE  table_name = 'EMPLOYEES';
```

	COLUMN_NAME	DATA_TYPE	DATA_LENGTH	DATA_PRECISION
1	EMPLOYEE_ID	NUMBER	22	6
2	FIRST_NAME	VARCHAR2	20	(null)
3	LAST_NAME	VARCHAR2	25	(null)
4	EMAIL	VARCHAR2	25	(null)
5	PHONE_NUMBER	VARCHAR2	20	(null)
6	HIRE_DATE	DATE	7	(null)
7	JOB_ID	VARCHAR2	10	(null)
8	SALARY	NUMBER	22	8
9	COMMISSION_PCT	NUMBER	22	2
10	MANAGER_ID	NUMBER	22	6
11	DEPARTMENT_ID	NUMBER	22	4

Constraint Information

- `USER_CONSTRAINTS` describes the constraint definitions on your tables.
- `USER_CONS_COLUMNS` describes columns that are owned by you and that are specified in constraints.

```
DESCRIBE user_constraints
```

```
Name                           Null     Type
------------------------------ -------- -----------------------
OWNER                          NOT NULL VARCHAR2(30)
CONSTRAINT_NAME                NOT NULL VARCHAR2(30)
CONSTRAINT_TYPE                         VARCHAR2(1)
TABLE_NAME                     NOT NULL VARCHAR2(30)
SEARCH_CONDITION                        LONG()
R_OWNER                                 VARCHAR2(30)
R_CONSTRAINT_NAME                       VARCHAR2(30)
DELETE_RULE                             VARCHAR2(9)
STATUS                                  VARCHAR2(8)
```

...

USER_CONSTRAINTS: Example

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

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

Querying USER_CONS_COLUMNS

```
DESCRIBE user_cons_columns
```

```
Name                           Null     Type
------------------------------ -------- ---------------
OWNER                          NOT NULL VARCHAR2(30)
CONSTRAINT_NAME                NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
POSITION                                NUMBER
```

```
SELECT constraint_name, column_name
FROM   user_cons_columns
WHERE  table_name = 'EMPLOYEES';
```

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

...

Lesson Agenda

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

ORACLE

View Information

1

```
DESCRIBE user_views
```

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

2

```
SELECT view_name FROM user_views;
```

	VIEW_NAME
1	EMP_DETAILS_VIEW

3

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

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

...

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

Sequence Information

```
DESCRIBE user_sequences
```

```
Name                           Null     Type
------------------------------ -------- -------------
SEQUENCE_NAME                  NOT NULL VARCHAR2(30)
MIN_VALUE                               NUMBER
MAX_VALUE                               NUMBER
INCREMENT_BY                   NOT NULL NUMBER
CYCLE_FLAG                              VARCHAR2(1)
ORDER_FLAG                              VARCHAR2(1)
CACHE_SIZE                     NOT NULL NUMBER
LAST_NUMBER                    NOT NULL NUMBER
```

Confirming Sequences

- Verify your sequence values in the `USER_SEQUENCES` data dictionary table.

```
SELECT    sequence_name, min_value, max_value,
          increment_by, last_number
FROM      user_sequences;
```

	SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
1	DEPARTMENTS_SEQ	1	9990	10	280
2	EMPLOYEES_SEQ	1	999999999999999...	1	207
3	LOCATIONS_SEQ	1	9900	100	3300

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

Index Information

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

```
DESCRIBE user_indexes
```

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

...

`USER_INDEXES`: Examples

a

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

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

b

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

	INDEX_NAME	TABLE_NAME
1	SYS_C0011777	EMP_LIB

Querying USER_IND_COLUMNS

```
DESCRIBE user_ind_columns
```

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

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

	INDEX_NAME	COLUMN_NAME	TABLE_NAME
1	LNAME_IDX	LAST_NAME	EMP_TEST

Synonym Information

```
DESCRIBE user_synonyms
```

```
Name                           Null     Type
------------------------------ -------- --------------
SYNONYM_NAME                   NOT NULL VARCHAR2(30)
TABLE_OWNER                             VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
DB_LINK                                 VARCHAR2(128)
```

```
SELECT *
FROM   user_synonyms;
```

	SYNONYM_NAME	TABLE_OWNER	TABLE_NAME	DB_LINK
1	TEAM2	ORA22	DEPARTMENTS	(null)

Lesson Agenda

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

Adding Comments to a Table

- You can add comments to a table or column by using the `COMMENT` statement:

```
COMMENT ON TABLE employees
IS 'Employee Information';
```

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

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

Quiz

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

1. Definitions of all the schema objects in the database
2. Default values for the columns
3. Integrity constraint information
4. Privileges and roles that each user has been granted
5. All of the above

Summary

In this lesson, you should have learned how to find information about your objects through the following dictionary views:

- DICTIONARY
- USER_OBJECTS
- USER_TABLES
- USER_TAB_COLUMNS
- USER_CONSTRAINTS
- USER_CONS_COLUMNS
- USER_VIEWS
- USER_SEQUENCES
- USER_INDEXES
- USER_SYNONYMS

Practice 3: Overview

This practice covers the following topics:

- Querying the dictionary views for table and column information
- Querying the dictionary views for constraint information
- Querying the dictionary views for view information
- Querying the dictionary views for sequence information
- Querying the dictionary views for synonym information
- Querying the dictionary views for index information
- Adding a comment to a table and querying the dictionary views for comment information

4

Manipulating Large Data Sets

Objectives

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

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

Lesson Agenda

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

Using Subqueries to Manipulate Data

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

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

Retrieving Data by Using a Subquery as Source

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

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

Inserting by Using a Subquery as a Target

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

```
1 rows inserted
```

Inserting by Using a Subquery as a Target

Verify the results.

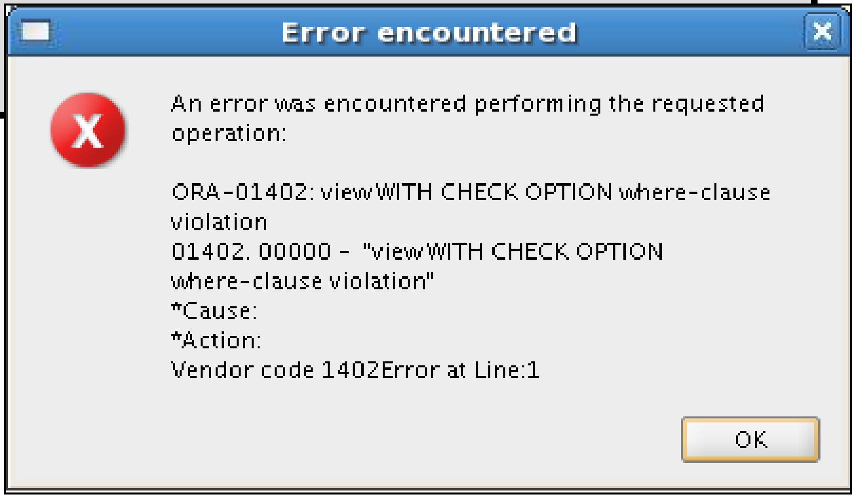
```
SELECT location_id, city, country_id
FROM   loc
```

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

Using the `WITH CHECK OPTION` Keyword on DML Statements

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

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



Lesson Agenda

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

Overview of the Explicit Default Feature

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

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;
```

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%';
```

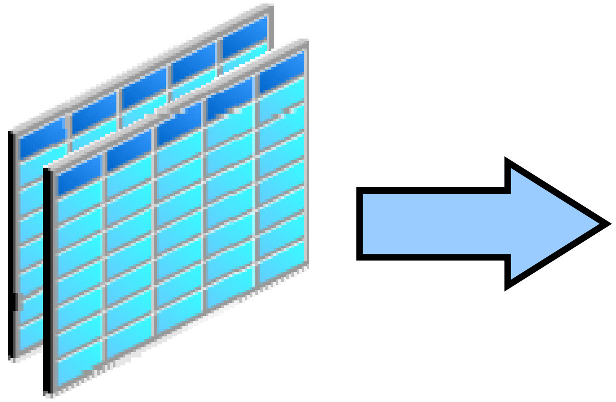
```
33 rows inserted
```

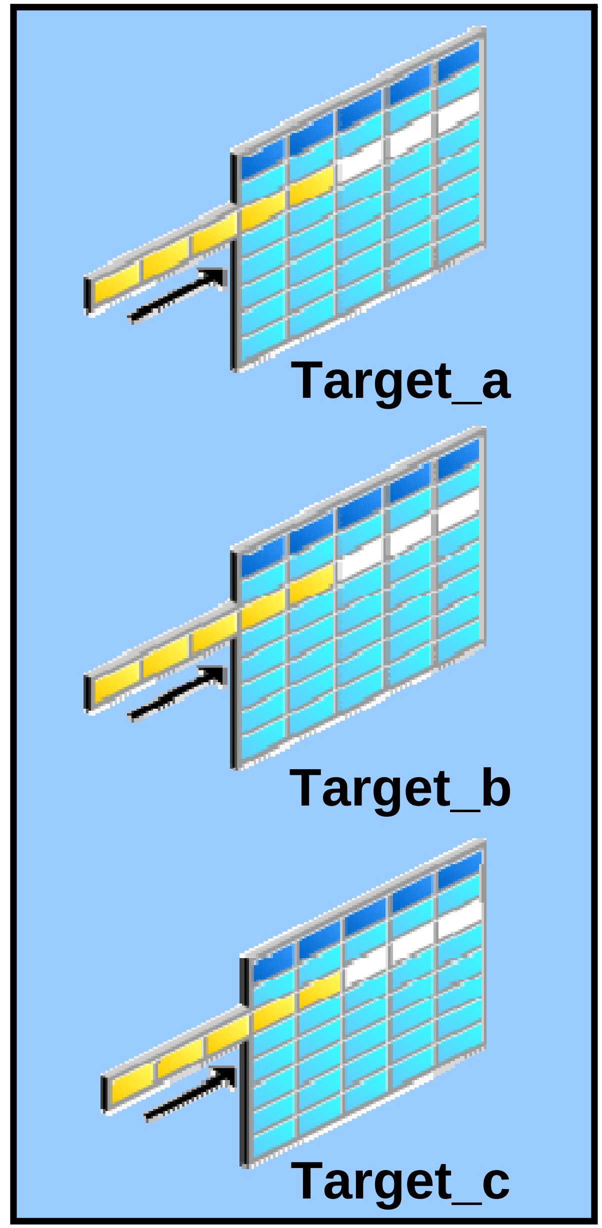
- Do not use the `VALUES` clause.
- Match the number of columns in the `INSERT` clause with that in the subquery.

Lesson Agenda

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

Overview of Multitable `INSERT` Statements






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

Subquery

Overview of Multitable `INSERT` Statements

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

Types of Multitable `INSERT` Statements

The different types of multitable `INSERT` statements are:

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

Multitable INSERT Statements

- Syntax for multitable `INSERT`:

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

- `conditional_insert_clause`:

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

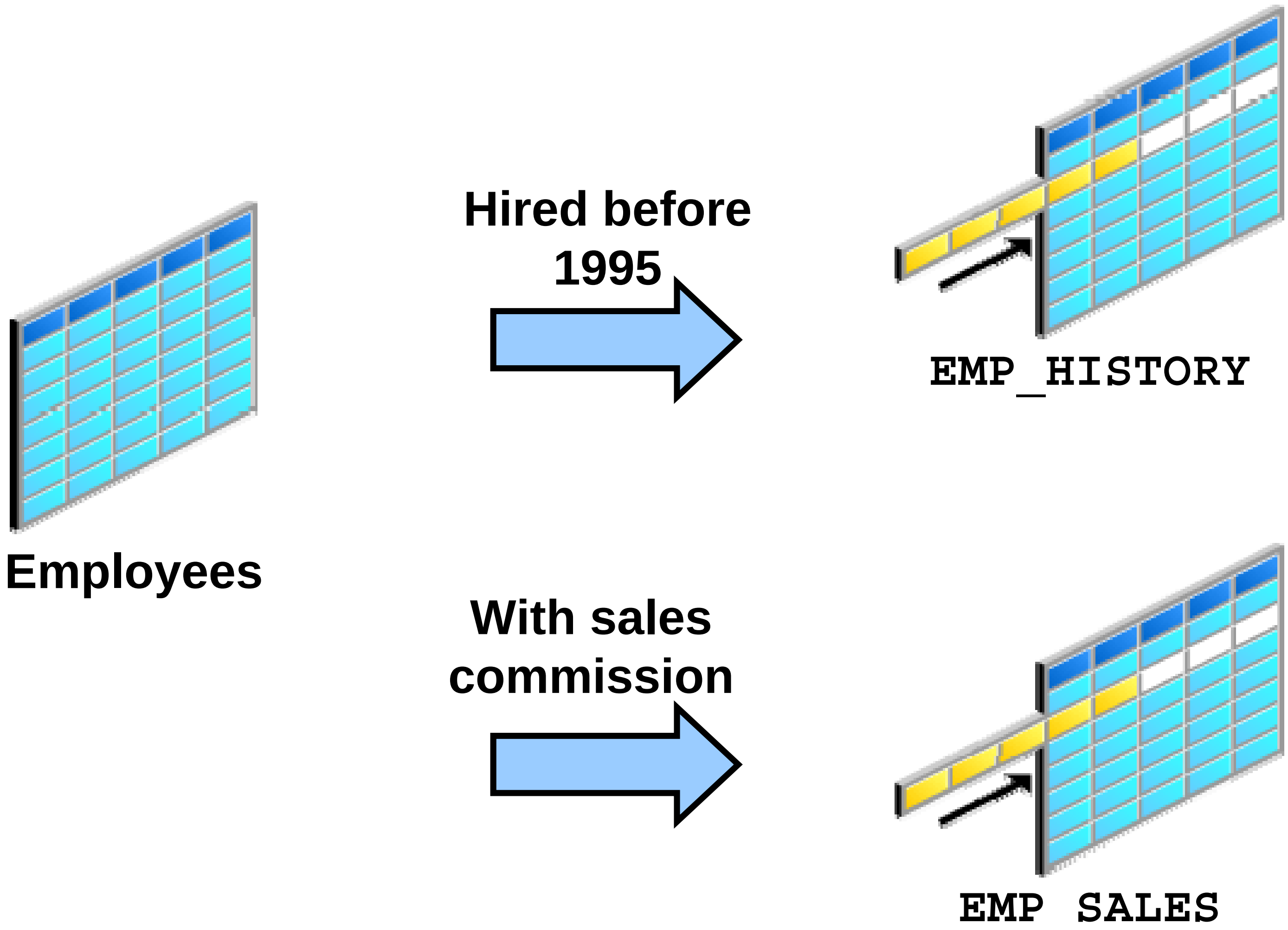
Unconditional INSERT ALL

- 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
```

Conditional `INSERT ALL`: Example

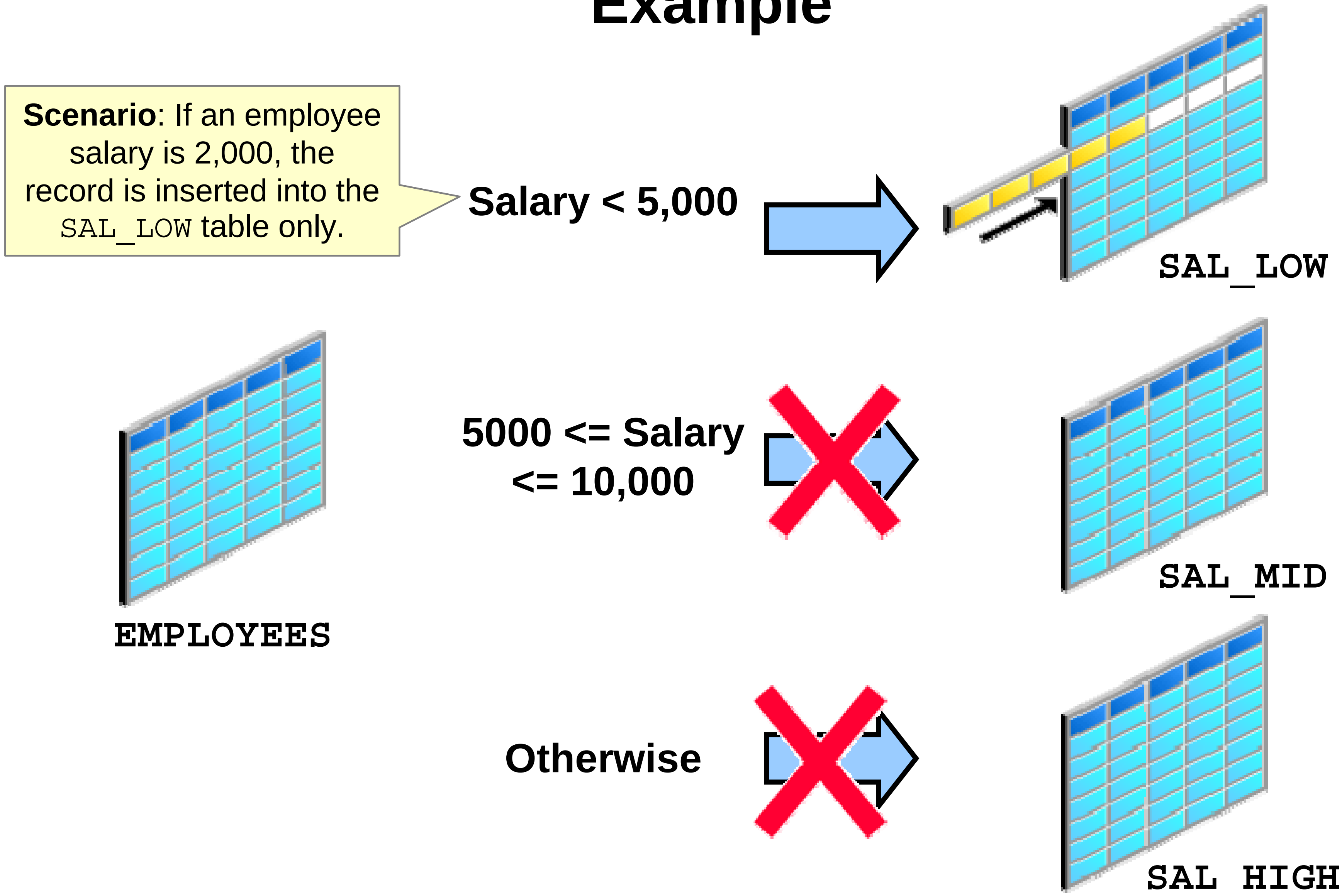


Conditional INSERT ALL

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

```
48 rows inserted
```

Conditional `INSERT FIRST`: Example



Conditional INSERT FIRST

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

```
107 rows inserted
```

Pivoting `INSERT`

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

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

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

Pivoting 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
```

Lesson Agenda

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

`MERGE` Statement

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

MERGE Statement Syntax

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

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

Merging Rows: Example

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

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

Merging Rows: Example

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

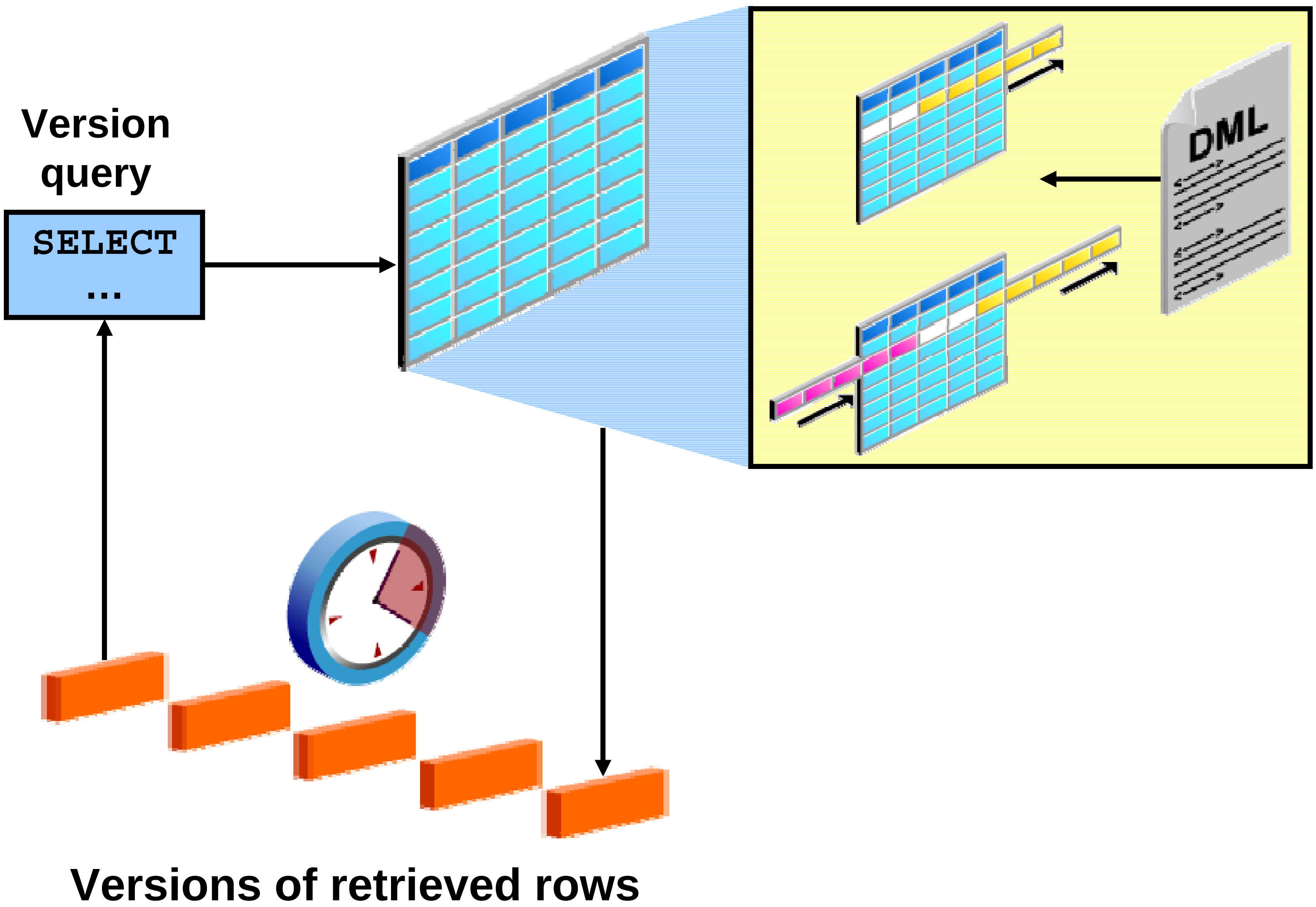
```
MERGE INTO copy_emp3 c
USING (SELECT * FROM EMPLOYEES ) e
ON (c.employee_id = e.employee_id)
WHEN MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
...
DELETE WHERE (E.COMMISSION_PCT IS NOT NULL)
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, ...
```

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

Lesson Agenda

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

Tracking Changes in Data



Example of the Flashback Version Query

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

1

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

COMMIT;
```

2

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

3

1

	SALARY
1	4200

3

	SALARY
1	5460
2	4200

VERSIONS BETWEEN Clause

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

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

Quiz

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

1. True
2. False

Summary

In this lesson, you should have learned how to:

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

Practice 4: Overview

This practice covers the following topics:

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

5

Managing Data in Different Time Zones

Objectives

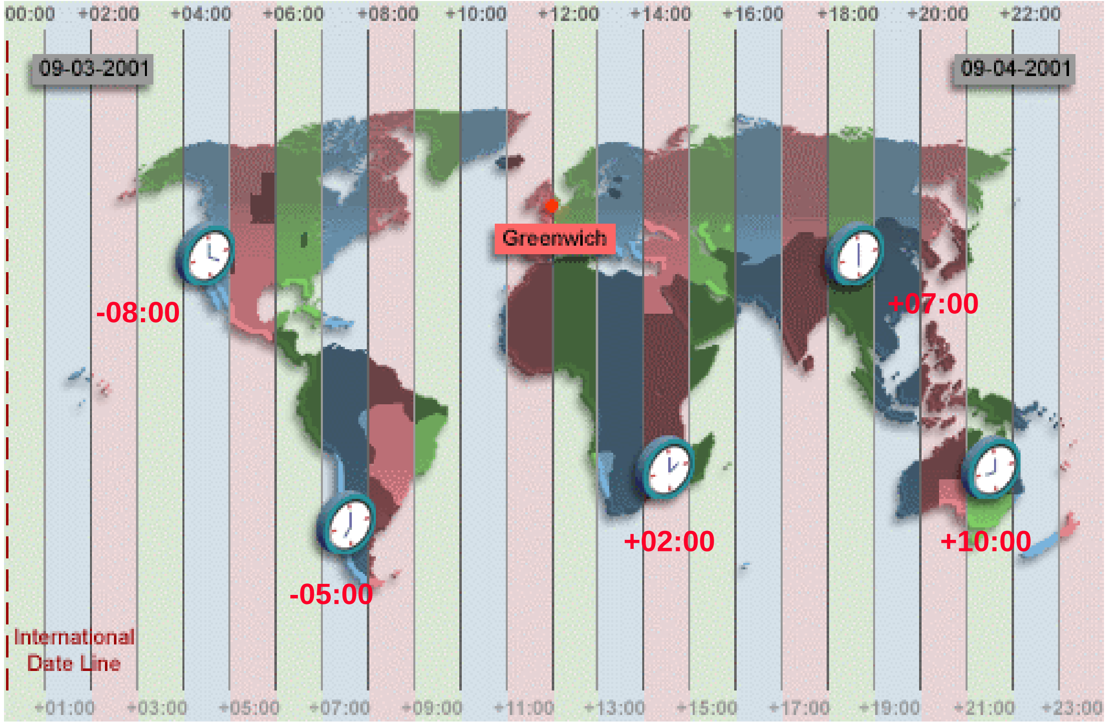
After completing this lesson, you should be able to do the following:

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

Lesson Agenda

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

Time Zones



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

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';
```

CURRENT_DATE, CURRENT_TIMESTAMP, and LOCALTIMESTAMP

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

Comparing Date and Time in a Session's Time Zone

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

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

SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;          1


SELECT SESSIONTIMEZONE, CURRENT_TIMESTAMP FROM DUAL;     2


SELECT SESSIONTIMEZONE, LOCALTIMESTAMP FROM DUAL;        3
```

Comparing Date and Time in a Session's Time Zone

Results of queries:

ALTER SESSION succeeded.

1

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

2

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

3

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

DBTIMEZONE and SESSIONTIMEZONE

- Display the value of the database time zone:

```
SELECT DBTIMEZONE FROM DUAL;
```

	DBTIMEZONE
1	+00:00

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

```
SELECT SESSIONTIMEZONE FROM DUAL;
```

	SESSIONTIMEZONE
1	-05:00

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` 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

Difference Between `DATE` and `TIMESTAMP`

A

```
-- when hire_date is
of type DATE

SELECT hire_date
FROM employees;
```

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

B

```
ALTER TABLE employees
MODIFY hire_date TIMESTAMP;

SELECT hire_date
FROM employees;
```

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

...

Comparing `TIMESTAMP` Data Types

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

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

```
SELECT * FROM web_orders;
```

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

Lesson Agenda

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

`INTERVAL` Data Types

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

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

ORACLE

`INTERVAL` Fields

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

INTERVAL YEAR TO MONTH: 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 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

Lesson Agenda

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

EXTRACT

- Display the YEAR component from the SYSDATE.

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

	EXTRACT(YEARFROMSYSDATE)
1	2009

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

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

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

TZ_OFFSET

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

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

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

FROM_TZ

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

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

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

TO_TIMESTAMP

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

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

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

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

TO_DSINTERVAL

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

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

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

...

Daylight Saving Time

- First Sunday in April
  - Time jumps from 01:59:59 AM to 03:00:00 AM.
  - Values from 02:00:00 AM to 02:59:59 AM are not valid.
- Last Sunday in October
  - Time jumps from 02:00:00 AM to 01:00:01 AM.
  - Values from 01:00:01 AM to 02:00:00 AM are ambiguous because they are visited twice.

Quiz

The `TIME_ZONE` session parameter may be set to:

1. A relative offset
2. Database time zone
3. OS local time zone
4. A named region

Summary

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

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

Practice 5: Overview

This practice covers using the datetime functions.

Retrieving Data by Using Subqueries

Objectives

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

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

ORACLE

Lesson Agenda

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

Multiple-Column Subqueries

Main query

```
WHERE (MANAGER_ID, DEPARTMENT_ID)  IN
```

Subquery

100	90
102	60
124	50

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

ORACLE

Column Comparisons

Multiple-column comparisons involving subqueries can be:

- Nonpairwise comparisons
- Pairwise comparisons

Pairwise Comparison Subquery

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

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

Nonpairwise Comparison Subquery

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

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

Lesson Agenda

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

Scalar Subquery Expressions

- A scalar subquery expression is a subquery that returns exactly one column value from one row.
- Scalar subqueries can be used in:
  - The condition and expression part of `DECODE` and `CASE`
  - All clauses of `SELECT` except `GROUP BY`
  - The `SET` clause and `WHERE` clause of an `UPDATE` statement

Scalar Subqueries: Examples

- Scalar subqueries in `CASE` expressions:

```
SELECT employee_id, last_name,
       (CASE
        WHEN department_id =                 20
                 (SELECT department_id
                  FROM departments
                      WHERE location_id = 1800)
         THEN 'Canada' ELSE 'USA' END) location
FROM   employees;
```

- Scalar subqueries in the `ORDER BY` clause:

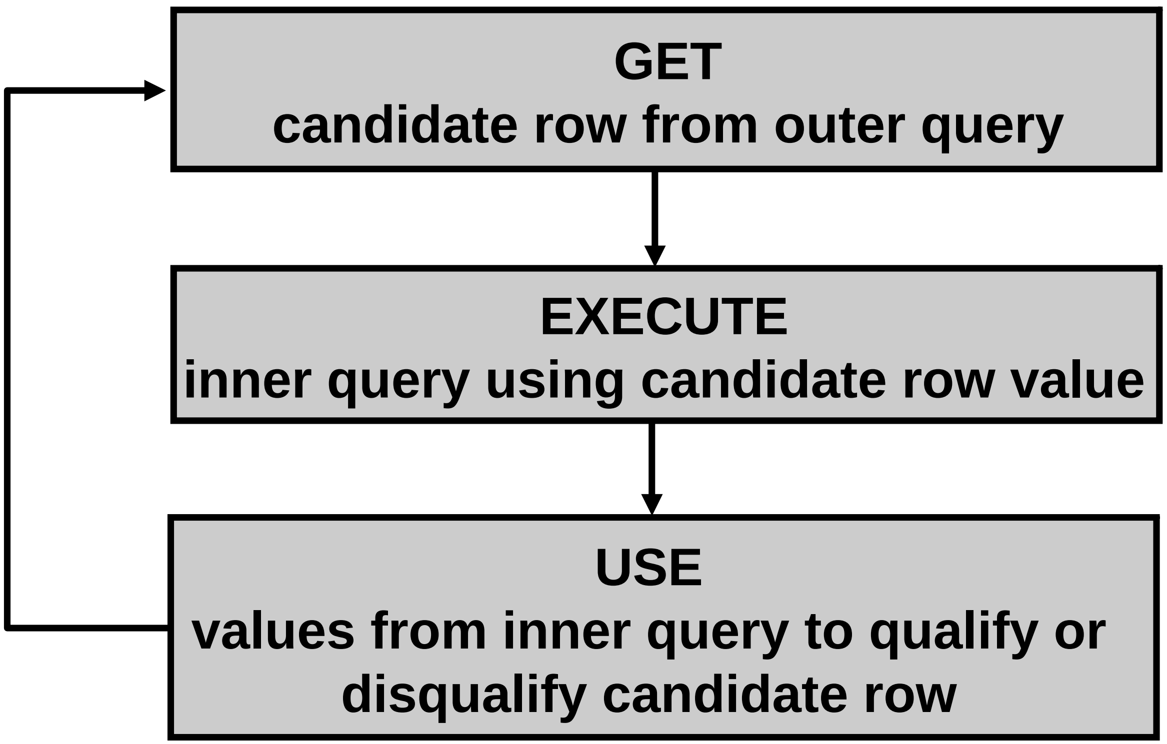
```
SELECT   employee_id, last_name
FROM     employees e
ORDER BY (SELECT department_name
           FROM departments d
            WHERE e.department_id = d.department_id);
```

Lesson Agenda

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

Correlated Subqueries

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



Correlated Subqueries

The subquery references a column from a table in the parent query.

```
SELECT column1, column2, ...
FROM   table1   Outer_table
WHERE  column1 operator
                      (SELECT  column1, column2
                       FROM    table2
                       WHERE   expr1 =
                            Outer_table.expr2);
```

Using Correlated Subqueries

Find all employees who earn more than the average salary in their department.

```
SELECT last_name, salary, department_id
FROM   employees outer_table
WHERE  salary >
                (SELECT AVG(salary)
                 FROM   employees inner_table
                 WHERE inner_table.department_id =
                 outer_table.department_id);
```

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

Using Correlated Subqueries

Display details of those employees who have changed jobs at least twice.

```
SELECT e.employee_id, last_name,e.job_id
FROM   employees e
WHERE  2 <= (SELECT COUNT(*)
             FROM   job_history
             WHERE  employee_id = e.employee_id);
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID
1	200	Whalen	AD_ASST
2	101	Kochhar	AD_VP
3	176	Taylor	SA_REP

Lesson Agenda

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

Using the `EXISTS` Operator

- The `EXISTS` operator tests for existence of rows in the results set of the subquery.
- If a subquery row value is found:
  - The search does not continue in the inner query
  - The condition is flagged `TRUE`
- If a subquery row value is not found:
  - The condition is flagged `FALSE`
  - The search continues in the inner query

Using the EXISTS Operator

```sql
SELECT employee_id, last_name, job_id, department_id
FROM   employees outer
WHERE  EXISTS ( SELECT 'X'
                FROM   employees
                WHERE  manager_id =
                       outer.employee_id);
```

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	201	Hartstein	MK_MAN	20
2	205	Higgins	AC_MGR	110
3	100	King	AD_PRES	90
4	101	Kochhar	AD_VP	90
5	102	De Haan	AD_VP	90
6	103	Hunold	IT_PROG	60
7	108	Greenberg	FI_MGR	100
8	114	Raphaely	PU_MAN	30

Find All Departments That Do Not Have Any Employees

```
SELECT department_id, department_name
FROM departments d
WHERE NOT EXISTS (SELECT 'X'
                  FROM   employees
                  WHERE  department_id = d.department_id);
```

	DEPARTMENT_ID	DEPARTMENT_NAME
1	120	Treasury
2	130	Corporate Tax
3	140	Control And Credit
4	150	Shareholder Services
5	160	Benefits
6	170	Manufacturing
7	180	Construction

...

All Rows Fetched: 16

ORACLE

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);
```

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);
```

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);
```

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

Lesson Agenda

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

`WITH` Clause

- Using the `WITH` clause, you can use the same query block in a `SELECT` statement when it occurs more than once within a complex query.
- The `WITH` clause retrieves the results of a query block and stores it in the user's temporary tablespace.
- The `WITH` clause may improve performance.

`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.

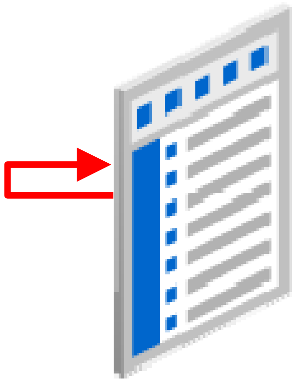
`WITH` Clause: Example

```
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;
```

Recursive `WITH` Clause

The Recursive `WITH` clause

- Enables formulation of recursive queries.
- Creates query with a name, called the Recursive `WITH` element name
- Contains two types of query blocks member: anchor and a recursive
- Is ANSI-compatible



Recursive `WITH` Clause: Example

FLIGHTS Table

	SOURCE	DESTIN	FLIGHT_TIME
1	San Jose	Los Angeles	1.3
2	New York	Boston	1.1
3	Los Angeles	New York	5.8

1

```
WITH Reachable_From (Source, Destin, TotalFlightTime) AS
 (
     SELECT Source, Destin, Flight_time
     FROM Flights
   UNION ALL
     SELECT incoming.Source, outgoing.Destin,
            incoming.TotalFlightTime+outgoing.Flight_time
     FROM Reachable_From incoming, Flights outgoing
     WHERE incoming.Destin = outgoing.Source
)
SELECT Source, Destin, TotalFlightTime
FROM Reachable_From;
```

2

	SOURCE	DESTIN	TOTALFLIGHTTIME
1	San Jose	Los Angeles	1.3
2	New York	Boston	1.1
3	Los Angeles	New York	5.8
4	San Jose	New York	7.1
5	Los Angeles	Boston	6.9
6	San Jose	Boston	8.2

3

Quiz

With a correlated subquery, the inner `SELECT` statement drives the outer `SELECT` statement.

1. True
2. False

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

- 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

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

7

Regular Expression Support

Objectives

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

- List the benefits of using regular expressions
- Use regular expressions to search for, match, and replace strings

Lesson Agenda

- Introduction to regular expressions
- Using metacharacters with regular expressions
- Using the regular expressions functions:
  - `REGEXP_LIKE`
  - `REGEXP_REPLACE`
  - `REGEXP_INSTR`
  - `REGEXP_SUBSTR`
- Accessing subexpressions
- Using the `REGEXP_COUNT` function
- Regular expressions and check constraints

What Are Regular Expressions?

- You use regular expressions to search for (and manipulate) simple and complex patterns in string data by using standard syntax conventions.
- You use a set of SQL functions and conditions to search for and manipulate strings in SQL and PL/SQL.
- You specify a regular expression by using:
  - Metacharacters, which are operators that specify the search algorithms
  - Literals, which are the characters for which you are searching

ORACLE

Benefits of Using Regular Expressions

Regular expressions enable you to implement complex match logic in the database with the following benefits:

- By centralizing match logic in Oracle Database, you avoid intensive string processing of SQL results sets by middle-tier applications.
- Using server-side regular expressions to enforce constraints, you eliminate the need to code data validation logic on the client.
- The built-in SQL and PL/SQL regular expression functions and conditions make string manipulations more powerful and easier than in previous releases of Oracle Database 11*g*.

Using the Regular Expressions Functions and Conditions in SQL and PL/SQL

Function or Condition Name	Description
`REGEXP_LIKE`	Is similar to the `LIKE` operator, but performs regular expression matching instead of simple pattern matching (condition)
`REGEXP_REPLACE`	Searches for a regular expression pattern and replaces it with a replacement string
`REGEXP_INSTR`	Searches a string for a regular expression pattern and returns the position where the match is found
`REGEXP_SUBSTR`	Searches for a regular expression pattern within a given string and extracts the matched substring
`REGEXP_COUNT`	Returns the number of times a pattern match is found in an input sting

Lesson Agenda

- Introduction to regular expressions
- Using metacharacters with regular expressions
- Using the regular expressions functions:
  - REGEXP_LIKE
  - REGEXP_REPLACE
  - REGEXP_INSTR
  - REGEXP_SUBSTR
- Accessing subexpressions
- Using the REGEXP_COUNT function

What Are Metacharacters?

- Metacharacters are special characters that have a special meaning such as a wildcard, a repeating character, a nonmatching character, or a range of characters.
- You can use several predefined metacharacter symbols in the pattern matching.
- For example, the `^(f|ht)tps?:$` regular expression searches for the following from the beginning of the string:
  – The literals `f` or `ht`
  – The `t` literal
  – The `p` literal, optionally followed by the `s` literal
  – The colon "`:`" literal at the end of the string

Using Metacharacters with Regular Expressions

Syntax	Description
`.`	Matches any character in the supported character set, except NULL
`+`	Matches one or more occurrences
`?`	Matches zero or one occurrence
`*`	Matches zero or more occurrences of the preceding subexpression
`{m}`	Matches exactly *m* occurrences of the preceding expression
`{m, }`	Matches at least *m* occurrences of the preceding subexpression
`{m,n}`	Matches at least *m*, but not more than *n*, occurrences of the preceding subexpression
`[...]`	Matches any single character in the list within the brackets
`|`	Matches one of the alternatives
`( ... )`	Treats the enclosed expression within the parentheses as a unit. The subexpression can be a string of literals or a complex expression containing operators.

Using Metacharacters with Regular Expressions

Syntax	Description
`^`	Matches the beginning of a string
`$`	Matches the end of a string
`\`	Treats the subsequent metacharacter in the expression as a literal
`\n`	Matches the *n*th (1–9) preceding subexpression of whatever is grouped within parentheses. The parentheses cause an expression to be remembered; a backreference refers to it.
`\d`	A digit character
`[:class:]`	Matches any character belonging to the specified POSIX character class
`[^:class:]`	Matches any single character *not* in the list within the brackets

Lesson Agenda

- Introduction to regular expressions
- Using metacharacters with regular expressions
- Using the regular expressions functions:
  - `REGEXP_LIKE`
  - `REGEXP_REPLACE`
  - `REGEXP_INSTR`
  - `REGEXP_SUBSTR`
- Accessing subexpressions
- Using the `REGEXP_COUNT` function

Regular Expressions Functions and Conditions: Syntax

```
REGEXP_LIKE   (source_char, pattern [,match_option]
```

```
REGEXP_INSTR  (source_char, pattern [, position
               [, occurrence [, return_option
               [, match_option [, subexpr]]]]])
```

```
REGEXP_SUBSTR (source_char, pattern [, position
               [, occurrence [, match_option
               [, subexpr]]]])
```

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

```
REGEXP_COUNT (source_char, pattern [, position

               [, occurrence [, match_option]]])
```

Performing a Basic Search by Using the `REGEXP_LIKE` Condition

```
REGEXP_LIKE(source_char, pattern [, match_parameter ])
```

```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$');
```

	FIRST_NAME	LAST_NAME
1	Steven	King
2	Steven	Markle
3	Stephen	Stiles

Replacing Patterns by Using the `REGEXP_REPLACE` Function

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

```
SELECT REGEXP_REPLACE(phone_number, '\.','-') AS phone
FROM employees;
```

Original

	LAST_NAME	PHONE
1	OConnell	650.507.9833
2	Grant	650.507.9844
3	Whalen	515.123.4444
4	Hartstein	515.123.5555

Partial results

	LAST_NAME	PHONE
1	OConnell	650-507-9833
2	Grant	650-507-9844
3	Whalen	515-123-4444
4	Hartstein	515-123-5555

Finding Patterns by Using the `REGEXP_INSTR` Function

```
REGEXP_INSTR  (source_char, pattern [, position [,
   occurrence [, return_option [, match_option]]]])
```

```
SELECT street_address,
REGEXP_INSTR(street_address,'[[:alpha:]]') AS
   First_Alpha_Position
FROM  locations;
```

	STREET_ADDRESS	FIRST_ALPHA_POSITION
1	1297 Via Cola di Rie	6
2	93091 Calle della Testa	7
3	2017 Shinjuku-ku	6
4	9450 Kamiya-cho	6

Extracting Substrings by Using the REGEXP_SUBSTR Function

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

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

	ROAD
1	Via
2	Calle
3	(null)
4	(null)
5	Jabberwocky

Lesson Agenda

- Introduction to regular expressions
- Using metacharacters with regular expressions
- Using the regular expressions functions:
  - REGEXP_LIKE
  - REGEXP_REPLACE
  - REGEXP_INSTR
  - REGEXP_SUBSTR
- Accessing subexpressions
- Using the REGEXP_COUNT function

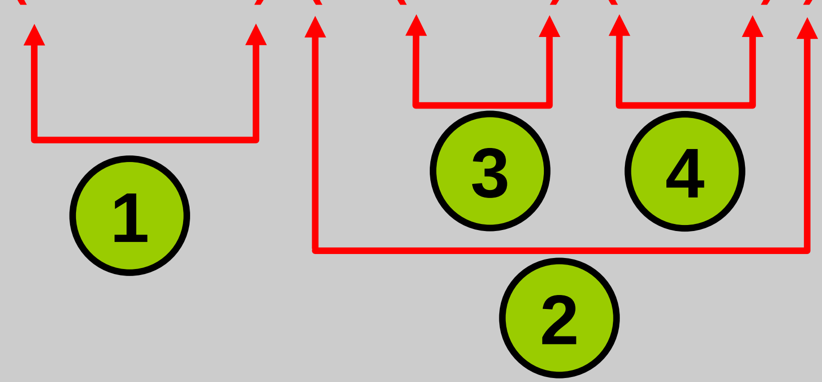
Subexpressions

Examine this expression:

```
(1 2 3)(4(5 6)(7 8))
```

The subexpressions are:

```
(1 2 3) (4(5 6) (7 8))
```



Using Subexpressions with Regular Expression Support

```
SELECT
  REGEXP_INSTR
① ('0123456789',       -- source char or search value
② '(123)(4(56)(78))', -- regular expression patterns
③ 1,                   -- position to start searching
④ 1,                   -- occurrence
⑤ 0,                   -- return option
⑥ 'i',                 -- match option (case insensitive)
⑦ 1)              -- sub-expression on which to search
    "Position"
FROM dual;
```

	Position
1	2

Why Access the *n*th Subexpression?

- A more realistic use: DNA sequencing
- You may need to find a specific subpattern that identifies a protein needed for immunity in mouse DNA.

```
SELECT
    REGEXP_INSTR('ccacctttccctccactcctcacgttctcacctgtaaagcgtccctc
    cctcatccccatgcccccttaccctgcagggtagagtaggctagaaaccagagagctccaagc
    tccatctgtggagaggtgccatccttgggctgcagagagaggagaatttgccccaaagctgcc
    tgcagagcttcaccacccttagtctcacaaagccttgagttcatagcatttcttgagttttca
    ccctgcccagcaggacactgcagcacccaaagggcttcccaggagtagggttgccctcaagag
    gctcttgggtctgatggccacatcctggaattgttttcaagttgatggtcacagccctgaggc
    atgtaggggcgtggggatgcgctctgctctgctctcctctcctgaacccctgaaccctctggc
    taccccagagcacttagagccag',
                       '(gtc(tcac)(aaag))',
                        1, 1, 0, 'i',
                        1) "Position"
FROM dual;
```

	Position
1	195

REGEXP_SUBSTR: Example

```
SELECT
  REGEXP_SUBSTR
① ('acgctgcactgca', -- source char or search value
②  'acg(.*)gca',     -- regular expression pattern
③  1,                -- position to start searching
④  1,                -- occurrence
⑤  'i',              -- match option (case insensitive)
⑥  1)                -- sub-expression
 "Value"
FROM dual;
```

	Value
1	ctgcact

Lesson Agenda

- Introduction to regular expressions
- Using metacharacters with regular expressions
- Using the regular expressions functions:
  - `REGEXP_LIKE`
  - `REGEXP_REPLACE`
  - `REGEXP_INSTR`
  - `REGEXP_SUBSTR`
- Accessing subexpressions
- Using the `REGEXP_COUNT` function

Using the REGEXP_COUNT Function

```
REGEXP_COUNT (source_char, pattern [, position
               [, occurrence [, match_option]]])
```

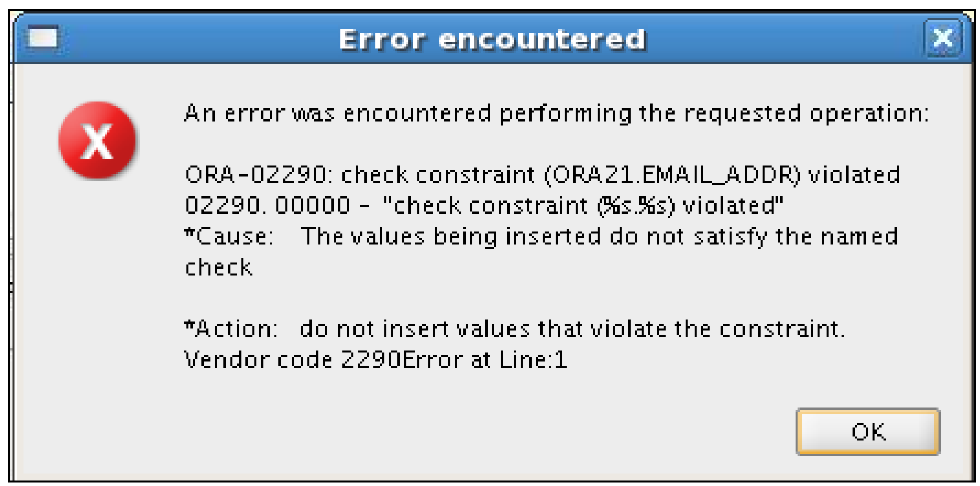
```
SELECT REGEXP_COUNT(
    'ccacctttccctccactcctcacgttctcacctgtaaagcgtccctccctcatccccatgcccccttaccctgcag
    ggtagagtaggctagaaaccagagagctccaagctccatctgtggagaggtgccatccttgggctgcagagagaggag
    aatttgccccaaagctgcctgcagagcttcaccacccttagtctcacaaagccttgagttcatagcatttcttgagtt
    ttcaccctgcccagcaggacactgcagcacccaaagggcttcccaggagtagggttgccctcaagaggctcttgggtc
    tgatggccacatcctggaattgttttcaagttgatggtcacagccctgaggcatgtaggggcgtggggatgcgctctg
    ctctgctctcctctcctgaacccctgaaccctctggctaccccagagcacttagagccag',
    'gtc') AS Count
FROM dual;
```

	COUNT
1	4

Regular Expressions and Check Constraints: Examples

```
ALTER TABLE emp8
  ADD CONSTRAINT email_addr
  CHECK(REGEXP_LIKE(email,'@')) NOVALIDATE;
```

```
INSERT INTO emp8 VALUES
  (500,'Christian','Patel','ChrisP2creme.com',
   1234567890,'12-Jan-2004','HR_REP',2000,null,102,40);
```



Quiz

With the use of regular expressions in SQL and PL/SQL, you can:

1. Avoid intensive string processing of SQL result sets by middle-tier applications
2. Avoid data validation logic on the client
3. Enforce constraints on the server

Summary

In this lesson, you should have learned how to use regular expressions to search for, match, and replace strings.

Practice 7: Overview

This practice covers using regular expressions functions to do the following:

- Searching for, replacing, and manipulating data
- Creating a new `CONTACTS` table and adding a `CHECK` constraint to the `p_number` column to ensure that phone numbers are entered into the database in a specific standard format
- Testing the adding of some phone numbers into the `p_number` column by using various formats

ORACLE

C

Using SQL Developer

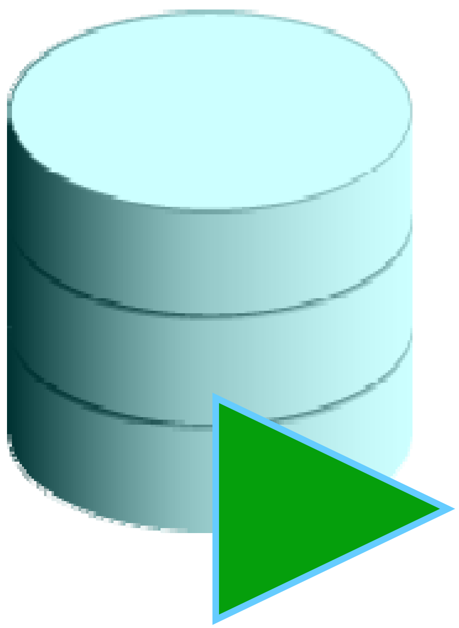
Objectives

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

- List the key features of Oracle SQL Developer
- Identify menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use SQL Worksheet
- Save and run SQL scripts
- Create and save reports

What Is Oracle SQL Developer?

- Oracle SQL Developer is a graphical tool that enhances productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema by using standard Oracle database authentication.

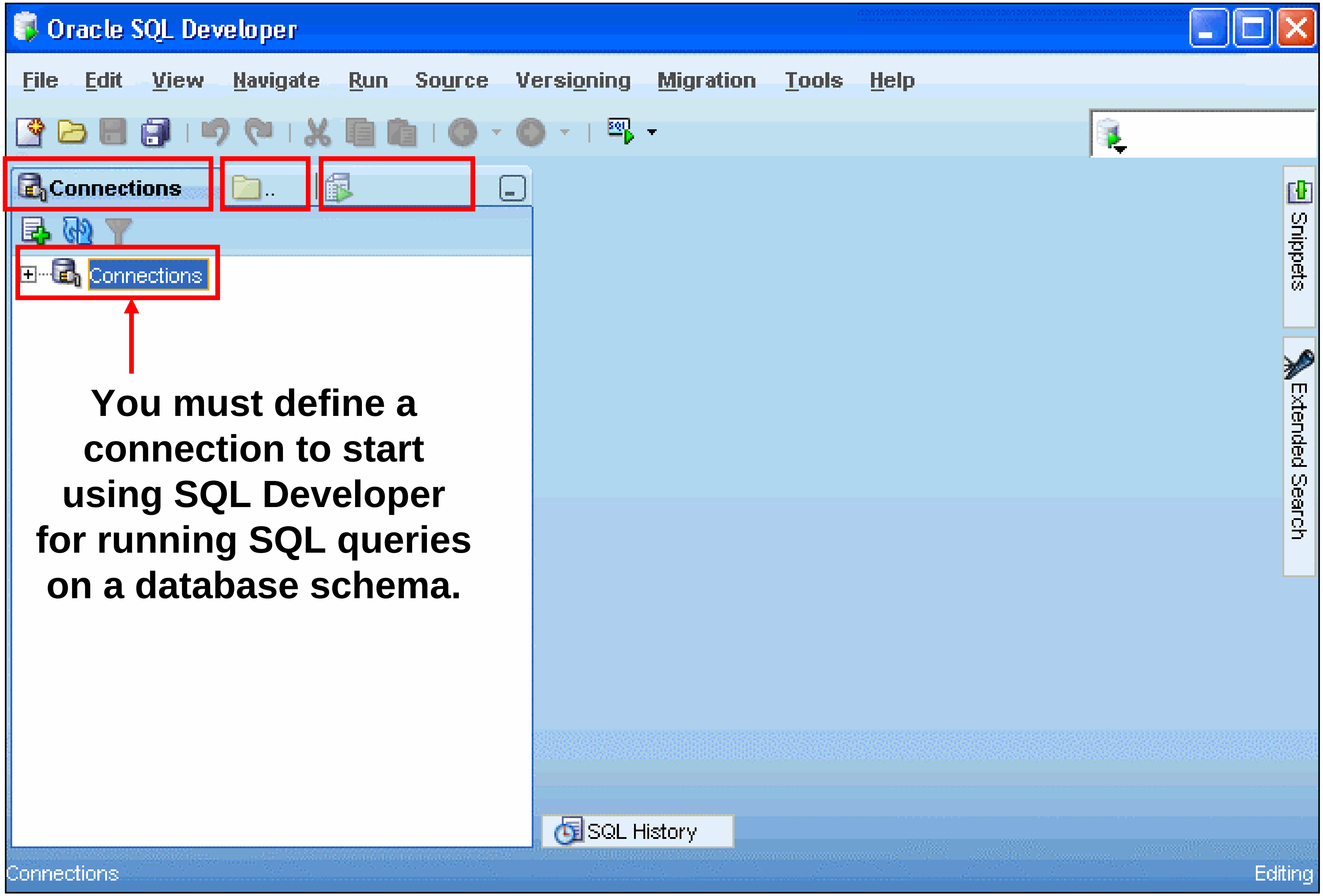


SQL Developer

Specifications of SQL Developer

- Shipped along with Oracle Database 11*g* Release 2
- Developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Default connectivity by using the Java Database Connectivity (JDBC) thin driver
- Connects to Oracle Database version 9.2.0.1 and later
- Freely downloadable from the following link:
  – http://www.oracle.com/technology/products/database/sql_developer/index.html

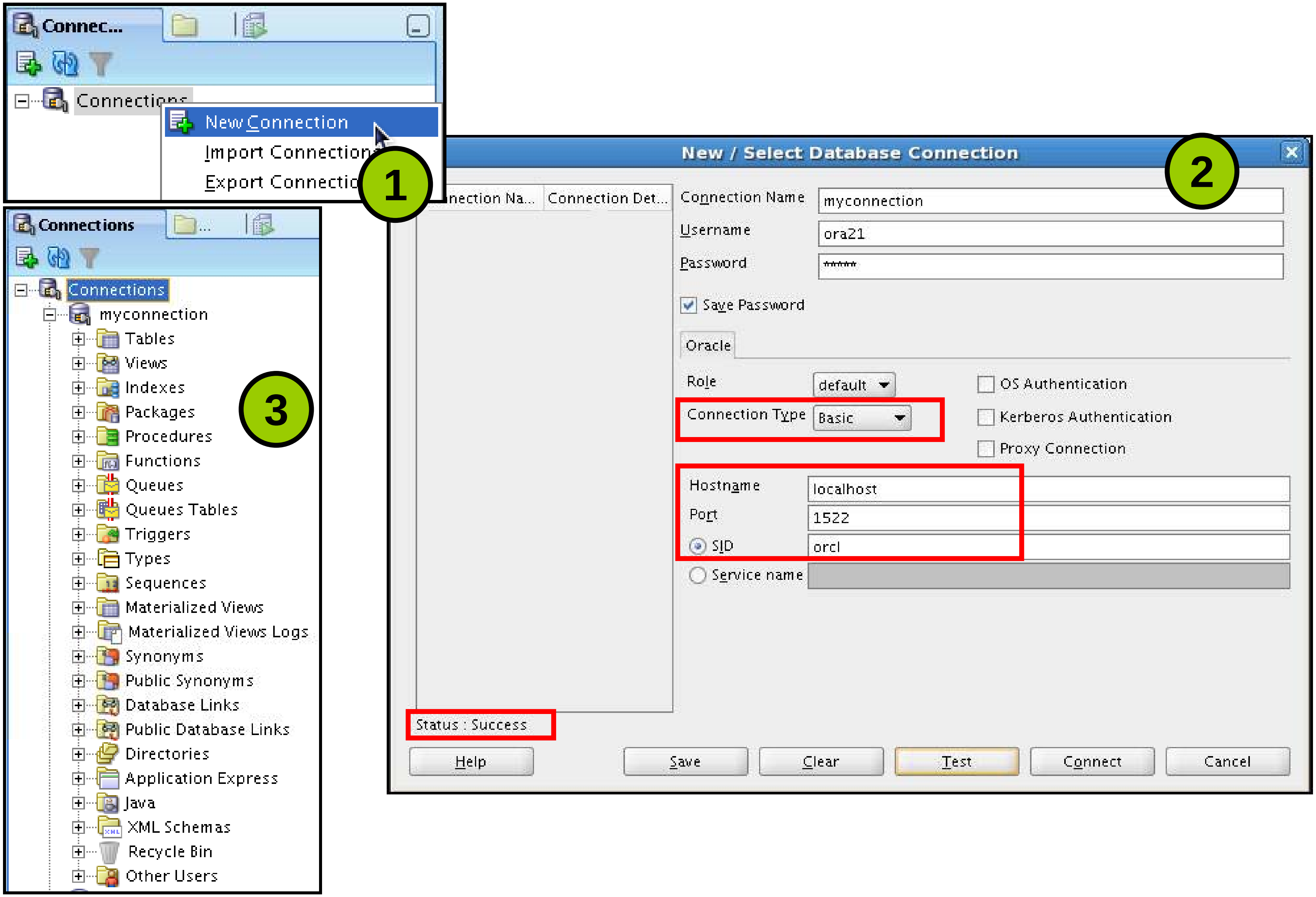
SQL Developer 1.5 Interface



Creating a Database Connection

- You must have at least one database connection to use SQL Developer.
- You can create and test connections for multiple:
  – Databases
  – Schemas
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an Extensible Markup Language (XML) file.
- Each additional database connection created is listed in the Connections Navigator hierarchy.

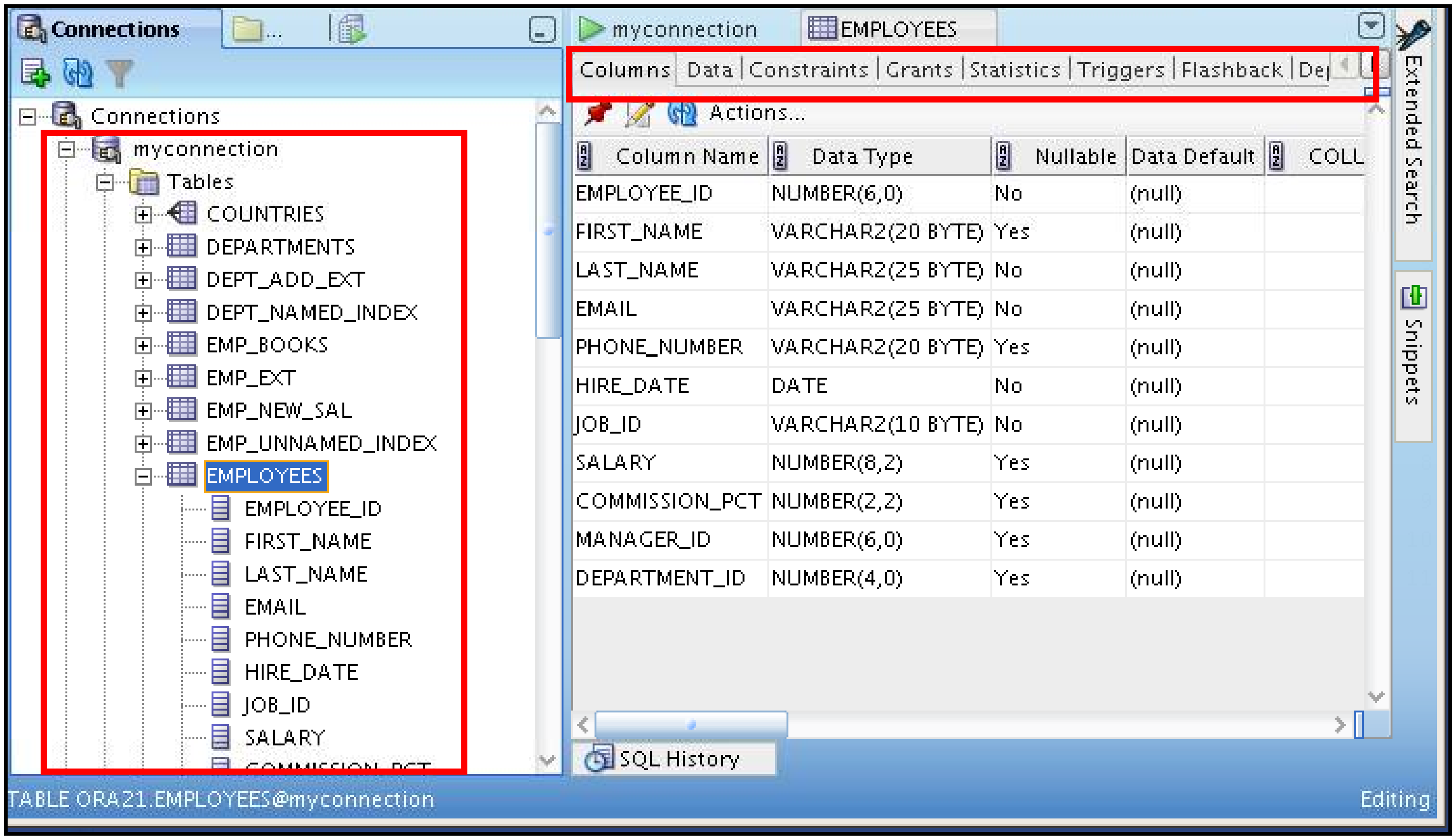
Creating a Database Connection



Browsing Database Objects

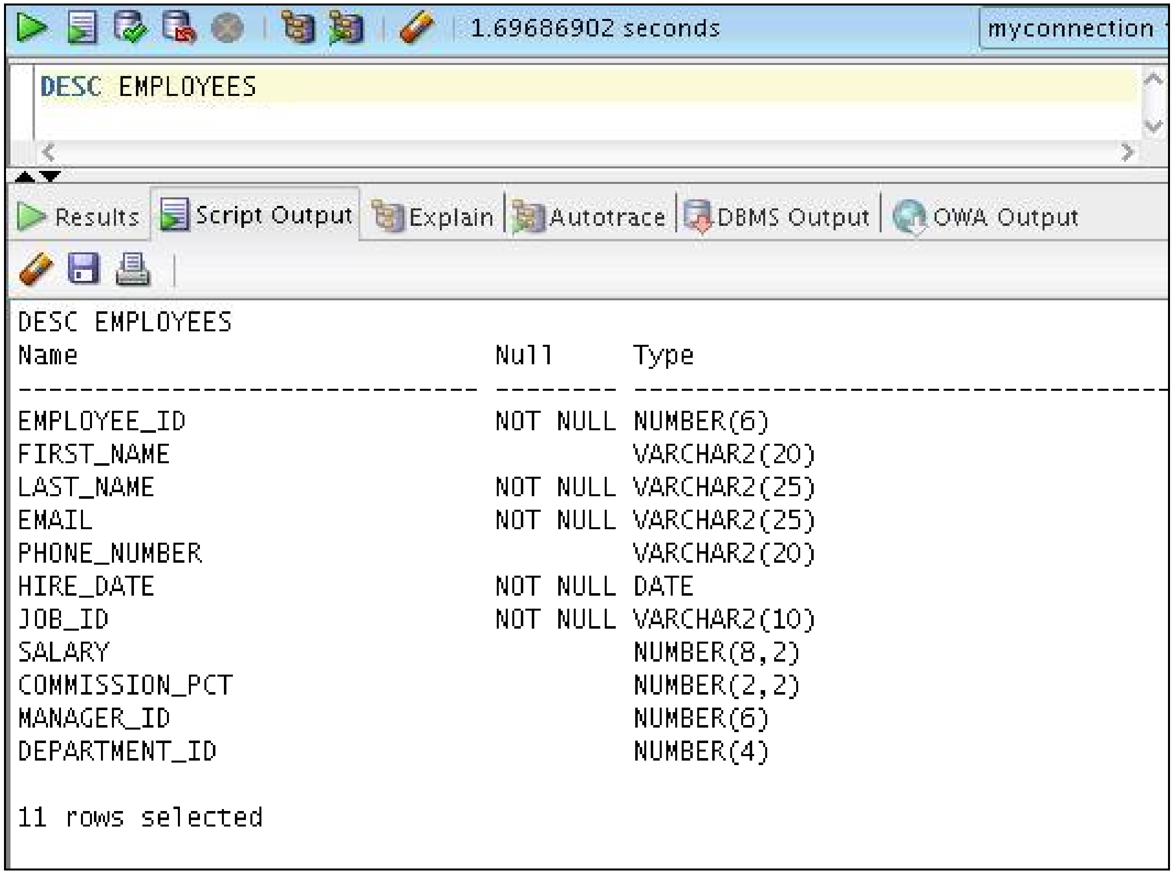
Use the Connections Navigator to:

- Browse through many objects in a database schema
- Review the definitions of objects at a glance



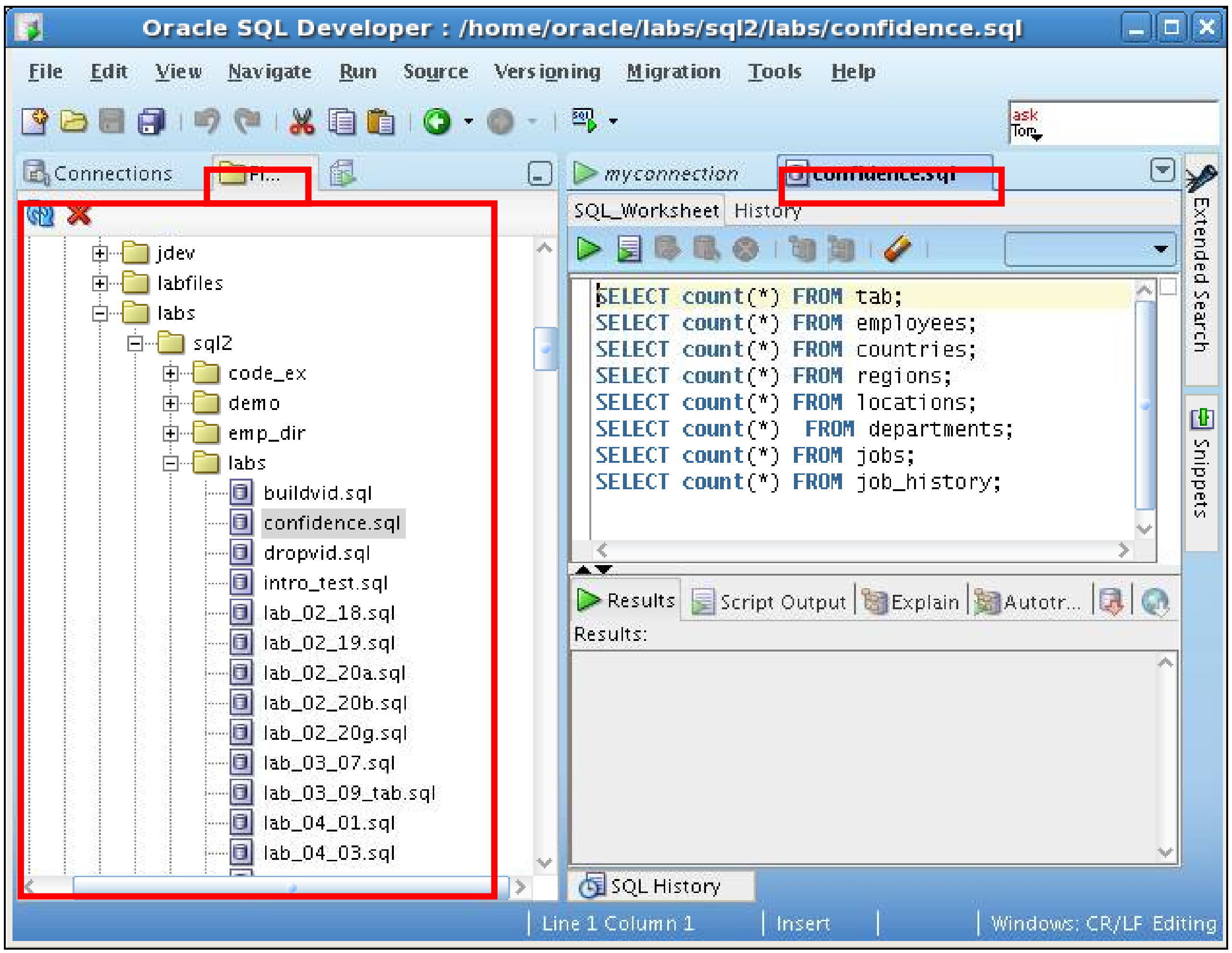
Displaying the Table Structure

Use the `DESCRIBE` command to display the structure of a table:



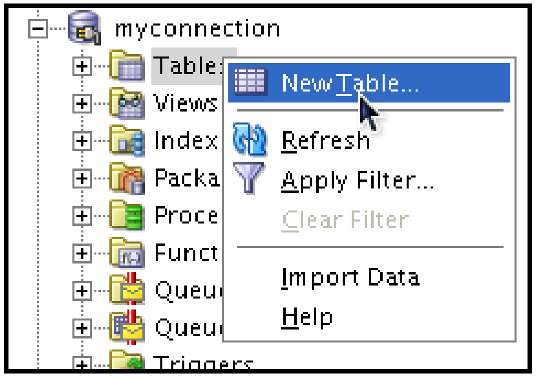
Browsing Files

Use the File Navigator to explore the file system and open system files.

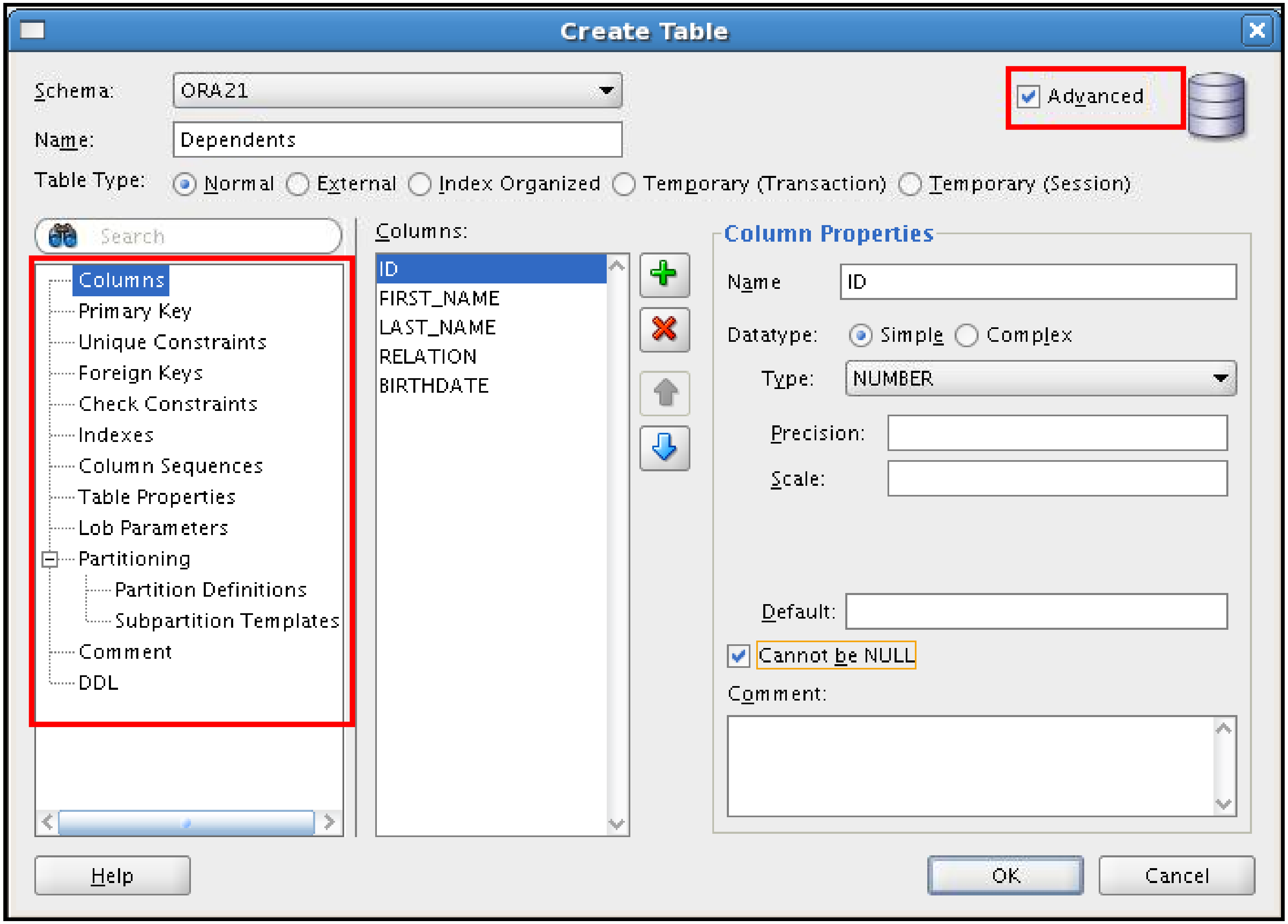


Creating a Schema Object

- SQL Developer supports the creation of any schema object by:
  - Executing a SQL statement in SQL Worksheet
  - Using the context menu
- Edit the objects by using an edit dialog box or one of the many context-sensitive menus.
- View the data definition language (DDL) for adjustments such as creating a new object or editing an existing schema object.

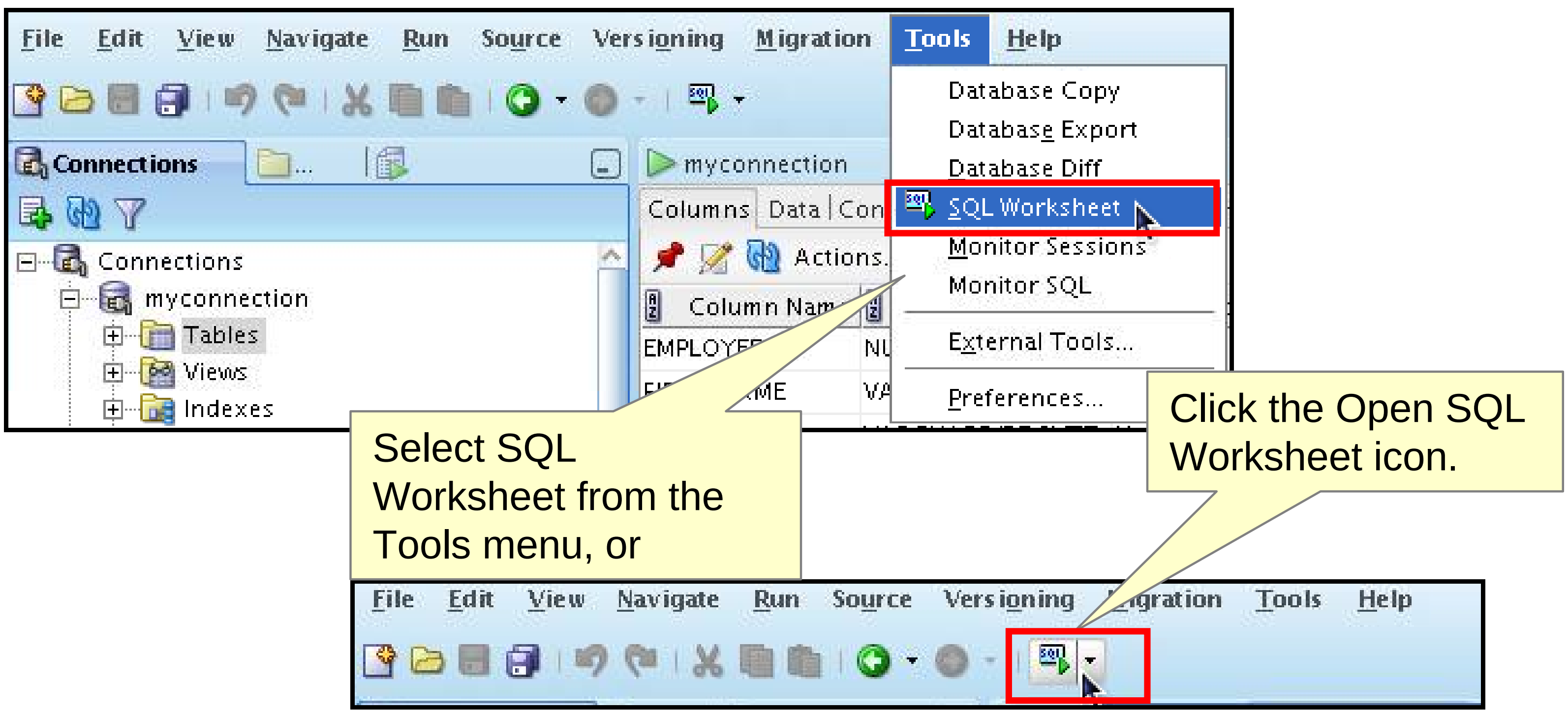


Creating a New Table: Example

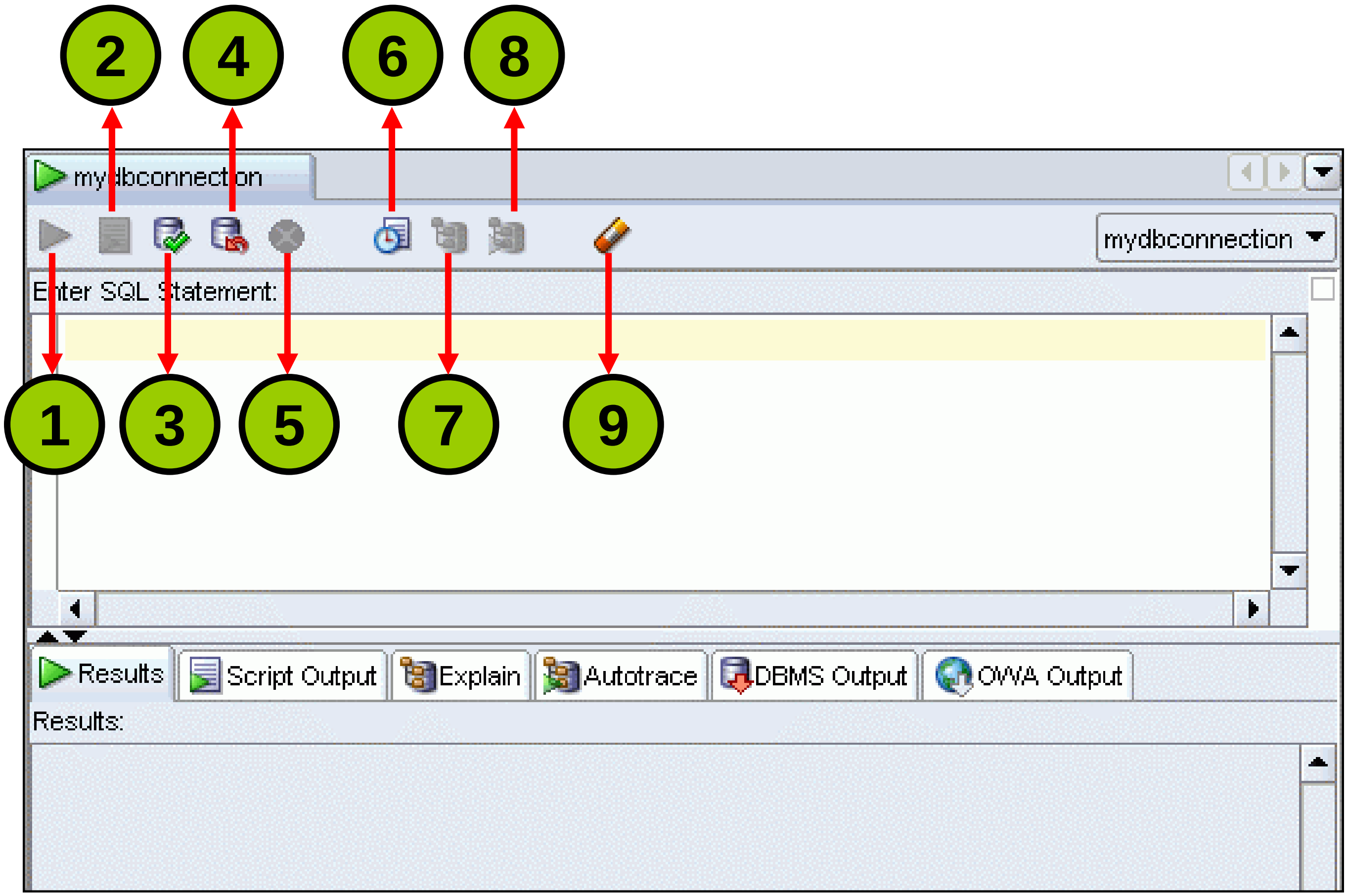


Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL *Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.

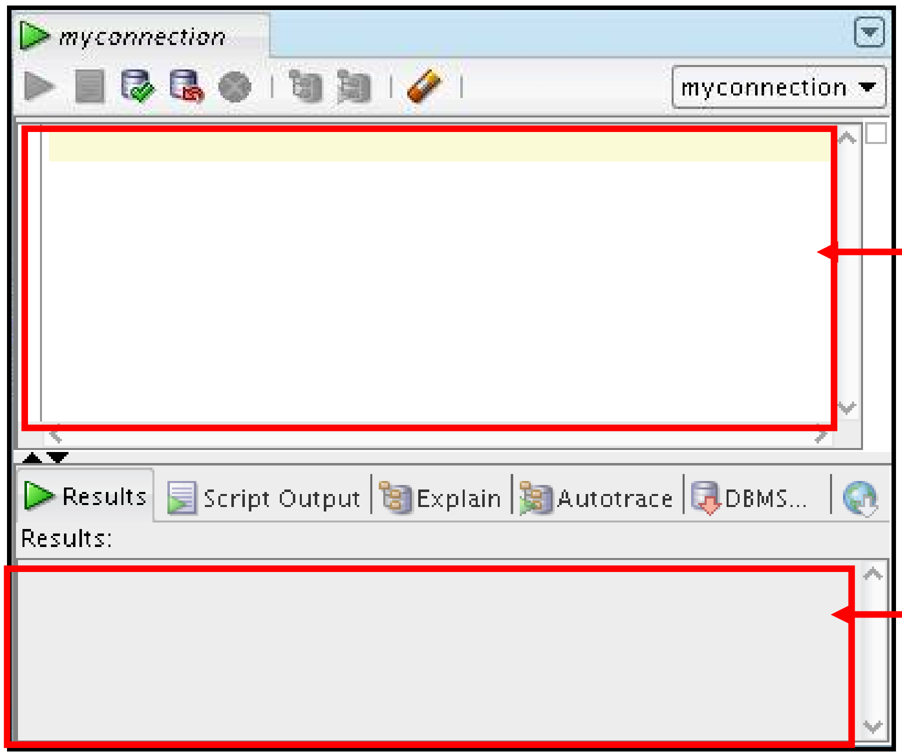


Using the SQL Worksheet



Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.

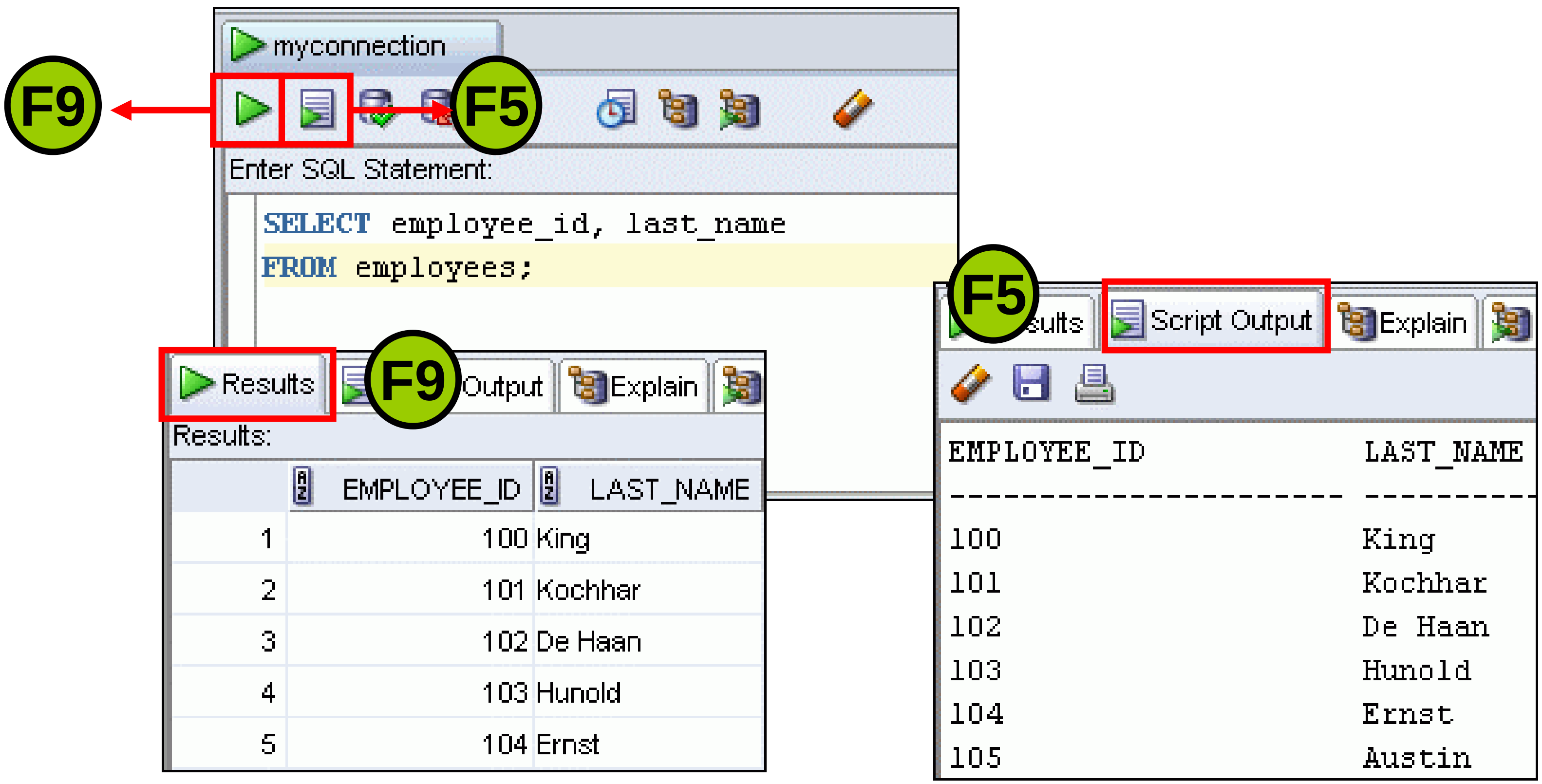


Enter SQL statements.

Results are shown here.

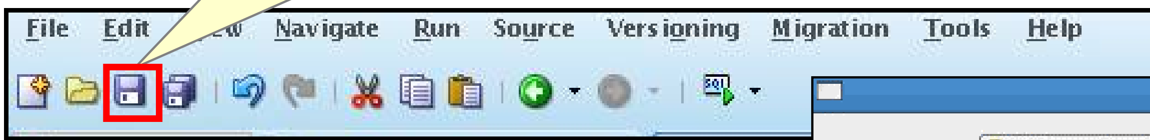
Executing SQL Statements

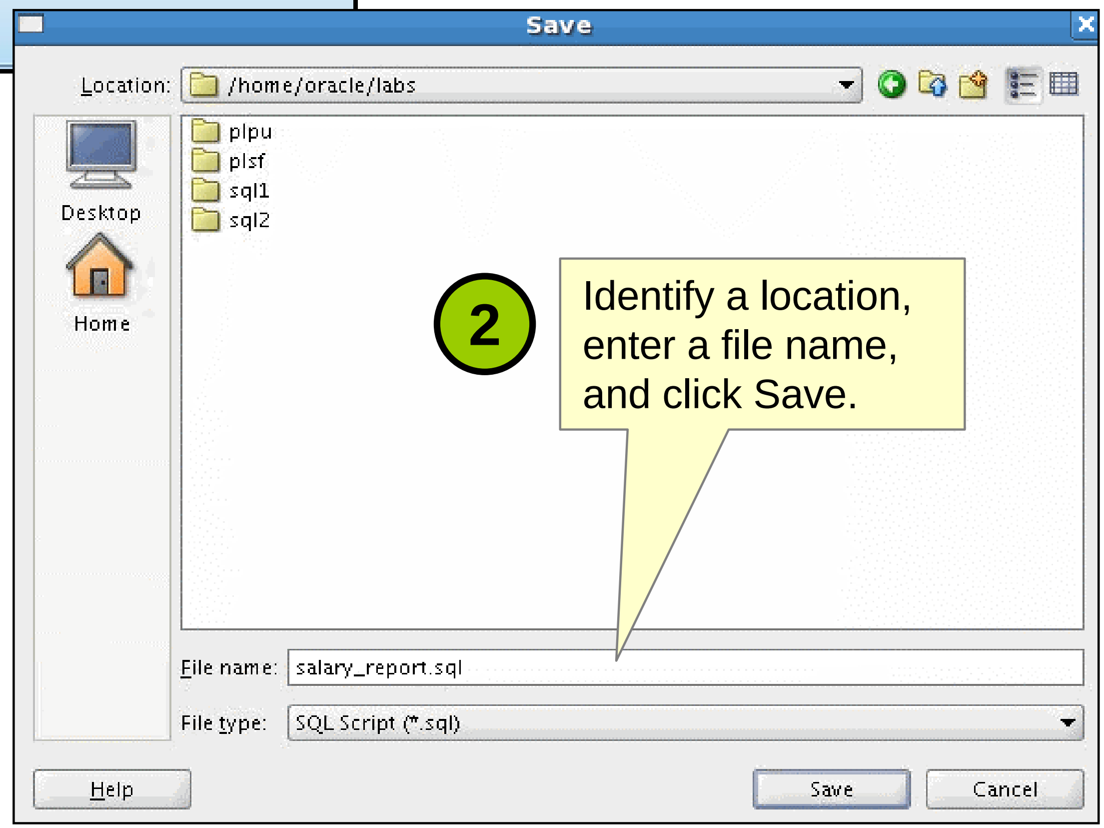
Use the Enter SQL Statement box to enter single or multiple SQL statements.



Saving SQL Scripts

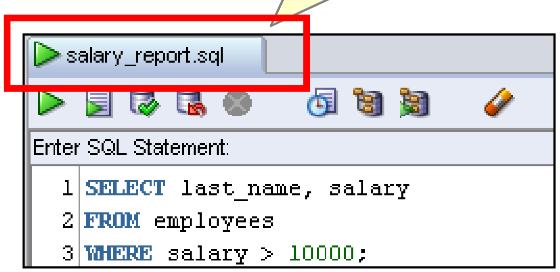
1 Click the Save icon to save your SQL statement to a file.





2 Identify a location, enter a file name, and click Save.

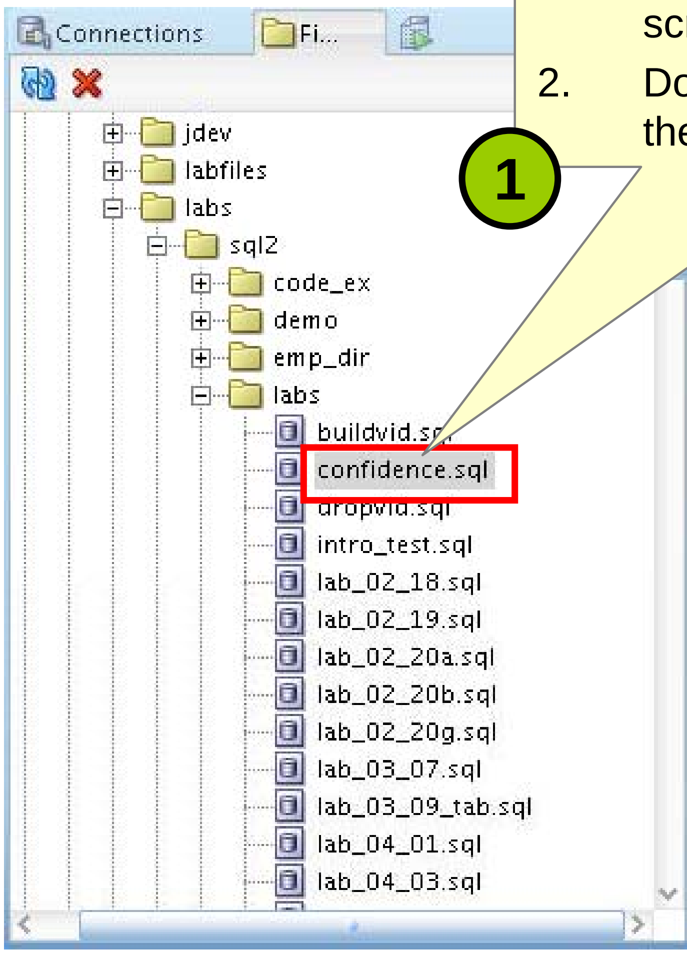
3 The contents of the saved file are visible and editable in your SQL Worksheet window.



Executing Saved Script Files: Method 1

1. Use the Files tab to locate the script file that you want to open.
2. Double-click the script to display the code in the SQL Worksheet.

1



To run the code, click either:

- Execute Script (F9), or
- Run Script (F5)

3

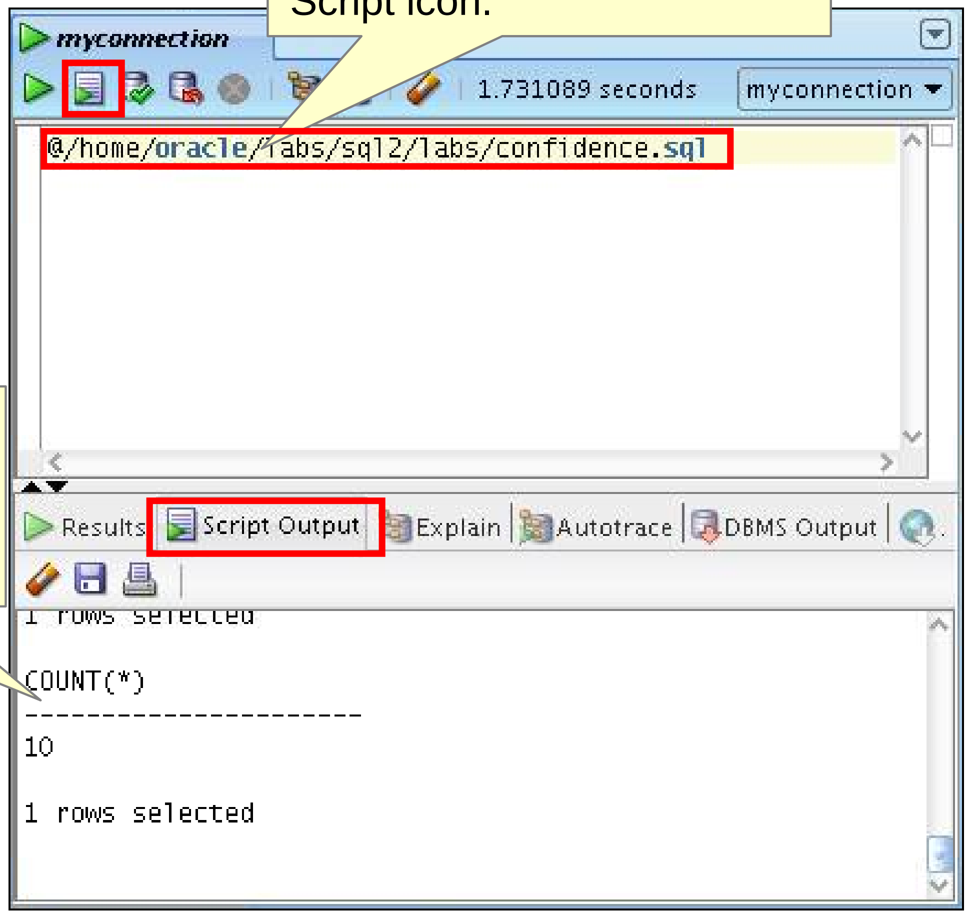
2

Select a connection from the drop-down list.

Executing Saved Script Files: Method 2

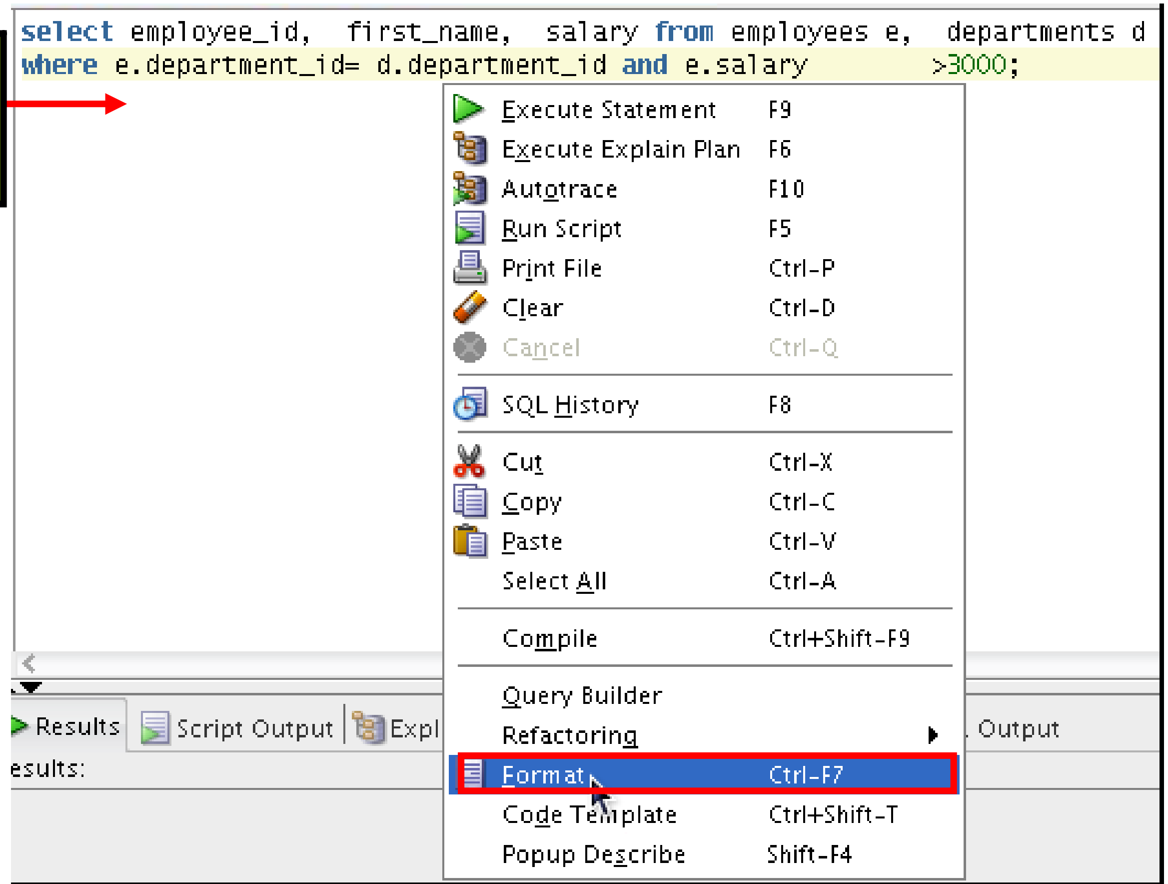
Use the @ command followed by the location and name of the file you want to execute, and click the Run Script icon.

The output from the script is displayed on the Script Output tabbed page.

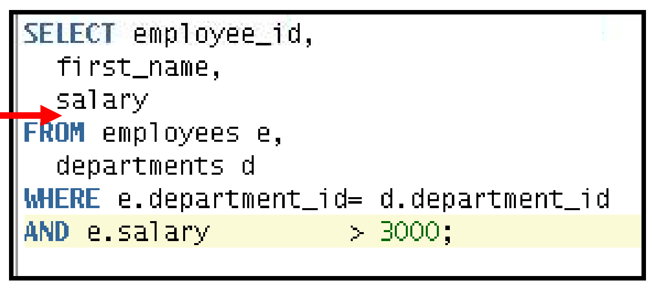


Formatting the SQL Code

Before formatting

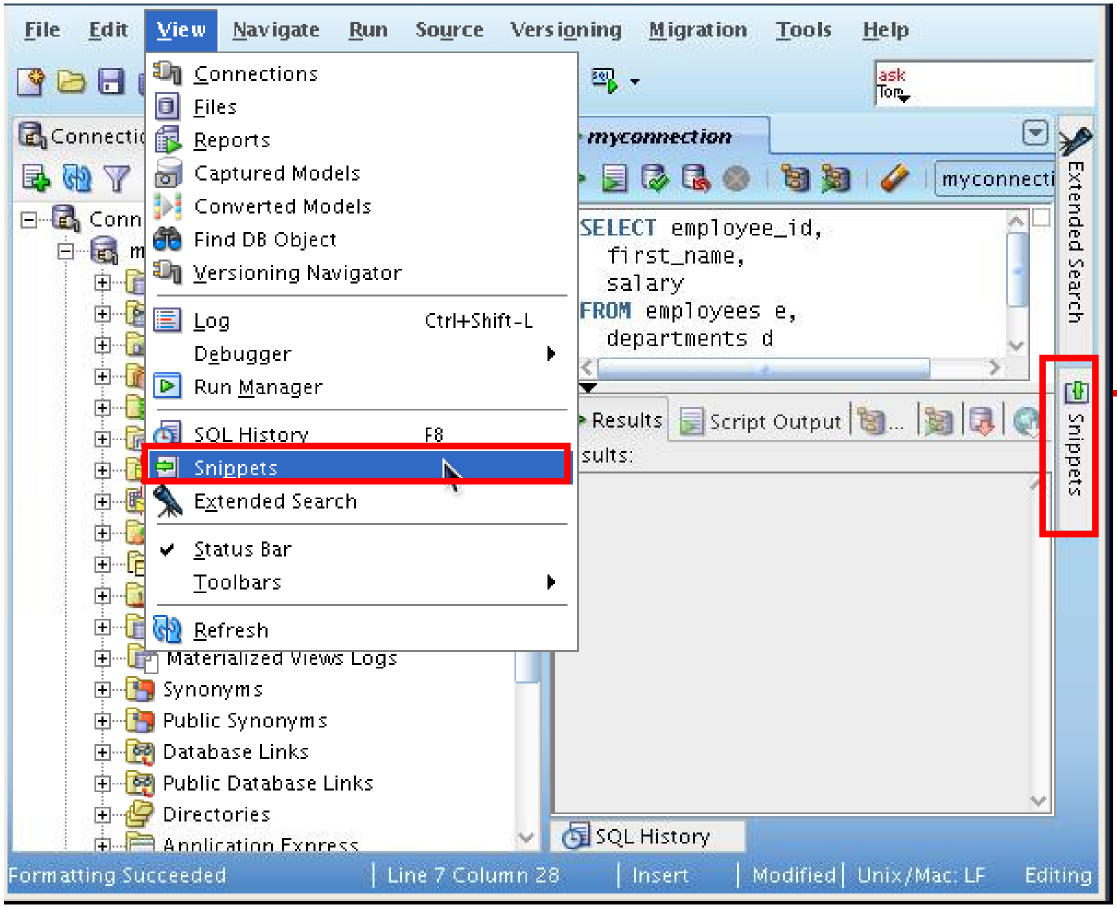


After formatting

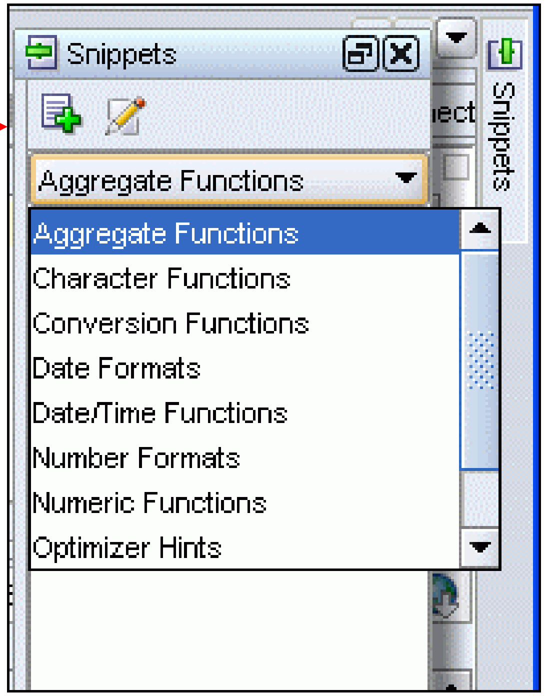


Using Snippets

Snippets are code fragments that may be just syntax or examples.

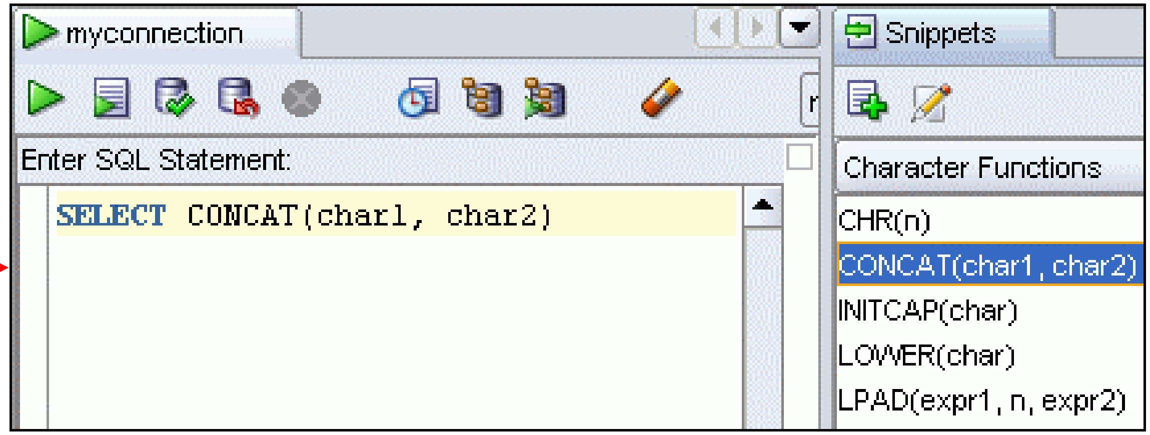


When you place your cursor here, it shows the Snippets window. From the drop-down list, you can select the functions category that you want.

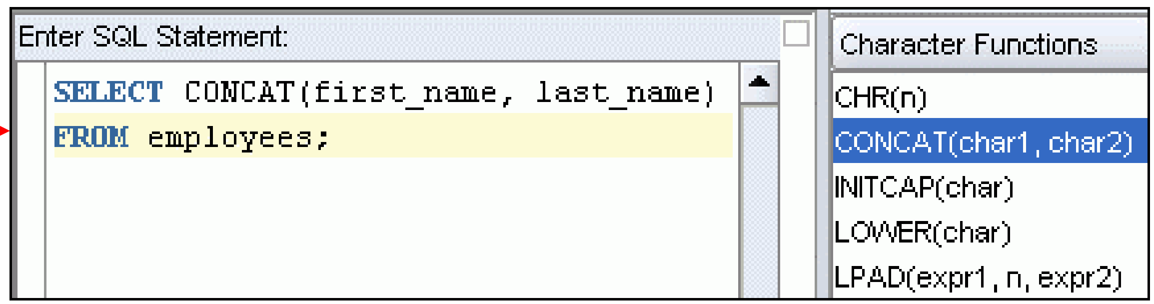


Using Snippets: Example

Inserting a snippet

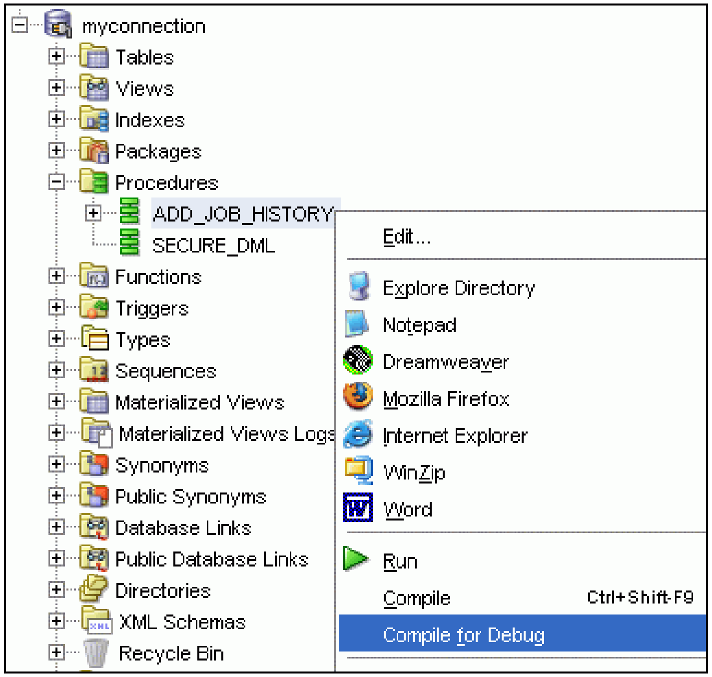


Editing the snippet



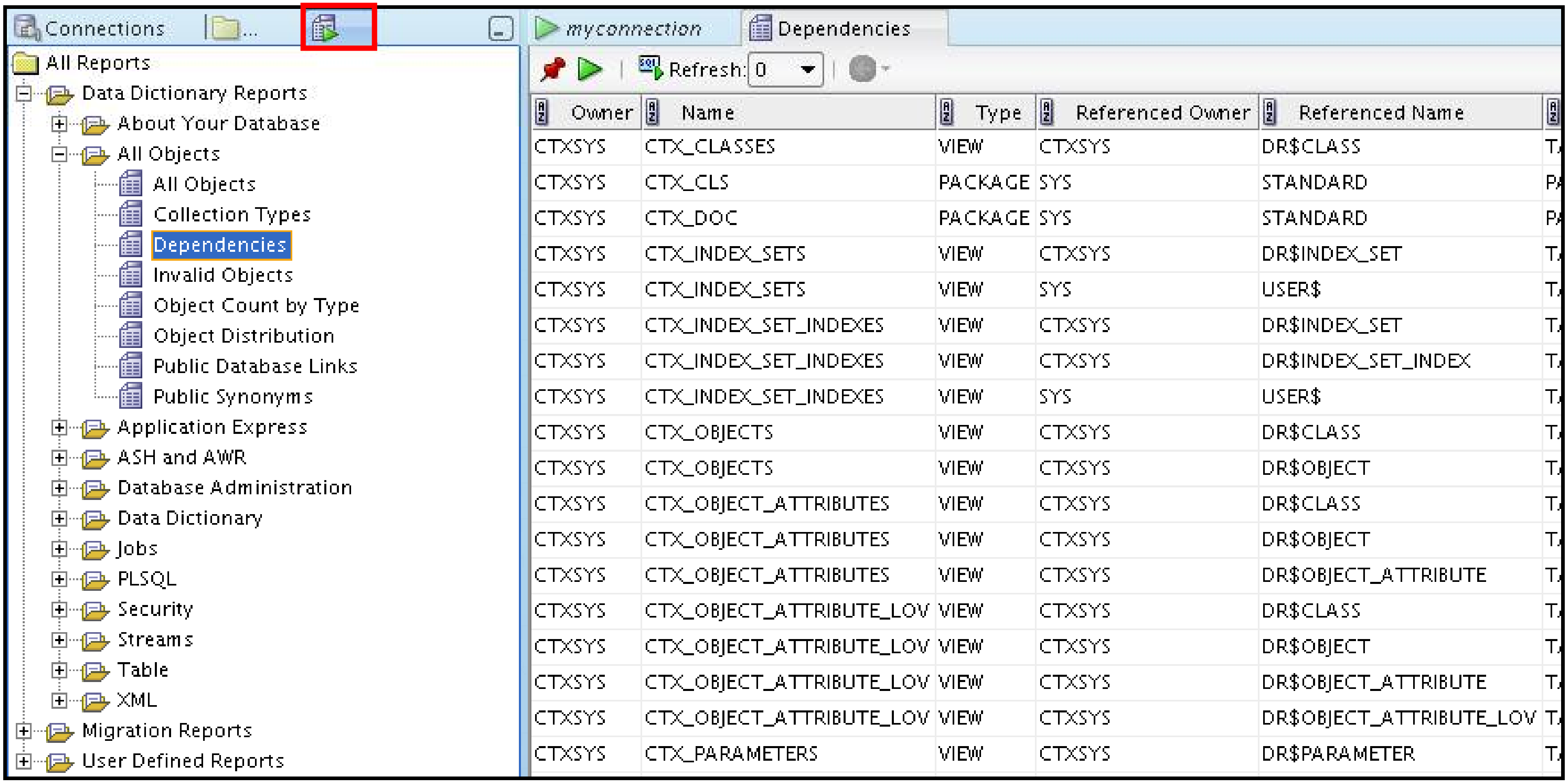
Debugging Procedures and Functions

- Use SQL Developer to debug PL/SQL functions and procedures.
- Use the "Compile for Debug" option to perform a PL/SQL compilation so that the procedure can be debugged.
- Use Debug menu options to set breakpoints, and to perform step into and step over tasks.



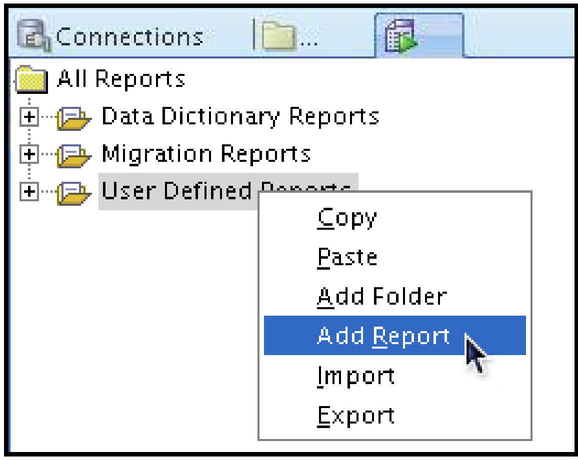
Database Reporting

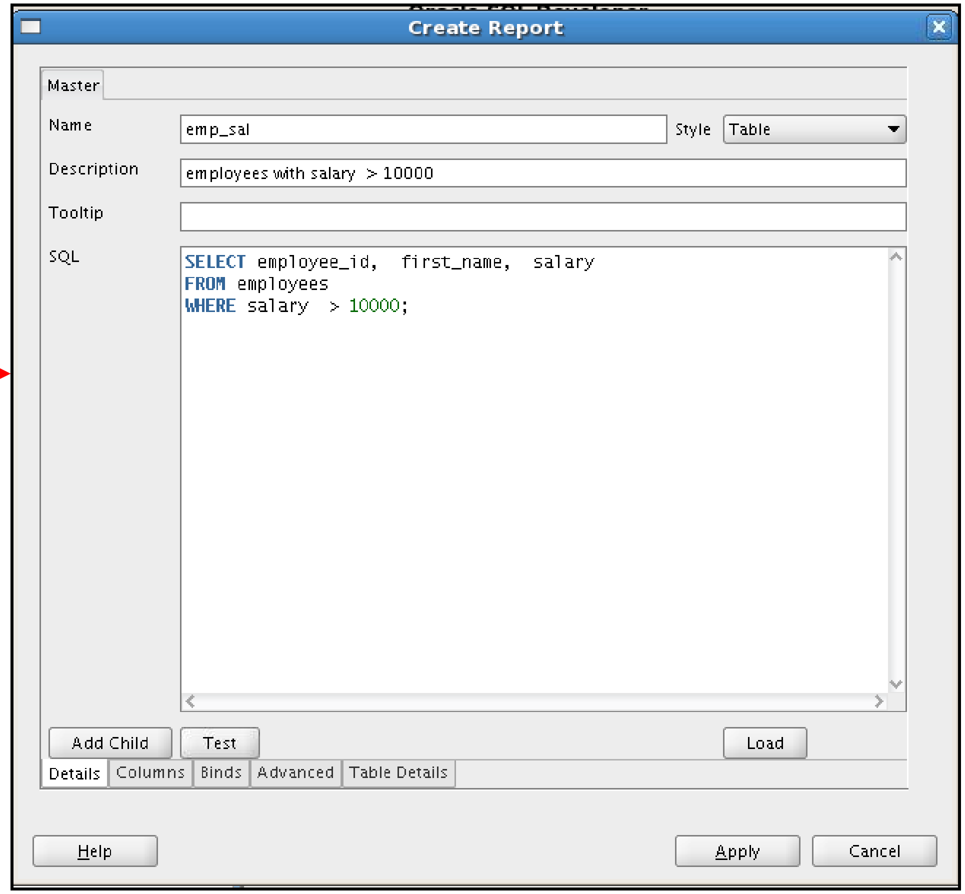
SQL Developer provides a number of predefined reports about the database and its objects.

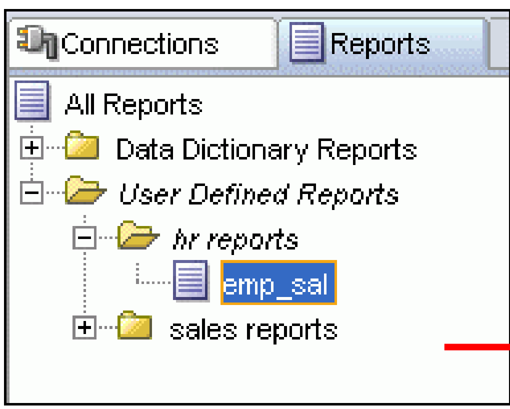


Creating a User-Defined Report

Create and save user-defined reports for repeated use.

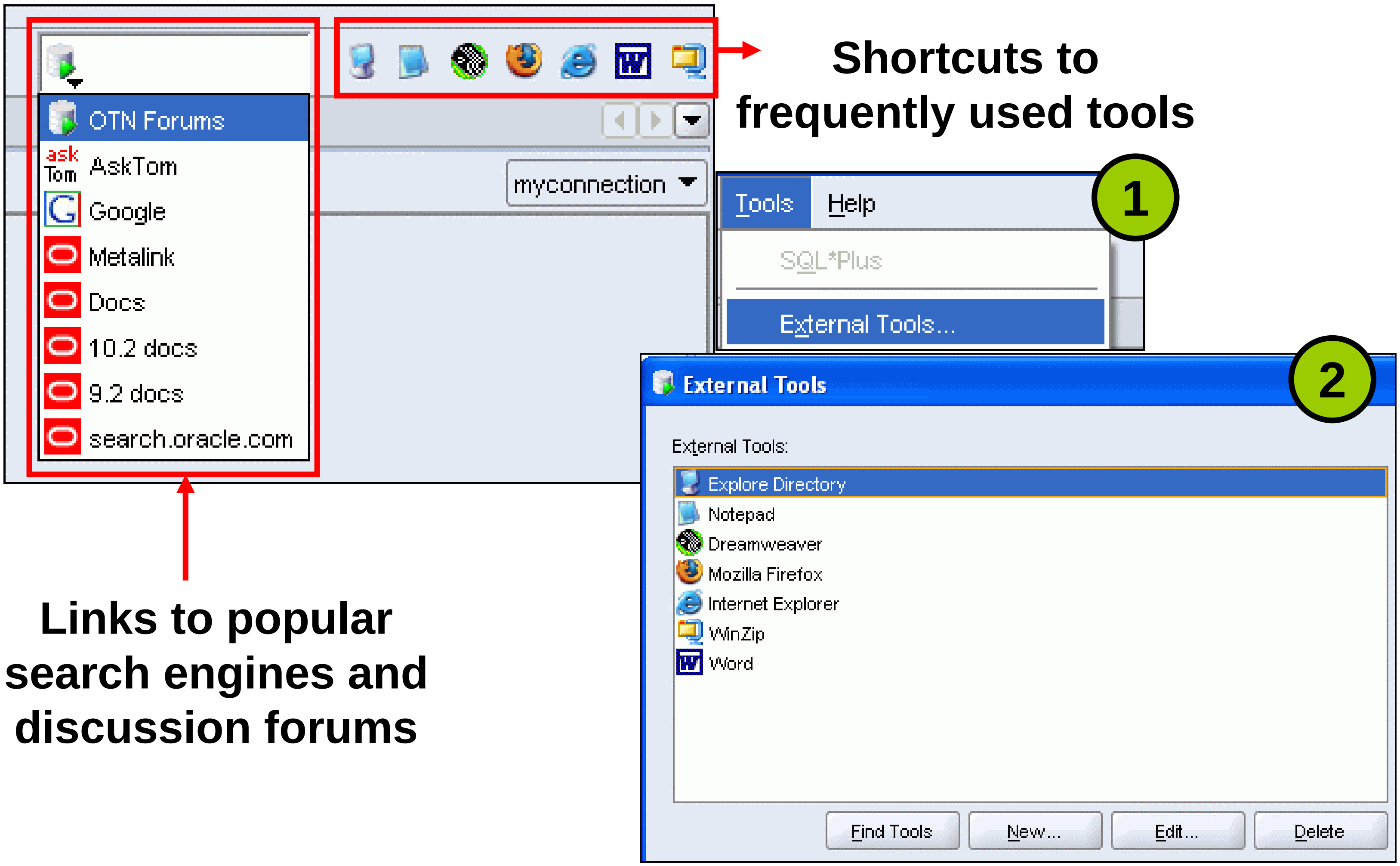






Organize reports in folders.

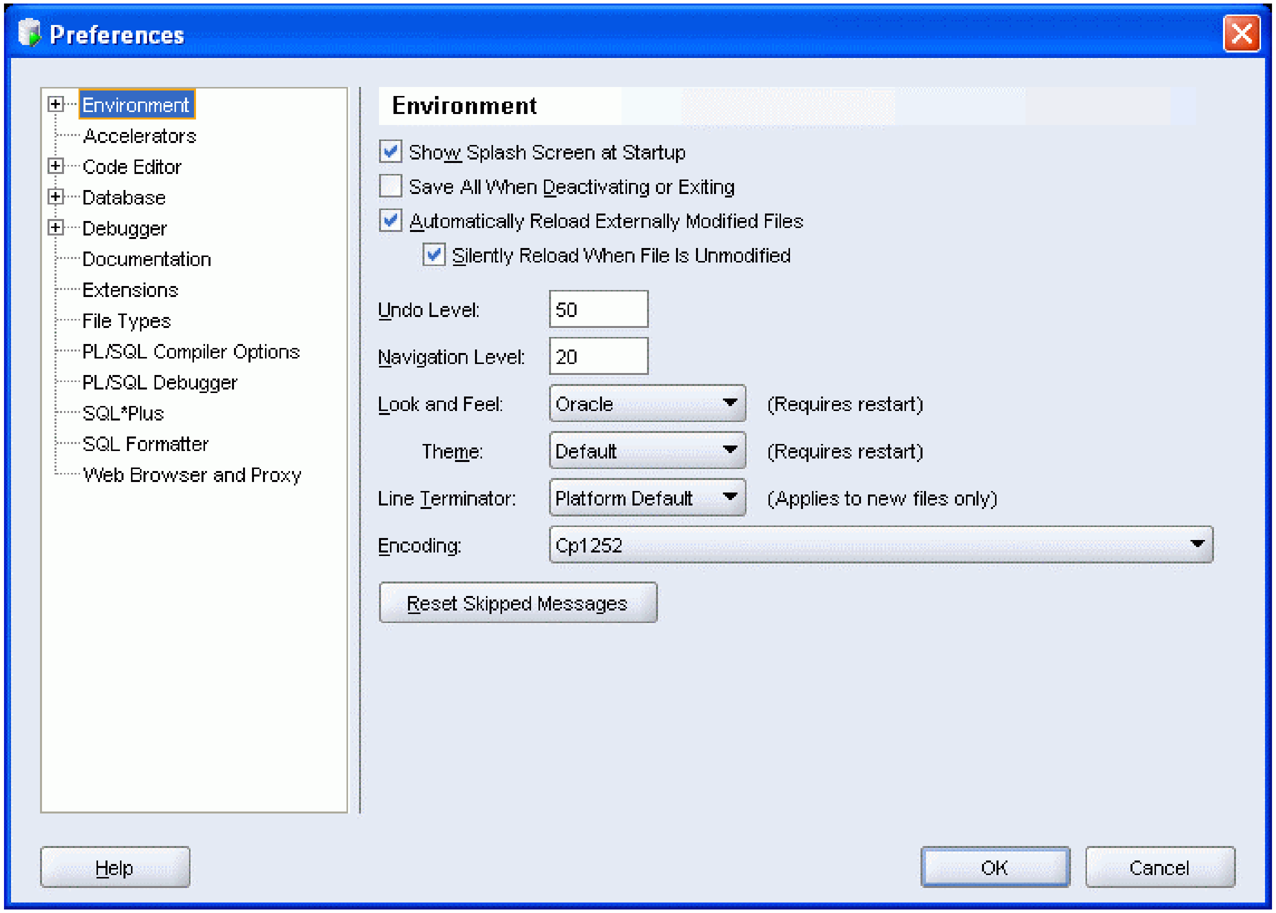
Search Engines and External Tools



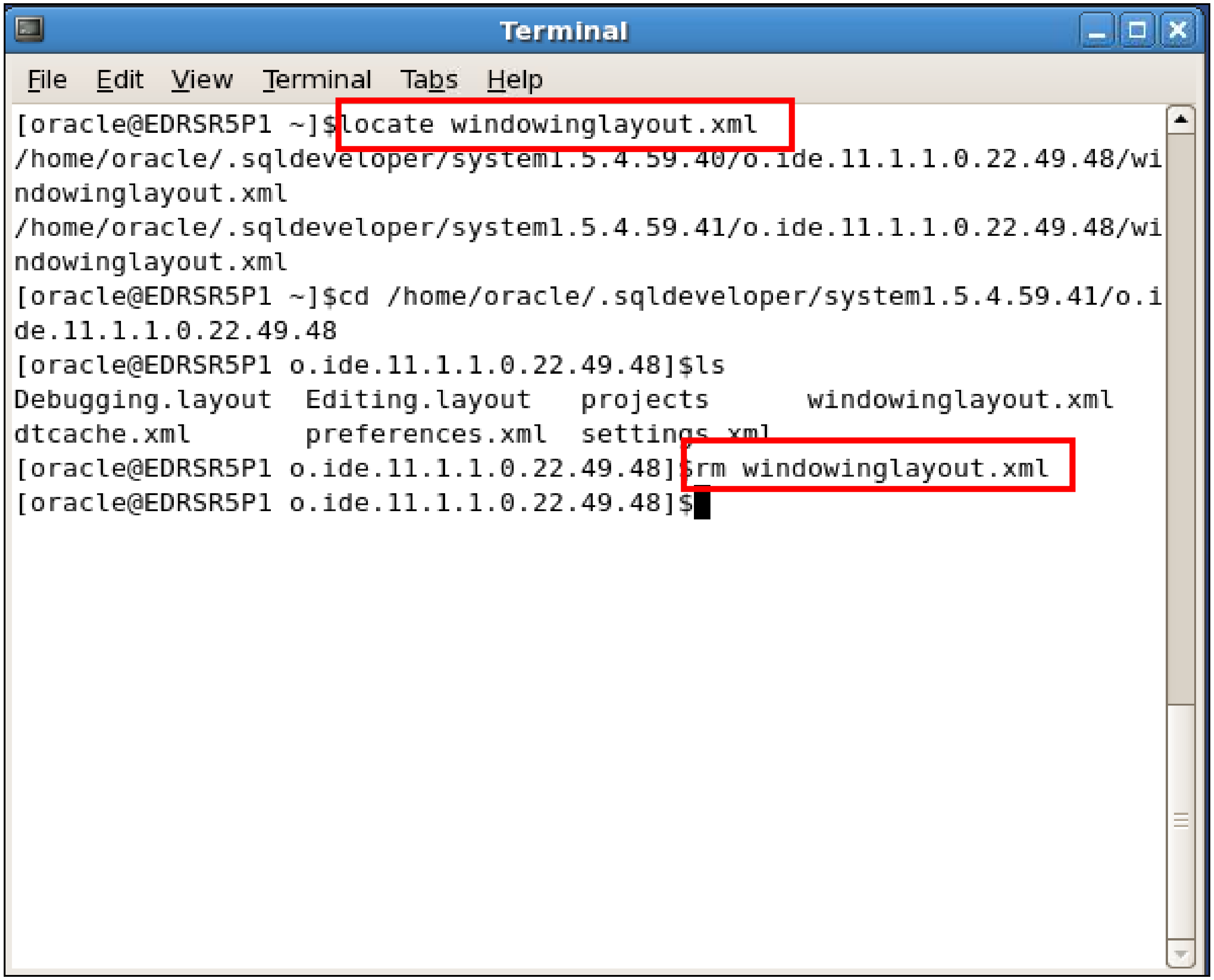
Links to popular search engines and discussion forums

Setting Preferences

- Customize the SQL Developer interface and environment.
- In the Tools menu, select Preferences.



Resetting the SQL Developer Layout



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

D

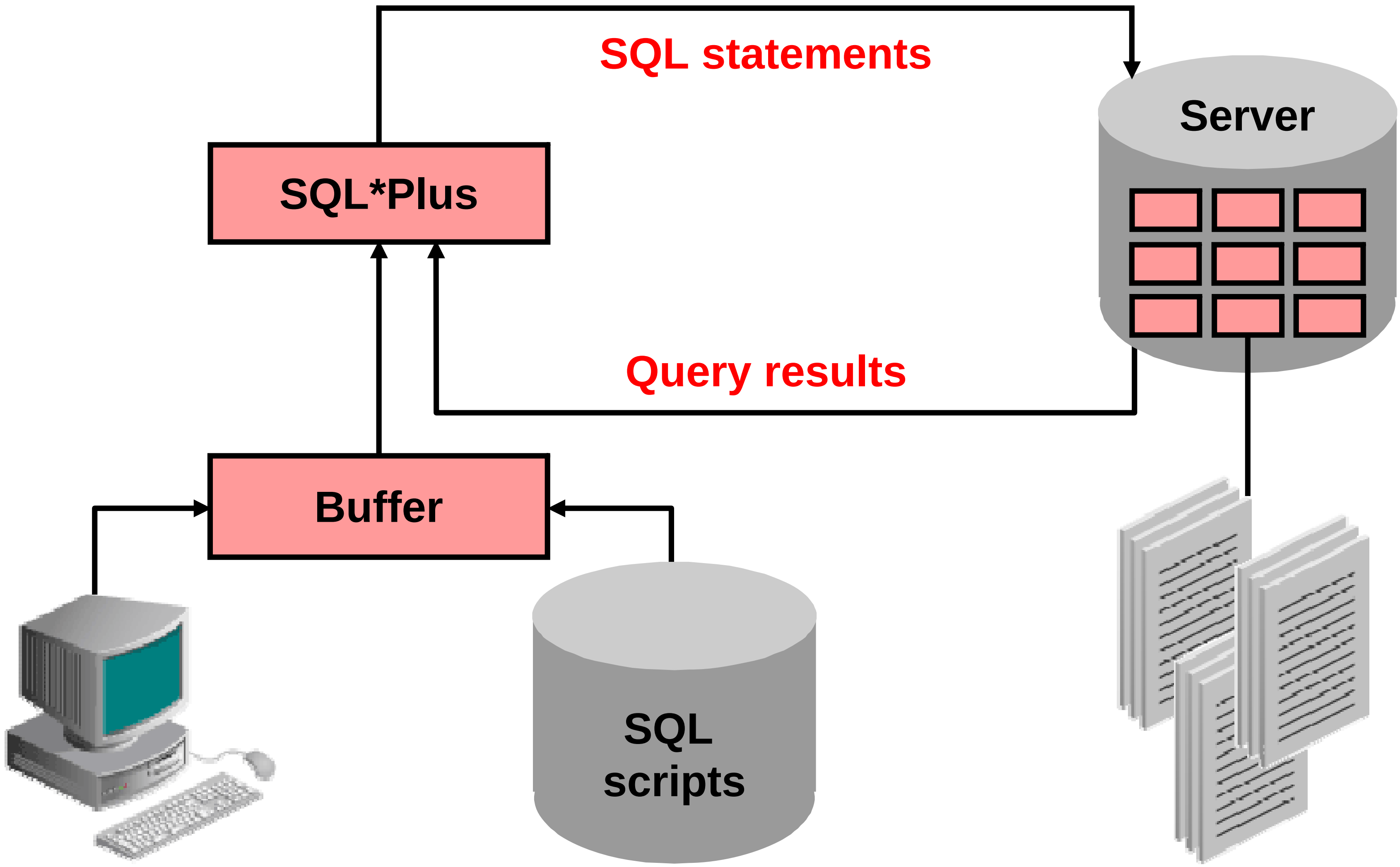
Using SQL*Plus

Objectives

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

- Log in to SQL*Plus
- Edit SQL commands
- Format the output using SQL*Plus commands
- Interact with script files

SQL and SQL*Plus Interaction



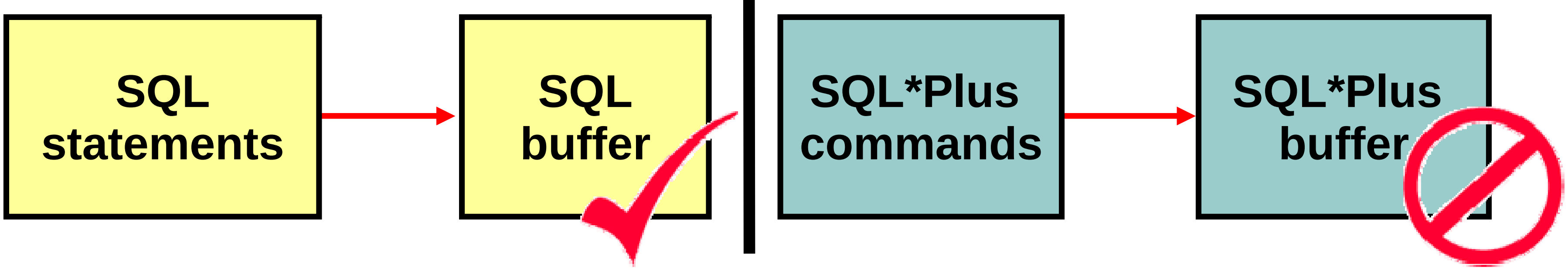
SQL Statements Versus SQL*Plus Commands

SQL

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

SQL*Plus

- An environment
- Oracle-proprietary
- Keywords can be abbreviated.
- Commands do not allow manipulation of values in the database.



Overview of SQL*Plus

- Log in to SQL*Plus.
- Describe the table structure.
- Edit your SQL statement.
- Execute SQL from SQL*Plus.
- Save SQL statements to files and append SQL statements to files.
- Execute saved files.
- Load commands from the file to buffer to edit.

Logging In to SQL*Plus

1

```
[oracle@EDRSR5P1 ~]$sqlplus

SQL*Plus: Release 11.2.0.0.2 Beta on Tue May 26 19:59:06 2009

Copyright (c) 1982, 2009, Oracle.  All rights reserved.

Enter user-name: ora21@orcl
Enter password:

Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.0.2 - Beta
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL>
```

sqlplus [*username*[/*password*[@*database*]]]

2

```
[oracle@EDRSR5P1 ~]$sqlplus ora21/ora21@orcl

SQL*Plus: Release 11.2.0.0.2 Beta on Tue May 26 19:58:06 2009

Copyright (c) 1982, 2009, Oracle.  All rights reserved.


Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.0.2 - Beta
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL>
```

ORACLE

Displaying the Table Structure

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

```
DESC[RIBE] tablename
```

Displaying the Table Structure

```
DESCRIBE departments
```

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

SQL*Plus Editing Commands

- `A[PPEND]` *`text`*
- `C[HANGE]` / *`old`* / *`new`*
- `C[HANGE]` / *`text`* /
- `CL[EAR] BUFF[ER]`
- `DEL`
- `DEL` *`n`*
- `DEL` *`m n`*

SQL*Plus Editing Commands

- `I[NPUT]`
- `I[NPUT]` *`text`*
- `L[IST]`
- `L[IST]` *`n`*
- `L[IST]` *`m n`*
- `R[UN]`
- *`n`*
- *`n text`*
- `0` *`text`*

Using `LIST`, `n`, and `APPEND`

```
LIST
  1   SELECT last_name
  2*  FROM   employees
```

```
1
  1*  SELECT last_name
```

```
A , job_id
  1*  SELECT last_name, job_id
```

```
LIST
  1   SELECT last_name, job_id
  2*  FROM   employees
```

Using the CHANGE Command

```
LIST
  1* SELECT * from employees
```

```
c/employees/departments
  1* SELECT * from departments
```

```
LIST
  1* SELECT * from departments
```

SQL*Plus File Commands

- `SAVE` *`filename`*
- `GET` *`filename`*
- `START` *`filename`*
- `@` *`filename`*
- `EDIT` *`filename`*
- `SPOOL` *`filename`*
- `EXIT`

Using the `SAVE` and `START` Commands

```
LIST
  1  SELECT last_name, manager_id, department_id
  2* FROM employees
```

```
SAVE my_query
  Created file my_query
```

```
START my_query

LAST_NAME                 MANAGER_ID DEPARTMENT_ID
------------------------- ---------- -------------
King                                            90
Kochhar                          100            90
...
107 rows selected.
```

`SERVEROUTPUT` Command

- Use the `SET SERVEROUT[PUT]` command to control whether to display the output of stored procedures or PL/SQL blocks in SQL*Plus.
- The `DBMS_OUTPUT` line length limit is increased from 255 bytes to 32767 bytes.
- The default size is now unlimited.
- Resources are not preallocated when `SERVEROUTPUT` is set.
- Because there is no performance penalty, use `UNLIMITED` unless you want to conserve physical memory.

```
SET SERVEROUT[PUT] {ON | OFF} [SIZE {n | UNL[IMITED]}]
    [FOR[MAT] {WRA[PPED] | WOR[D_WRAPPED] | TRU[NCATED]}]
```

Using the SQL*Plus SPOOL Command

```
SPO[OL] [file_name[.ext] [CRE[ATE] | REP[LACE] |
    APP[END]] | OFF | OUT]
```

Option	Description
file_name[.ext]	Spools output to the specified file name
CRE[ATE]	Creates a new file with the name specified
REP[LACE]	Replaces the contents of an existing file. If the file does not exist, REPLACE creates the file.
APP[END]	Adds the contents of the buffer to the end of the file you specify
OFF	Stops spooling
OUT	Stops spooling and sends the file to your computer's standard (default) printer

ORACLE

Using the `AUTOTRACE` Command

- It displays a report after the successful execution of SQL data manipulation statements (DML) statements such as `SELECT`, `INSERT`, `UPDATE`, or `DELETE`.
- The report can now include execution statistics and the query execution path.

```
SET AUTOT[RACE] {ON | OFF | TRACE[ONLY]} [EXP[LAIN]]
    [STAT[ISTICS]]
```

```
SET AUTOTRACE ON
-- The AUTOTRACE report includes both the optimizer
-- execution path and the SQL statement execution
-- statistics
```

ORACLE

Summary

In this appendix, you should have learned how to use SQL*Plus as an environment to do the following:

- Execute SQL statements
- Edit SQL statements
- Format the output
- Interact with script files

E

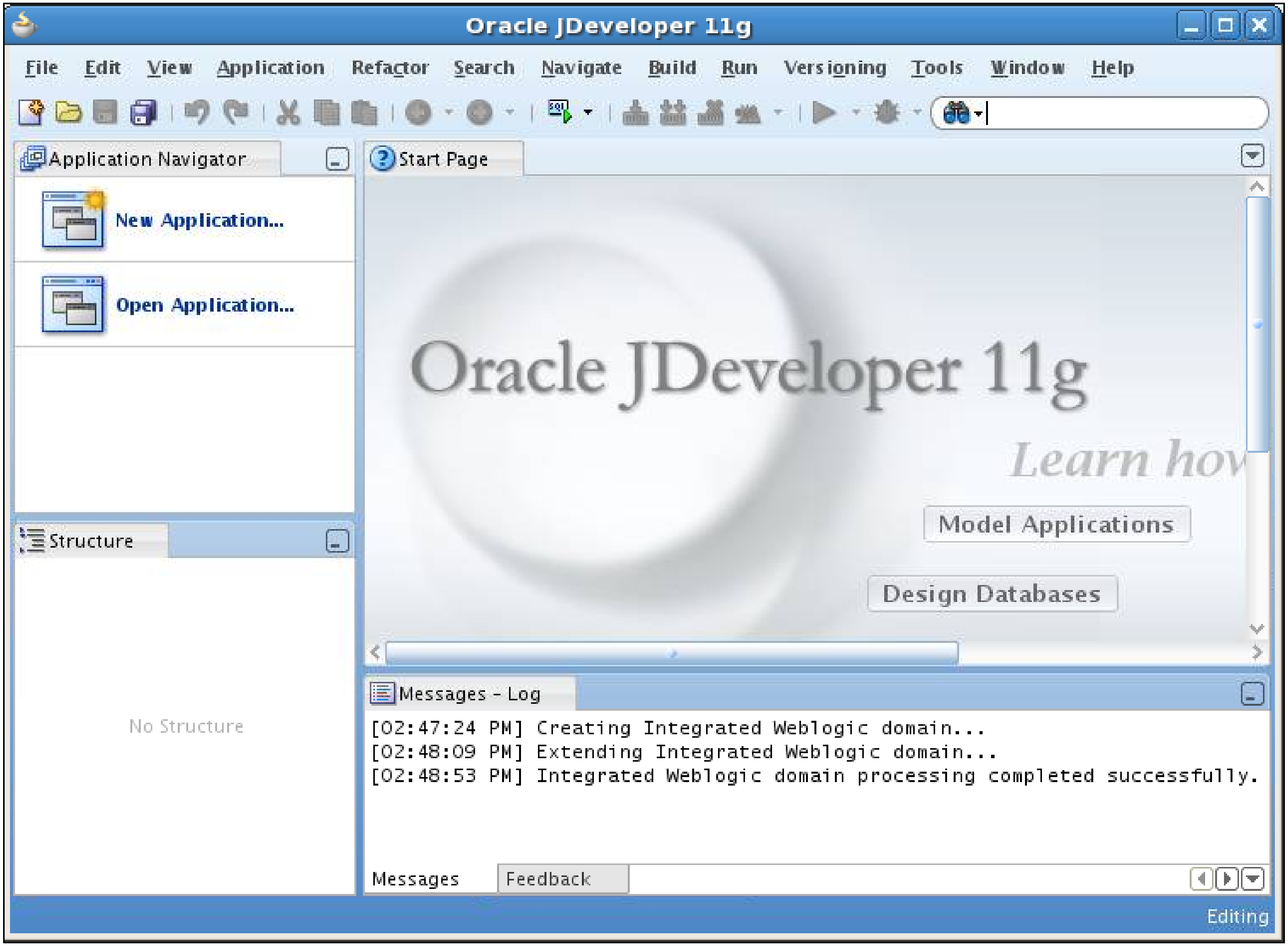
Using JDeveloper

Objectives

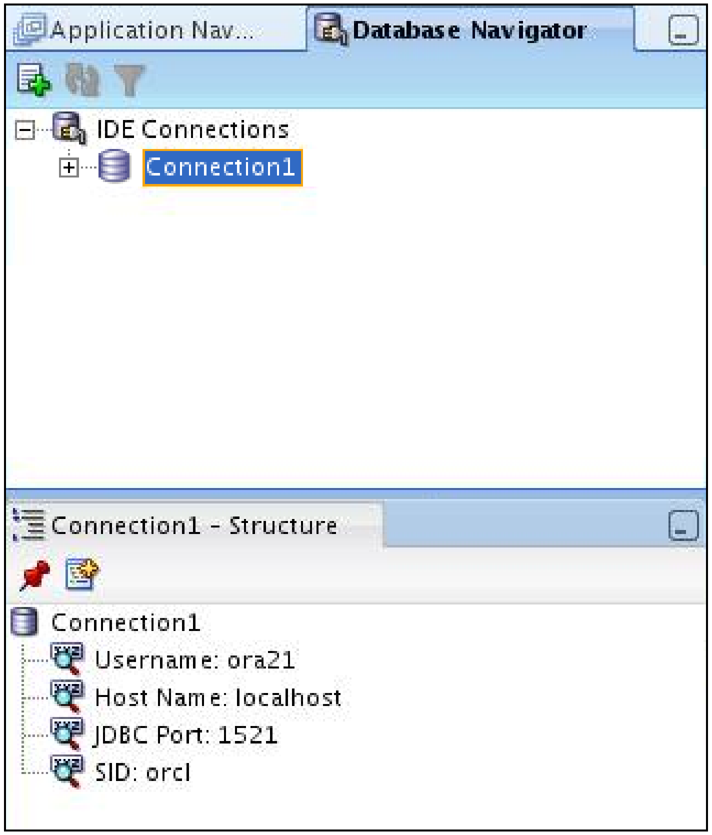
After completing this appendix, you should be able to do the following:

- List the key features of Oracle JDeveloper
- Create a database connection in JDeveloper
- Manage database objects in Jdeveloper
- Use JDeveloper to execute SQL Commands
- Create and run PL/SQL Program Units

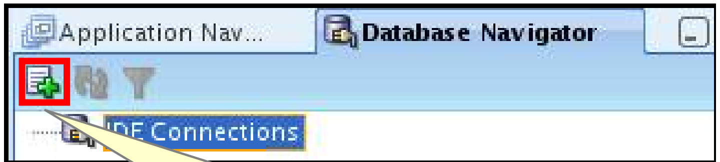
Oracle JDeveloper



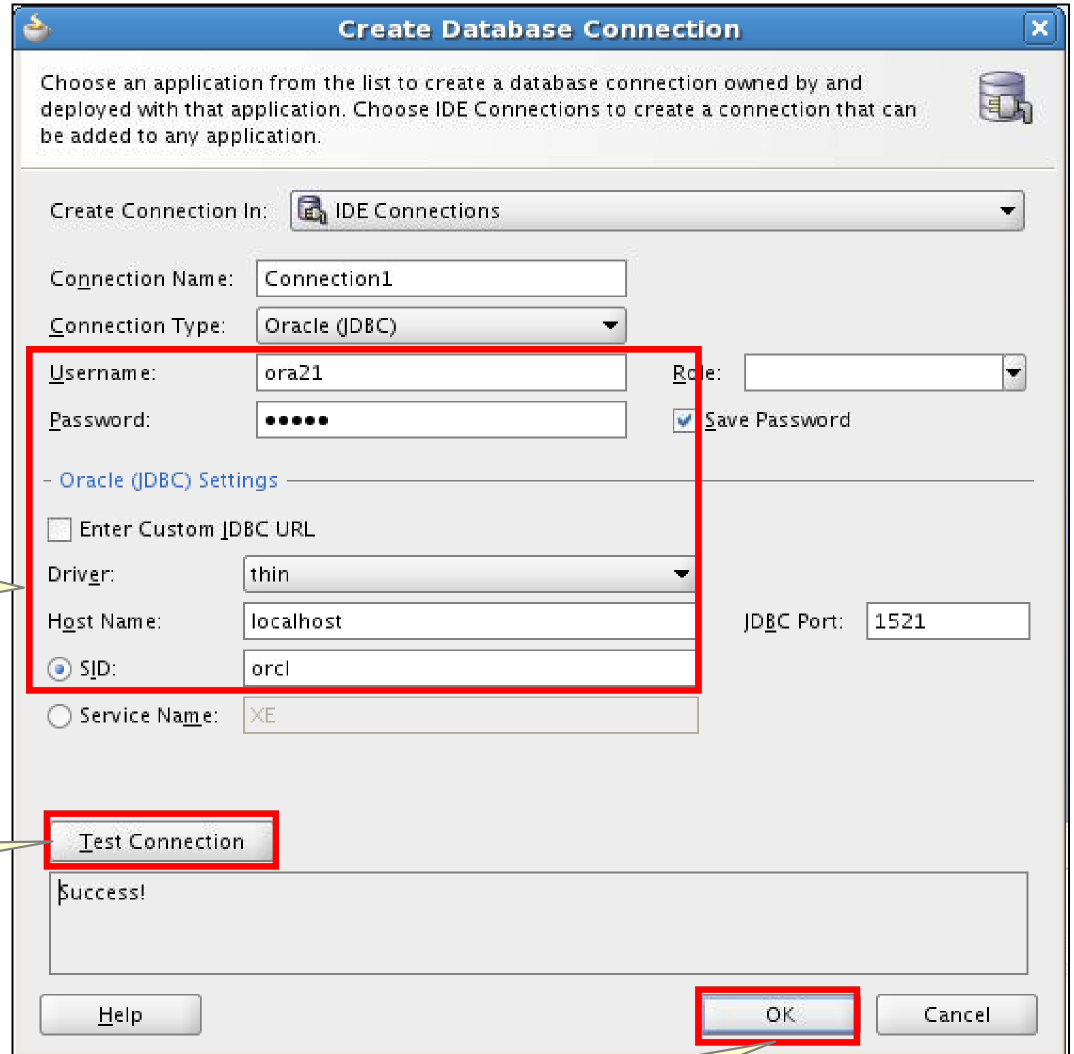
Database Navigator



Creating a Connection



1 Click the New Connection icon in the Database Navigator.



2 In the Create Database Connection window, enter the username, password and the SID.

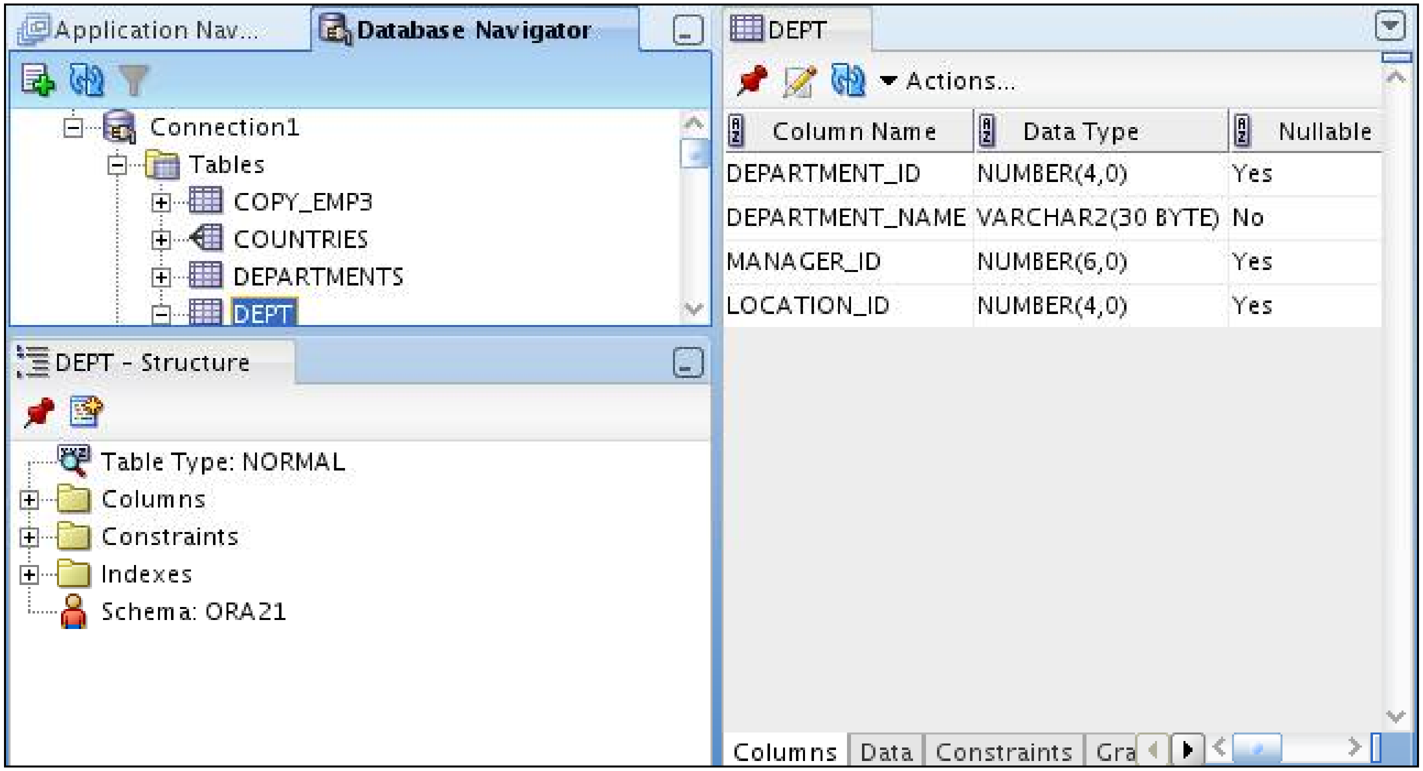
3 Test the connection.

4 Click OK.

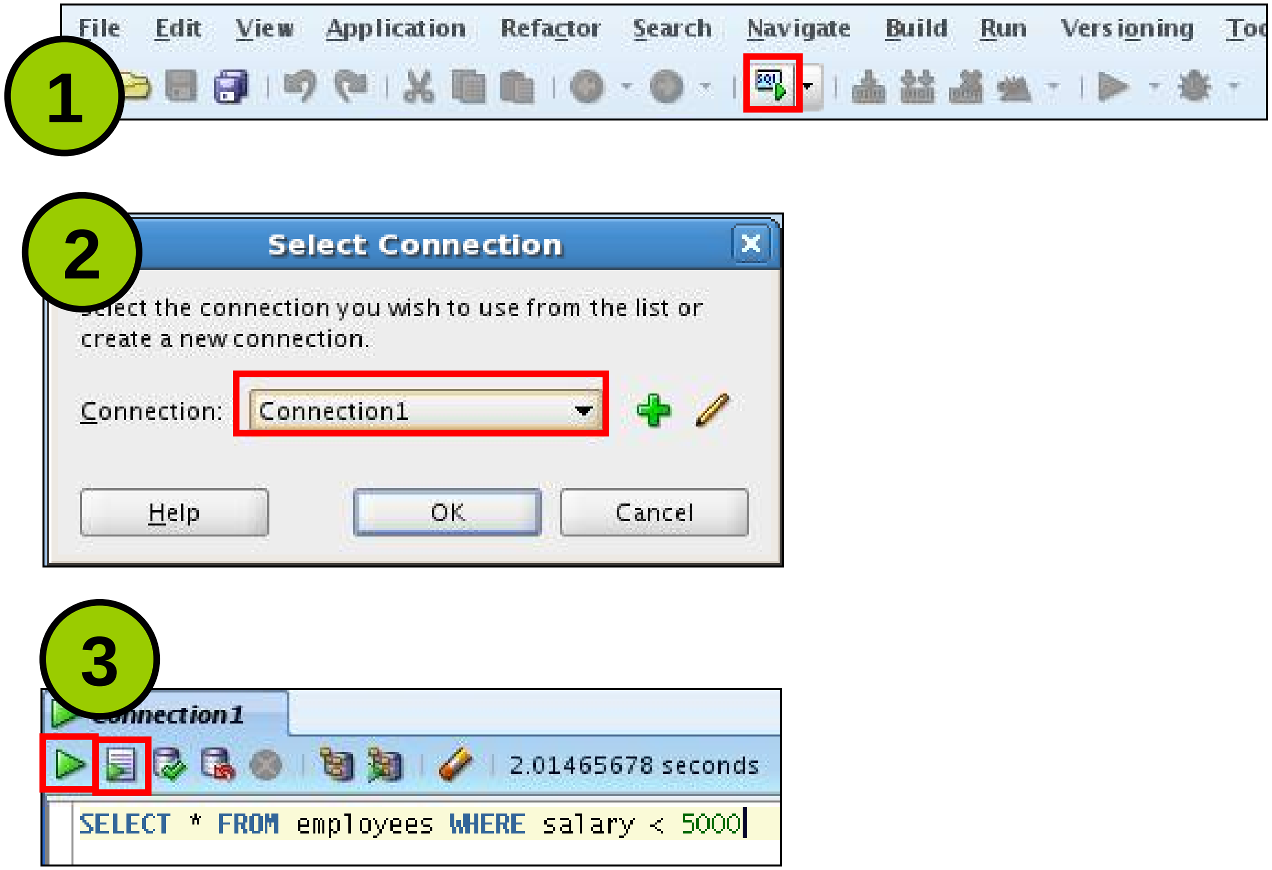
Browsing Database Objects

Use the Database Navigator to:

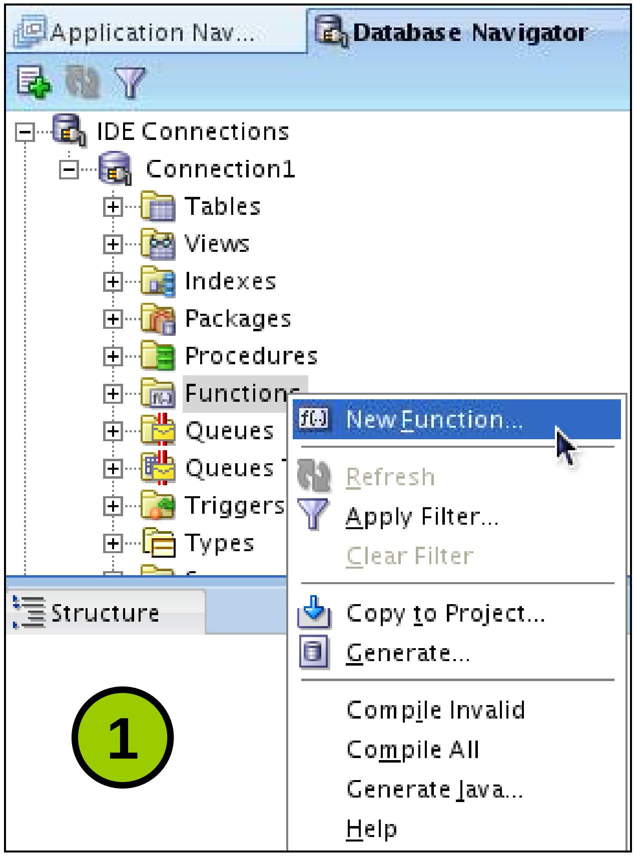
- Browse through many objects in a database schema
- Review the definitions of objects at a glance

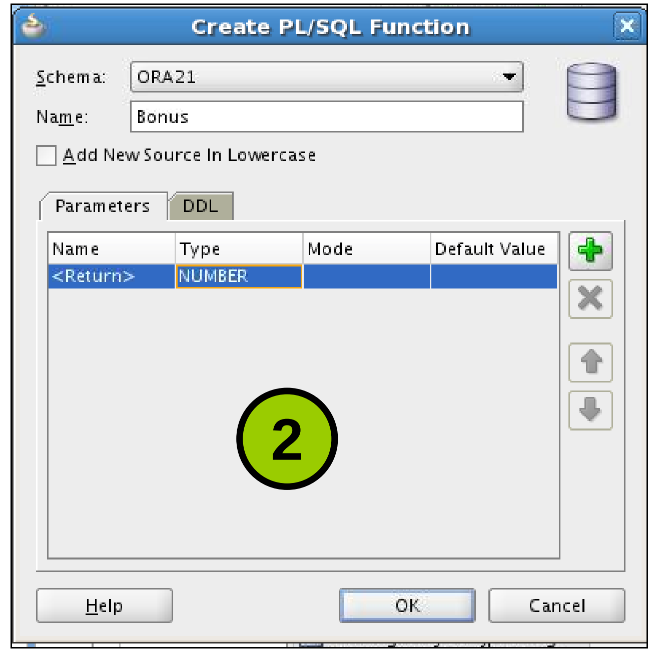


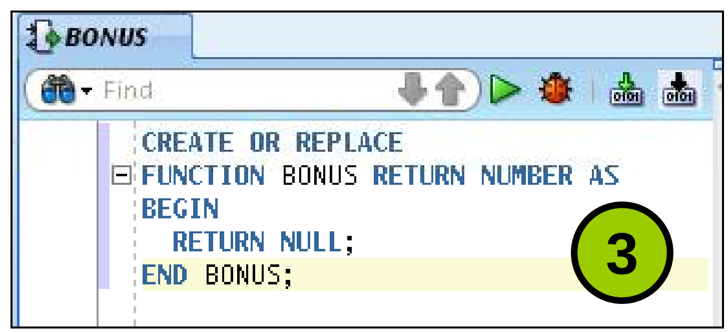
Executing SQL Statements



Creating Program Units

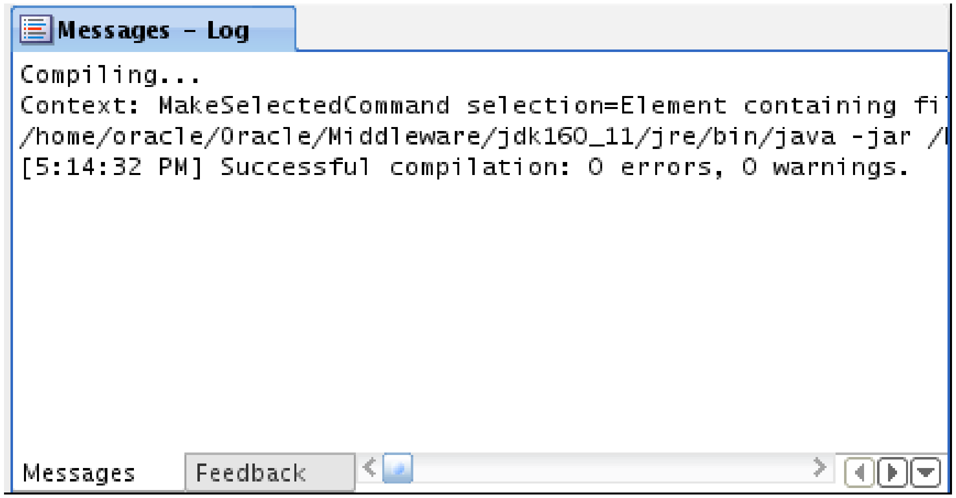




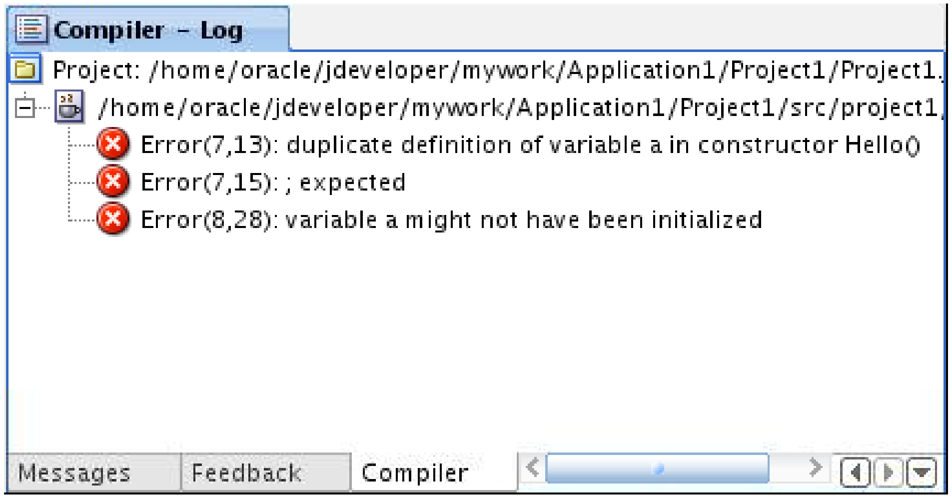


Skeleton of the function

Compiling

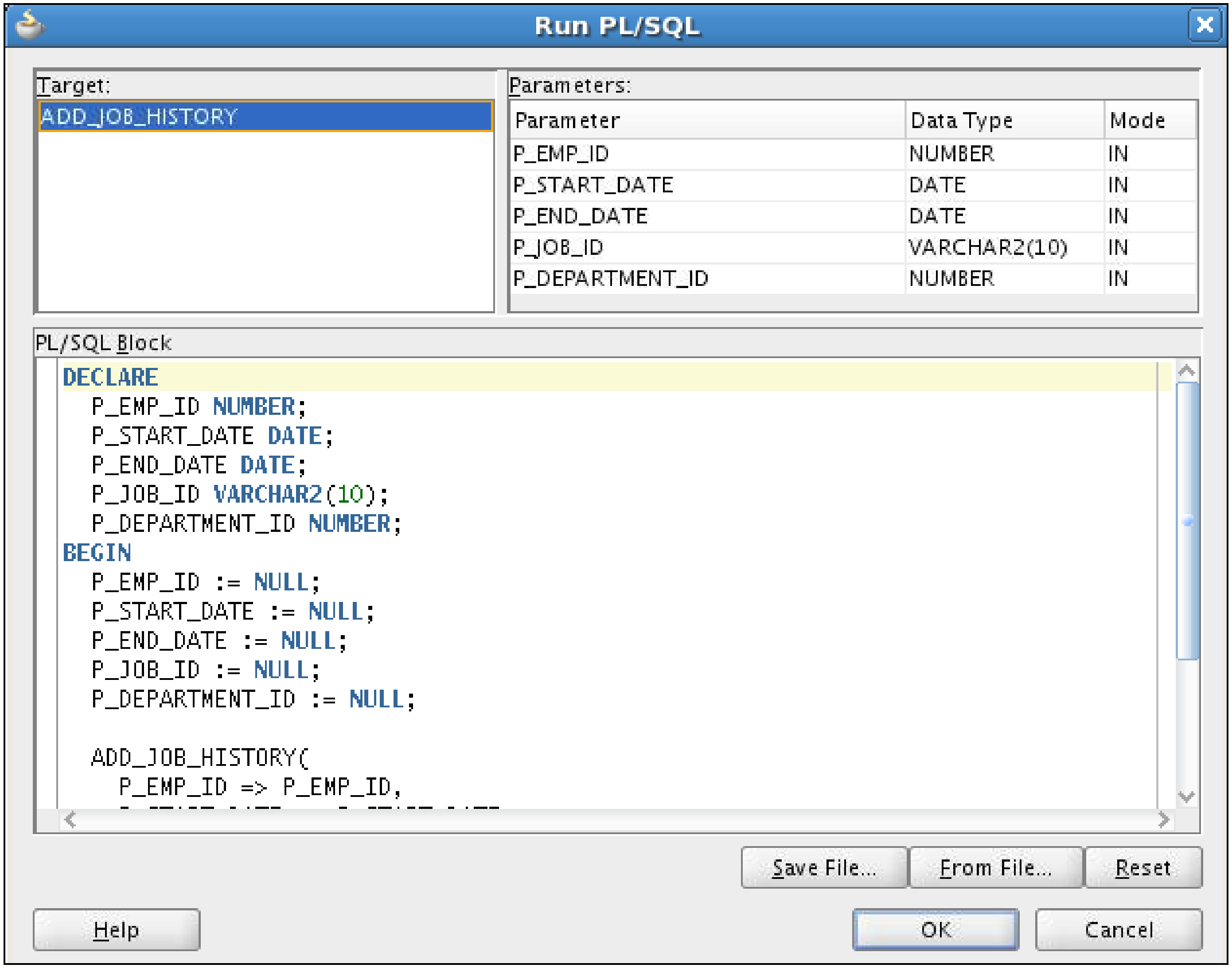


Compilation with errors

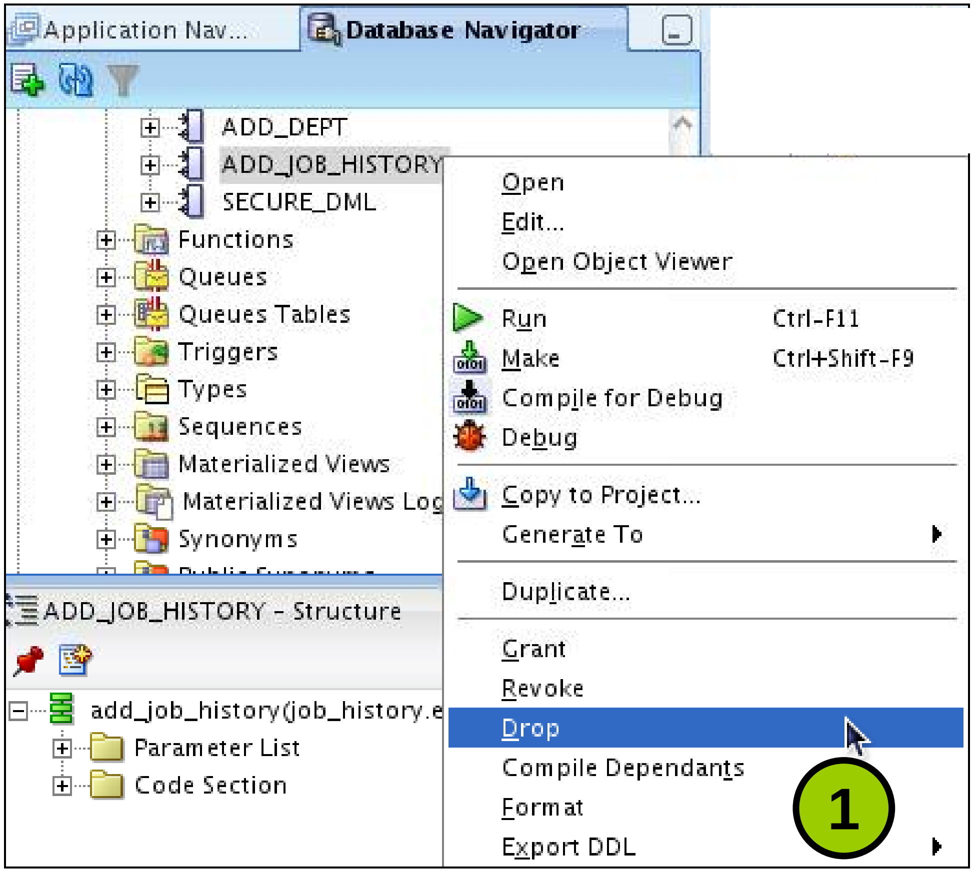


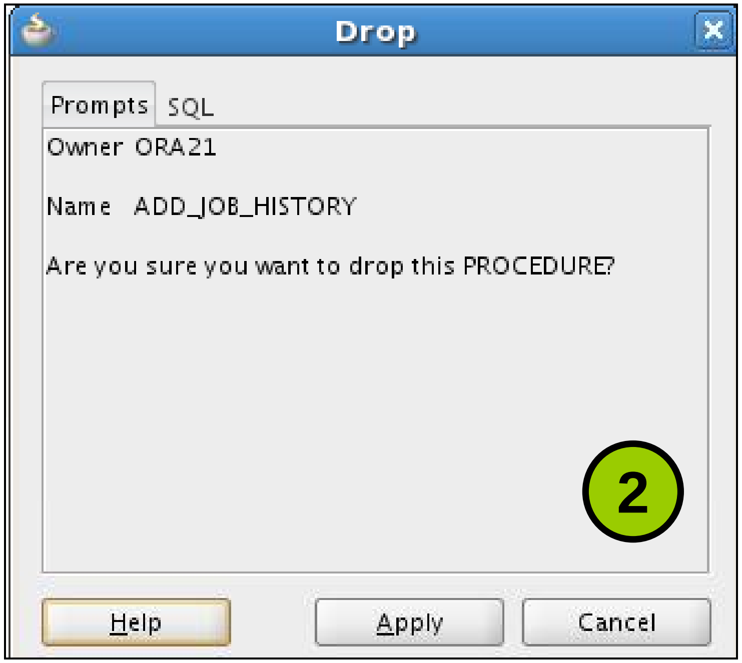
Compilation without errors

Running a Program Unit

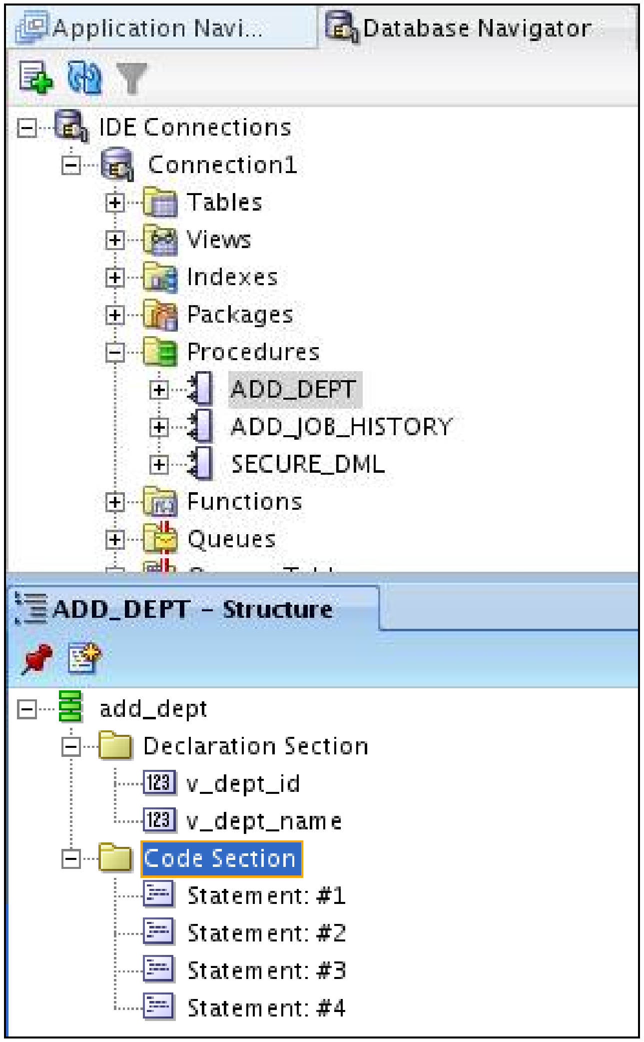


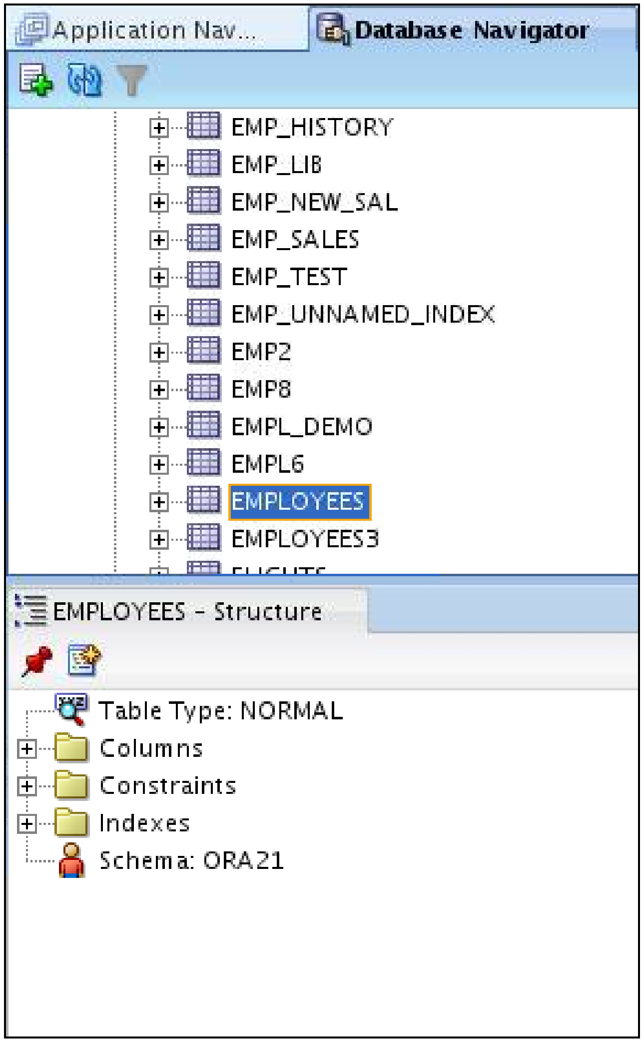
Dropping a Program Unit



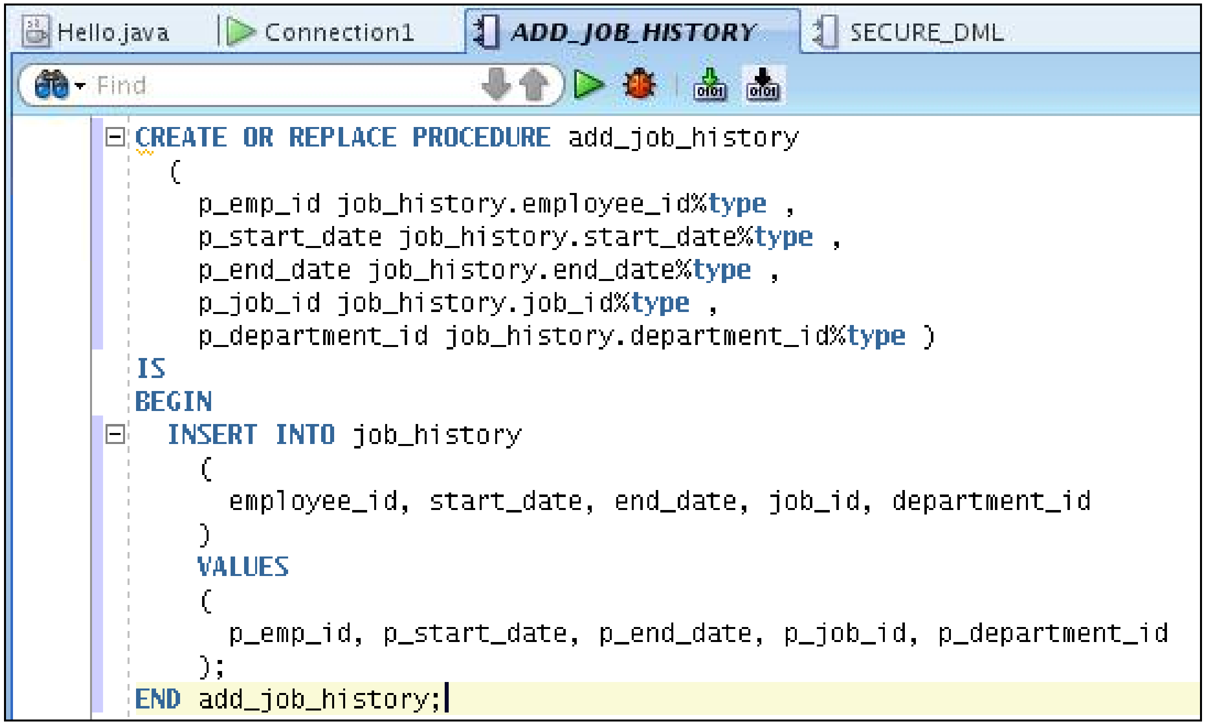


Structure Window

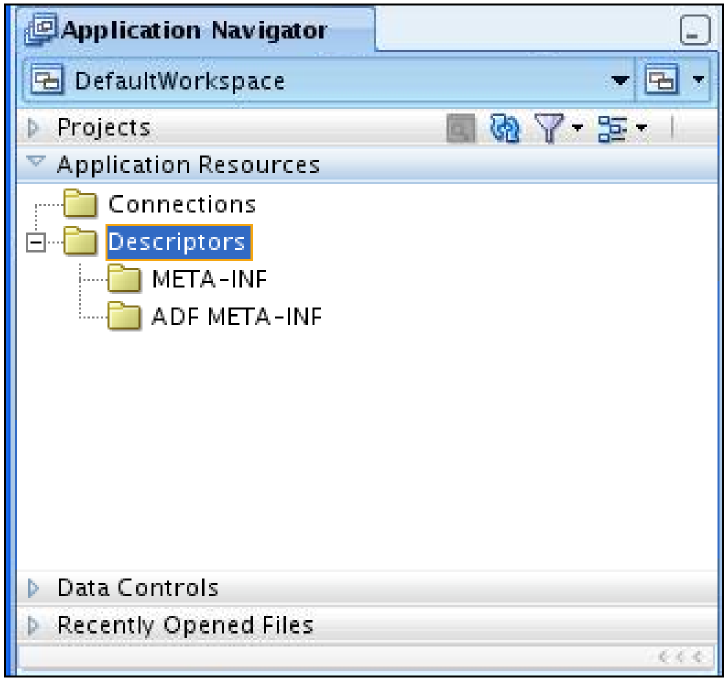




Editor Window



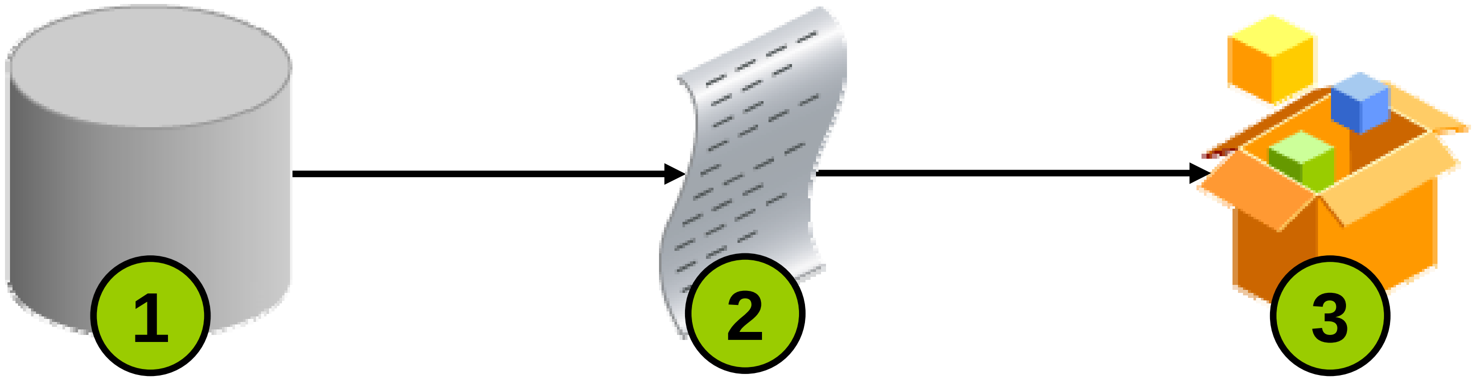
Application Navigator



Deploying Java Stored Procedures

Before deploying Java stored procedures, perform the following steps:

1. Create a database connection.
2. Create a deployment profile.
3. Deploy the objects.



Publishing Java to PL/SQL

TrimLob.java | TRIMLOBPROC

```
public class TrimLob
{
  public static void main (String args []) throws SQLException {
  Connection conn=null;
   if (System.getProperty("oracle.jserver.version") != null)
  {
   conn = DriverManager.getConnection("jdbc:default:connection:");
  }
  else
  {
  DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
   conn = DriverManager.getConnection("jdbc:oracle:thin:scott/tiger");
  }
```

1

TrimLob.java | TRIMLOBPROC

```
CREATE OR REPLACE PROCEDURE TRIMLOBPROC
as language java
  name 'TrimLob.main(java.lang.String[])';
/
```

2

How Can I Learn More About JDeveloper 11*g* ?

Topic	Web site
Oracle JDeveloper Product Page	http://www.oracle.com/technology/products/jdev/index.html
Oracle JDeveloper 11*g* Tutorials	http://www.oracle.com/technology/obe/obe11jdev/11/index.html
Oracle JDeveloper 11*g* Product Documentation	http://www.oracle.com/technology/documentation/jdev.html
Oracle JDeveloper 11*g* Discussion Forum	http://forums.oracle.com/forums/forum.jspa?forumID=83

Summary

In this appendix, you should have learned how to use JDeveloper to do the following:

- List the key features of Oracle JDeveloper
- Create a database connection in JDeveloper
- Manage database objects in JDeveloper
- Use JDeveloper to execute SQL Commands
- Create and run PL/SQL Program Units

F

Generating Reports by Grouping Related Data

Objectives

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

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

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%';
```

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 ;
```

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];
```

ORACLE

`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.

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: Example

```
SELECT    department_id, job_id, SUM(salary)
FROM      employees
WHERE     department_id < 60
GROUP BY ROLLUP(department_id, job_id);
```

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	10	AD_ASST	4400
2	10	(null)	4400
3	20	MK_MAN	13000
4	20	MK_REP	6000
5	20	(null)	19000
6	30	PU_MAN	11000
7	30	PU_CLERK	13900
8	30	(null)	24900
9	40	HR_REP	6500
10	40	(null)	6500
11	50	ST_MAN	36400
12	50	SH_CLERK	64300
13	50	ST_CLERK	55700
14	50	(null)	156400
15	(null)	(null)	211200

1

2

3

ORACLE

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: Example

```
SELECT    department_id, job_id, SUM(salary)
FROM      employees
WHERE     department_id < 60
GROUP BY CUBE (department_id, job_id) ;
```

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	(null)	(null)	211200
2	(null)	HR_REP	6500
3	(null)	MK_MAN	13000
4	(null)	MK_REP	6000
5	(null)	PU_MAN	11000
6	(null)	ST_MAN	36400
7	(null)	AD_ASST	4400
8	(null)	PU_CLERK	13900
9	(null)	SH_CLERK	64300
10	(null)	ST_CLERK	55700
11	10	(null)	4400
12	10	AD_ASST	4400
13	20	(null)	19000
14	20	MK_MAN	13000
15	20	MK_REP	6000
16	30	(null)	24900

1

2

3

4

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];
```

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

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: Example

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

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
1	(null)	SH_CLERK	122	3200
2	(null)	AC_MGR	101	12000
3	(null)	ST_MAN	100	7280
4	(null)	ST_CLERK	121	2675

…

1

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
39	110	AC_MGR	(null)	12000
40	90	AD_PRES	(null)	24000
41	60	IT_PROG	(null)	5760
42	100	FI_MGR	(null)	12000

2

…

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: Example

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

1 2

	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

...

3 4

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
40	100	FI_MGR	101	12000
41	100	FI_ACCOUNT	108	39600
42	100	(null)	(null)	51600
43	110	AC_MGR	101	12000
44	110	AC_ACCOUNT	205	8300
45	110	(null)	(null)	20300
46	(null)	(null)	(null)	691400

Concatenated Groupings

- Concatenated groupings offer a concise way to generate useful combinations of groupings.
- To specify concatenated grouping sets, you separate multiple grouping sets, `ROLLUP` and `CUBE` operations with commas so that the Oracle server combines them into a single `GROUP BY` clause.
- The result is a cross-product of groupings from each `GROUPING SET`.

```
GROUP BY GROUPING SETS(a, b), GROUPING SETS(c, d)
```

Concatenated Groupings: Example

```
SELECT    department_id, job_id, manager_id,
          SUM(salary)
FROM      employees
GROUP BY department_id,
          ROLLUP(job_id),
          CUBE(manager_id);
```

	DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
1	(null)	SA_REP	149	7000
2	10	AD_ASST	101	4400
3	20	MK_MAN	100	13000
4	20	MK_REP	201	6000
...				
	90	AD_VP	100	34000
	90	AD_PRES	(null)	24000
...				
	(null)	SA_REP	(null)	7000
	10	AD_ASST	(null)	4400
...				
	110	(null)	101	12000
	110	(null)	205	8300
	110	(null)	(null)	20300

ORACLE

Summary

In this appendix, you should have learned how to use the:

- `ROLLUP` operation to produce subtotal values
- `CUBE` operation to produce cross-tabulation values
- `GROUPING` function to identify the row values created by `ROLLUP` or `CUBE`
- `GROUPING SETS` syntax to define multiple groupings in the same query
- `GROUP BY` clause to combine expressions in various ways:
  - Composite columns
  - Concatenated grouping sets

Hierarchical Retrieval

Objectives

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

- Interpret the concept of a hierarchical query
- Create a tree-structured report
- Format hierarchical data
- Exclude branches from the tree structure

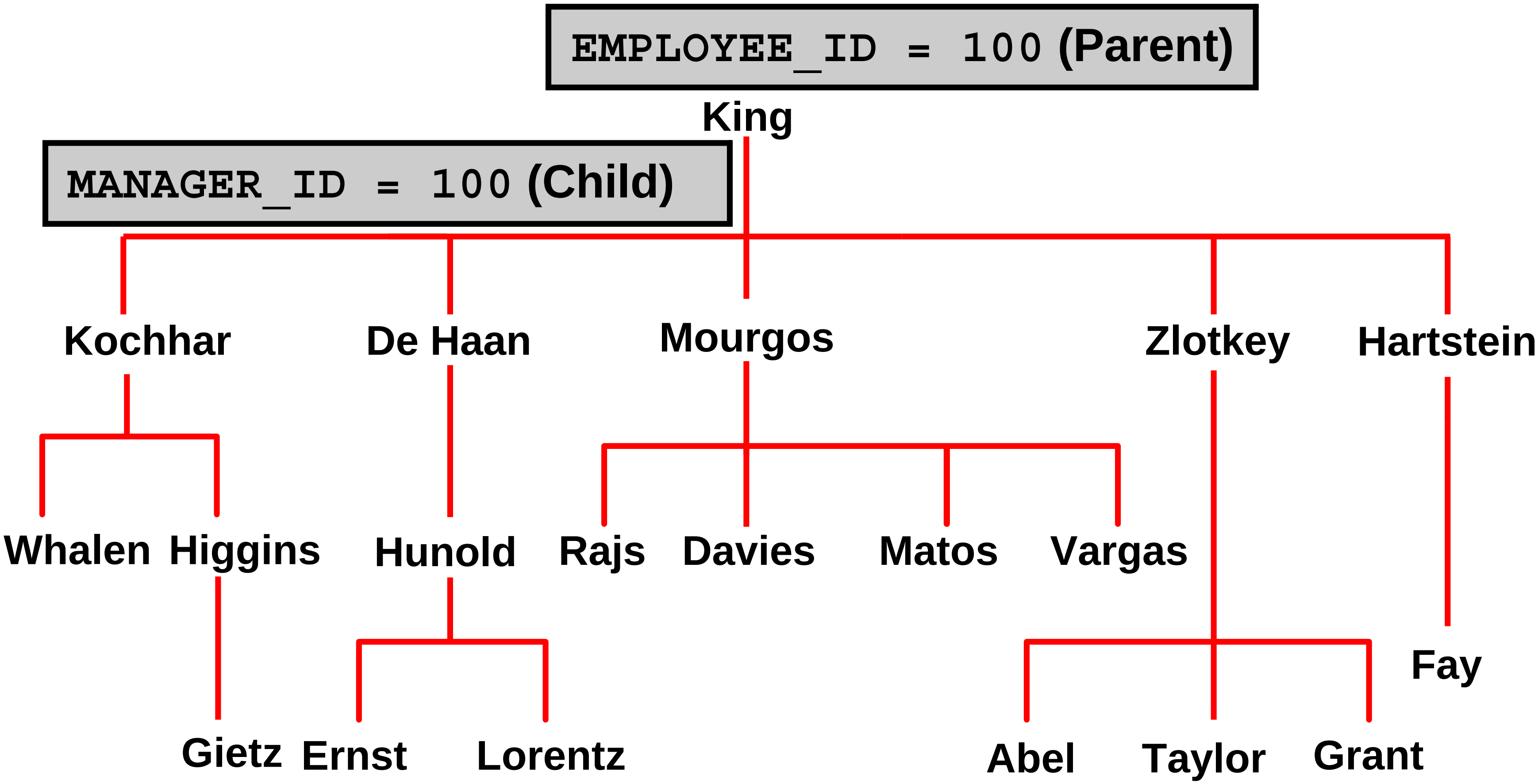
Sample Data from the EMPLOYEES Table

	EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
1	100	King	AD_PRES	(null)
2	101	Kochhar	AD_VP	100
3	102	De Haan	AD_VP	100
4	103	Hunold	IT_PROG	102
5	104	Ernst	IT_PROG	103
6	107	Lorentz	IT_PROG	103

...

16	200	Whalen	AD_ASST	101
17	201	Hartstein	MK_MAN	100
18	202	Fay	MK_REP	201
19	205	Higgins	AC_MGR	101
20	206	Gietz	AC_ACCOUNT	205

Natural Tree Structure



Hierarchical Queries

```
SELECT [LEVEL], column, expr...
FROM   table
[WHERE condition(s)]
[START WITH condition(s)]
[CONNECT BY PRIOR condition(s)] ;
```

condition:

```
expr comparison_operator expr
```

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

```
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: 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)

Walking the Tree: From the Top Down

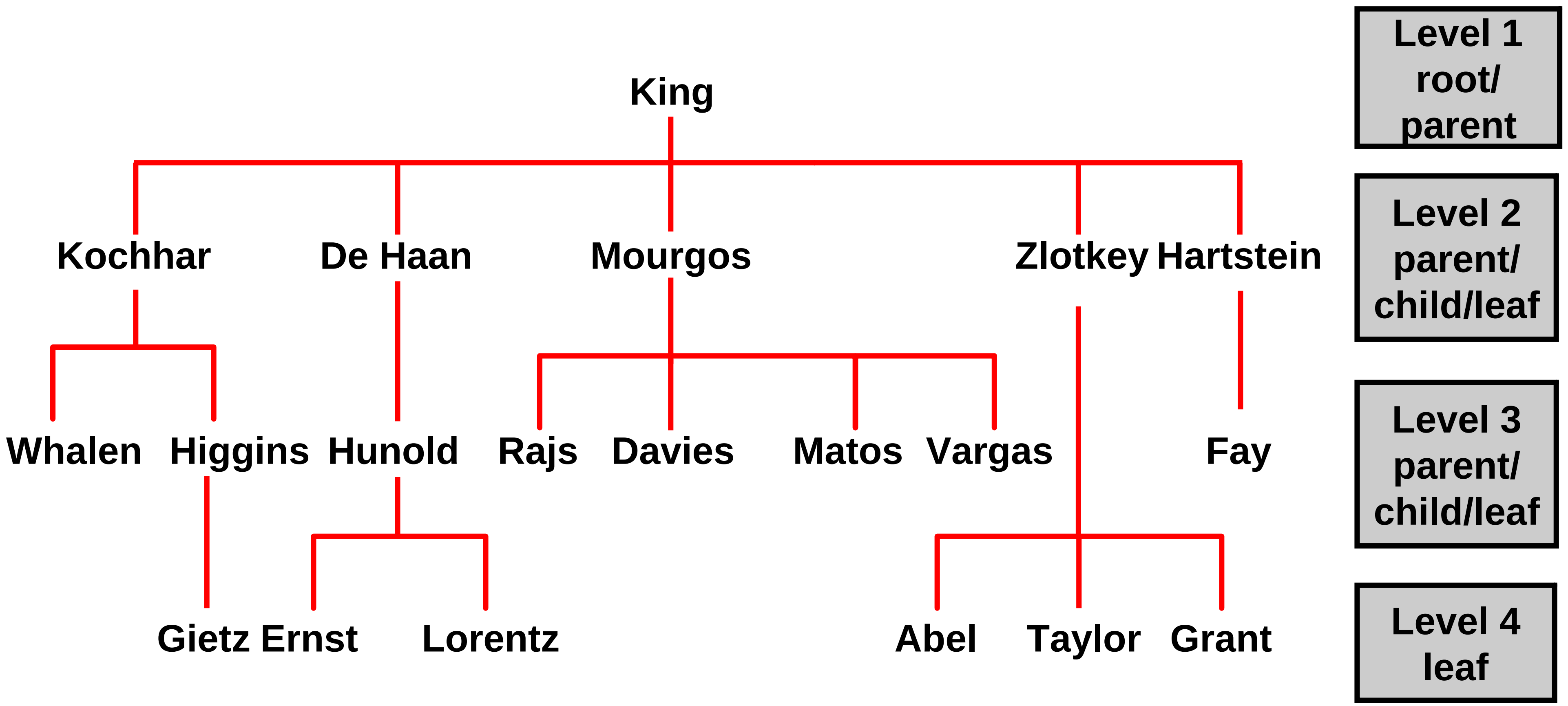
```
SELECT  last_name||' reports to '||
PRIOR   last_name "Walk Top Down"
FROM    employees
START   WITH last_name = 'King'
CONNECT BY PRIOR employee_id = manager_id ;
```

	Walk Top Down
1	King reports to
2	King reports to
3	Kochhar reports to King
4	Greenberg reports to Kochhar
5	Faviet reports to Greenberg

...

105	Grant reports to Zlotkey
106	Johnson reports to Zlotkey
107	Hartstein reports to King
108	Fay reports to Hartstein

Ranking Rows with the `LEVEL` Pseudocolumn



Formatting Hierarchical Reports Using `LEVEL` and `LPAD`

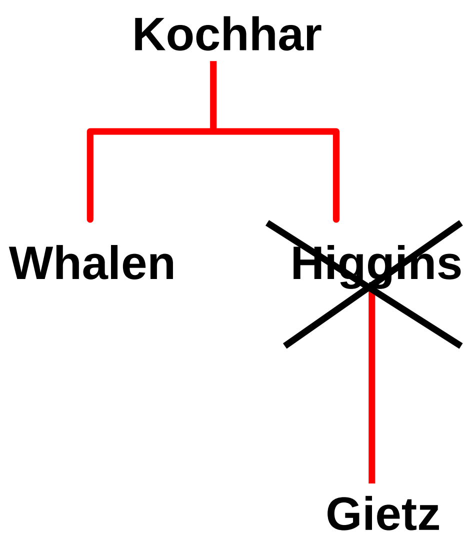
Create a report displaying company management levels, beginning with the highest level and indenting each of the following levels.

```
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
```

Pruning Branches

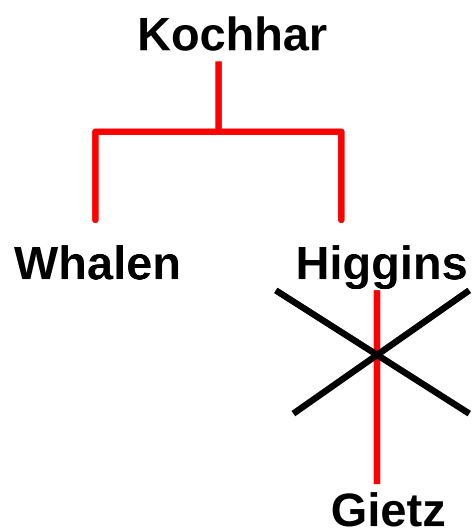
Use the `WHERE` clause to eliminate a node.

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



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

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



Summary

In this appendix, you should have learned that you can:

- Use hierarchical queries to view a hierarchical relationship between rows in a table
- Specify the direction and starting point of the query
- Eliminate nodes or branches by pruning

H

Writing Advanced Scripts

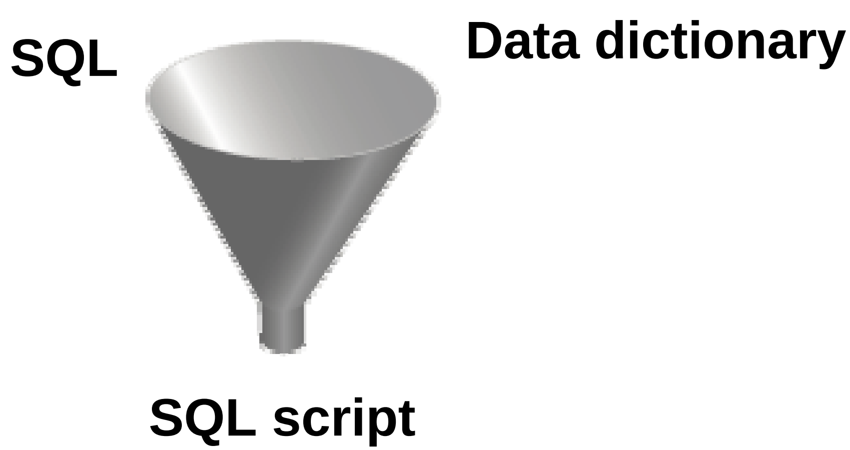
Objectives

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

Using SQL to Generate SQL

- SQL can be used to generate scripts in SQL.
- The data dictionary is:
  - A collection of tables and views that contain database information
  - Created and maintained by the Oracle server



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;

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.

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
```

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 the Contents of a Table to a File

Source	Result
`'''X'''`	`'X'`
`''''`	`'`
`''''||department_name||''''`	`'Administration'`
`''', '''`	`','`
`''');'`	`');`

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;
```

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

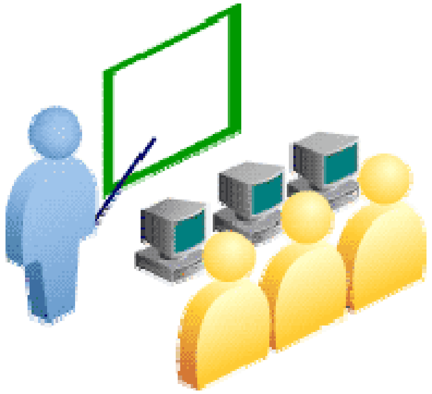
I

Oracle Database Architectural Components

Objectives

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

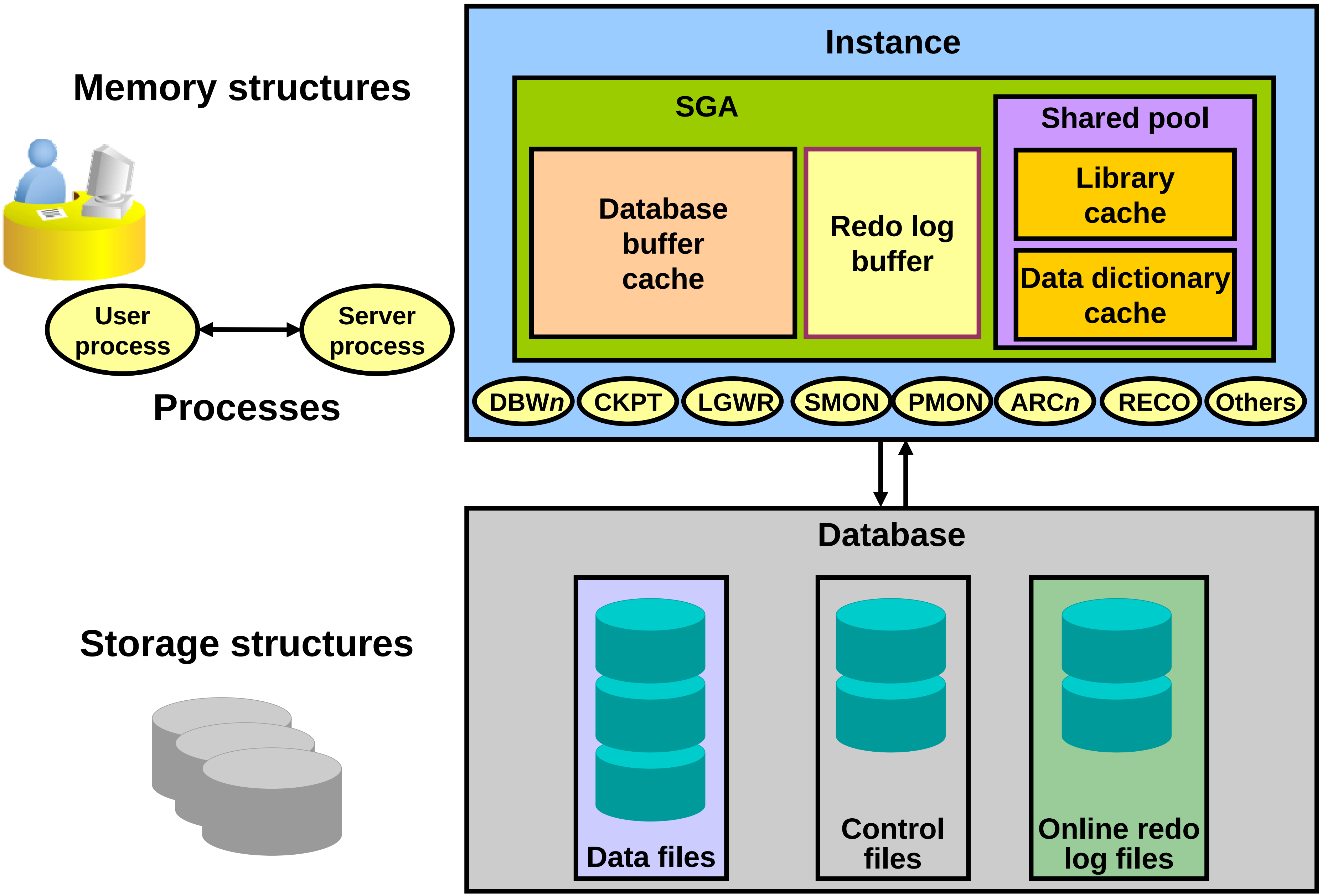
- List the major database architectural components
- Describe the background processes
- Explain the memory structures
- Correlate the logical and physical storage structures



Oracle Database Architecture: Overview

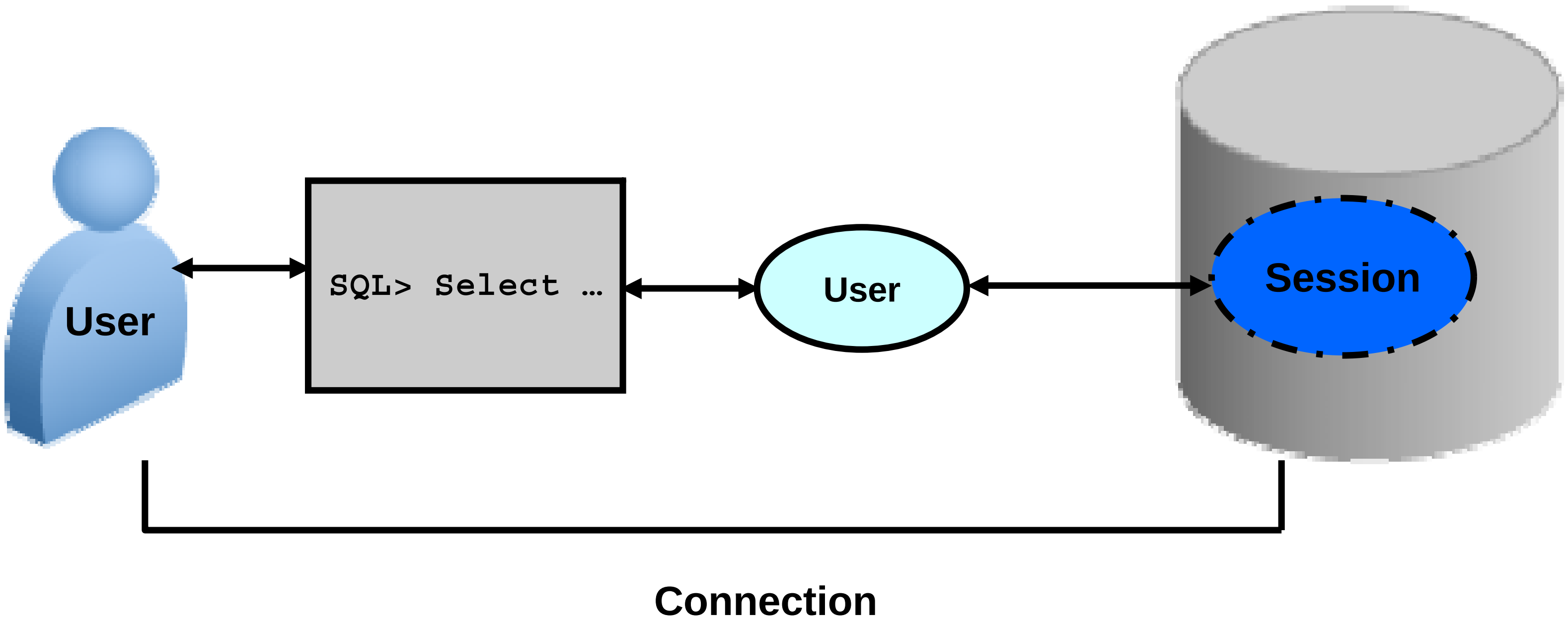
The Oracle Relational Database Management System (RDBMS) is a database management system that provides an open, comprehensive, integrated approach to information management.

Oracle Database Server Structures

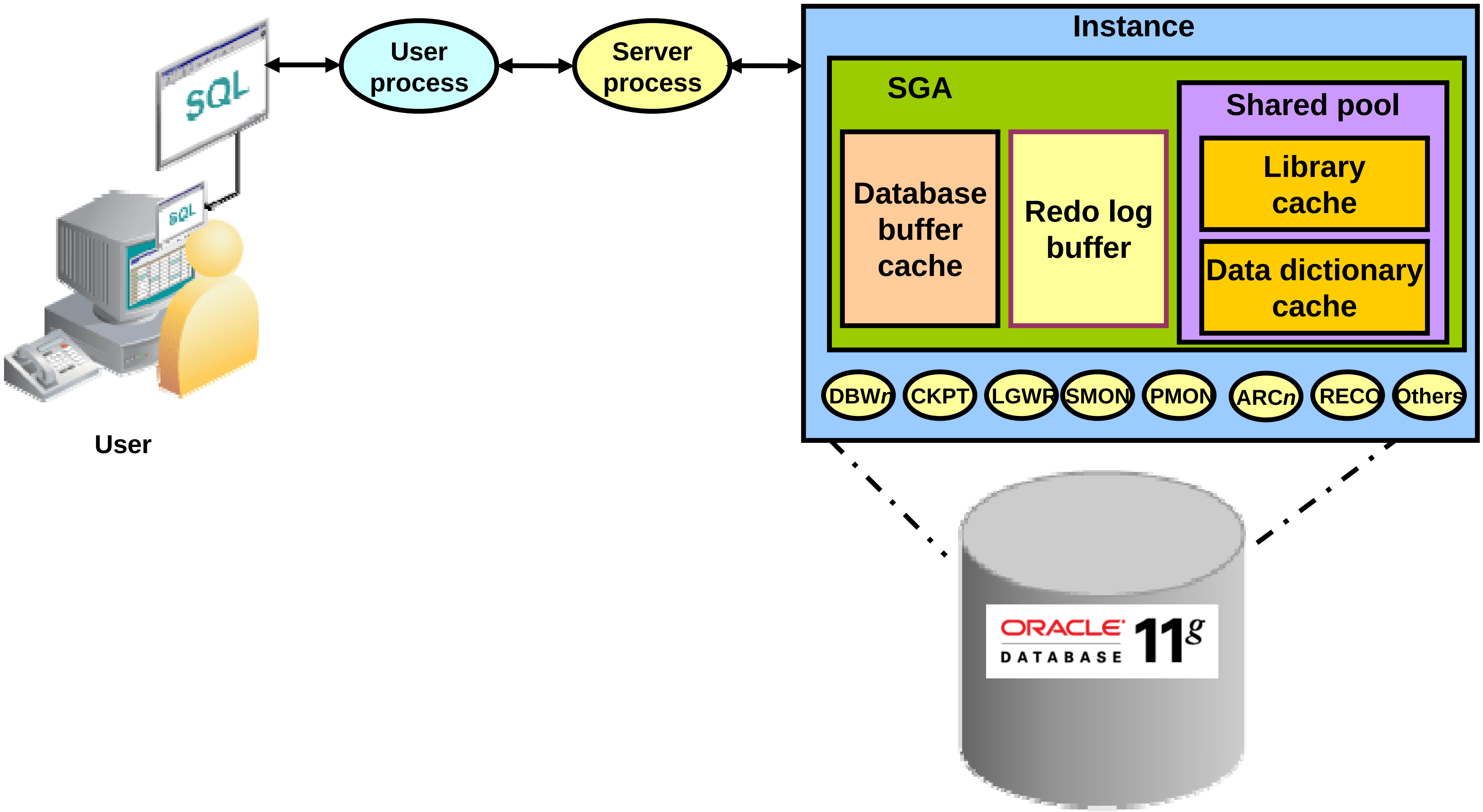


Connecting to the Database

- Connection: Communication pathway between a user process and a database instance
- Session: A specific connection of a user to a database instance through a user process

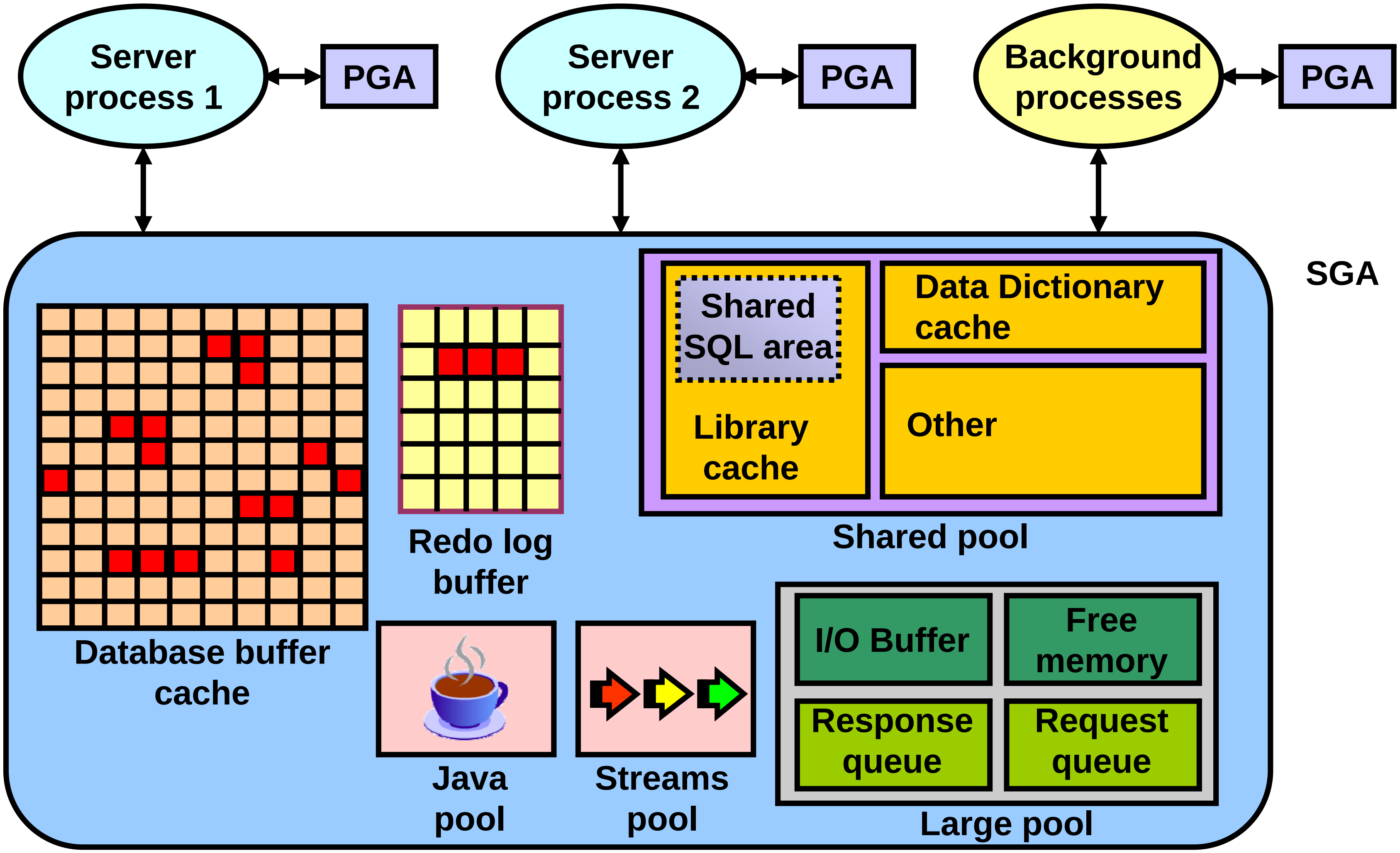


Interacting with an Oracle Database



Oracle Memory Architecture

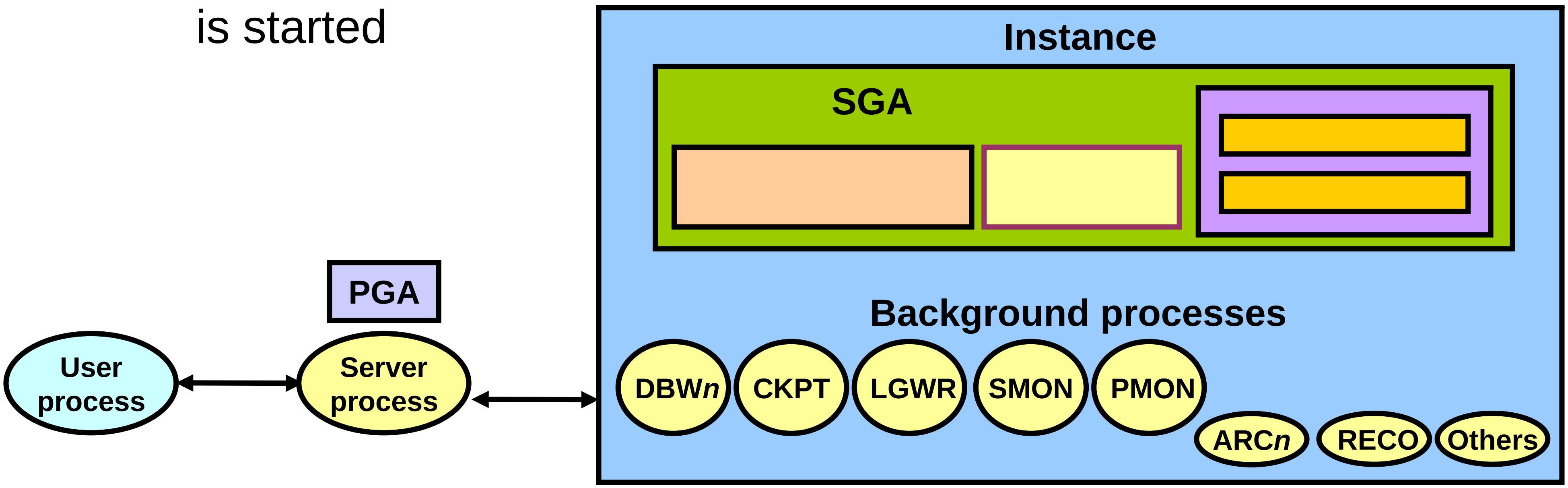
DB structures
→Memory
- Process
- Storage



Process Architecture

DB structures
- Memory
→ Process
- Storage

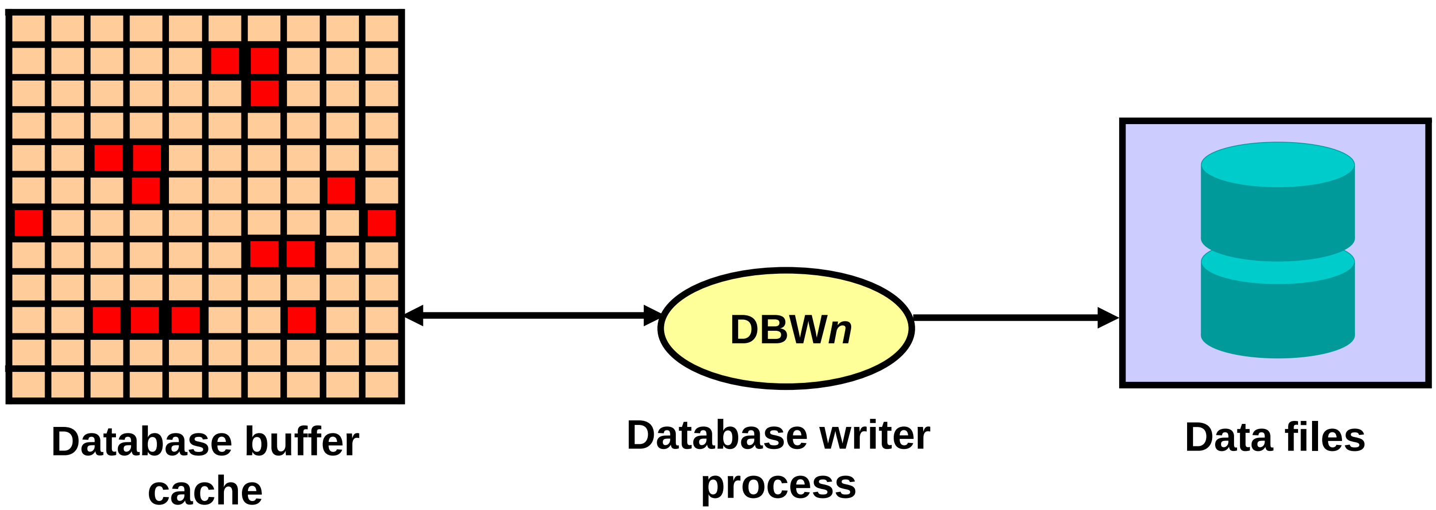
- User process:
  - Is started when a database user or a batch process connects to the Oracle Database
- Database processes:
  - Server process: Connects to the Oracle instance and is started when a user establishes a session
  - Background processes: Are started when an Oracle instance is started



Database Writer Process

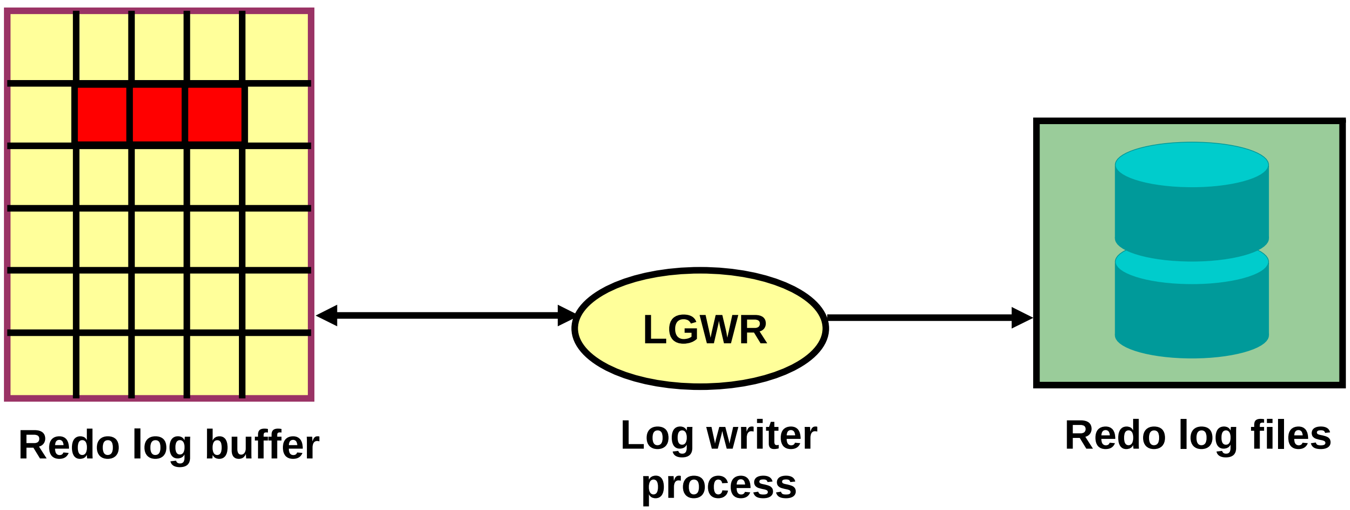
Writes modified (dirty) buffers in the database buffer cache to disk:

- Asynchronously while performing other processing
- Periodically to advance the checkpoint



Log Writer Process

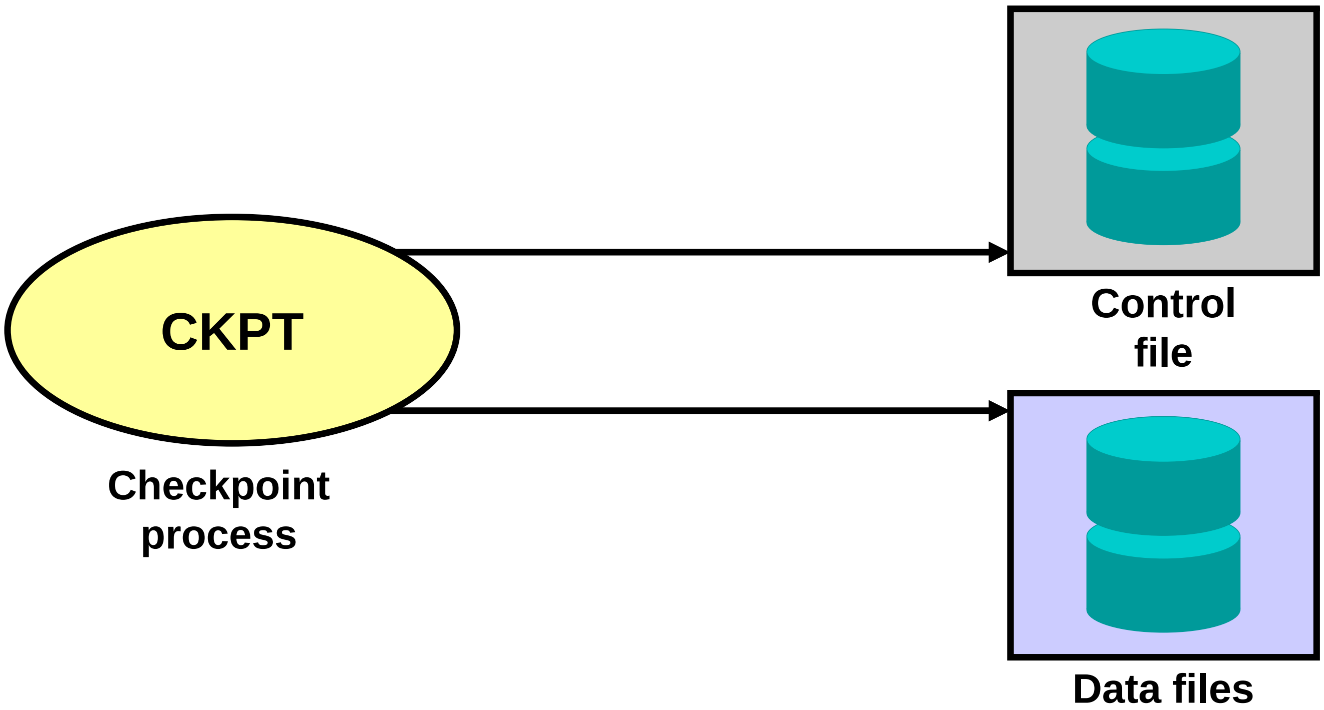
- Writes the redo log buffer to a redo log file on disk
- LGWR writes:
  - A process commits a transaction
  - When the redo log buffer is one-third full
  - Before a DBW*n* process writes modified buffers to disk



Checkpoint Process

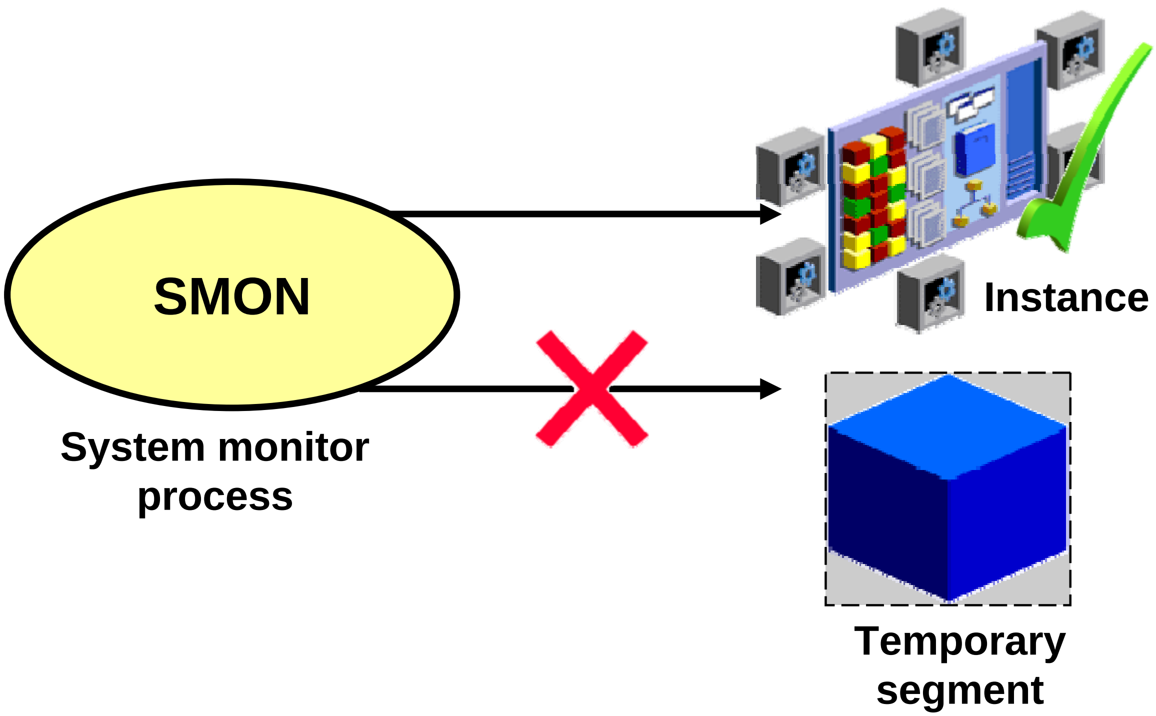
Records checkpoint information in:

- The control file
- Each datafile header



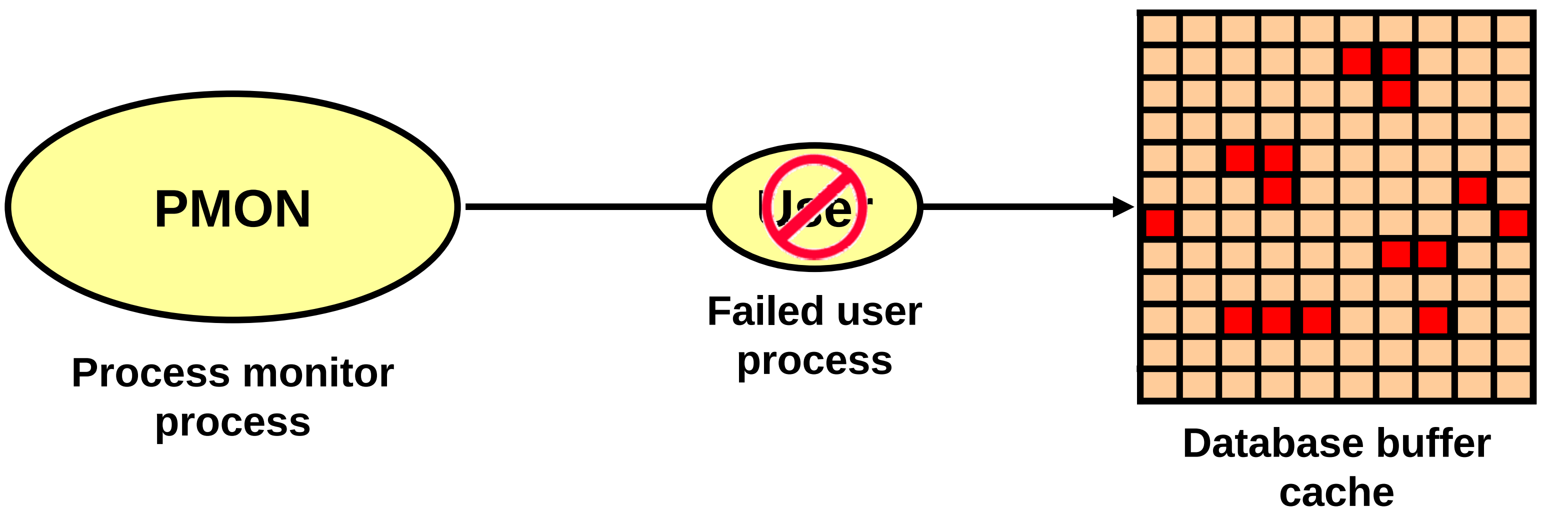
System Monitor Process

- Performs recovery at instance startup
- Cleans up unused temporary segments



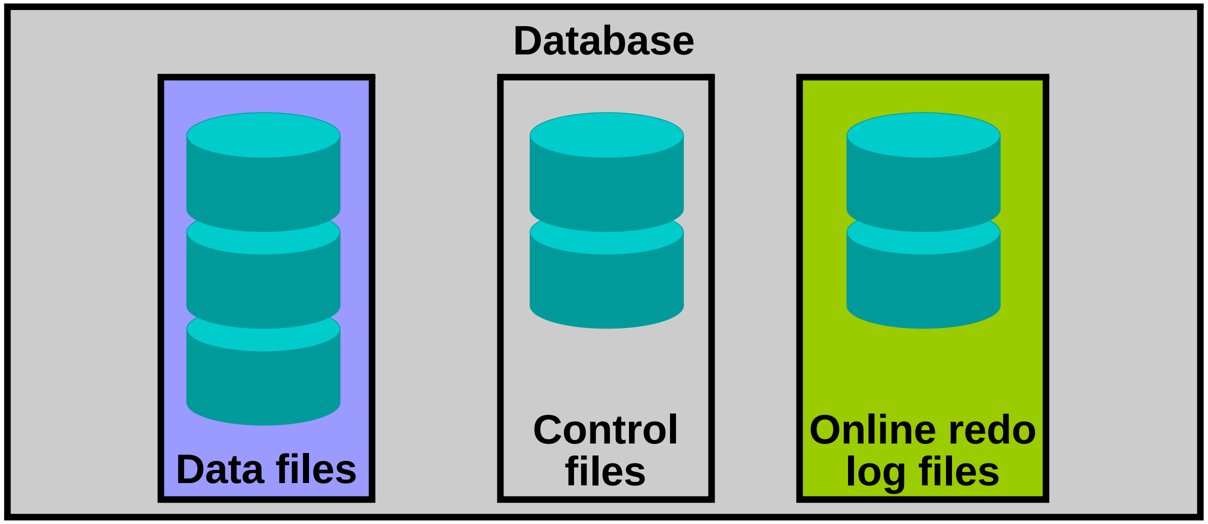
Process Monitor Process

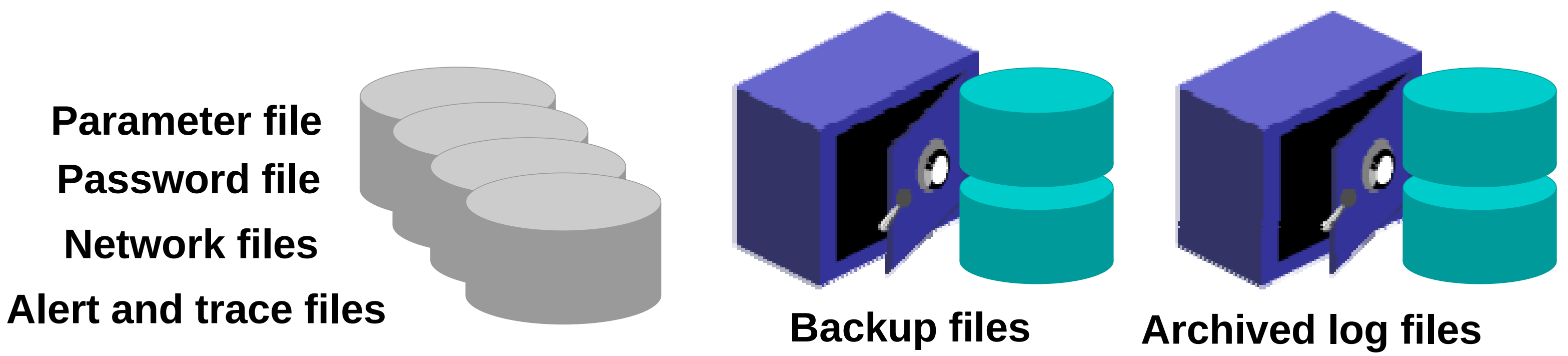
- Performs process recovery when a user process fails:
  - Cleans up the database buffer cache
  - Frees resources used by the user process
- Monitors sessions for idle session timeout
- Dynamically registers database services with listeners



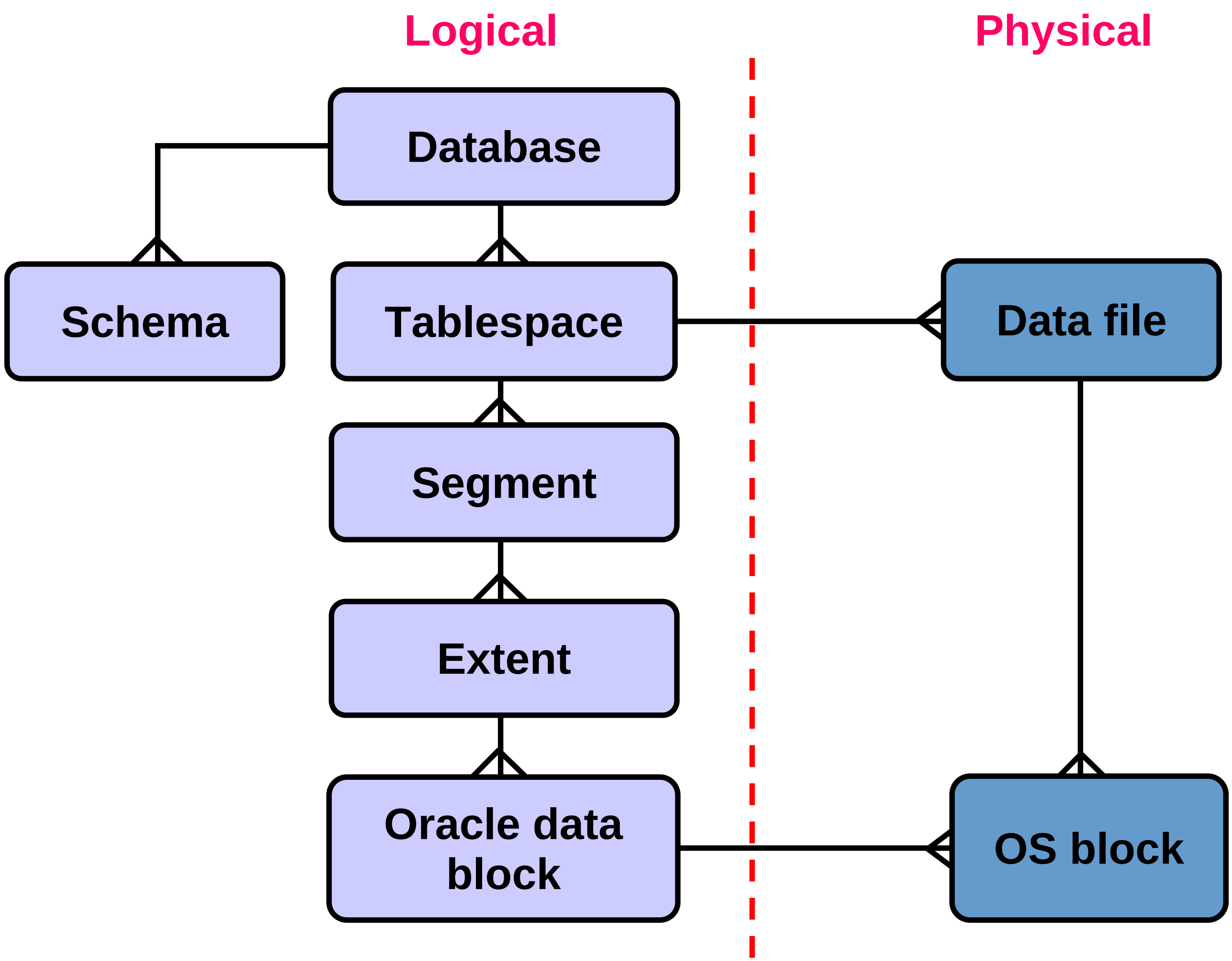
Oracle Database Storage Architecture

DB structures
- Memory
- Process
→ Storage





Logical and Physical Database Structures



Processing a SQL Statement

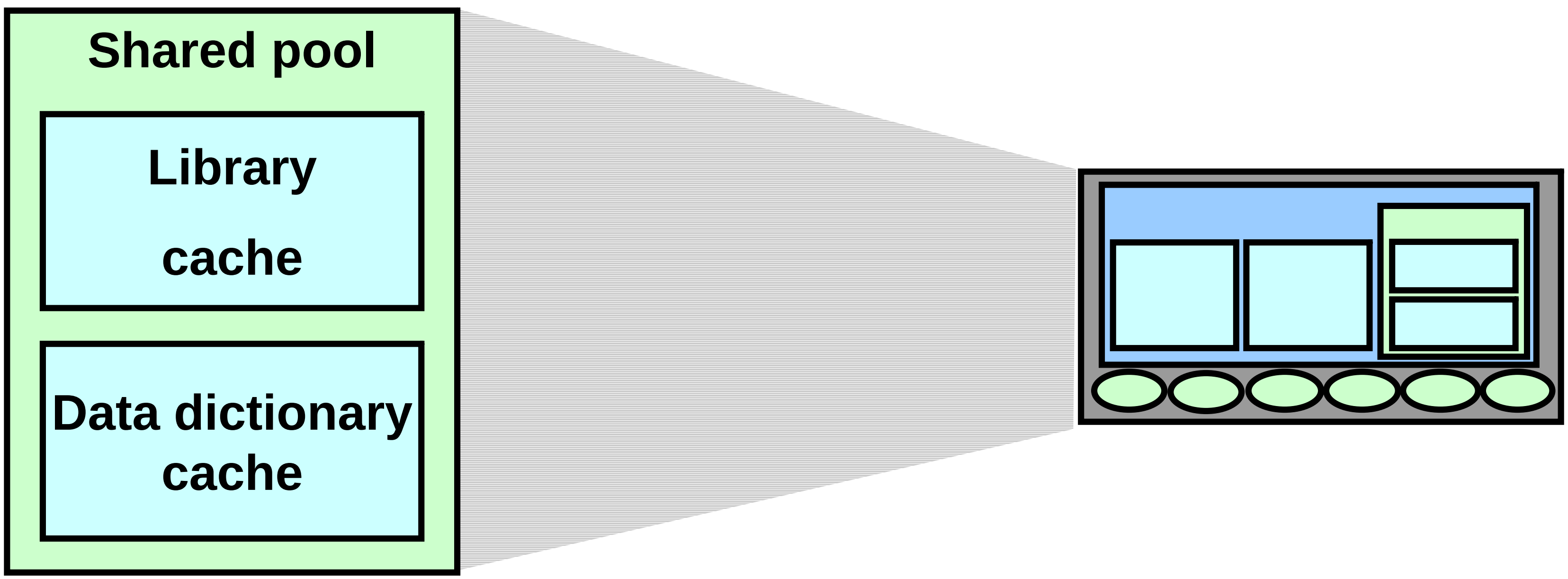
- Connect to an instance using:
  - The user process
  - The server process
- The Oracle server components that are used depend on the type of SQL statement:
  - Queries return rows.
  - Data manipulation language (DML) statements log changes.
  - Commit ensures transaction recovery.
- Some Oracle server components do not participate in SQL statement processing.

Processing a Query

- Parse:
  - Search for an identical statement.
  - Check the syntax, object names, and privileges.
  - Lock the objects used during parse.
  - Create and store the execution plan.
- Execute: Identify the rows selected.
- Fetch: Return the rows to the user process.

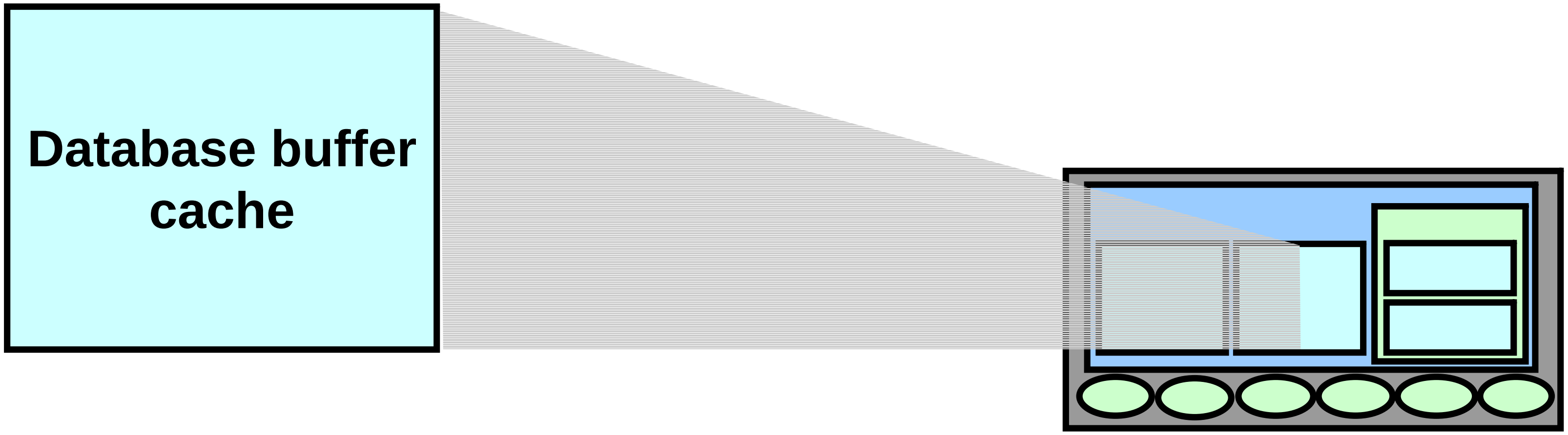
Shared Pool

- 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`.



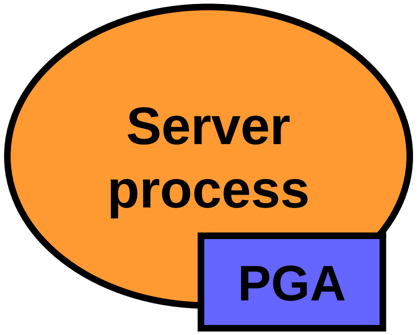
Database Buffer Cache

- The database buffer cache stores the most recently used blocks.
- The size of a buffer is based on `DB_BLOCK_SIZE`.
- The number of buffers is defined by `DB_BLOCK_BUFFERS`.

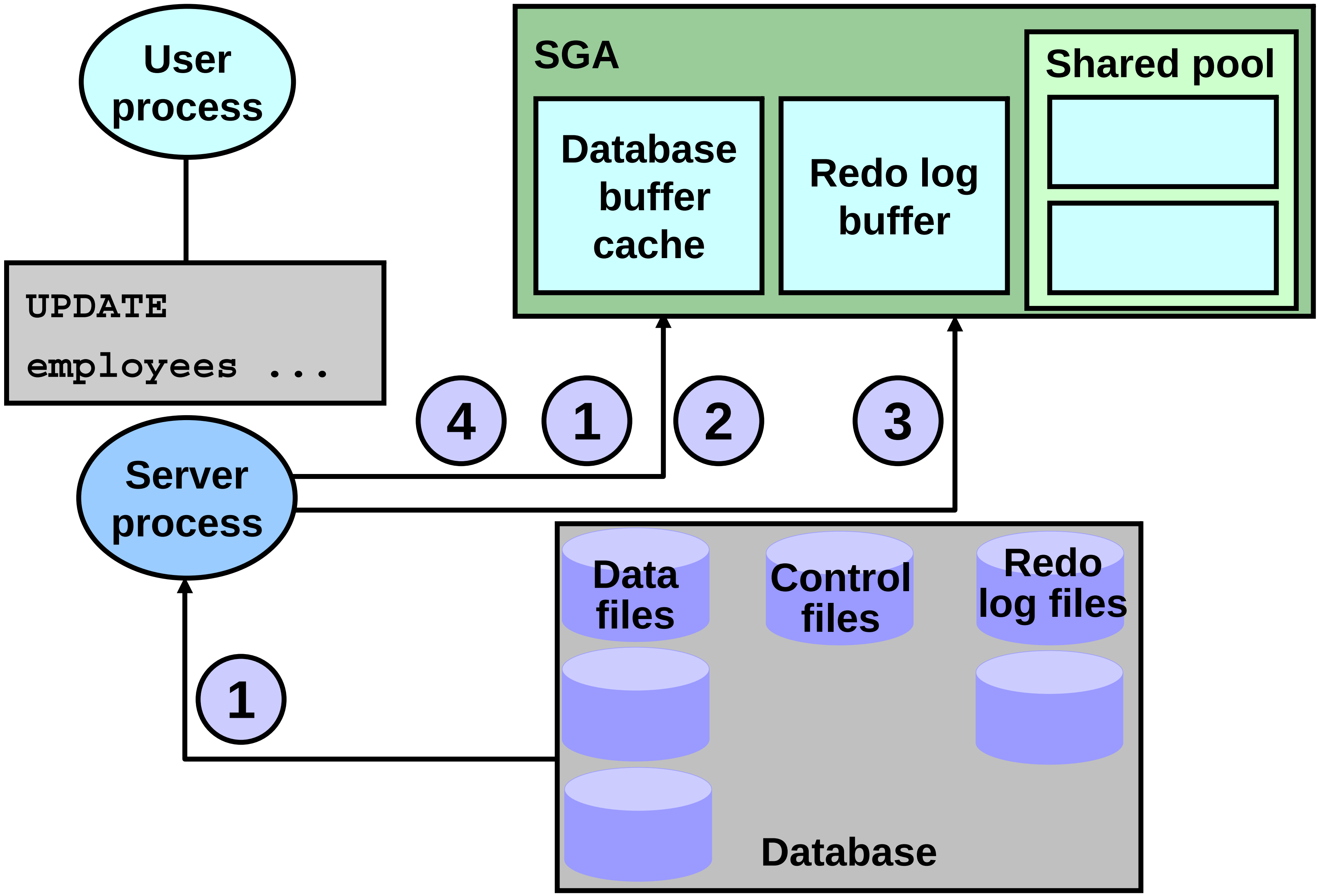


Program Global Area (PGA)

- Is not shared
- Is writable only by the server process
- Contains:
  - Sort area
  - Session information
  - Cursor state
  - Stack space

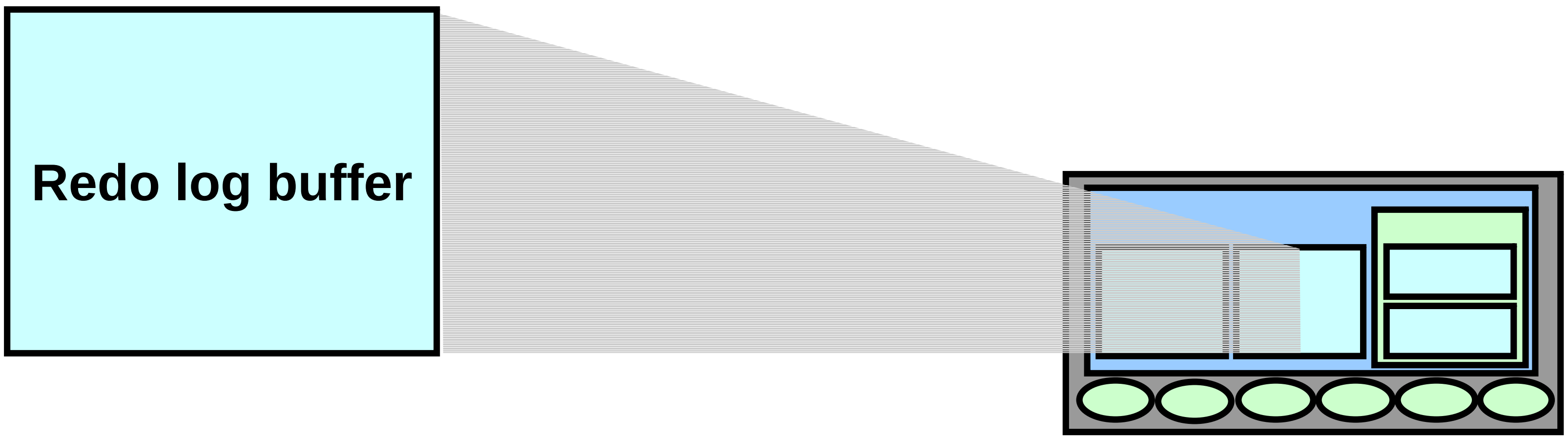


Processing a DML Statement

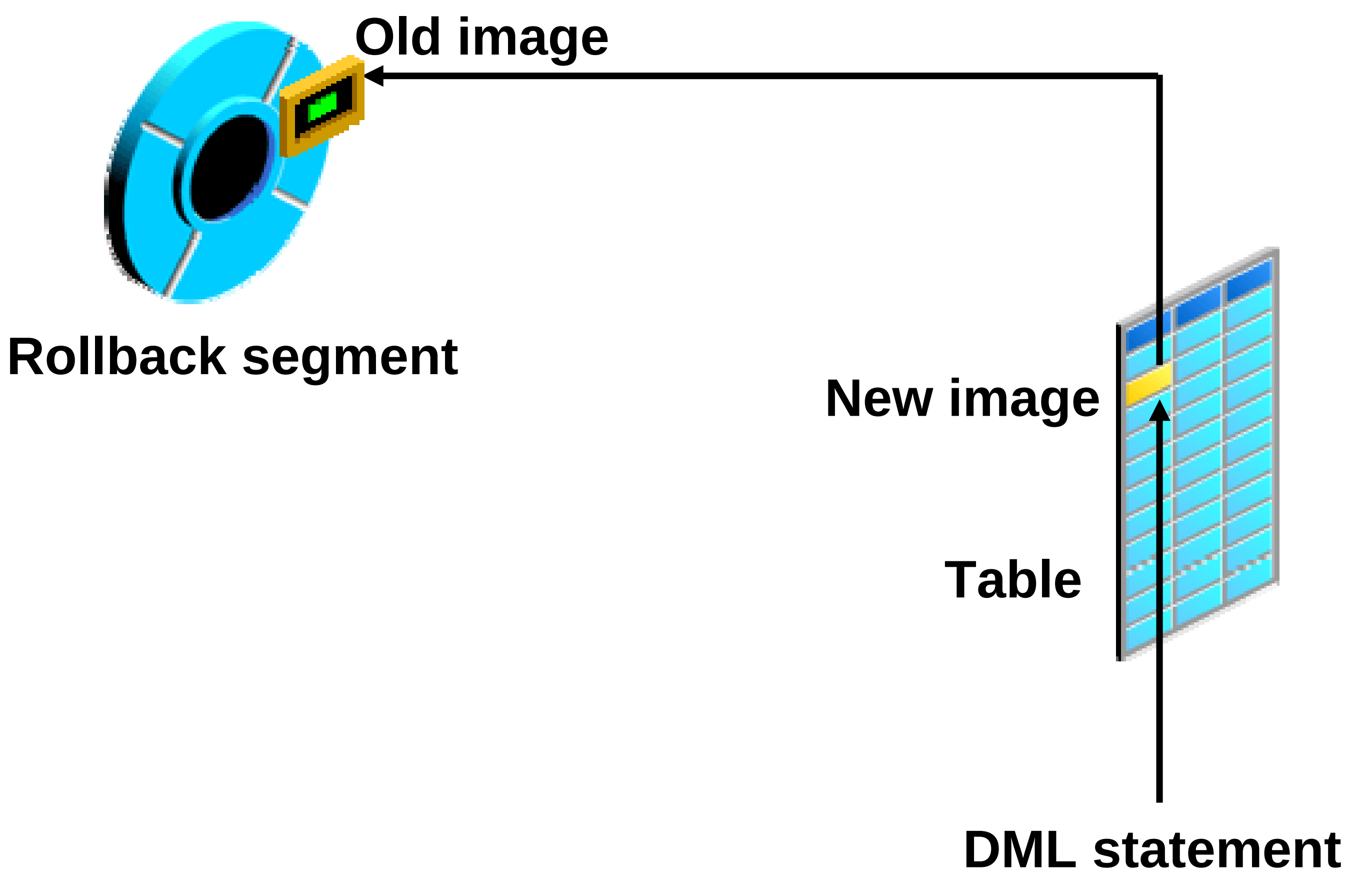


Redo Log Buffer

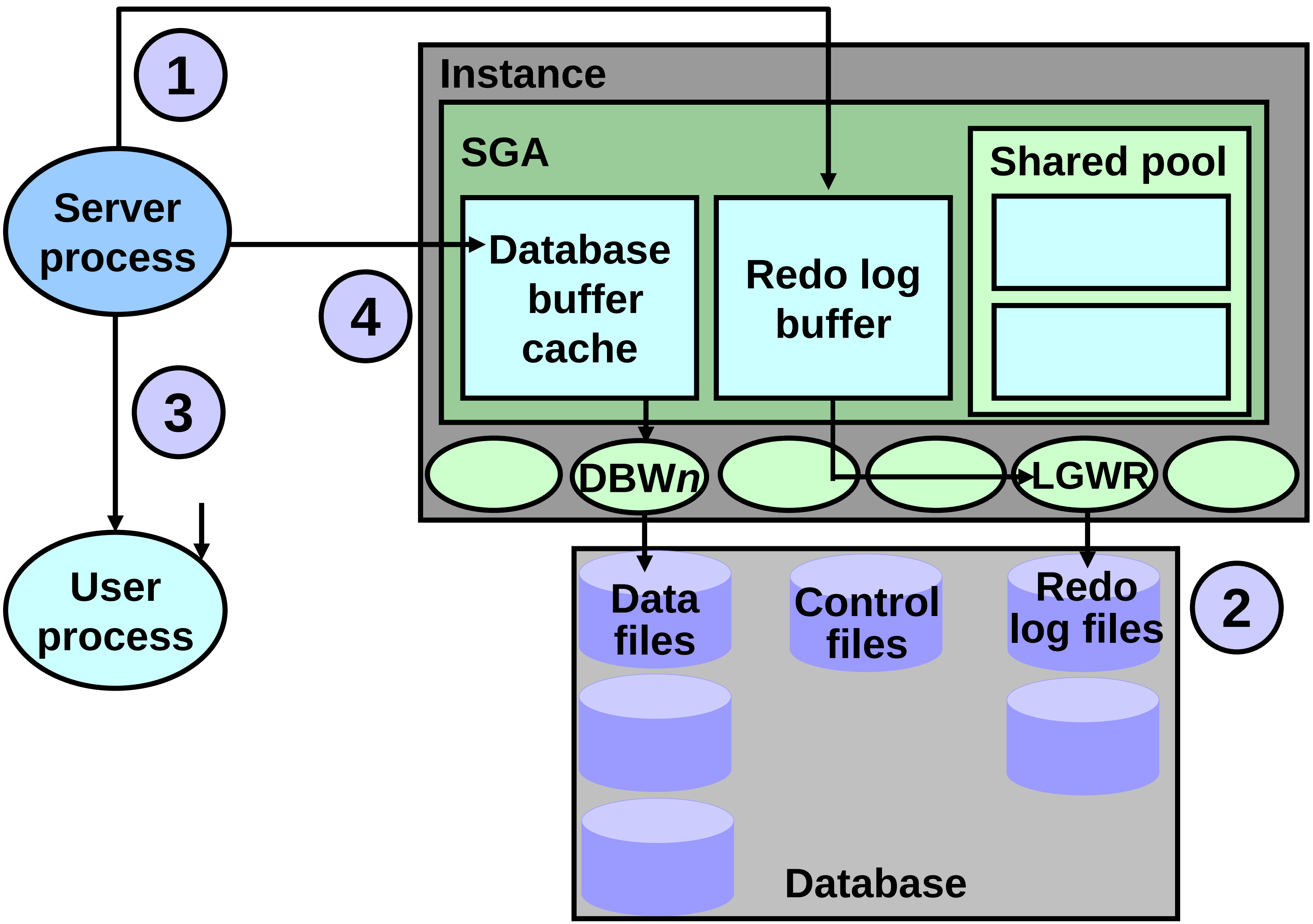
- Has its size defined by `LOG_BUFFER`
- Records changes made through the instance
- Is used sequentially
- Is a circular buffer



Rollback Segment



`COMMIT` Processing



Summary of the Oracle Database Architecture