Oracle Database 10*g*: SQL Fundamentals II

Electronic Presentation

D17111GC11
Production 1.1
August 2004
D39754

Author

Priya Vennapusa

Technical Contributors and Reviewers

Brian Boxx
Andrew Brannigan
Zarko Cesljas
Marjolein Dekkers
Joel Goodman
Nancy Greenberg
Stefan Grenstad
Rosita Hanoman
Angelika Krupp
Christopher Lawless
Malika Marghadi
Priya Nathan
Ruediger Steffan

Publisher

Hemachitra K

Introduction

Objectives

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

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

Course Objectives

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

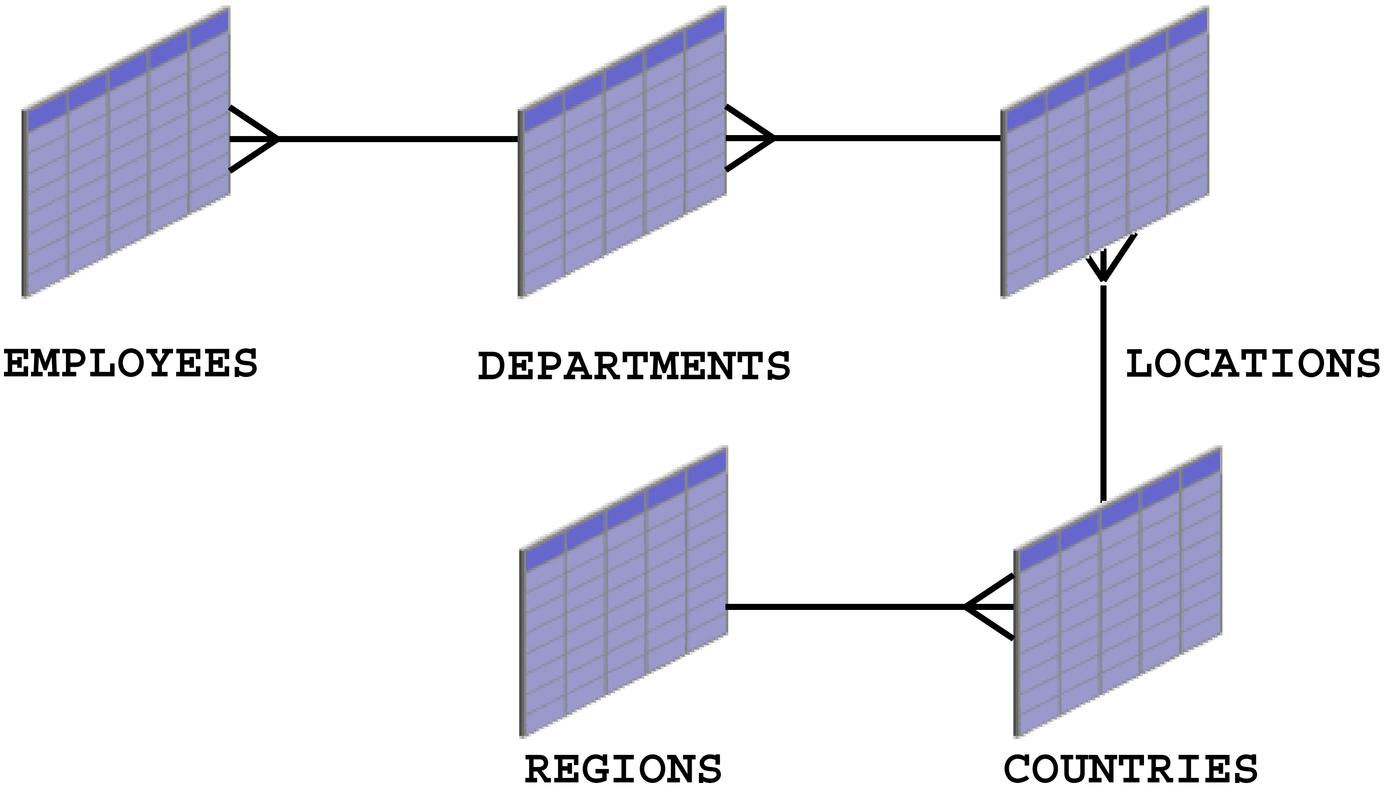
- **Use advanced SQL data retrieval techniques to retrieve data from database tables**
- **Apply advanced techniques in a practice that simulates real life**

Course Overview

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

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

Course Application



Summary

In this lesson, you should have learned the following:

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

1

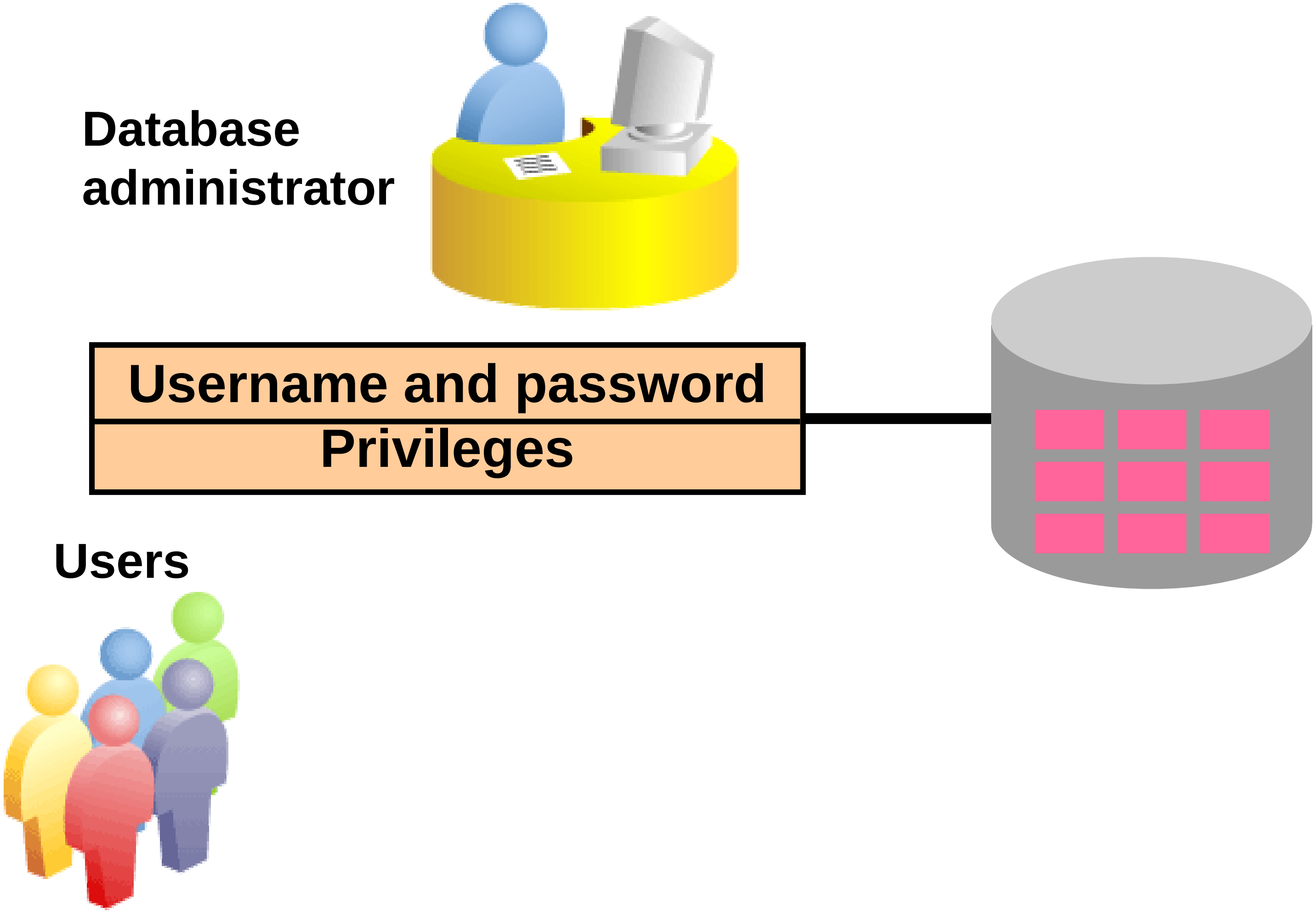
Controlling User Access

Objectives

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

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

Controlling User Access



Database administrator

Username and password

Privileges

Users

Privileges

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

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  HR
IDENTIFIED BY   HR;
User created.
```

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...];
```

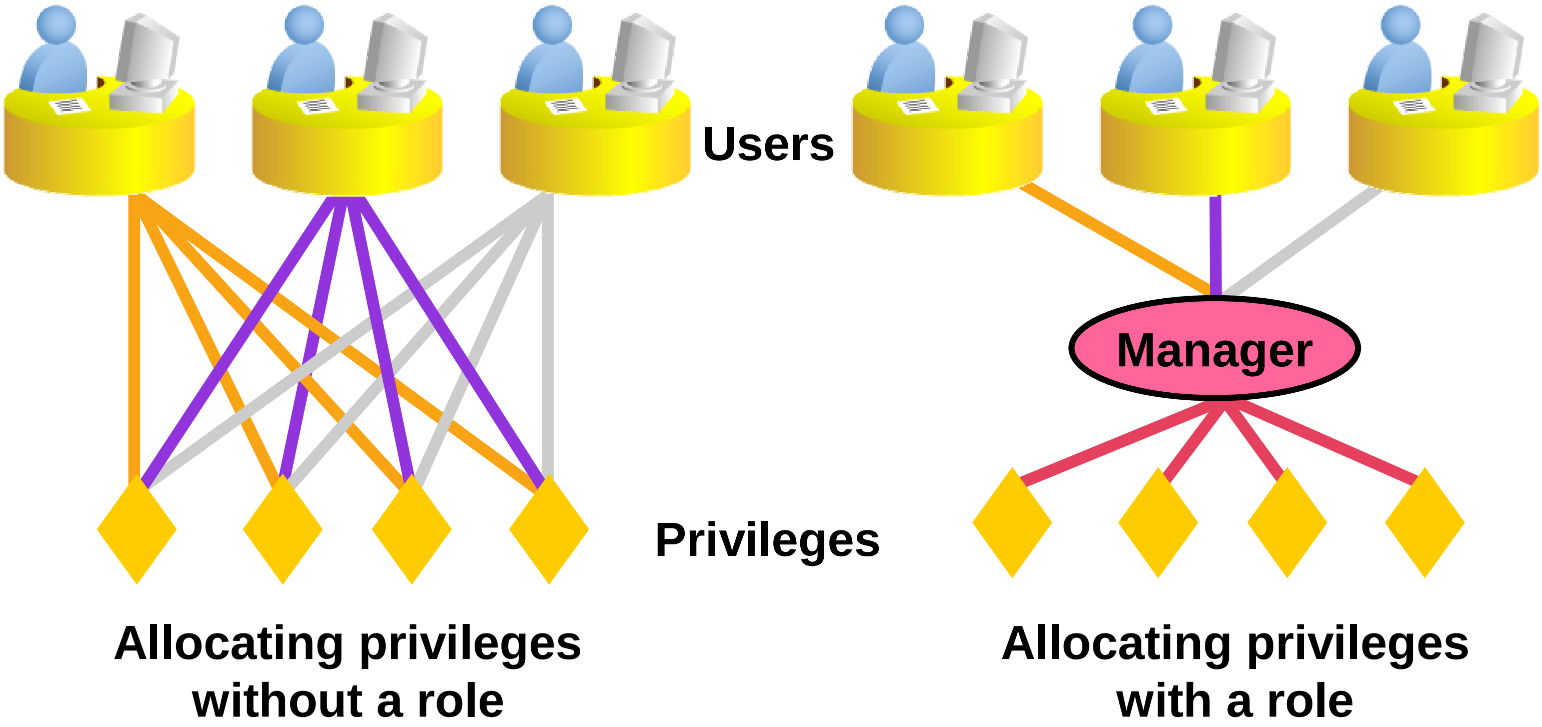
- An application developer, for example, may have the following system privileges:
  - `CREATE SESSION`
  - `CREATE TABLE`
  - `CREATE SEQUENCE`
  - `CREATE VIEW`
  - `CREATE PROCEDURE`

Granting System Privileges

The DBA can grant specific system privileges to a user.

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

What Is a Role?


Users
Manager
Privileges
Allocating privileges without a role
Allocating privileges with a role

Creating and Granting Privileges to a Role

- Create a role

```
CREATE ROLE manager;
Role created.
```

- Grant privileges to a role

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

- Grant a role to users

```
GRANT manager TO DE HAAN, KOCHHAR;
Grant succeeded.
```

Changing Your Password

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

```
ALTER USER HR
IDENTIFIED BY employ;
User altered.
```

Object Privileges

Object Privilege	Table	View	Sequence	Procedure
ALTER	√		√	
DELETE	√	√		
EXECUTE				√
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     sue, rich;
Grant succeeded.
```

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

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

Passing On Your Privileges

- Give a user authority to pass along privileges.

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

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

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

Confirming Privileges Granted

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

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

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

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

Summary

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

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

Practice 1: Overview

This practice covers the following topics:

- Granting other users privileges to your table
- Modifying another user’s table through the privileges granted to you
- Creating a synonym
- Querying the data dictionary views related to privileges

2

Manage Schema Objects

Objectives

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

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

The `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**

The 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]...);
```

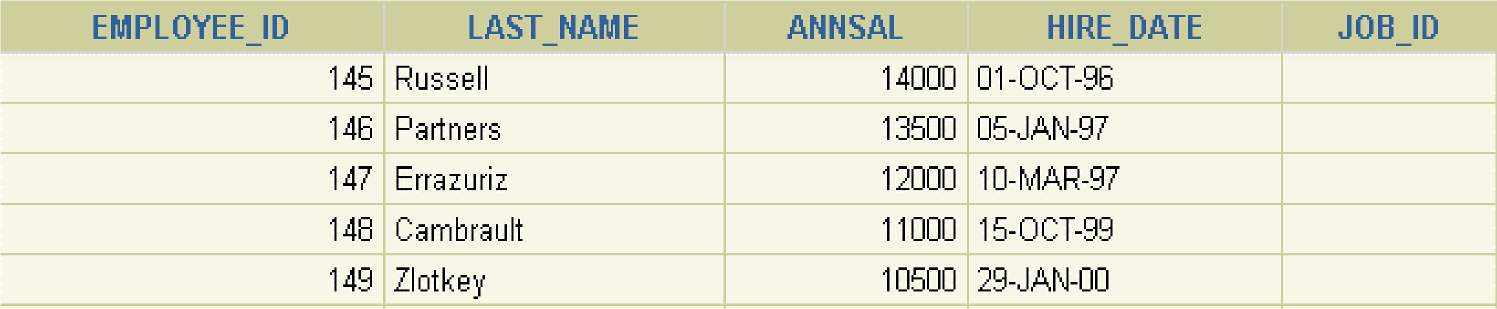
```
ALTER TABLE table
DROP       (column);
```

Adding a Column

- You use the ADD clause to add columns.

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

- The new column becomes the last column.



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

…

Modifying a Column

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

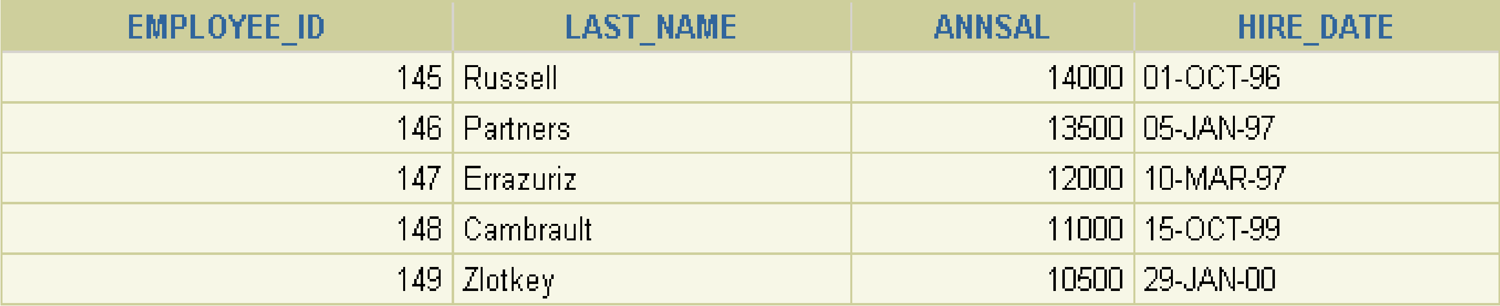
```
ALTER TABLE dept80
MODIFY      (last_name VARCHAR2(30));
Table altered.
```

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

Dropping a Column

Use the DROP COLUMN clause to drop columns you no longer need from the table.

```
ALTER TABLE  dept80
DROP COLUMN  job_id;
Table altered.
```



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

The `SET UNUSED` Option

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

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

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

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;
Table altered.
```

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

ON DELETE CASCADE

Delete child rows when a parent key is deleted.

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

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

Dropping a Constraint

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

```
ALTER TABLE emp2
DROP CONSTRAINT emp_mgr_fk;
Table altered.
```

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

```
ALTER TABLE dept2
DROP PRIMARY KEY CASCADE;
Table altered.
```

Disabling Constraints

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

```
ALTER TABLE emp2
DISABLE CONSTRAINT emp_dt_fk;
Table altered.
```

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;
Table altered.
```

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

Cascading Constraints

Example:

```
ALTER TABLE emp2
DROP COLUMN employee_id CASCADE CONSTRAINTS;
Table altered.
```

```
ALTER TABLE test1
DROP (pk, fk, col1) CASCADE CONSTRAINTS;
Table altered.
```

Overview of Indexes

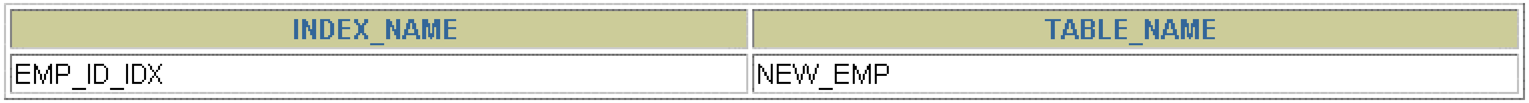
Indexes are created:

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

CREATE INDEX with CREATE TABLE Statement

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

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



INDEX_NAME	TABLE_NAME
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));

Index created.

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;
Index dropped.
```

- 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;
```

The FLASHBACK TABLE Statement

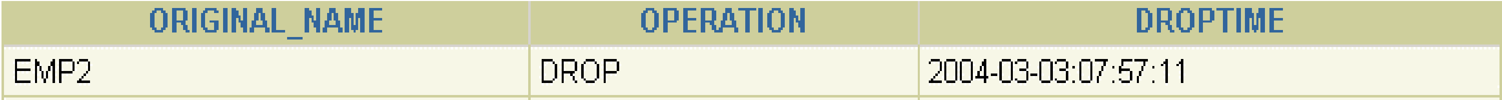
- Repair tool for accidental table modifications
  - Restores a table to an earlier point in time
  - Benefits: Ease of use, availability, fast execution
  - Performed in place
- Syntax:

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

The FLASHBACK TABLE Statement

```
DROP TABLE emp2;
Table dropped
```

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

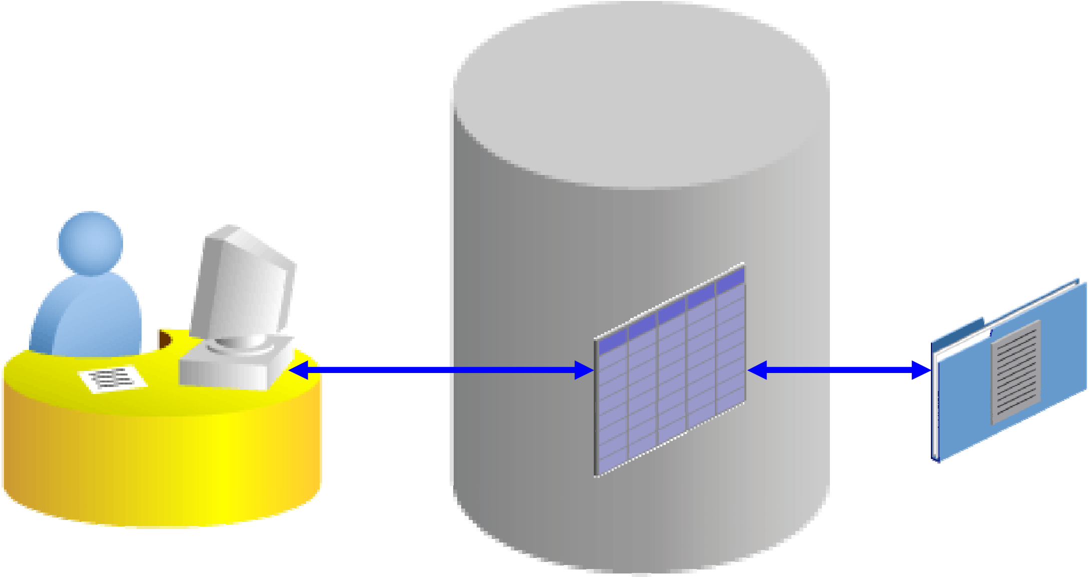


ORIGINAL_NAME	OPERATION	DROPTIME
EMP2	DROP	2004-03-03:07:57:11

…

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

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 hr;
```

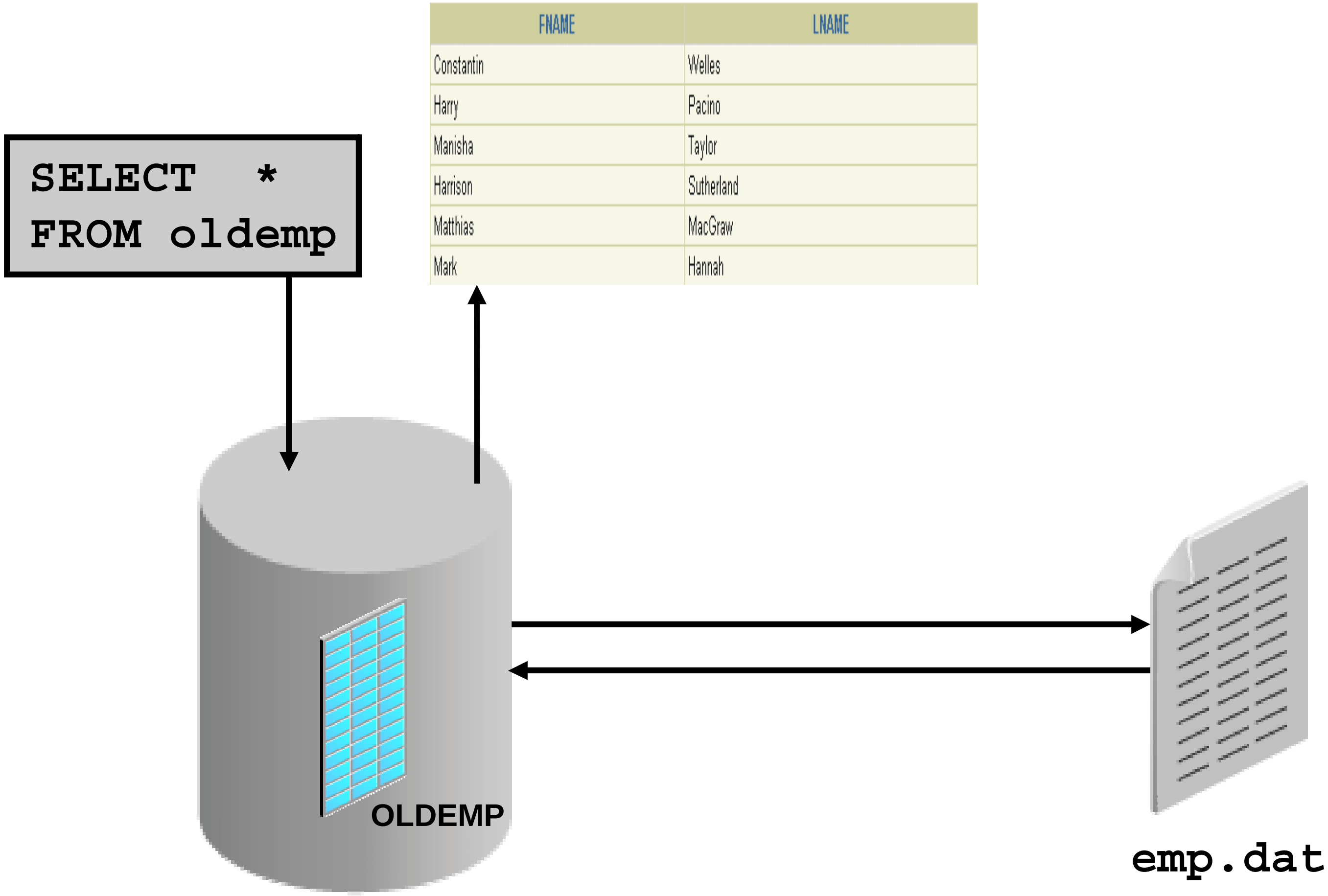
Creating an External Table

```
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 Using `ORACLE_LOADER`

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

Querying External Tables



```
SELECT  *
FROM oldemp
```

FNAME	LNAME
Constantin	Welles
Harry	Pacino
Manisha	Taylor
Harrison	Sutherland
Matthias	MacGraw
Mark	Hannah

Summary

In this lesson, you should have learned how to:

- Add constraints
- Create indexes
- Create a primary key constraint using an index
- Create indexes using the `CREATE TABLE` statement
- Creating function-based indexes
- Drop columns and set column `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

3

Manipulating Large Data Sets

Objectives

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

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

Using Subqueries to Manipulate Data

You can use subqueries in DML statements to:

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

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 created.
```

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

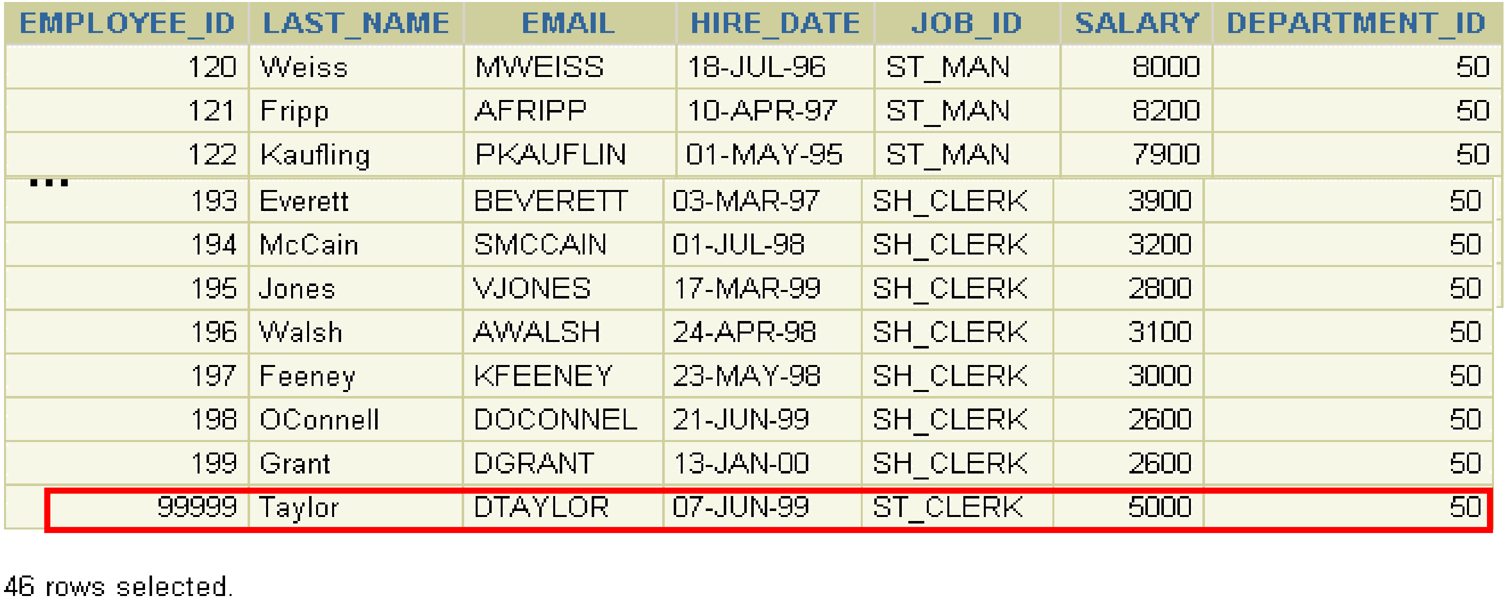
Inserting Using a Subquery as a Target

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

1 row created.
```

Inserting Using a Subquery as a Target

Verify the results.

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

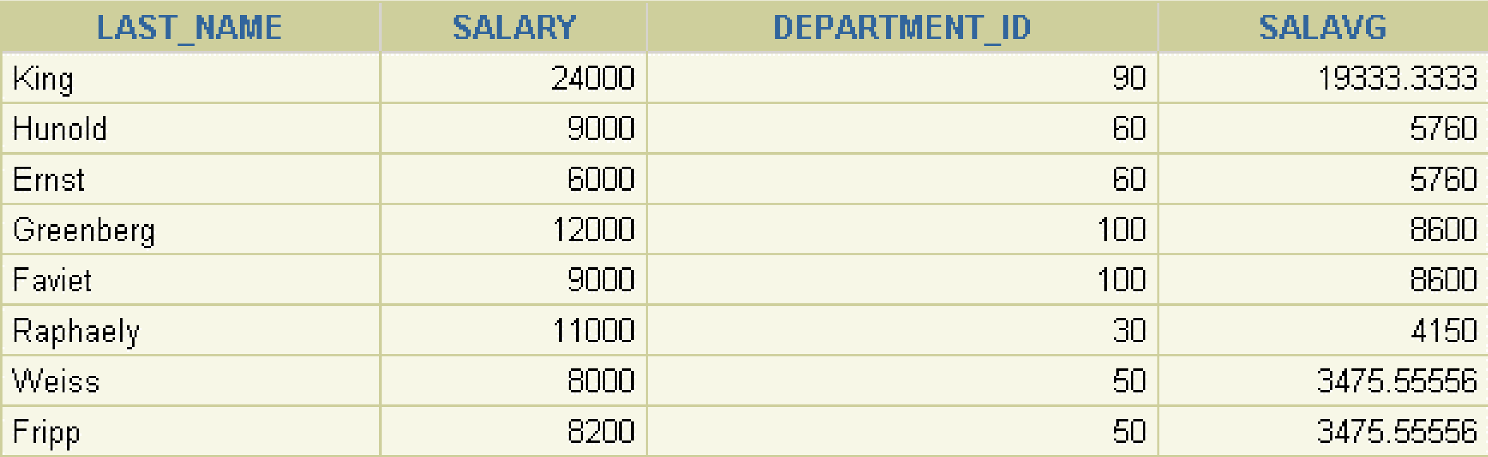


EMPLOYEE_ID	LAST_NAME	EMAIL	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
120	Weiss	MWEISS	18-JUL-96	ST_MAN	8000	50
121	Fripp	AFRIPP	10-APR-97	ST_MAN	8200	50
122	Kaufling	PKAUFLIN	01-MAY-95	ST_MAN	7900	50
...						
193	Everett	BEVERETT	03-MAR-97	SH_CLERK	3900	50
194	McCain	SMCCAIN	01-JUL-98	SH_CLERK	3200	50
195	Jones	VJONES	17-MAR-99	SH_CLERK	2800	50
196	Walsh	AWALSH	24-APR-98	SH_CLERK	3100	50
197	Feeney	KFEENEY	23-MAY-98	SH_CLERK	3000	50
198	OConnell	DOCONNEL	21-JUN-99	SH_CLERK	2600	50
199	Grant	DGRANT	13-JAN-00	SH_CLERK	2600	50
99999	Taylor	DTAYLOR	07-JUN-99	ST_CLERK	5000	50

46 rows selected.

Retrieving Data with a Subquery as Source

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



LAST_NAME	SALARY	DEPARTMENT_ID	SALAVG
King	24000	90	19333.3333
Hunold	9000	60	5760
Ernst	6000	60	5760
Greenberg	12000	100	8600
Faviet	9000	100	8600
Raphaely	11000	30	4150
Weiss	8000	50	3475.55556
Fripp	8200	50	3475.55556

...

Updating Two Columns with a Subquery

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

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

Updating Rows Based on Another Table

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

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

Deleting Rows Based on Another Table

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

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

1 row deleted.
```

Using the `WITH CHECK OPTION` Keyword on DML Statements

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

```
INSERT INTO  (SELECT employee_id, last_name, email,
                    hire_date, job_id, salary
              FROM   empl3
              WHERE  department_id = 50
                  WITH CHECK OPTION)
VALUES (99998, 'Smith', 'JSMITH',
        TO_DATE('07-JUN-99', 'DD-MON-RR'),
        'ST_CLERK', 5000);
INSERT INTO
         *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Overview of the Explicit Default Feature

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

Using Explicit Default Values

- `DEFAULT` **with** `INSERT`**:**

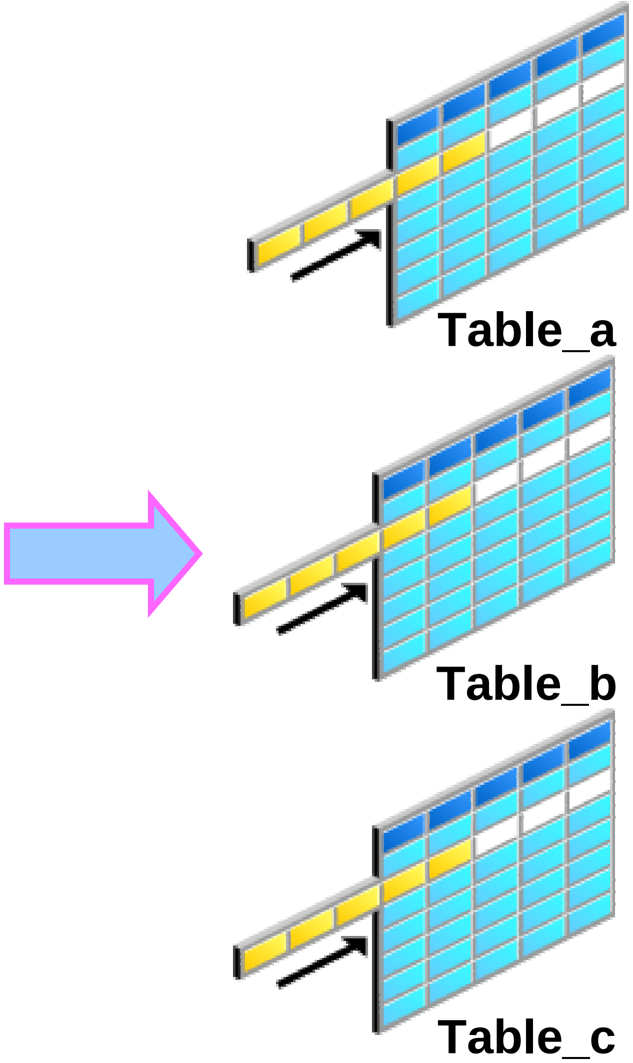
```
INSERT INTO deptm3
   (department_id, department_name, manager_id)
VALUES (300, 'Engineering', DEFAULT);
```

- `DEFAULT` **with** `UPDATE`**:**

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

Overview of Multitable INSERT Statements

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



Overview of Multitable `INSERT` Statements

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

Types of Multitable `INSERT` Statements

The different types of multitable `INSERT` statements are:

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

Multitable INSERT Statements

- Syntax

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

- conditional_insert_clause

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

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 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 created.
```

Conditional INSERT ALL

- Select the EMPLOYEE_ID, HIRE_DATE, SALARY, and MANAGER_ID values from the EMPLOYEES table for those employees whose EMPLOYEE_ID is greater than 200.
- If the SALARY is greater than $10,000, insert these values into the SAL_HISTORY table using a conditional multitable INSERT statement.
- If the MANAGER_ID is greater than 200, insert these values into the MGR_HISTORY table using a conditional multitable INSERT statement.

Conditional INSERT ALL

```
INSERT ALL
  WHEN SAL > 10000 THEN
     INTO sal_history VALUES(EMPID,HIREDATE,SAL)
  WHEN MGR > 200   THEN
     INTO mgr_history VALUES(EMPID,MGR,SAL)
     SELECT employee_id EMPID,hire_date HIREDATE,
            salary SAL, manager_id MGR
     FROM   employees
     WHERE  employee_id > 200;
4 rows created.
```

Conditional INSERT FIRST

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

Conditional INSERT FIRST

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

Pivoting `INSERT`

- Suppose you receive a set of sales records from a nonrelational database table, `SALES_SOURCE_DATA`, in the following format:

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

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

  `EMPLOYEE_ID, WEEK, SALES`

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

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 created.
```

The `MERGE` Statement

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

The MERGE Statement Syntax

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

```
MERGE INTO table_name table_alias
  USING (table|view|sub_query) alias
  ON (join condition)
  WHEN MATCHED THEN
    UPDATE SET
    col1 = col_val1,
    col2 = col2_val
  WHEN NOT MATCHED THEN
    INSERT (column_list)
    VALUES (column_values);
```

Merging Rows

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

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

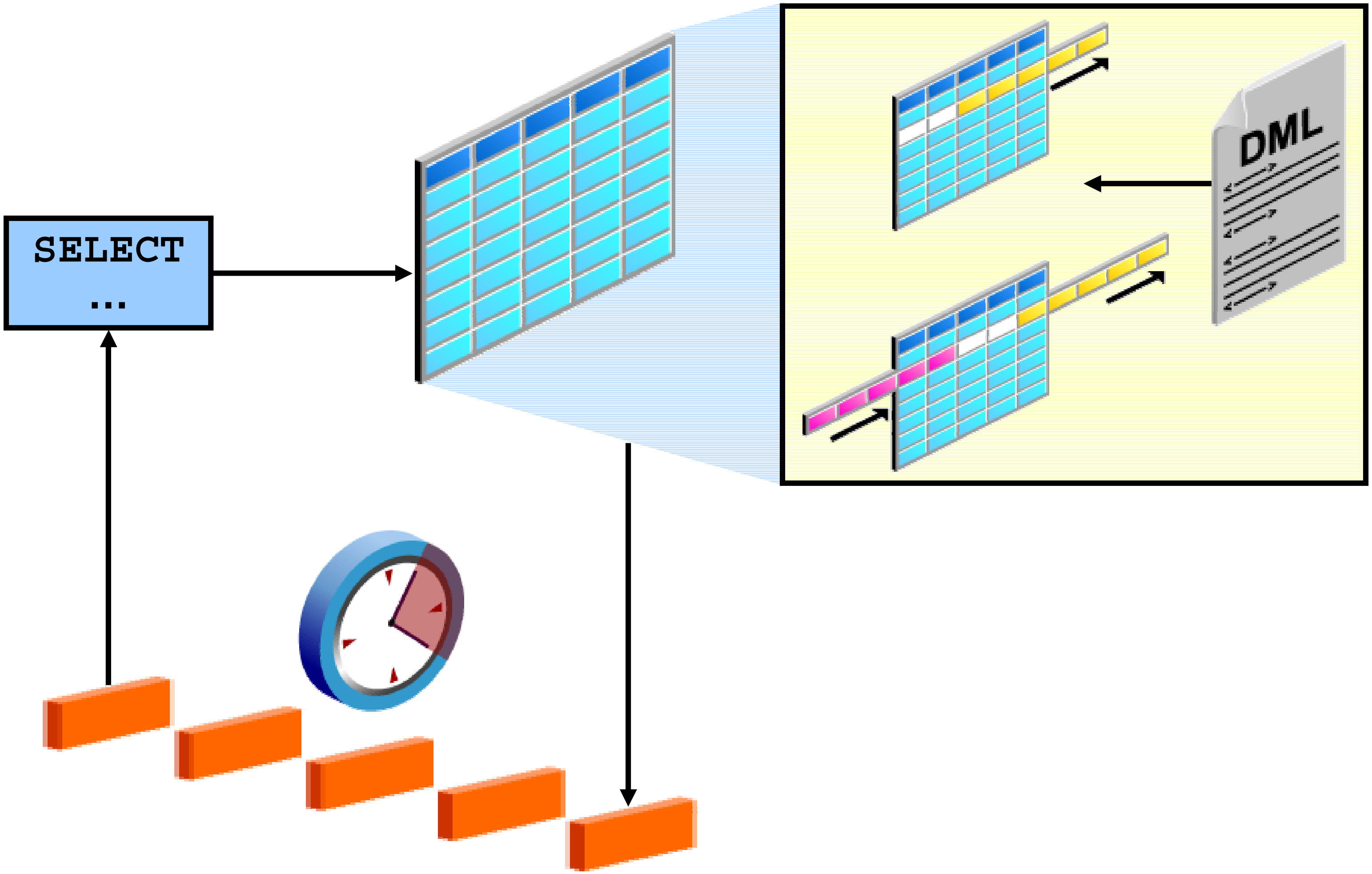
Merging Rows

```
TRUNCATE TABLE empl3;

SELECT *
FROM empl3;
no rows selected
```

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

```
SELECT *
FROM empl3;

107 rows selected.
```

Tracking Changes in Data

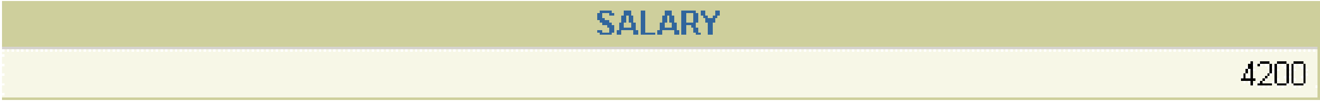


Versions of retrieved rows

Example of the Flashback Version Query

1

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



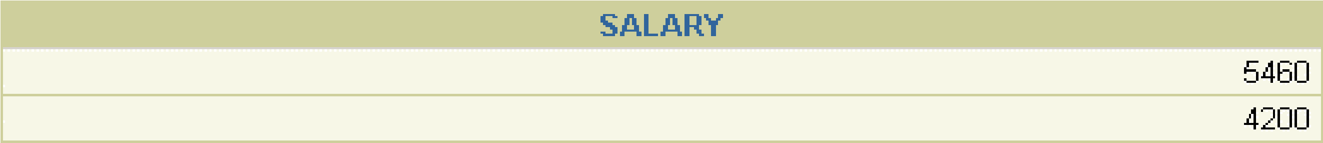
SALARY
4200

2

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

COMMIT;
```

3

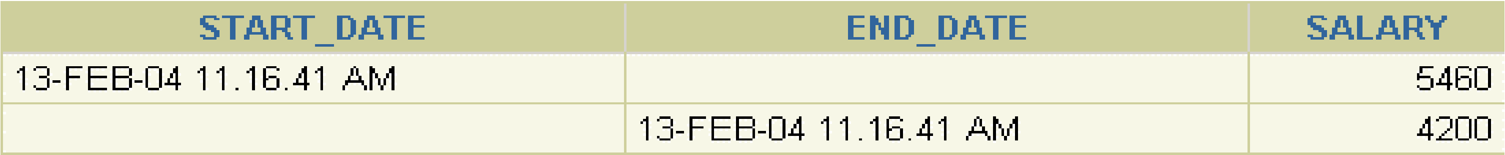
```
SELECT salary FROM employees3
  VERSIONS BETWEEN SCN MINVALUE AND MAXVALUE
WHERE  employee_id = 107;
```



SALARY
5460
4200

The 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
13-FEB-04 11.16.41 AM		5460
	13-FEB-04 11.16.41 AM	4200

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 `ALL INSERT`
  - Conditional `FIRST INSERT`
- Merge rows in a table
- Manipulate data using subqueries
- Track the changes to data over a period of time

Practice 3: Overview

This practice covers the following topics:

- Performing multitable `INSERTs`
- Performing `MERGE` operations
- Tracking row versions

4

Generating Reports by Grouping Related Data

Objectives

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

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

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];
```

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.

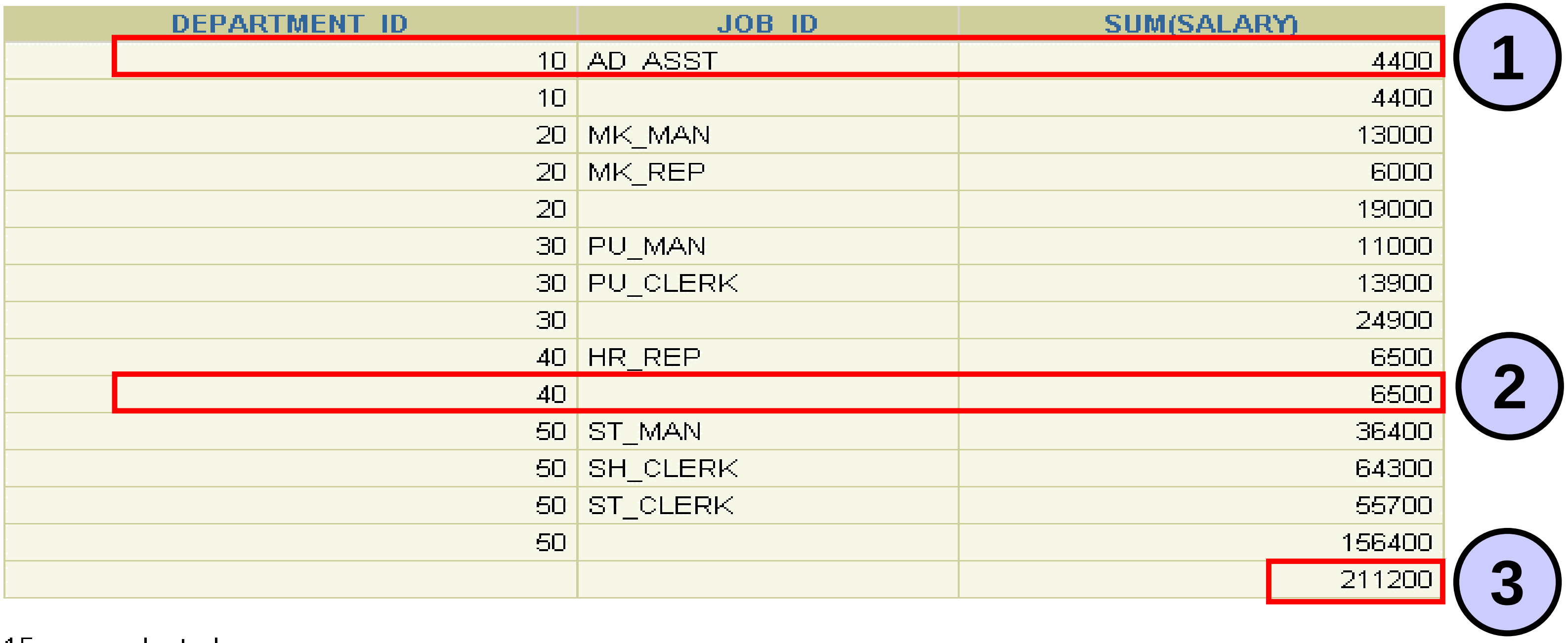
ROLLUP Operator

- 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)
10	AD_ASST	4400
10		4400
20	MK_MAN	13000
20	MK_REP	6000
20		19000
30	PU_MAN	11000
30	PU_CLERK	13900
30		24900
40	HR_REP	6500
40		6500
50	ST_MAN	36400
50	SH_CLERK	64300
50	ST_CLERK	55700
50		156400
		211200

15 rows selected.

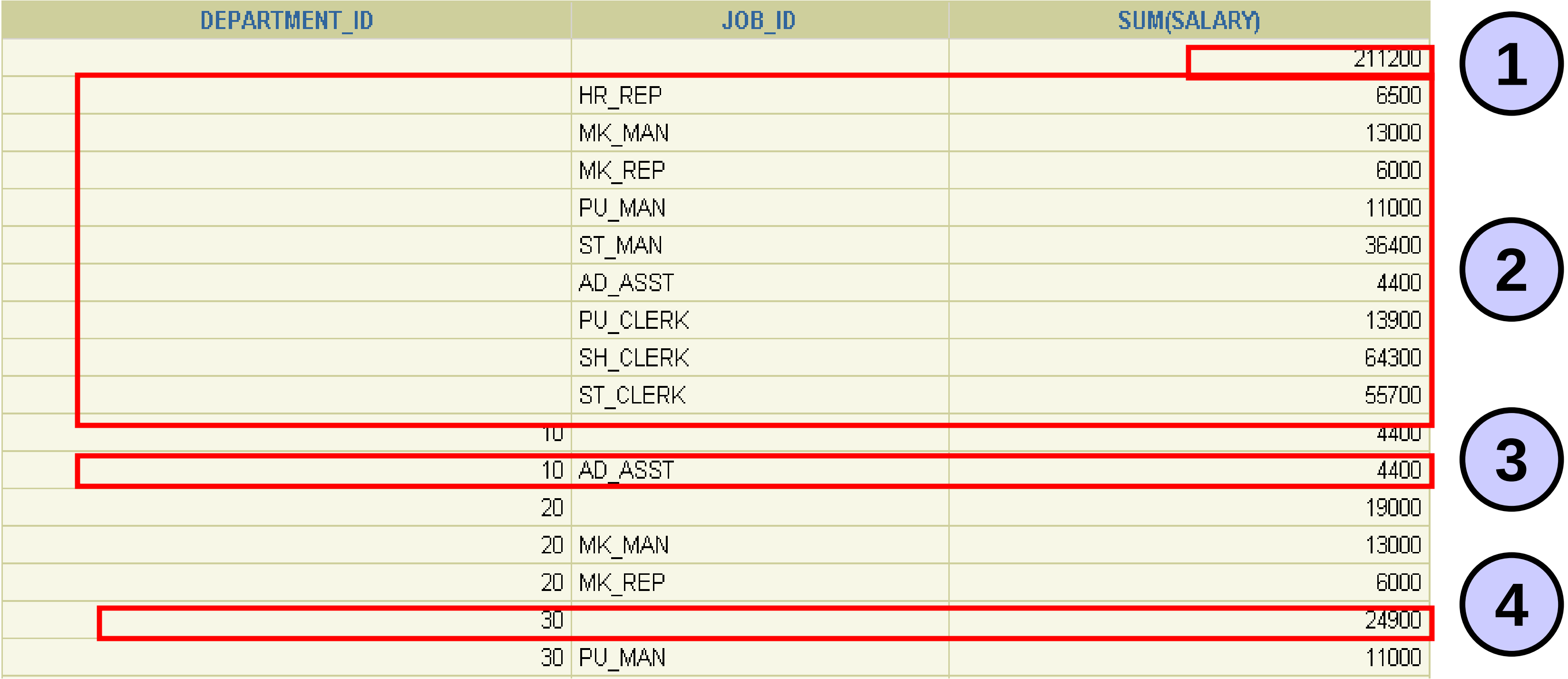
CUBE Operator

- 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)
		211200
	HR_REP	6500
	MK_MAN	13000
	MK_REP	6000
	PU_MAN	11000
	ST_MAN	36400
	AD_ASST	4400
	PU_CLERK	13900
	SH_CLERK	64300
	ST_CLERK	55700
10		4400
10	AD_ASST	4400
20		19000
20	MK_MAN	13000
20	MK_REP	6000
30		24900
30	PU_MAN	11000

GROUPING Function

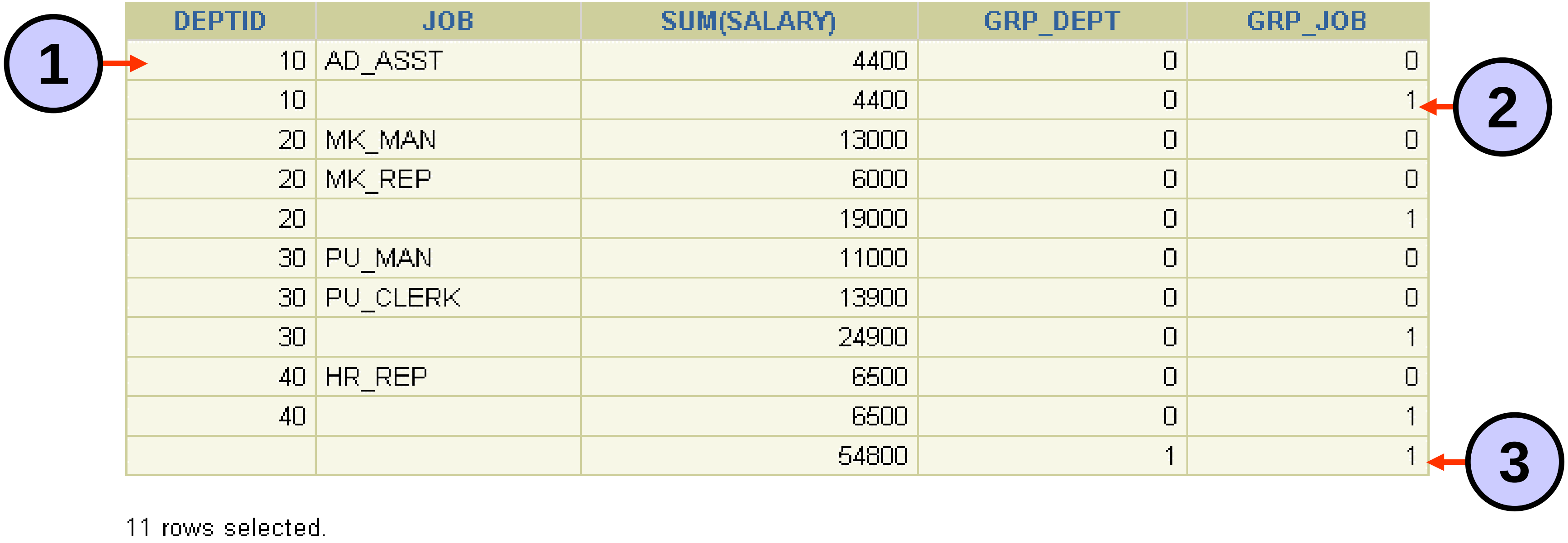
The GROUPING function:

- 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
10	AD_ASST	4400	0	0
10		4400	0	1
20	MK_MAN	13000	0	0
20	MK_REP	6000	0	0
20		19000	0	1
30	PU_MAN	11000	0	0
30	PU_CLERK	13900	0	0
30		24900	0	1
40	HR_REP	6500	0	0
40		6500	0	1
		54800	1	1

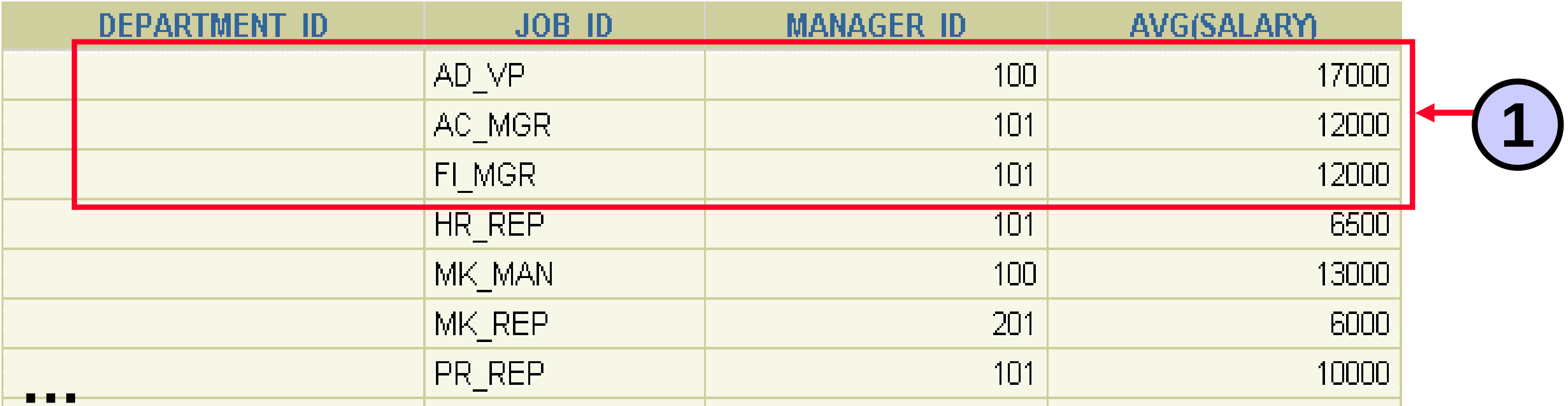
11 rows selected.

`GROUPING SETS`

- `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 the performance benefit.

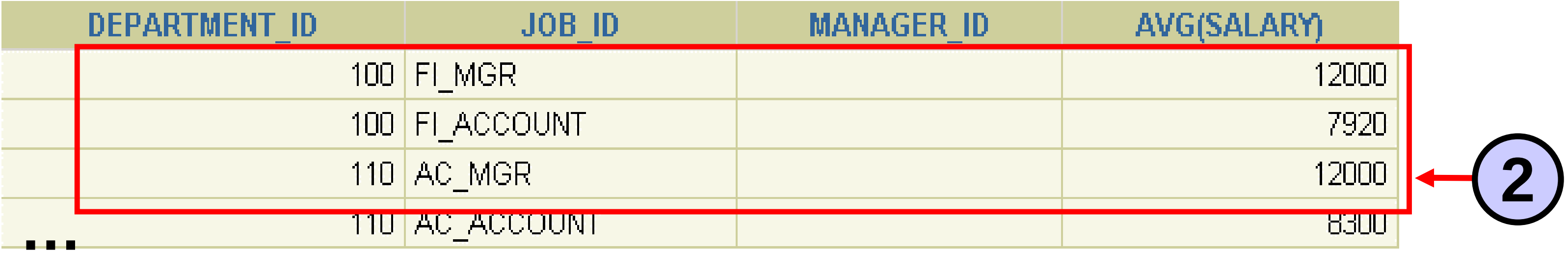
GROUPING SETS: Example

```
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)
	AD_VP	100	17000
	AC_MGR	101	12000
	FI_MGR	101	12000
	HR_REP	101	6500
	MK_MAN	100	13000
	MK_REP	201	6000
	PR_REP	101	10000

…



DEPARTMENT_ID	JOB_ID	MANAGER_ID	AVG(SALARY)
100	FI_MGR		12000
100	FI_ACCOUNT		7920
110	AC_MGR		12000
110	AC_ACCOUNT		8300

…

Composite Columns

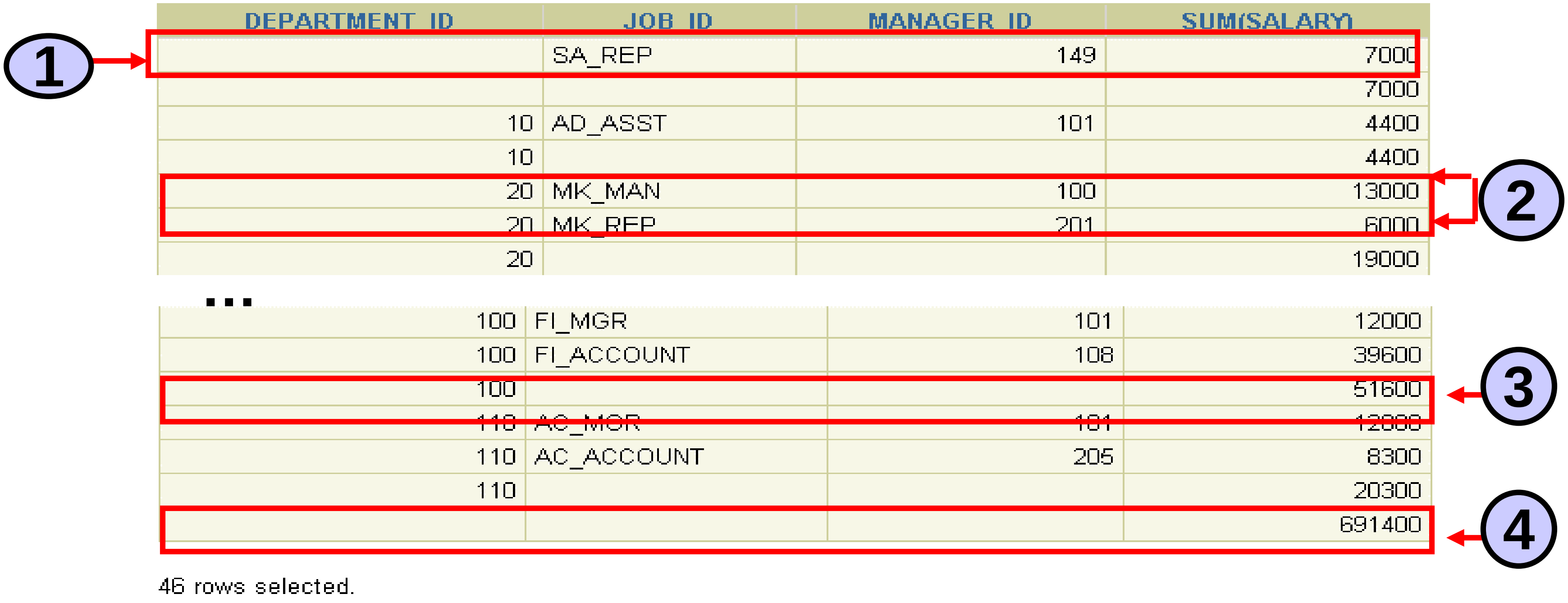
- **A composite column is a collection of columns that are treated as a unit.**

```
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));
```



DEPARTMENT_ID	JOB_ID	MANAGER_ID	SUM(SALARY)
	SA_REP	149	7000
			7000
10	AD_ASST	101	4400
10			4400
20	MK_MAN	100	13000
20	MK_REP	201	6000
20			19000

...

100	FI_MGR	101	12000
100	FI_ACCOUNT	108	39600
100			51600
110	AC_MGR	101	12000
110	AC_ACCOUNT	205	8300
110			20300
			691400

46 rows selected.

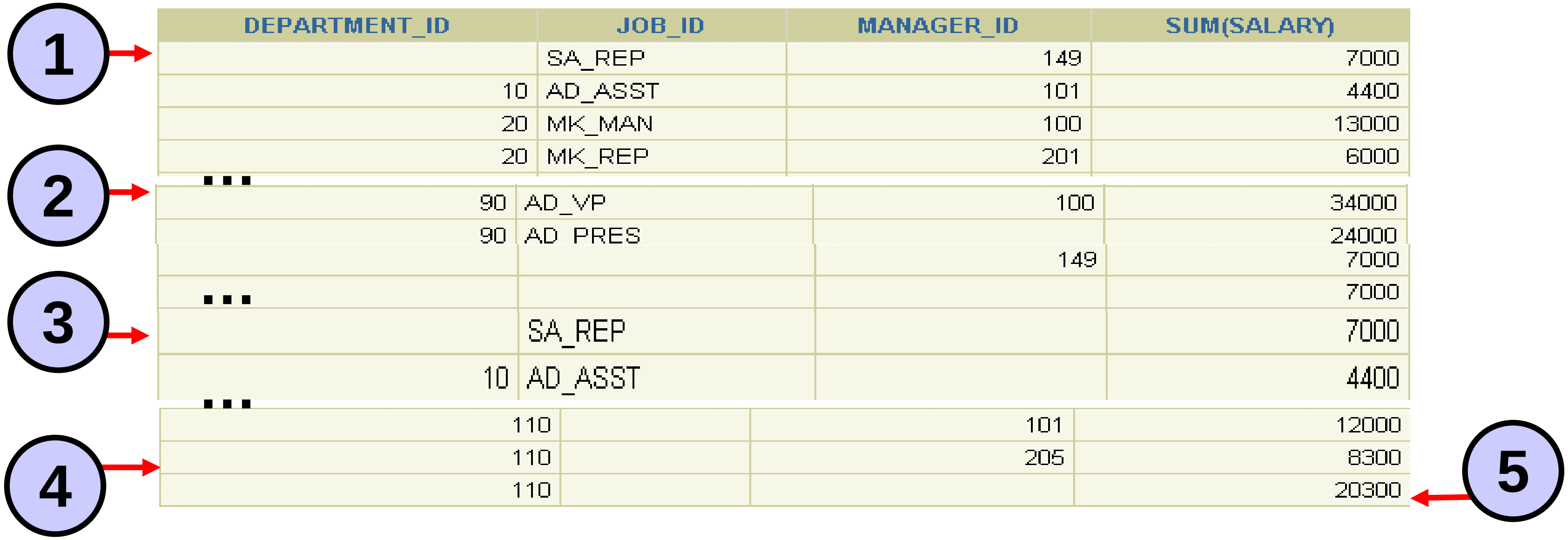
Concatenated Groupings

- 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)
	SA_REP	149	7000
10	AD_ASST	101	4400
20	MK_MAN	100	13000
20	MK_REP	201	6000
...			
90	AD_VP	100	34000
90	AD_PRES		24000
		149	7000
			7000
...			
	SA_REP		7000
10	AD_ASST		4400
...			
110		101	12000
110		205	8300
110			20300

93 rows selected.

Summary

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

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

Practice 4: Overview

This practice covers using:

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

5

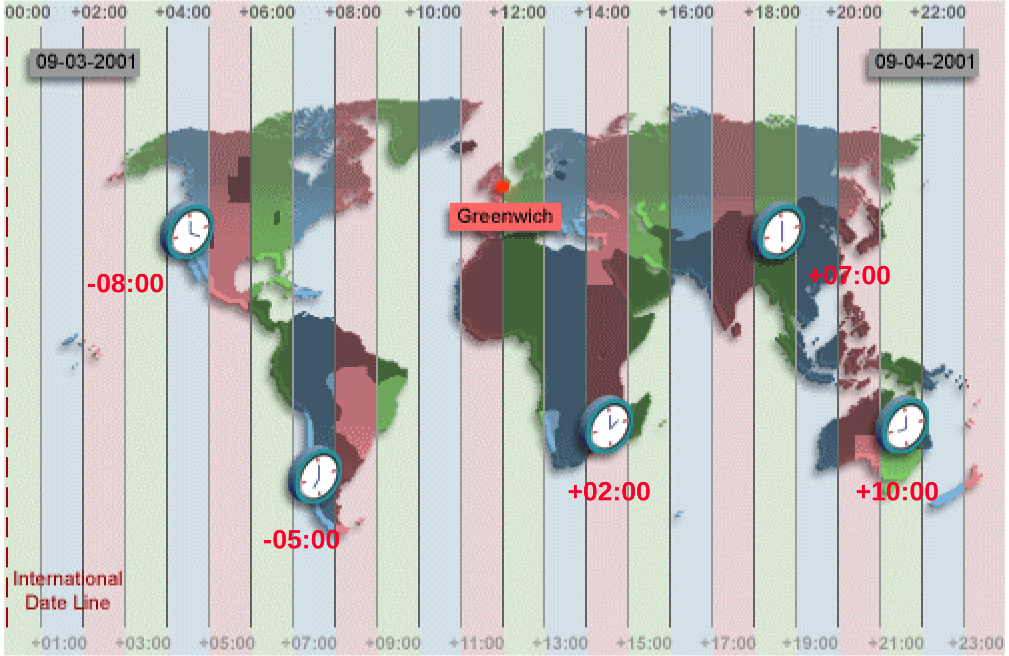
Managing Data in Different Time Zones

Objectives

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

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

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

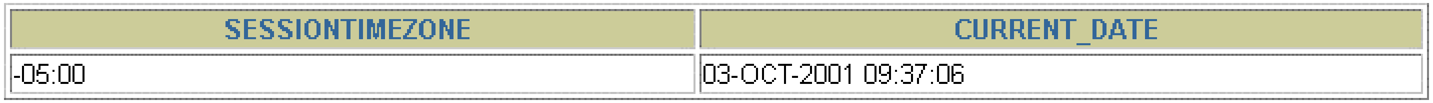
- CURRENT_DATE
  - Returns the current date from the system
  - Has a data type of DATE
- CURRENT_TIMESTAMP
  - Returns the current timestamp from the system
  - Has a data type of TIMESTAMP WITH TIME ZONE
- LOCALTIMESTAMP
  - Returns the current timestamp from user session
  - Has a data type of TIMESTAMP

CURRENT_DATE

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

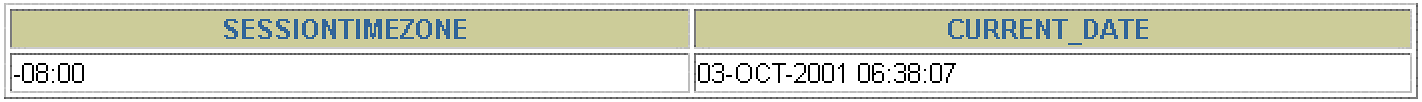
```
ALTER SESSION
SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
```

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



SESSIONTIMEZONE	CURRENT_DATE
-05:00	03-OCT-2001 09:37:06

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

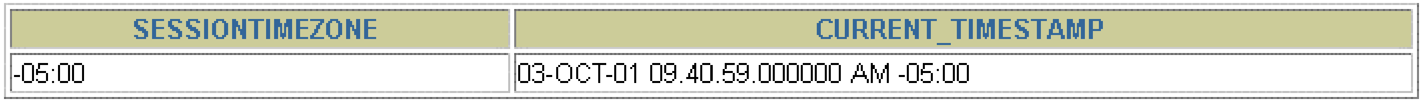


SESSIONTIMEZONE	CURRENT_DATE
-08:00	03-OCT-2001 06:38:07

CURRENT_TIMESTAMP

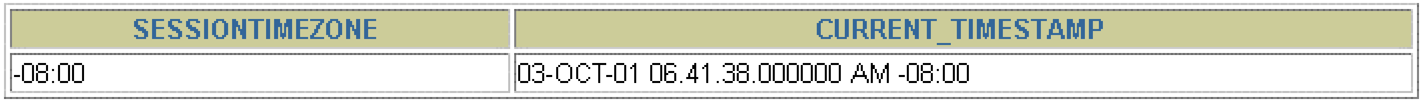
Display the current date and fractional time in the session's time zone.

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



SESSIONTIMEZONE	CURRENT_TIMESTAMP
-05:00	03-OCT-01 09.40.59.000000 AM -05:00

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

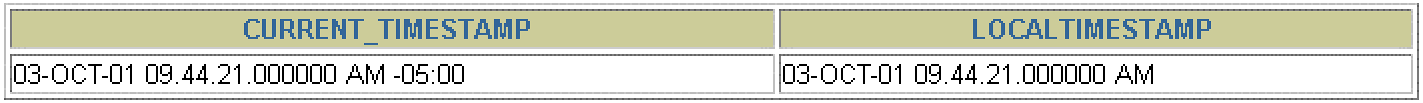


SESSIONTIMEZONE	CURRENT_TIMESTAMP
-08:00	03-OCT-01 06.41.38.000000 AM -08:00

LOCALTIMESTAMP

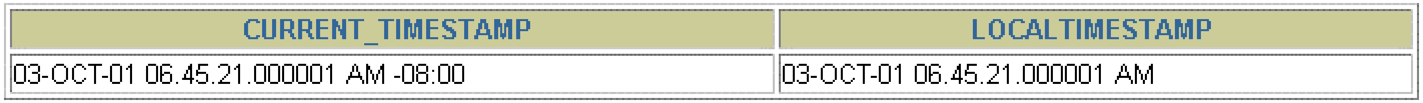
- Display the current date and time in the session’s time zone in a value of `TIMESTAMP` data type.

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



CURRENT_TIMESTAMP	LOCALTIMESTAMP
03-OCT-01 09.44.21.000000 AM -05:00	03-OCT-01 09.44.21.000000 AM

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



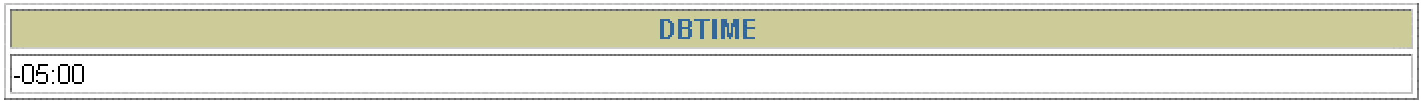
CURRENT_TIMESTAMP	LOCALTIMESTAMP
03-OCT-01 06.45.21.000001 AM -08:00	03-OCT-01 06.45.21.000001 AM

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

DBTIMEZONE and SESSIONTIMEZONE

- Display the value of the database time zone.

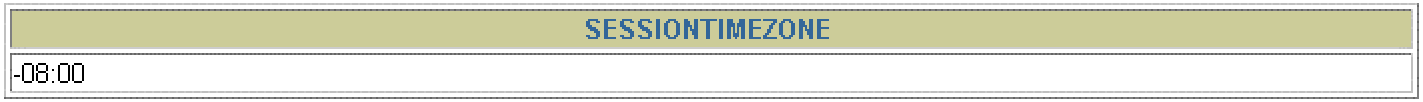
```
SELECT DBTIMEZONE FROM DUAL;
```



```
DBTIME
-05:00
```

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

```
SELECT SESSIONTIMEZONE FROM DUAL;
```



```
SESSIONTIMEZONE
-08:00
```

`TIMESTAMP` Data Type

- The `TIMESTAMP` data type is an extension of the `DATE` data type.
- It stores the year, month, and day of the `DATE` data type, plus hour, minute, and second values, as well as the fractional second value.
- Variations in `TIMESTAMP` are:
  - `TIMESTAMP [(fractional_seconds_precision)]`
  - `TIMESTAMP [(fractional_seconds_precision)] WITH TIME ZONE`
  - `TIMESTAMP [(fractional_seconds_precision)] WITH LOCAL TIME ZONE`

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 a time zone offset in its value

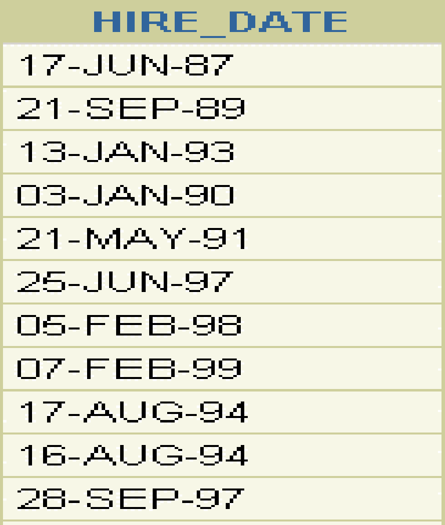
TIMESTAMP Fields

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 emp5;
```

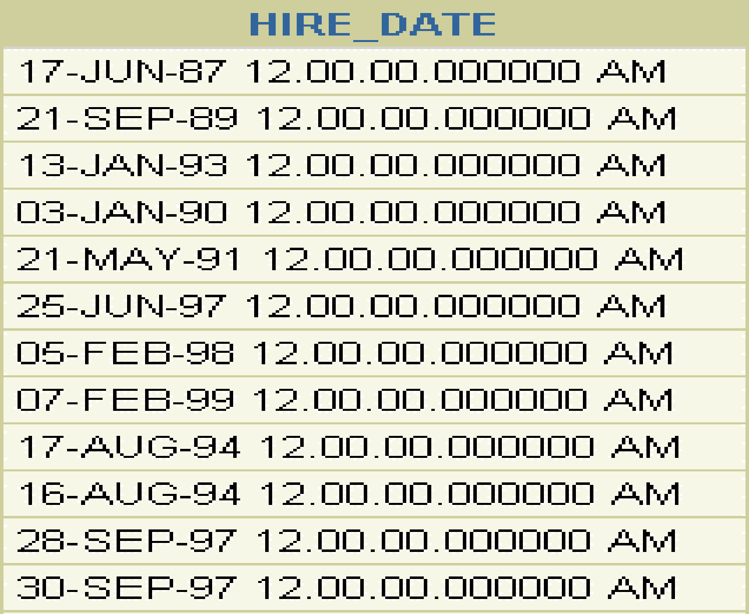


HIRE_DATE
17-JUN-87
21-SEP-89
13-JAN-93
03-JAN-90
21-MAY-91
25-JUN-97
05-FEB-98
07-FEB-99
17-AUG-94
16-AUG-94
28-SEP-97

…

B

```
ALTER TABLE emp5
MODIFY hire_date TIMESTAMP;

SELECT hire_date
FROM emp5;
```



HIRE_DATE
17-JUN-87 12.00.00.000000 AM
21-SEP-89 12.00.00.000000 AM
13-JAN-93 12.00.00.000000 AM
03-JAN-90 12.00.00.000000 AM
21-MAY-91 12.00.00.000000 AM
25-JUN-97 12.00.00.000000 AM
05-FEB-98 12.00.00.000000 AM
07-FEB-99 12.00.00.000000 AM
17-AUG-94 12.00.00.000000 AM
16-AUG-94 12.00.00.000000 AM
28-SEP-97 12.00.00.000000 AM
30-SEP-97 12.00.00.000000 AM

…

TIMESTAMP WITH TIME ZONE Data Type

- TIMESTAMP WITH TIME ZONE is a variant of TIMESTAMP that includes a time zone displacement in its value.
- The time zone displacement is the difference, in hours and minutes, between local time and UTC.
- It is specified as:

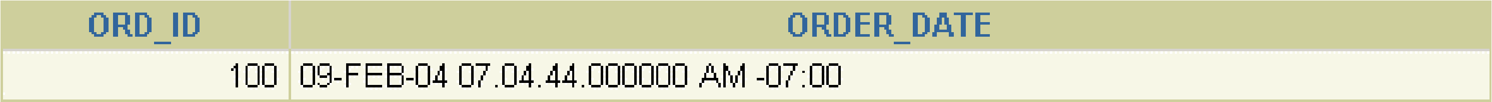
```
TIMESTAMP[(fractional_seconds_precision)]
WITH TIME ZONE
```

TIMESTAMP WITH TIMEZONE: Example

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

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

```
SELECT * FROM web_orders;
```



ORD_ID	ORDER_DATE
100	09-FEB-04 07.04.44.000000 AM -07:00

TIMESTAMP WITH LOCAL TIMEZONE

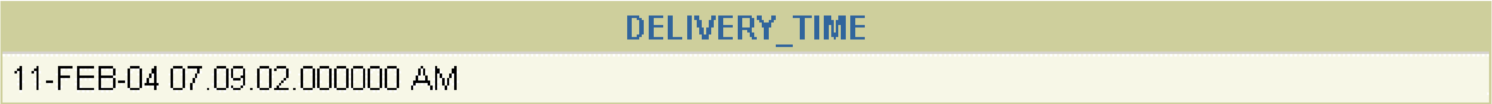
- `TIMESTAMP WITH LOCAL TIME ZONE` is another variant of `TIMESTAMP` that includes a time zone displacement in its value.
- Data stored in the database is normalized to the database time zone.
- The time zone displacement is not stored as part of the column data.
- The Oracle database returns the data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

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

TIMESTAMP WITH LOCAL TIMEZONE: Example

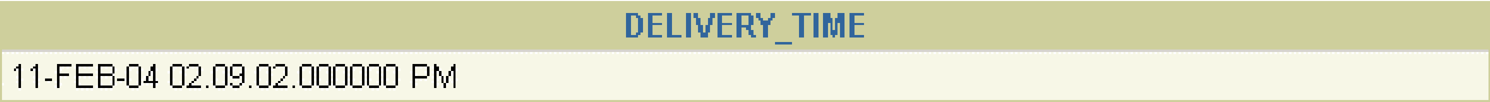
```
CREATE TABLE shipping (delivery_time TIMESTAMP WITH
LOCAL TIME ZONE);

INSERT INTO shipping VALUES(current_timestamp + 2);
```

```
SELECT * FROM shipping;
```



DELIVERY_TIME
11-FEB-04 07.09.02.000000 AM

```
ALTER SESSION SET TIME_ZONE = 'EUROPE/LONDON';

SELECT * FROM shipping;
```



DELIVERY_TIME
11-FEB-04 02.09.02.000000 PM

INTERVAL Data Types

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

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

INTERVAL 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` Data Type

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

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

- **For example:**

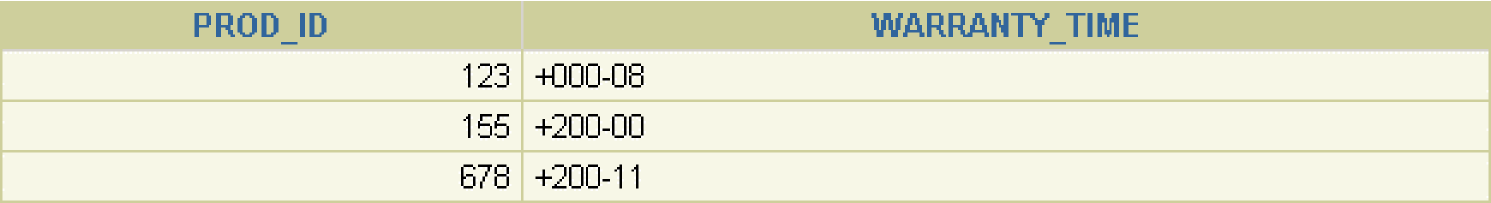
```
'312-2' assigned to INTERVAL YEAR(3) TO MONTH

Indicates an interval of 312 years and 2 months
```

```
'312-0' assigned to INTERVAL YEAR(3) TO MONTH

Indicates 312 years and 0 months
```

```
'0-3' assigned to INTERVAL YEAR TO MONTH

Indicates an interval of 3 months
```

INTERVAL YEAR TO MONTH: 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
123	+000-08
155	+200-00
678	+200-11

INTERVAL DAY TO SECOND Data Type

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

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

- For example:

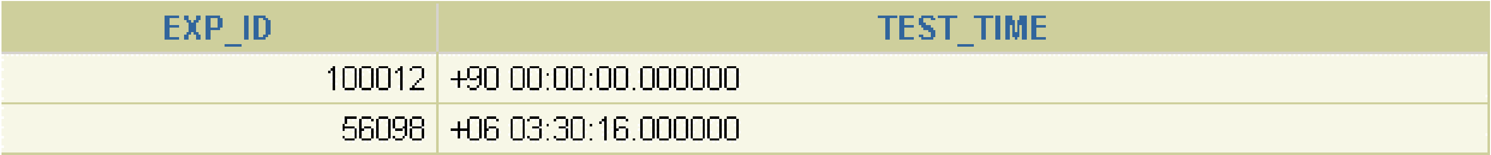
```
INTERVAL '6 03:30:16' DAY TO SECOND

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

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

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

INTERVAL DAY TO SECOND Data Type: 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;
```

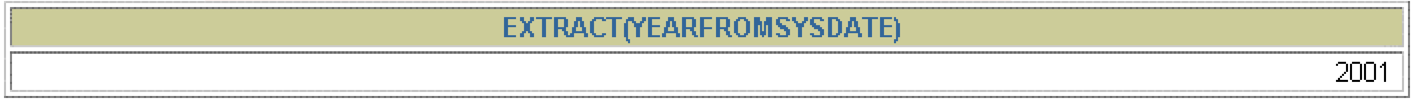


EXP_ID	TEST_TIME
100012	+90 00:00:00.000000
56098	+06 03:30:16.000000

EXTRACT

- Display the YEAR component from the SYSDATE.

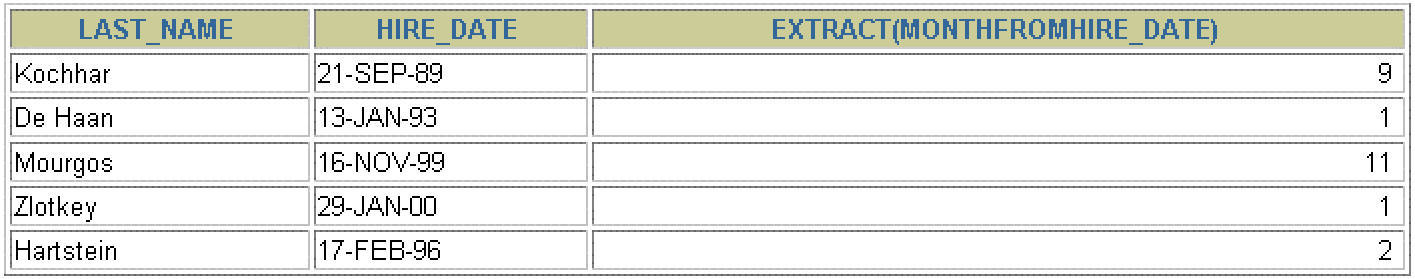
```
SELECT EXTRACT (YEAR FROM SYSDATE) FROM DUAL;
```



EXTRACT(YEARFROMSYSDATE)
2001

- 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;
```

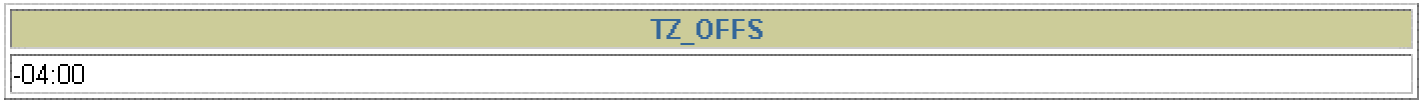


LAST_NAME	HIRE_DATE	EXTRACT(MONTHFROMHIRE_DATE)
Kochhar	21-SEP-89	9
De Haan	13-JAN-93	1
Mourgos	16-NOV-99	11
Zlotkey	29-JAN-00	1
Hartstein	17-FEB-96	2

TZ_OFFSET

- Display the time zone offset for the time zone `'US/Eastern'`.

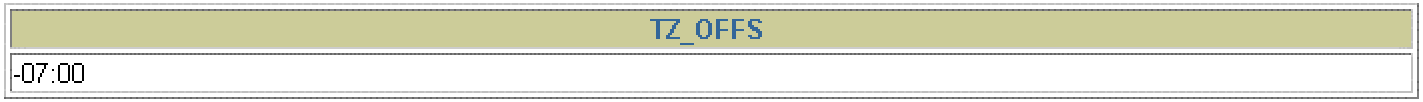
```
SELECT TZ_OFFSET('US/Eastern') FROM DUAL;
```



```
TZ_OFFS
-04:00
```

- Display the time zone offset for the time zone `'Canada/Yukon'`.

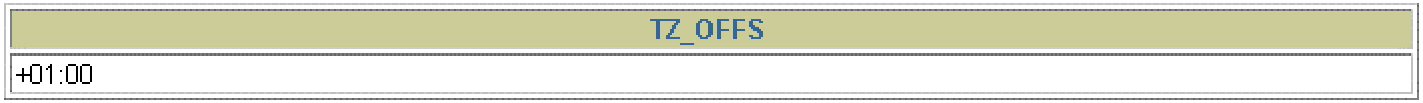
```
SELECT TZ_OFFSET('Canada/Yukon') FROM DUAL;
```



```
TZ_OFFS
-07:00
```

- Display the time zone offset for the time zone `'Europe/London'`.

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

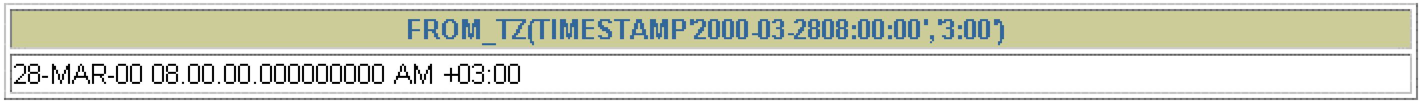


```
TZ_OFFS
+01:00
```

TIMESTAMP Conversion Using FROM_TZ

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

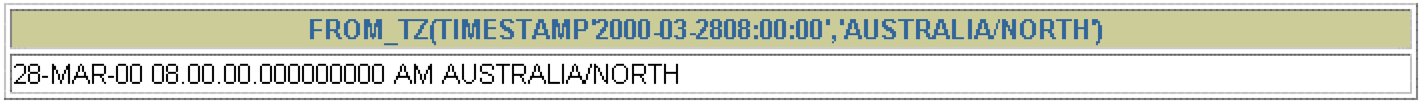
```
SELECT FROM_TZ(TIMESTAMP
         '2000-03-28 08:00:00','3:00')
FROM DUAL;
```



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

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

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

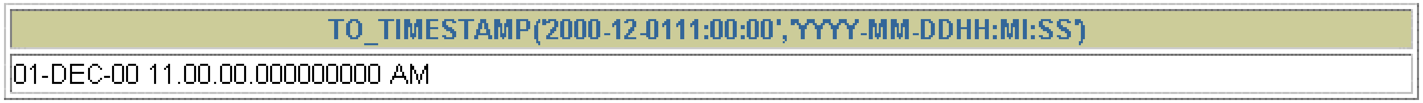


```
FROM_TZ(TIMESTAMP'2000-03-2808:00:00','AUSTRALIA/NORTH')
28-MAR-00 08.00.00.000000000 AM AUSTRALIA/NORTH
```

Converting to `TIMESTAMP` Using `TO_TIMESTAMP` and `TO_TIMESTAMP_TZ`

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

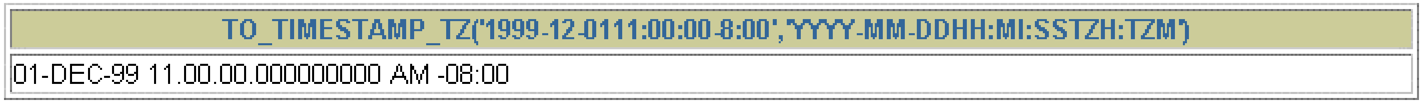
```
SELECT TO_TIMESTAMP ('2000-12-01 11:00:00',
                     'YYYY-MM-DD HH:MI:SS')
FROM DUAL;
```



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

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

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

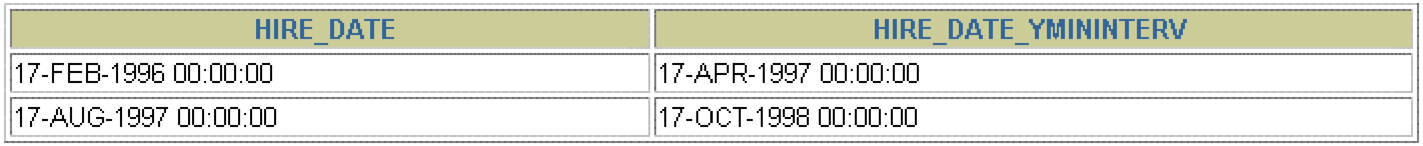


```
TO_TIMESTAMP_TZ('1999-12-0111:00:00-8:00','YYYY-MM-DDHH:MI:SSTZH:TZM')
01-DEC-99 11.00.00.000000000 AM -08:00
```

Time Interval Conversion with `TO_YMINTERVAL`

Display a date that is one year, 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;
```

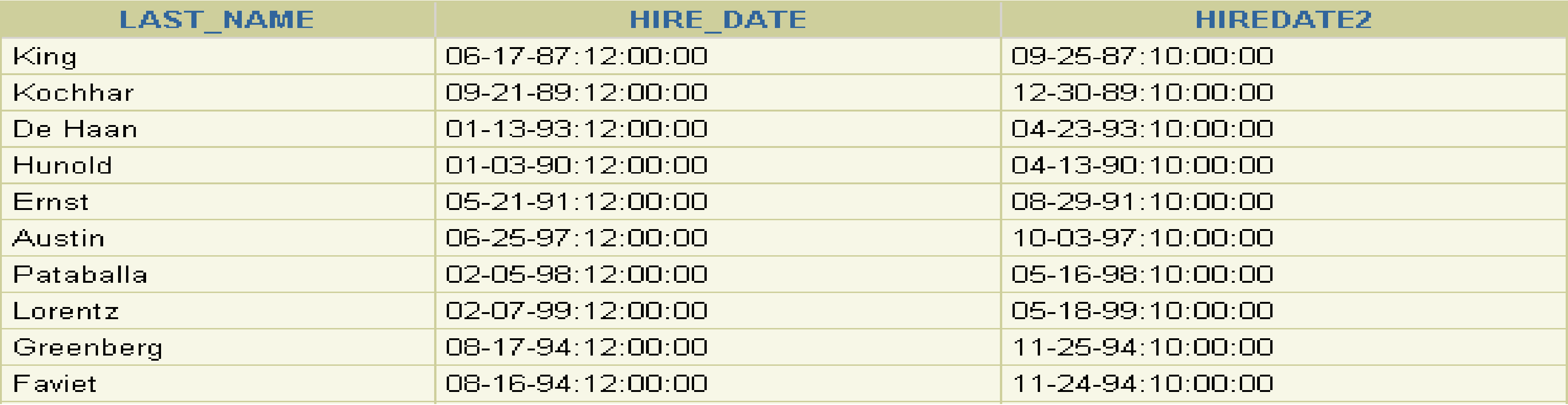


HIRE_DATE	HIRE_DATE_YMININTERV
17-FEB-1996 00:00:00	17-APR-1997 00:00:00
17-AUG-1997 00:00:00	17-OCT-1998 00:00:00

Using TO_DSINTERVAL: Example

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

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



LAST_NAME	HIRE_DATE	HIREDATE2
King	06-17-87:12:00:00	09-25-87:10:00:00
Kochhar	09-21-89:12:00:00	12-30-89:10:00:00
De Haan	01-13-93:12:00:00	04-23-93:10:00:00
Hunold	01-03-90:12:00:00	04-13-90:10:00:00
Ernst	05-21-91:12:00:00	08-29-91:10:00:00
Austin	06-25-97:12:00:00	10-03-97:10:00:00
Pataballa	02-05-98:12:00:00	05-16-98:10:00:00
Lorentz	02-07-99:12:00:00	05-18-99:10:00:00
Greenberg	08-17-94:12:00:00	11-25-94:10:00:00
Faviet	08-16-94:12:00:00	11-24-94:10:00:00

•••

Daylight Saving Time

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

Summary

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

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

Practice 5: Overview

This practice covers using the datetime functions.

6

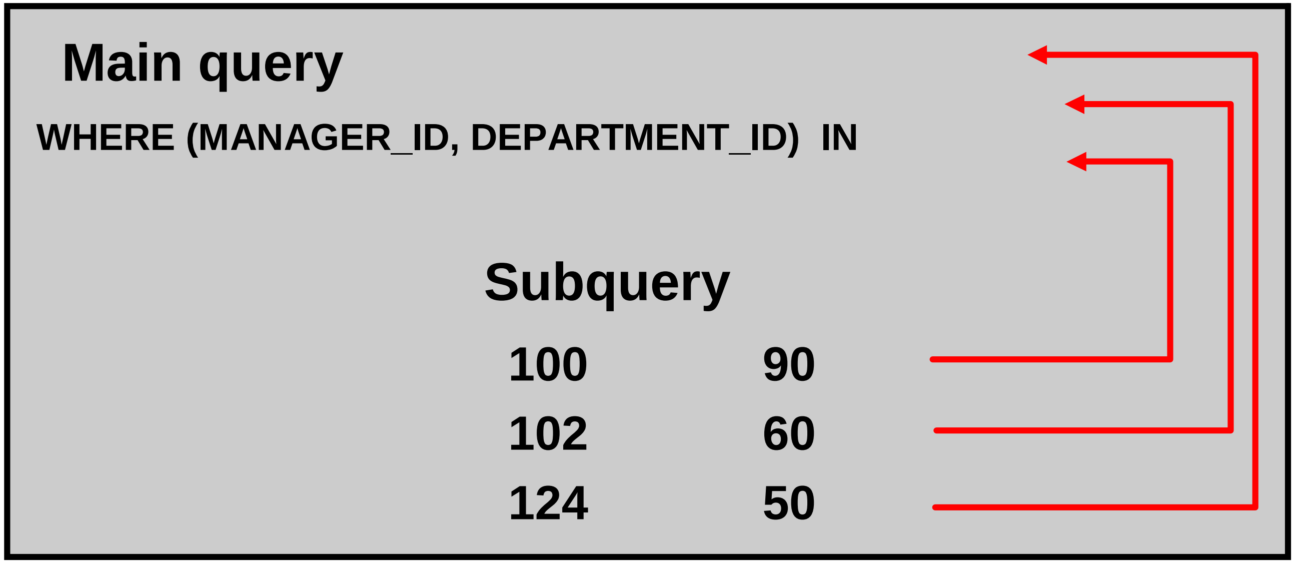
Retrieving Data Using Subqueries

Objectives

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

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

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.

Column Comparisons

Column comparisons in a multiple-column subquery can be:

- Pairwise comparisons
- Nonpairwise comparisons

Pairwise Comparison Subquery

Display the details of the employees who are managed by the same manager *and* work in the same department as the employees with `EMPLOYEE_ID` 199 or 174.

```
SELECT  employee_id, manager_id, department_id
FROM    employees
WHERE  (manager_id, department_id) IN
                      (SELECT manager_id, department_id
                       FROM   employees
                       WHERE  employee_id IN (199,174))
AND     employee_id NOT IN (199,174);
```

Nonpairwise Comparison Subquery

Display the details of the employees who are managed by the same manager as the employees with `EMPLOYEE_ID` 174 or 199 *and* work in the same department as the employees with `EMPLOYEE_ID` 174 or 199.

```
SELECT  employee_id, manager_id, department_id
FROM    employees
WHERE   manager_id IN
                  (SELECT   manager_id
                   FROM     employees
                   WHERE    employee_id IN (174,199))
AND     department_id IN
                  (SELECT   department_id
                   FROM     employees
                   WHERE    employee_id IN (174,199))

AND    employee_id NOT IN(174,199);
```

Scalar Subquery Expressions

- A scalar subquery expression is a subquery that returns exactly one column value from one row.
- Scalar subqueries can be used in:
  - Condition and expression part of `DECODE` and `CASE`
  - All clauses of `SELECT` except `GROUP BY`

Scalar Subqueries: Examples

- Scalar subqueries in `CASE` expressions

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

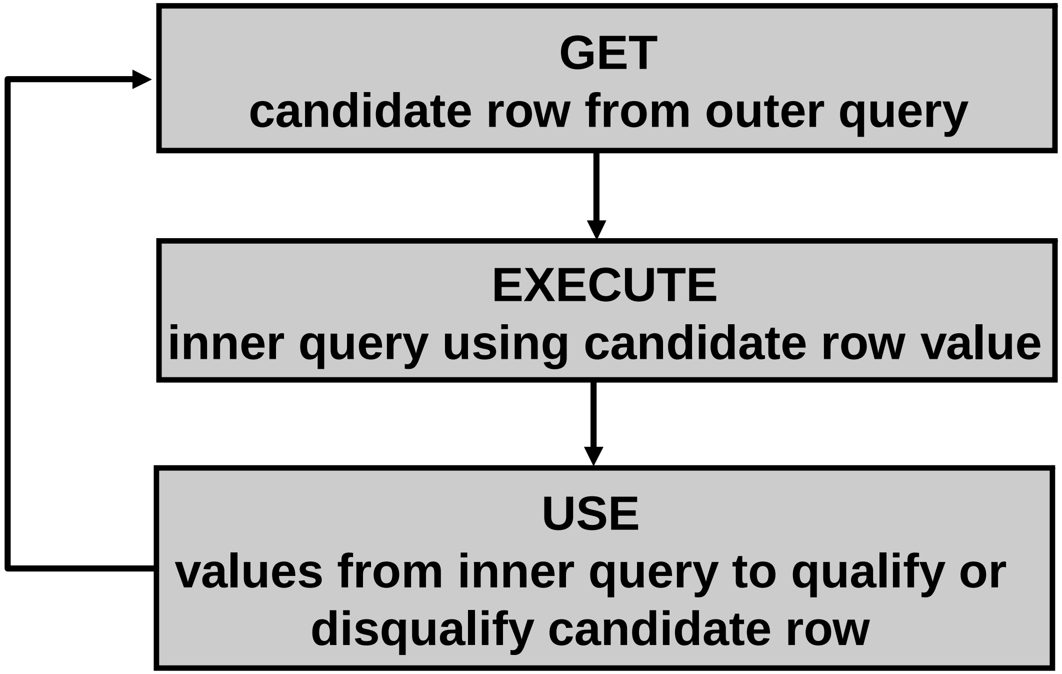
20

- Scalar subqueries in `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);
```

Correlated Subqueries

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



GET
candidate row from outer query

EXECUTE
inner query using candidate row value

USE
values from inner query to qualify or disqualify candidate row

Correlated Subqueries

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

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

Using Correlated Subqueries

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

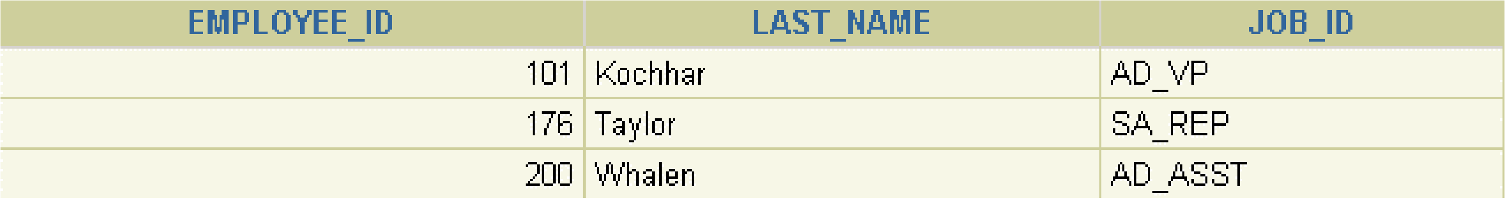
```
SELECT last_name, salary, department_id
FROM   employees outer
WHERE  salary >
                (SELECT AVG(salary)
                  FROM   employees
                  WHERE  department_id =
                  outer.department_id);
```

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

Using Correlated Subqueries

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);
```



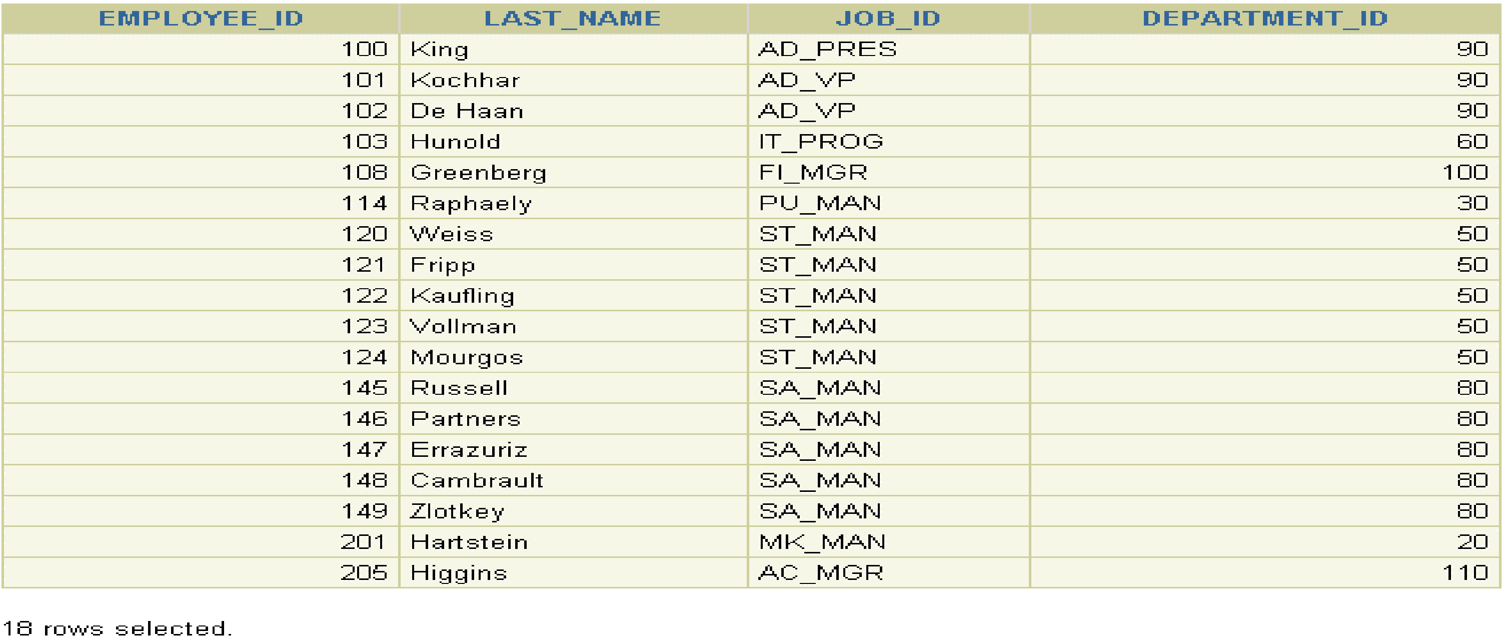
EMPLOYEE_ID	LAST_NAME	JOB_ID
101	Kochhar	AD_VP
176	Taylor	SA_REP
200	Whalen	AD_ASST

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

Find Employees Who Have at Least One Person Reporting to Them

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

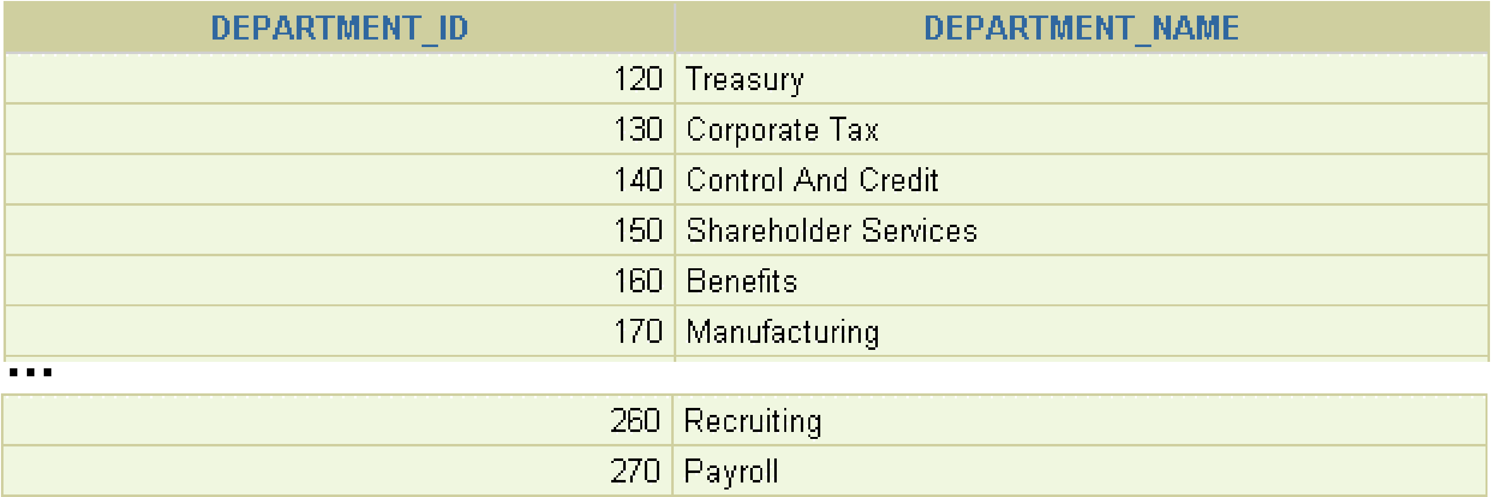


EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
108	Greenberg	FI_MGR	100
114	Raphaely	PU_MAN	30
120	Weiss	ST_MAN	50
121	Fripp	ST_MAN	50
122	Kaufling	ST_MAN	50
123	Vollman	ST_MAN	50
124	Mourgos	ST_MAN	50
145	Russell	SA_MAN	80
146	Partners	SA_MAN	80
147	Errazuriz	SA_MAN	80
148	Cambrault	SA_MAN	80
149	Zlotkey	SA_MAN	80
201	Hartstein	MK_MAN	20
205	Higgins	AC_MGR	110

18 rows selected.

Find All Departments That Do Not Have Any Employees

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



DEPARTMENT_ID	DEPARTMENT_NAME
120	Treasury
130	Corporate Tax
140	Control And Credit
150	Shareholder Services
160	Benefits
170	Manufacturing
...	
260	Recruiting
270	Payroll

16 rows selected.

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);
```

The `WITH` Clause

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

WITH Clause: 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;
```

Summary

In this lesson, you should have learned the following:

- 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

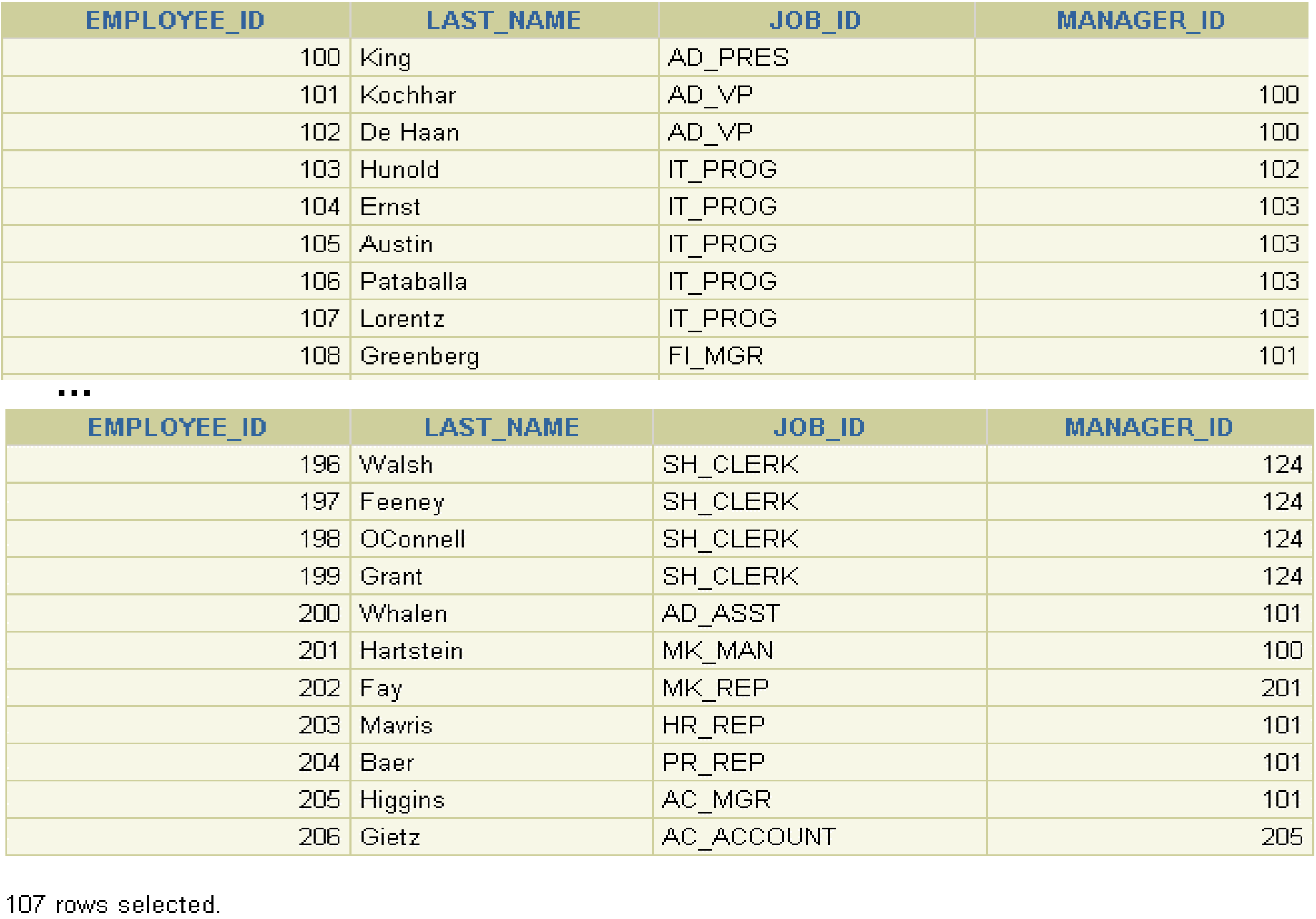
Hierarchical Retrieval

Objectives

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

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

Sample Data from the `EMPLOYEES` Table



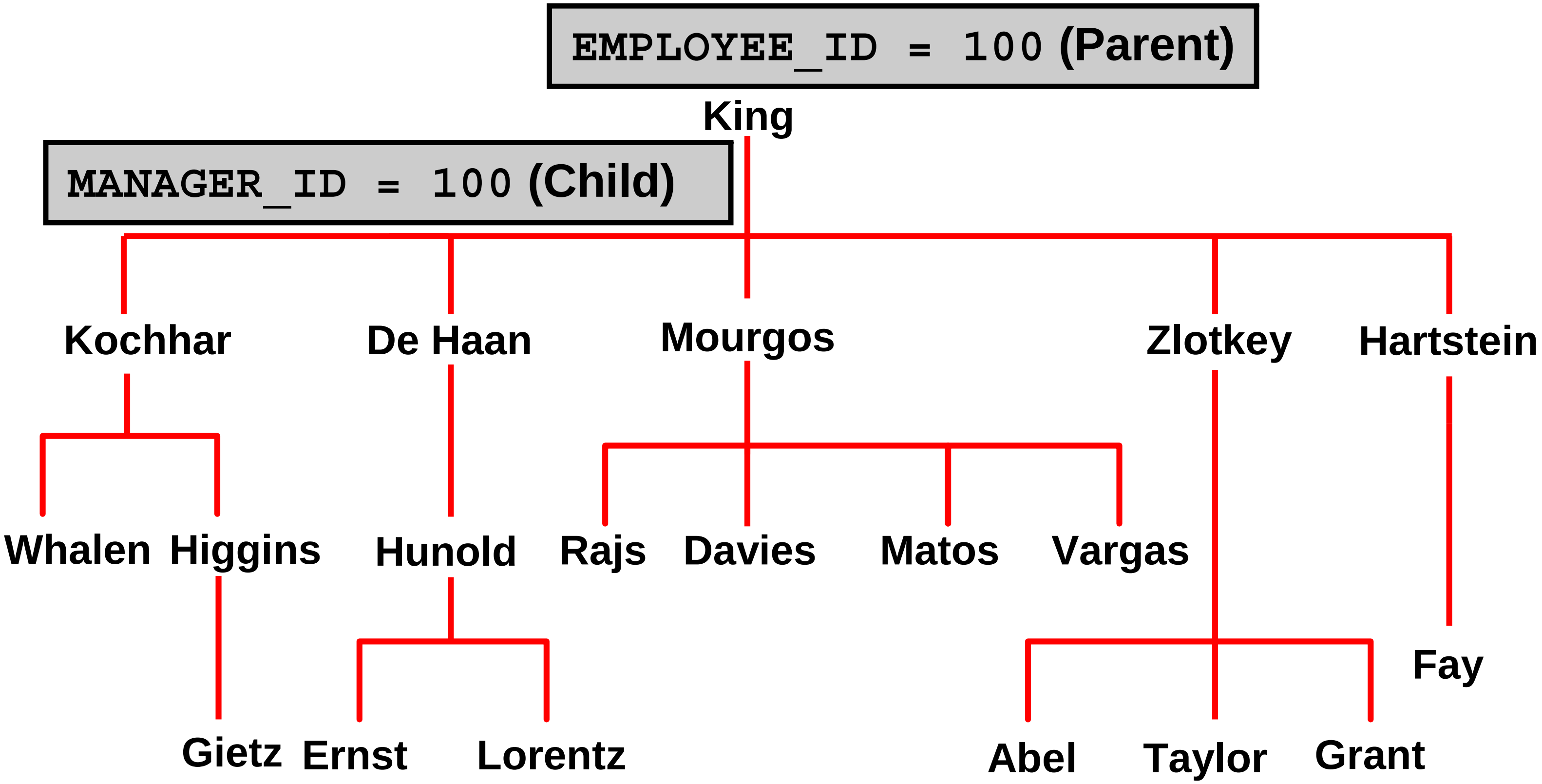
EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
100	King	AD_PRES	
101	Kochhar	AD_VP	100
102	De Haan	AD_VP	100
103	Hunold	IT_PROG	102
104	Ernst	IT_PROG	103
105	Austin	IT_PROG	103
106	Pataballa	IT_PROG	103
107	Lorentz	IT_PROG	103
108	Greenberg	FI_MGR	101

…

EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
196	Walsh	SH_CLERK	124
197	Feeney	SH_CLERK	124
198	OConnell	SH_CLERK	124
199	Grant	SH_CLERK	124
200	Whalen	AD_ASST	101
201	Hartstein	MK_MAN	100
202	Fay	MK_REP	201
203	Mavris	HR_REP	101
204	Baer	PR_REP	101
205	Higgins	AC_MGR	101
206	Gietz	AC_ACCOUNT	205

107 rows selected.

Natural Tree Structure



Hierarchical Queries

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

WHERE ***condition***:

```
expr comparison_operator expr
```

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
```

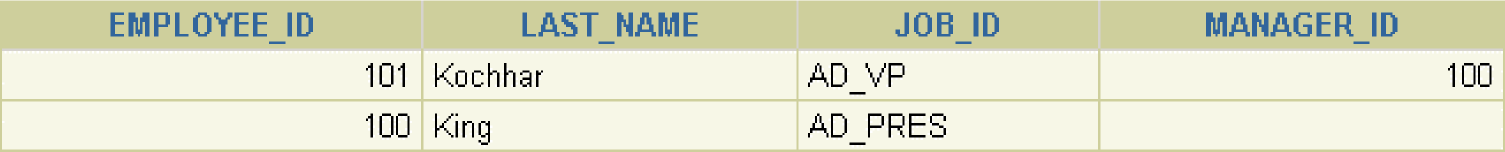
Direction

Top down → **Column1 = Parent Key**
Column2 = Child Key

Bottom up → **Column1 = Child Key**
Column2 = Parent Key

Walking the Tree: From the Bottom Up

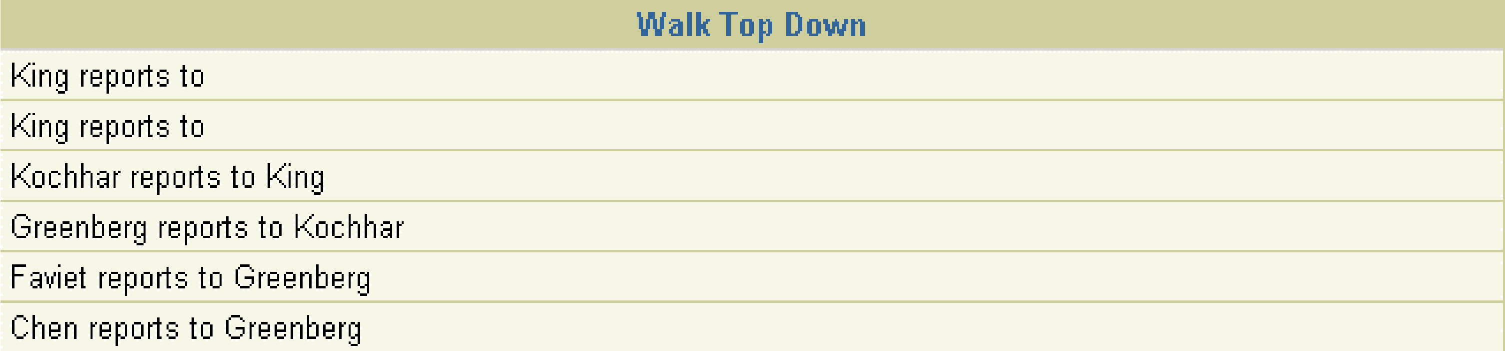
```
SELECT employee_id, last_name, job_id, manager_id
FROM   employees
START  WITH  employee_id = 101
CONNECT BY PRIOR manager_id = employee_id ;
```



EMPLOYEE_ID	LAST_NAME	JOB_ID	MANAGER_ID
101	Kochhar	AD_VP	100
100	King	AD_PRES	

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 ;
```

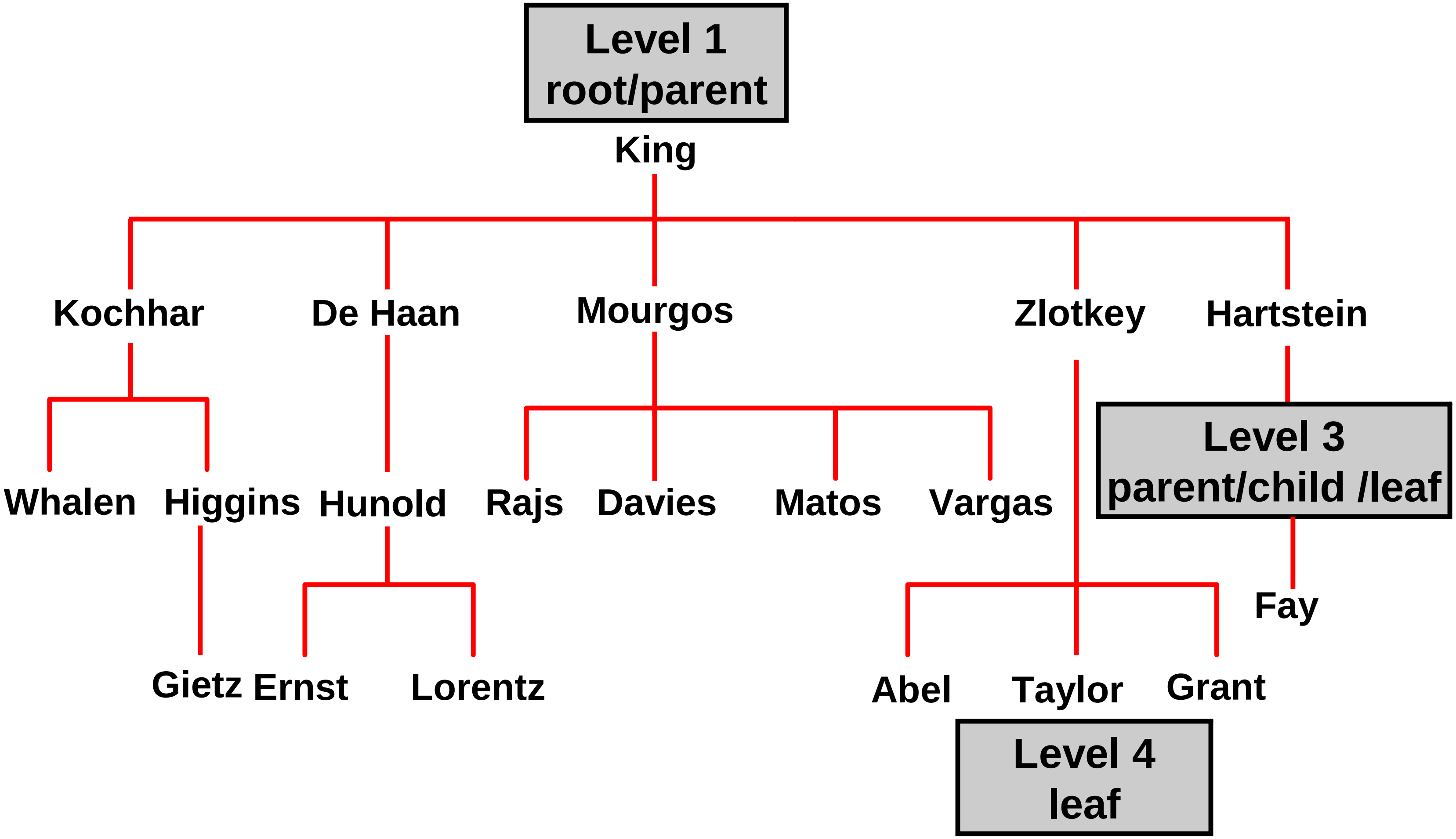


```
Walk Top Down
King reports to
King reports to
Kochhar reports to King
Greenberg reports to Kochhar
Faviet reports to Greenberg
Chen reports to Greenberg
```

...

108 rows selected.

Ranking Rows with the `LEVEL` Pseudocolumn



Formatting Hierarchical Reports Using `LEVEL` and `LPAD`

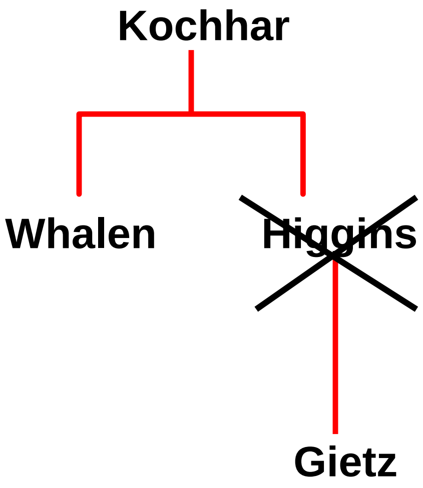
Create a report displaying company management levels, beginning with the highest level and indenting each of the following levels.

```
COLUMN org_chart FORMAT A12
SELECT LPAD(last_name, LENGTH(last_name)+(LEVEL*2)-2,'_')
       AS org_chart
FROM   employees
START WITH last_name='King'
CONNECT BY PRIOR employee_id=manager_id
```

Pruning Branches

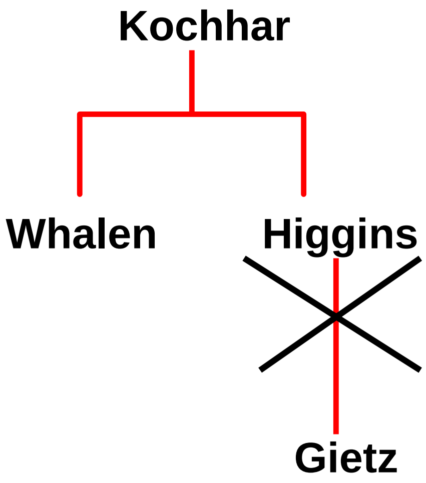
Use the WHERE clause to eliminate a node.

```
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 lesson, you should have learned the following:

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

Practice 7: Overview

This practice covers the following topics:

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

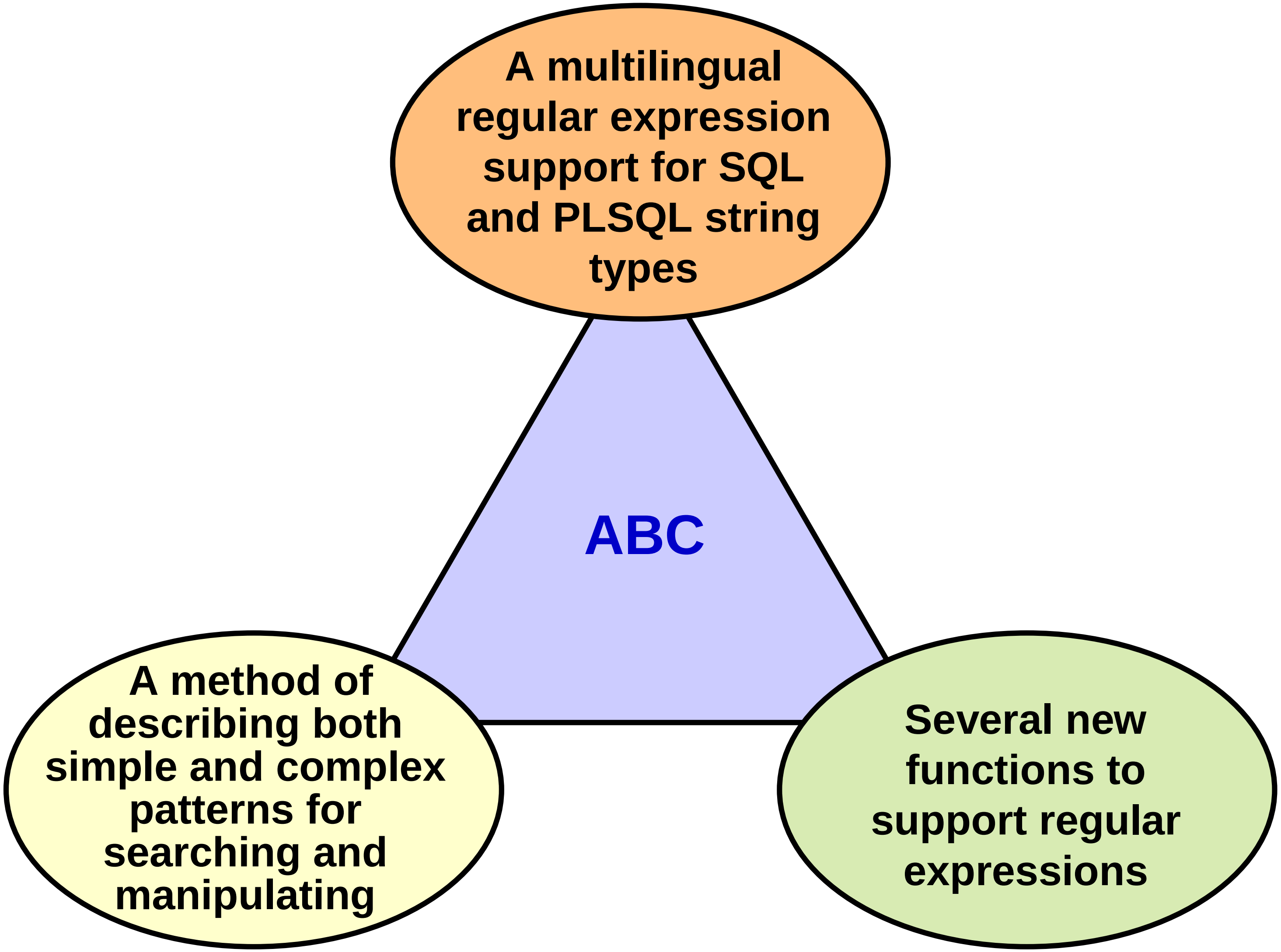
8

Regular Expression Support

Objectives

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

Regular Expression Overview



A multilingual regular expression support for SQL and PLSQL string types

ABC

A method of describing both simple and complex patterns for searching and manipulating

Several new functions to support regular expressions

Meta Characters

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

Using Meta Characters

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

1

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

2

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

3

Regular Expression Functions

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

The REGEXP Function Syntax

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

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

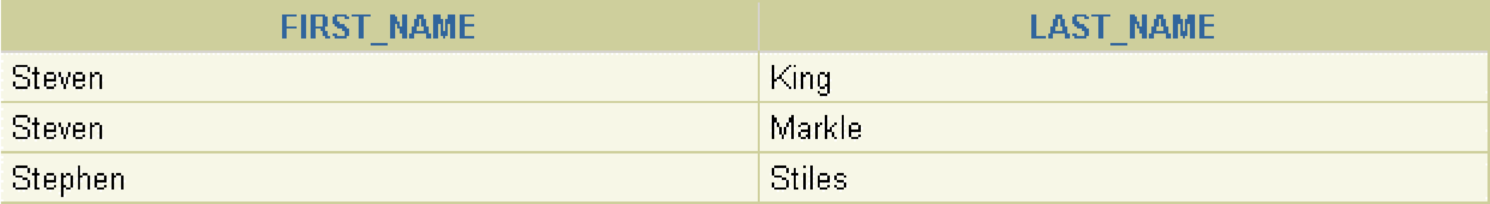
```
REGEXP_SUBSTR (srcstr, pattern [, position
               [, occurrence [, match_option]]])
```

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

ORACLE

Performing Basic Searches

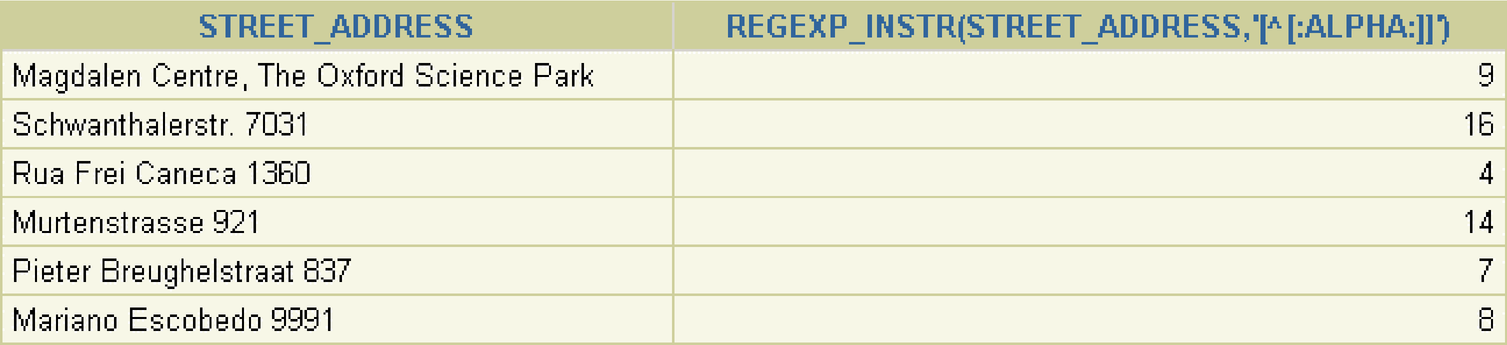
```
SELECT first_name, last_name
FROM employees
WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$');
```



FIRST_NAME	LAST_NAME
Steven	King
Steven	Markle
Stephen	Stiles

Checking the Presence of a Pattern

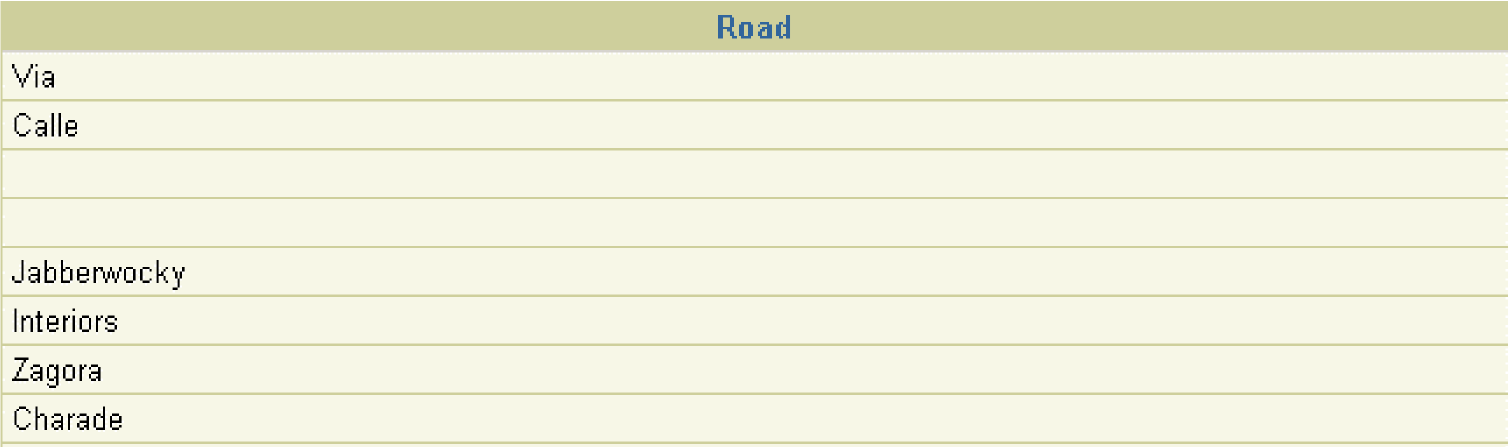
```
SELECT street_address,
   REGEXP_INSTR(street_address,'[^[:alpha:]]')
FROM   locations
WHERE
   REGEXP_INSTR(street_address,'[^[:alpha:]]')> 1;
```



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

Example of Extracting Substrings

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

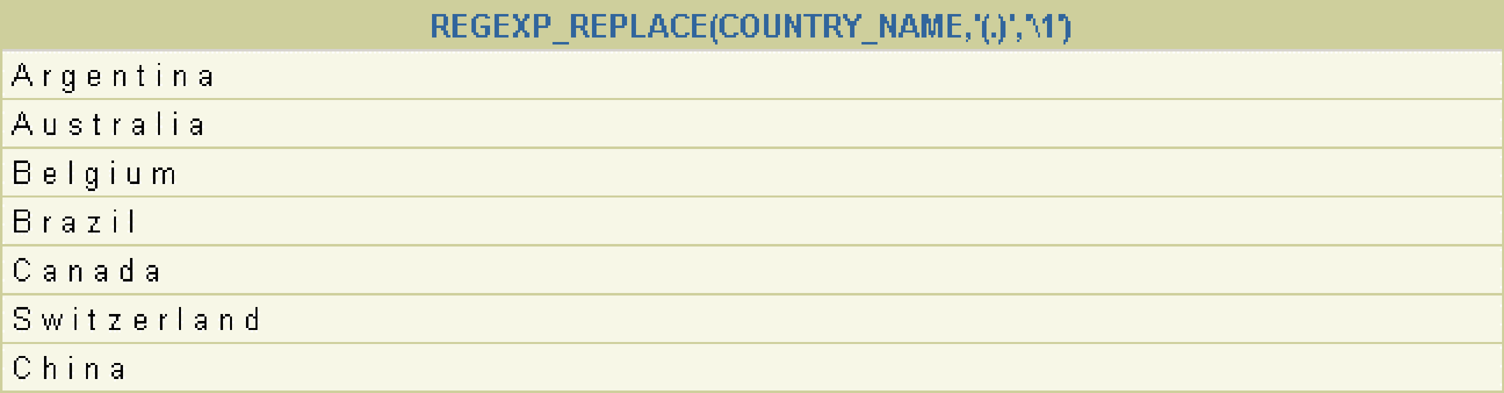


```
Road
Via
Calle


Jabberwocky
Interiors
Zagora
Charade
```

…

Replacing Patterns

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



REGEXP_REPLACE(COUNTRY_NAME,'(.)','\1')
A r g e n t i n a
A u s t r a l i a
B e l g i u m
B r a z i l
C a n a d a
S w i t z e r l a n d
C h i n a

…

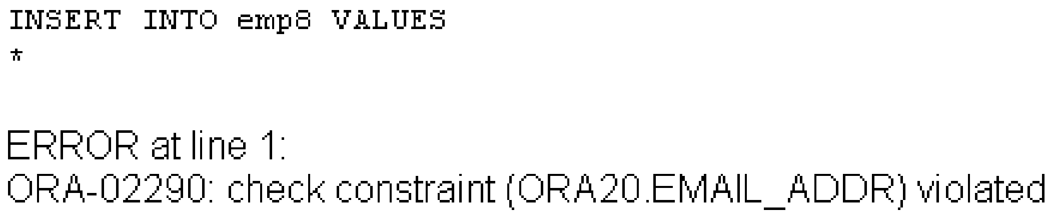
Regular Expressions and Check Constraints

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

1

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

2



```
INSERT INTO emp8 VALUES
*

ERROR at line 1:
ORA-02290: check constraint (ORA20.EMAIL_ADDR) violated
```

Summary

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

Practice 8: Overview

This practice covers using regular expressions.

C

Writing Advanced Scripts

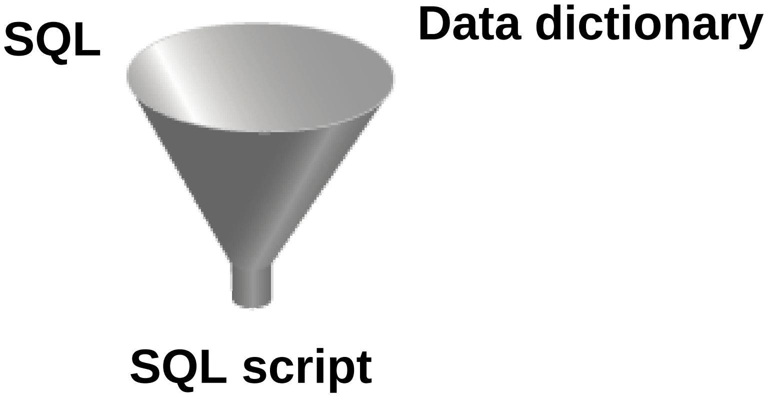
Objectives

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

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

Using SQL to Generate SQL

- SQL can be used to generate scripts in SQL
- The data dictionary:
  - Is a collection of tables and views that contain database information
  - Is created and maintained by the Oracle server



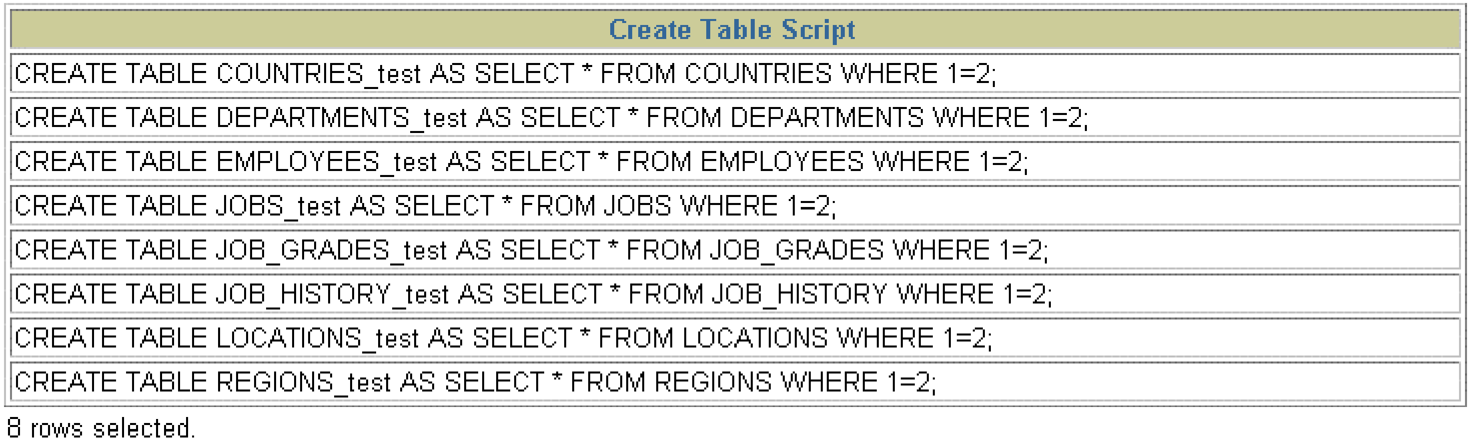
SQL

Data dictionary

SQL script

Creating a Basic Script

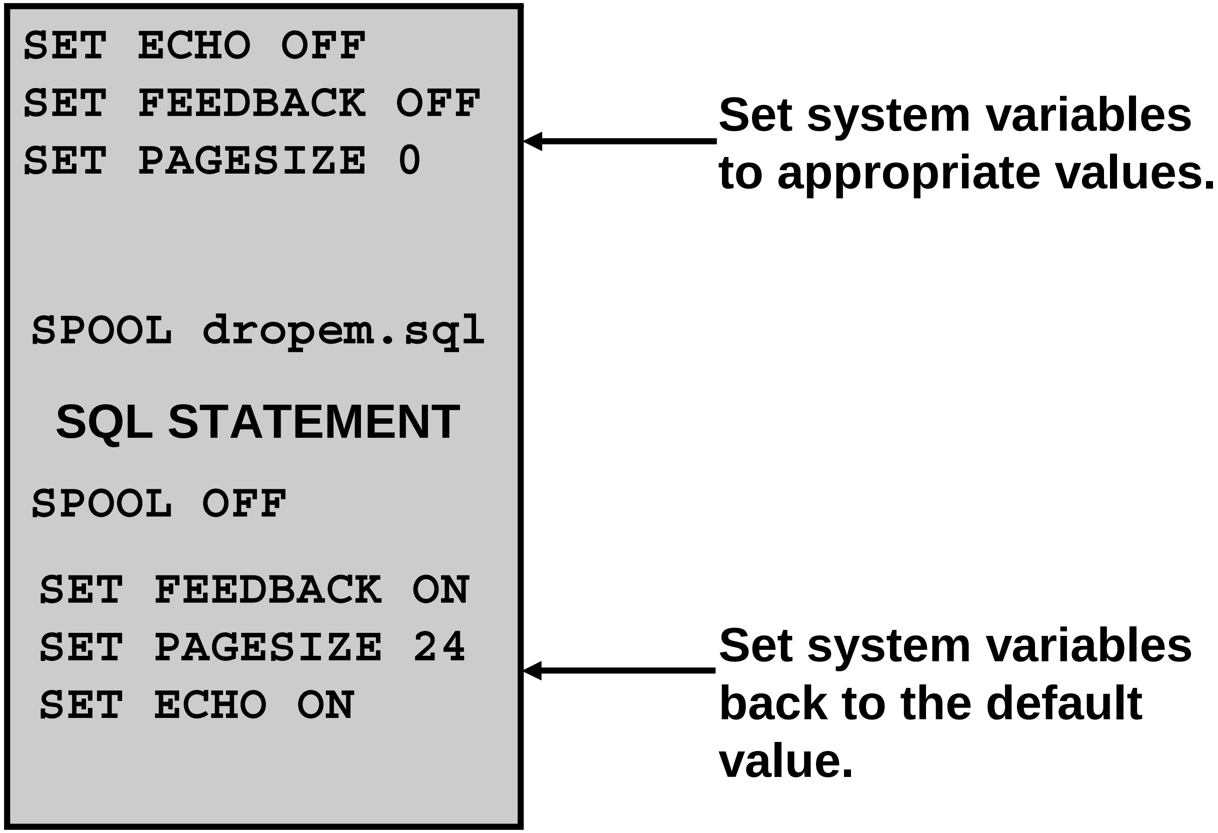
```
SELECT 'CREATE TABLE ' || table_name ||
       '_test ' || 'AS SELECT * FROM '
       || table_name ||' WHERE 1=2;'
       AS "Create Table Script"
FROM   user_tables;
```



Create Table Script
CREATE TABLE COUNTRIES_test AS SELECT * FROM COUNTRIES WHERE 1=2;
CREATE TABLE DEPARTMENTS_test AS SELECT * FROM DEPARTMENTS WHERE 1=2;
CREATE TABLE EMPLOYEES_test AS SELECT * FROM EMPLOYEES WHERE 1=2;
CREATE TABLE JOBS_test AS SELECT * FROM JOBS WHERE 1=2;
CREATE TABLE JOB_GRADES_test AS SELECT * FROM JOB_GRADES WHERE 1=2;
CREATE TABLE JOB_HISTORY_test AS SELECT * FROM JOB_HISTORY WHERE 1=2;
CREATE TABLE LOCATIONS_test AS SELECT * FROM LOCATIONS WHERE 1=2;
CREATE TABLE REGIONS_test AS SELECT * FROM REGIONS WHERE 1=2;

8 rows selected.

Controlling the Environment



```
SET ECHO OFF
SET FEEDBACK OFF
SET PAGESIZE 0


SPOOL dropem.sql
```

SQL STATEMENT

```
SPOOL OFF

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

Set system variables to appropriate values.

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 the following:

- 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.

D

Oracle Architectural Components

Objectives

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

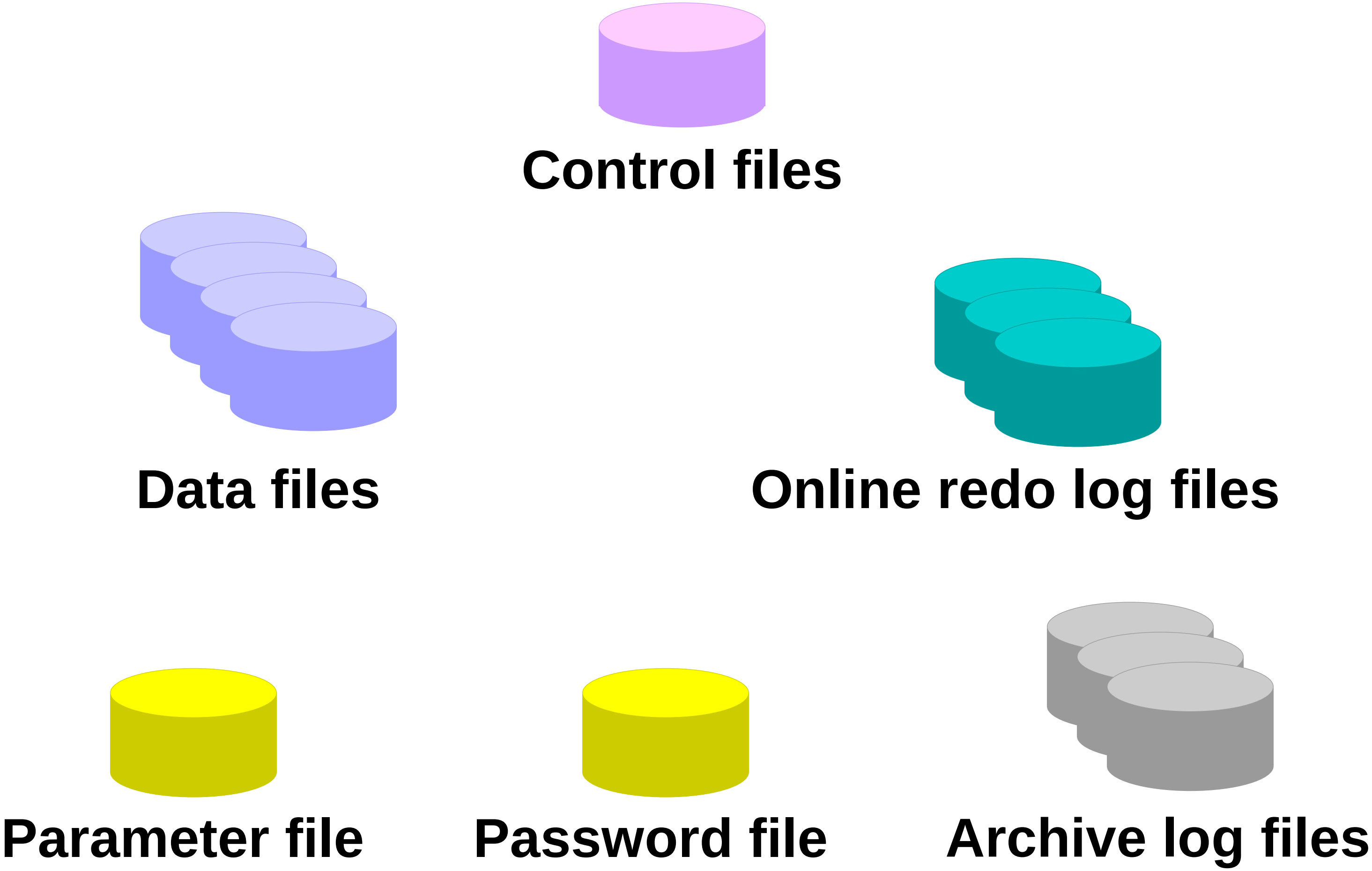
- Describe the Oracle server architecture and its main components
- List the structures involved in connecting a user to an Oracle instance
- List the stages in processing:
  - Queries
  - DML statements
  - Commits

Oracle Database Architecture: Overview

The Oracle database consists of two main components:

- The database or the physical structures
- The instance or the memory structures

Database Physical Architecture



Control files

Data files

Online redo log files

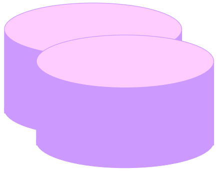
Parameter file

Password file

Archive log files

Control Files

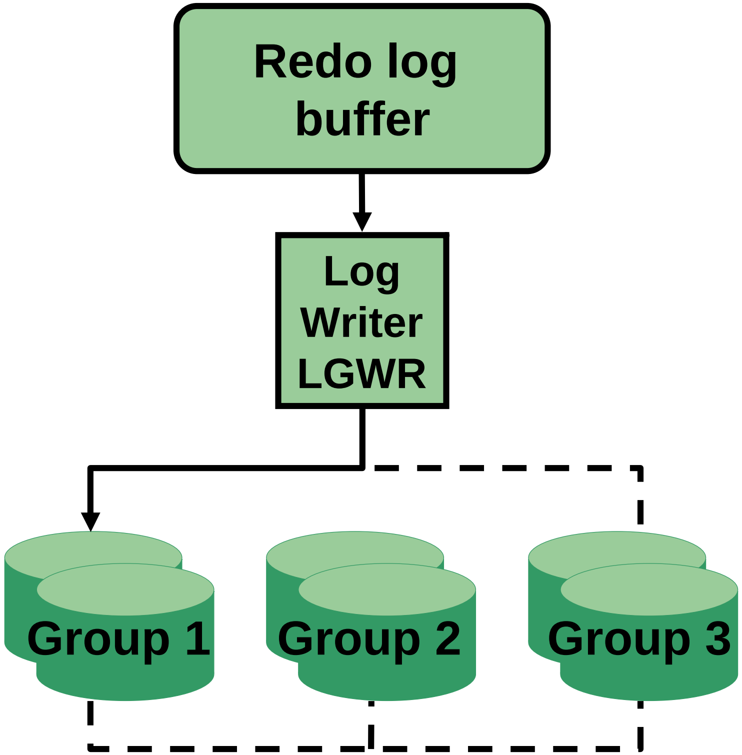
- Contains physical database structure information
- Multiplexed to protect against loss
- Read at mount stage



Control files

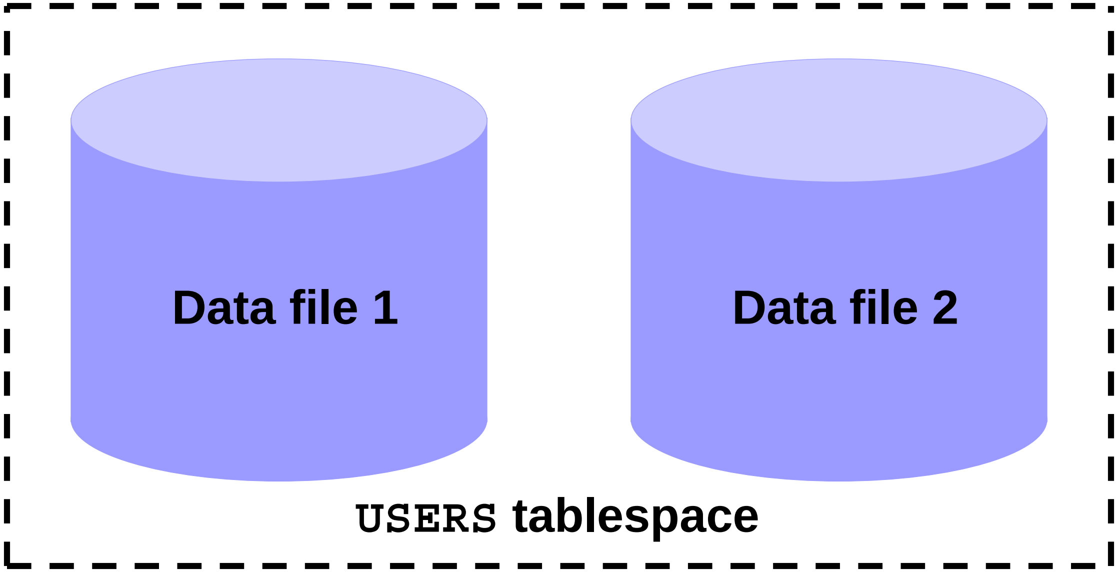
Redo Log Files

- Record changes to the database
- Multiplexed to protect against loss



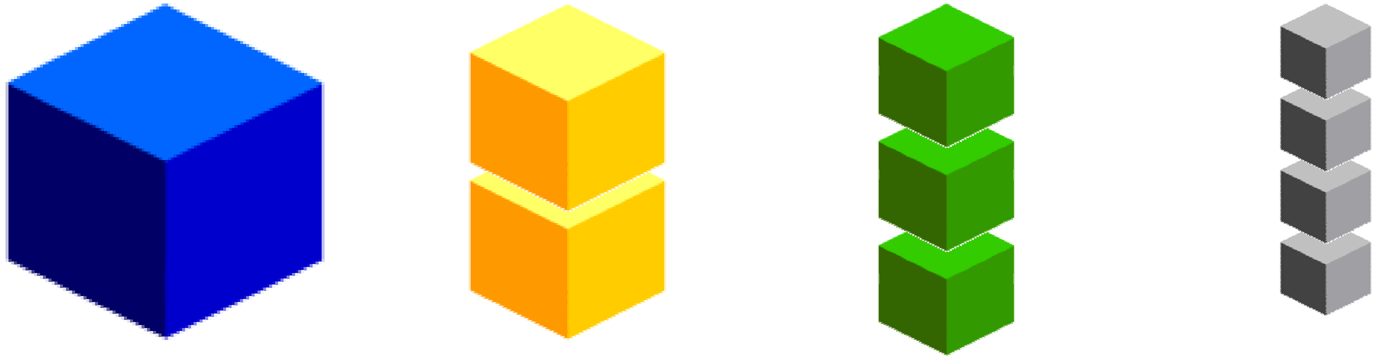
Tablespaces and Data Files

- Tablespaces consist of one or more data files.
- Data files belong to only one tablespace.



Segments, Extents, and Blocks

- Segments exist within a tablespace.
- Segments consist of a collection of extents.
- Extents are a collection of data blocks.
- Data blocks are mapped to OS blocks.



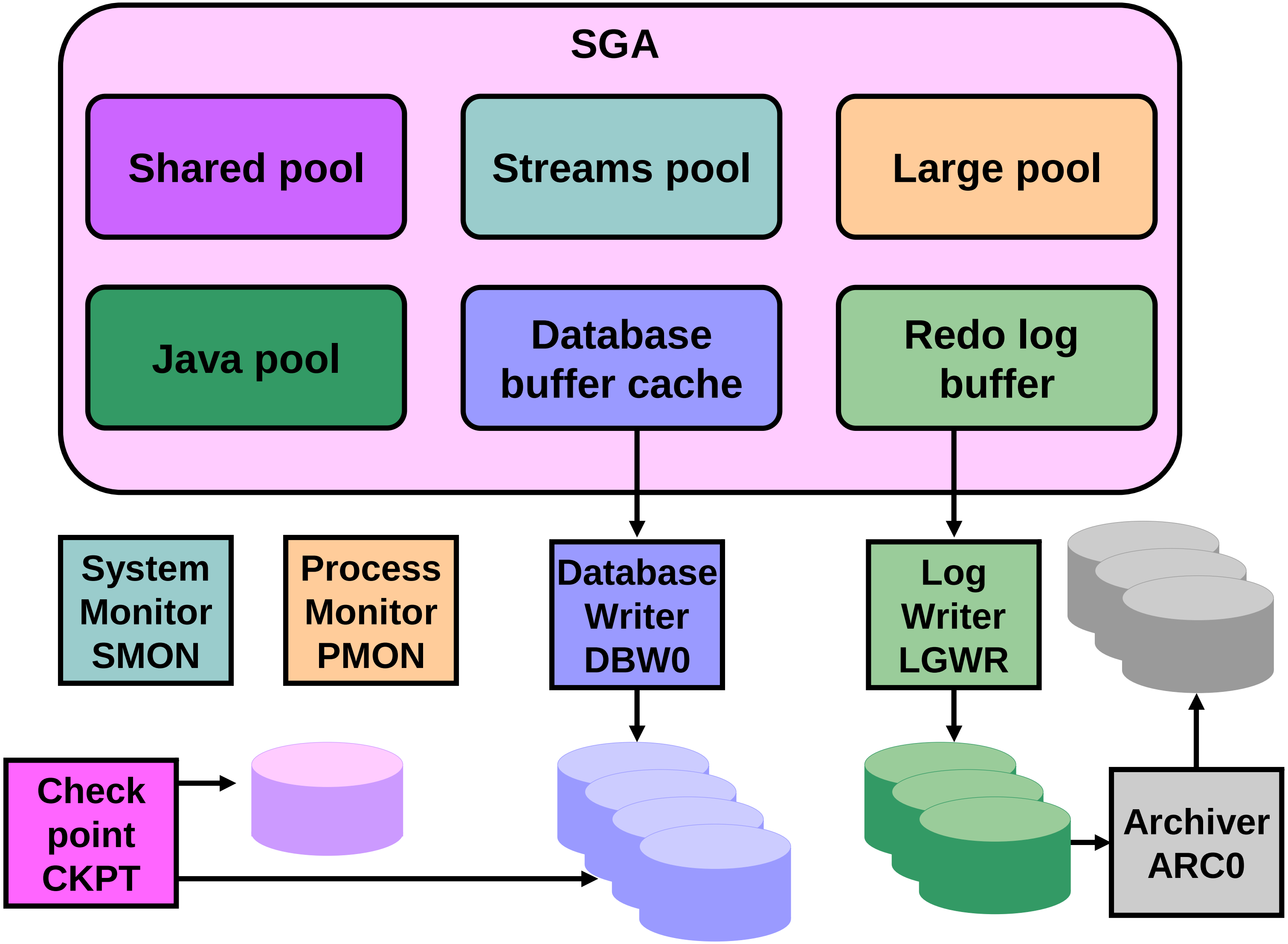
Segment

Extents

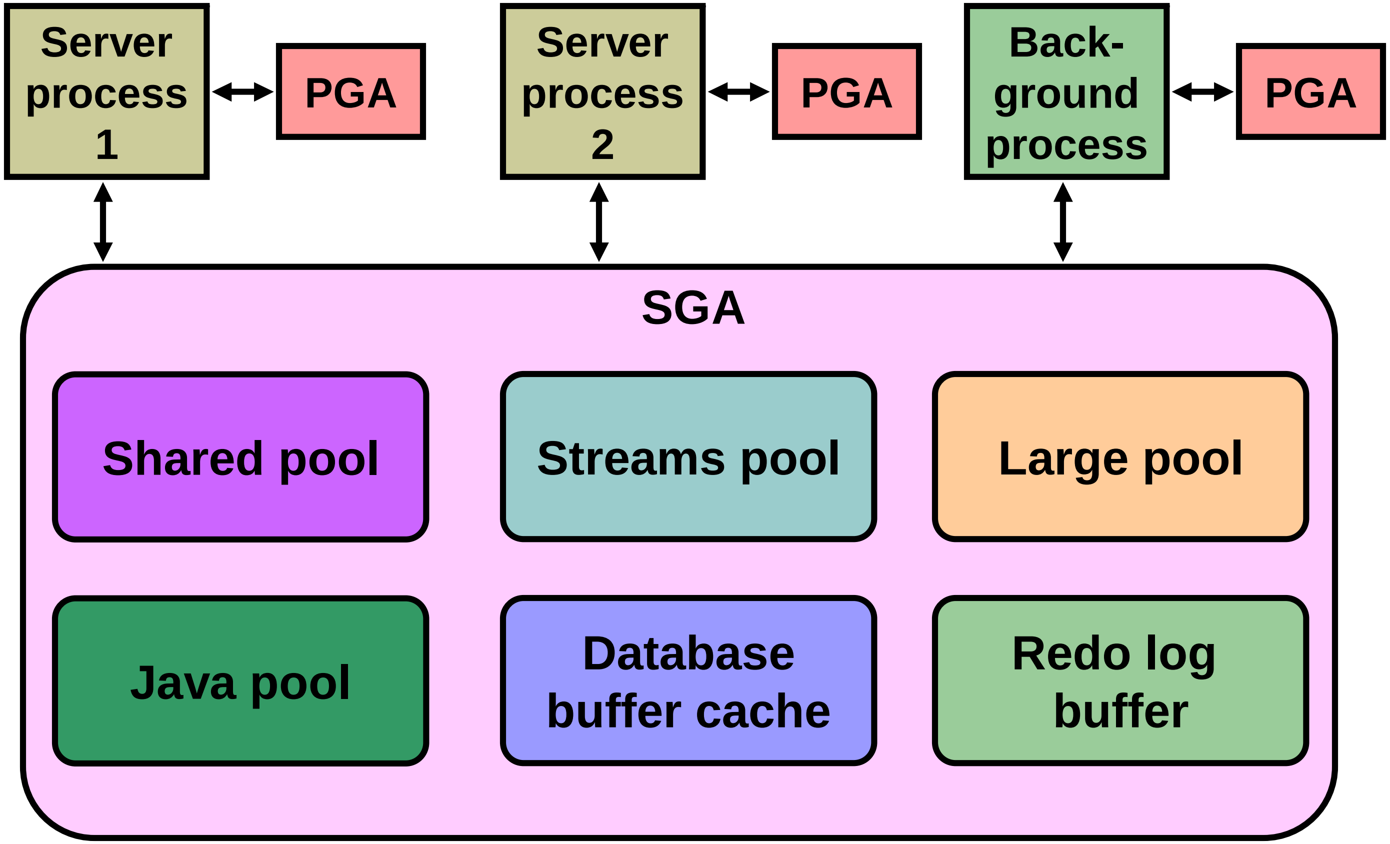
Data blocks

OS blocks

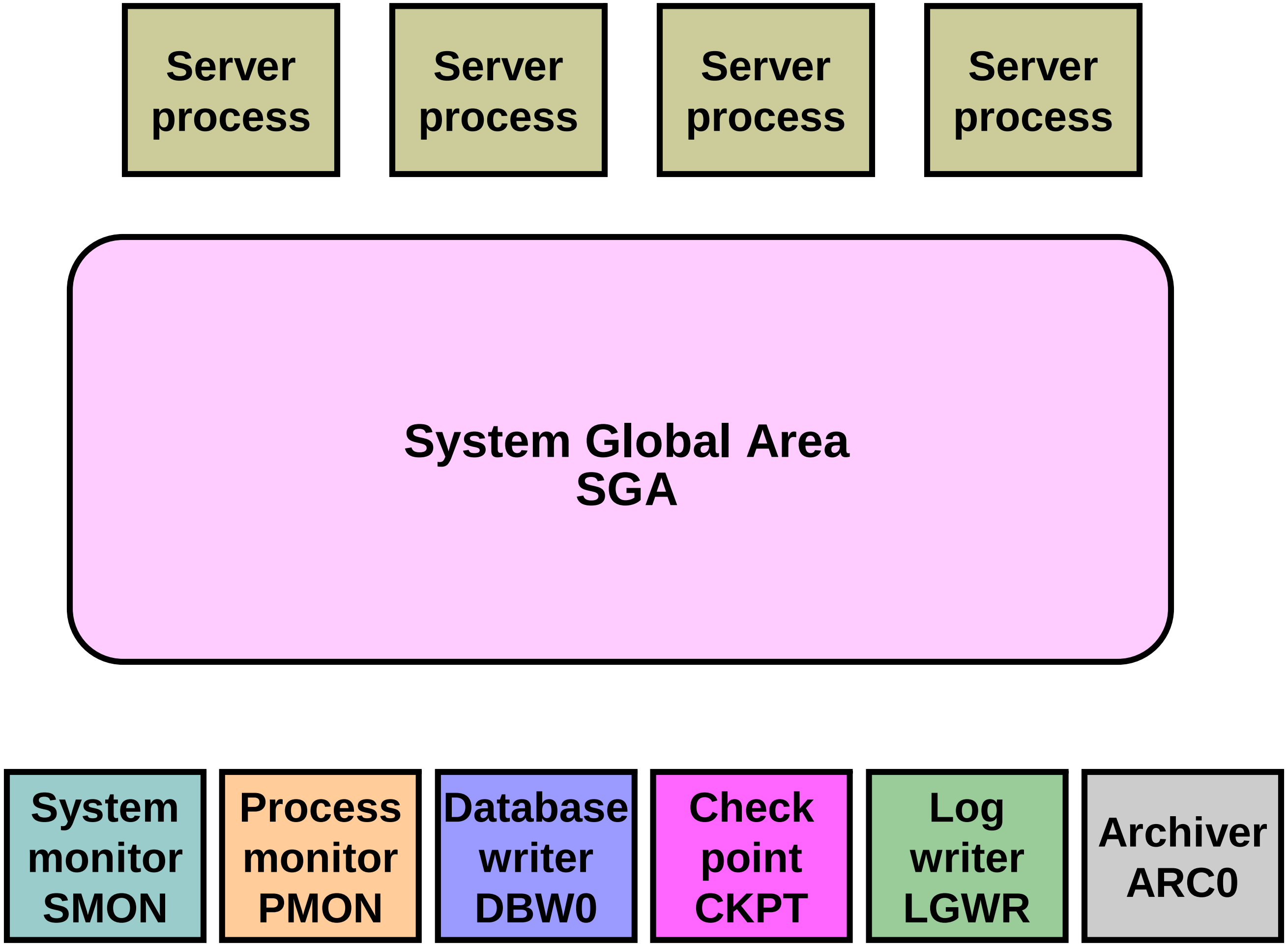
Oracle Instance Management



Oracle Memory Structures

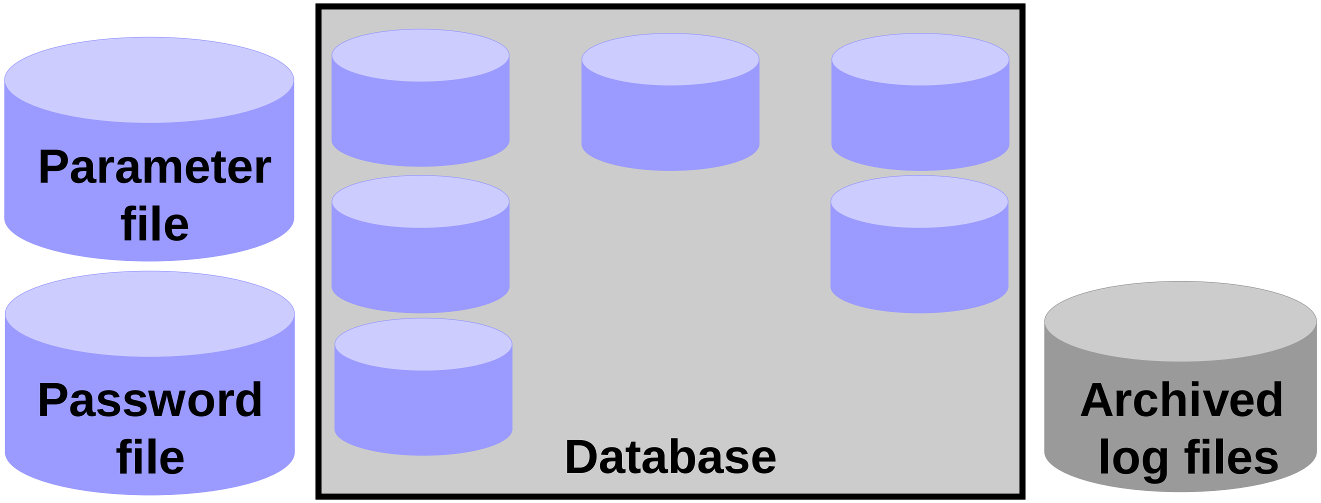


Oracle Processes



Background processes

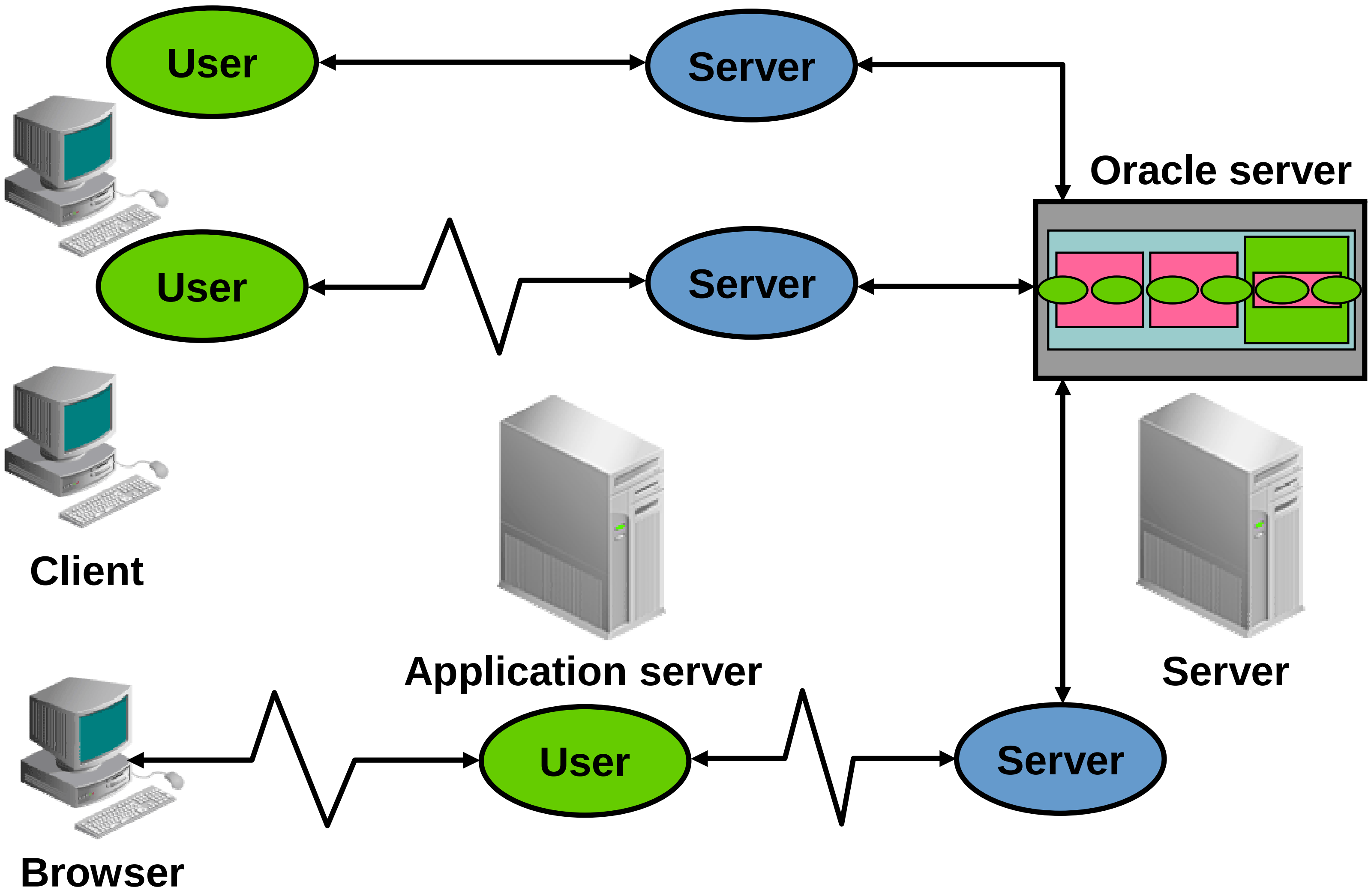
Other Key Physical Structures



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
  - DML statements log changes
  - Commit ensures transaction recovery
- Some Oracle server components do not participate in SQL statement processing.

Connecting to an Instance

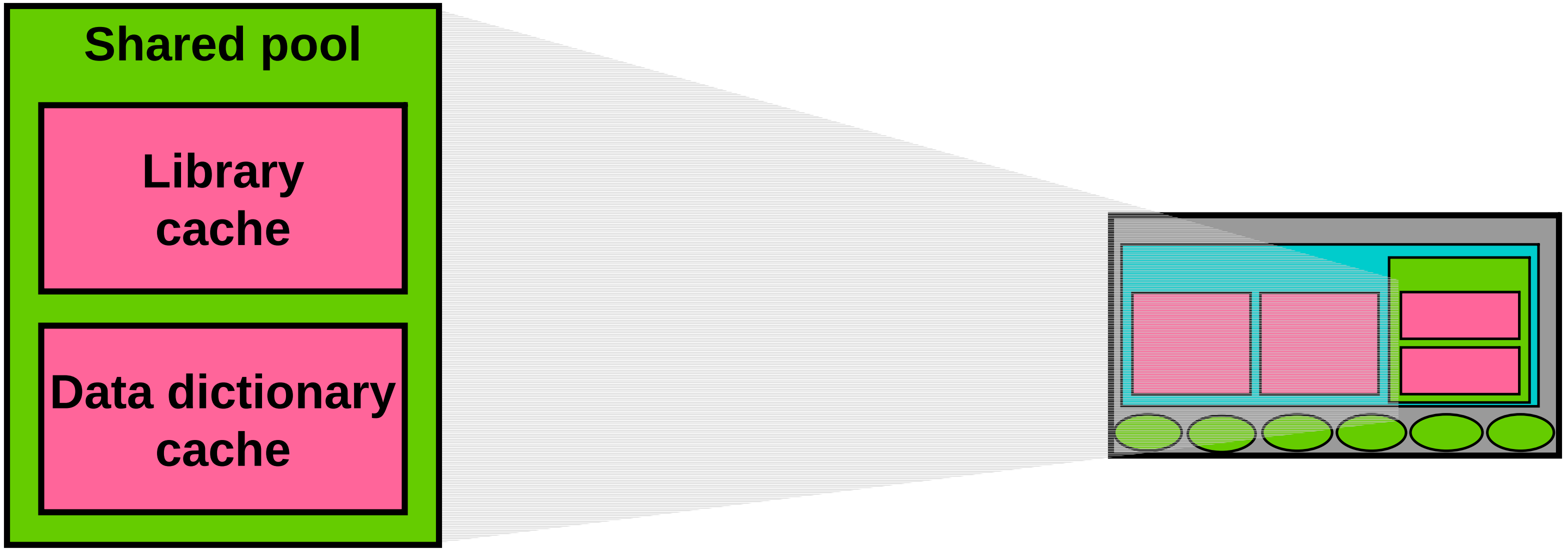


Processing a Query

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

The 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`.



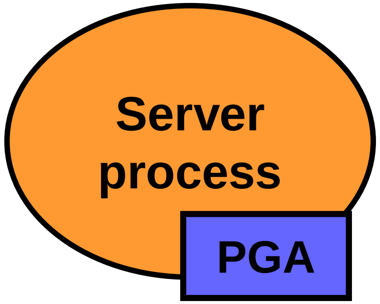
Database Buffer Cache

- Stores the most recently used blocks
- Size of a buffer based on `DB_BLOCK_SIZE`
- Number of buffers defined by `DB_BLOCK_BUFFERS`

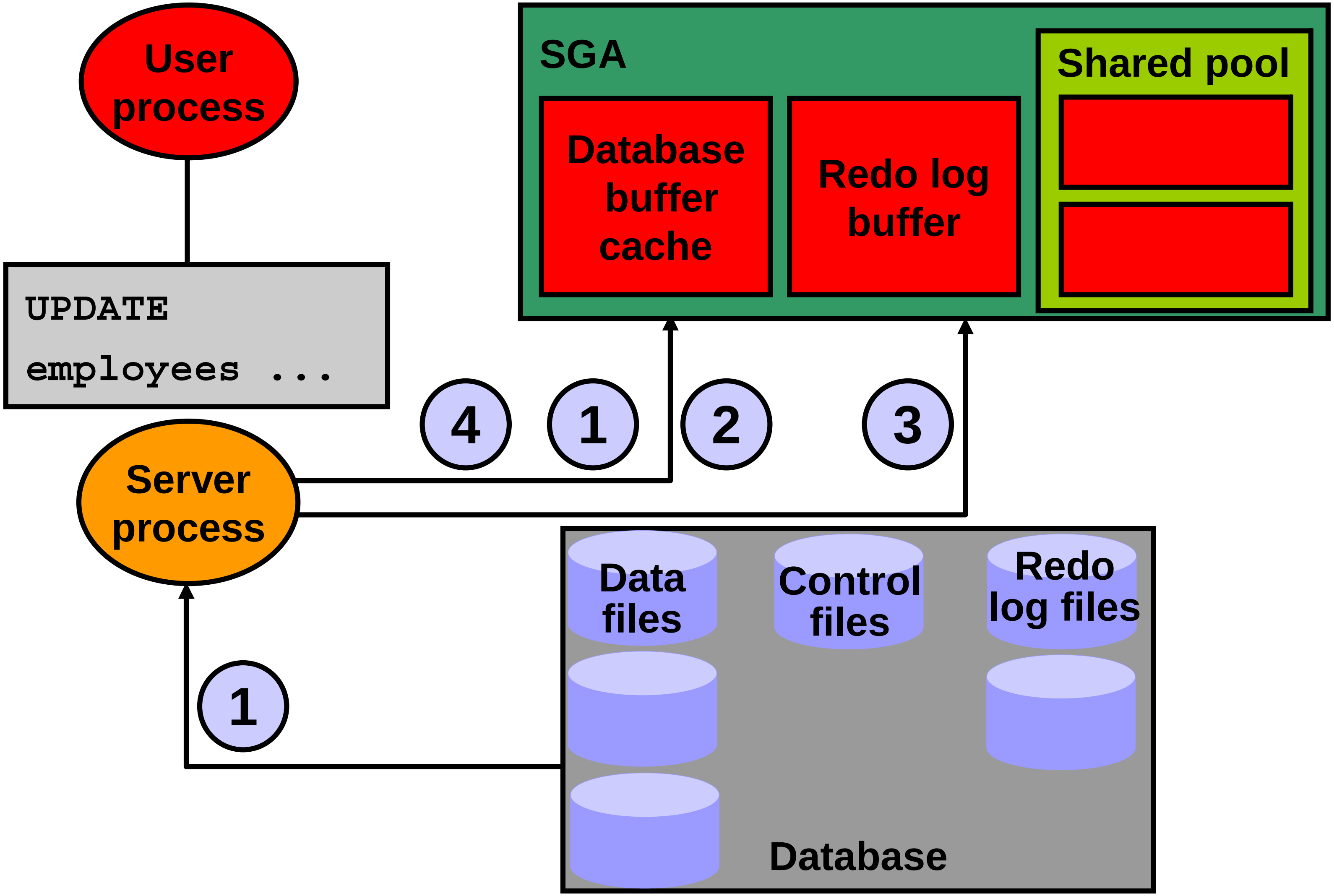
Database buffer cache

Program Global Area (PGA)

- Not shared
- Writable only by the server process
- Contains:
  - Sort area
  - Session information
  - Cursor state
  - Stack space



Processing a DML Statement

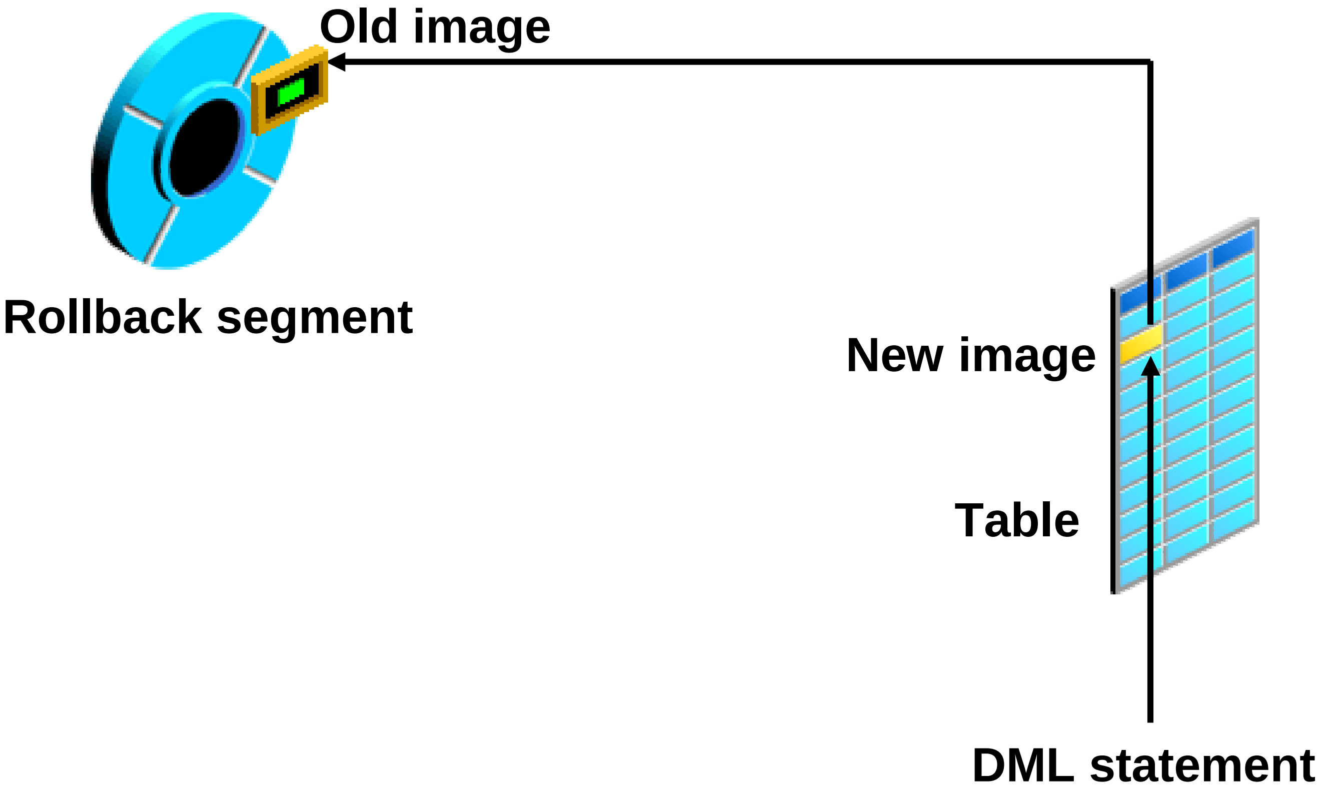


Redo Log Buffer

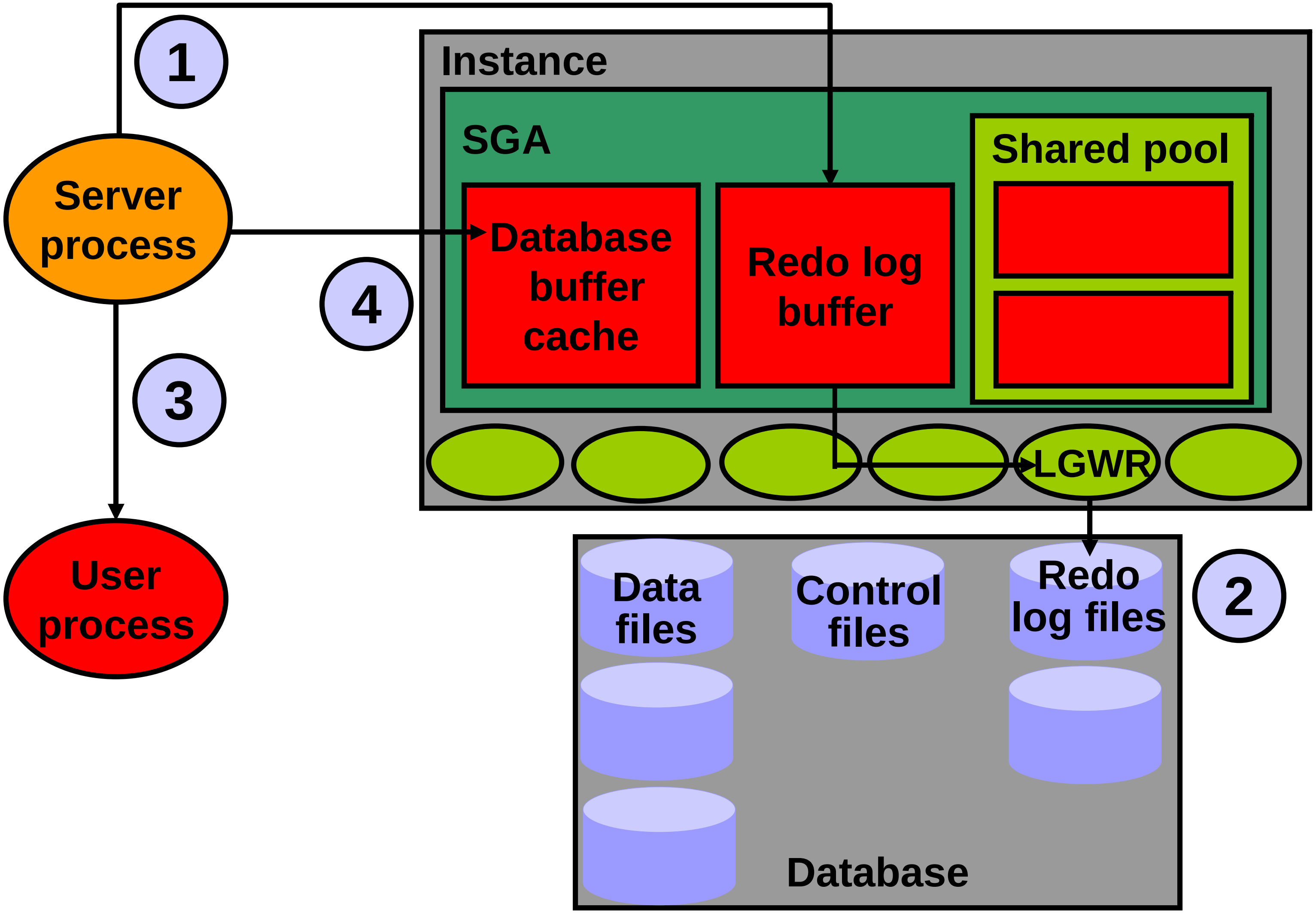
- Has its size defined by `LOG_BUFFER`
- Records changes made through the instance
- Is used sequentially
- Is a circular buffer

Database buffer cache

Rollback Segment



COMMIT Processing



Summary

In this appendix, you should have learned how to:

- Identify database files: data files, control files, and online redo logs
- Describe SGA memory structures: DB buffer cache, shared SQL pool, and redo log buffer
- Explain primary background processes: `DBW0`, `LGWR`, `CKPT`, `PMON`, `SMON`, and `ARC0`
- List SQL processing steps: parse, execute, fetch