SQL

Agenda

- Creating and Managing Tables.
- Including Constraints.
- Creating Views.
- Other Database Objects.
- Writing Basic SQL SELECT Statements.
- Restricting and Sorting Data.
- Single-Row Functions.
- Displaying Data from Multiple Tables.
- Aggregating Data Using Group Functions.
- Subqueries.
- Substitution Variables.
- Manipulating Data.
- Advanced Subqueries.

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Numeric value generator
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

Creating and Managing Tables

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Numeric value generator
Index	Improves the performance of some queries
Synonym	Gives alternative names to objects

Naming Rules

Table names and column names:

- Must begin with a letter
- Must be 1–30 characters long
- Must contain only A–Z, a–z, 0–9, _, $, and #
- Must not duplicate the name of another object owned by the same user
- Must not be an Oracle server reserved word

Creating Tables

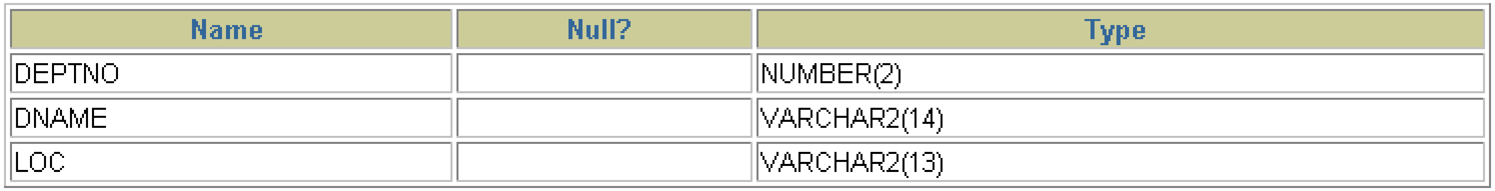
- Create the table.

```
 CREATE TABLE [schema.]table
              (column datatype [DEFAULT expr][, ...]);
```

```
CREATE TABLE dept
            (deptno  NUMBER(2),
             dname   VARCHAR2(14),
             loc     VARCHAR2(13));
Table created.
```

- Confirm table creation.

```
 DESCRIBE dept
```



Name	Null?	Type
DEPTNO		NUMBER(2)
DNAME		VARCHAR2(14)
LOC		VARCHAR2(13)

Tables in the Oracle Database

- User Tables:
  - Are a collection of tables created and maintained by the user
  - Contain user information
- Data Dictionary:
  - Is a collection of tables created and maintained by the Oracle Server
  - Contain database information

Querying the Data Dictionary

- See the names of tables owned by the user.

```
SELECT table_name
FROM   user_tables ;
```

- View distinct object types owned by the user.

```
SELECT DISTINCT object_type
FROM   user_objects ;
```

- View tables, views, synonyms, and sequences owned by the user.

```
SELECT *
FROM   user_catalog ;
```

Data Types

Data Type	Description
`VARCHAR2(`*`size`*`)`	Variable-length character data
`CHAR(`*`size`*`)`	Fixed-length character data
`NUMBER(`*`p,s)`*	Variable-length numeric data
`DATE`	Date and time values
`LONG`	Variable-length character data up to 2 gigabytes
`CLOB`	Character data up to 4 gigabytes
`RAW and LONG RAW`	Raw binary data
`BLOB`	Binary data up to 4 gigabytes
`BFILE`	Binary data stored in an external file; up to 4 gigabytes
`ROWID`	A 64 base number system representing the unique address of a row in its table.

Creating a Table by Using a Subquery Syntax

- Create a table and insert rows by combining the CREATE TABLE statement and the AS *subquery* option.

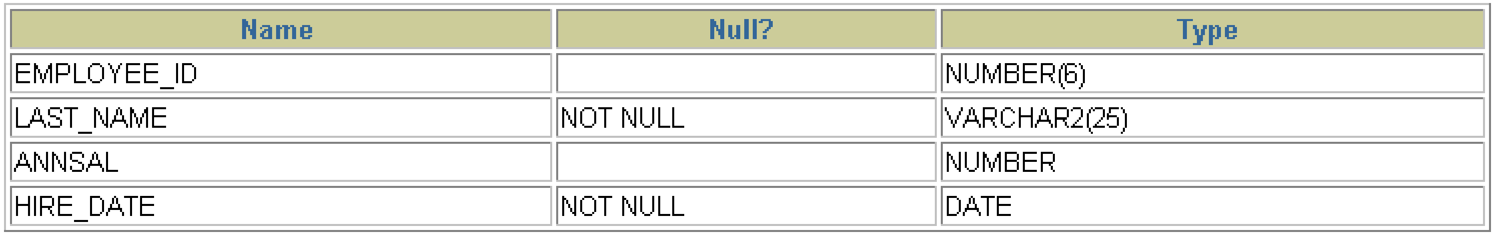
```
CREATE TABLE table
        [(column, column...)]
AS subquery;
```

- Match the number of specified columns to the number of subquery columns.
- Define columns with column names and default values.

Creating a Table by Using a Subquery

```
CREATE TABLE  dept80
  AS
    SELECT   employee_id, last_name,
             salary*12 ANNSAL,
             hire_date
    FROM     employees
    WHERE    department_id = 80;
Table created.
```

```
 DESCRIBE dept80
```



Name	Null?	Type
EMPLOYEE_ID		NUMBER(6)
LAST_NAME	NOT NULL	VARCHAR2(25)
ANNSAL		NUMBER
HIRE_DATE	NOT NULL	DATE

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.
```

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

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

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

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
SET   UNUSED  (column);
OR
ALTER TABLE  table
SET   UNUSED COLUMN column;
```

```
ALTER TABLE table
DROP  UNUSED COLUMNS;
```

Dropping a Table

- All data and structure in the table is deleted.
- Any pending transactions are committed.
- All indexes are dropped.
- You *cannot* roll back the DROP TABLE statement.

```
DROP TABLE dept80;
Table dropped.
```

Changing the Name of an Object

- To change the name of a table, view, sequence, or synonym, you execute the RENAME statement.

```
RENAME dept TO detail_dept;
Table renamed.
```

- You must be the owner of the object.

Truncating a Table

- The TRUNCATE TABLE statement:
  - Removes all rows from a table
  - Releases the storage space used by that table

```
TRUNCATE TABLE detail_dept;
Table truncated.
```

- You cannot roll back row removal when using TRUNCATE.
- Alternatively, you can remove rows by using the DELETE statement.

Adding Comments to a Table

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

```
COMMENT ON TABLE employees
IS 'Employee Information';
Comment created.
```

```
COMMENT ON column employees.name
IS 'Employee name';
Comment created.
```

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

The DEFAULT Option

- Specify a default value for a column during an insert.

```
... hire_date DATE DEFAULT SYSDATE, ...
```

- Literal values, expressions, or SQL functions are legal values.
- Another column’s name or a pseudocolumn are illegal values.
- The default data type must match the column data type.

Including Constraints

What are Constraints?

- Constraints enforce rules at the table level.
- Constraints prevent the deletion of a table if there are dependencies.
- The following constraint types are valid:
  - NOT NULL
  - UNIQUE
  - PRIMARY KEY
  - FOREIGN KEY
  - CHECK

Constraint Guidelines

- Name a constraint or the Oracle server generates a name by using the SYS_C*n* format.
- Create a constraint either:
  - At the same time as the table is created, or
  - After the table has been created
- Define a constraint at the column or table level.
- View a constraint in the data dictionary.

Defining Constraints

```
CREATE TABLE [schema.]table
             (column datatype [DEFAULT expr]
              [column_constraint],
              ...
              [table_constraint][,...]);
```

```
CREATE TABLE employees(
              employee_id  NUMBER(6),
              first_name   VARCHAR2(20),
              ...
              job_id       VARCHAR2(10) NOT NULL,
              CONSTRAINT emp_emp_id_pk
                           PRIMARY KEY (EMPLOYEE_ID));
```

Defining Constraints

- Column level constraint

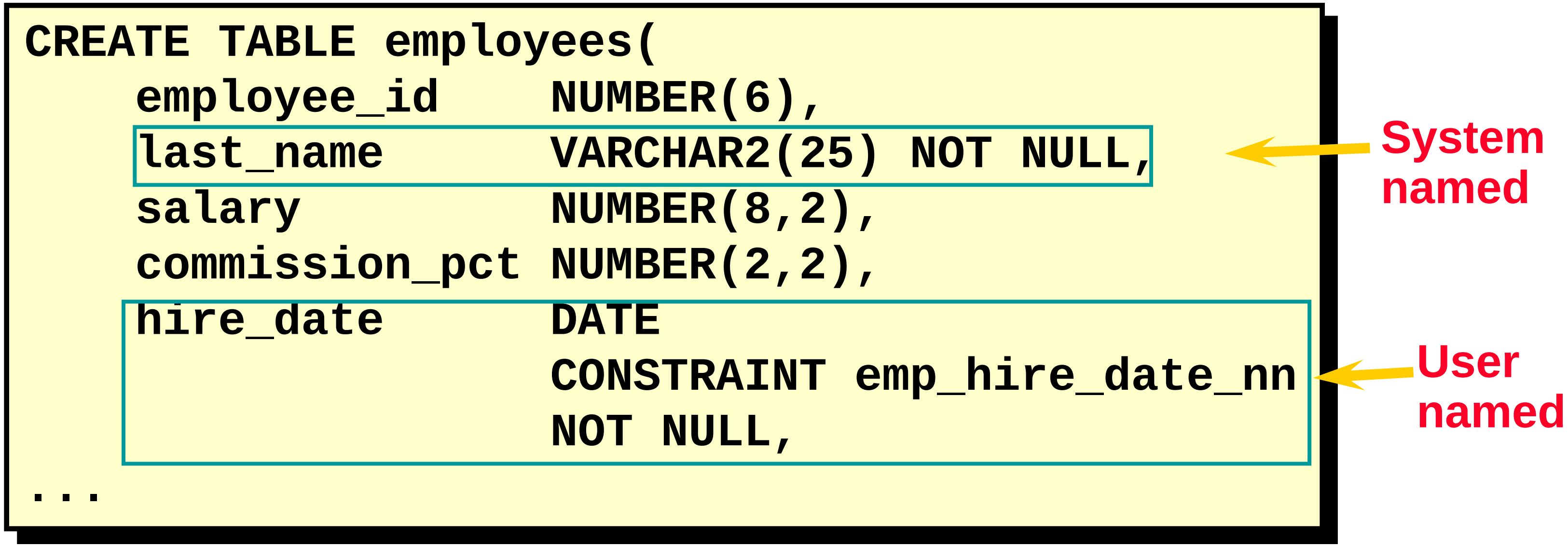
```
column [CONSTRAINT constraint_name] constraint_type,
```

- Table level constraint

```
column,...
  [CONSTRAINT constraint_name] constraint_type
  (column, ...),
```

The NOT NULL Constraint

Is defined at the column level:



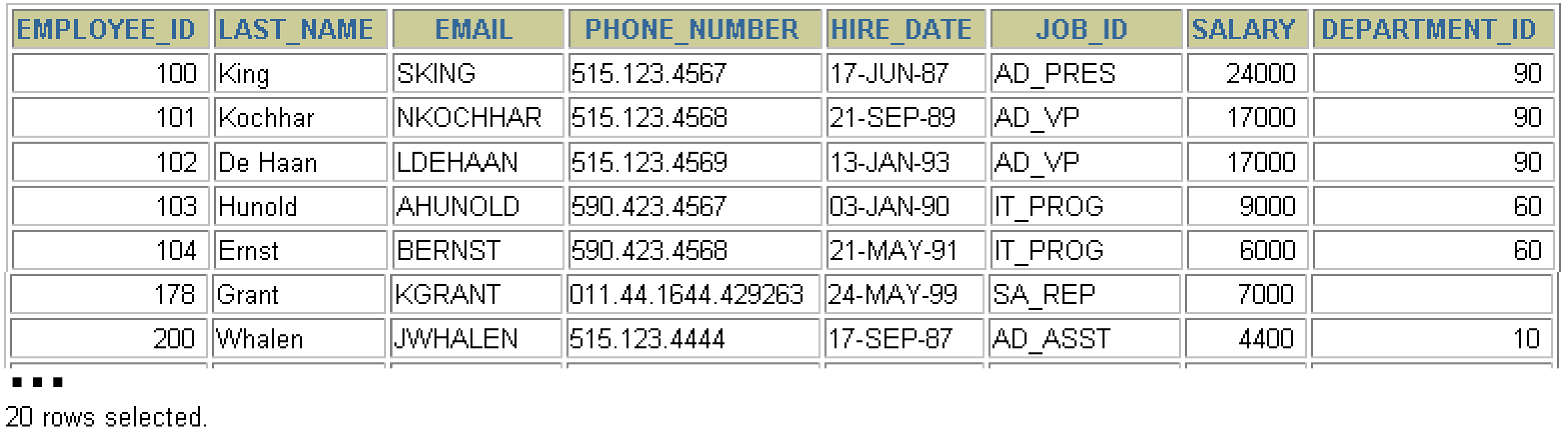
```
CREATE TABLE employees(
    employee_id    NUMBER(6),
    last_name      VARCHAR2(25) NOT NULL,
    salary         NUMBER(8,2),
    commission_pct NUMBER(2,2),
    hire_date      DATE
                   CONSTRAINT emp_hire_date_nn
                   NOT NULL,
...
```

System named

User named

The NOT NULL Constraint

Ensures that null values are not permitted for the column:



EMPLOYEE_ID	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	DEPARTMENT_ID
100	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000	90
101	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000	90
102	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000	90
103	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000	60
104	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000	60
178	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	
200	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400	10

...

20 rows selected.

NOT NULL constraint (No row can contain a null value for this column.)

NOT NULL constraint

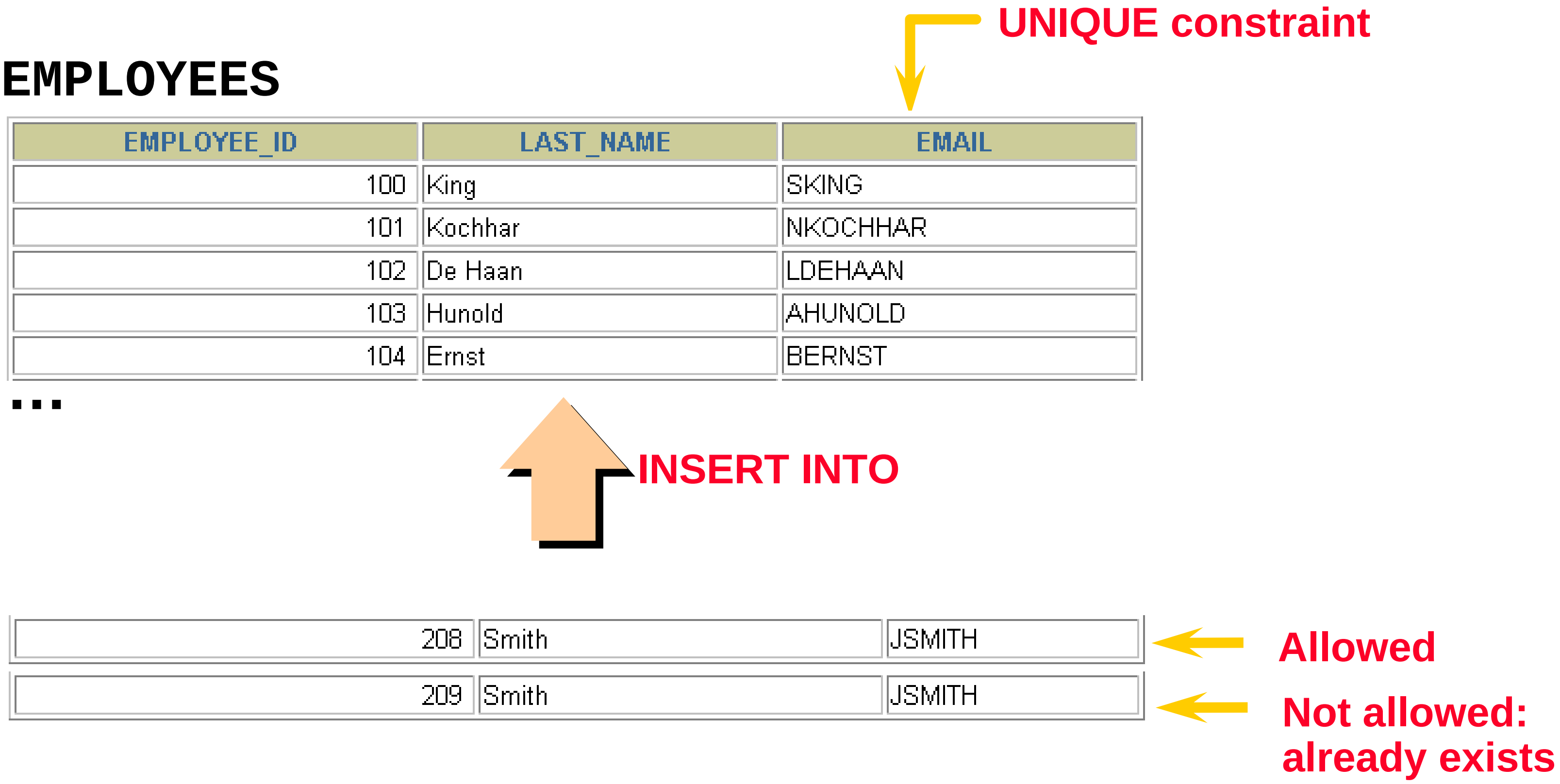
Absence of NOT NULL constraint (Any row can contain null for this column.)

The UNIQUE Constraint

Defined at either the table level or the column level:

```
CREATE TABLE employees(
    employee_id      NUMBER(6) UNIQUE,
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    hire_date        DATE NOT NULL,
...
    CONSTRAINT emp_email_uk UNIQUE(email));
```

The UNIQUE Constraint



UNIQUE constraint

EMPLOYEES

EMPLOYEE_ID	LAST_NAME	EMAIL
100	King	SKING
101	Kochhar	NKOCHHAR
102	De Haan	LDEHAAN
103	Hunold	AHUNOLD
104	Ernst	BERNST

...

INSERT INTO

208	Smith	JSMITH
209	Smith	JSMITH

Allowed

Not allowed: already exists

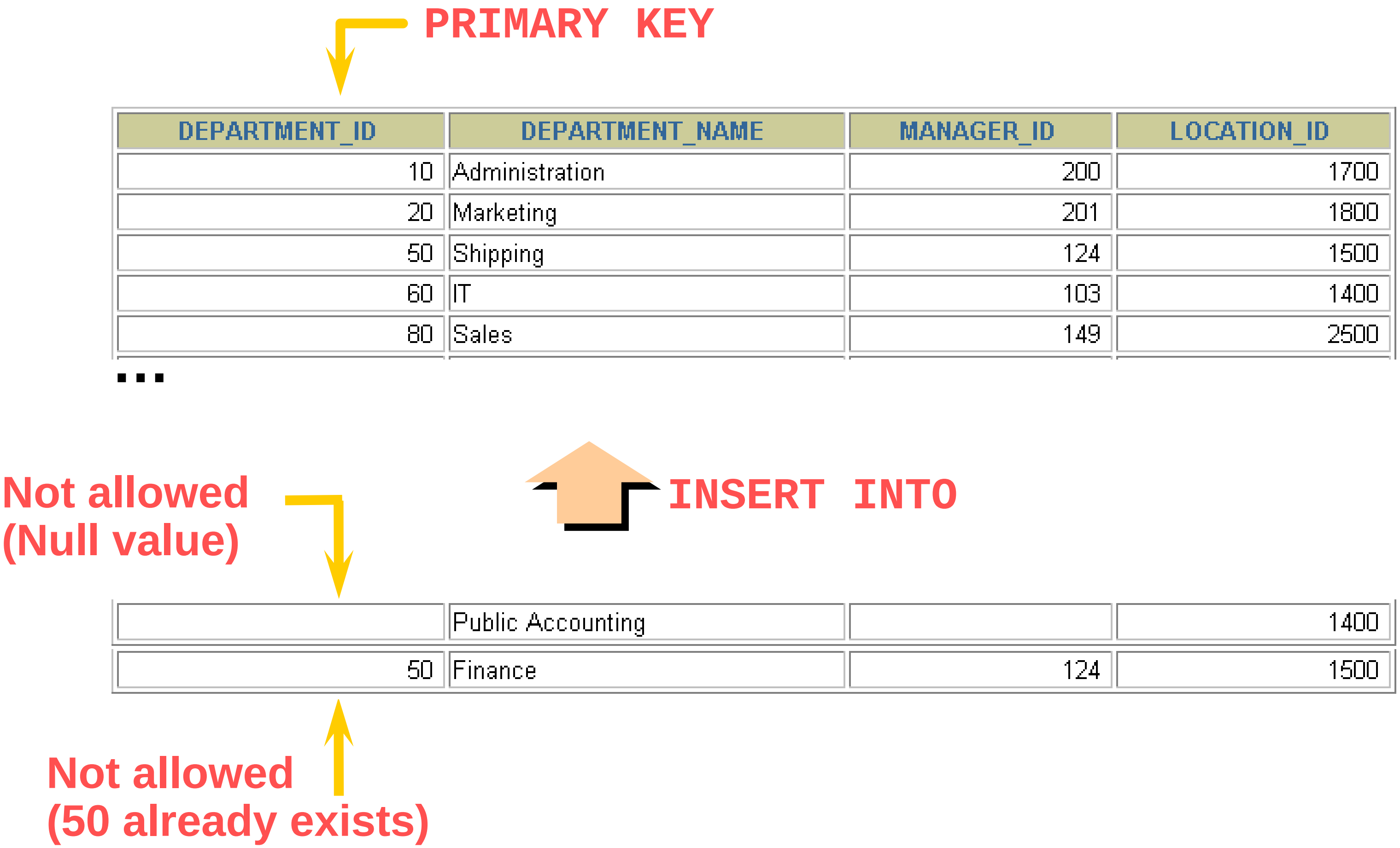
The PRIMARY KEY Constraint

Defined at either the table level or the column level:

```
CREATE TABLE   departments(
    department_id          NUMBER(4),
    department_name        VARCHAR2(30)
      CONSTRAINT dept_name_nn NOT NULL,
    manager_id             NUMBER(6),
    location_id            NUMBER(4),
      CONSTRAINT dept_id_pk PRIMARY KEY(department_id));
```

The PRIMARY KEY Constraint

DEPARTMENTS



PRIMARY KEY

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500

...

INSERT INTO

Not allowed
(Null value)

	Public Accounting		1400
50	Finance	124	1500

Not allowed
(50 already exists)

The FOREIGN KEY Constraint

Defined at either the table level or the column level:

```
CREATE TABLE employees(
    employee_id      NUMBER(6),
    last_name        VARCHAR2(25) NOT NULL,
    email            VARCHAR2(25),
    salary           NUMBER(8,2),
    commission_pct   NUMBER(2,2),
    department_id    NUMBER(6),
    hire_date        DATE NOT NULL,
...
    department_id    NUMBER(4),
    CONSTRAINT emp_dept_fk FOREIGN KEY (department_id)
      REFERENCES departments(department_id),
    CONSTRAINT emp_email_uk UNIQUE(email));
```

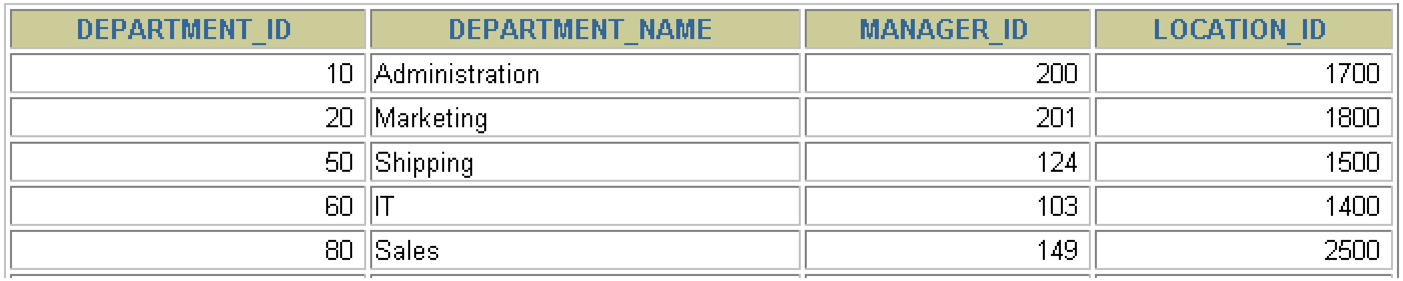
FOREIGN KEY Constraint Keywords

- FOREIGN KEY: Defines the column in the child table at the table constraint level
- REFERENCES: Identifies the table and column in the parent table
- ON DELETE CASCADE: Deletes the dependent rows in the child table when a row in the parent table is deleted.
- ON DELETE SET NULL: Converts dependent foreign key values to null

The FOREIGN KEY Constraint

DEPARTMENTS

PRIMARY KEY

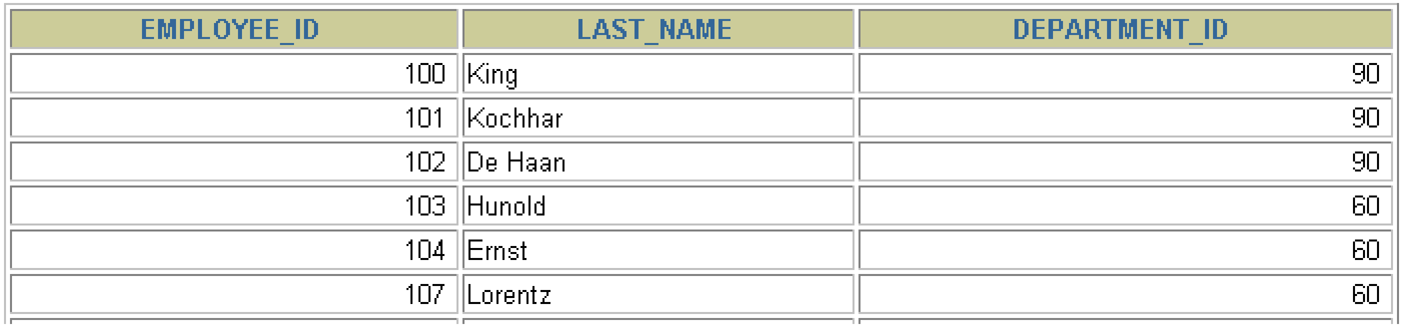


DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500

...

EMPLOYEES

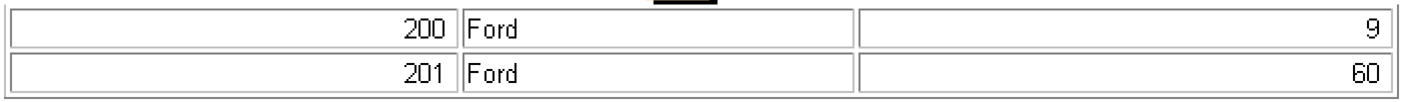
FOREIGN KEY



EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
102	De Haan	90
103	Hunold	60
104	Ernst	60
107	Lorentz	60

...

INSERT INTO



200	Ford	9
201	Ford	60

Not allowed (9 does not exist)

Allowed

The CHECK Constraint

- Defines a condition that each row must satisfy
- The following expressions are not allowed:
  - References to CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns
  - Calls to SYSDATE, UID, USER, and USERENV functions
  - Queries that refer to other values in other rows

```
..., salary  NUMBER(2)
     CONSTRAINT emp_salary_min
            CHECK (salary > 0),...
```

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
ADD [CONSTRAINT constraint] type (column);
```

Adding a Constraint

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

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

Dropping a Constraint

- Remove the manager constraint from the EMPLOYEES table.

```
ALTER TABLE       employees
DROP CONSTRAINT   emp_manager_fk;
Table altered.
```

- Remove the PRIMARY KEY constraint on the DEPARTMENTS table and drop the associated FOREIGN KEY constraint on the EMPLOYEES.DEPARTMENT_ID column.

```
ALTER TABLE   departments
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           employees
DISABLE CONSTRAINT    emp_emp_id_pk CASCADE;
Table altered.
```

Enabling Constraints

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

```
ALTER TABLE            employees
ENABLE CONSTRAINT      emp_emp_id_pk;
Table altered.
```

- A UNIQUE or PRIMARY KEY 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:

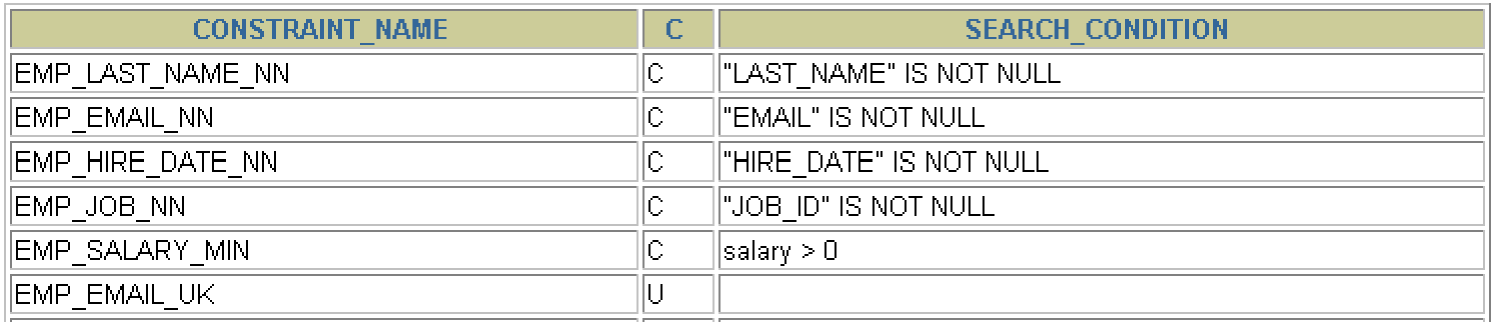
```
ALTER TABLE test1
DROP (pk) CASCADE CONSTRAINTS;
Table altered.
```

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

Viewing Constraints

Query the USER_CONSTRAINTS table to view all constraint definitions and names.

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



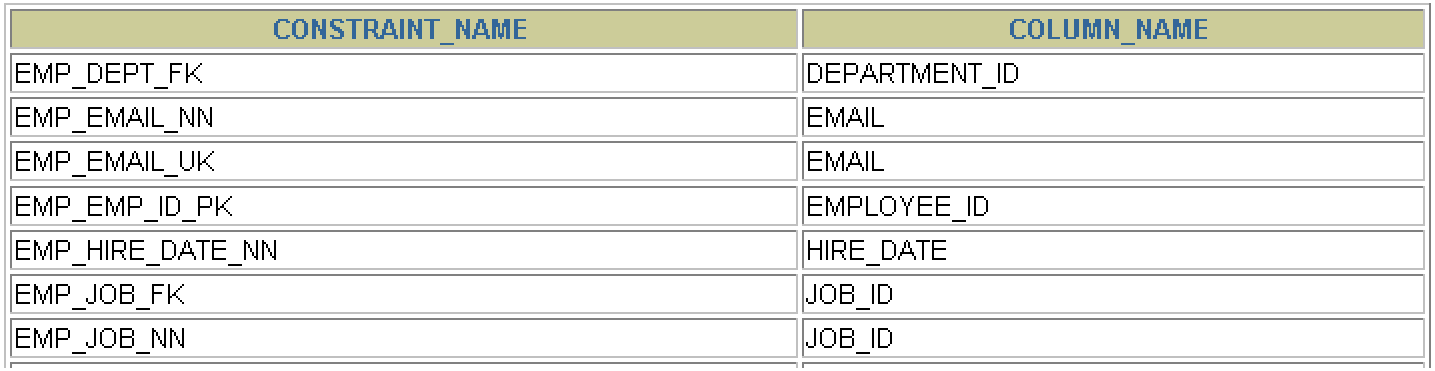
CONSTRAINT_NAME	C	SEARCH_CONDITION
EMP_LAST_NAME_NN	C	"LAST_NAME" IS NOT NULL
EMP_EMAIL_NN	C	"EMAIL" IS NOT NULL
EMP_HIRE_DATE_NN	C	"HIRE_DATE" IS NOT NULL
EMP_JOB_NN	C	"JOB_ID" IS NOT NULL
EMP_SALARY_MIN	C	salary > 0
EMP_EMAIL_UK	U	

…

Viewing the Columns Associated with Constraints

View the columns associated with the constraint names in the USER_CONS_COLUMNS view.

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



CONSTRAINT_NAME	COLUMN_NAME
EMP_DEPT_FK	DEPARTMENT_ID
EMP_EMAIL_NN	EMAIL
EMP_EMAIL_UK	EMAIL
EMP_EMP_ID_PK	EMPLOYEE_ID
EMP_HIRE_DATE_NN	HIRE_DATE
EMP_JOB_FK	JOB_ID
EMP_JOB_NN	JOB_ID

...

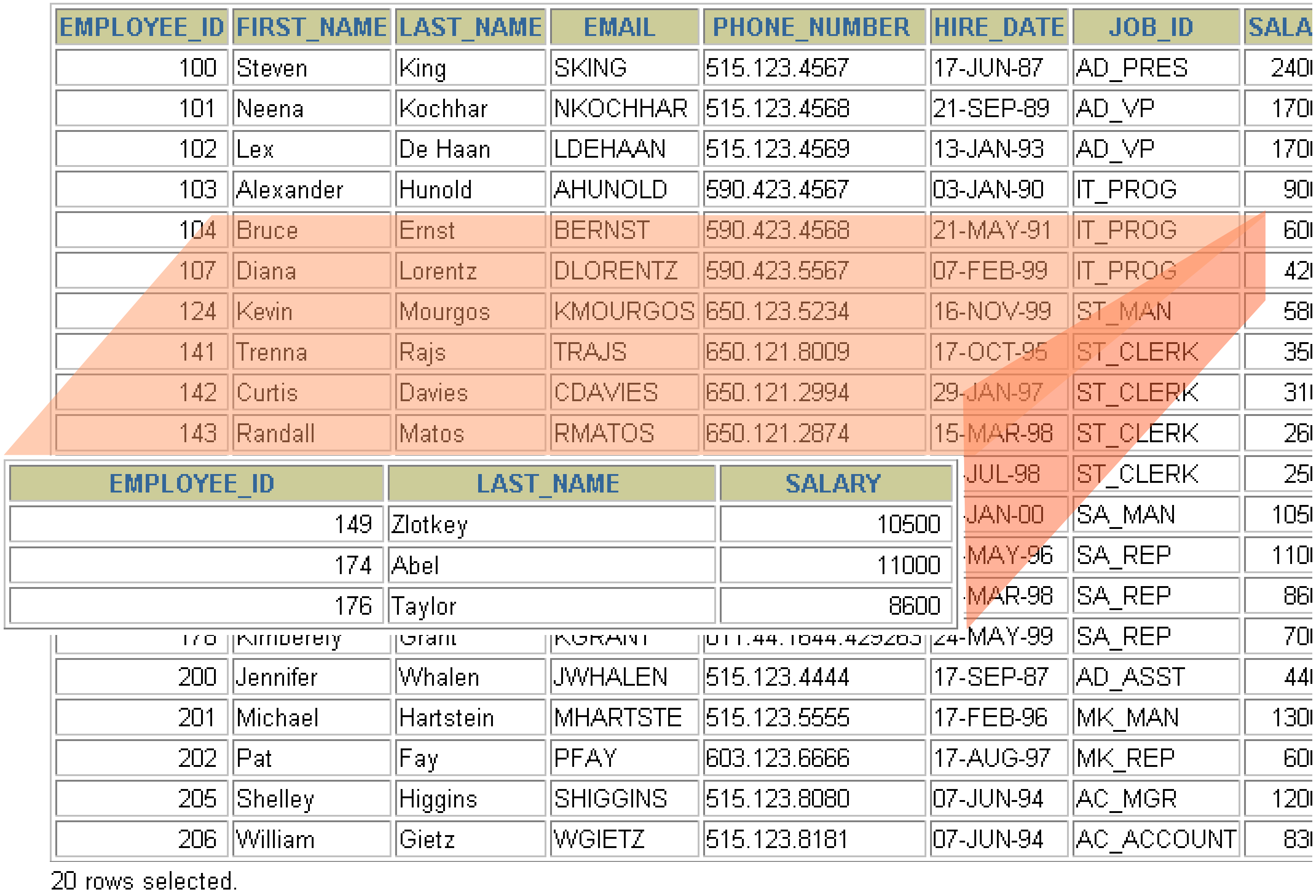
Creating Views

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Generates primary key values
Index	Improves the performance of some queries
Synonym	Alternative name for an object

What is a View?

EMPLOYEES Table:



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALA
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	240
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	170
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	170
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	90
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	42
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	58
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	35
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	31
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	26
					JUL-98	ST_CLERK	25
					JAN-00	SA_MAN	105
					MAY-96	SA_REP	110
					MAR-98	SA_REP	86
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	70
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	44
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	130
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	60
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	120
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	83

EMPLOYEE_ID	LAST_NAME	SALARY
149	Zlotkey	10500
174	Abel	11000
176	Taylor	8600

20 rows selected.

Why Use Views?

A view is a logical entity. It is simply the representation of a SQL statement that has a data dictionary entry that defines this view.

- To restrict data access
- To make complex queries easy
- To provide data independence
- To present different views of the same data.

Creating a View

- You embed a subquery within the `CREATE VIEW` statement.
- Create a view, `EMPVU80`, that contains details of employees in department 80.

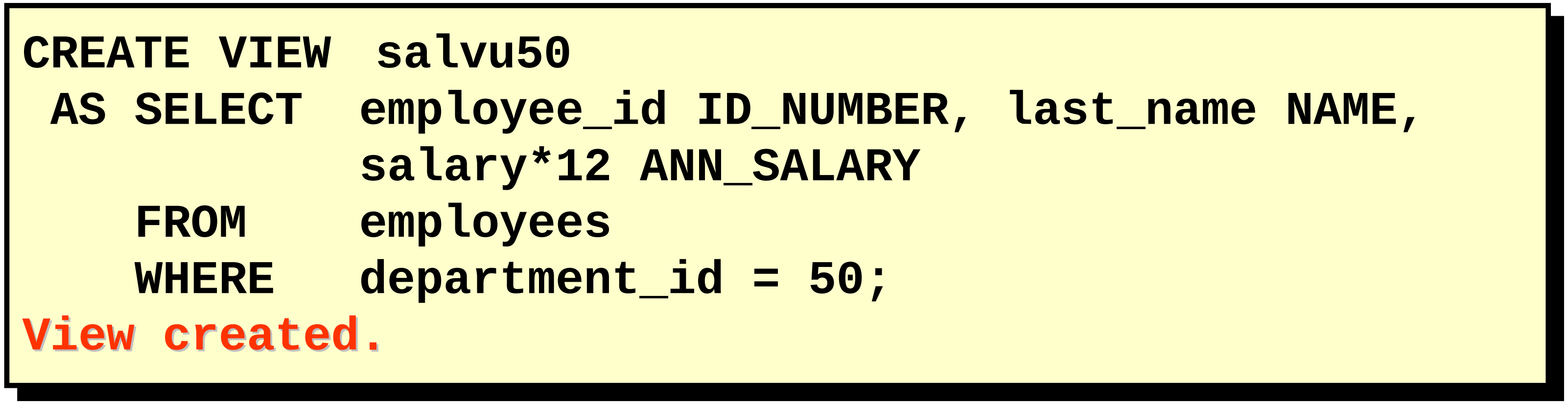
```
CREATE VIEW  empvu80
 AS SELECT   employee_id, last_name, salary
    FROM     employees
    WHERE    department_id = 80;
View created.
```

- Describe the structure of the view by using the `DESCRIBE` command.

```
DESCRIBE empvu80
```

Creating a View

- Create a view by using column aliases in the subquery.

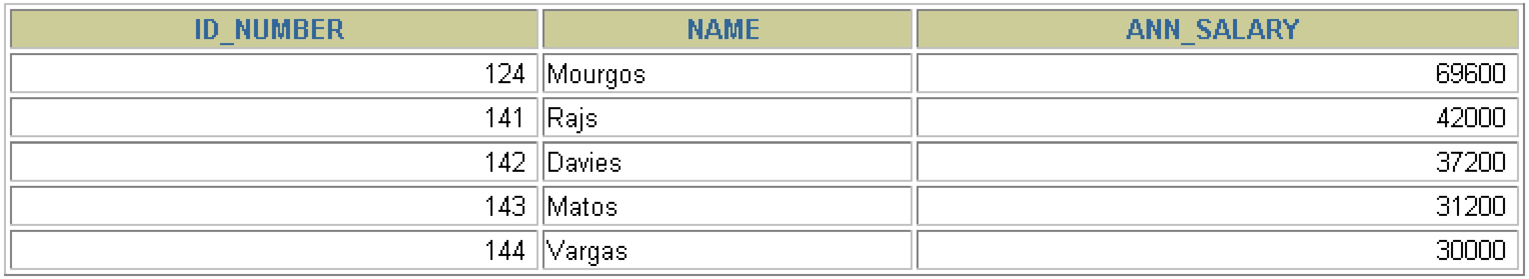


```
CREATE VIEW  salvu50
 AS SELECT  employee_id ID_NUMBER, last_name NAME,
            salary*12 ANN_SALARY
    FROM    employees
    WHERE   department_id = 50;
View created.
```

- Select the columns from this view by the given alias names.

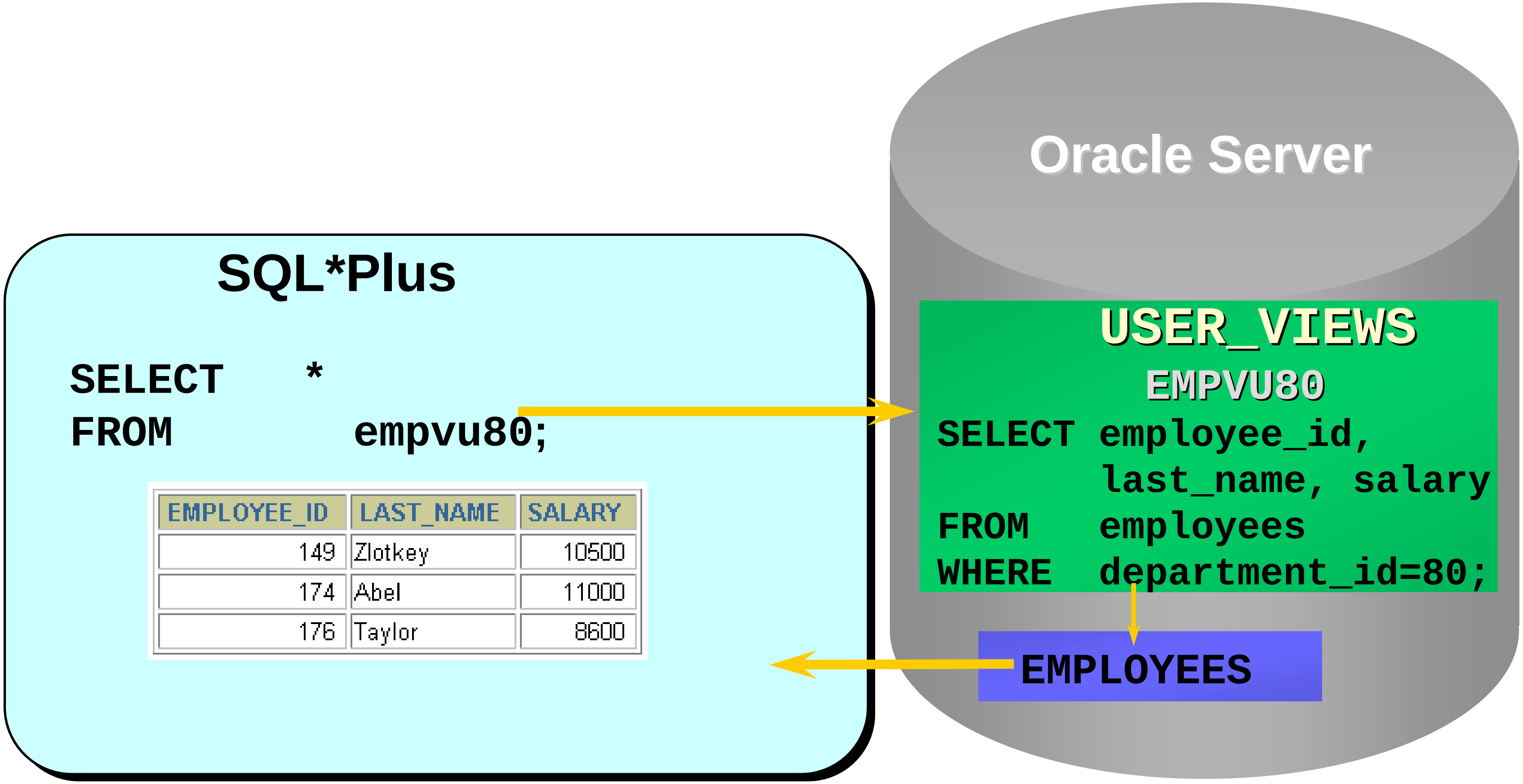
Retrieving Data from a View

```
SELECT *
FROM  salvu50;
```



ID_NUMBER	NAME	ANN_SALARY
124	Mourgos	69600
141	Rajs	42000
142	Davies	37200
143	Matos	31200
144	Vargas	30000

Querying a View



SQL*Plus

```
SELECT    *
FROM          empvu80;
```

EMPLOYEE_ID	LAST_NAME	SALARY
149	Zlotkey	10500
174	Abel	11000
176	Taylor	8600

Oracle Server

USER_VIEWS

EMPVU80

```
SELECT employee_id,
       last_name, salary
FROM   employees
WHERE  department_id=80;
```

EMPLOYEES

Modifying a View

- Modify the EMPVU80 view by using CREATE OR REPLACE VIEW clause. Add an alias for each column name.

```
CREATE OR REPLACE VIEW empvu80
  (id_number, name, sal, department_id)
AS SELECT  employee_id, first_name || ' ' || last_name,
           salary, department_id
   FROM    employees
   WHERE   department_id = 80;
View created.
```

- Column aliases in the CREATE VIEW clause are listed in the same order as the columns in the subquery.

Creating a Complex View

Create a complex view that contains group functions to display values from two tables.

```
CREATE VIEW dept_sum_vu
  (name, minsal, maxsal, avgsal)
AS SELECT    d.department_name, MIN(e.salary),
             MAX(e.salary),AVG(e.salary)
   FROM      employees e, departments d
   WHERE     e.department_id = d.department_id
   GROUP BY  d.department_name;
View created.
```

Rules for Performing DML Operations on a View

- You can perform DML operations on simple views.
- You cannot remove a row if the view contains the following:
  - Group functions
  - A `GROUP BY` clause
  - The `DISTINCT` keyword
  - The pseudocolumn `ROWNUM` keyword

Rules for Performing DML Operations on a View

You cannot modify data in a view if it contains:

- Group functions
- A `GROUP  BY` clause
- The `DISTINCT` keyword
- The pseudocolumn `ROWNUM` keyword
- Columns defined by expressions

Rules for Performing DML Operations on a View

You cannot add data through a view if the view includes:

- Group functions
- A `GROUP BY` clause
- The `DISTINCT` keyword
- The pseudocolumn `ROWNUM` keyword
- Columns defined by expressions
- `NOT NULL` columns in the base tables that are not selected by the view

Using the WITH CHECK OPTION Clause

- You can ensure that DML operations performed on the view stay within the domain of the view by using the WITH CHECK OPTION clause. WITH CHECK OPTION is designed for updatable views where as CHECK constraint specifies valid values for an individual column.

```
CREATE OR REPLACE VIEW empvu20
AS SELECT   *
   FROM     employees
   WHERE    department_id = 20
   WITH CHECK OPTION CONSTRAINT empvu20_ck ;
View created.
```

- Any attempt to change the department number for any row in the view fails because it violates the WITH CHECK OPTION constraint.

Denying DML Operations

- You can ensure that no DML operations occur by adding the `WITH READ ONLY` option to your view definition.
- Any attempt to perform a DML on any row in the view results in an Oracle server error.

```
CREATE OR REPLACE VIEW empvu10
    (employee_number, employee_name, job_title)
AS SELECT   employee_id, last_name, job_id
   FROM     employees
   WHERE    department_id = 10
   WITH READ ONLY;
View created.
```

Removing a View

You can remove a view without losing data because a view is based on underlying tables in the database.

```
DROP VIEW view;
```

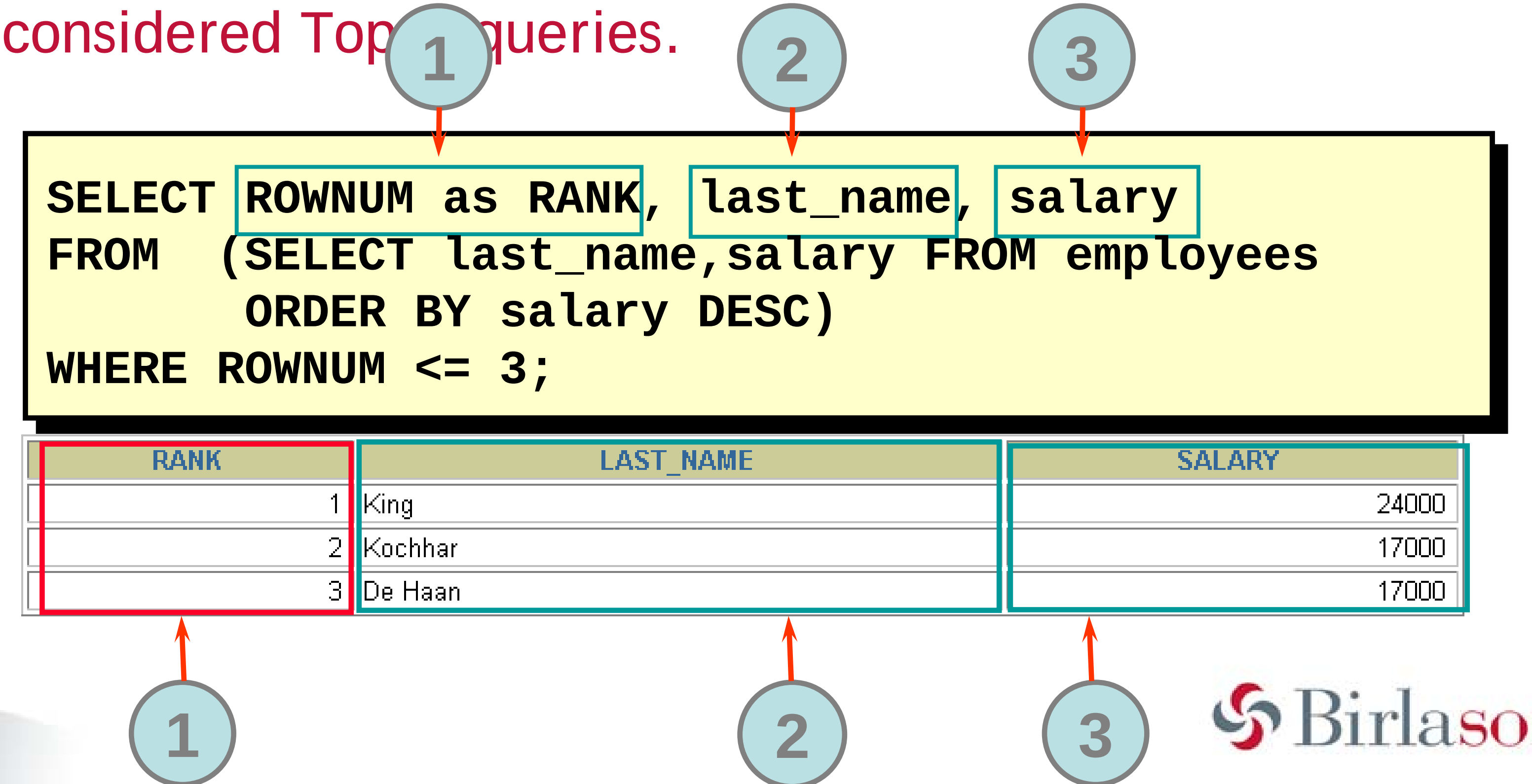
```
DROP VIEW empvu80;
View dropped.
```

Inline Views

- An inline view is a subquery with an alias (or correlation name) that you can use within a SQL statement.
- A named subquery in the FROM clause of the main query is an example of an inline view.
- An inline view is not a schema object.

Top-N Analysis

- Top-N queries ask for the n largest or smallest values of a column. For example:
  - What are the ten best selling products?
  - What are the ten worst selling products?
- Both largest values and smallest values sets are considered Top-N queries.



```
SELECT ROWNUM as RANK, last_name, salary
FROM   (SELECT last_name,salary FROM employees
        ORDER BY salary DESC)
WHERE ROWNUM <= 3;
```

RANK	LAST_NAME	SALARY
1	King	24000
2	Kochhar	17000
3	De Haan	17000

Other Database Objects

Database Objects

Object	Description
Table	Basic unit of storage; composed of rows and columns
View	Logically represents subsets of data from one or more tables
Sequence	Generates primary key values
Index	Improves the performance of some queries
Synonym	Alternative name for an object

What Is a Sequence?

A sequence:

- Automatically generates unique numbers
- Is a sharable object
- Is typically used to create a primary key value
- Replaces application code
- Speeds up the efficiency of accessing sequence values when cached in memory

Creating a Sequence

- Create a sequence named DEPT_DEPTID_SEQ to be used for the primary key of the DEPARTMENTS table.
- Do not use the CYCLE option.

```
CREATE SEQUENCE sequence
       [INCREMENT BY n]
       [START WITH n]
       [{MAXVALUE n | NOMAXVALUE}]
       [{MINVALUE n | NOMINVALUE}]
       [{CYCLE | NOCYCLE}]
       [{CACHE n | NOCACHE}];
```

```
CREATE SEQUENCE dept_deptid_seq
                INCREMENT BY 10
                START WITH 120
                MAXVALUE 9999
                NOCACHE
                NOCYCLE;
Sequence created.
```

Options in a Sequence

- INCREMENT BY - Tells the system how to increment the sequence. If it is positive, the values are ascending; if it is negative, the values are descending.
- START WITH - Tells the system which integer to start with.
- MINVALUE - Tells the system how low the sequence can go. For ascending sequences, it defaults to 1; for descending sequences, the default value is 10e27-1.
- MAXVALUE - Tells the system the highest value that will be allowed. For descending sequences, the default is 1; for ascending sequences, the default is 10e27-1.
- CYCLE - Causes the sequences to automatically recycle to minvalue when maxvalue is reached for ascending sequences; for descending sequences, it causes a recycle from minvalue back to maxvalue.
- CACHE - Caches the specified number of sequence values into the buffers in the SGA. This speeds access, but all cached numbers are lost when the database is shut down. The default value is 20; maximum value is maxvalue-minvalue.

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;
```

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

NEXTVAL and CURRVAL Pseudocolumns

- NEXTVAL returns the next available sequence value. It returns a unique value every time it is referenced, even for different users.
- CURRVAL obtains the current sequence value.
- NEXTVAL must be issued for that sequence before CURRVAL contains a value.

Using a Sequence

- Insert a new department named “Support” in location ID 2500.

```
INSERT INTO departments(department_id,
            department_name, location_id)
VALUES      (dept_deptid_seq.NEXTVAL,
            'Support', 2500);
1 row created.
```

- View the current value for the DEPT_DEPTID_SEQ sequence.

```
SELECT   dept_deptid_seq.CURRVAL
FROM     dual;
```

Using a Sequence

- Caching sequence values in memory gives faster access to those values.
- Gaps in sequence values can occur when:
  - A rollback occurs
  - The system crashes
  - A sequence is used in another table
- If the sequence was created with `NOCACHE`, view the next available value, by querying the `USER_SEQUENCES` table.

Modifying a Sequence

Change the increment value, maximum value, minimum value, cycle option, or cache option.

```
ALTER SEQUENCE dept_deptid_seq
               INCREMENT BY 20
               MAXVALUE 999999
               NOCACHE
               NOCYCLE;
Sequence altered.
```

Guidelines for Modifying a Sequence

- You must be the owner or have the ALTER privilege for the sequence.
- Only future sequence numbers are affected.
- The sequence must be dropped and re-created to restart the sequence at a different number.
- Some validation is performed.

Removing a Sequence

- Remove a sequence from the data dictionary by using the DROP SEQUENCE statement.
- Once removed, the sequence can no longer be referenced.

```
DROP SEQUENCE dept_deptid_seq;
Sequence dropped.
```

What is an Index?

An index:

- Is a schema object
- Is used by the Oracle server to speed up the retrieval of rows by using a pointer
- Can reduce disk I/O by using a rapid path access method to locate data quickly
- Is independent of the table it indexes
- Is used and maintained automatically by the Oracle server

How Are Indexes Created?

- Automatically: A unique index is created automatically when you define a `PRIMARY KEY` or `UNIQUE` constraint in a table definition.
- Manually: Users can create nonunique indexes on columns to speed up access to the rows.

Creating an Index

- Create an index on one or more columns.

```
CREATE INDEX index
ON table (column[, column]...);
```

- Improve the speed of query access to the LAST_NAME column in the EMPLOYEES table.

```
CREATE INDEX  emp_last_name_idx
ON employees(last_name);
Index created.
```

When to Create an Index

You should create an index if:

- A column contains a wide range of values
- A column contains a large number of null values
- One or more columns are frequently used together in a WHERE clause or a join condition
- The table is large and most queries are expected to retrieve less than 2 to 4 percent of the rows

When Not to Create an Index

It is usually not worth creating an index if:

- The table is small
- The columns are not often used as a condition in the query
- Most queries are expected to retrieve more than 2 to 4 percent of the rows in the table
- The indexed columns are referenced as part of an expression

Confirming Indexes

- The USER_INDEXES data dictionary view contains the name of the index and its uniqueness.
- The USER_IND_COLUMNS view contains the index name, the table name, and the column name.

```
SELECT   ic.index_name, ic.column_name,
         ic.column_position col_pos,ix.uniqueness
FROM     user_indexes ix, user_ind_columns ic
WHERE    ic.index_name = ix.index_name
AND      ic.table_name = 'EMPLOYEES';
```

Removing an Index

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

```
DROP INDEX index;
```

- Remove the UPPER_LAST_NAME_IDX index from the data dictionary.

```
DROP INDEX upper_last_name_idx;
Index dropped.
```

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

Synonyms

Simplify access to objects by creating a synonym (another name for an object). With synonyms, you can:

- Ease referring to a table owned by another user
- Shorten lengthy object names

```
CREATE [PUBLIC] SYNONYM synonym
FOR    object;
```

Creating and Removing Synonyms

- Create a shortened name for the DEPT_SUM_VU view.

```
CREATE SYNONYM  d_sum
FOR  dept_sum_vu;
Synonym Created.
```

- Drop a synonym.

```
DROP SYNONYM d_sum;
Synonym dropped.
```

Writing Basic SQL SELECT Statements

Writing SQL Statements

- SQL statements are not case sensitive.
- SQL statements can be on one or more lines.
- Keywords cannot be abbreviated or split across lines.
- Clauses are usually placed on separate lines.
- Indents are used to enhance readability.

Basic SELECT Statement

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

- SELECT identifies *what* columns
- FROM identifies *which* table

Basic SELECT Statement

Selecting All Columns

```
SELECT *
FROM   departments;
```

Selecting Specific Columns

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

Using Arithmetic Operators

```
SELECT last_name, salary, salary + 300
FROM   employees;
```

Arithmetic Expressions

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

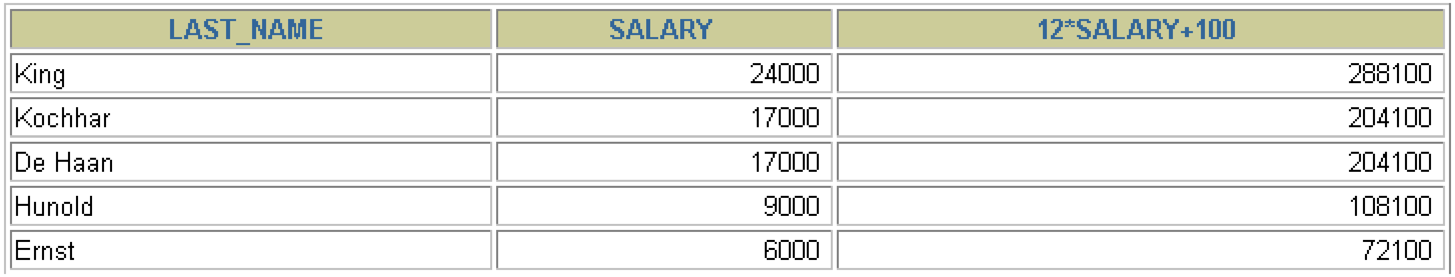
Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

Operator Precedence

* / + –

Operator Precedence

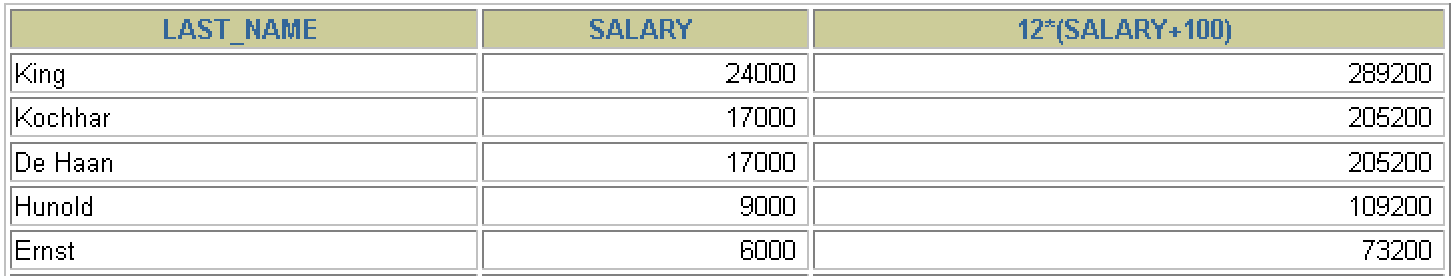
```
SELECT last_name, salary, 12*salary+100
FROM   employees;
```



LAST_NAME	SALARY	12*SALARY+100
King	24000	288100
Kochhar	17000	204100
De Haan	17000	204100
Hunold	9000	108100
Ernst	6000	72100

Using Parentheses

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

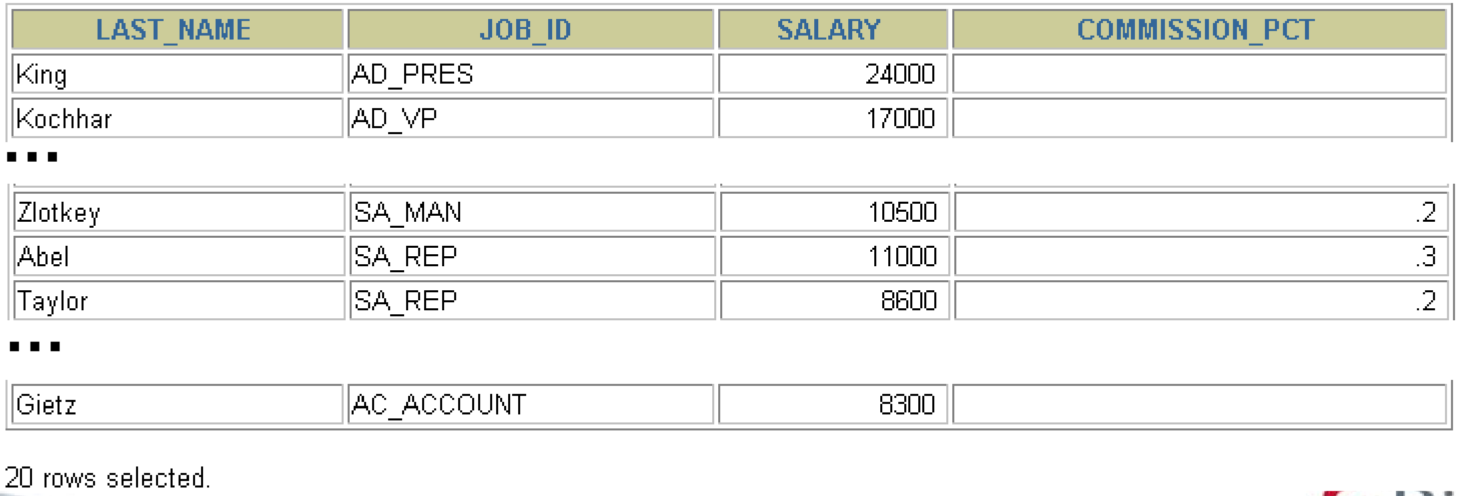


LAST_NAME	SALARY	12*(SALARY+100)
King	24000	289200
Kochhar	17000	205200
De Haan	17000	205200
Hunold	9000	109200
Ernst	6000	73200

Defining a Null Value

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

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



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

```
20 rows selected.
```

Defining a Column Alias

A column alias:

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

Using Column Aliases

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

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

Concatenation Operator

A concatenation operator:

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

Using the Concatenation Operator

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

```
SELECT last_name  ||' is a '||job_id
       AS "Employee Details"
FROM   employees;
```

Restricting and Sorting Data

Limiting the Rows Selected

- Restrict the rows returned by using the WHERE clause.

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

- The WHERE clause follows the FROM clause.

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

Character Strings and Dates

- Character strings and date values are enclosed in single quotation marks.
- Character values are case sensitive, and date values are format sensitive.
- The default date format is DD-MON-RR.

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

Comparison Conditions

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
`BETWEEN ..AND..`	Between two values (inclusive),
`IN(set)`	Match any in list of values
`LIKE`	Match a character pattern
`IS NULL`	Is a null value

Using Comparison Conditions

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

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

Lower limit   Upper limit

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

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

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

Logical Conditions

Operator	Meaning
AND	Returns TRUE if *both* component conditions are true
OR	Returns TRUE if *either* component condition is true
NOT	Returns TRUE if the following condition is false

Using the Logical Operator

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

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

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

Rules of Precedence

Order Evaluated	Operator
1	Arithmetic operators
2	Concatenation operator
3	Comparison conditions
4	IS [NOT] NULL, LIKE, [NOT] IN
5	[NOT] BETWEEN
6	NOT logical condition
7	AND logical condition
8	OR logical condition

Override rules of precedence by using parentheses.

Rules of Precedence

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

Use parentheses to force priority.

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

ORDER BY Clause

- Sort 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 ;
```

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

Sorting by Column Alias

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

Sorting by Multiple Columns

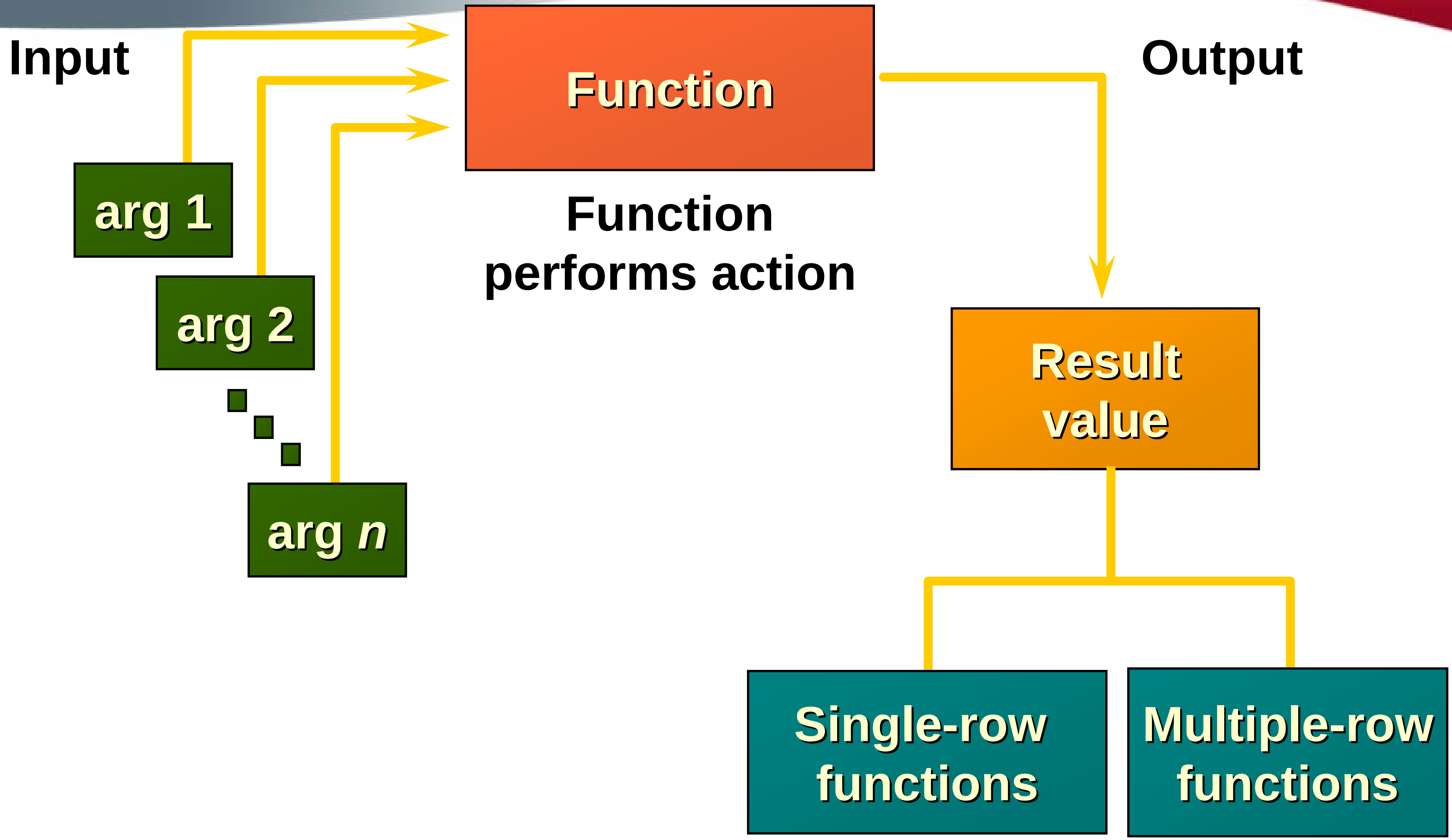
- The order of ORDER BY list is the order of sort.

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

- You can sort by a column that is not in the SELECT list.

Single-Row Functions

SQL Functions



Input

arg 1

arg 2

arg *n*

Function

Function performs action

Output

Result value

Single-row functions

Multiple-row functions

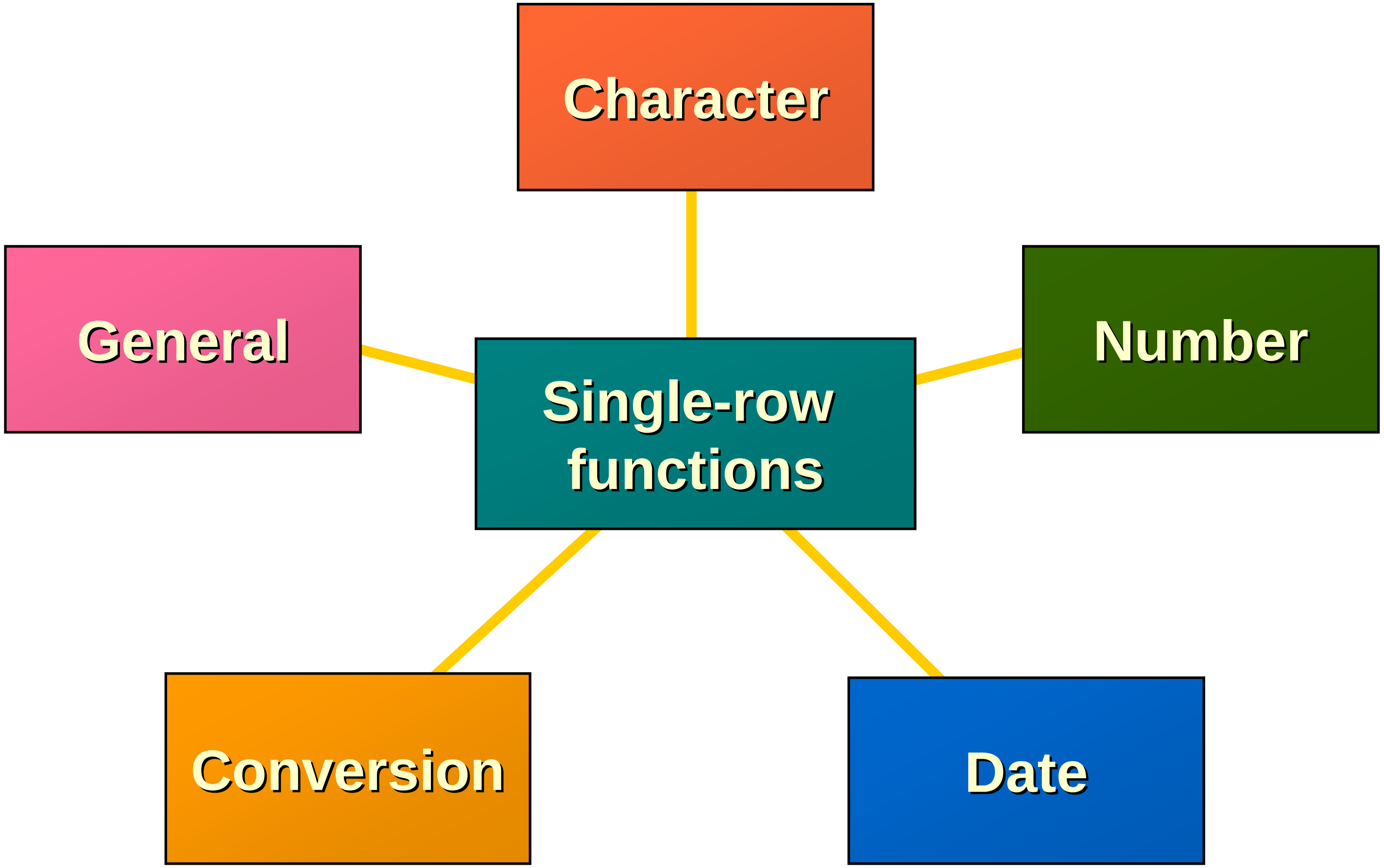
Single-Row Functions

Single row functions:

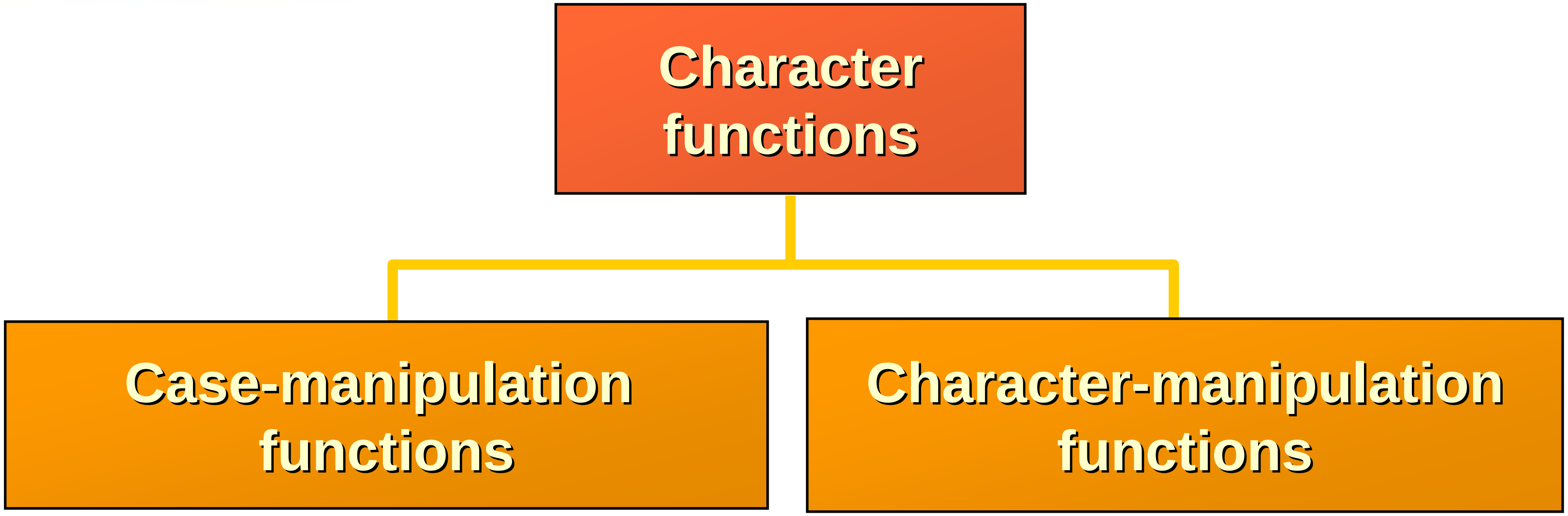
- Manipulate data items
- Accept arguments and return one value
- Act on each row returned
- Return one result per row
- May modify the data type
- Can be nested
- Accept arguments which can be a column or an expression

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

Single-Row Functions



Character Functions



Character functions

Case-manipulation functions

Character-manipulation functions

```
LOWER
UPPER
INITCAP
```

```
CONCAT
SUBSTR
LENGTH
INSTR
LPAD | RPAD
TRIM
REPLACE
```

Case Manipulation Functions

These functions convert case for character strings.

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

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

Character-Manipulation Functions

These functions manipulate character strings:

Function	Result
CONCAT('Hello', 'World')	HelloWorld
SUBSTR('HelloWorld',1,5)	Hello
LENGTH('HelloWorld')	10
INSTR('HelloWorld', 'W')	6
LPAD(salary,10,'*')	*****24000
RPAD(salary, 10, '*')	24000*****
TRIM('H' FROM 'HelloWorld')	elloWorld

Number Functions

- ROUND: Rounds value to specified decimal

  ROUND(45.926, 2) → 45.93

- TRUNC:Truncates value to specified decimal

  TRUNC(45.926, 2) → 45.92

- MOD: Returns remainder of division

  MOD(1600, 300) → 100

Working with Date Functions

Date Functions are used for….

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

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

Date Functions

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

Using Date Functions

- MONTHS_BETWEEN ('01-SEP-95','11-JAN-94')

  → 19.6774194

- ADD_MONTHS ('11-JAN-94',6) → '11-JUL-94'

- NEXT_DAY ('01-SEP-95','FRIDAY')

  → '08-SEP-95'

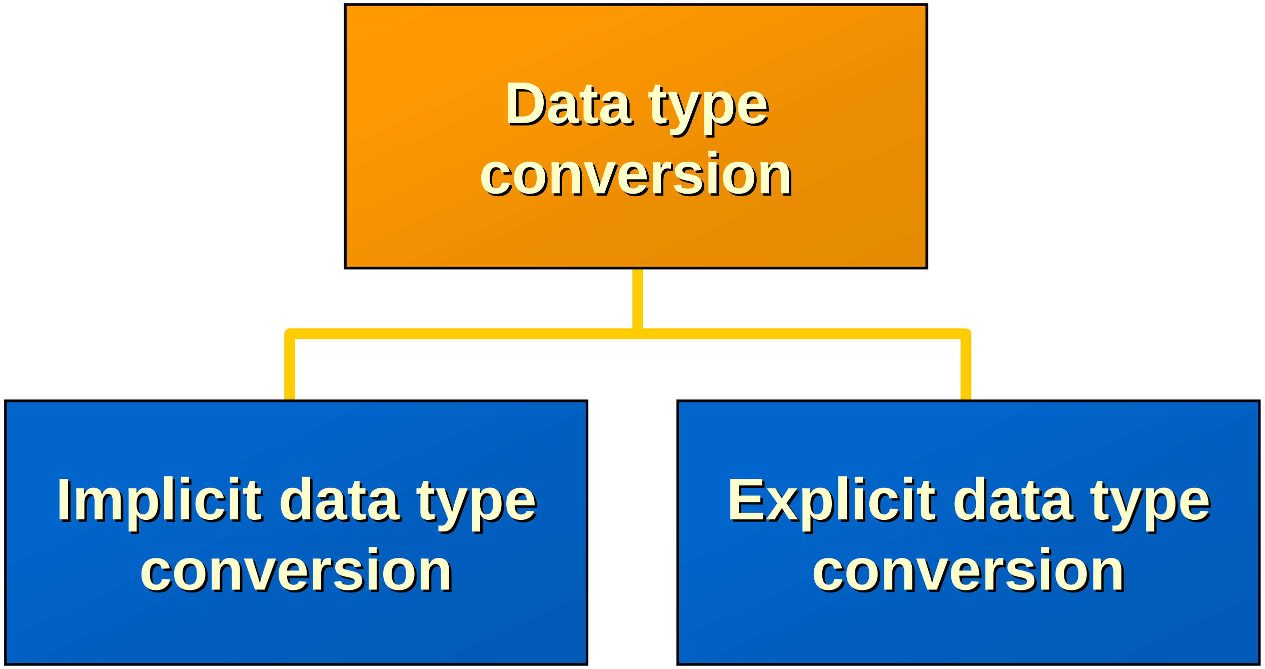
- LAST_DAY('01-FEB-95') → '28-FEB-95'

Using Date Functions

Assume SYSDATE = '25-JUL-95':

- ROUND(SYSDATE,'MONTH') → 01-AUG-95
- ROUND(SYSDATE ,'YEAR') → 01-JAN-96
- TRUNC(SYSDATE ,'MONTH') → 01-JUL-95
- TRUNC(SYSDATE ,'YEAR') → 01-JAN-95

Conversion Functions


Data type conversion

Implicit data type conversion

Explicit data type conversion

Implicit Data Type Conversion

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

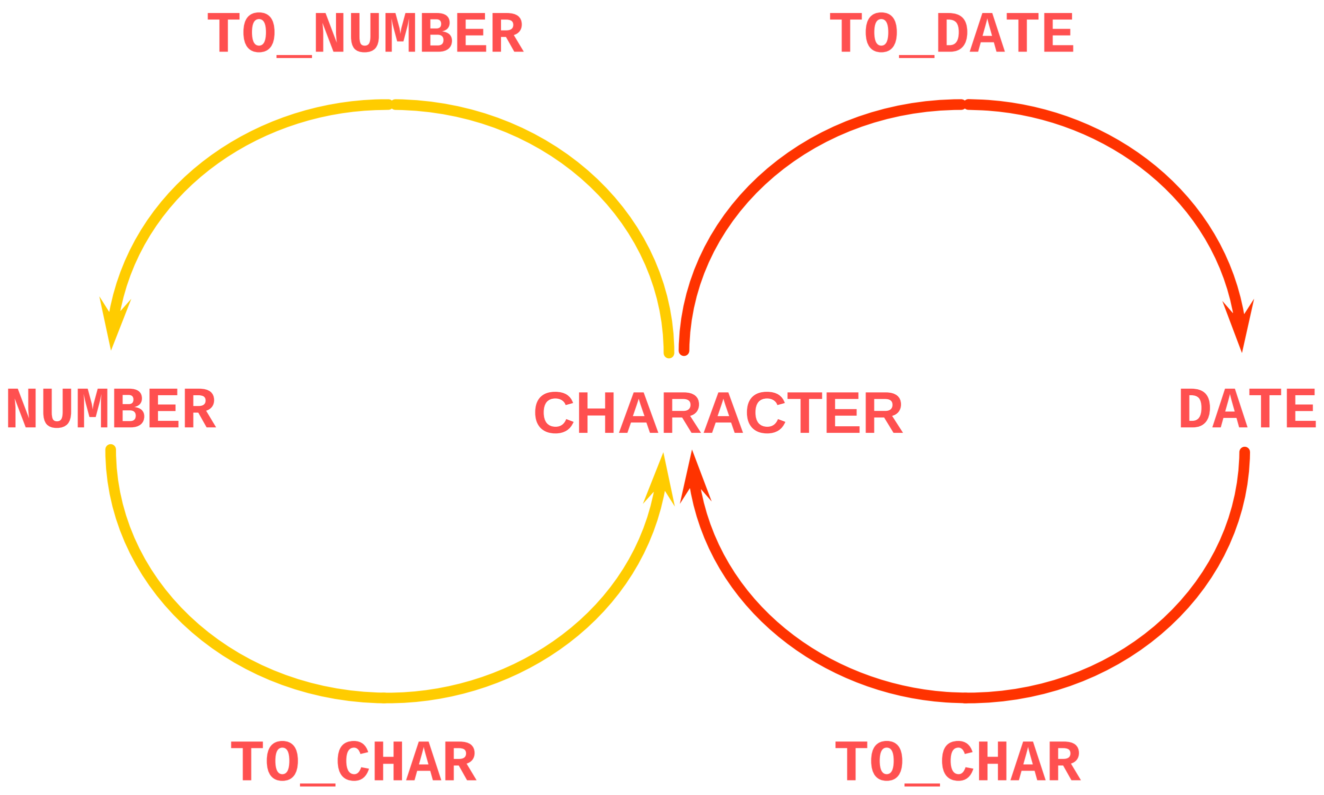
From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE
NUMBER	VARCHAR2
DATE	VARCHAR2

Implicit Data Type Conversion

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

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

Explicit Data Type Conversion



Using the TO_CHAR Function with Dates

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

The format model:

- Must be enclosed in single quotation marks and is case sensitive
- Can include any valid date format element
- Is separated from the date value by a comma

Elements of the Date Format Model

YYYY	Full year in numbers
YEAR	Year spelled out
MM	Two-digit value for month
MONTH	Full name of the month
MON	Three-letter abbreviation of the month
DY	Three-letter abbreviation of the day of the week
DAY	Full name of the day of the week
DD	Numeric day of the month

Elements of the Date Format Model

- Time elements format the time portion of the date.

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

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

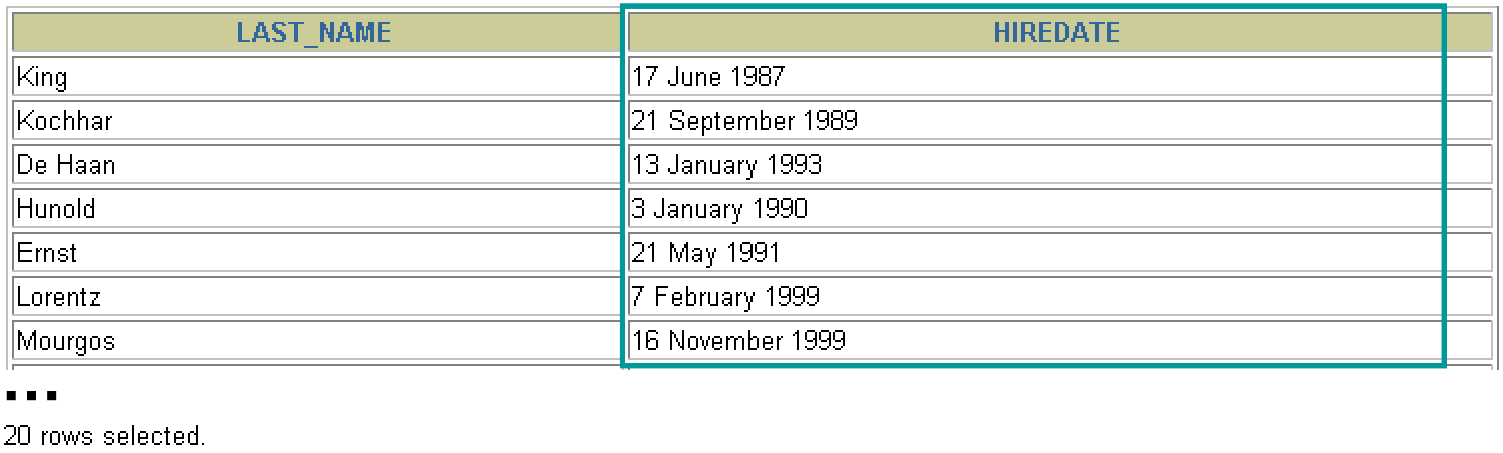
DD "of" MONTH	12 of OCTOBER

- Number suffixes spell out numbers.

ddspth	fourteenth

Using the TO_CHAR Function with Dates

```
SELECT last_name,
       TO_CHAR(hire_date, 'fmDD Month YYYY')
       AS HIREDATE
FROM   employees;
```



LAST_NAME	HIREDATE
King	17 June 1987
Kochhar	21 September 1989
De Haan	13 January 1993
Hunold	3 January 1990
Ernst	21 May 1991
Lorentz	7 February 1999
Mourgos	16 November 1999

...

20 rows selected.

Using the TO_CHAR Function with Numbers

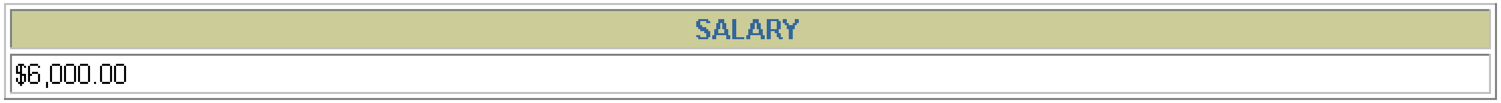
```
TO_CHAR(number, 'format_model')
```

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

9	Represents a number
0	Forces a zero to be displayed
$	Places a floating dollar sign
L	Uses the floating local currency symbol
.	Prints a decimal point
,	Prints a thousand indicator

Using the TO_CHAR Function with Numbers

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



```
SALARY
$6,000.00
```

Using the TO_NUMBER and TO_DATE Functions

- Convert a character string to a number format using the TO_NUMBER function:

```
TO_NUMBER(char[, 'format_model'])
```

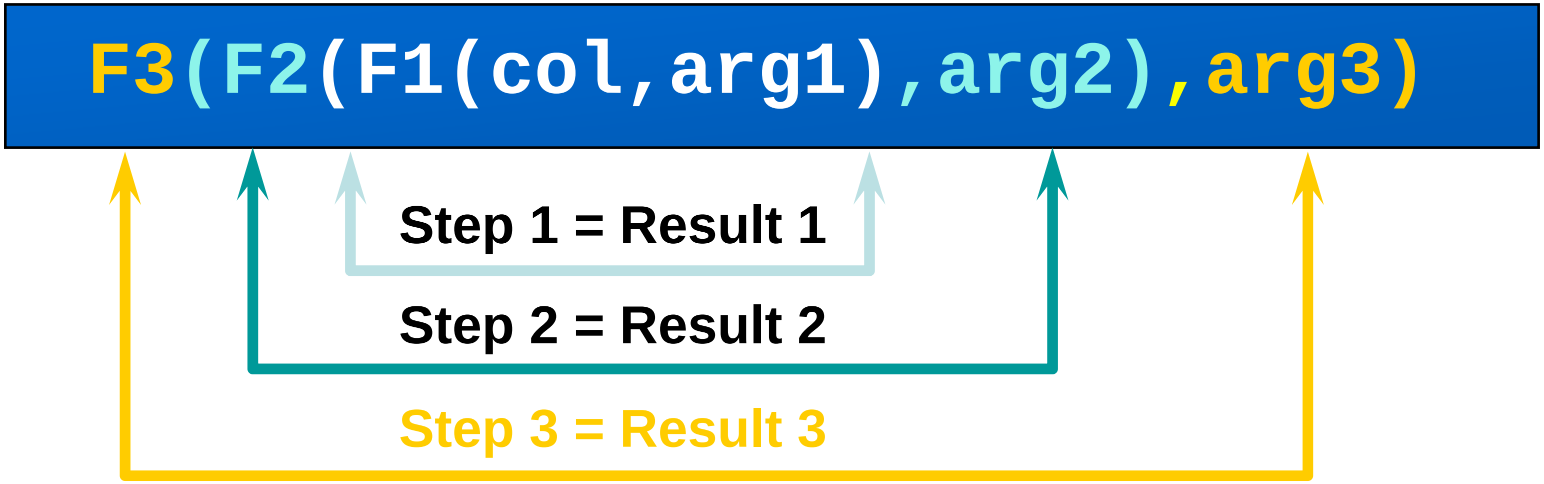
- Convert a character string to a date format using the TO_DATE function:

```
TO_DATE(char[, 'format_model'])
```

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

Nesting Functions

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



```
F3(F2(F1(col,arg1),arg2),arg3)
```

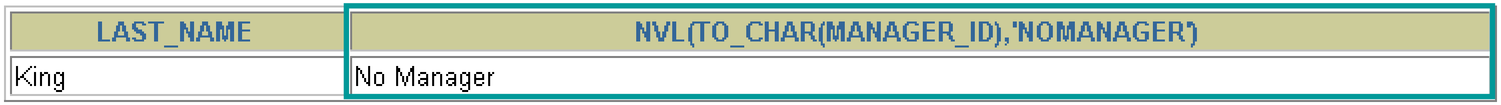
Step 1 = Result 1

Step 2 = Result 2

Step 3 = Result 3

Nesting Functions

```
SELECT last_name,
       NVL(TO_CHAR(manager_id), 'No Manager')
FROM   employees
WHERE  manager_id IS NULL;
```



LAST_NAME	NVL(TO_CHAR(MANAGER_ID),'NOMANAGER')
King	No Manager

General Functions

These functions work with any data type and pertain to using nulls.

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

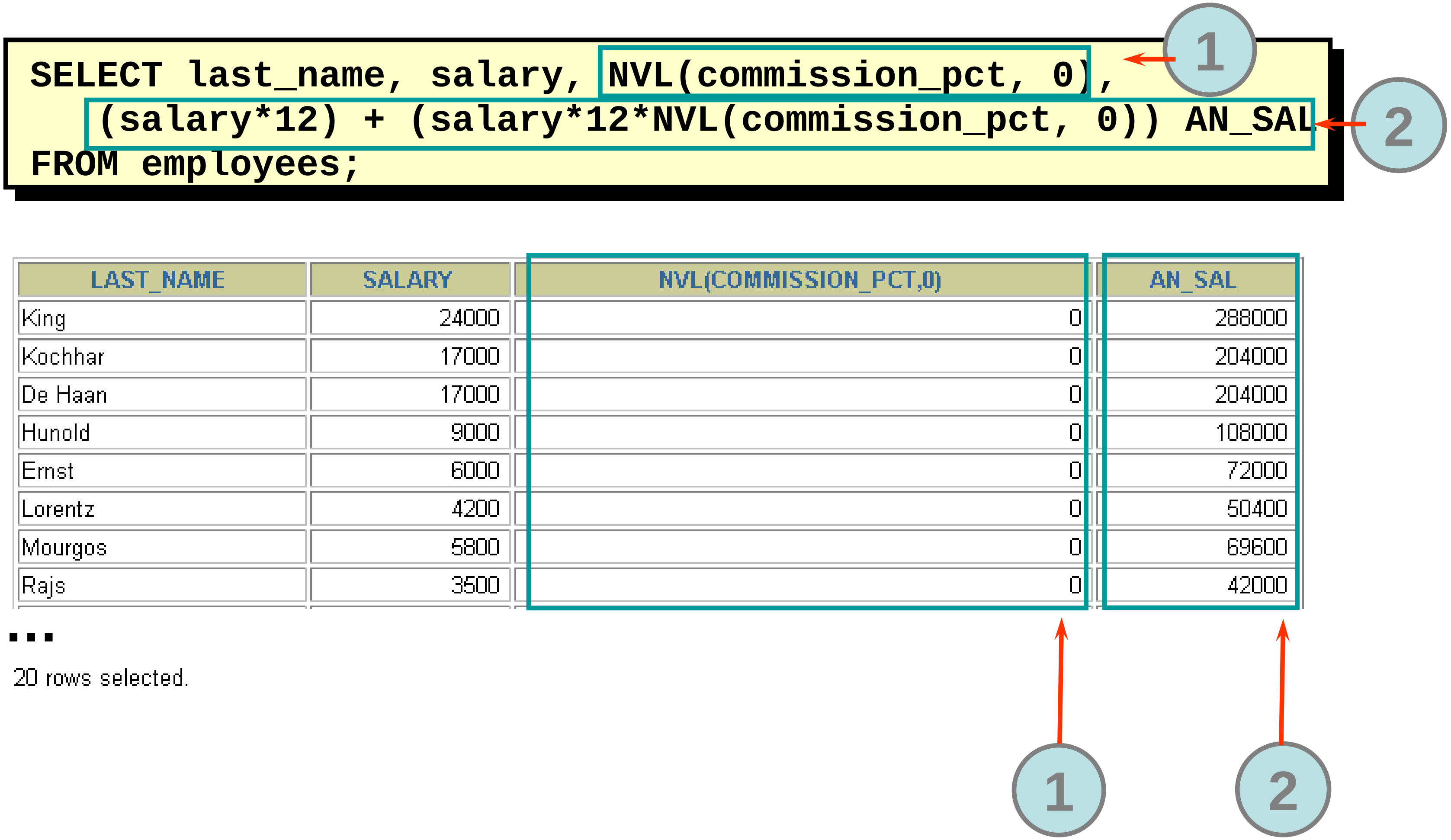
NVL Function

Converts a null to an actual value.

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

Using the NVL Function

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

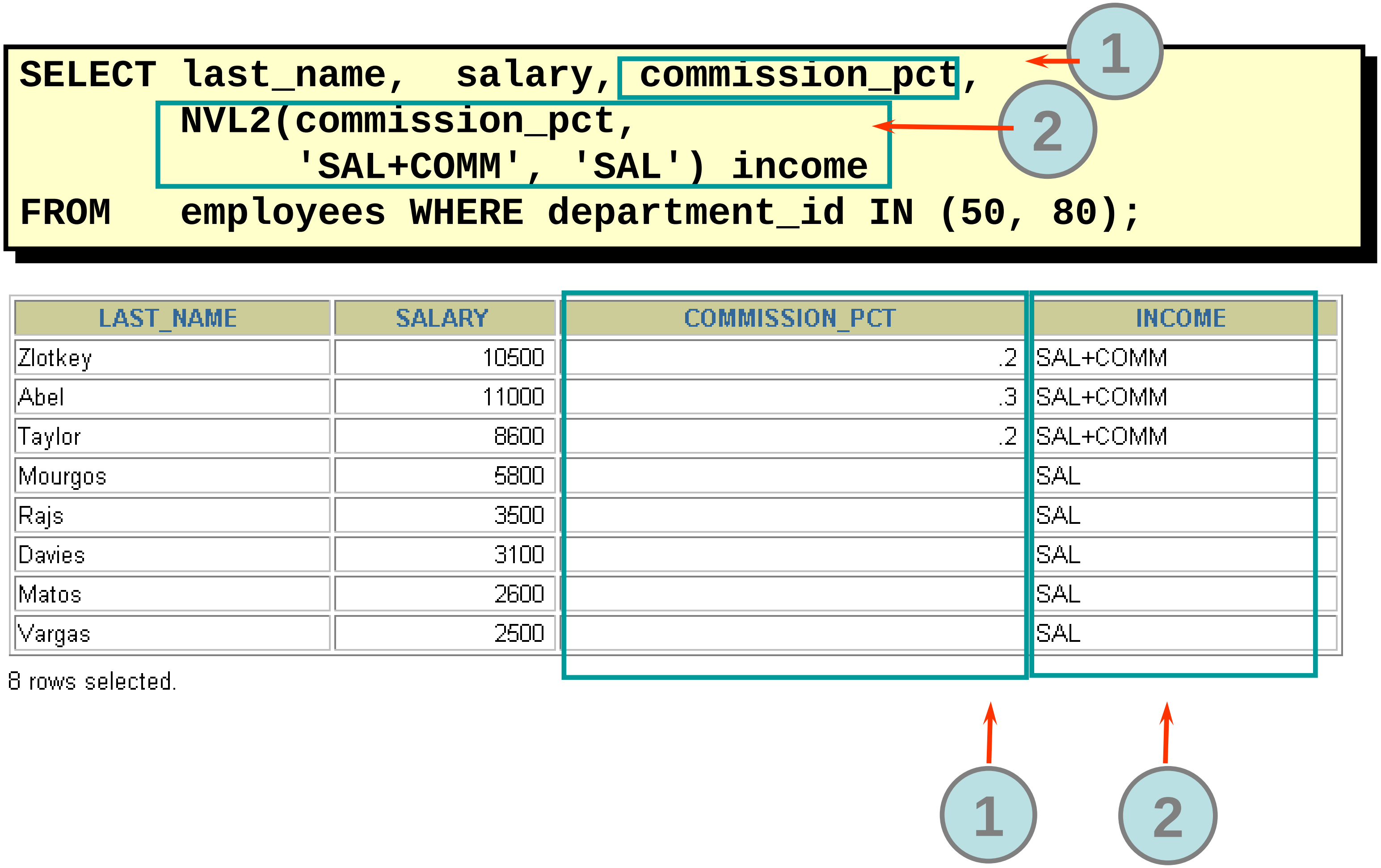


LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
King	24000	0	288000
Kochhar	17000	0	204000
De Haan	17000	0	204000
Hunold	9000	0	108000
Ernst	6000	0	72000
Lorentz	4200	0	50400
Mourgos	5800	0	69600
Rajs	3500	0	42000

...

20 rows selected.

Using the NVL2 Function



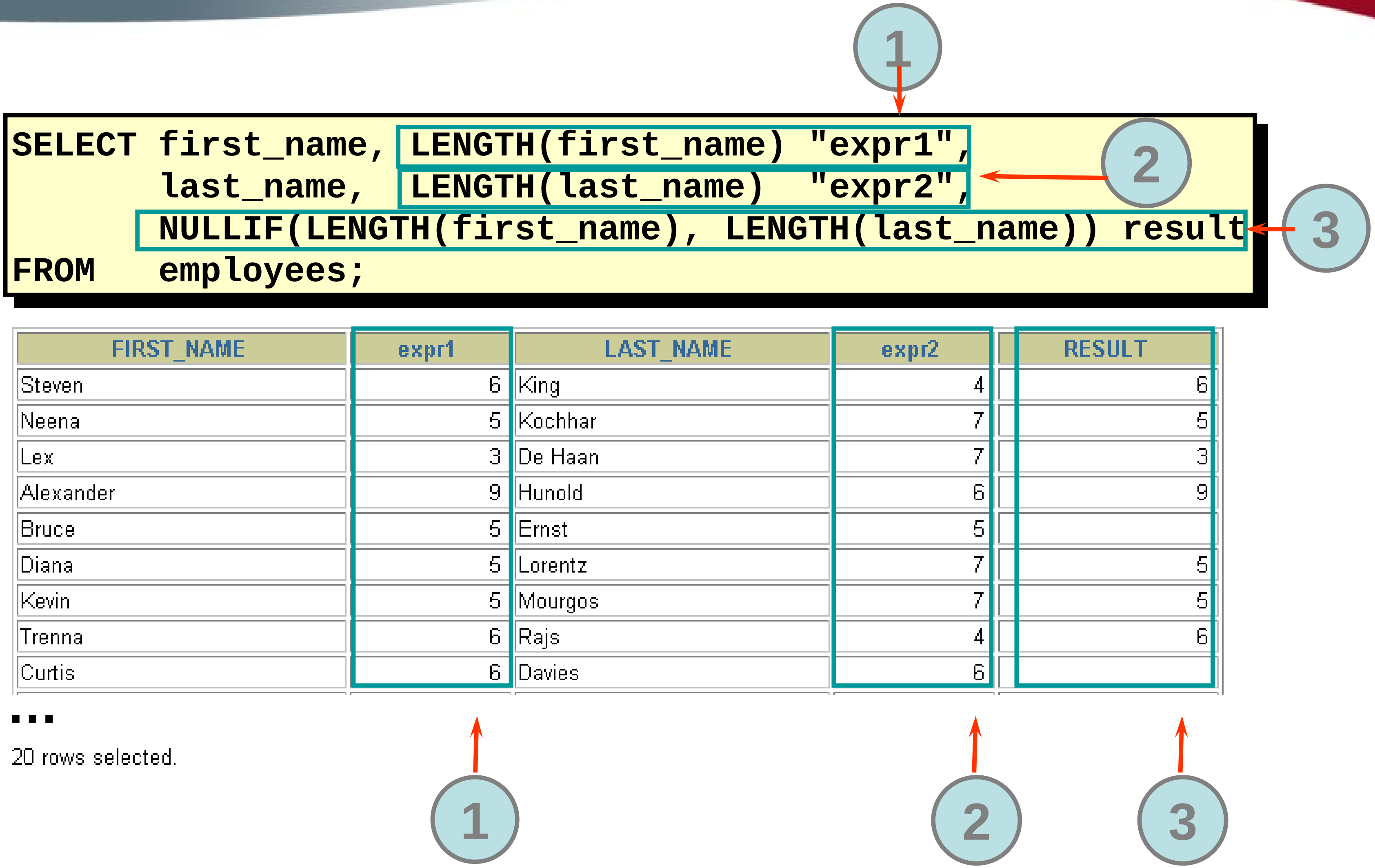
```
SELECT last_name,  salary, commission_pct,
       NVL2(commission_pct,
            'SAL+COMM', 'SAL') income
FROM   employees WHERE department_id IN (50, 80);
```

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

8 rows selected.

Birlasoft®

Using the NULLIF Function



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

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

…

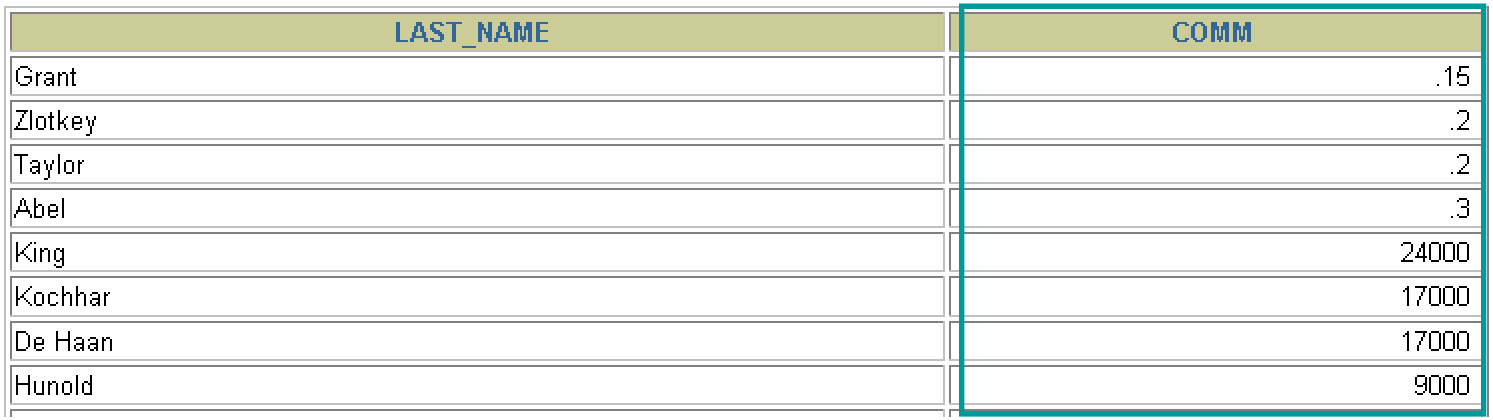
20 rows selected.

Using the COALESCE Function

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

Using the COALESCE Function

```
SELECT   last_name,
         COALESCE(commission_pct, salary, 10) comm
FROM     employees
ORDER BY commission_pct;
```



LAST_NAME	COMM
Grant	.15
Zlotkey	.2
Taylor	.2
Abel	.3
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000

...

20 rows selected.

Conditional Expressions

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

The CASE Expression

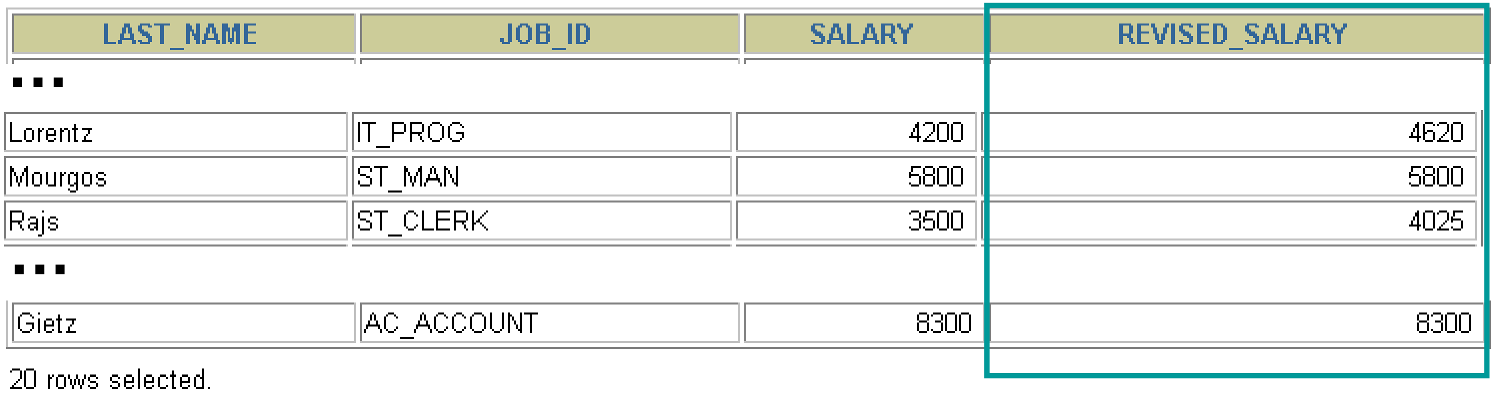
Facilitates conditional inquiries by doing the work of an IF-THEN-ELSE statement:

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

Using the CASE Expression

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

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



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

20 rows selected.
```

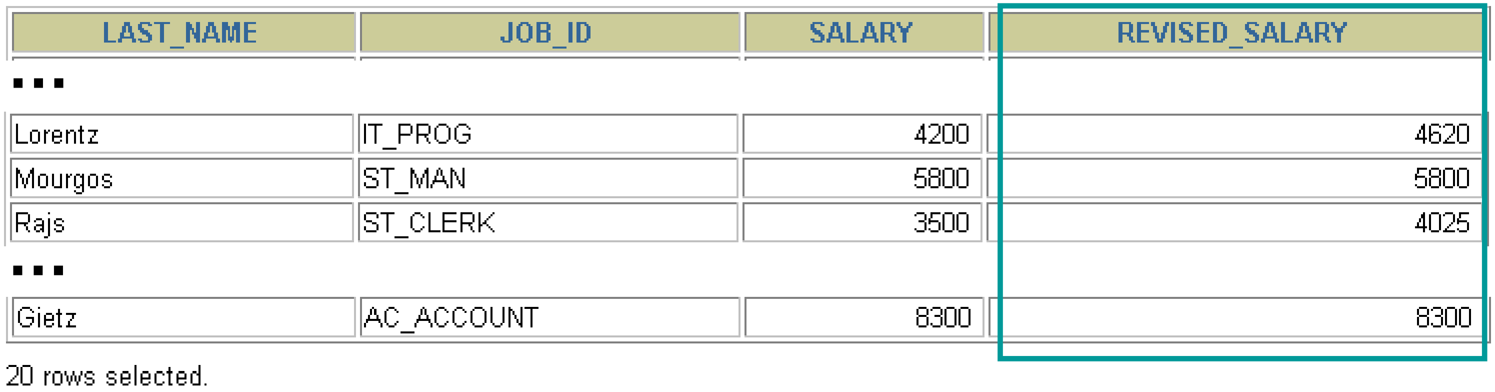
The DECODE Function

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

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

Using the DECODE Function

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



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

20 rows selected.

Using the DECODE Function

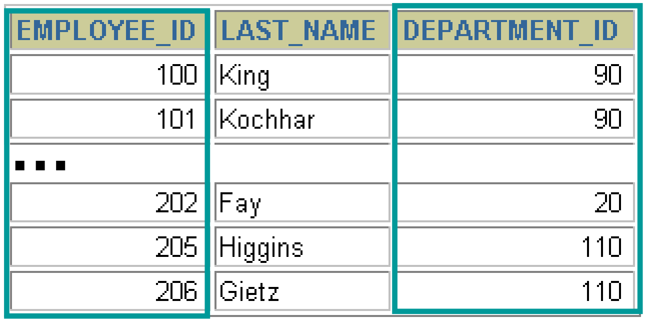
Display the applicable tax rate for each employee in department 80.

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

Displaying Data from Multiple Tables

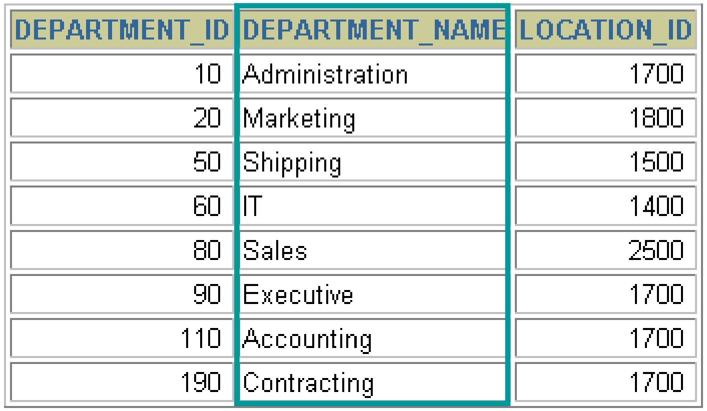
Obtaining Data from Multiple Tables

EMPLOYEES

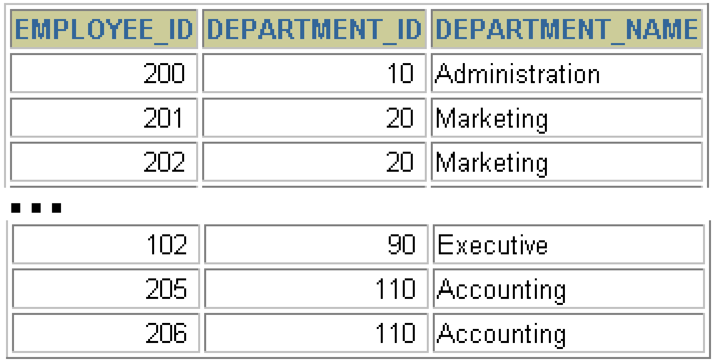


EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90
...		
202	Fay	20
205	Higgins	110
206	Gietz	110

DEPARTMENTS



DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700



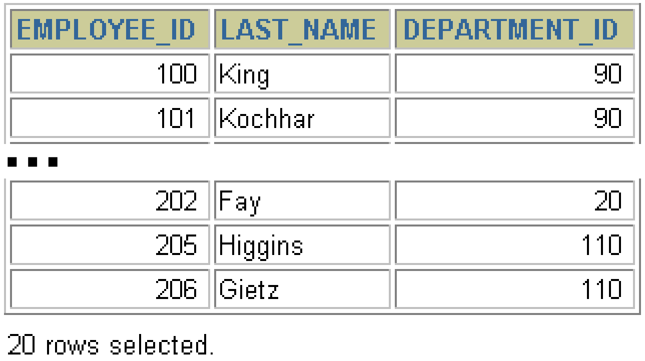
EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
200	10	Administration
201	20	Marketing
202	20	Marketing
...		
102	90	Executive
205	110	Accounting
206	110	Accounting

Cartesian Products

- A Cartesian product is formed when:
  - A join condition is omitted
  - A join condition is invalid
  - All rows in the first table are joined to all rows in the second table
- To avoid a Cartesian product, always include a valid join condition in a WHERE clause.

Generating a Cartesian Product

EMPLOYEES (20 rows)



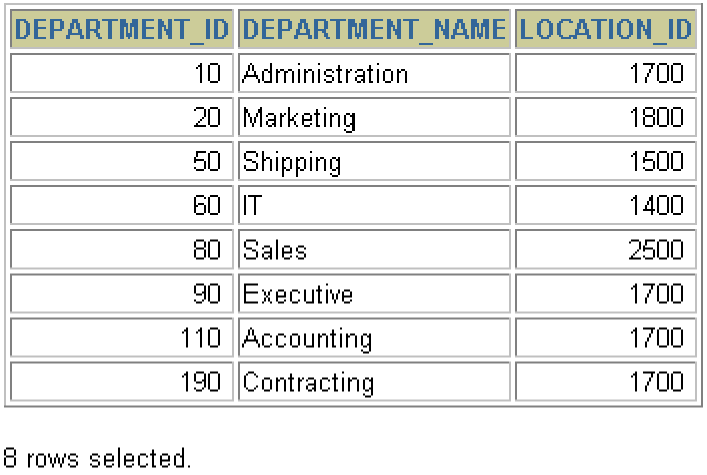
EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
100	King	90
101	Kochhar	90

...

202	Fay	20
205	Higgins	110
206	Gietz	110

20 rows selected.

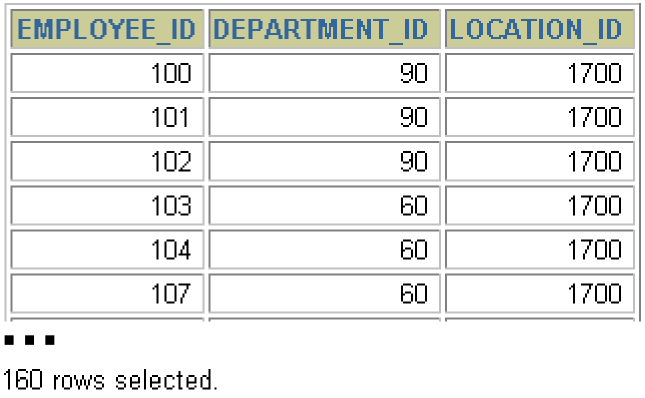
DEPARTMENTS (8 rows)



DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
50	Shipping	1500
60	IT	1400
80	Sales	2500
90	Executive	1700
110	Accounting	1700
190	Contracting	1700

8 rows selected.

Cartesian product: 20x8=160 rows



EMPLOYEE_ID	DEPARTMENT_ID	LOCATION_ID
100	90	1700
101	90	1700
102	90	1700
103	60	1700
104	60	1700
107	60	1700

...

160 rows selected.

Joining Tables Using Oracle Syntax

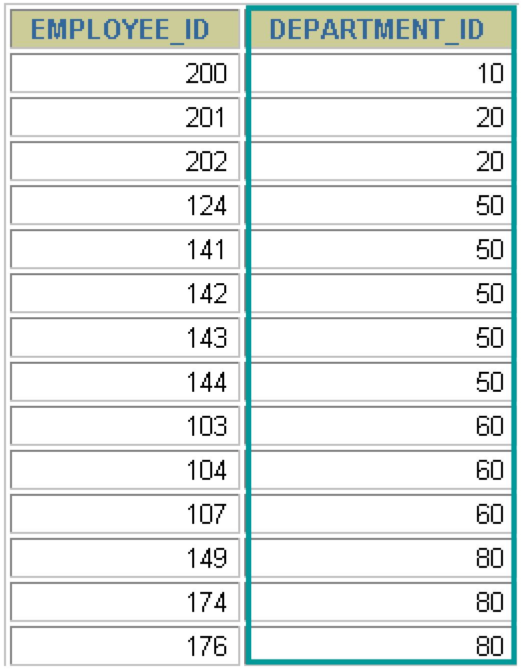
Use a join to query data from more than one table.

```
SELECT   table1.column, table2.column
FROM     table1, table2
WHERE    table1.column1 = table2.column2;
```

- Write the join condition in the WHERE clause.
- Prefix the column name with the table name when the same column name appears in more than one table.

What is an Equijoin?

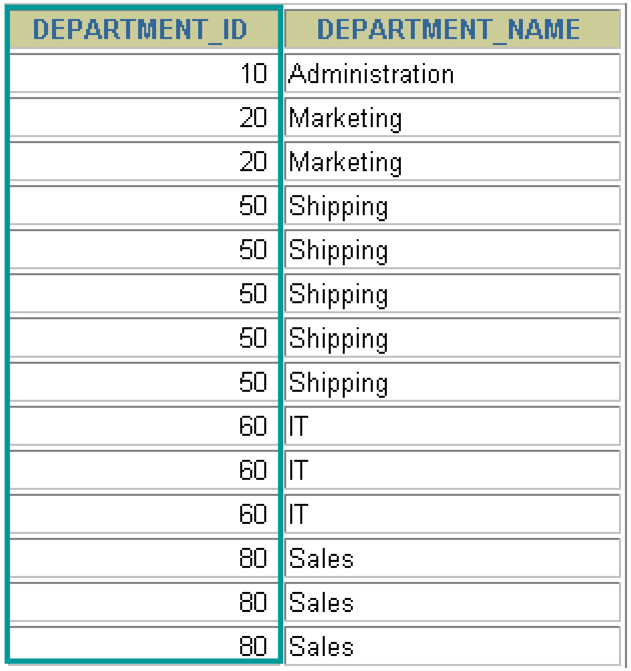
EMPLOYEES



EMPLOYEE_ID	DEPARTMENT_ID
200	10
201	20
202	20
124	50
141	50
142	50
143	50
144	50
103	60
104	60
107	60
149	80
174	80
176	80

...

DEPARTMENTS



DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
60	IT
60	IT
60	IT
80	Sales
80	Sales
80	Sales

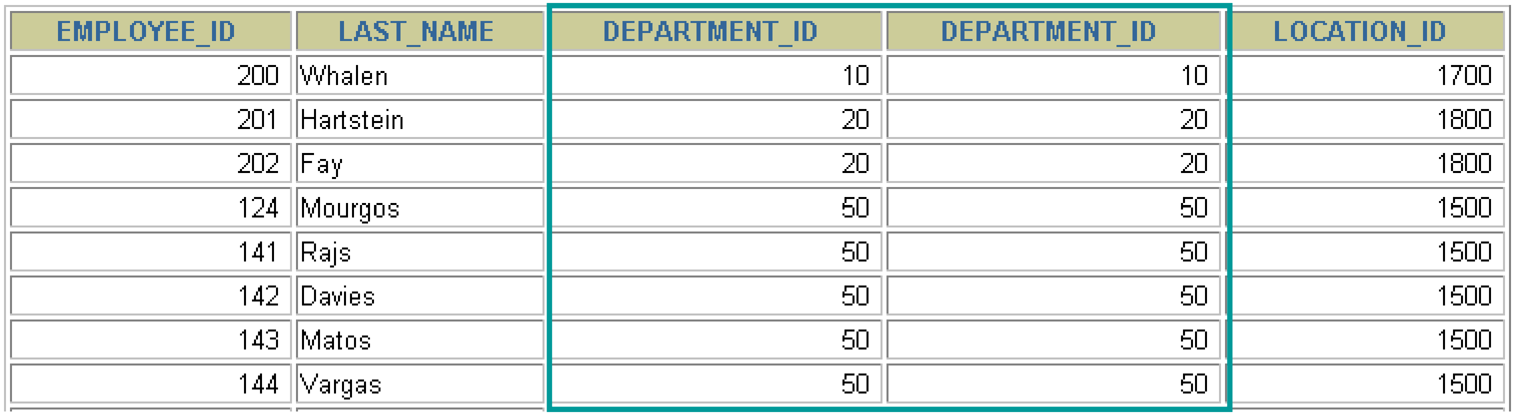
...

Foreign key

Primary key

Retrieving Records with Equijoins

```
SELECT employees.employee_id, employees.last_name,
       employees.department_id, departments.department_id,
       departments.location_id
FROM   employees, departments
WHERE  employees.department_id = departments.department_id;
```



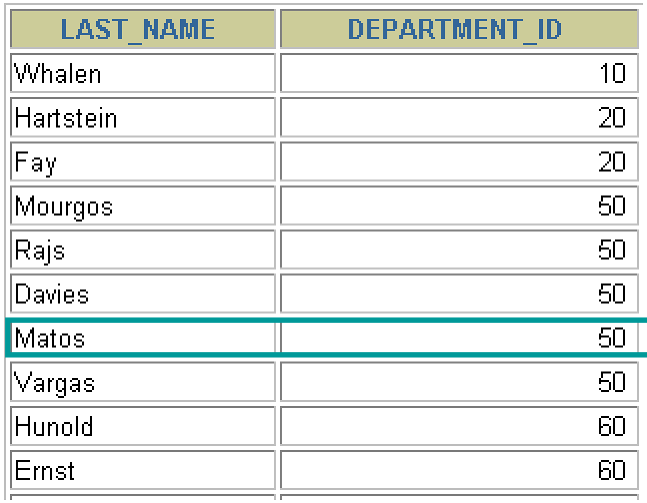
EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500
144	Vargas	50	50	1500

...

19 rows selected.

Additional Search Conditions Using the AND Operator

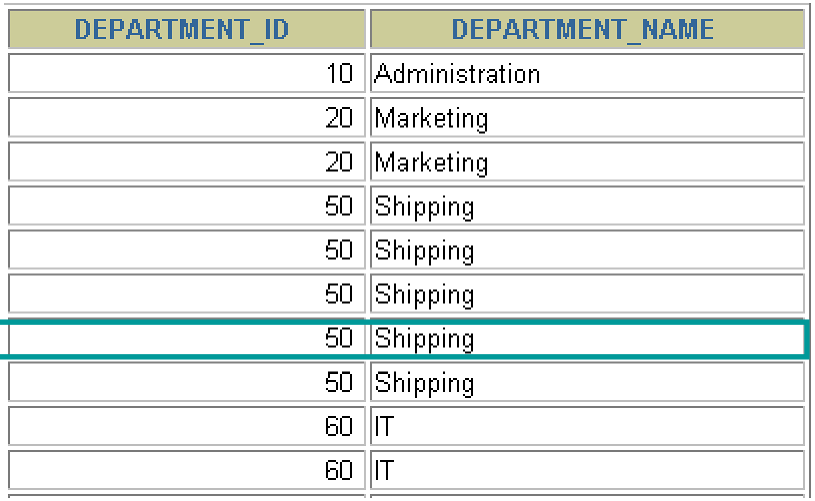
EMPLOYEES



LAST_NAME	DEPARTMENT_ID
Whalen	10
Hartstein	20
Fay	20
Mourgos	50
Rajs	50
Davies	50
Matos	50
Vargas	50
Hunold	60
Ernst	60

…

DEPARTMENTS

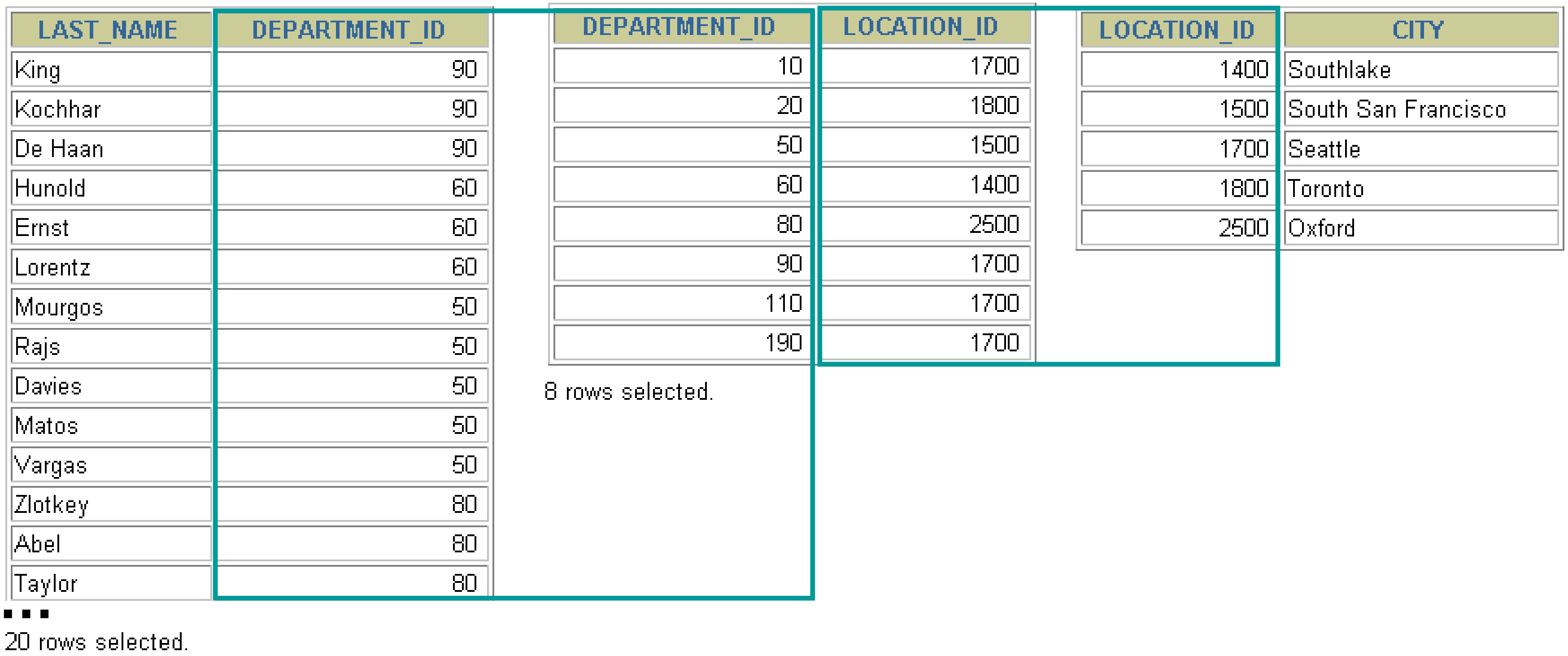


DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
60	IT
60	IT

…

```
SELECT employees.employee_id, employees.last_name,
       employees.department_id, departments.department_id,
       departments.location_id
FROM   employees, departments
WHERE  employees.department_id = departments.department_id
       and departments.department_id = 50;
```

Joining More than Two Tables



EMPLOYEES

LAST_NAME	DEPARTMENT_ID
King	90
Kochhar	90
De Haan	90
Hunold	60
Ernst	60
Lorentz	60
Mourgos	50
Rajs	50
Davies	50
Matos	50
Vargas	50
Zlotkey	80
Abel	80
Taylor	80

…

20 rows selected.

DEPARTMENTS

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

8 rows selected.

LOCATIONS

LOCATION_ID	CITY
1400	Southlake
1500	South San Francisco
1700	Seattle
1800	Toronto
2500	Oxford

- To join *n* tables together, you need a minimum of n-1 join conditions. For example, to join three tables, a minimum of two joins is required.

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Improve performance by using table prefixes.
- Distinguish columns that have identical names but reside in different tables by using column aliases.

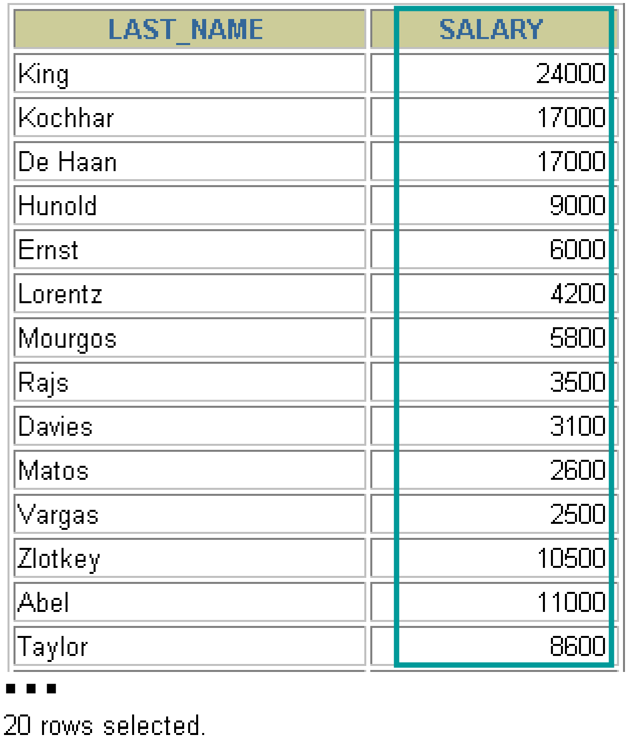
Using Table Aliases

- Simplify queries by using table aliases.
- Improve performance by using table prefixes.

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e , departments d
WHERE  e.department_id = d.department_id;
```

Non-Equijoins

EMPLOYEES

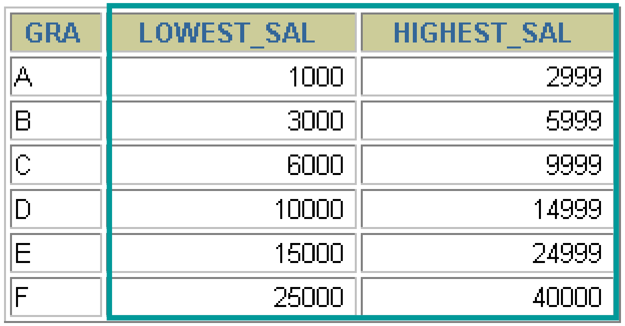


LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000
Ernst	6000
Lorentz	4200
Mourgos	5800
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500
Zlotkey	10500
Abel	11000
Taylor	8600

…

20 rows selected.

JOB_GRADES

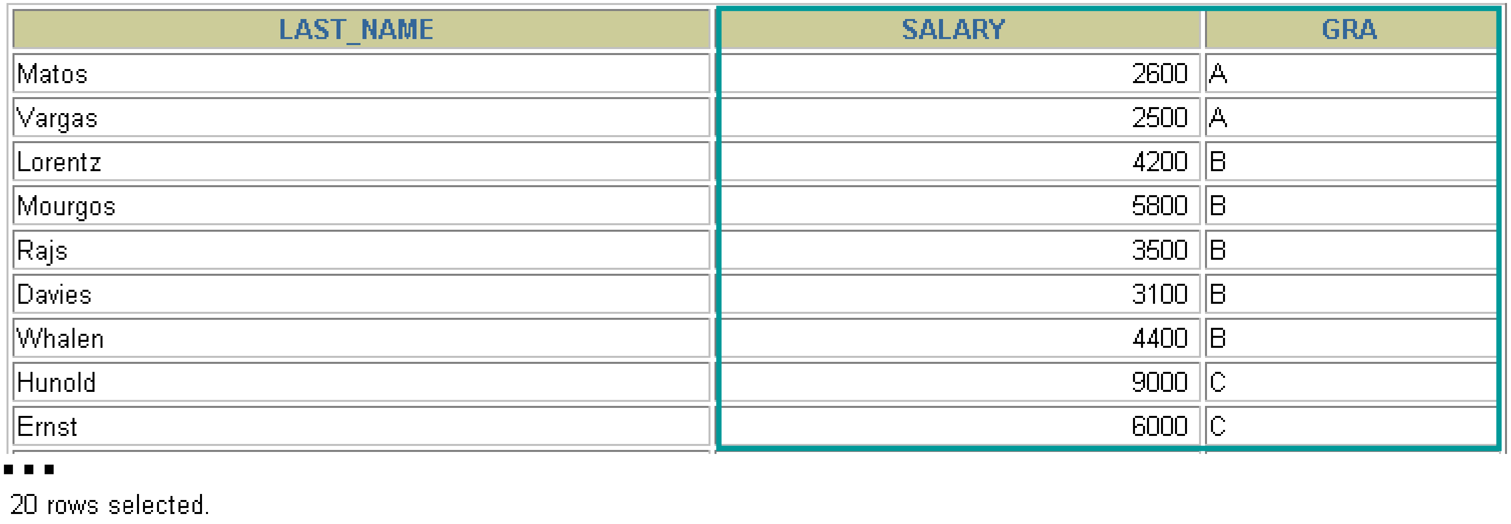


GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

Salary in the `EMPLOYEES` table must be between lowest salary and highest salary in the `JOB_GRADES` table.

Retrieving Records with Non-Equijoins

```
SELECT e.last_name, e.salary, j.grade_level
FROM   employees e, job_grades j
WHERE  e.salary
       BETWEEN j.lowest_sal AND j.highest_sal;
```



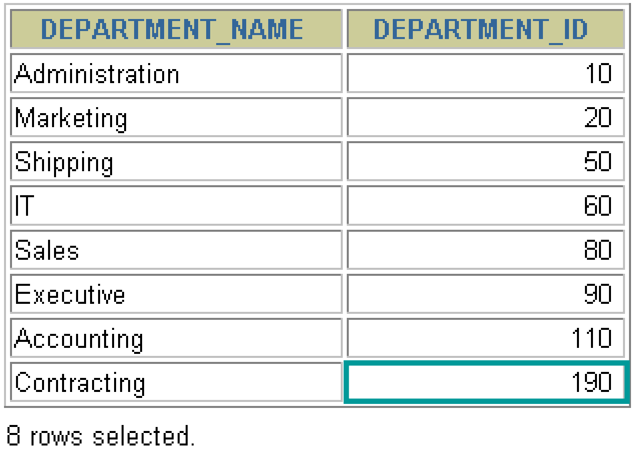
LAST_NAME	SALARY	GRA
Matos	2600	A
Vargas	2500	A
Lorentz	4200	B
Mourgos	5800	B
Rajs	3500	B
Davies	3100	B
Whalen	4400	B
Hunold	9000	C
Ernst	6000	C

…

20 rows selected.

Outer Joins

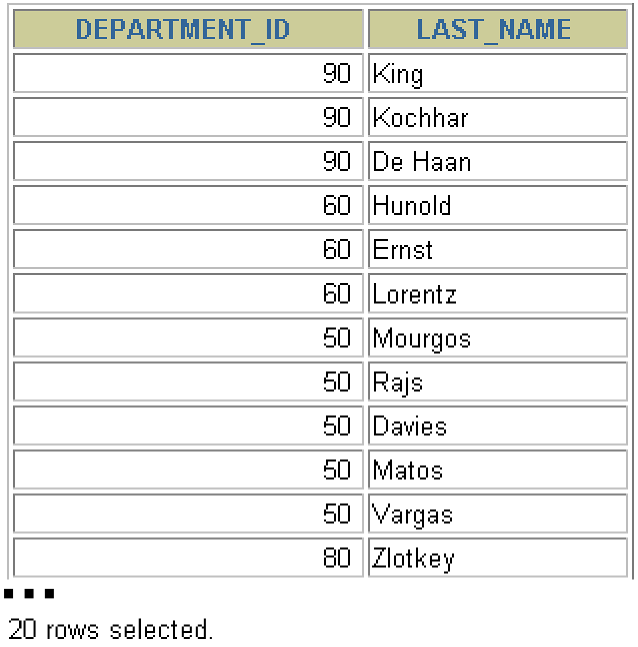
DEPARTMENTS



DEPARTMENT_NAME	DEPARTMENT_ID
Administration	10
Marketing	20
Shipping	50
IT	60
Sales	80
Executive	90
Accounting	110
Contracting	190

8 rows selected.

EMPLOYEES



DEPARTMENT_ID	LAST_NAME
90	King
90	Kochhar
90	De Haan
60	Hunold
60	Ernst
60	Lorentz
50	Mourgos
50	Rajs
50	Davies
50	Matos
50	Vargas
80	Zlotkey

…

20 rows selected.

There are no employees in department 190.

Outer Joins Syntax

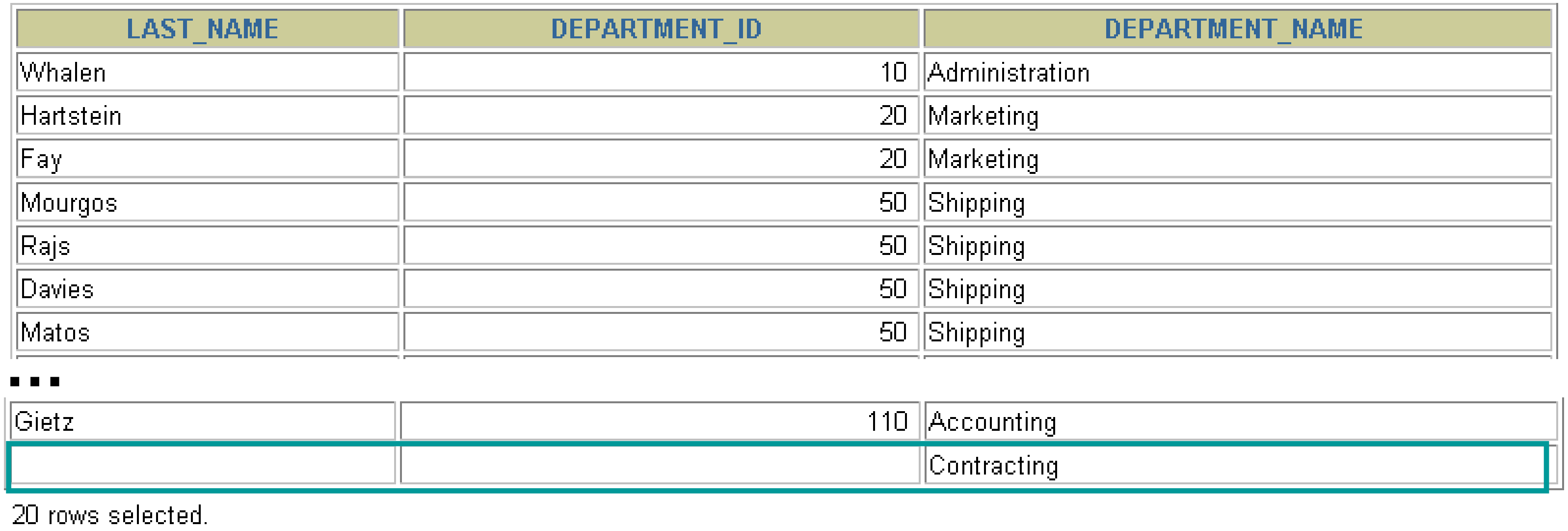
- You use an outer join to also see rows that do not meet the join condition.
- The Outer join operator is the plus sign (+).

```
SELECT table1.column, table2.column
FROM   table1, table2
WHERE  table1.column(+) = table2.column;
```

```
SELECT table1.column, table2.column
FROM   table1, table2
WHERE  table1.column = table2.column(+);
```

Using Outer Joins

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

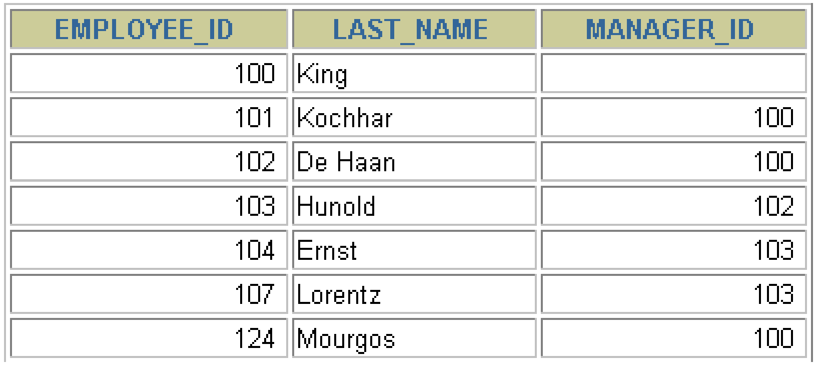


LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Mourgos	50	Shipping
Rajs	50	Shipping
Davies	50	Shipping
Matos	50	Shipping
...		
Gietz	110	Accounting
		Contracting

20 rows selected.

Self Joins

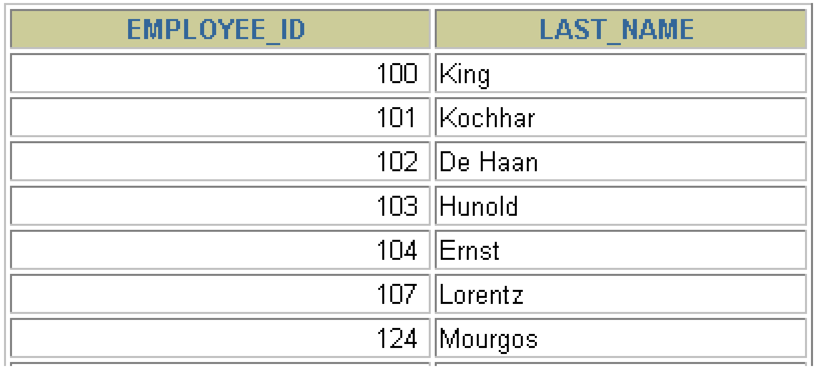
EMPLOYEES (WORKER)



EMPLOYEE_ID	LAST_NAME	MANAGER_ID
100	King	
101	Kochhar	100
102	De Haan	100
103	Hunold	102
104	Ernst	103
107	Lorentz	103
124	Mourgos	100

…

EMPLOYEES (MANAGER)



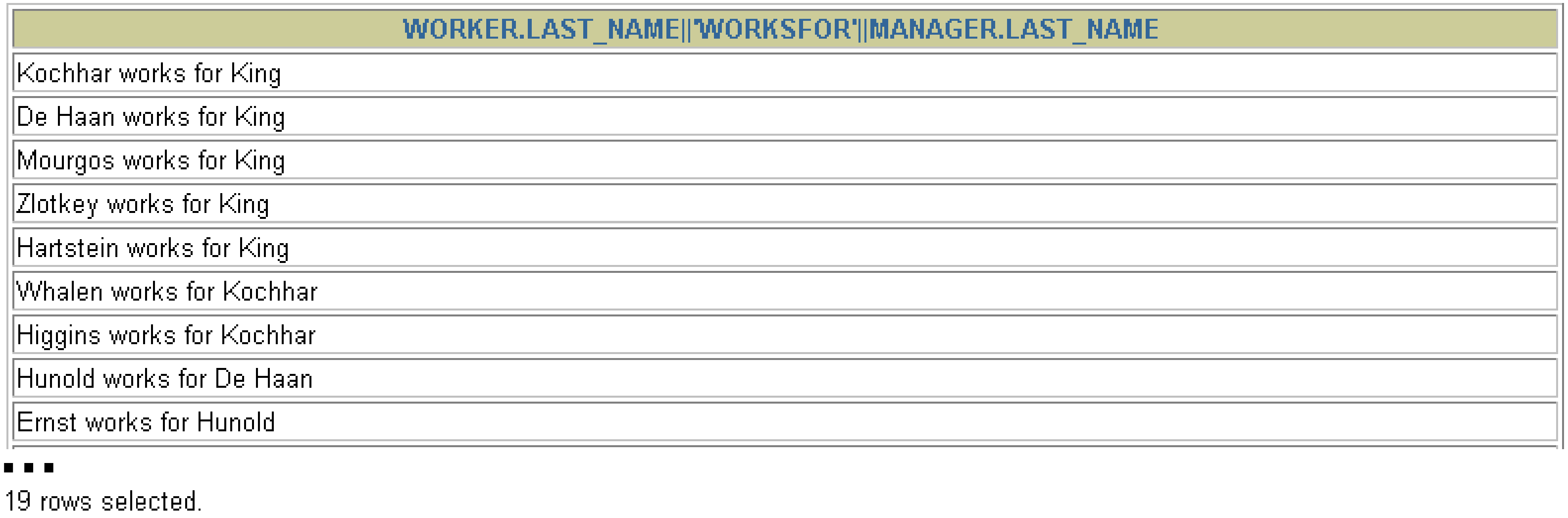
EMPLOYEE_ID	LAST_NAME
100	King
101	Kochhar
102	De Haan
103	Hunold
104	Ernst
107	Lorentz
124	Mourgos

…

MANAGER_ID in the WORKER table is equal to EMPLOYEE_ID in the MANAGER table.

Joining a Table to Itself

```
SELECT worker.last_name || ' works for '
       || manager.last_name
FROM   employees worker, employees manager
WHERE  worker.manager_id = manager.employee_id ;
```



WORKER.LAST_NAME||'WORKSFOR'||MANAGER.LAST_NAME
Kochhar works for King
De Haan works for King
Mourgos works for King
Zlotkey works for King
Hartstein works for King
Whalen works for Kochhar
Higgins works for Kochhar
Hunold works for De Haan
Ernst works for Hunold

…

19 rows selected.

Joining Tables Using SQL: 1999 Syntax

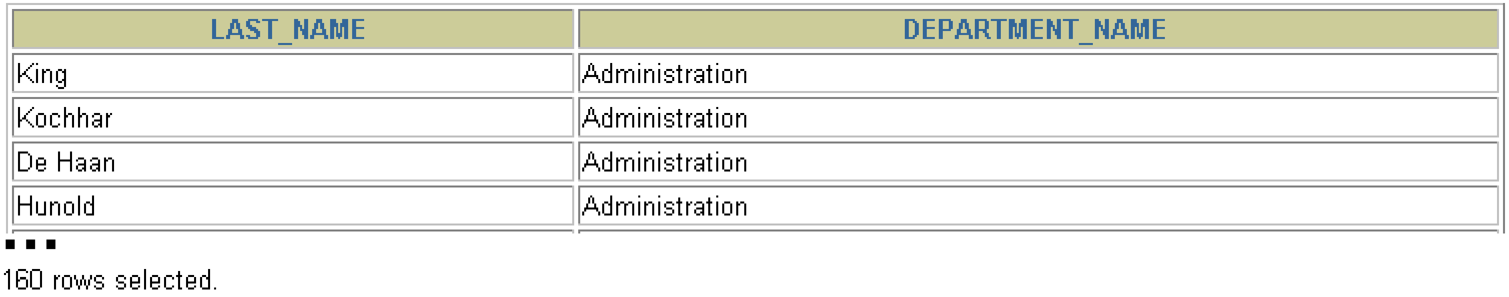
Use a join to query data from more than one table.

```
SELECT   table1.column, table2.column
FROM     table1
[CROSS JOIN table2] |
[NATURAL JOIN table2] |
[JOIN table2 USING (column_name)] |
[JOIN table2
  ON(table1.column_name = table2.column_name)] |
[LEFT|RIGHT|FULL OUTER JOIN table2
  ON (table1.column_name = table2.column_name)];
```

Creating Cross Joins

- The `CROSS JOIN` clause produces the cross-product of two tables.
- This is the same as a Cartesian product between the two tables.

```
SELECT last_name, department_name
FROM   employees
CROSS JOIN departments ;
```



LAST_NAME	DEPARTMENT_NAME
King	Administration
Kochhar	Administration
De Haan	Administration
Hunold	Administration

...

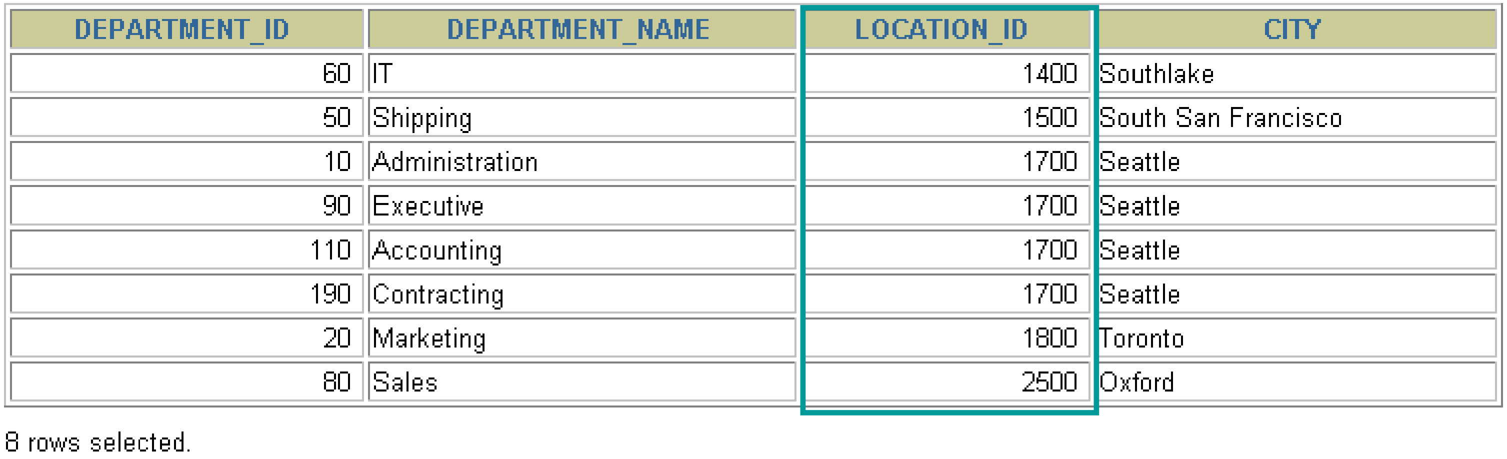
160 rows selected.

Creating Natural Joins

- The `NATURAL JOIN` clause is based on all columns in the two tables that have the same name.
- It selects rows from the two tables that have equal values in all matched columns.
- If the columns having the same names have different data types, an error is returned.

Retrieving Records with Natural Joins

```
SELECT department_id, department_name,
       location_id, city
FROM   departments
NATURAL JOIN locations ;
```



DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
60	IT	1400	Southlake
50	Shipping	1500	South San Francisco
10	Administration	1700	Seattle
90	Executive	1700	Seattle
110	Accounting	1700	Seattle
190	Contracting	1700	Seattle
20	Marketing	1800	Toronto
80	Sales	2500	Oxford

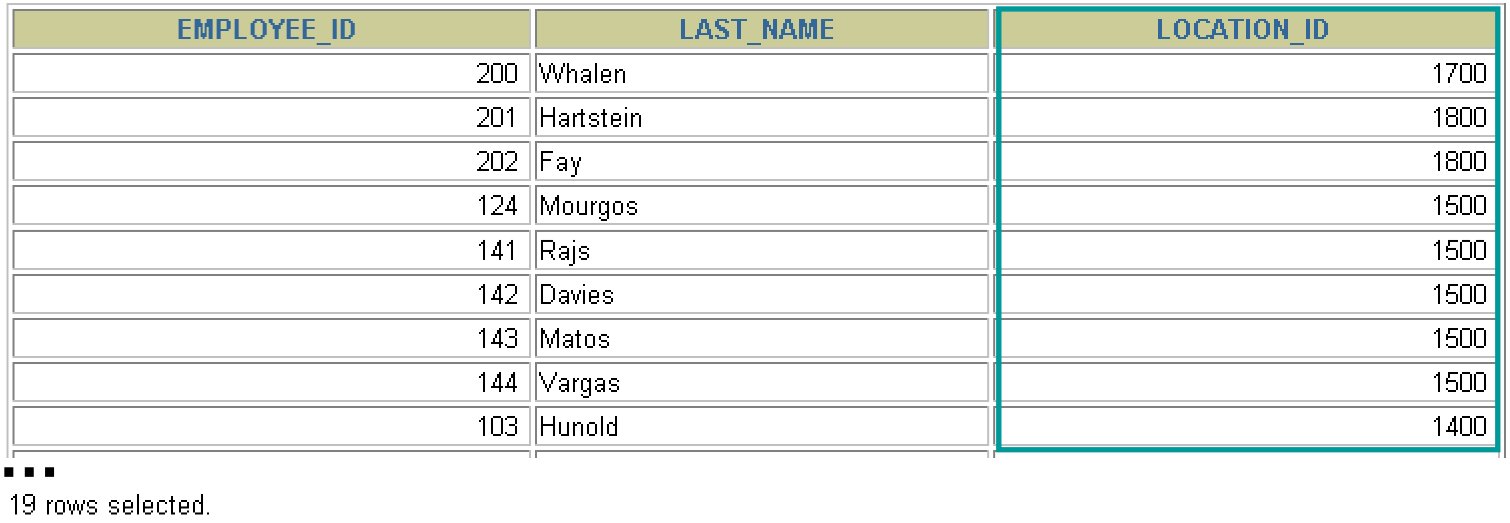
8 rows selected.

Creating Joins with the USING Clause

- If several columns have the same names but the data types do not match, the NATURAL JOIN clause can be modified with the USING clause to specify the columns that should be used for an equijoin.
- Use the USING clause to match only one column when more than one column matches.
- Do not use a table name or alias in the referenced columns.
- The NATURAL JOIN and USING clauses are mutually exclusive.

Retrieving Records with the USING Clause

```
SELECT e.employee_id, e.last_name, d.location_id
FROM   employees e JOIN departments d
USING (department_id) ;
```



EMPLOYEE_ID	LAST_NAME	LOCATION_ID
200	Whalen	1700
201	Hartstein	1800
202	Fay	1800
124	Mourgos	1500
141	Rajs	1500
142	Davies	1500
143	Matos	1500
144	Vargas	1500
103	Hunold	1400

…

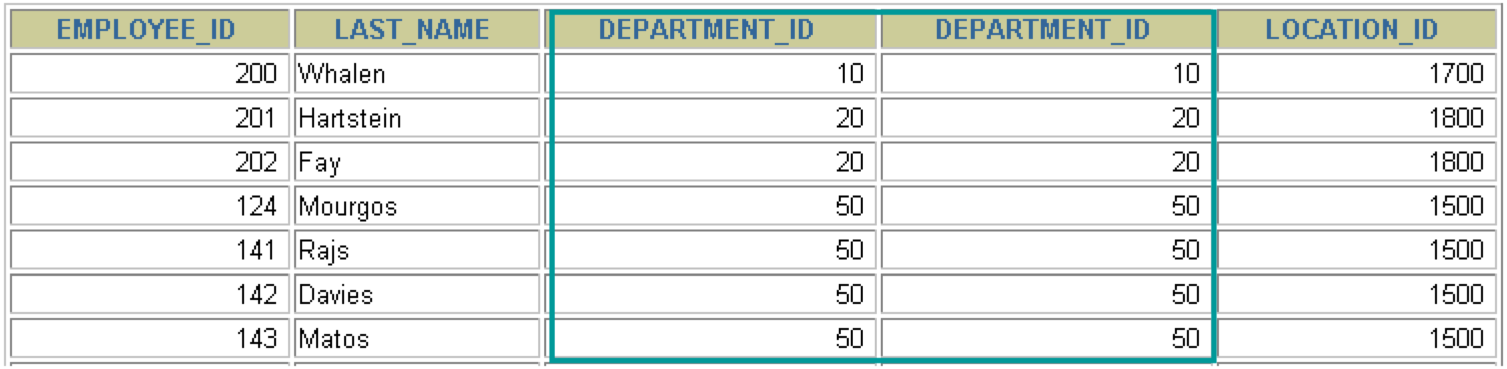
19 rows selected.

Creating Joins with the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- To specify arbitrary conditions or specify columns to join, the ON clause is used.
- The join condition is separated from other *search* conditions.
- The ON clause makes code easy to understand.

Retrieving Records with the ON Clause

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id);
```



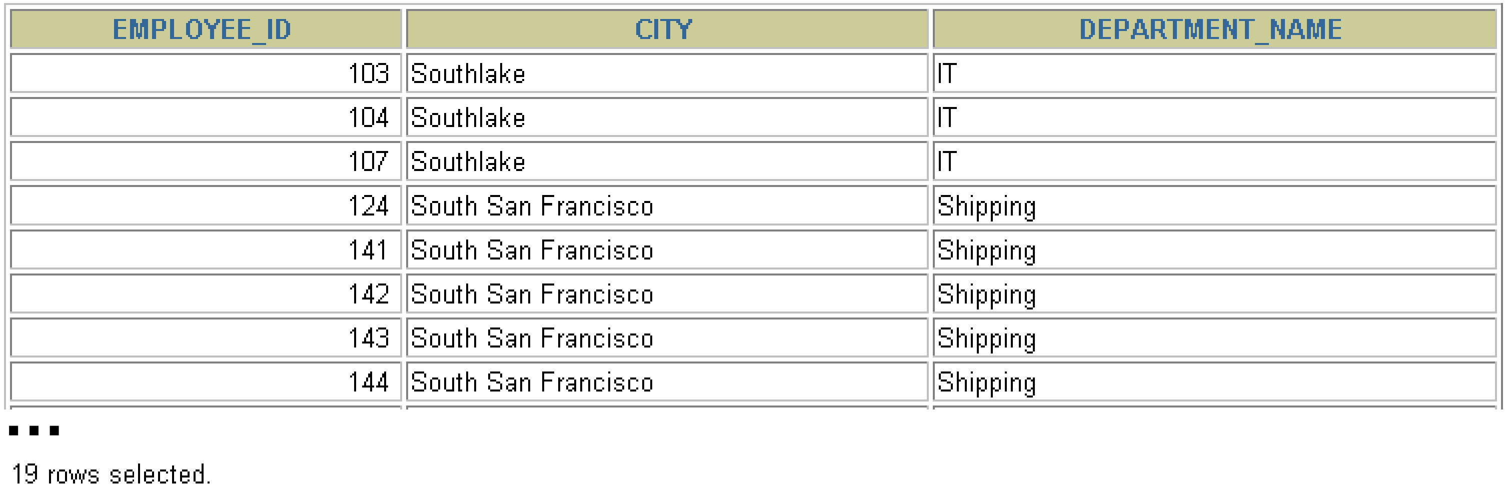
EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500

...

19 rows selected.

Creating Three-Way Joins with the ON Clause

```
SELECT employee_id, city, department_name
FROM   employees e
JOIN   departments d
ON     d.department_id = e.department_id
JOIN   locations l
ON     d.location_id = l.location_id;
```



EMPLOYEE_ID	CITY	DEPARTMENT_NAME
103	Southlake	IT
104	Southlake	IT
107	Southlake	IT
124	South San Francisco	Shipping
141	South San Francisco	Shipping
142	South San Francisco	Shipping
143	South San Francisco	Shipping
144	South San Francisco	Shipping

…

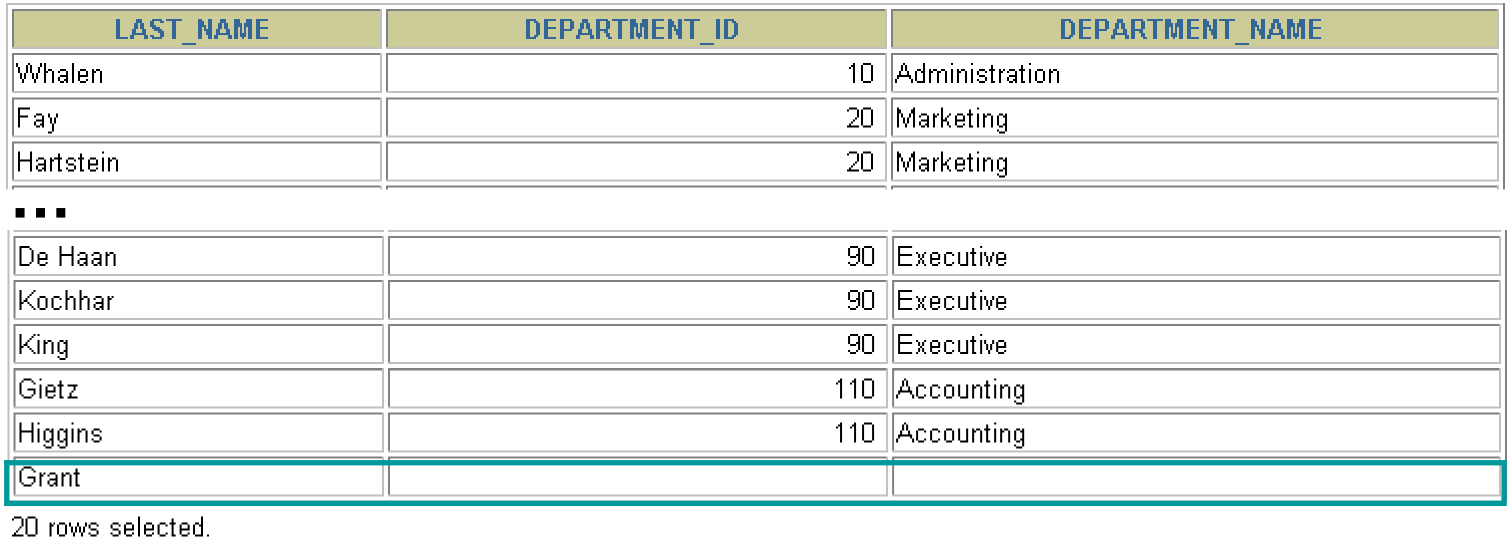
19 rows selected.

INNER Versus OUTER Joins

- The join of two tables returning only matched rows is an inner join.
- A join between two tables that returns the results of the inner join as well as unmatched rows left (or right) tables is a left (or right) outer join.
- A join between two tables that returns the results of an inner join as well as the results of a left and right join is a full outer join.

LEFT OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e
LEFT OUTER JOIN departments d
ON   (e.department_id = d.department_id) ;
```

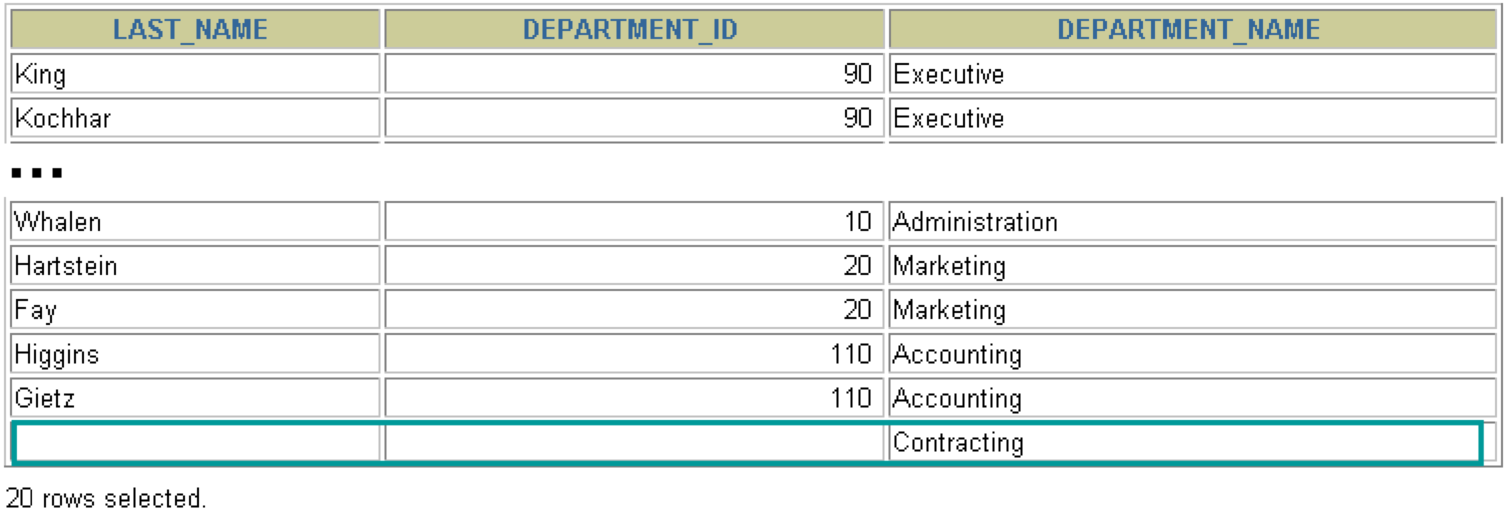


LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
Hartstein	20	Marketing
...		
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		

20 rows selected.

RIGHT OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e
RIGHT OUTER JOIN departments d
ON     (e.department_id = d.department_id) ;
```

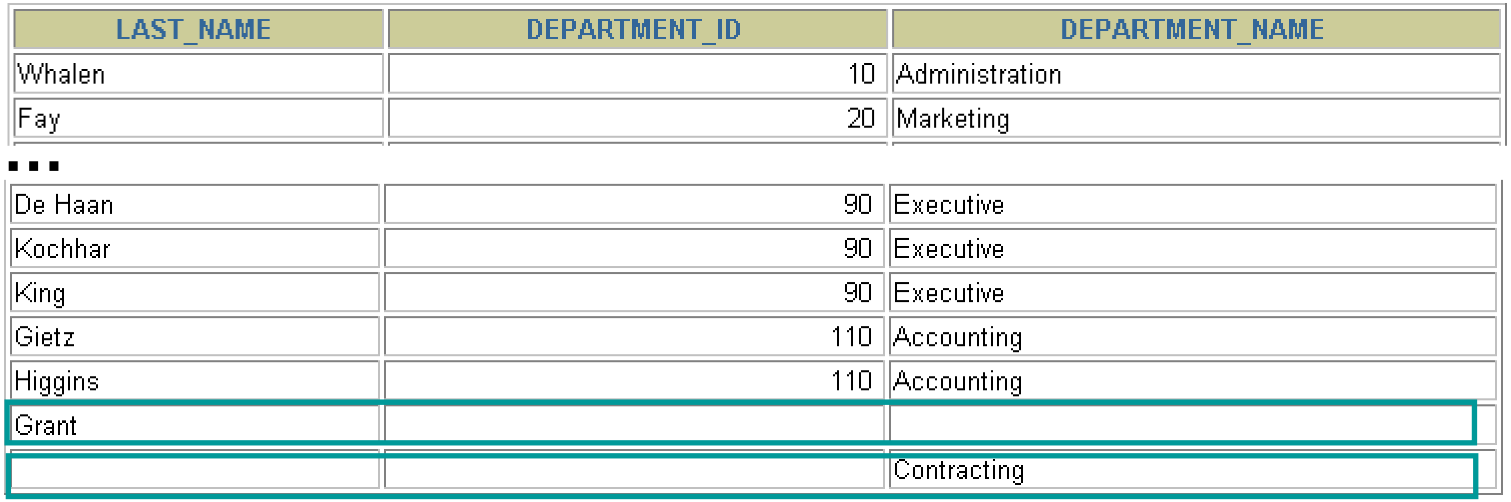


LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
King	90	Executive
Kochhar	90	Executive
...		
Whalen	10	Administration
Hartstein	20	Marketing
Fay	20	Marketing
Higgins	110	Accounting
Gietz	110	Accounting
		Contracting

20 rows selected.

FULL OUTER JOIN

```
SELECT e.last_name, e.department_id, d.department_name
FROM   employees e
FULL OUTER JOIN departments d
ON   (e.department_id = d.department_id) ;
```

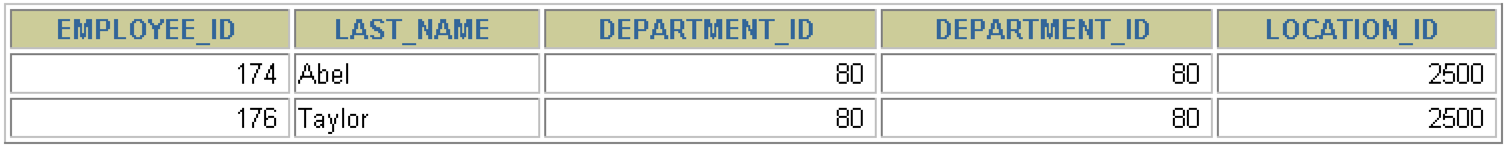


LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
Whalen	10	Administration
Fay	20	Marketing
...		
De Haan	90	Executive
Kochhar	90	Executive
King	90	Executive
Gietz	110	Accounting
Higgins	110	Accounting
Grant		
		Contracting

```
21 rows selected.
```

Additional Conditions

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e JOIN departments d
ON     (e.department_id = d.department_id)
AND    e.manager_id = 149 ;
```



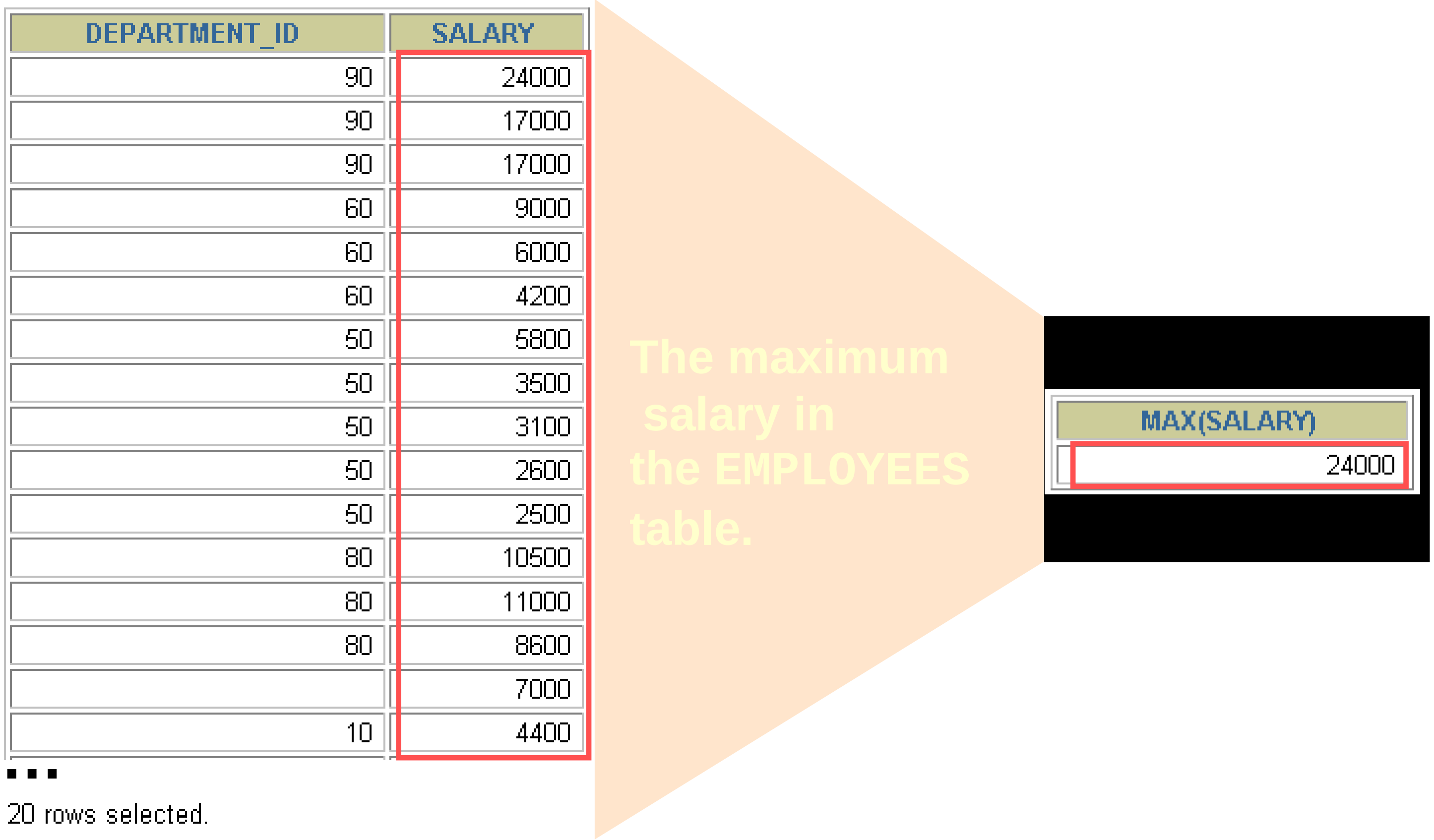
EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
174	Abel	80	80	2500
176	Taylor	80	80	2500

Aggregating Data Using Group Functions

What Are Group Functions?

Group functions operate on sets of rows to give one result per group.

EMPLOYEES



DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600
	7000
10	4400

…

20 rows selected.

The maximum salary in the EMPLOYEES table.

MAX(SALARY)
24000

Types of Group Functions

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

Group functions ignore null values in the column.

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

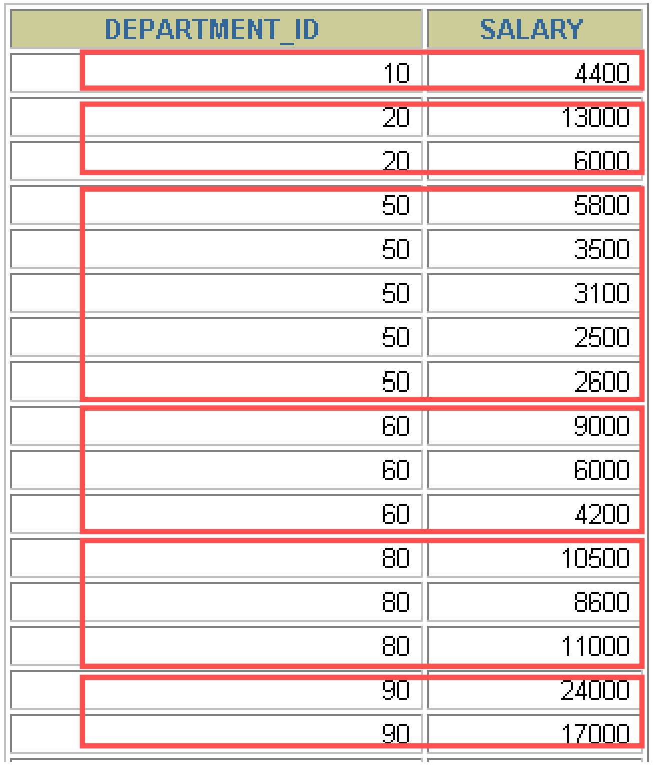
```
SELECT  AVG(NVL(commission_pct, 0))
FROM    employees;
```

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

Birlasoft®

Creating Groups of Data

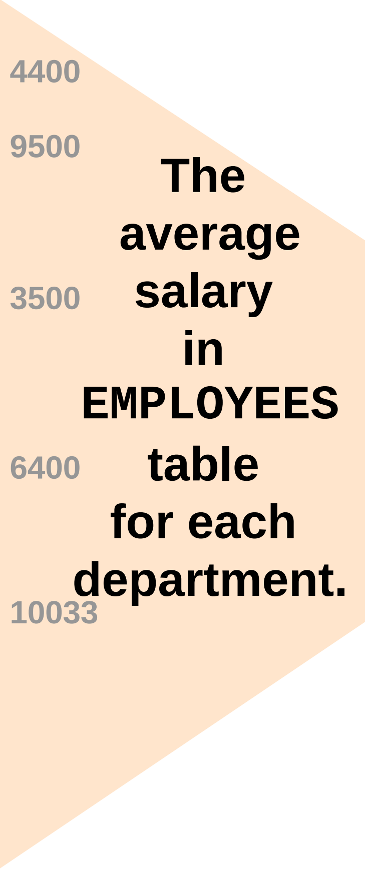
EMPLOYEES



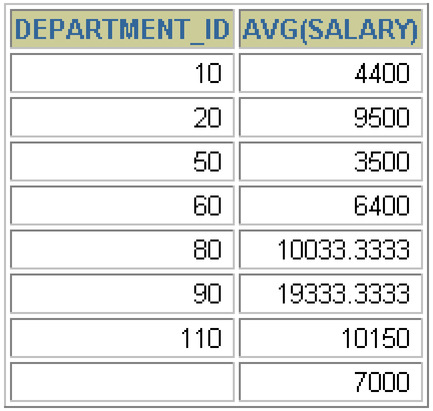
DEPARTMENT_ID	SALARY
10	4400
20	13000
20	6000
50	5800
50	3500
50	3100
50	2500
50	2600
60	9000
60	6000
60	4200
80	10500
80	8600
80	11000
90	24000
90	17000

...

20 rows selected.



The average salary in EMPLOYEES table for each department.



DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

Birlasoft®

Creating Groups of Data

- The GROUP BY column does not have to be in the SELECT list
- You cannot use the WHERE clause to restrict groups.
- You use the HAVING clause to restrict groups.
- You cannot use group functions in the WHERE clause.

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

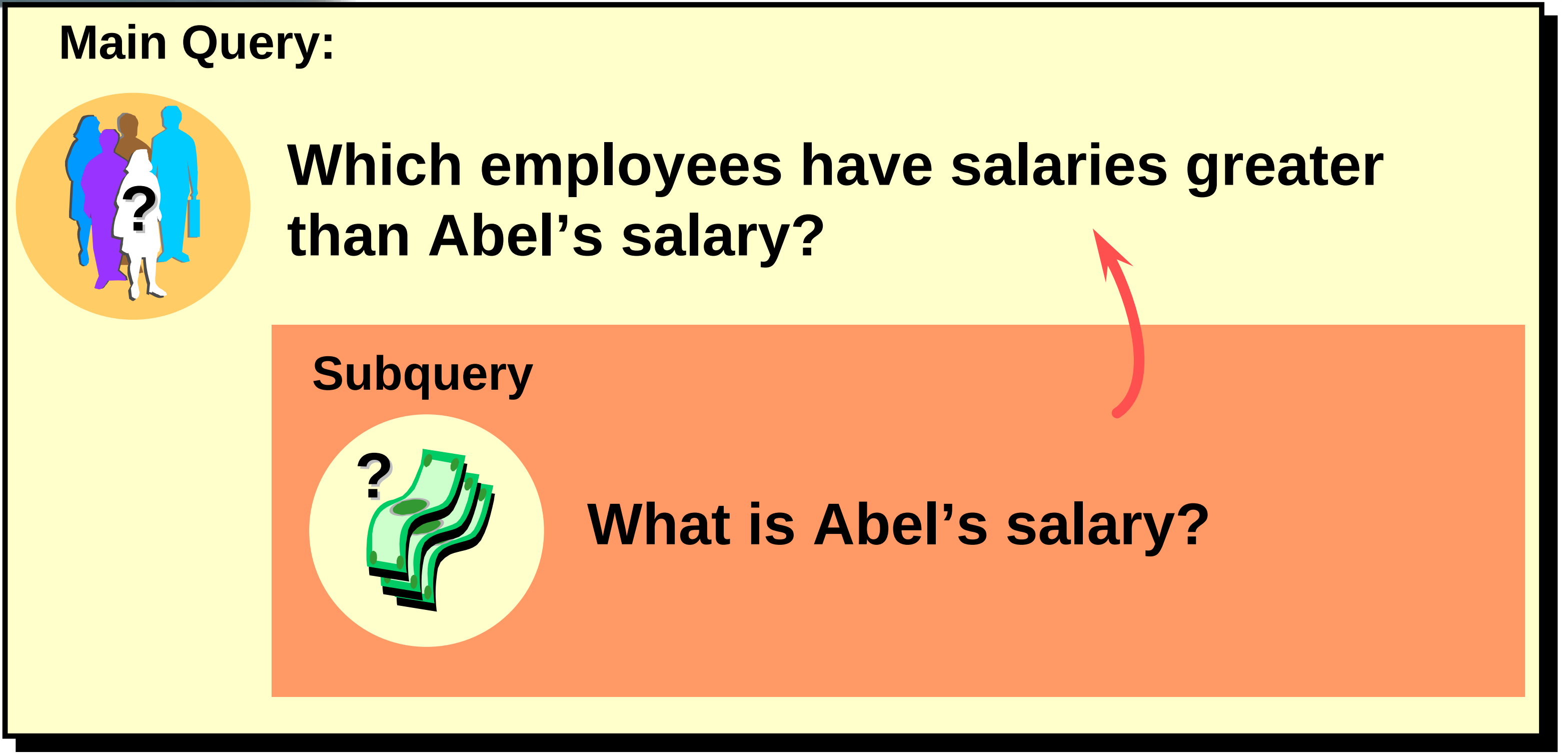
```
SELECT    department_id dept_id, job_id, SUM(salary)
FROM      employees
GROUP BY department_id, job_id ;
```

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

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

Subqueries

Using a Subquery



Main Query:

Which employees have salaries greater than Abel’s salary?

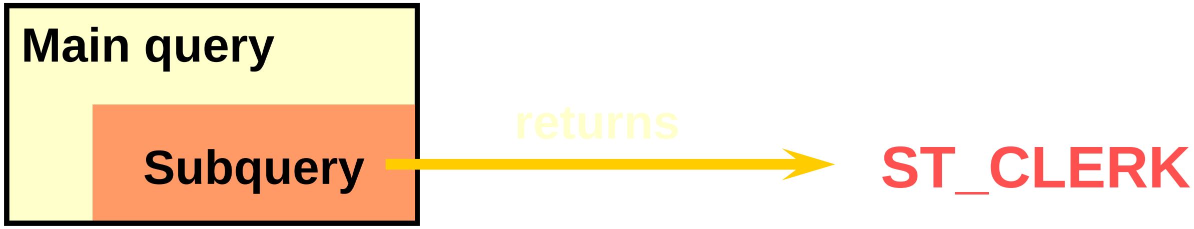
Subquery

What is Abel’s salary?

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

Using a Subquery

- Single-row subquery



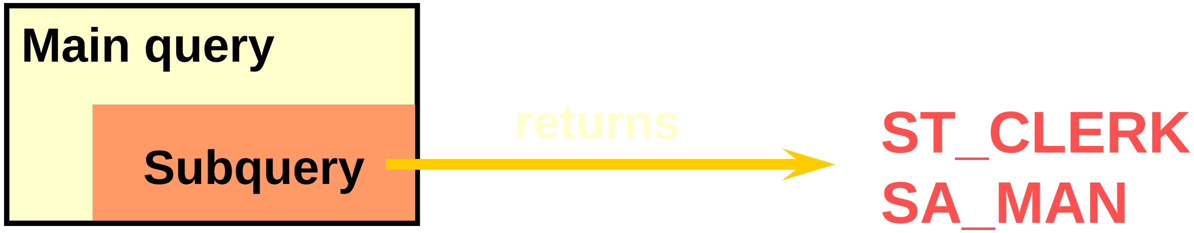
Main query

Subquery

returns

ST_CLERK

- Multiple-row subquery



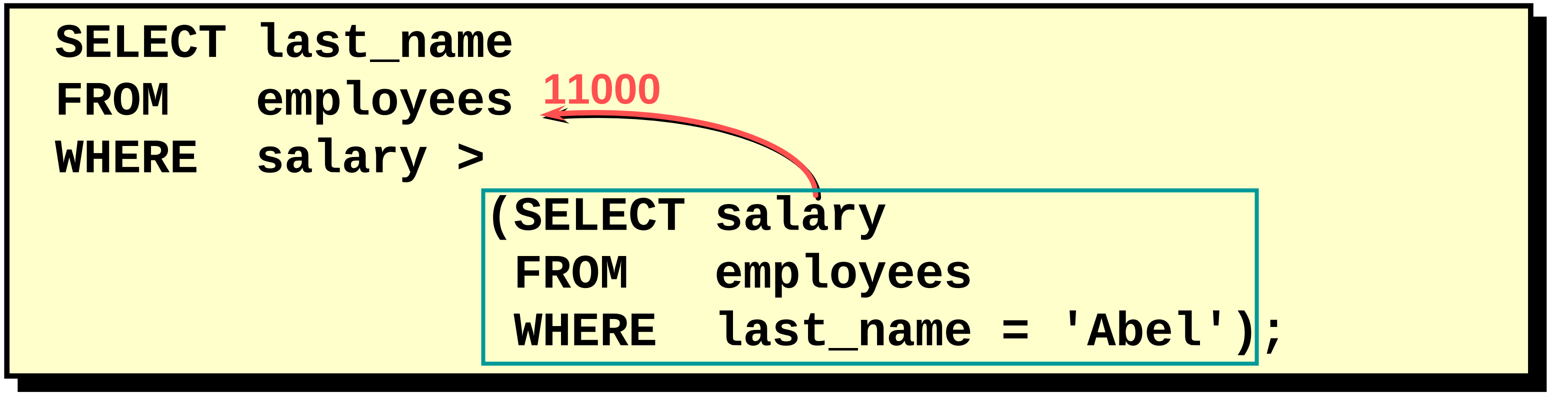
Main query

Subquery

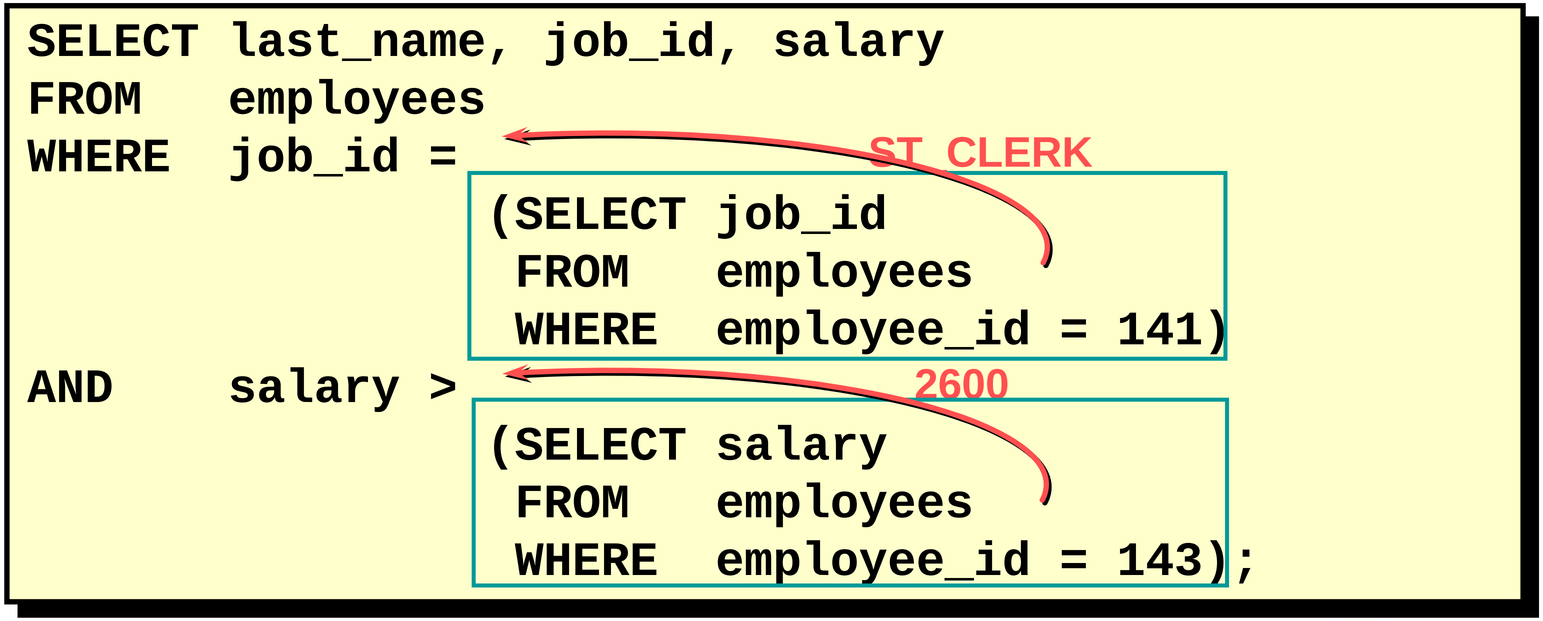
returns

ST_CLERK
SA_MAN

Executing Single-Row Subqueries

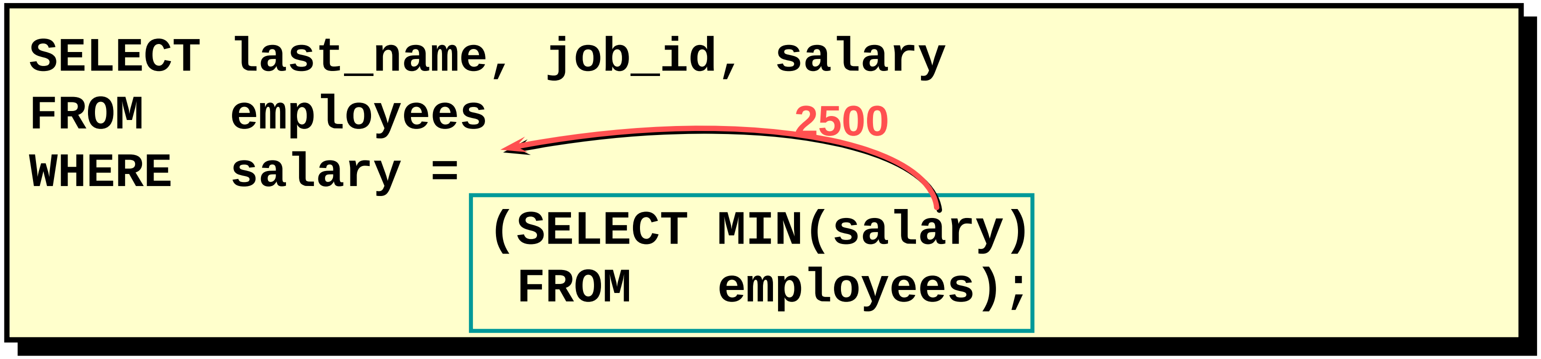


```
SELECT last_name
FROM   employees
WHERE  salary >
               (SELECT salary
                FROM   employees
                WHERE  last_name = 'Abel');
```



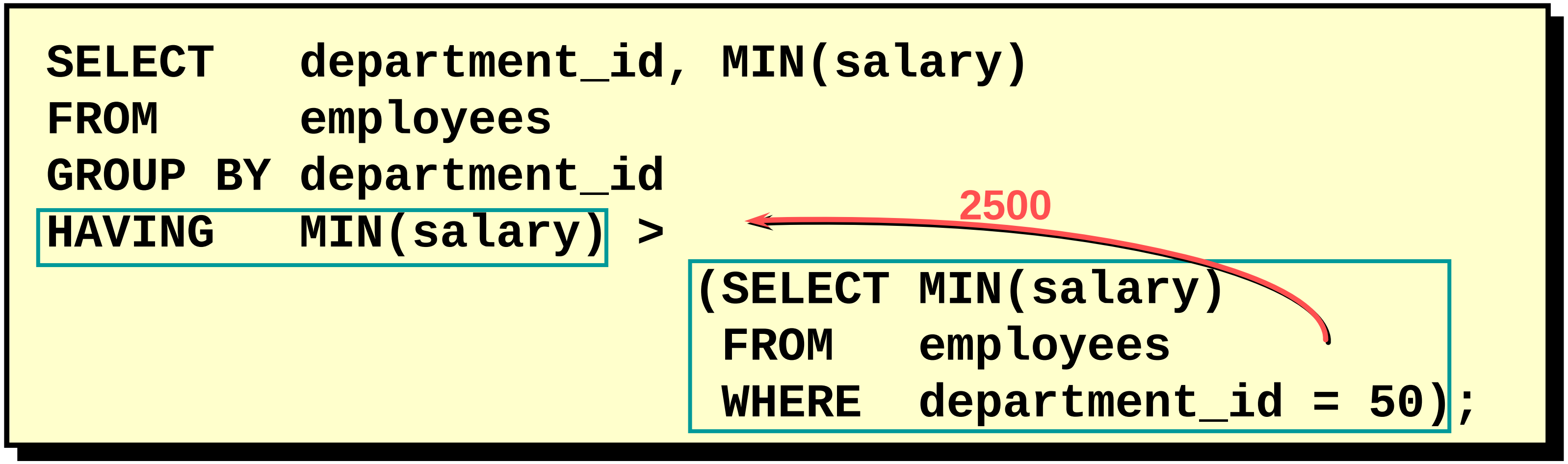
```
SELECT last_name, job_id, salary
FROM   employees
WHERE  job_id =
               (SELECT job_id
                FROM   employees
                WHERE  employee_id = 141)
AND    salary >
               (SELECT salary
                FROM   employees
                WHERE  employee_id = 143);
```

Using Group Functions in a Subquery



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

The HAVING Clause with Subqueries



```
SELECT   department_id, MIN(salary)
FROM     employees
GROUP BY department_id
HAVING   MIN(salary) >
                          (SELECT MIN(salary)
                           FROM   employees
                           WHERE  department_id = 50);
```

Executing Multiple-Row Subqueries

```
SELECT employee_id, last_name
FROM   employees
WHERE  salary =
                (SELECT   MIN(salary)
                 FROM     employees
                 GROUP BY department_id);
```

```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary < ANY
                    (SELECT salary
                     FROM   employees
                     WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```

9000, 6000, 4200

```
SELECT emp.last_name
FROM   employees emp
WHERE  emp.employee_id NOT IN
                            (SELECT mgr.manager_id
                             FROM   employees mgr);
```

Substitution Variables

Substitution Variables

Use substitution variables to:

- Temporarily store values
  - Single ampersand (&)
  - Double ampersand (&&)
  - `DEFINE` command
- Pass variable values between SQL statements
- Dynamically alter headers and footers
- You can predefine variables using the `DEFINE` command.

  `DEFINE` *`variable`* `=` *`value`* creates a user variable with the CHAR data type.
- If you need to predefine a variable that includes spaces, you must enclose the value within single quotation marks when using the `DEFINE` command.
- A defined variable is available for the session

Using the & Substitution Variable

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

```
SELECT last_name, department_id, salary*12
FROM   employees
WHERE  job_id = '&job_title' ;
```

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

```
DEFINE job_title = IT_PROG
DEFINE job_title
DEFINE JOB_TITLE       = "IT_PROG" (CHAR)
```

Manipulating Data

Data Manipulation Language

- A DML statement is executed when you:
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
- A *transaction* consists of a collection of DML statements that form a logical unit of work.

Inserting New Rows

- Insert a new row containing values for each column.
- List values in the default order of the columns in the table.
- Optionally, list the columns in the `INSERT` clause.
- Enclose character and date values within single quotation marks.

```
INSERT INTO departments(department_id, department_name,
                        manager_id, location_id)
VALUES      (70, 'Public Relations', 100, 1700);
1 row created.
```

```
INSERT INTO  departments (department_id,
                          department_name□ □)
VALUES       (30, 'Purchasing');
1 row created.
```

Inserting Special Values

The SYSDATE function records the current date and time.

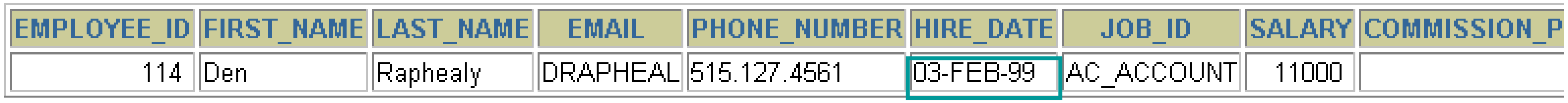
```
INSERT INTO employees (employee_id,
                 first_name, last_name,
                 email, phone_number,
                 hire_date, job_id, salary,
                 commission_pct, manager_id,
                 department_id)
VALUES          (113,
                 'Louis', 'Popp',
                 'LPOPP', '515.124.4567',
                 SYSDATE, 'AC_ACCOUNT', 6900,
                 NULL, 205, 100);
1 row created.
```

Inserting Specific Date Values

- Add a new employee.

```
INSERT INTO employees
VALUES      (114,
             'Den', 'Raphealy',
             'DRAPHEAL', '515.127.4561',
             TO_DATE('FEB 3, 1999', 'MON DD, YYYY'),
             'AC_ACCOUNT', 11000, NULL, 100, 30);
1 row created.
```

- Verify your addition.



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_P
114	Den	Raphealy	DRAPHEAL	515.127.4561	03-FEB-99	AC_ACCOUNT	11000	

Copying Rows from Another Table

- Write your INSERT statement with a subquery.

```
INSERT INTO sales_reps(id, name, salary, commission_pct)
  SELECT employee_id, last_name, salary, commission_pct
  FROM   employees
  WHERE  job_id LIKE '%REP%';

4 rows created.
```

- Do not use the VALUES clause.
- Match the number of columns in the INSERT clause to those in the subquery.

The UPDATE Statement Syntax

- Modify existing rows with the UPDATE statement.
- Update more than one row at a time, if required.

```
UPDATE employees
SET    department_id = 70
WHERE  employee_id = 113;
1 row updated.
```

```
UPDATE   copy_emp
SET      department_id = 110;
22 rows updated.
```

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

The DELETE Statement

You can remove existing rows from a table by using the DELETE statement.

```
 DELETE FROM departments
 WHERE  department_name = 'Finance';
1 row deleted.
```

```
DELETE FROM  copy_emp;
22 rows deleted.
```

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

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

You cannot delete a row that contains a primary key that is used as a foreign key in another table.

Using Explicit Default Feature

- With the explicit default feature, you can use the DEFAULT keyword as a column value where the column default 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.

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

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

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

Merging Rows

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

```
MERGE INTO copy_emp  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);
```

Database Transactions

A database transaction consists of one of the following:

- DML statements which constitute one consistent change to the data
- One DDL statement
- One DCL statement

Database Transactions

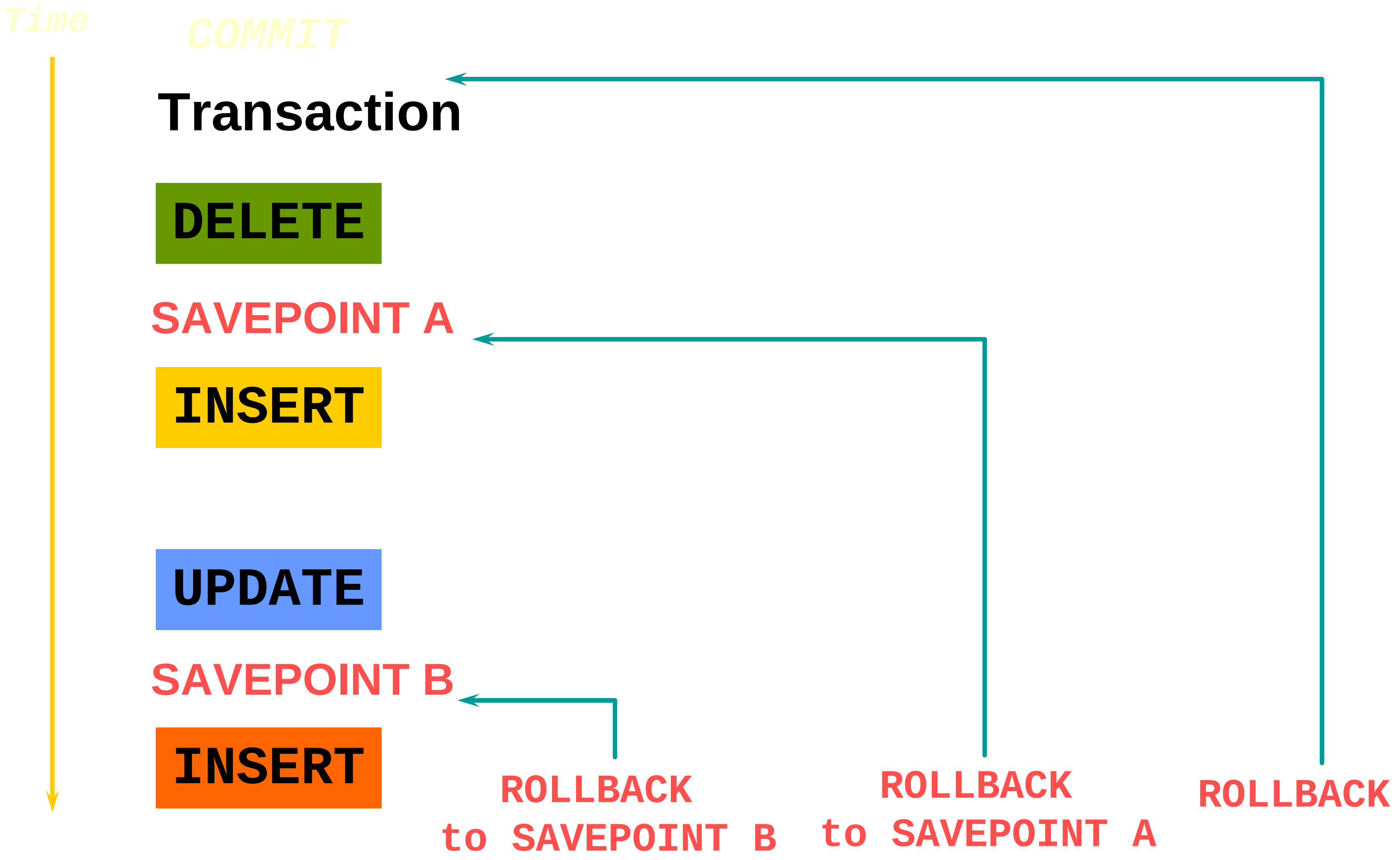
- Begin when the first DML SQL statement is executed
- End with one of the following events:
  - A `COMMIT` or `ROLLBACK` statement is issued
  - A DDL or DCL statement executes (automatic commit)
  - The user exits current SQL session
  - The system crashes

Advantages of COMMIT and ROLLBACK Statements

With COMMIT and ROLLBACK statements, you can:

- Ensure data consistency
- Preview data changes before making changes permanent
- Group logically related operations

Controlling Transactions



Time

COMMIT

Transaction

DELETE

SAVEPOINT A

INSERT

UPDATE

SAVEPOINT B

INSERT

ROLLBACK to SAVEPOINT B

ROLLBACK to SAVEPOINT A

ROLLBACK

Rolling Back Changes to a Marker

- Create a marker in a current transaction by using the SAVEPOINT statement.
- Roll back to that marker by using the ROLLBACK TO SAVEPOINT statement.

```
UPDATE...
SAVEPOINT update_done;
Savepoint created.
INSERT...
ROLLBACK TO update_done;
Rollback complete.
```

Implicit Transaction Processing

- An automatic commit occurs under the following circumstances:
  - DDL statement is issued
  - DCL statement is issued
  - Normal exit from SQL session, without explicitly issuing `COMMIT` or `ROLLBACK` statements
- An automatic rollback occurs under an abnormal termination of SQL session or a system failure.

State of the Data Before COMMIT or ROLLBACK

- The previous state of the data can be recovered.
- The current user can review the results of the DML operations by using the SELECT statement.
- Other users *cannot* view the results of the DML statements by the current user.
- The affected rows are *locked*; other users cannot change the data within the affected rows.

State of the Data after COMMIT

- Data changes are made permanent in the database.
- The previous state of the data is permanently lost.
- All users can view the results.
- Locks on the affected rows are released; those rows are available for other users to manipulate.
- All savepoints are erased.

Committing Data

- Make the changes.

```
DELETE FROM employees
WHERE  employee_id = 99999;
1 row deleted.

INSERT INTO departments
VALUES (290, 'Corporate Tax', NULL, 1700);
1 row inserted.
```

- Commit the changes.

```
COMMIT;
Commit complete.
```

State of the Data After ROLLBACK

Discard all pending changes by using the ROLLBACK statement:

- Data changes are undone.
- Previous state of the data is restored.
- Locks on the affected rows are released.

```
DELETE FROM copy_emp;
22 rows deleted.
ROLLBACK;
Rollback complete.
```

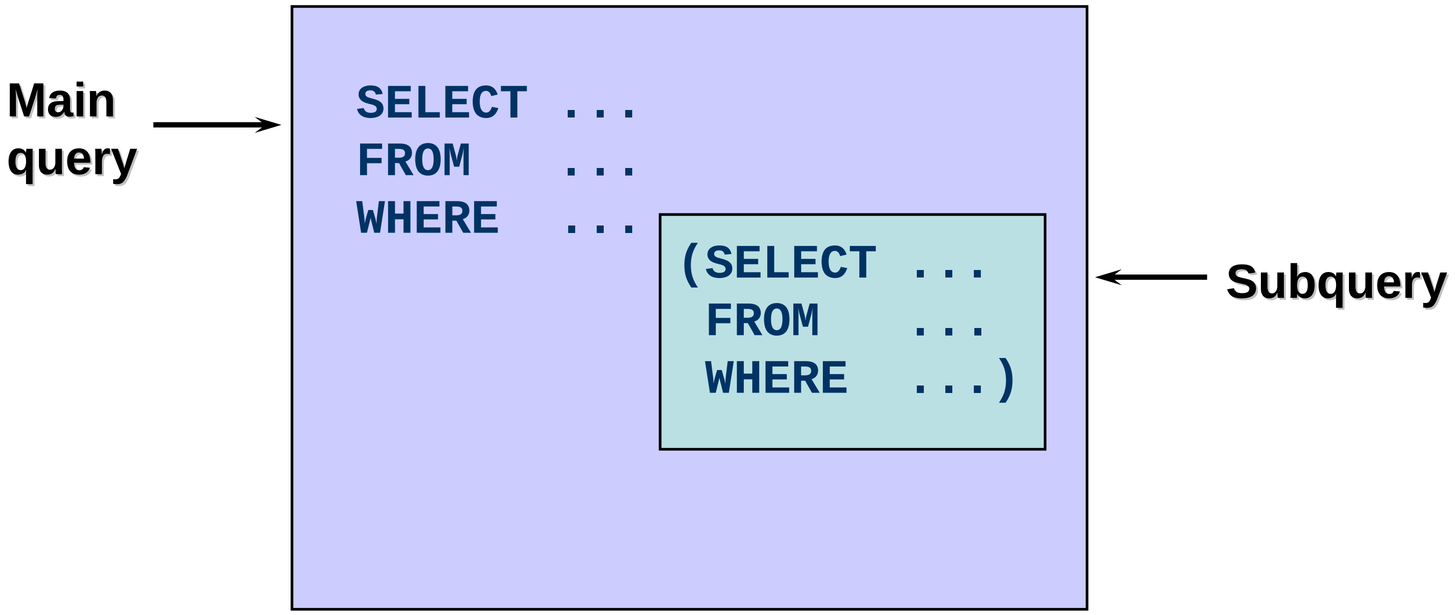
Statement-Level Rollback

- If a single DML statement fails during execution, only that statement is rolled back.
- The Oracle server implements an implicit savepoint.
- All other changes are retained.
- The user should terminate transactions explicitly by executing a `COMMIT` or `ROLLBACK` statement.

Advanced Subqueries

What Is a Subquery?

A subquery is a SELECT statement embedded in a clause of another SQL statement.



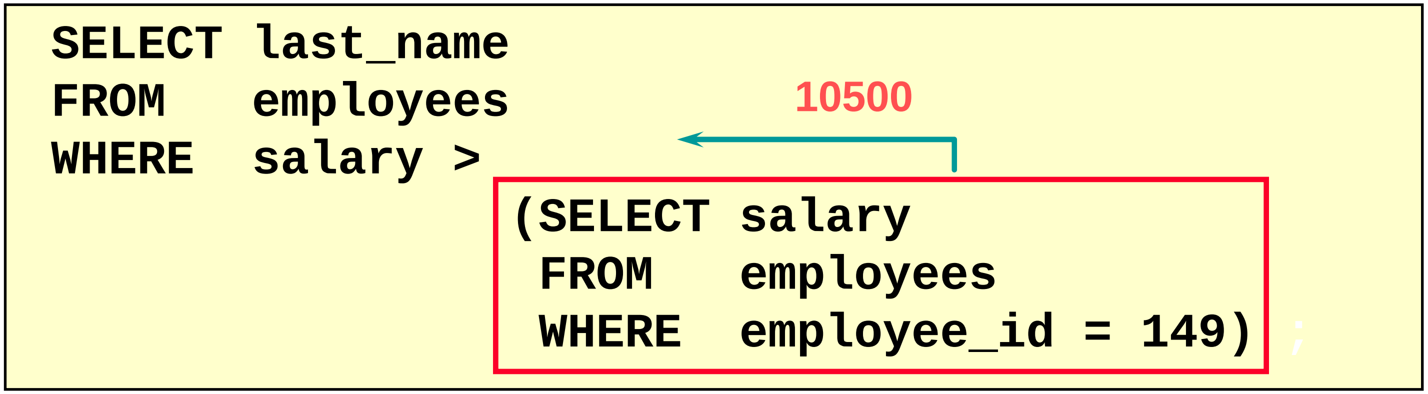
```
SELECT ...
FROM   ...
WHERE  ...
              (SELECT ...
               FROM   ...
               WHERE  ...)
```

Subqueries

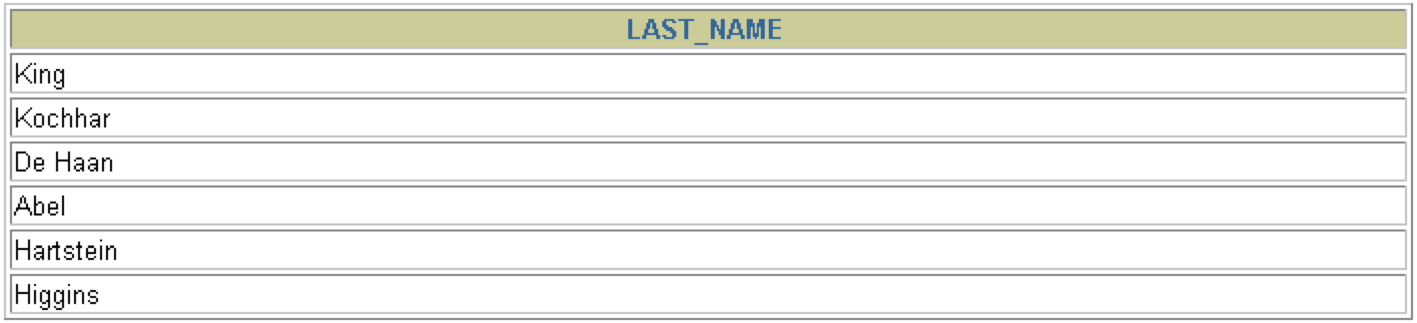
```
SELECT select_list
FROM   table
WHERE  expr operator (SELECT select_list
                        FROM   table);
```

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

Using a Subquery



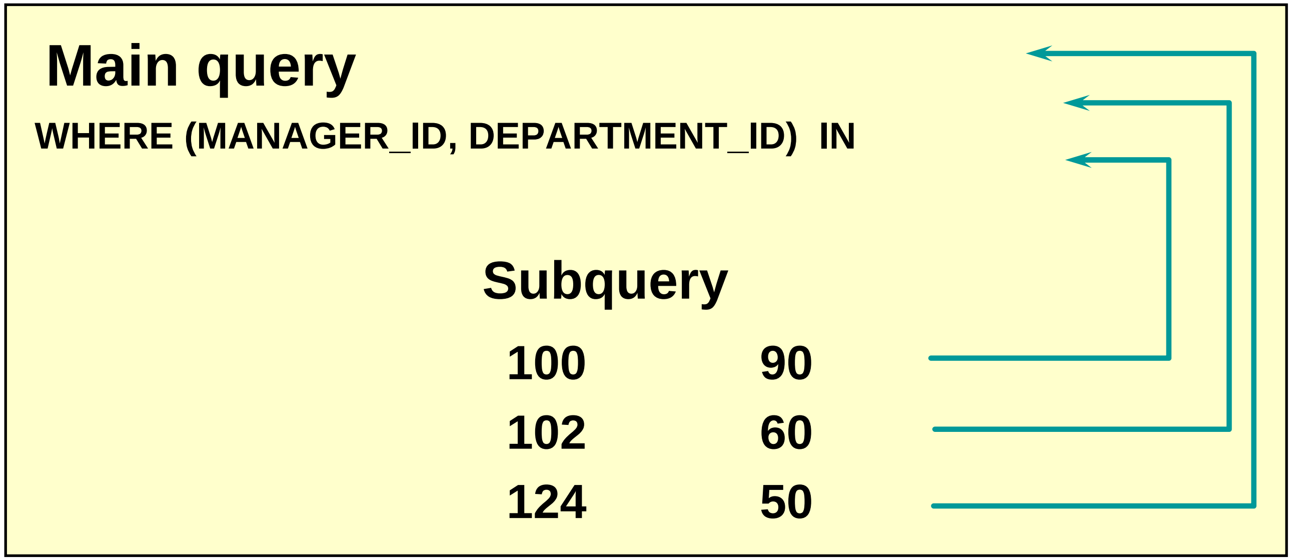
```
SELECT last_name
FROM   employees
WHERE  salary >
                (SELECT salary
                 FROM   employees
                 WHERE  employee_id = 149) ;
```



LAST_NAME
King
Kochhar
De Haan
Abel
Hartstein
Higgins

6 rows selected.

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 178 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 (178,174))
AND    employee_id NOT IN (178,174);
```

Nonpairwise Comparison Subquery

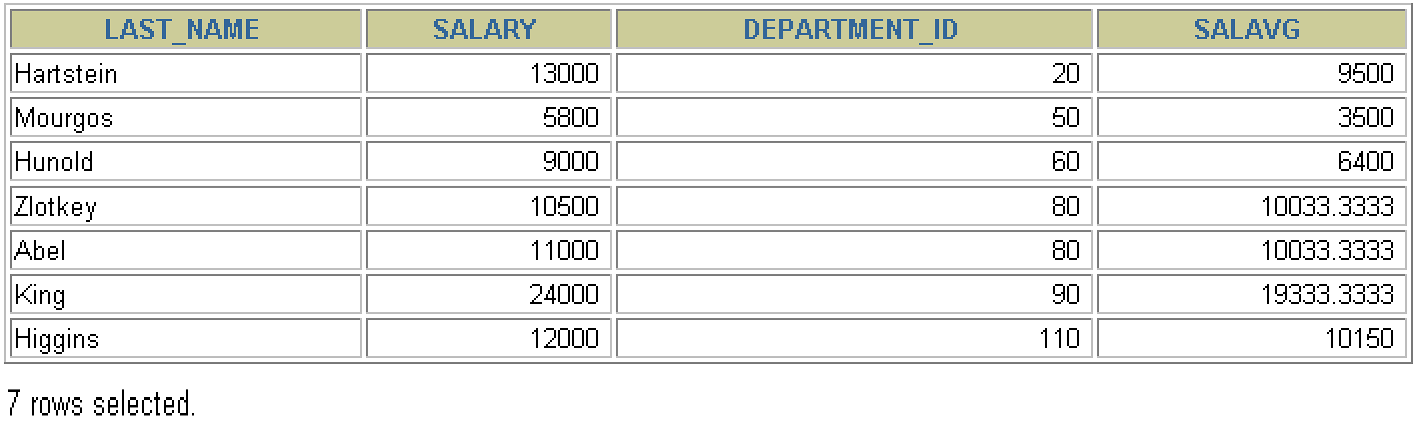
Display the details of the employees who are managed by the same manager as the employees with EMPLOYEE_ID 174 or 141 *and* work in the same department as the employees with EMPLOYEE_ID 174 or 141.

```
SELECT  employee_id, manager_id, department_id
FROM    employees
WHERE   manager_id IN
                  (SELECT  manager_id
                   FROM    employees
                   WHERE   employee_id IN (174,141))
AND     department_id IN
                  (SELECT  department_id
                   FROM    employees
                   WHERE   employee_id IN (174,141))

AND    employee_id NOT IN(174,141);
```

Using a Subquery in the FROM Clause

```
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
Hartstein	13000	20	9500
Mourgos	5800	50	3500
Hunold	9000	60	6400
Zlotkey	10500	80	10033.3333
Abel	11000	80	10033.3333
King	24000	90	19333.3333
Higgins	12000	110	10150

7 rows selected.

Scalar Subquery Expressions

- A scalar subquery expression is a subquery that returns exactly one column value from one row.
- Scalar subqueries were supported in Oracle8*i* only in a limited set of cases, For example:
  - SELECT statement (FROM and WHERE clauses)
  - VALUES list of an INSERT statement
- In Oracle9*i*, 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;
```

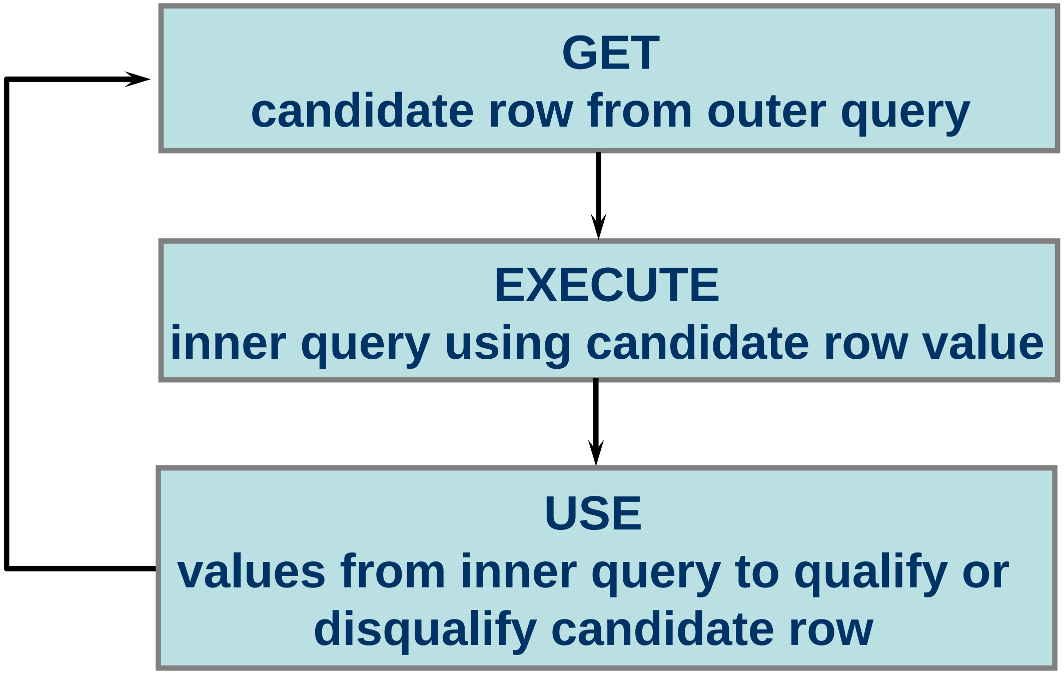
20

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

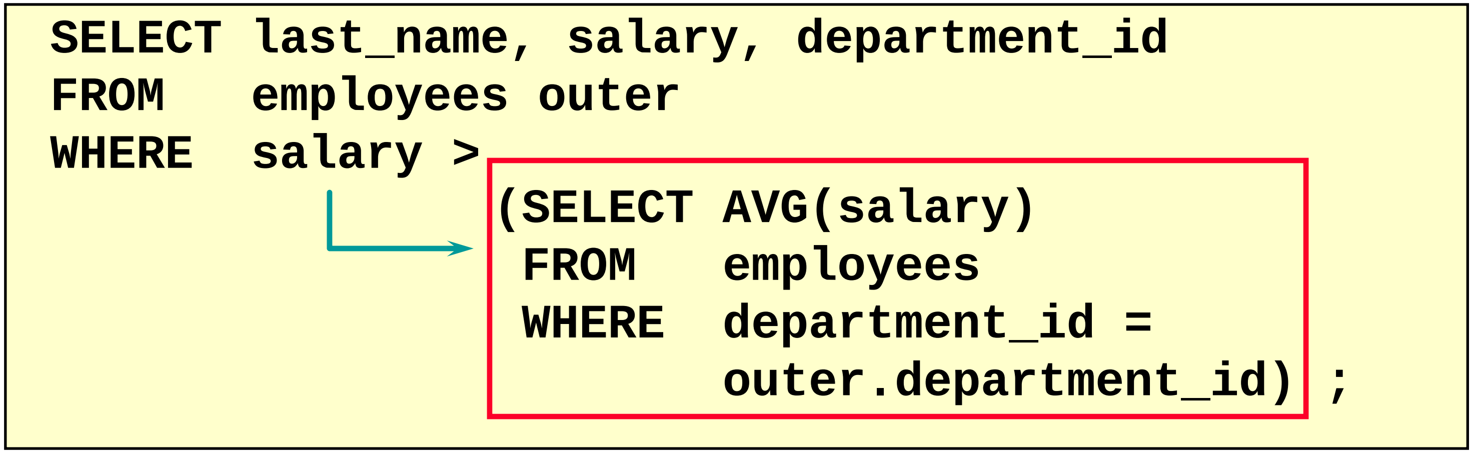
Correlated Subqueries

```
SELECT column1, column2, ...
FROM   table1  outer
WHERE  column1 operator
                     (SELECT  colum1, column2
                      FROM    table2
                      WHERE   expr1 =
                                  outer.expr2);
```

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

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) ;
```

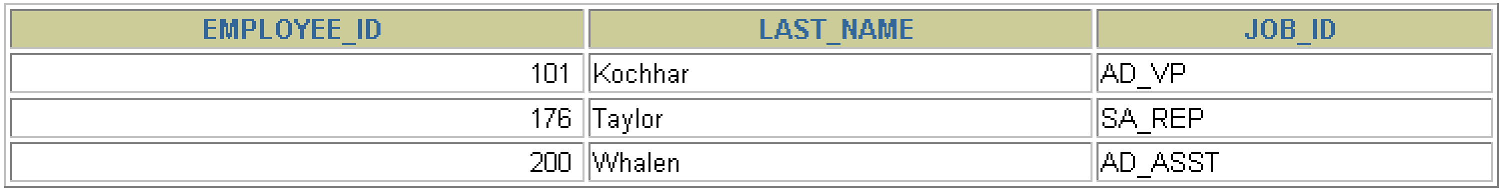
Each time a row from the outer query is processed, the inner query is evaluated.

Birlasoft®

Using Correlated Subqueries

Display details of those employees who have switched 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

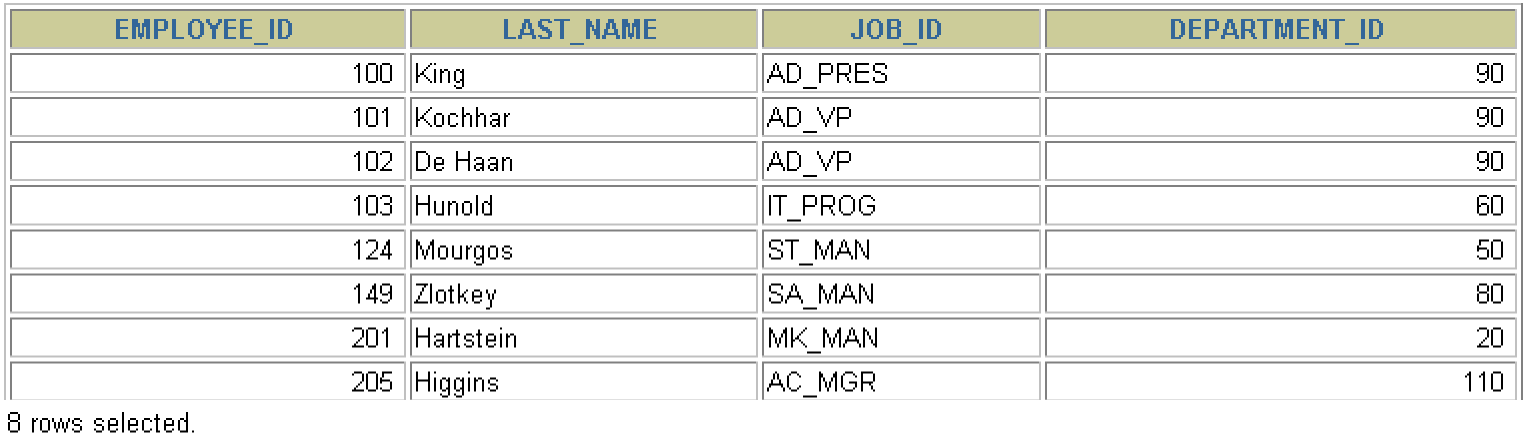
Using the EXISTS Operator

- 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

Using the EXISTS Operator

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);
```



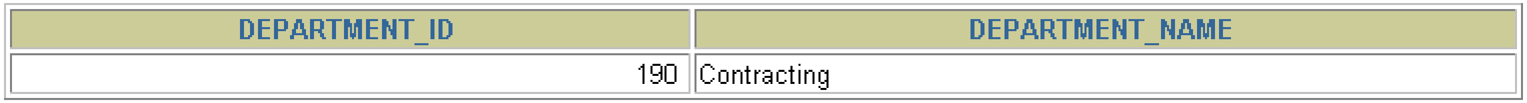
EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
100	King	AD_PRES	90
101	Kochhar	AD_VP	90
102	De Haan	AD_VP	90
103	Hunold	IT_PROG	60
124	Mourgos	ST_MAN	50
149	Zlotkey	SA_MAN	80
201	Hartstein	MK_MAN	20
205	Higgins	AC_MGR	110

8 rows selected.

Using the NOT EXISTS Operator

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
190	Contracting

Correlated UPDATE

```
UPDATE table1 alias1
SET    column = (SELECT expression
                 FROM   table2 alias2
                 WHERE  alias1.column =
                        alias2.column);
```

Use a correlated subquery to update rows in one table based on rows from another table.

Correlated UPDATE

- Denormalize the EMPLOYEES table by adding a column to store the department name.
- Populate the table by using a correlated update.

```
ALTER TABLE employees
ADD(department_name VARCHAR2(14));
```

```
UPDATE employees e
SET    department_name =
              (SELECT department_name
               FROM   departments d
               WHERE  e.department_id = d.department_id);
```

Correlated DELETE

```
DELETE FROM table1 alias1
WHERE  column operator
             (SELECT expression
              FROM   table2 alias2
              WHERE  alias1.column = alias2.column);
```

Use a correlated subquery to delete rows in one table based on rows from another table.

Correlated DELETE

Use a correlated subquery to delete only those rows from the EMPLOYEES table that also exist in the EMP_HISTORY table.

```
DELETE FROM employees E
WHERE employee_id =
           (SELECT employee_id
            FROM   emp_history
            WHERE  employee_id = E.employee_id);
```

Thank You!