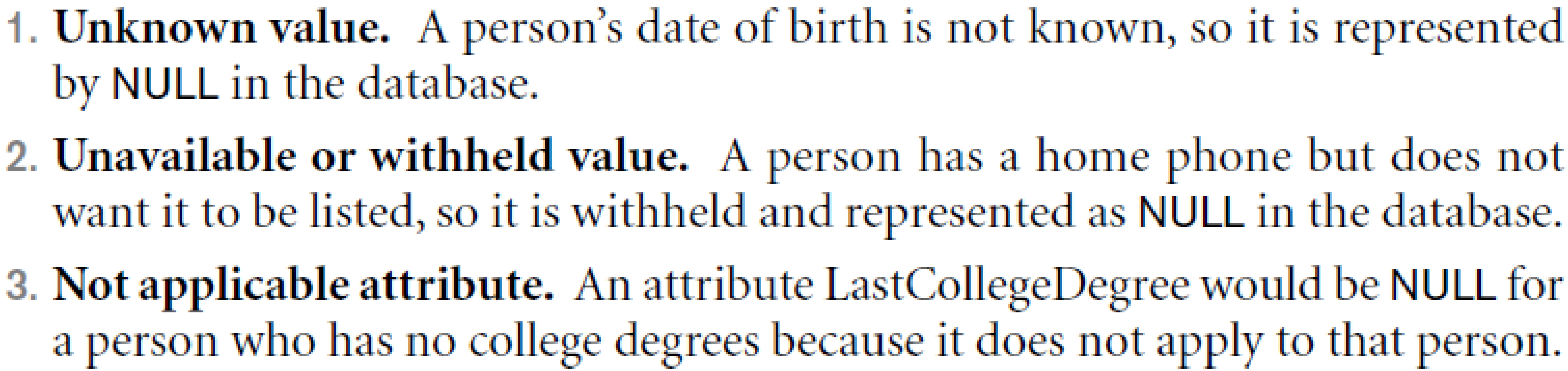
Advanced SQL

Comparisons involving NULL and three valued logic

- Meanings of NULL



1. **Unknown value.** A person's date of birth is not known, so it is represented by NULL in the database.
2. **Unavailable or withheld value.** A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.
3. **Not applicable attribute.** An attribute LastCollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

Comparisons involving NULL and three valued logic

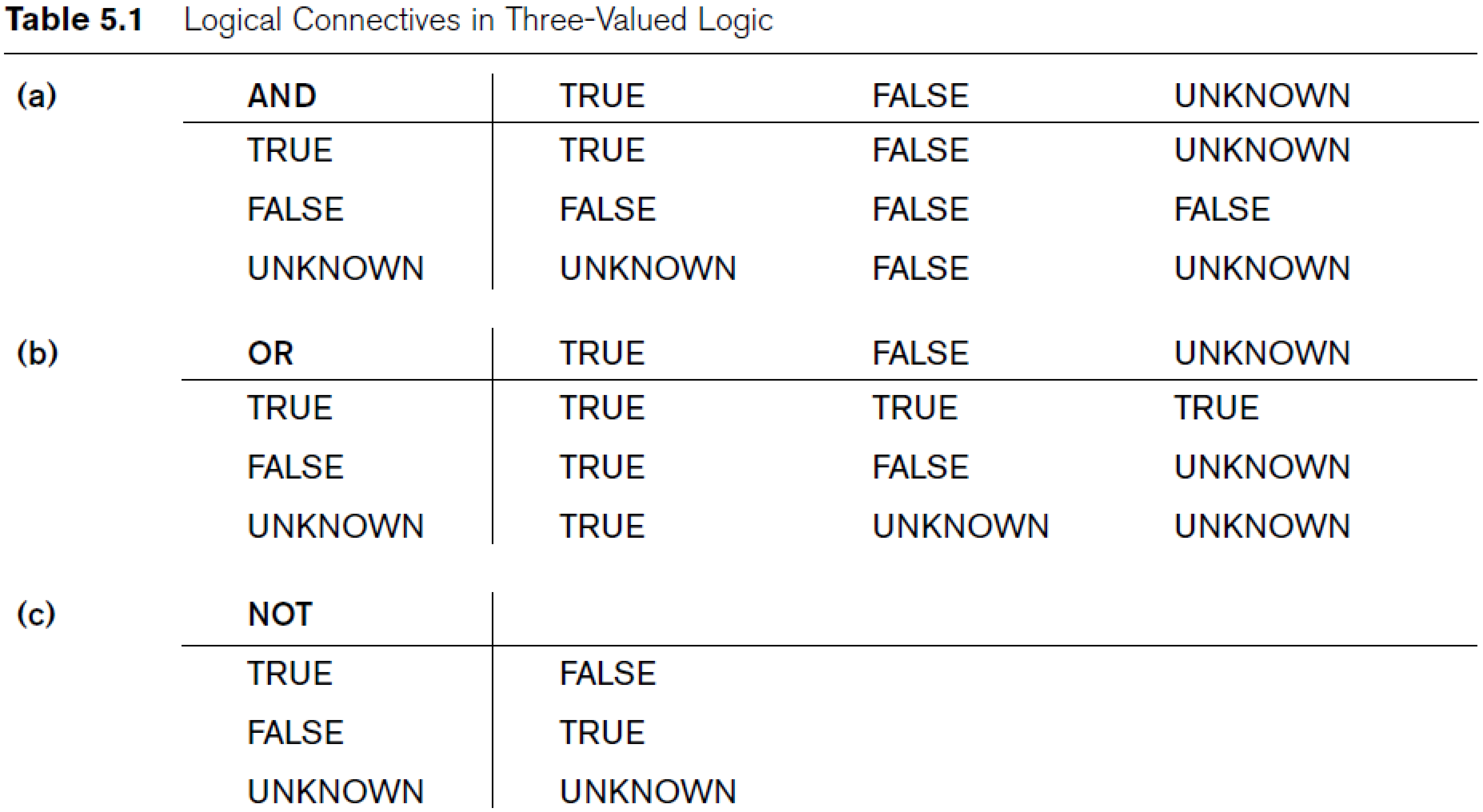


Table 5.1 Logical Connectives in Three-Valued Logic

(a)

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

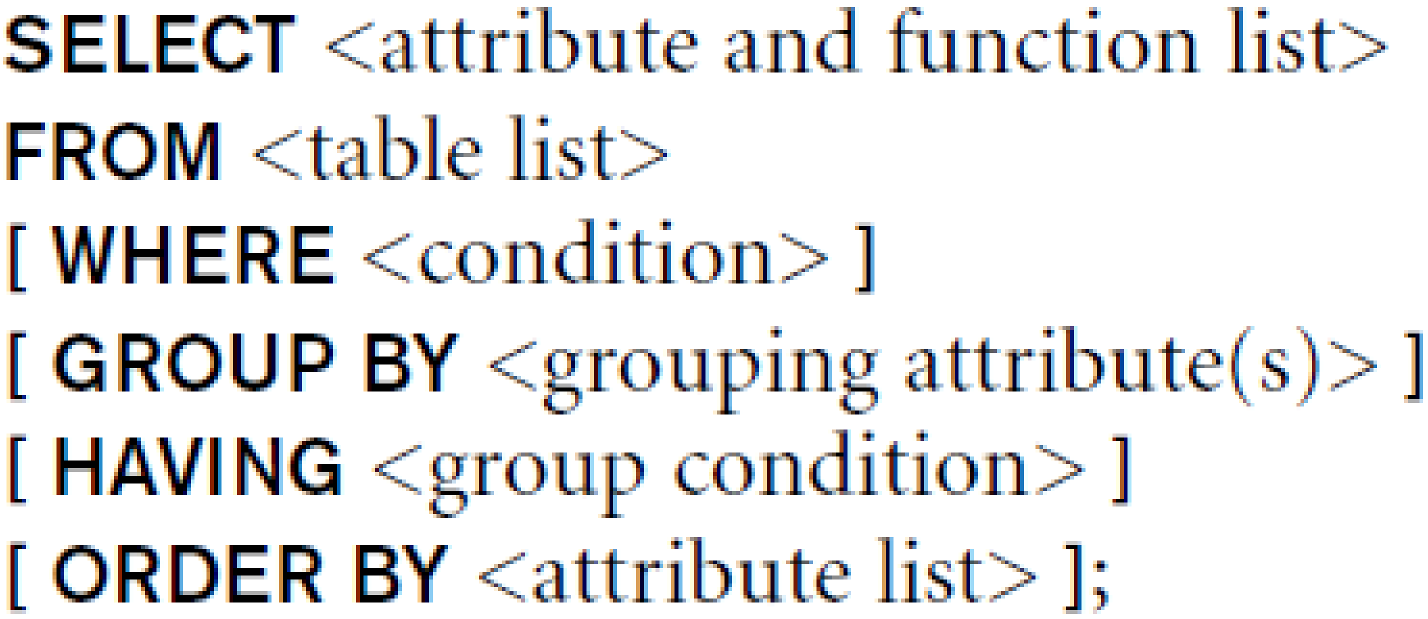
(b)

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

General Template of Queries

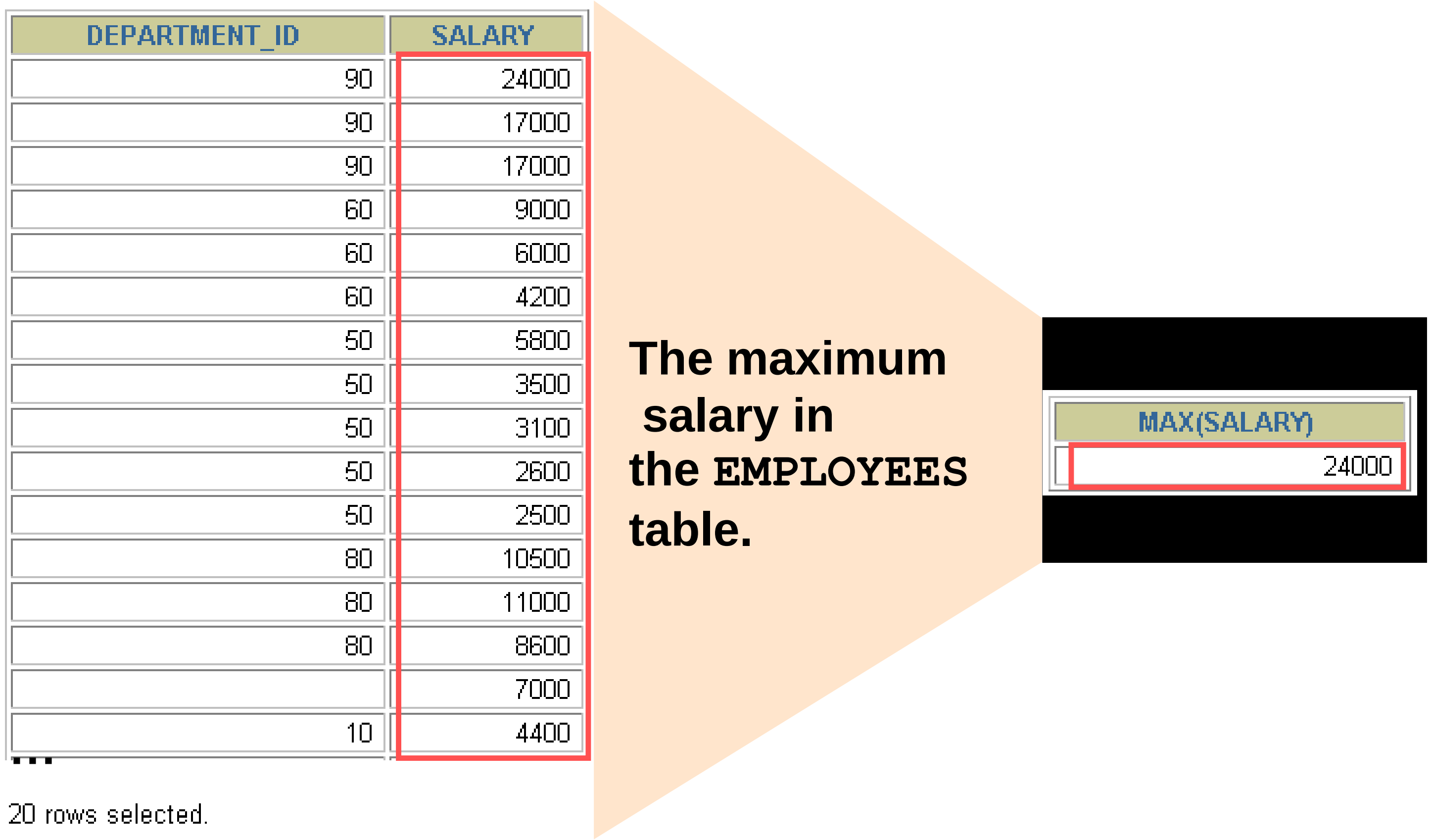


```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
```

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

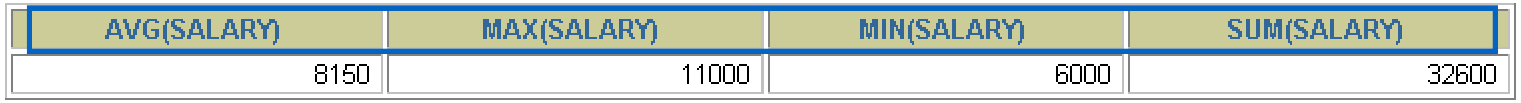
Group Functions Syntax

```
SELECT        [column,] group_function(column), ...
FROM          table
[WHERE        condition]
[GROUP BY     column]
[ORDER BY     column];
```

Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

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

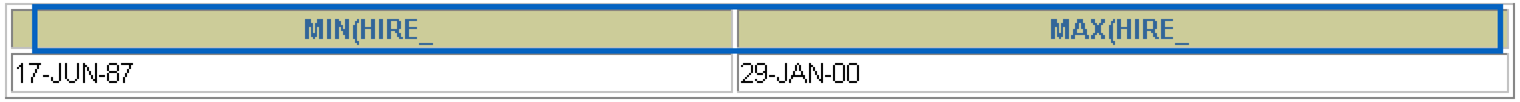


AVG(SALARY)	MAX(SALARY)	MIN(SALARY)	SUM(SALARY)
8150	11000	6000	32600

Using the MIN and MAX Functions

You can use MIN and MAX for any data type.

```
SELECT MIN(hire_date), MAX(hire_date)
FROM   employees;
```



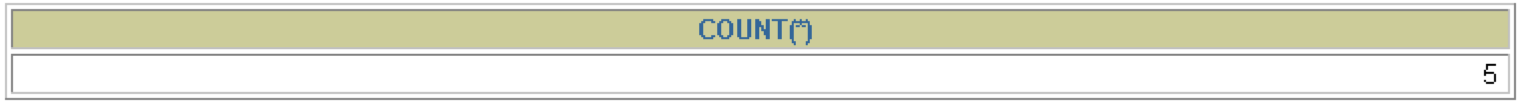
MIN(HIRE_	MAX(HIRE_
17-JUN-87	29-JAN-00

```
SELECT MIN(last_name), MAX(last_name)
FROM   employees;
```

Using the COUNT Function

COUNT(*) returns the number of rows in a table.

```
SELECT COUNT(*)
FROM   employees
WHERE  department_id = 50;
```

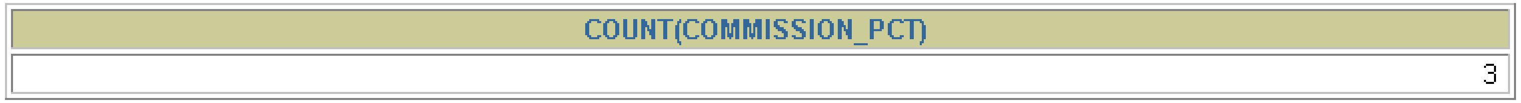


```
COUNT(*)
       5
```

Using the COUNT Function

- COUNT(*expr*) returns the number of rows with non-null values for the *expr*.
- Display the number of department values in the EMPLOYEES table, excluding the null values.

```
SELECT COUNT(commission_pct)
FROM   employees
WHERE  department_id = 80;
```

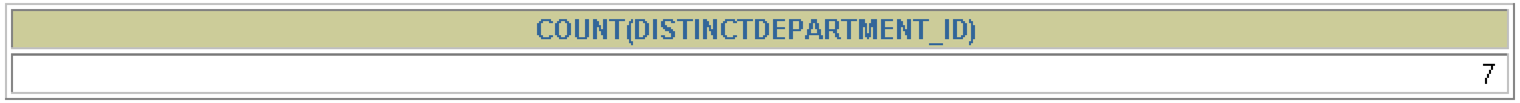


COUNT(COMMISSION_PCT)
3

Using the DISTINCT Keyword

- COUNT(DISTINCT expr) returns the number of distinct non-null values of the *expr*.
- Display the number of distinct department values in the EMPLOYEES table.

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

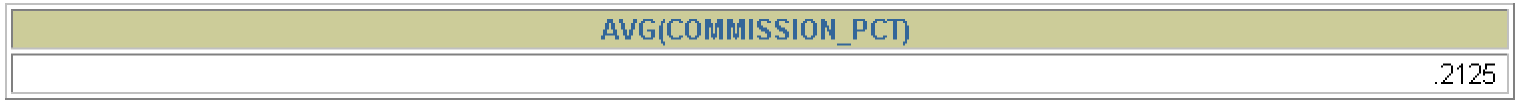


COUNT(DISTINCTDEPARTMENT_ID)
7

Group Functions and Null Values

Group functions ignore null values in the column.

```
SELECT AVG(commission_pct)
FROM   employees;
```

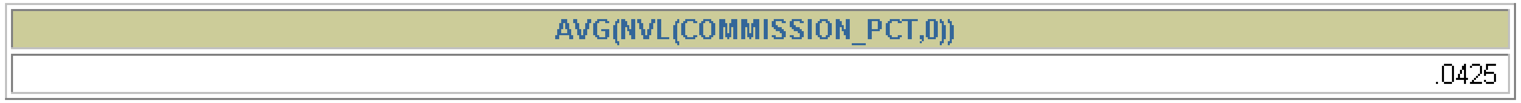


```
AVG(COMMISSION_PCT)
              .2125
```

Using the NVL Function with Group Functions

The NVL function forces group functions to include null values.

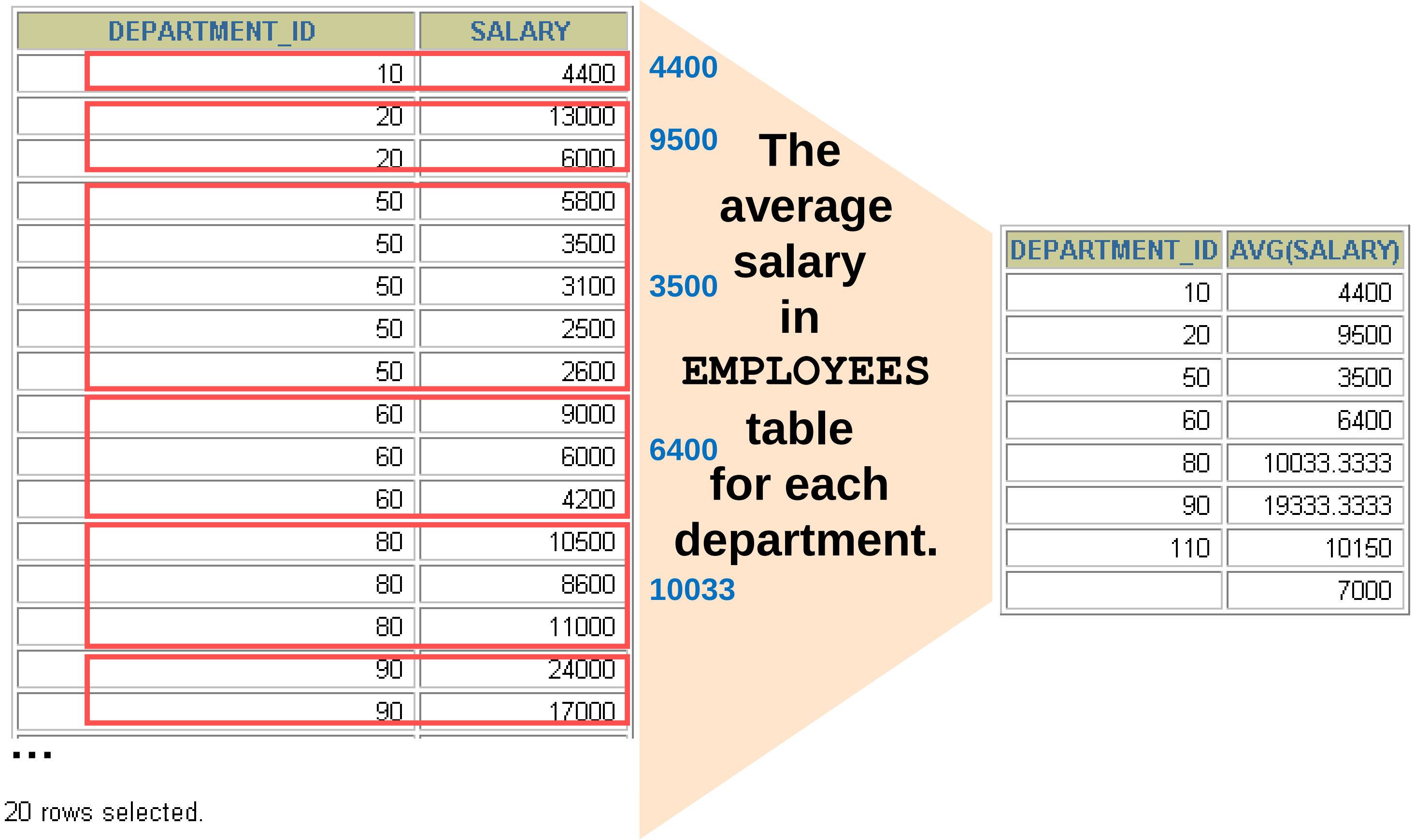
```
SELECT AVG(NVL(commission_pct, 0))
FROM   employees;
```



AVG(NVL(COMMISSION_PCT,0))
.0425

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

Creating Groups of Data: The GROUP BY Clause Syntax

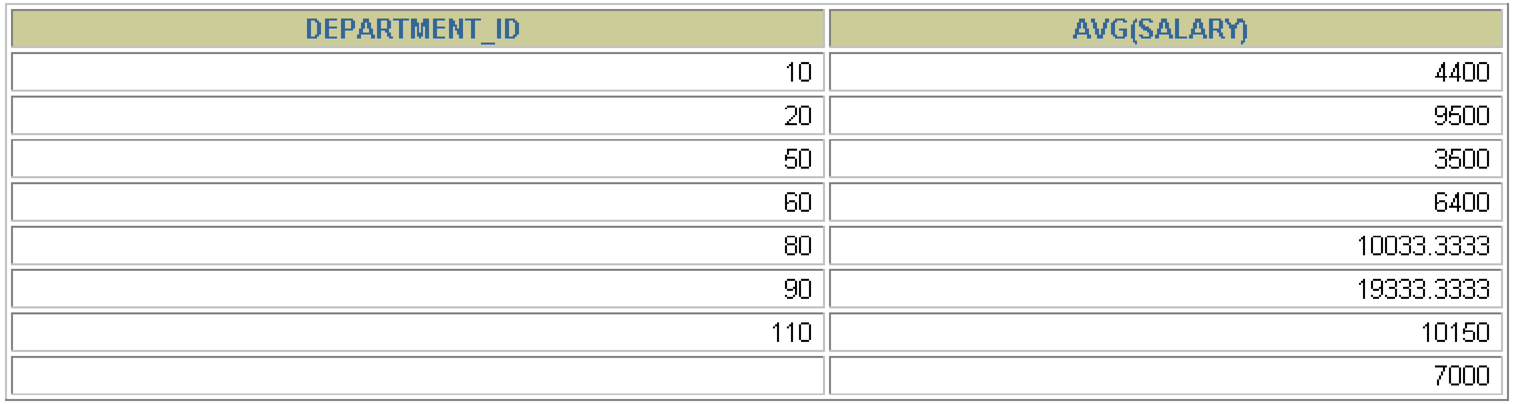
```
SELECT         column, group_function(column)
FROM           table
[WHERE         condition]
[GROUP BY      group_by_expression]
[ORDER BY      column];
```

Divide rows in a table into smaller groups by using the GROUP BY clause.

Using the GROUP BY Clause

All columns in the SELECT list that are not in group functions must be in the GROUP BY clause.

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



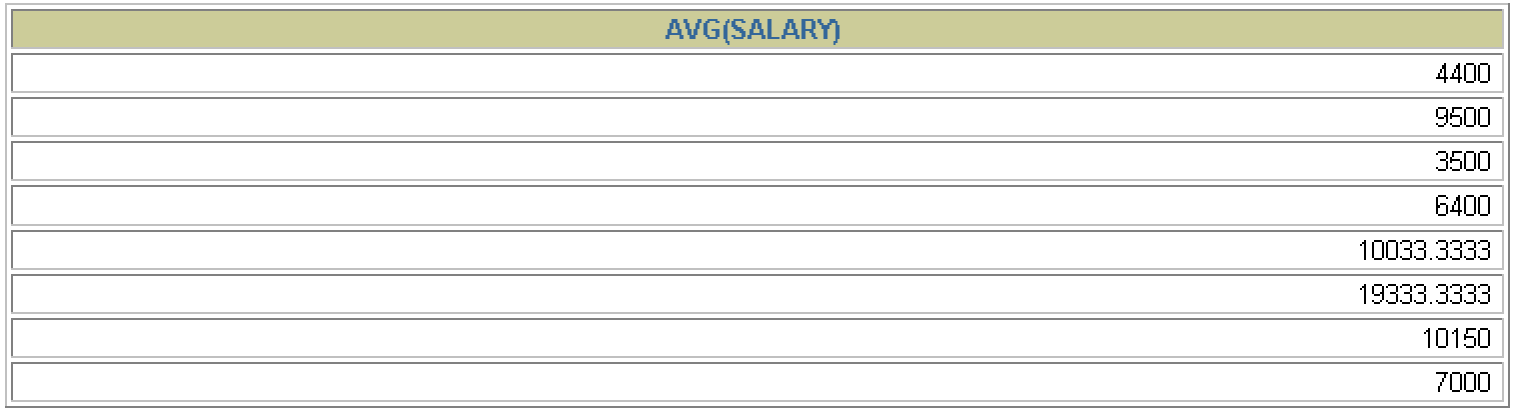
DEPARTMENT_ID	AVG(SALARY)
10	4400
20	9500
50	3500
60	6400
80	10033.3333
90	19333.3333
110	10150
	7000

8 rows selected.

Using the GROUP BY Clause

The GROUP BY column does not have to be in the SELECT list.

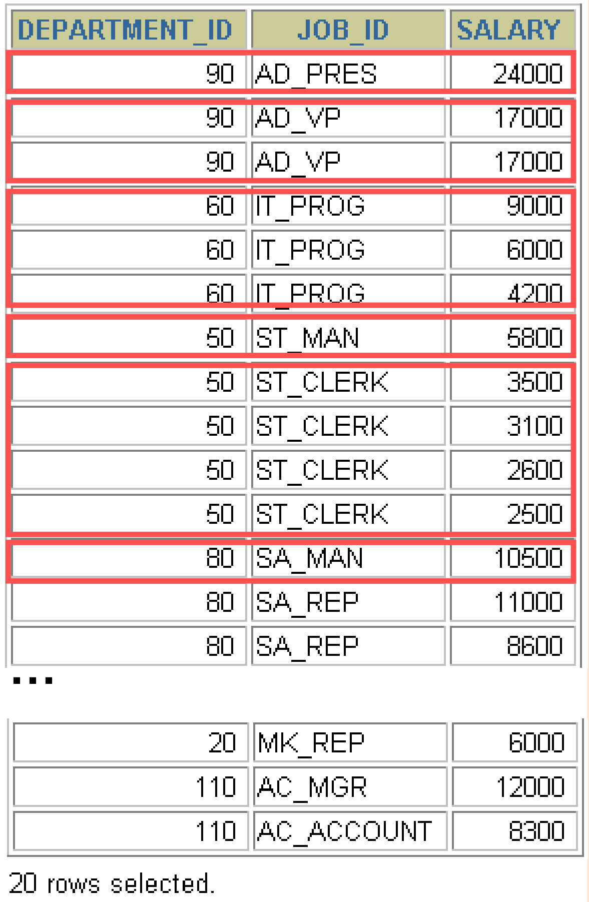
```
SELECT   AVG(salary)
FROM     employees
GROUP BY department_id ;
```



AVG(SALARY)
4400
9500
3500
6400
10033.3333
19333.3333
10150
7000

Grouping by More Than One Column

EMPLOYEES



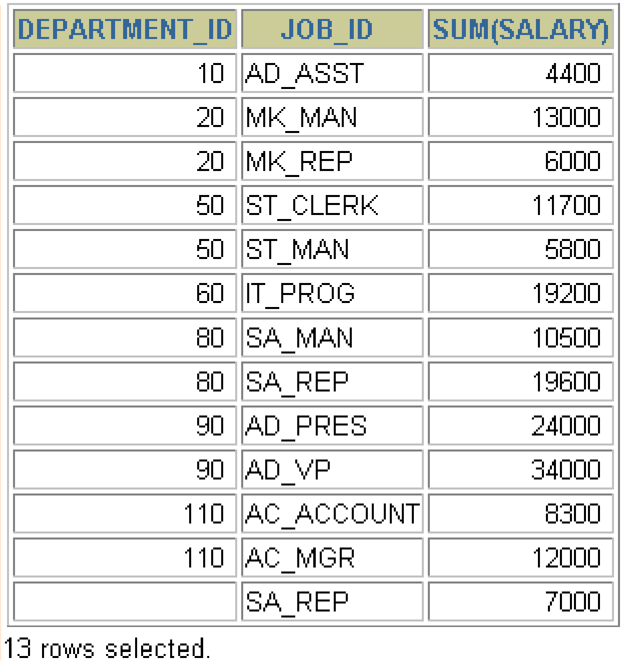
DEPARTMENT_ID	JOB_ID	SALARY
90	AD_PRES	24000
90	AD_VP	17000
90	AD_VP	17000
60	IT_PROG	9000
60	IT_PROG	6000
60	IT_PROG	4200
50	ST_MAN	5800
50	ST_CLERK	3500
50	ST_CLERK	3100
50	ST_CLERK	2600
50	ST_CLERK	2500
80	SA_MAN	10500
80	SA_REP	11000
80	SA_REP	8600

...

20	MK_REP	6000
110	AC_MGR	12000
110	AC_ACCOUNT	8300

20 rows selected.

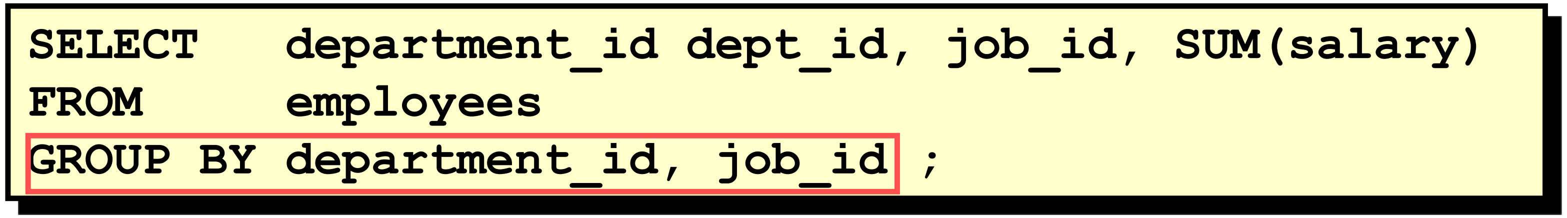
"Add up the salaries in the EMPLOYEES table for each job, grouped by department.



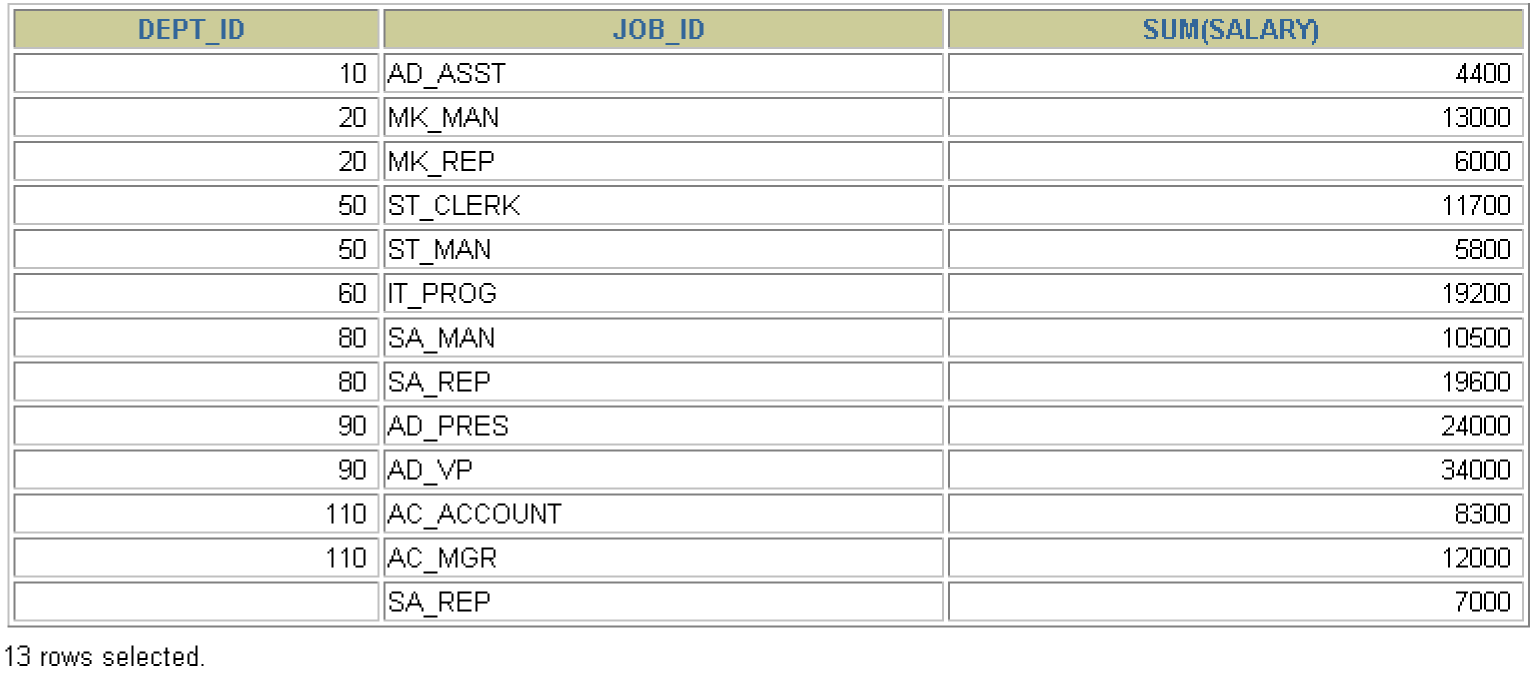
DEPARTMENT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

Using the GROUP BY Clause on Multiple Columns



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



DEPT_ID	JOB_ID	SUM(SALARY)
10	AD_ASST	4400
20	MK_MAN	13000
20	MK_REP	6000
50	ST_CLERK	11700
50	ST_MAN	5800
60	IT_PROG	19200
80	SA_MAN	10500
80	SA_REP	19600
90	AD_PRES	24000
90	AD_VP	34000
110	AC_ACCOUNT	8300
110	AC_MGR	12000
	SA_REP	7000

13 rows selected.

Illegal Queries
Using Group Functions

Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause.

```
SELECT department_id, COUNT(last_name)
FROM   employees;
```

```
SELECT department_id, COUNT(last_name)
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

Column missing in the GROUP BY clause

Illegal Queries
Using Group Functions

- 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.

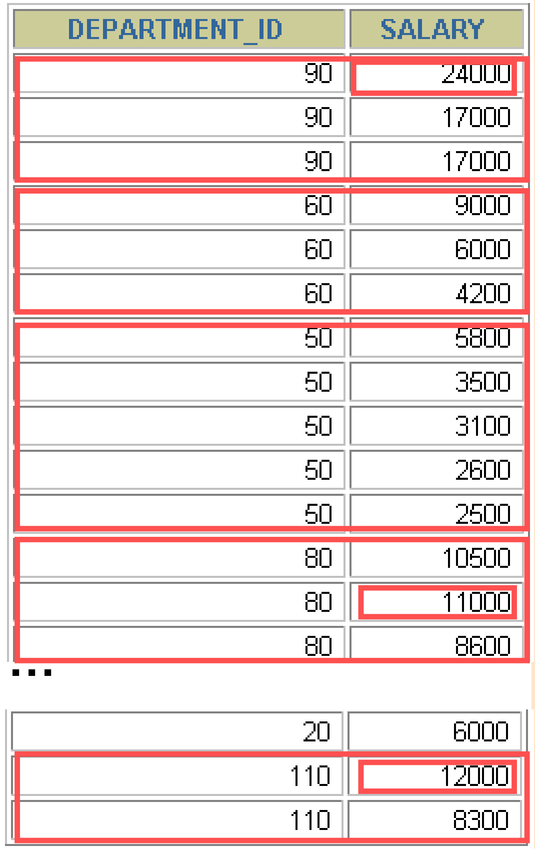
```
SELECT   department_id, AVG(salary)
FROM     employees
WHERE    AVG(salary) > 8000
GROUP BY department_id;
```

```
WHERE  AVG(salary) > 8000
       *
ERROR at line 3:
ORA-00934: group function is not allowed here
```

Can’t use the WHERE clause to restrict groups

Excluding Group Results

EMPLOYEES



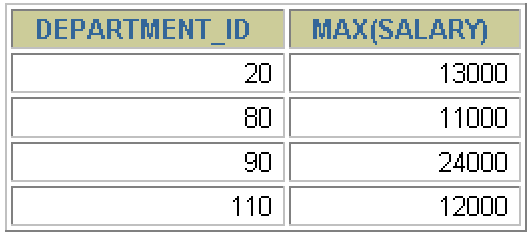
DEPARTMENT_ID	SALARY
90	24000
90	17000
90	17000
60	9000
60	6000
60	4200
50	5800
50	3500
50	3100
50	2600
50	2500
80	10500
80	11000
80	8600

…

20	6000
110	12000
110	8300

20 rows selected.

The maximum salary per department when it is greater than $10,000



DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

Excluding Group Results: The HAVING Clause

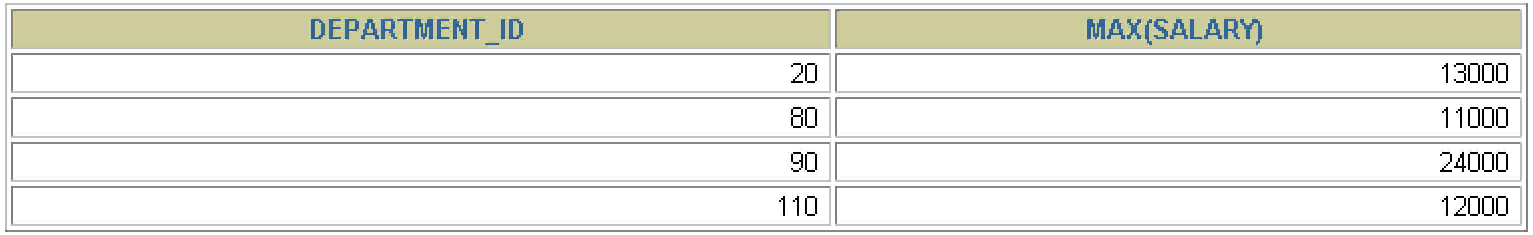
Use the HAVING clause to restrict groups:

1. Rows are grouped.
2. The group function is applied.
3. Groups matching the HAVING clause are displayed.

```
SELECT        column, group_function
FROM          table
[WHERE        condition]
[GROUP BY     group_by_expression]
[HAVING       group_condition]
[ORDER BY     column];
```

Using the HAVING Clause

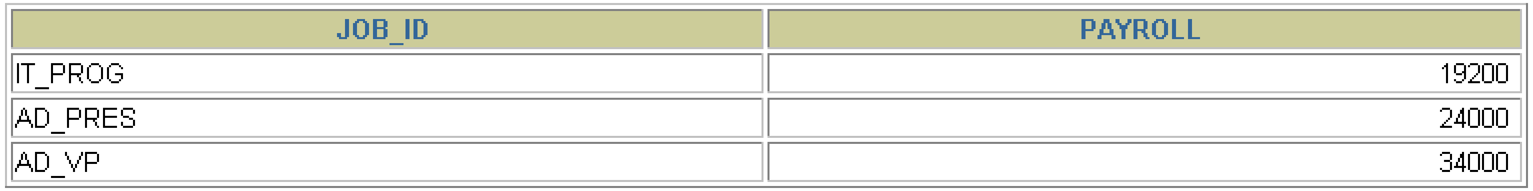
```
SELECT   department_id, MAX(salary)
FROM     employees
GROUP BY department_id
HAVING   MAX(salary)>10000 ;
```



DEPARTMENT_ID	MAX(SALARY)
20	13000
80	11000
90	24000
110	12000

Using the HAVING Clause

```
SELECT   job_id, SUM(salary) PAYROLL
FROM     employees
WHERE    job_id NOT LIKE '%REP%'
GROUP BY job_id
HAVING   SUM(salary) > 13000
ORDER BY SUM(salary);
```

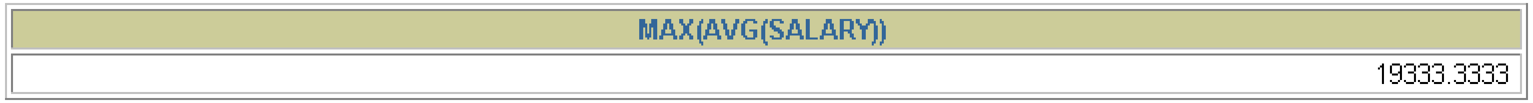


JOB_ID	PAYROLL
IT_PROG	19200
AD_PRES	24000
AD_VP	34000

Nesting Group Functions

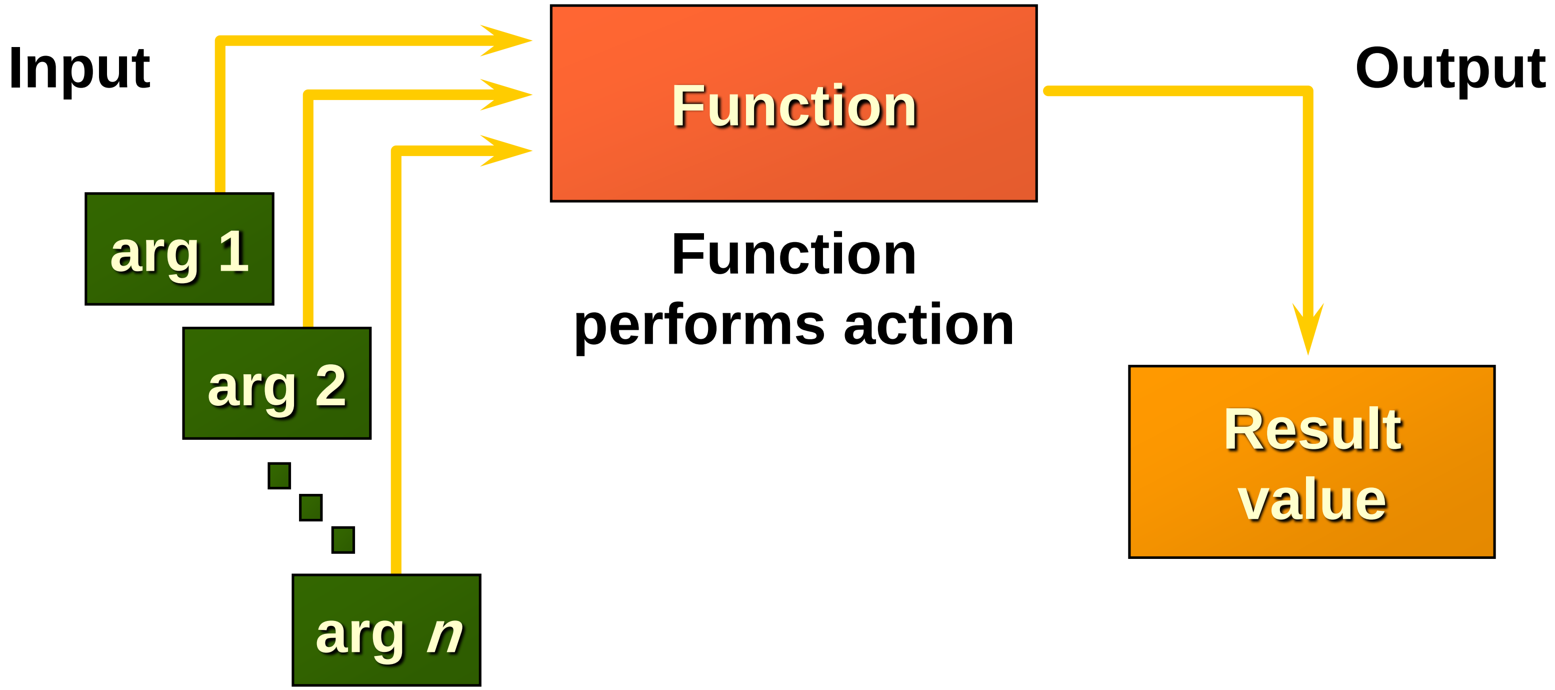
Display the maximum average salary.

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



MAX(AVG(SALARY))
19333.3333

SQL Functions



Input

arg 1

arg 2

arg *n*

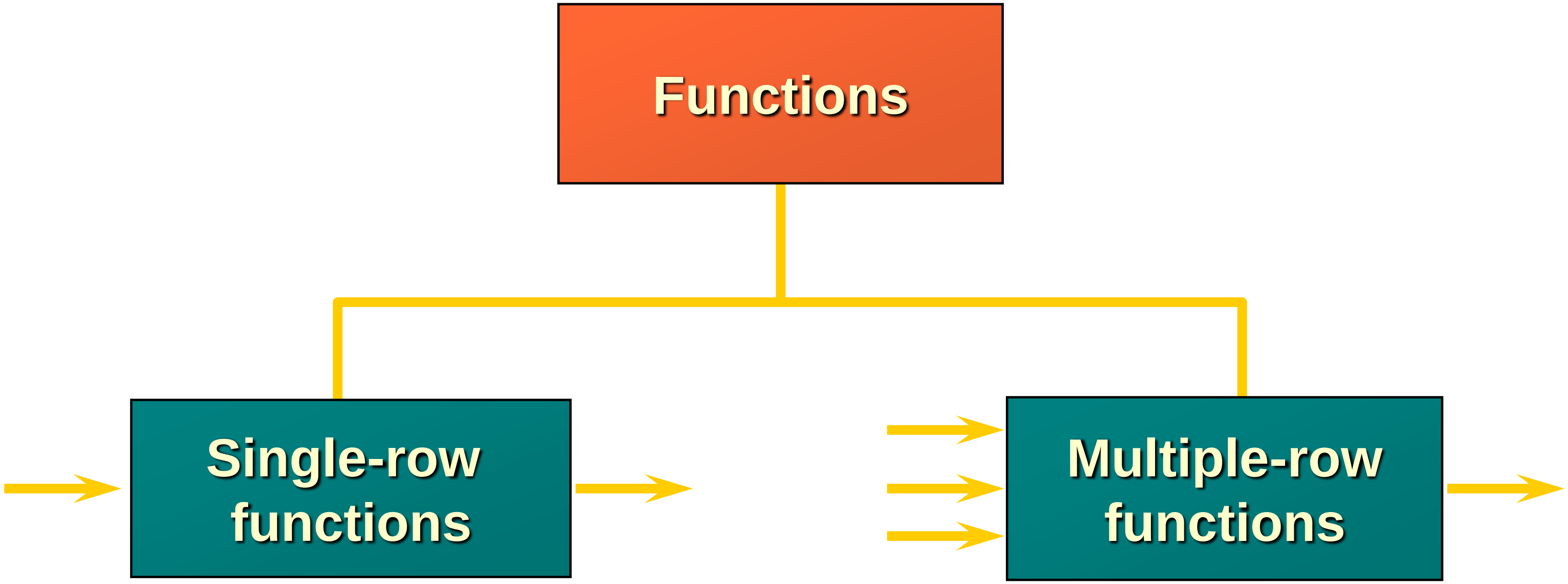
Function

Function performs action

Output

Result value

Two Types of SQL Functions



Functions

Single-row functions

Multiple-row functions

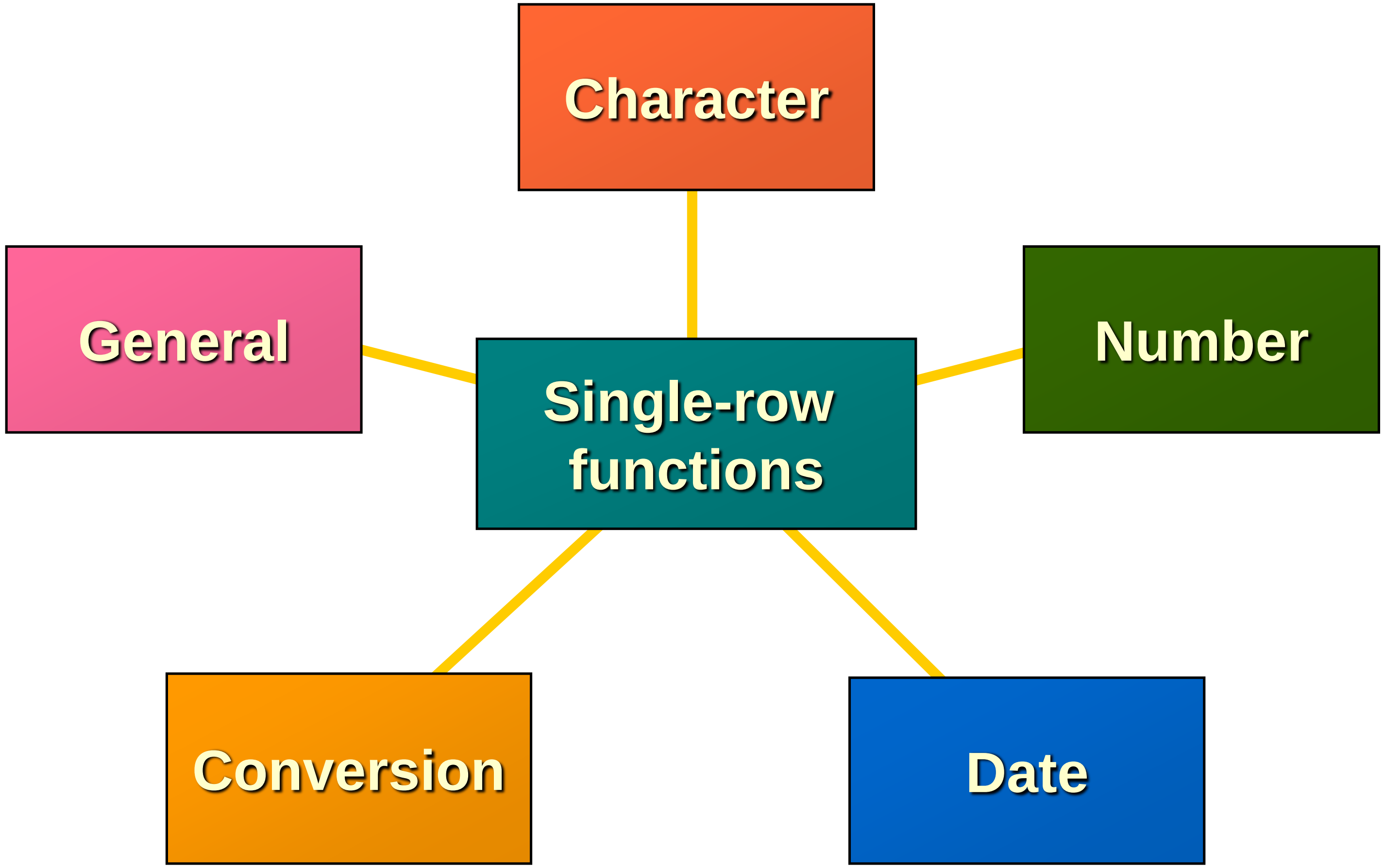
Single-Row Functions

Single row functions:

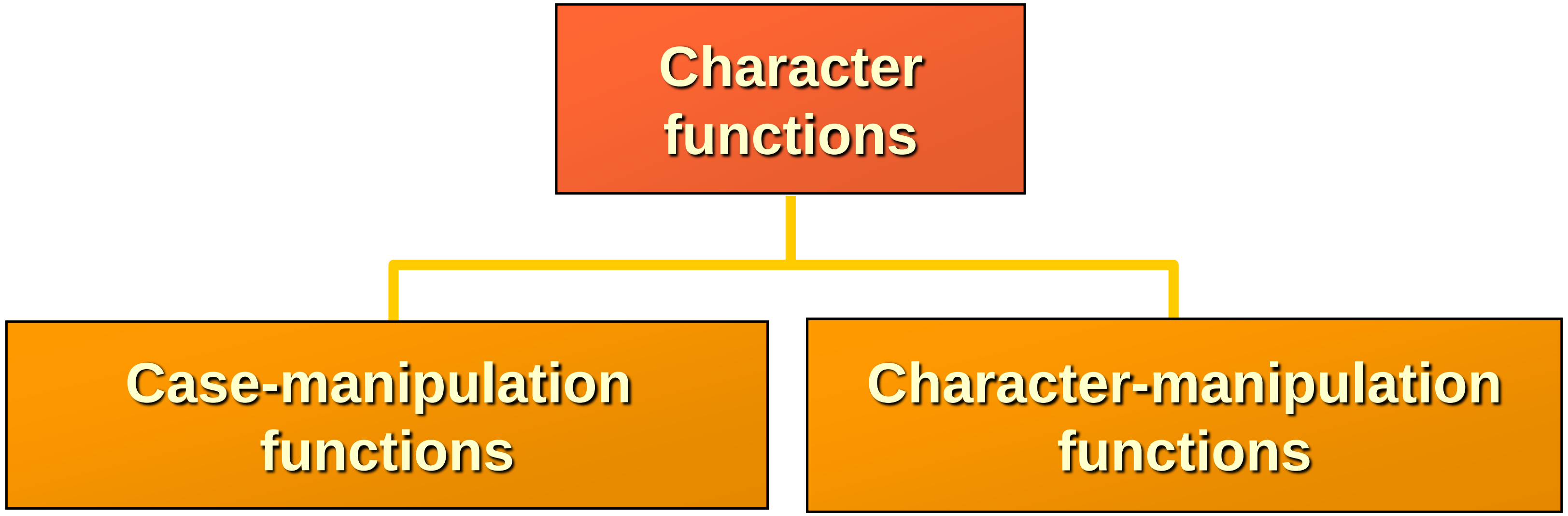
- Manipulate data items
- Accept arguments and return one value
- Act on each row returned
- Return one result per row
- May modify the data type
- Can be nested
- Accept arguments which can be a column or an expression

```
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.

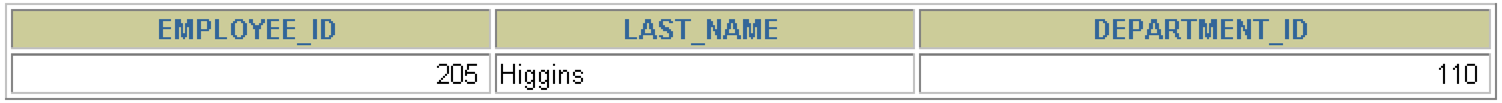
Function	Result
`LOWER('SQL Course')`	`sql course`
`UPPER('SQL Course')`	`SQL COURSE`
`INITCAP('SQL Course')`	`Sql Course`

Using Case Manipulation Functions

Display the employee number, name, and department number for employee Higgins:

```
SELECT employee_id, last_name, department_id
FROM   employees
WHERE  last_name = 'higgins';
no rows selected
```

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



EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
205	Higgins	110

Character functions

```
SQL> SELECT UPPER(name)
  2  FROM first_pay;

UPPER(NAME)
--------------------
LINDA COSTA
JOHN DAVIDSON
SUSAN ASH
STEPHEN YORK
RICHARD JONES
JOANNE BROWN
```

Originally the data was in mixed case, the UPPER function converts it to UPPER case for this display.

Listing of the table after the UPPER function shows the data unaffected.

```
SQL> SELECT *
    FROM first_pay;

PAY_ NAME                 JO STARTDATE    SALARY     BONUS
---- -------------------- -- --------- --------- ---------
1111 Linda Costa          CI 15-JAN-97     45000      1000
2222 John Davidson        IN 25-SEP-92     40000      1500
3333 Susan Ash            AP 05-FEB-00     25000       500
4444 Stephen York         CM 03-JUL-97     42000      2000
5555 Richard Jones        CI 30-OCT-92     50000      2000
6666 Joanne Brown         IN 18-AUG-94     48000      2000
```

Character functions

```
SQL> SELECT LOWER(name), LOWER(jobcode)
  2  FROM first_pay;

LOWER(NAME)          LO
-------------------- --
linda costa          ci
john davidson        in
susan ash            ap
stephen york         cm
richard jones        ci
joanne brown         in
```

The LOWER function converts fields to lower case for display. NAME was originally in mixed case and jobcode was originally in upper case.

Character functions

```
SQL> SELECT INITCAP(startdate)
  2  FROM first_pay;

INITCAP(STARTDATE)
-----------------------------------
15-Jan-97
25-Sep-92
05-Feb-00
03-Jul-97
30-Oct-92
```

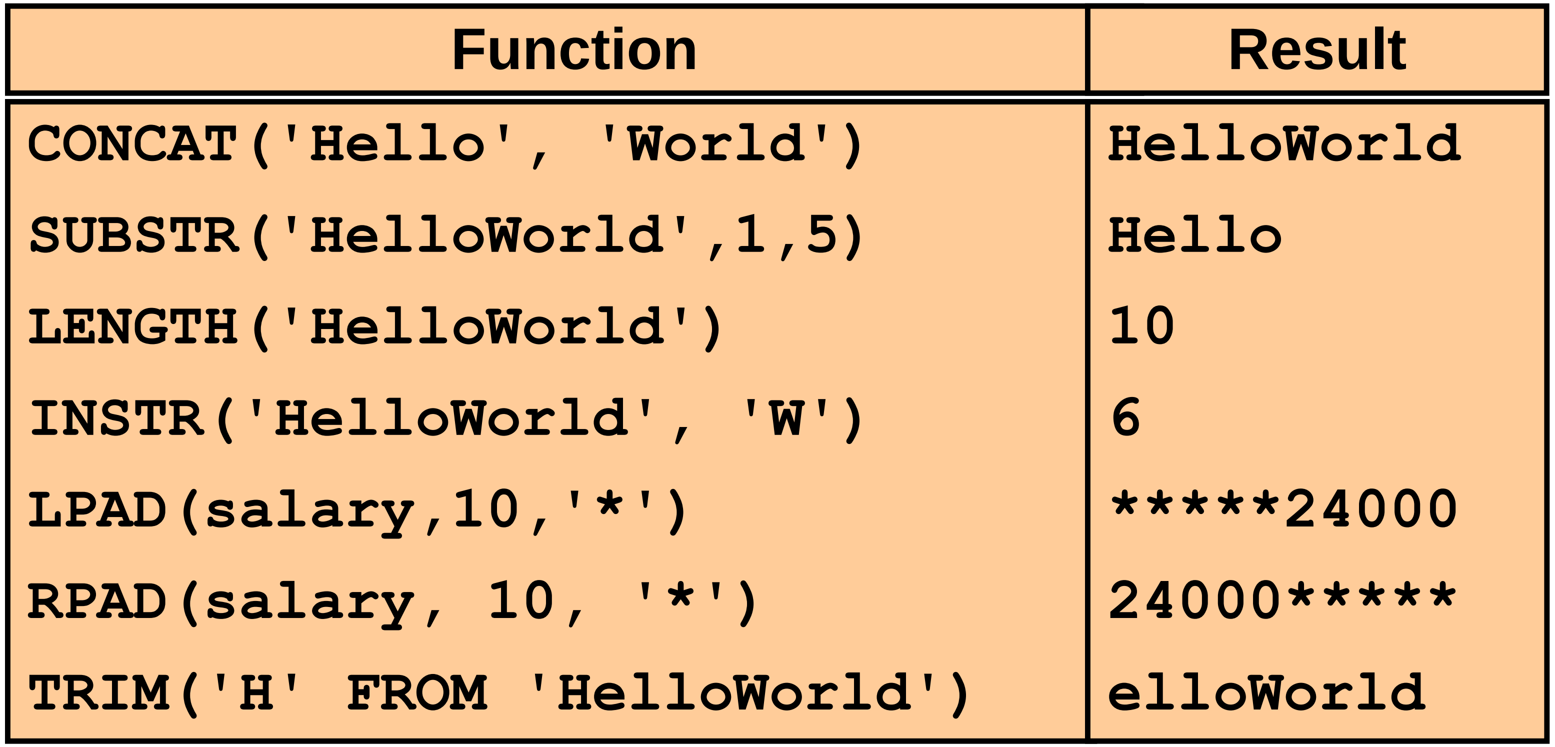
On the table, the months are stored in uppercase, here they are shown with an initial capital followed by lower case. In this example, you can clearly see the function included in the column header.

```
SQL> SELECT INITCAP('mrs. grocer')
  2  FROM dual;

INITCAP('MR
-----------
Mrs. Grocer
```

I am using the function to show particular words with initial capitals.

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`

Character functions

```
SQL> SELECT RPAD(name,20,'-'), LPAD(salary,9,'*'),
             LPAD(bonus,5,'$')
       FROM first_pay;

RPAD(NAME,20,'-')    LPAD(SALA LPAD(
-------------------- --------- -----
Linda Costa--------- ****45000 $1000
John Davidson------- ****40000 $1500
Susan Ash----------- ****25000 $$500
Stephen York-------- ****42000 $2000
Richard Jones------- ****50000 $2000
Joanne Brown-------- ****48000 $2000
```

The RPAD function pads to the right and the LPAD function pads to the left. In this example, name is right padded to its length of 20 characters with the -. Salary is left padded with * to its length of 9 and bonus is left padded with $ to its length of 5.

This kind of padding can be especially important with numeric fields that you do not want altered.

Character functions

```
SQL> SELECT SUBSTR(datefst,4,3), datefst
       FROM donor;

SUB DATEFST
--- ---------
JUL 03-JUL-98
MAY 24-MAY-97
JAN 03-JAN-98
MAR 04-MAR-92
MAR 04-MAR-92
APR 04-APR-98
```

name of column/field

position of first character of substring

length of substring

SUBSTR can be used to extract certain characters of data from a data string. In this case, I am extracting the month.

The month starts in position 4 and goes for 3 characters. Therefore I use SUBSTR(datefst,4,3).

NOTE: If length is not specified, you will get everything from the start point on.

```
SQL> SELECT SUBSTR(datefst,4)
     FROM donor;

SUBSTR(DATEFST,4)
------------------------------
JUL-98
MAY-97
JAN-98
MAR-92
MAR-92
APR-98
```

Character functions

```
SQL> SELECT * FROM donor;

IDNO  NAME            STADR           CITY       ST ZIP   DATEFST     YRGOAL CONTACT
----- --------------- --------------- ---------- -- ----- --------- --------- ------------
11111 Stephen Daniels 123 Elm St      Seekonk    MA 02345 03-JUL-98       500 John Smith
12121 Jennifer Ames   24 Benefit St   Providence RI 02045 24-MAY-97       400 Susan Jones
22222 Carl Hersey     24 Benefit St   Providence RI 02045 03-JAN-98           Susan Jones
23456 Susan Ash       21 Main St      Fall River MA 02720 04-MAR-92       100 Amy Costa
33333 Nancy Taylor    26 Oak St       Fall River MA 02720 04-MAR-92        50 John Adams
34567 Robert Brooks   36 Pine St      Fall River MA 02720 04-APR-98        50 Amy Costa

6 rows selected.
```

```
SQL> SELECT datefst, INSTR(datefst,'A')
  2  FROM donor;
```

column/field being examined

character being looked for

```
DATEFST    INSTR(DATEFST,'A')
---------  ------------------
03-JUL-98                   0
24-MAY-97                   5
03-JAN-98                   5
04-MAR-92                   5
04-MAR-92                   5
04-APR-98                   4

6 rows selected.
```

No A in JUL A in
5th character position in MAY A in 5th
character position in JAN A in 5th
character position in MAR A in 5th
character position in MAR A in 4th
character position in APR

Character functions

```
SQL> SELECT name, LENGTH(name), stadr, LENGTH(stadr), city, LENGTH(city)
  2  FROM donor;

NAME            LENGTH(NAME) STADR           LENGTH(STADR) CITY       LENGTH(CITY)
--------------- ------------ --------------- ------------- ---------- ------------
Stephen Daniels           15 123 Elm St                 10 Seekonk               7
Jennifer Ames             13 24 Benefit St              13 Providence           10
Carl Hersey               11 24 Benefit St              13 Providence           10
Susan Ash                  9 21 Main St                 10 Fall River           10
Nancy Taylor              12 26 Oak St                   9 Fall River           10
Robert Brooks             13 36 Pine St                 10 Fall River           10
```

LENGTH tells the length of the characters entered into the column/field.

NOTE: Embedded spaces are counted.

Character functions

```
SQL> SELECT jobcode, REPLACE(jobcode,'CI','IT')
  2  FROM first_pay;

JO REPL
-- ----
CI IT
IN IN
AP AP
CM CM
CI IT
IN IN
```

In this example, all rows/records that contain CI as the jobcode are displayed with IT as the jobcode.

Character functions

```
SQL> SELECT SUBSTR(startdate,4,3) || ' ' || SUBSTR(startdate,1,2)
        || ', ' || SUBSTR(startdate,8,2)
        FROM first_pay;

SUBSTR(STA
----------
JAN 15, 97
SEP 25, 92
FEB 05, 00
JUL 03, 97
OCT 30, 92
AUG 18, 94
```

This code extracts the month for the date, concatenates it with a space, then extracts the day from the date, concatenates it with a comma space and extracts the year from the date.

Character function

SUBSTR(UPPER(name),1,2)

First UPPER converts the name to upper case. Then SUBSTR takes the upper case name starts at character 1 and extracts 2 characters. The characters are therefore the first two characters of the name.

SUBSTR(idno,4,2)

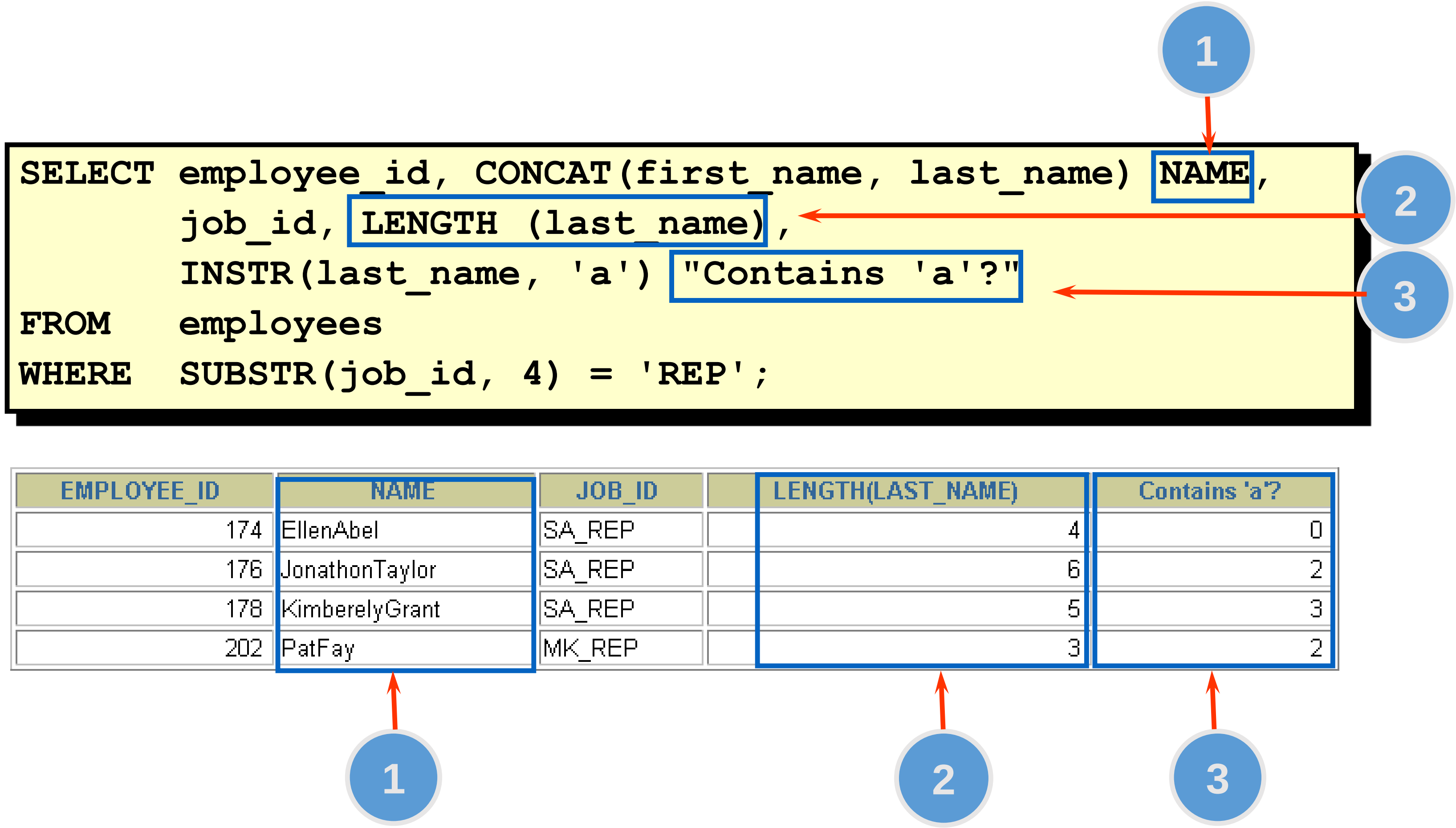
This code will start with the fourth character of the column/field idno and extract two characters. In other words, it will extract the fourth and fifth characters.

```
SQL> SELECT SUBSTR(UPPER(name),1,2) || SUBSTR(stadr,1,INSTR(stadr,' ')-1)
          || SUBSTR(idno,4,2)
     FROM donor;

SUBSTR(UPPER(NAME),
-------------------
ST12311
JE2421
CA2422
SU2156
NA2633
RO3667

6 rows selected.
```

SUBSTR(stadr,1,INSTR(stadr,' ')-1)

This code will extract a substring from stadr. It will start with the first character. The number of characters taken will be determined by using INSTR to find the space in the street address and then subtract 1 from it. Essentially this gives you the street number. Note that INSTR is determine before SUBSTR.

Using the Character-Manipulation Functions



```
SELECT employee_id, CONCAT(first_name, last_name) NAME,
       job_id, LENGTH (last_name),
       INSTR(last_name, 'a') "Contains 'a'?"
FROM   employees
WHERE  SUBSTR(job_id, 4) = 'REP';
```

EMPLOYEE_ID	NAME	JOB_ID	LENGTH(LAST_NAME)	Contains 'a'?
174	EllenAbel	SA_REP	4	0
176	JonathonTaylor	SA_REP	6	2
178	KimberelyGrant	SA_REP	5	3
202	PatFay	MK_REP	3	2

Number Functions

- ROUND: Rounds value to specified decimal

  ROUND(45.926, 2) → 45.93

- TRUNC: Truncates value to specified decimal

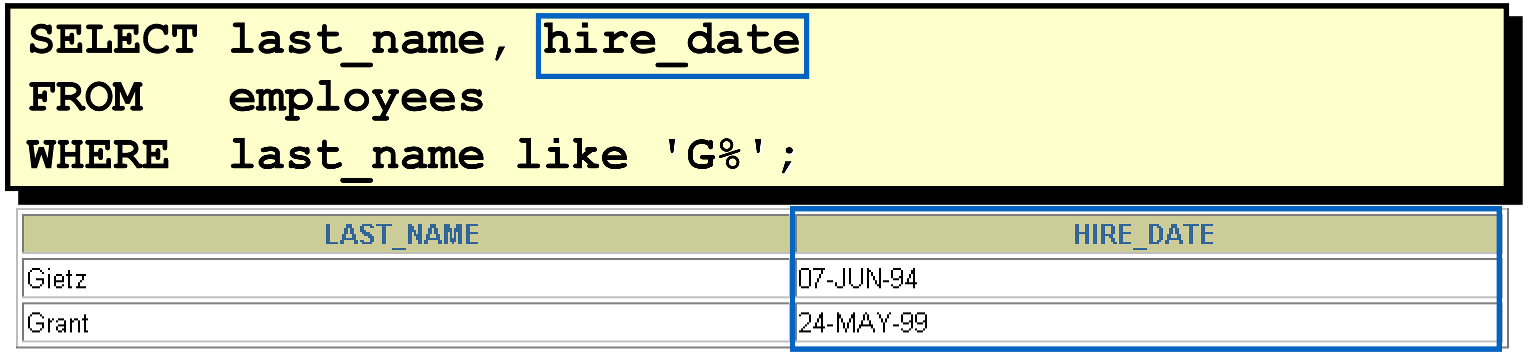
  TRUNC(45.926, 2) → 45.92

- MOD: Returns remainder of division

  MOD(1600, 300) → 100

Working with Dates

- Oracle database stores dates in an internal numeric format: century, year, month, day, hours, minutes, seconds.
- The default date display format is DD-MON-YY.



```
SELECT last_name, hire_date
FROM   employees
WHERE  last_name like 'G%';
```

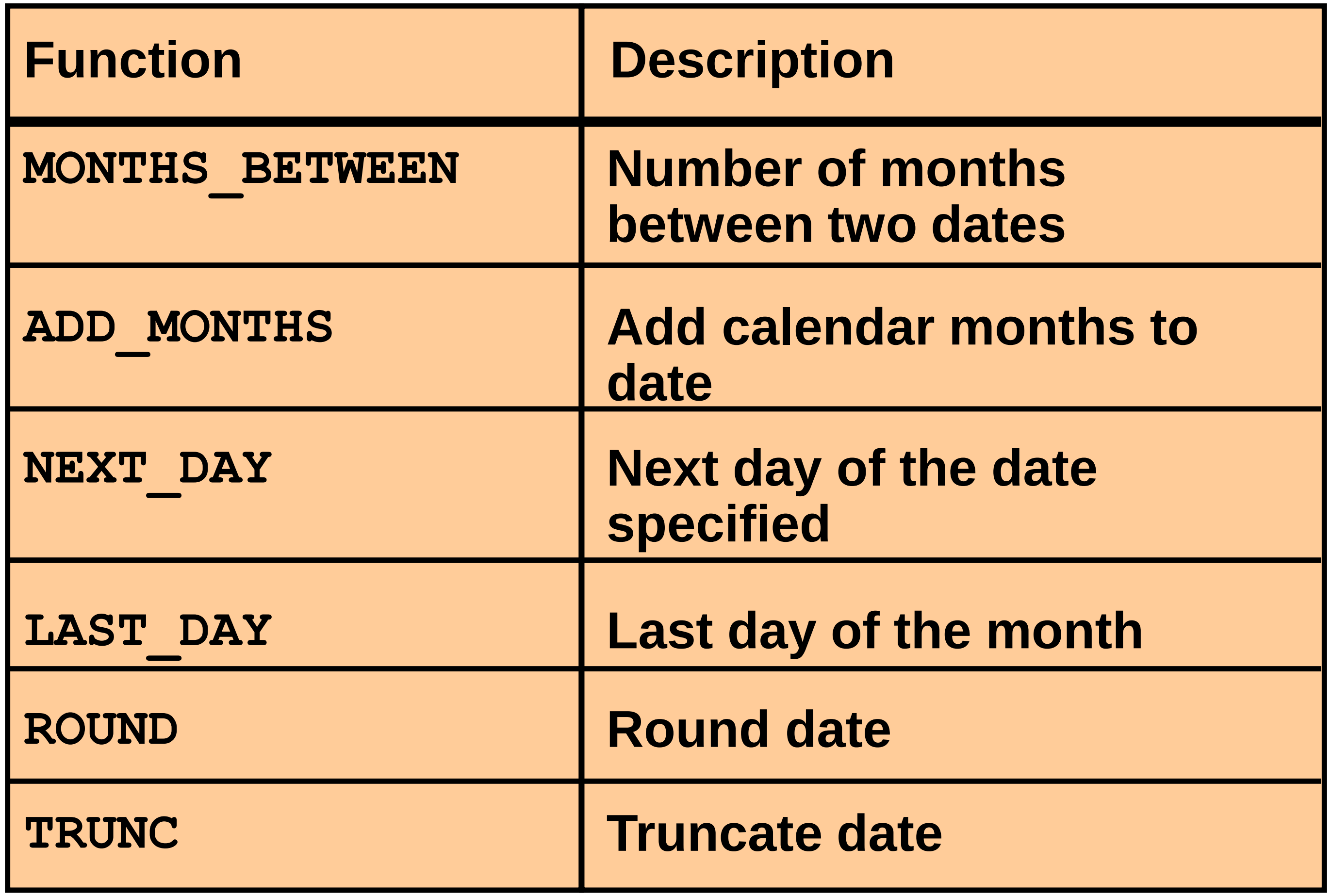
LAST_NAME	HIRE_DATE
Gietz	07-JUN-94
Grant	24-MAY-99

Working with Dates

`SYSDATE` is a function that returns:

- Date / Time

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 (‘11-SEP-95','11-JAN-94')

  → 20

- 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'`:

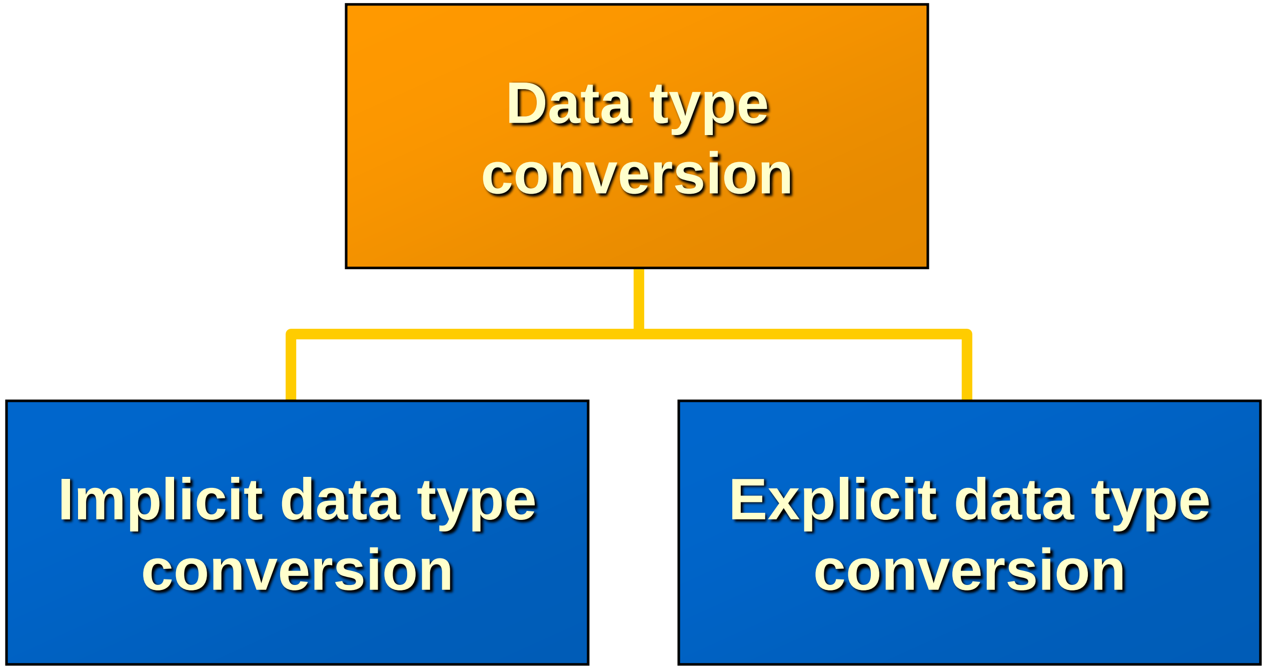
- `ROUND(SYSDATE,'MONTH')` → `01-AUG-95`
- `ROUND(SYSDATE ,'YEAR')` → `01-JAN-96`
- `TRUNC(SYSDATE ,'MONTH')` → `01-JUL-95`
- `TRUNC(SYSDATE ,'YEAR')` → `01-JAN-95`

Date Function : Example

- For all employees employed for fewer than 200 months, display the employee number, hire date, number of months employed, six-month review date, first Friday after hire date, and last day of the month when hired.

```
SELECT        empno, hiredate,
  MONTHS_BETWEEN(SYSDATE, hiredate)
  TENURE,
  ADD_MONTHS(hiredate, 6) REVIEW,
  NEXT_DAY(hiredate, 'FRIDAY'),
  LAST_DAY(hiredate)
FROM emp
WHERE   MONTHS_BETWEEN
  (SYSDATE,hiredate)<200;
```

Conversion Functions

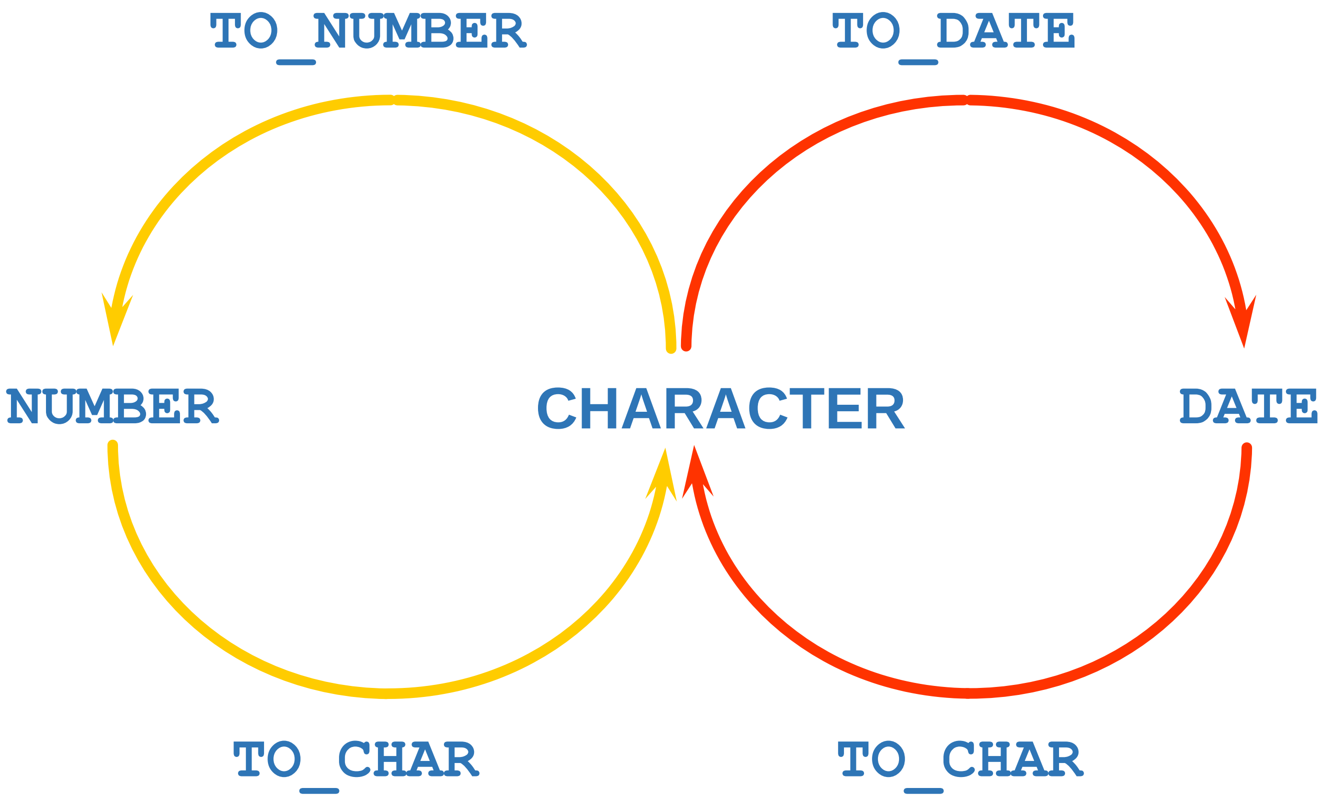


Implicit Datatype Conversion

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

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

Explicit Data Type Conversion



Using the TO_CHAR Function with Dates

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

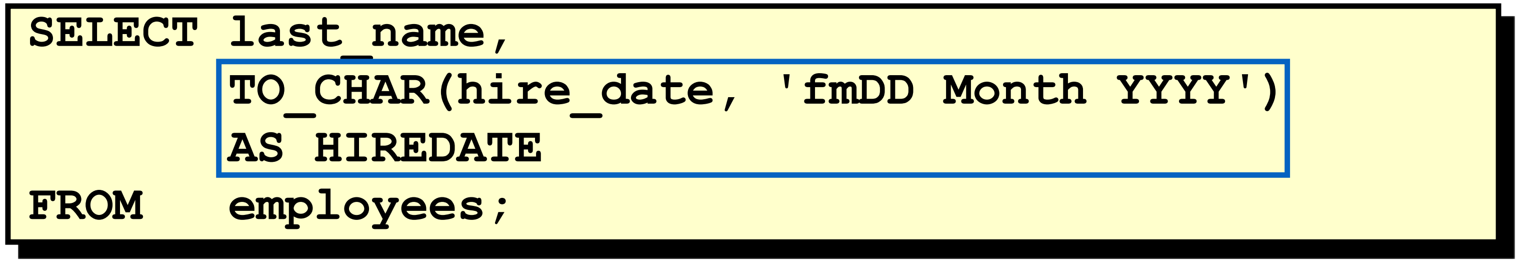
The format model:

- Must be enclosed in single quotation marks and is case sensitive
- Can include any valid date format element
- Has an *fm* element to remove padded blanks or suppress leading zeros
- Is separated from the date value by a comma

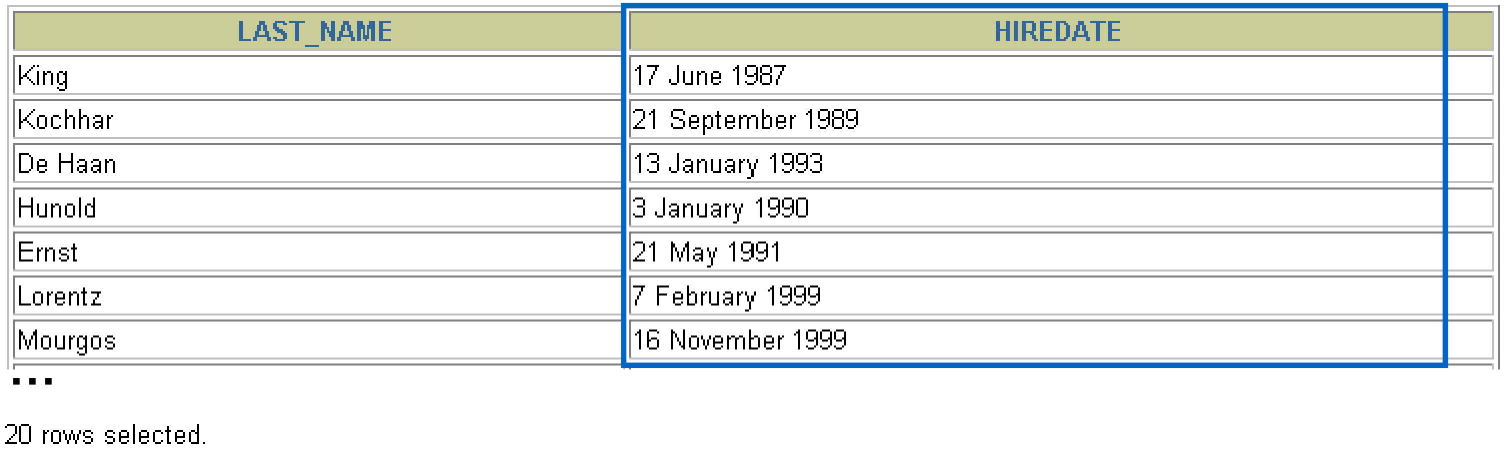
Elements of the Date Format Model

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

Using the TO_CHAR Function with Dates



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



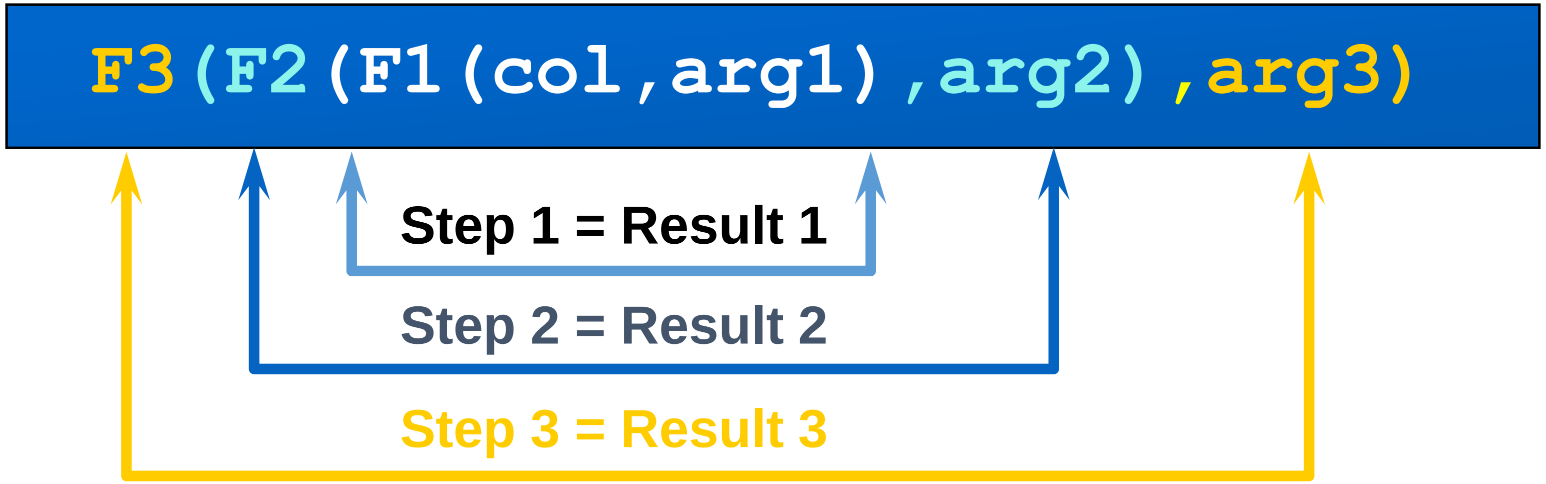
LAST_NAME	HIREDATE
King	17 June 1987
Kochhar	21 September 1989
De Haan	13 January 1993
Hunold	3 January 1990
Ernst	21 May 1991
Lorentz	7 February 1999
Mourgos	16 November 1999

...

20 rows selected.

Nesting Functions

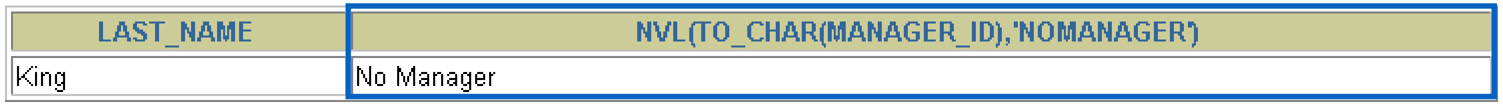
- Single-row functions can be nested to any level.
- Nested functions are evaluated from deepest level to the least deep level.



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

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)`

`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')`

Introduction

- Querying one table already done & practiced!
- Real power of relational database
  - Storage of data in multiple tables
  - Necessitates creating queries to use multiple tables
- Two Basic approaches for processing multiple tables
  - Sub-queries
  - Join

Processing Multiple Tables Using Joins

- Join - Most frequently used operation - brings together data from multiple tables into one resultant table

- Join can be achieved in two ways

  - Implicitly by referring in a WHERE clause to the matching of common columns over which the tables are joined

  - Explicitly by JOIN…..ON commands in FROM clause

What is the Join?

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

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

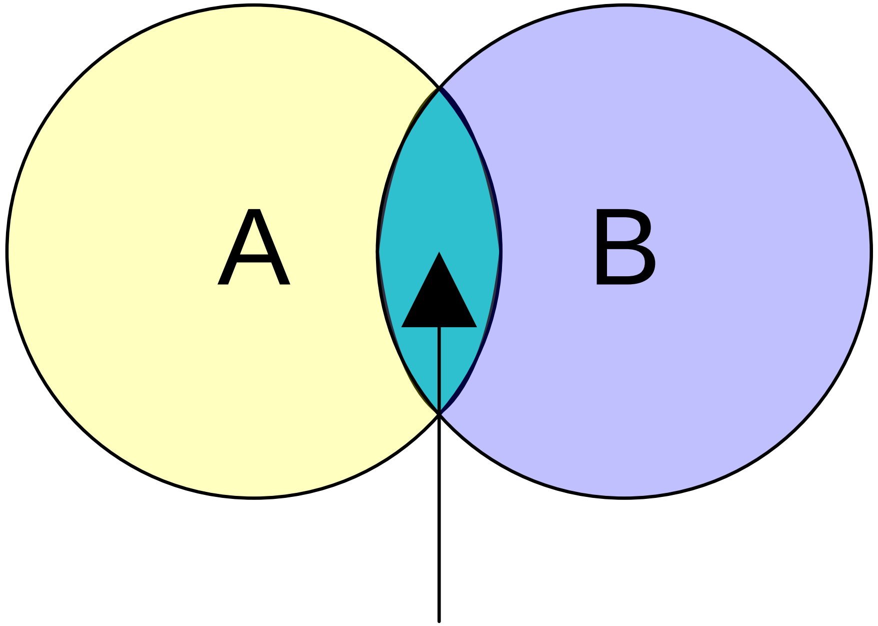
- Write the join condition in the WHERE clause
- Prefix the column name with the table name when the same column name appears in more than one table

Types of Joins

Joins that are compliant with the SQL include the following:

- Equijoin / Inner Join
- Natural joins
- Self join
- Non-equijoin
- Outer join
- Cross Join

SQL Joins: Defining Join Types: `INNER JOIN`



INTERSECTION

SQL Joins Defining Join Types: `INNER JOIN`

- An **`INNER JOIN`** is also an *equijoin*, or equality join between equals.
- An **`INNER JOIN`** matches on one or a set of columns values from one table:
  - When one table is involved, an **`INNER JOIN`** creates an intersection between two copies of a single table (typically done with two different column names).
  - When two or more tables are involved, an **`INNER JOIN`** creates an intersection between the tables based on designated column names.

Defining Join Types: `INNER JOIN`

- Create an `INNER JOIN` by placing a position specific set of tables in the `FROM` clause followed by an `ON` or `USING` clause.
- Equality statements are between one or more columns in two copies of one table or two tables:
- When the columns share the same name and data type,
  - use the `USING` clause.
- When the columns have different names but the same data type,
  - use the `ON` clause.
- If only the word `JOIN` is used, an `INNER JOIN` is assumed by the SQL parser.

Defining Join Types: **INNER JOIN**

-
  ```
  SELECT  a.column1, b.column2
   FROM   table1 a, table2 b
    WHERE  a.columnpk = b.columnfk;
  ```

-
  ```
   SELECT   a.column1, b.column2
   FROM    table1 a [INNER] JOIN table2 b
     ON     a.columnpk = b.columnfk;
  ```

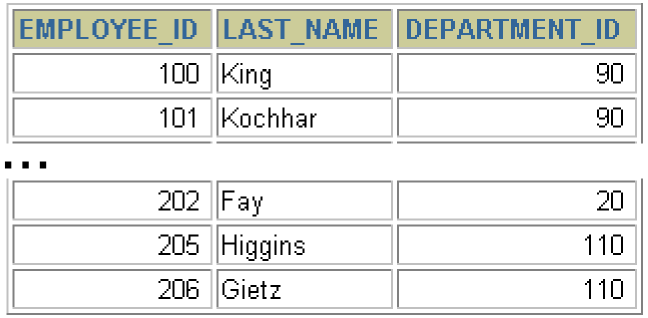
-
  ```
   SELECT   a.column1, b.column2
   FROM      table1 a [INNER] JOIN table2 b
     USING (same_column_name);
  ```

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.

Generating a Cartesian Product

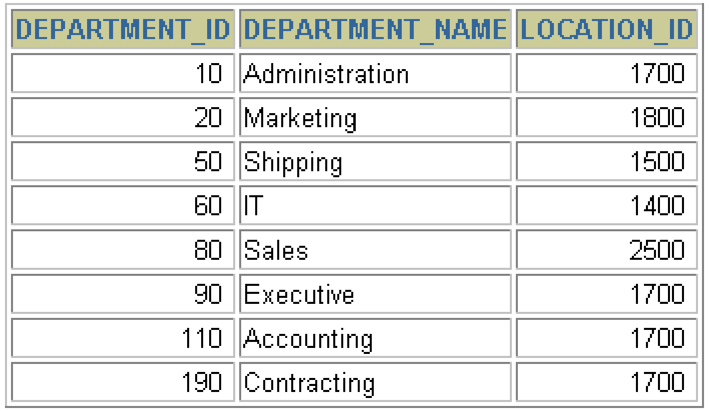
EMPLOYEES (20 rows)



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

20 rows selected.

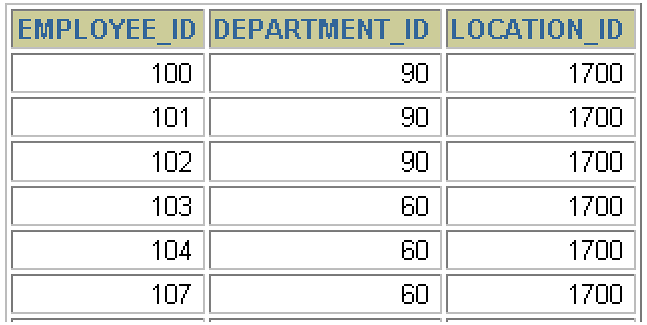
DEPARTMENTS (8 rows)



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:
20 x 8 = 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

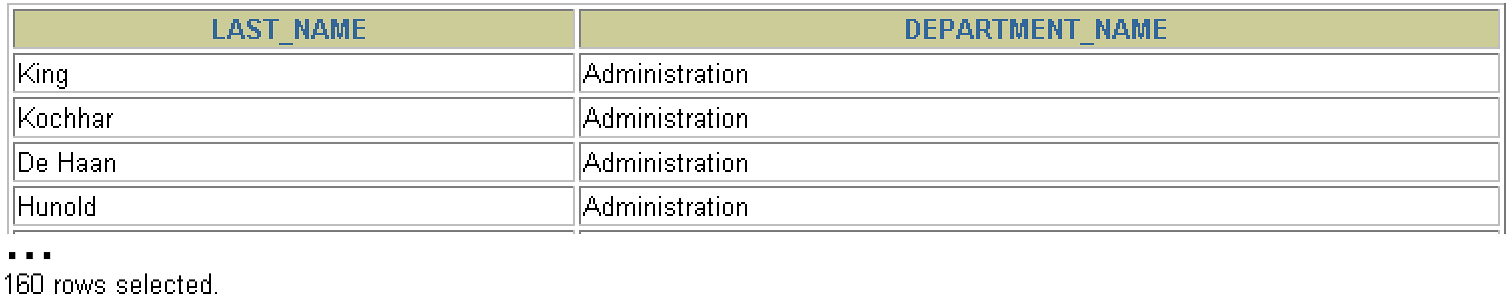
…

160 rows selected.

Creating Cross Joins

- The CROSS JOIN clause produces the cross-product of two tables.
- This is also called 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

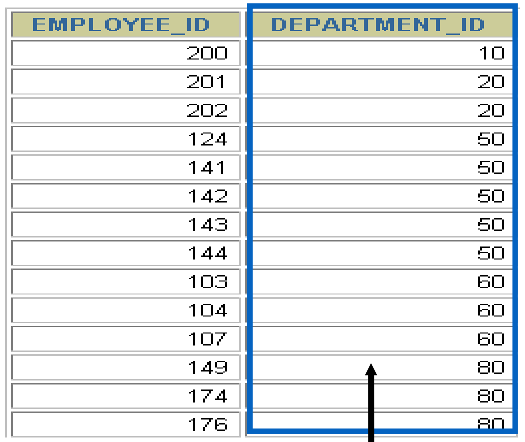
...

160 rows selected.

Retrieving Record with Equijoin

Employees ∞ Department

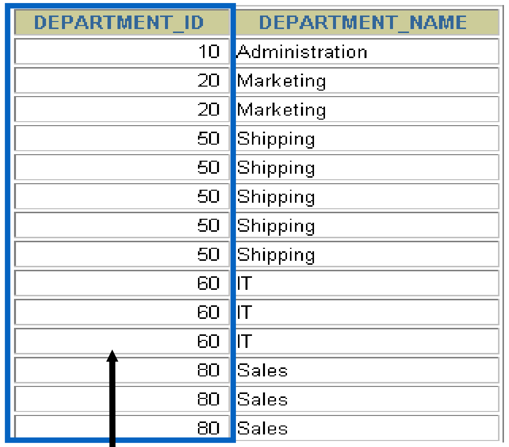
EMPLOYEES



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

Foreign key

DEPARTMENTS



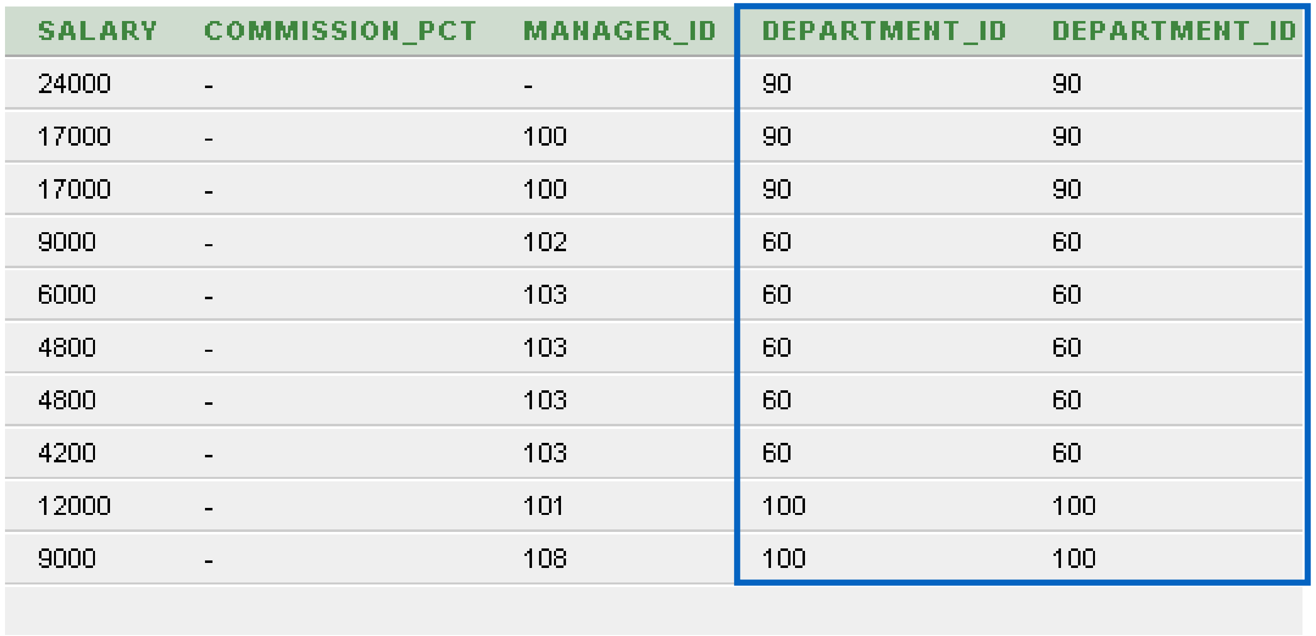
DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
20	Marketing
50	Shipping
50	Shipping
50	Shipping
50	Shipping
50	Shipping
60	IT
60	IT
60	IT
80	Sales
80	Sales
80	Sales

Primary key

Using Equijoin

Write SQL statement to do this: Employees ∞ Department

```
Select  *
From  employees ,departments
Where employees.department_id = departments.department_id
```



SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID	DEPARTMENT_ID
24000	-	-	90	90
17000	-	100	90	90
17000	-	100	90	90
9000	-	102	60	60
6000	-	103	60	60
4800	-	103	60	60
4800	-	103	60	60
4200	-	103	60	60
12000	-	101	100	100
9000	-	108	100	100

Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.
- Use column aliases to distinguish columns that have identical names but reside in different tables.

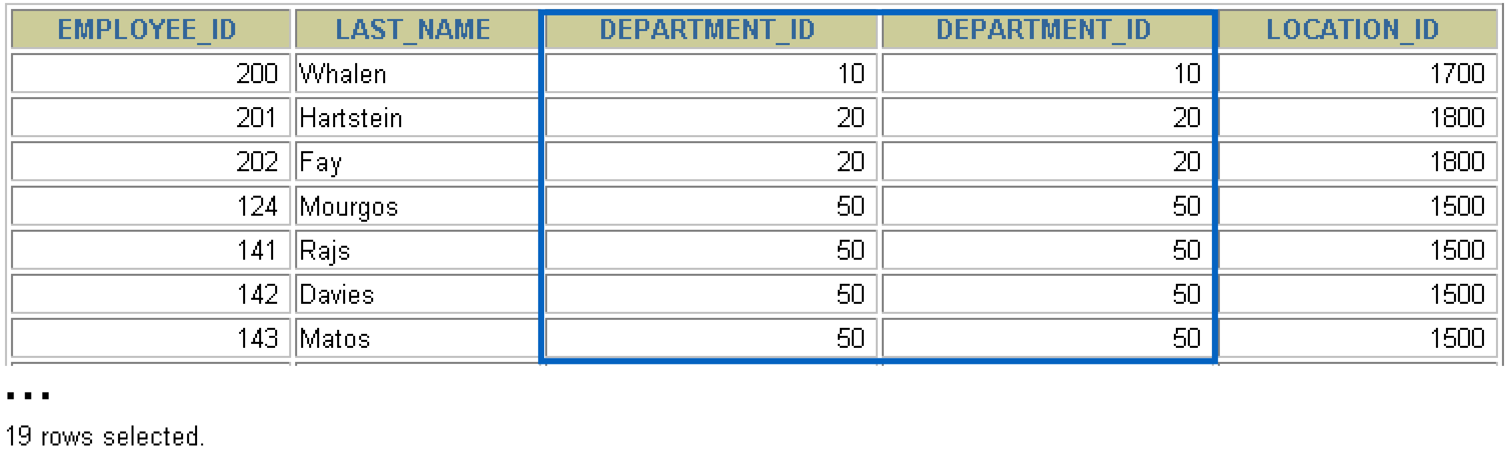
Using Table Aliases

- Use table aliases to simplify queries.
- Use table aliases to improve performance.

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

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 INNER JOIN departments d
ON     (e.department_id = d.department_id);
```



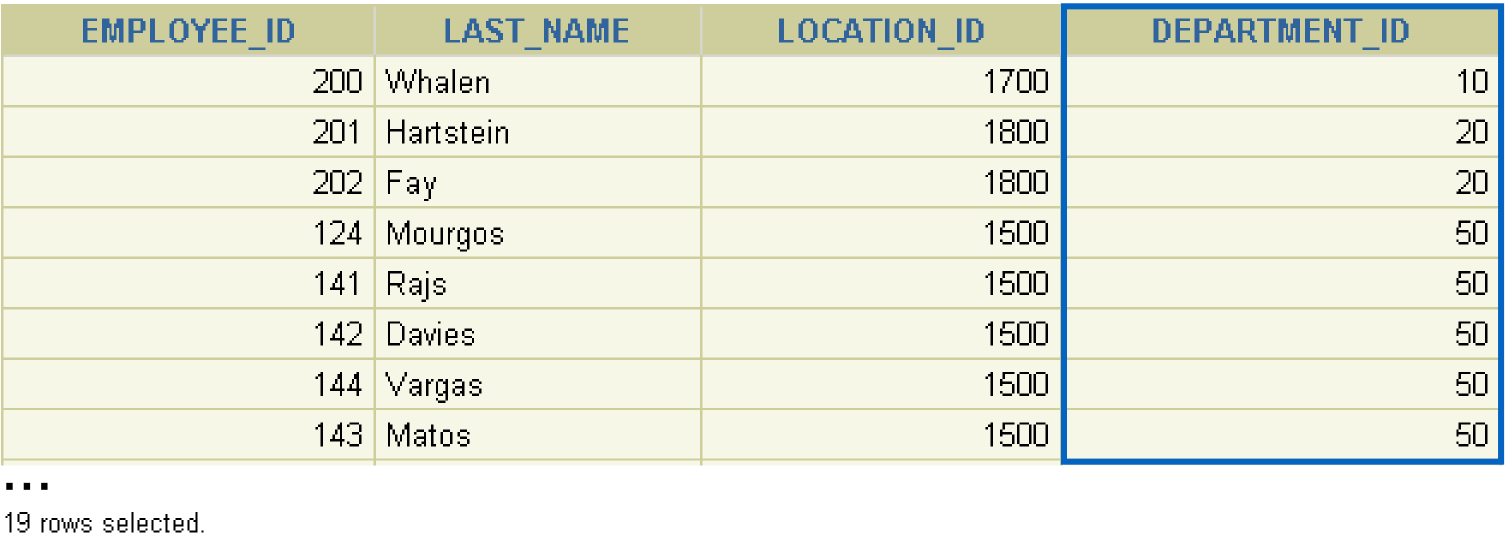
EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID	LOCATION_ID
200	Whalen	10	10	1700
201	Hartstein	20	20	1800
202	Fay	20	20	1800
124	Mourgos	50	50	1500
141	Rajs	50	50	1500
142	Davies	50	50	1500
143	Matos	50	50	1500

...

19 rows selected.

Retrieving Records with the USING Clause

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



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

...

19 rows selected.

```
SELECT s.sid, s.name, r.bid
FROM Sailors s INNER JOIN Reserves r
ON s.sid = r.sid
```

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

sid	bid	day
22	101	10/10/96
95	103	11/12/96

s.sid	s.name	r.bid
22	Dustin	101
95	Bob	103

Joins Example

- Show all customers and order date who have placed an order

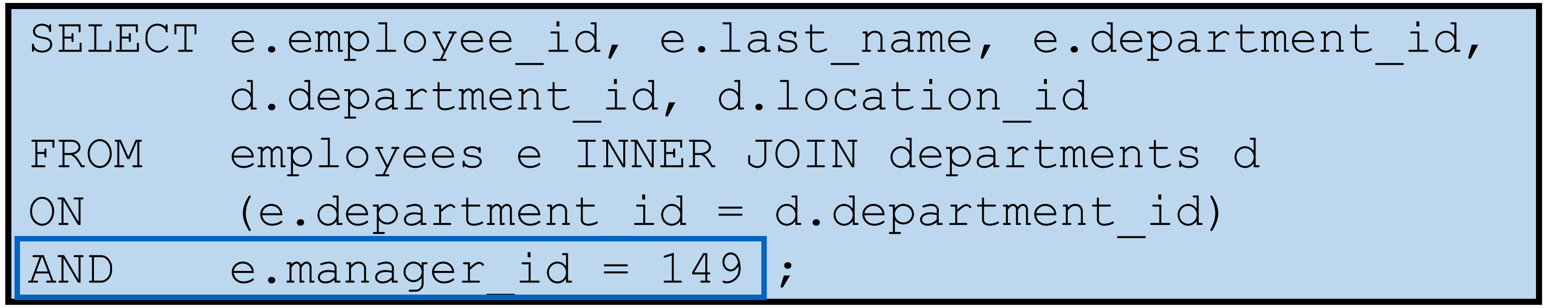
```
SELECT CUSTOMER_NAME , ORDER_DATE
          FROM CUSTOMER, ORDER
                    WHERE CUSTOMER.CUSTOMER_ID = ORDER.CUSTOMER_ID
```

```
SELECT CUSTOMER_NAME , ORDER_DATE
      FROM CUSTOMER INNER JOIN ORDER
      ON        CUSTOMER.CUSTOMER_ID = ORDER.CUSTOMER_ID
```

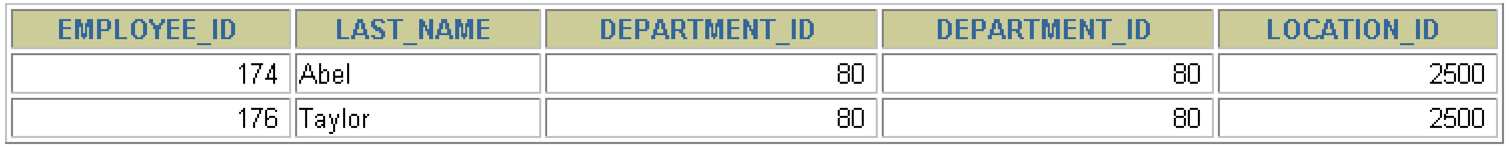
```
SELECT CUSTOMER_NAME , ORDER_DATE
  FROM CUSTOMER INNER JOIN ORDER
       USING CUSTOMER_ID
```

Applying Additional Conditions to a Join

- Show employee id , last name, dept id and location id who have a manager ID 149.



```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
FROM   employees e INNER 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

Joins Example

- Show the students’ name and marks who failed in course CSC271

```
SELECT   S.std_name,   R.marks
         FROM    Student S   INNER JOIN   Result R
         ON      S.std_id = R.std_id
         AND     R.marks<50          AND          course_id = ‘CSC271’
```

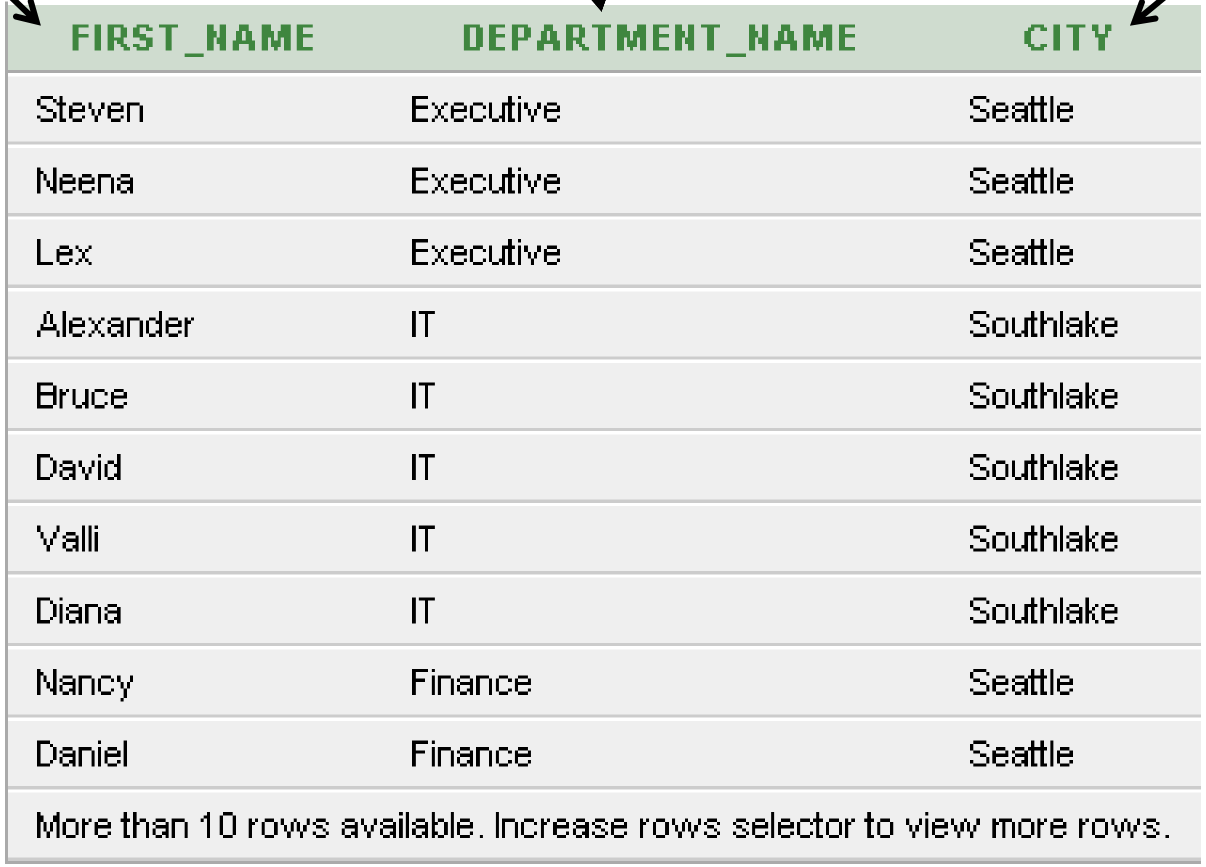
```
SELECT   S.std_name,   R.marks
         FROM    Student S   INNER JOIN   Result R
         USING   std_id
         AND     R.marks<50          AND          course_id = ‘CSC271’
```

Joining More than two table

Employees

Departments

Locations



FIRST_NAME	DEPARTMENT_NAME	CITY
Steven	Executive	Seattle
Neena	Executive	Seattle
Lex	Executive	Seattle
Alexander	IT	Southlake
Bruce	IT	Southlake
David	IT	Southlake
Valli	IT	Southlake
Diana	IT	Southlake
Nancy	Finance	Seattle
Daniel	Finance	Seattle

More than 10 rows available. Increase rows selector to view more rows.

Joining More than two table

```
select first_name,department_name,city
from employees E,departments D,locations L
where E.department_id=D.department_id
      and D.location_id=L.location_id
```

```
select first_name,department_name,city
from employees
JOIN departments
ON(employees.department_id=departments.department_id)
JOIN locations
ON(departments.location_id=locations.location_id)
```

```
select first_name,department_name,city
from employees JOIN departments using(department_id)
               JOIN locations using(location_id)
```

SQL Joins Defining Join Types: Non-equijoin

- A *non-equijoin* is an indirect match:
  - Occurs when one column value is found in the range between two other column values
  - Uses the **`BETWEEN`** operator.
  - Also occurs when one column value is found by matching against a criterion using an inequality operator.

SQL Joins Defining Join Types: Non-equijoin

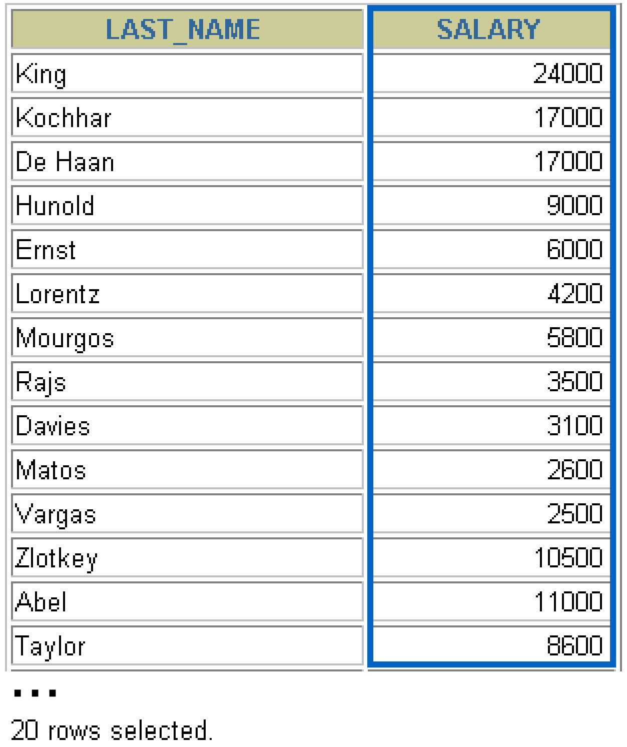
- Example:

```
SELECT   a.column1, b.column2
FROM     table1 a, table2 b
WHERE    a.columnpk >= b.columnfk;
```

```
SELECT    a.column1, b.column2
FROM      table1 a, table2 b
WHERE     a.cola BETWEEN  b.colx AND b.coly;
```

Non-Equijoins

EMPLOYEES

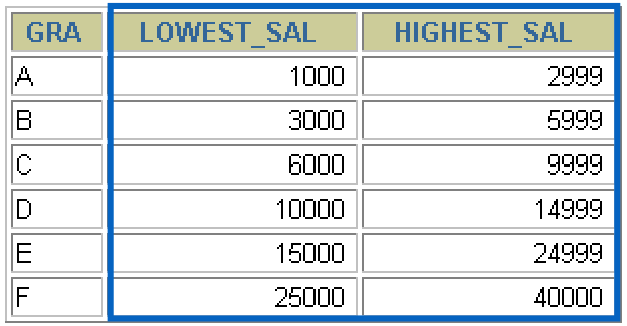


LAST_NAME	SALARY
King	24000
Kochhar	17000
De Haan	17000
Hunold	9000
Ernst	6000
Lorentz	4200
Mourgos	5800
Rajs	3500
Davies	3100
Matos	2600
Vargas	2500
Zlotkey	10500
Abel	11000
Taylor	8600

…

20 rows selected.

JOB_GRADES

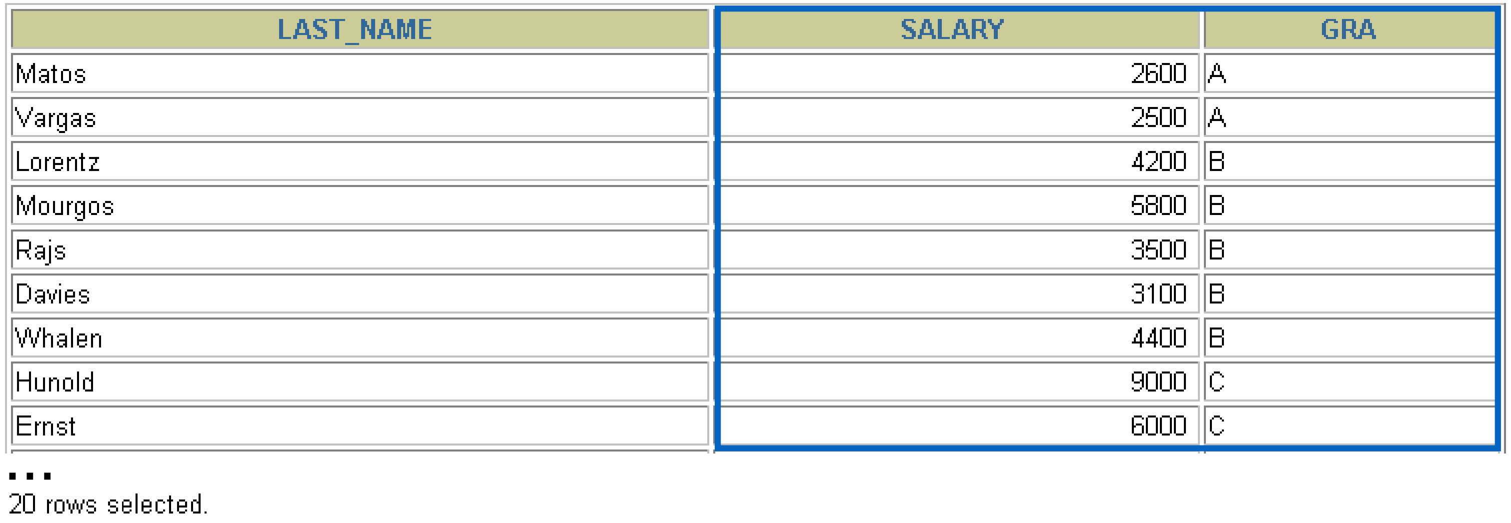


GRA	LOWEST_SAL	HIGHEST_SAL
A	1000	2999
B	3000	5999
C	6000	9999
D	10000	14999
E	15000	24999
F	25000	40000

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 JOIN job_grades j
ON     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.

Types of Joins

Joins that are compliant with the SQL include the following:

- Equijoin / Inner Join
- Cross Join
- Non-equijoin
- Natural joins
- Outer join
- Self join

SQL Joins

Defining Join Types: Natural Join

- We have already learned that an EQUI JOIN performs a JOIN against equality or matching column(s) values of the associated tables and an equal sign (=) is used as comparison operator in the where clause to refer equality.

- The SQL NATURAL JOIN is a type of EQUI JOIN and is structured in such a way that, columns with same name of associate tables will appear once only.

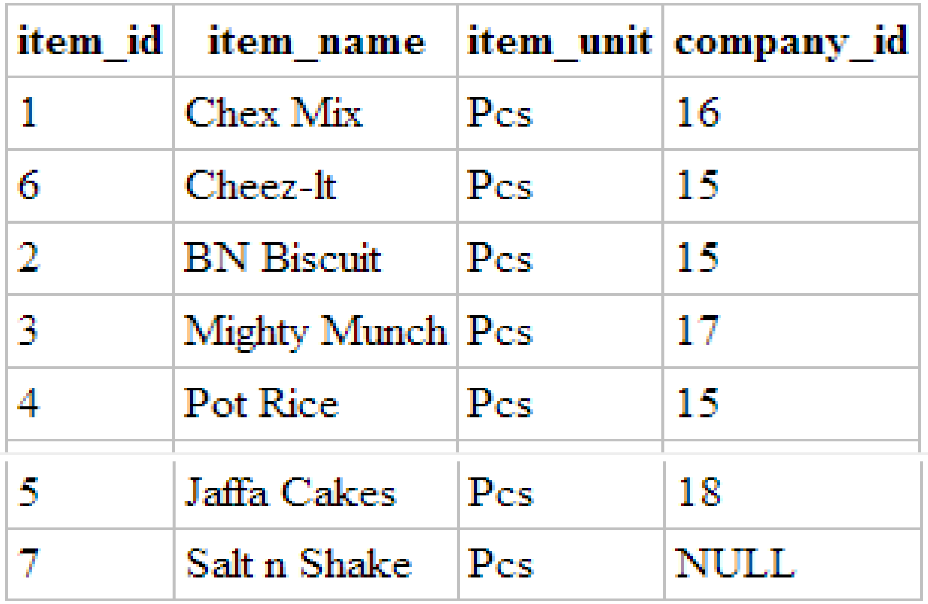
Natural Join : Guidelines

- The associated tables have one or more pairs of identically named columns.
- The columns must be the same data type.
- No need to use ON clause in a natural join.

```
SELECT   a.column1, b.column2
 FROM table1 a NATURAL JOIN table2 b;
```

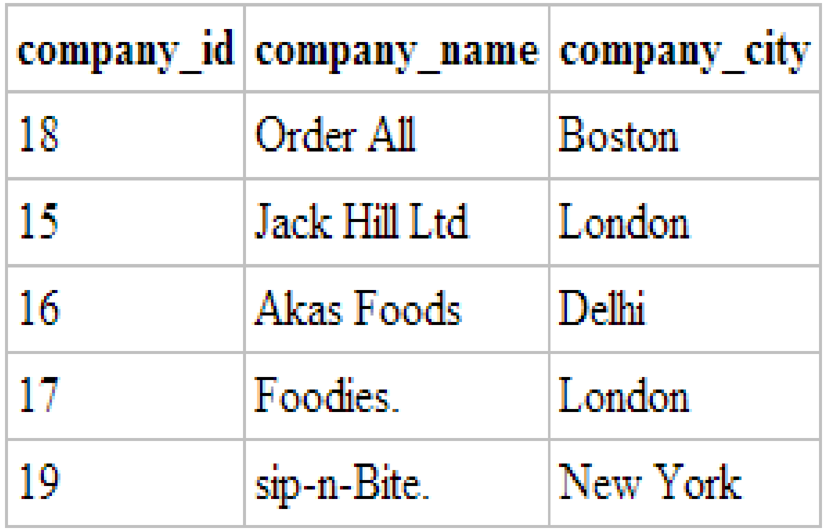
NATURAL JOIN - EXAMPLE

Food



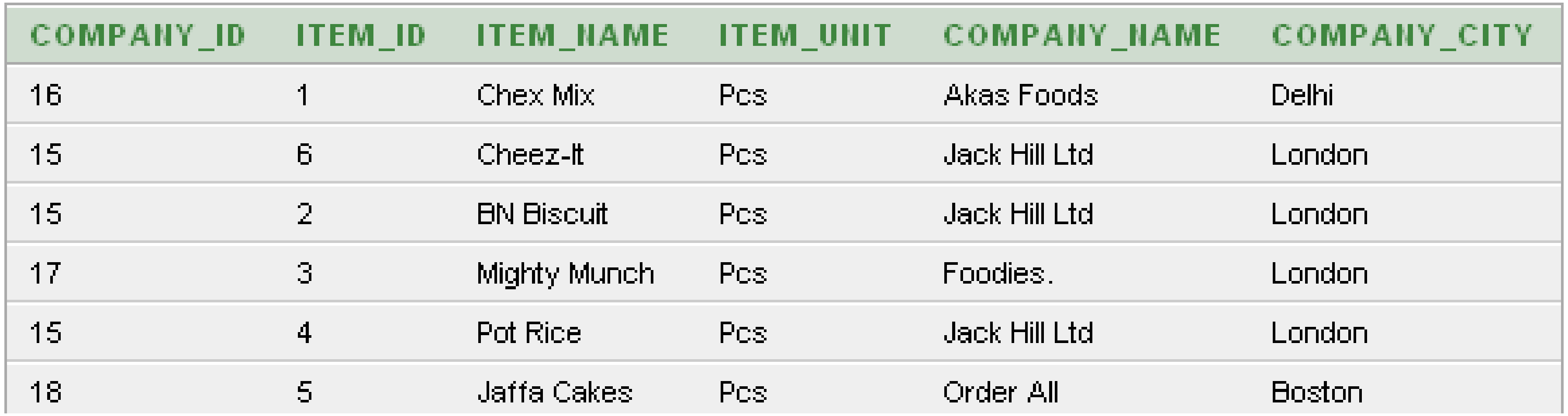
item_id	item_name	item_unit	company_id
1	Chex Mix	Pcs	16
6	Cheez-It	Pcs	15
2	BN Biscuit	Pcs	15
3	Mighty Munch	Pcs	17
4	Pot Rice	Pcs	15
5	Jaffa Cakes	Pcs	18
7	Salt n Shake	Pcs	NULL

COMPANY

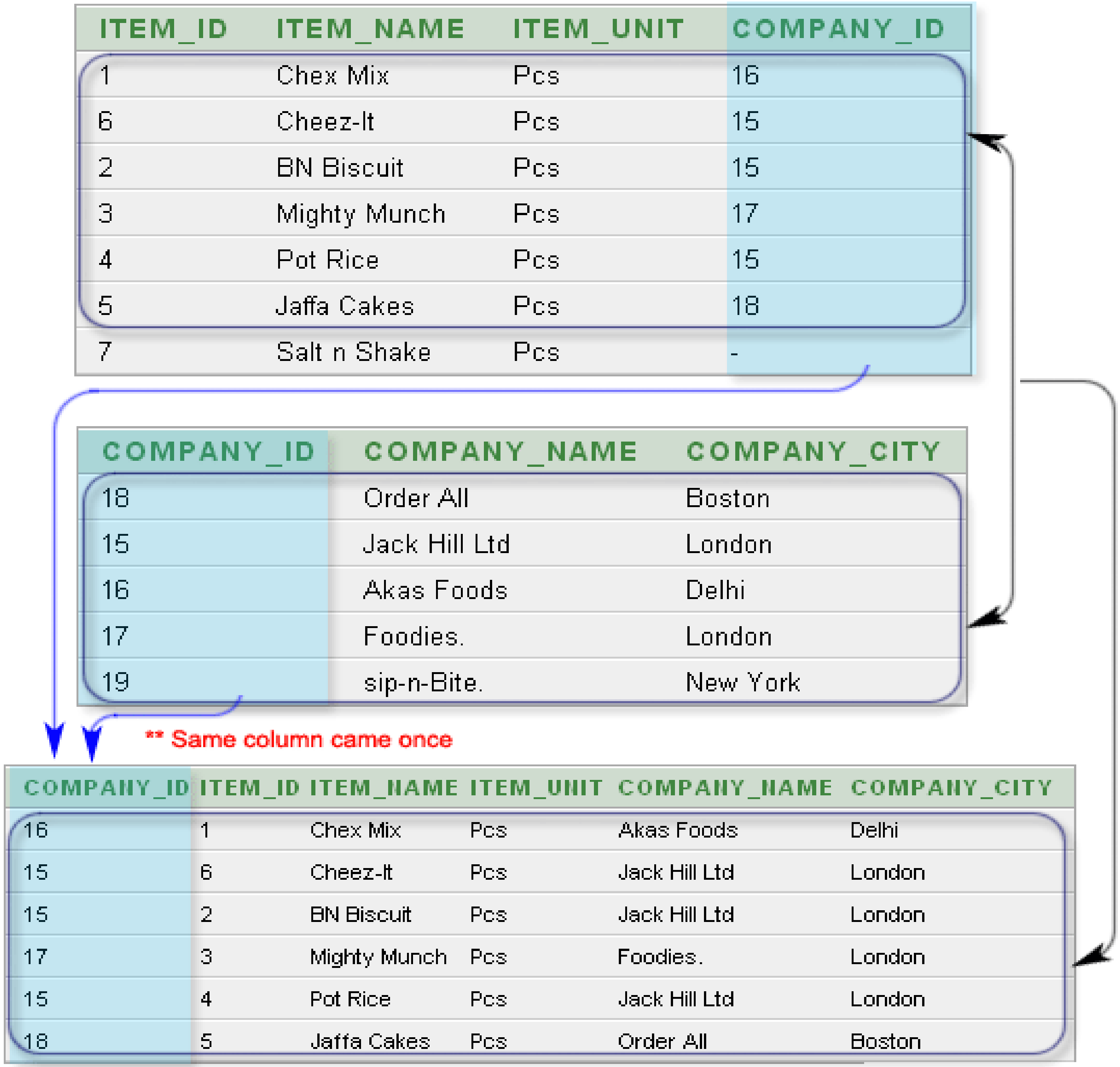


company_id	company_name	company_city
18	Order All	Boston
15	Jack Hill Ltd	London
16	Akas Foods	Delhi
17	Foodies.	London
19	sip-n-Bite.	New York

- Select * from Food NATURAL JOIN Company



COMPANY_ID	ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_NAME	COMPANY_CITY
16	1	Chex Mix	Pcs	Akas Foods	Delhi
15	6	Cheez-It	Pcs	Jack Hill Ltd	London
15	2	BN Biscuit	Pcs	Jack Hill Ltd	London
17	3	Mighty Munch	Pcs	Foodies.	London
15	4	Pot Rice	Pcs	Jack Hill Ltd	London
18	5	Jaffa Cakes	Pcs	Order All	Boston



ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_ID
1	Chex Mix	Pcs	16
6	Cheez-It	Pcs	15
2	BN Biscuit	Pcs	15
3	Mighty Munch	Pcs	17
4	Pot Rice	Pcs	15
5	Jaffa Cakes	Pcs	18
7	Salt n Shake	Pcs	-

COMPANY_ID	COMPANY_NAME	COMPANY_CITY
18	Order All	Boston
15	Jack Hill Ltd	London
16	Akas Foods	Delhi
17	Foodies.	London
19	sip-n-Bite.	New York

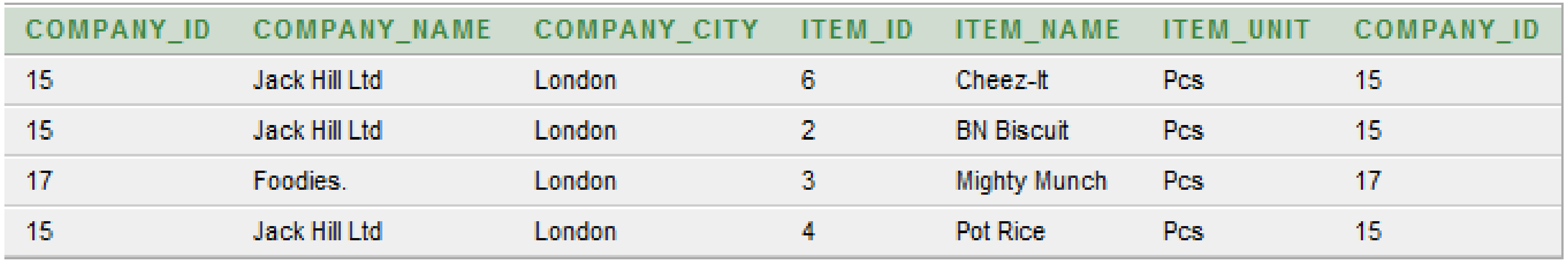
** Same column came once

COMPANY_ID	ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_NAME	COMPANY_CITY
16	1	Chex Mix	Pcs	Akas Foods	Delhi
15	6	Cheez-It	Pcs	Jack Hill Ltd	London
15	2	BN Biscuit	Pcs	Jack Hill Ltd	London
17	3	Mighty Munch	Pcs	Foodies.	London
15	4	Pot Rice	Pcs	Jack Hill Ltd	London
18	5	Jaffa Cakes	Pcs	Order All	Boston

Difference btw INNER JOIN & NATURAL JOIN

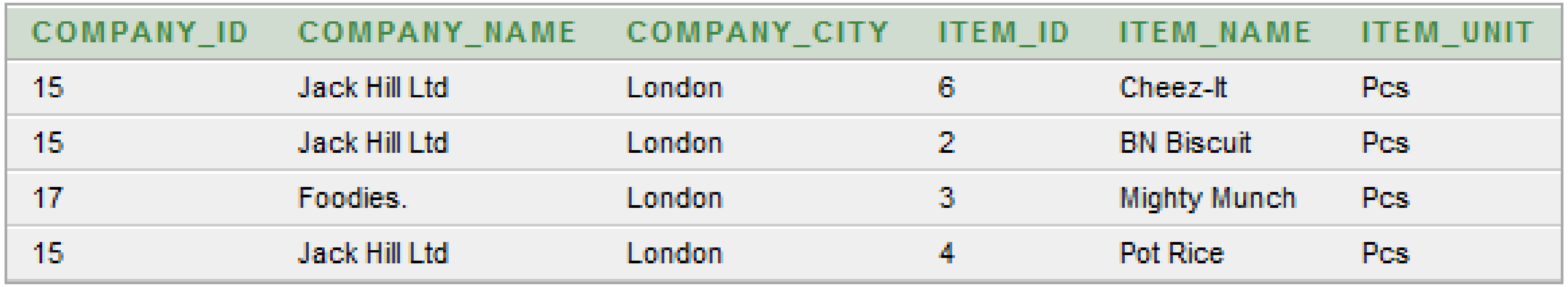
- SELECT * FROM company INNER JOIN food

  ON company.company_id = food.company_id;



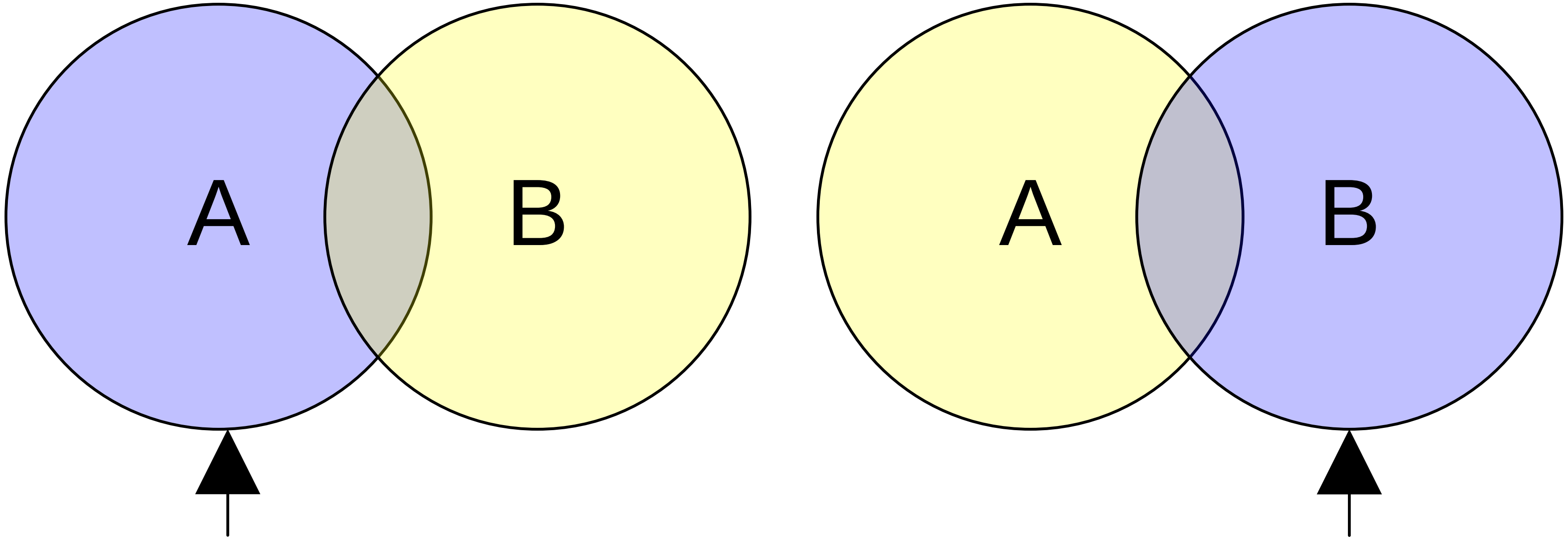
COMPANY_ID	COMPANY_NAME	COMPANY_CITY	ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_ID
15	Jack Hill Ltd	London	6	Cheez-It	Pcs	15
15	Jack Hill Ltd	London	2	BN Biscuit	Pcs	15
17	Foodies.	London	3	Mighty Munch	Pcs	17
15	Jack Hill Ltd	London	4	Pot Rice	Pcs	15

- Select * from company NATURAL JOIN food



COMPANY_ID	COMPANY_NAME	COMPANY_CITY	ITEM_ID	ITEM_NAME	ITEM_UNIT
15	Jack Hill Ltd	London	6	Cheez-It	Pcs
15	Jack Hill Ltd	London	2	BN Biscuit	Pcs
17	Foodies.	London	3	Mighty Munch	Pcs
15	Jack Hill Ltd	London	4	Pot Rice	Pcs

SQL Joins Defining Join Types: Outer Join



LEFT JOIN RIGHT JOIN

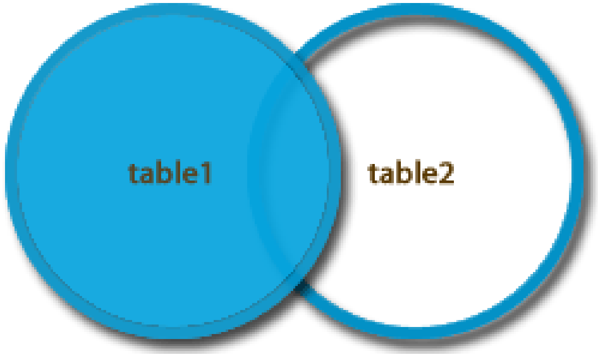
SQL Joins Outer Join

- ANSI Syntax:
  - These are defined by **LEFT JOIN** and **RIGHT JOIN** operators.
  - Both **LEFT [OUTER] JOIN** and **RIGHT [OUTER] JOIN** are synonymous with **LEFT JOIN** and **RIGHT JOIN** respectively, the **OUTER** is assumed when left out.
  - The **LEFT [OUTER] JOIN** returns all matched rows, plus all unmatched rows from the table on the left of the join clause(use nulls in fields of non-matching tuples)
  - The **RIGHT [OUTER] JOIN** returns all matched rows, plus all unmatched rows from the table on the right of the join clause.

SQL Joins Defining Join Types: Outer Join

- Oracle Syntax:
  - The “ `(+)` ” symbol is used to create an **`OUTER JOIN`**.
  - When the “ `(+)` ” symbol is on the right of the join operand, it acts as the equivalent of a **`LEFT JOIN`**.
  - When the “ `(+)` ” it is on the left of the join operand, it is the equivalent of a **`RIGHT JOIN`**.

Left Outer Join



- ANSI SQL Example:

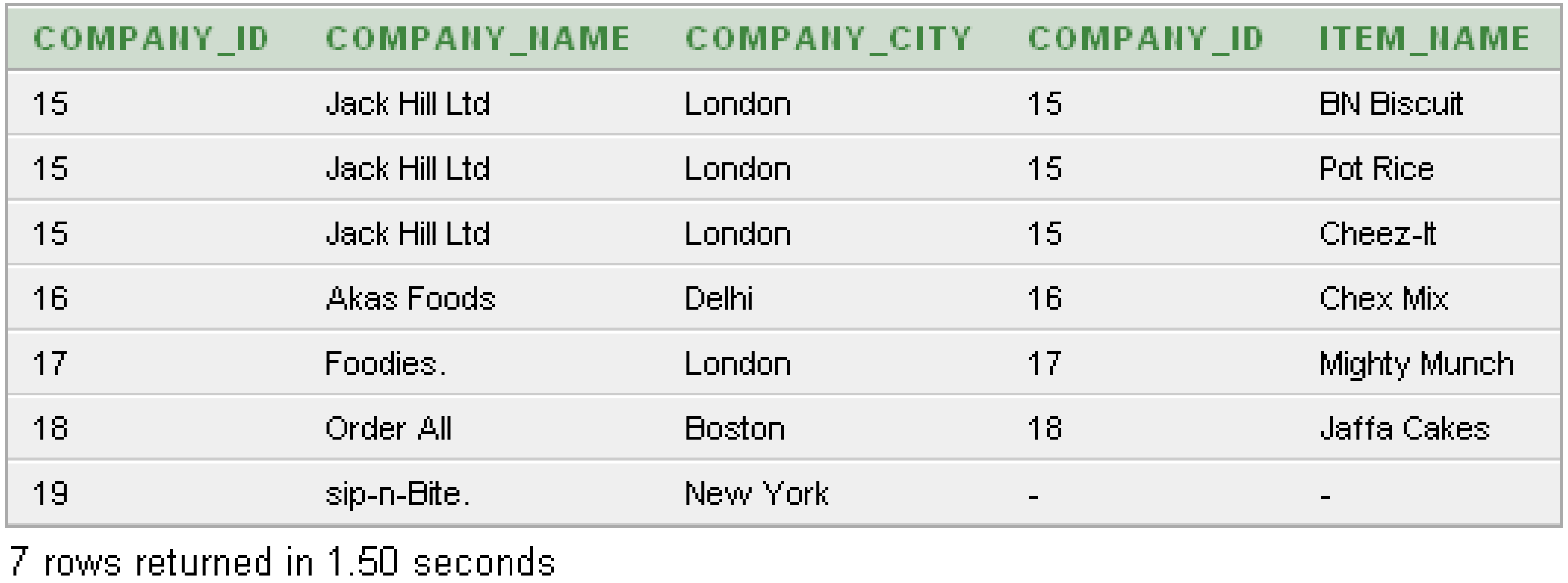
```
SELECT    a.column1, b.column2
FROM      table1 a LEFT [OUTER] JOIN table2
 b
ON        a.columnpk = b.columnfk;
```

- Oracle Example (left join):

```
SELECT    a.column1, b.column2
FROM      table1 a, table2 b
WHERE     a.columnpk = b.columnfk(+);
```

LEFT OUTER JOIN

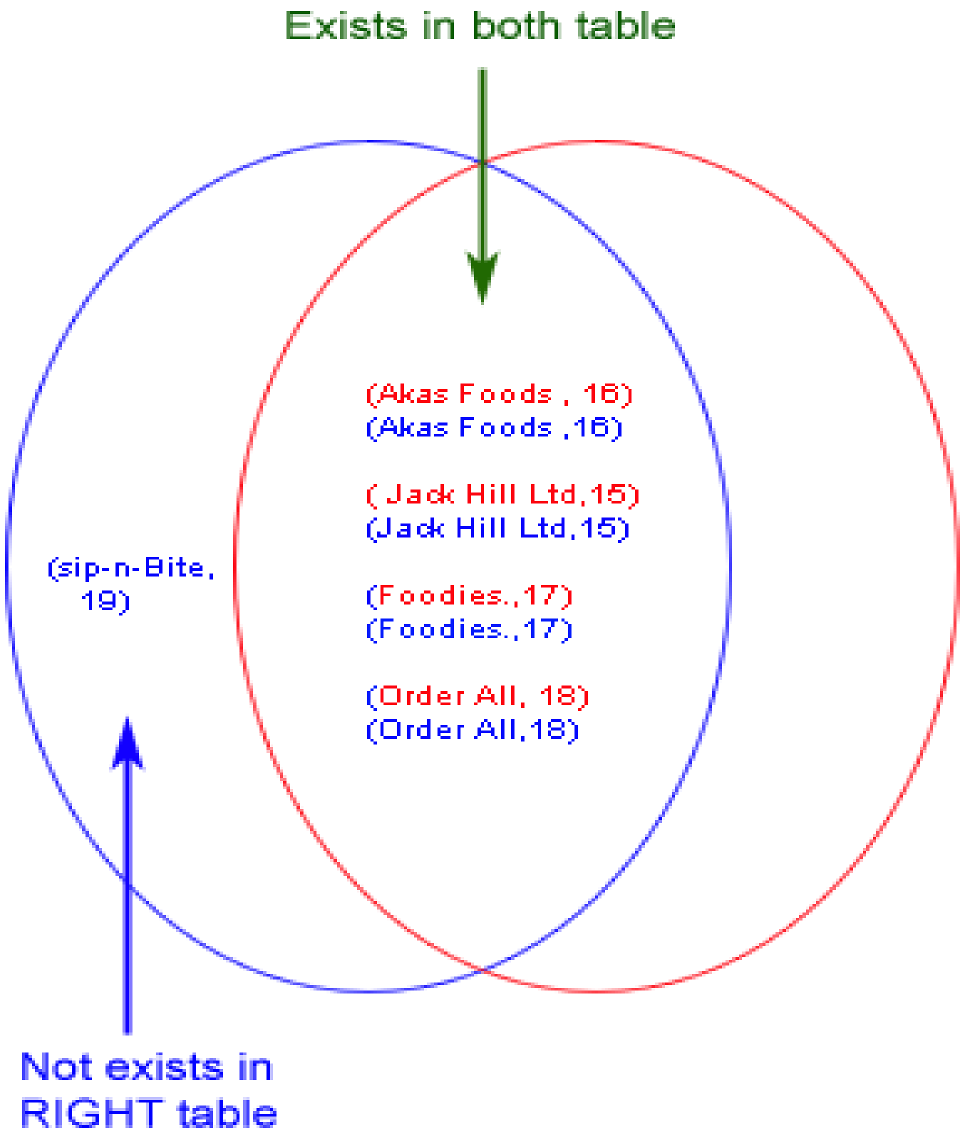
-
  ```
  SELECT c.company_id,c.company_name,  c.company_city,
  f.company_id, f.item_name
  FROM   company  c LEFT OUTER JOIN food f
                  ON c.company_id = f.company_id;
  ```



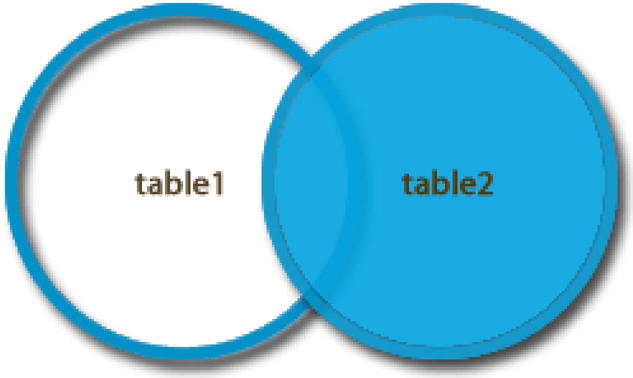
COMPANY_ID	COMPANY_NAME	COMPANY_CITY	COMPANY_ID	ITEM_NAME
15	Jack Hill Ltd	London	15	BN Biscuit
15	Jack Hill Ltd	London	15	Pot Rice
15	Jack Hill Ltd	London	15	Cheez-It
16	Akas Foods	Delhi	16	Chex Mix
17	Foodies.	London	17	Mighty Munch
18	Order All	Boston	18	Jaffa Cakes
19	sip-n-Bite.	New York	-	-

7 rows returned in 1.50 seconds

LEFT OUTER JOIN



Right Outer Join



- ANSI SQL Example:

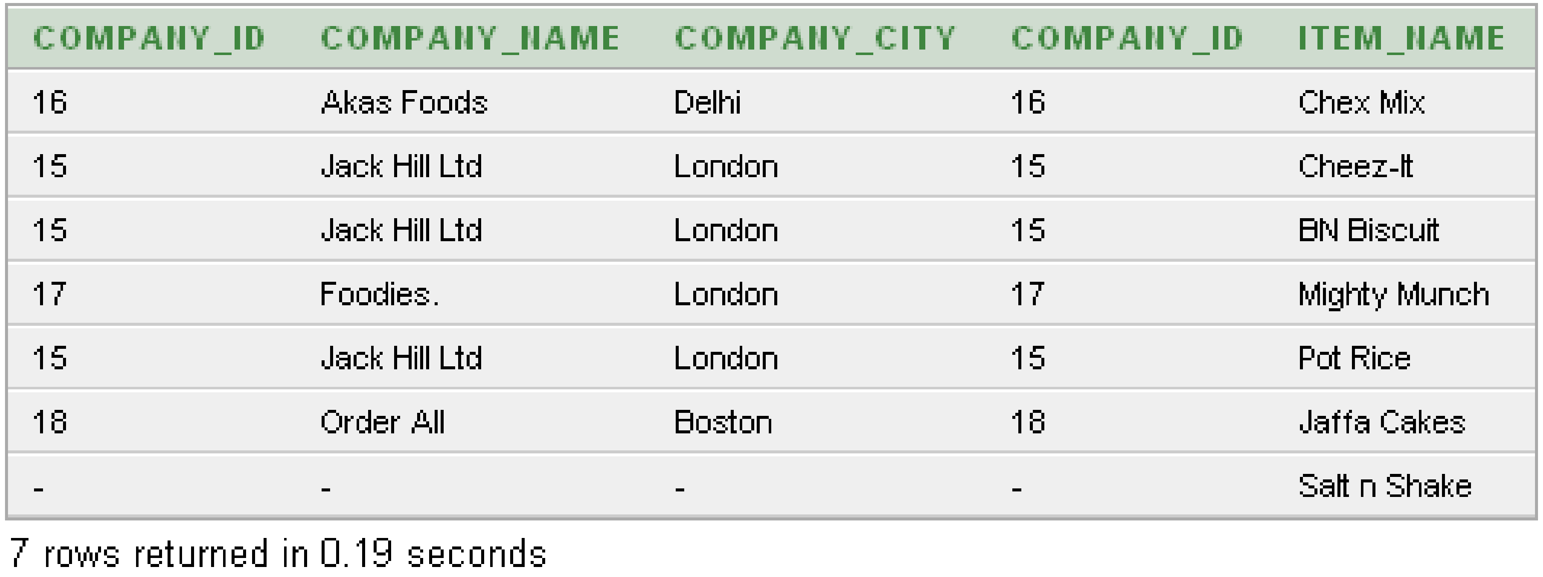
```
SELECT    a.column1, b.column2
FROM      table1 a RIGHT [OUTER] JOIN table2 b
ON        a.columnpk = b.columnfk;
```

- Oracle Example (left join):

```
SELECT    a.column1, b.column2
FROM      table1 a, table2 b
ON        a.columnpk(+) = b.columnfk;
```

RIGHT OUTER JOIN

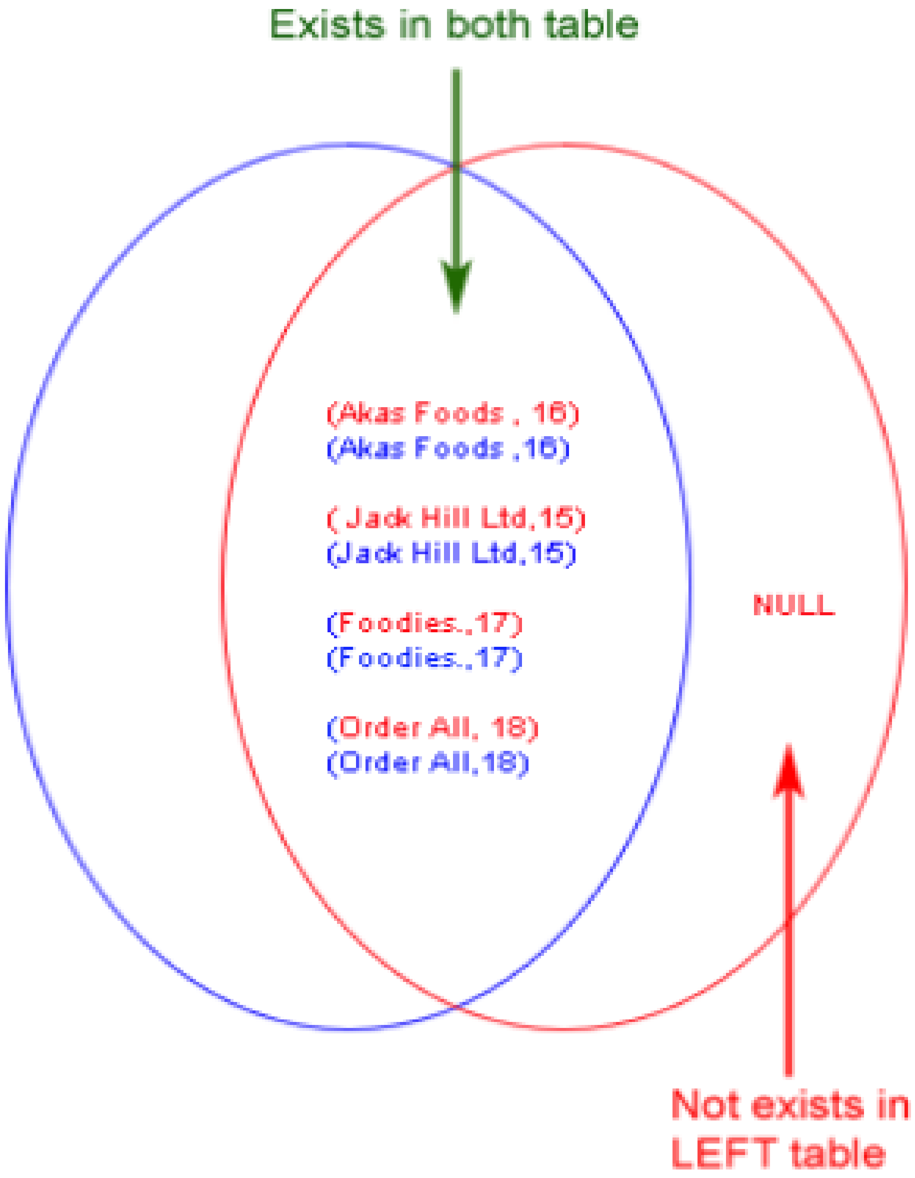
- SELECT c.company_id,c.company_name, c.company_city, f.company_id, f.item_name
  FROM company c RIGHT OUTER JOIN food f
  ON c.company_id = f.company_id;



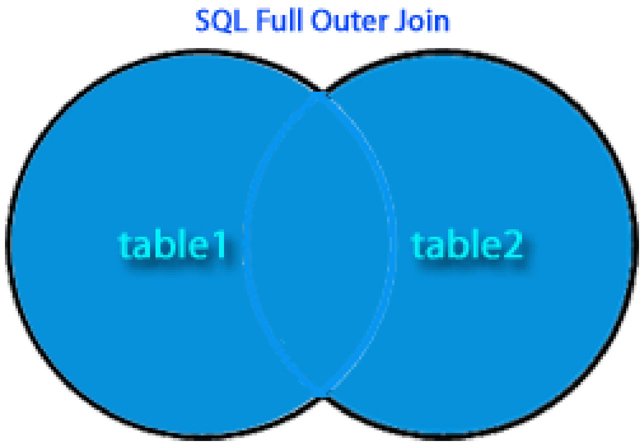
COMPANY_ID	COMPANY_NAME	COMPANY_CITY	COMPANY_ID	ITEM_NAME
16	Akas Foods	Delhi	16	Chex Mix
15	Jack Hill Ltd	London	15	Cheez-It
15	Jack Hill Ltd	London	15	BN Biscuit
17	Foodies.	London	17	Mighty Munch
15	Jack Hill Ltd	London	15	Pot Rice
18	Order All	Boston	18	Jaffa Cakes
-	-	-	-	Salt n Shake

7 rows returned in 0.19 seconds

RIGHT OUTER JOIN



Full Outer Join



- A match that includes all matches between two tables plus all non-matches whether on the left or right side of a join.

- SQL Example:

```
SELECT    a.column1, b.column2
FROM      table1 a FULL OUTER JOIN table2
 b
ON        a.columnpk = b.columnfk;
```

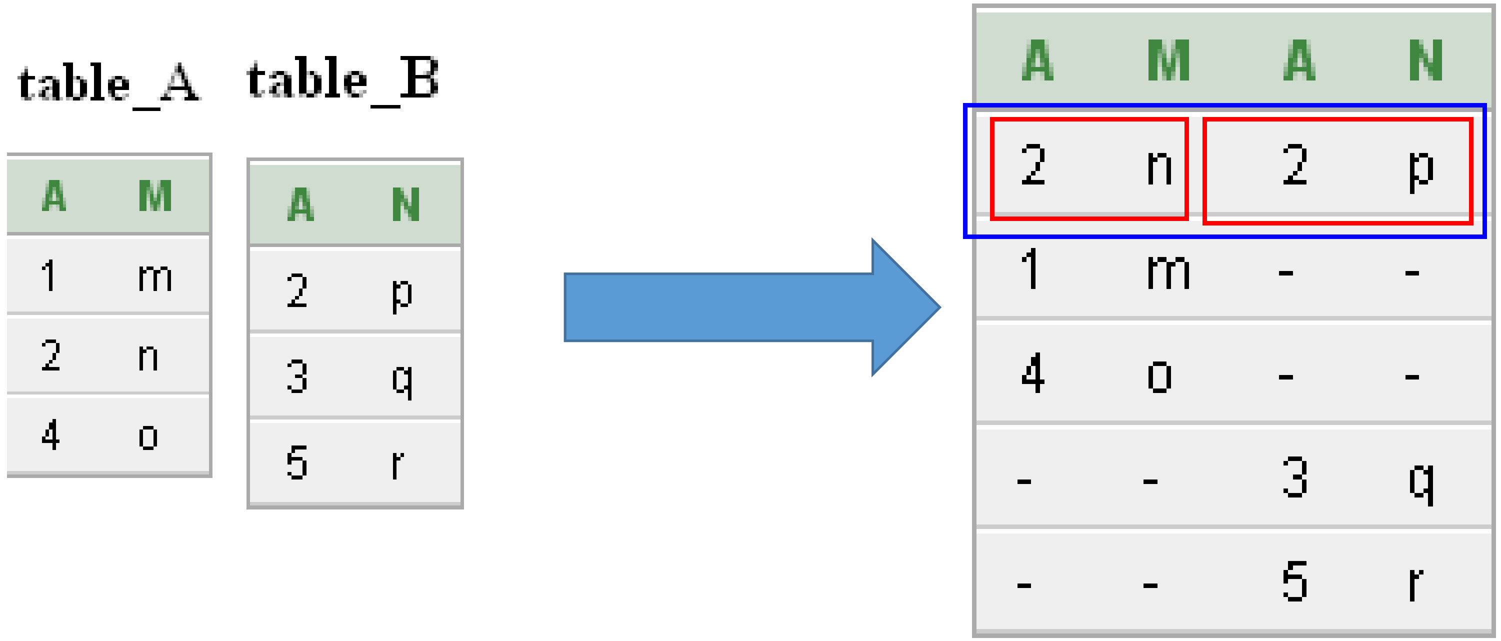
- Oracle syntax: The **UNION** operator to mimic the behavior.

Full Outer Join - Example

- SELECT * FROM

  table_A FULL OUTER JOIN table_B

  ON table_A.A=table_B.A;



table_A

A	M
1	m
2	n
4	o

table_B

A	N
2	p
3	q
5	r

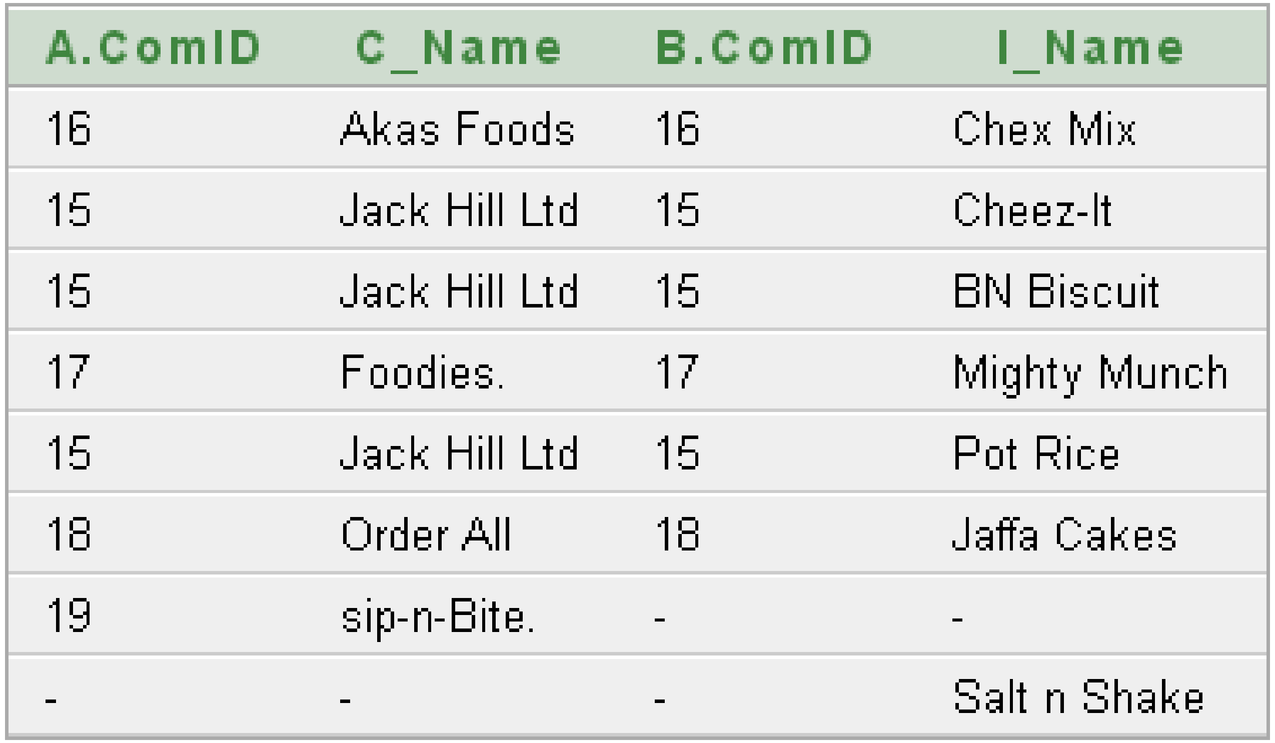
A	M	A	N
2	n	2	p
1	m	-	-
4	o	-	-
-	-	3	q
-	-	5	r

Full OUTER JOIN

- SELECT a.company_id AS "a.ComID", a.company_name AS "C_Name", b.company_id AS "b.ComID", b.item_name AS "I_Name"

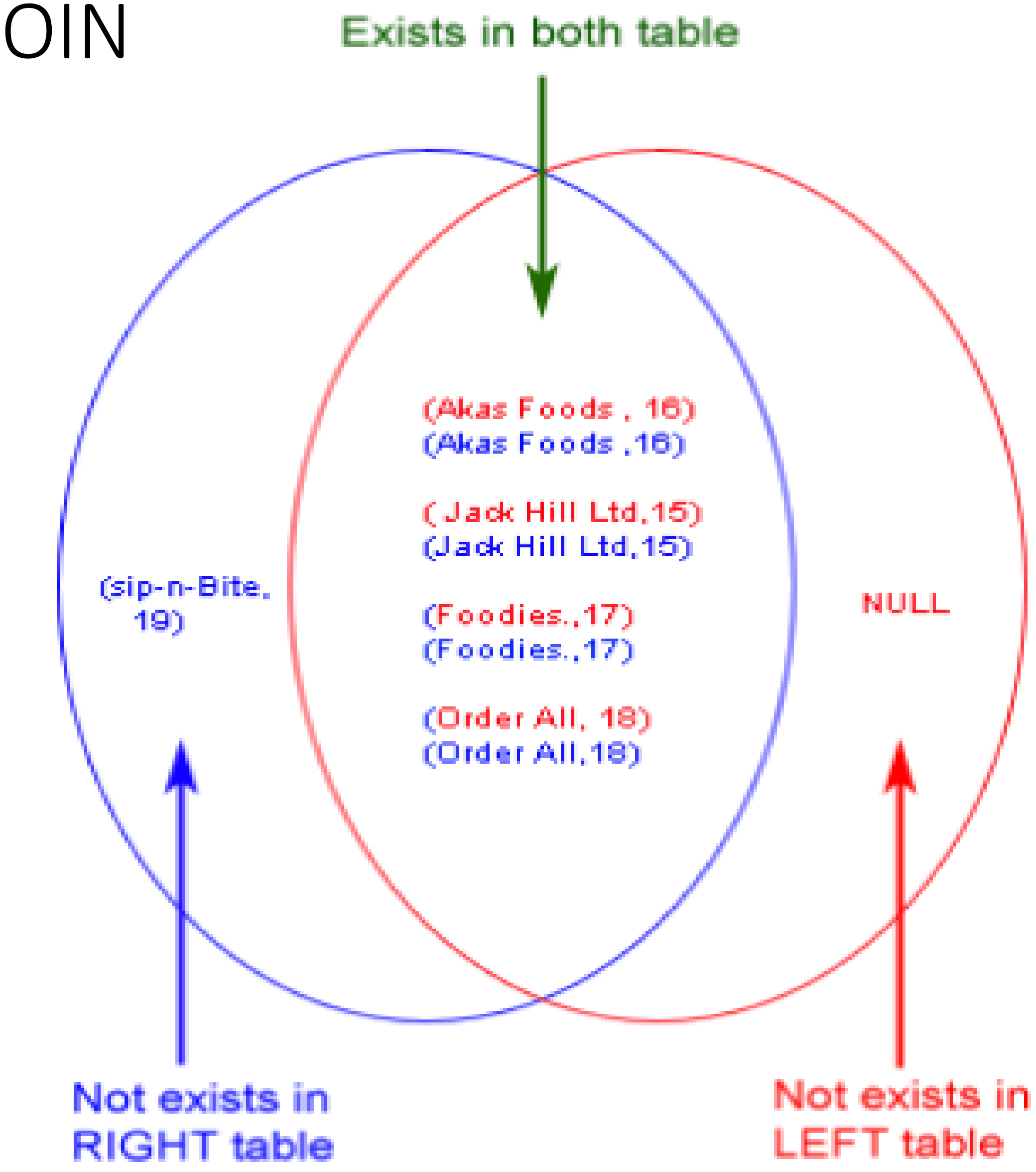
  FROM company a FULL OUTER JOIN foods b

  ON a.company_id = b.company_id;



A.ComID	C_Name	B.ComID	I_Name
16	Akas Foods	16	Chex Mix
15	Jack Hill Ltd	15	Cheez-It
15	Jack Hill Ltd	15	BN Biscuit
17	Foodies.	17	Mighty Munch
15	Jack Hill Ltd	15	Pot Rice
18	Order All	18	Jaffa Cakes
19	sip-n-Bite.	-	-
-	-	-	Salt n Shake

Full OUTER JOIN



Full Outer Join

- The combination of LEFT OUTER JOIN and RIGHT OUTER JOIN and combined by, using UNION clause

```
SELECT   a.column1, b.column2
FROM     table1 a LEFT [OUTER] JOIN table2 b
ON    a.columnpk = b.columnfk
UNION
SELECT   a.column1, b.column2
FROM     table1 a RIGHT [OUTER] JOIN table2 b
ON    a.columnpk = b.columnfk;
```

Full Outer Join – oracle example

```
SELECT    a.column1, b.column2
FROM      table1 a, table2 b
WHERE     a.columnpk(+) = b.columnfk
UNION
SELECT    a.column1, b.column2
FROM      table1 a, table2 b
WHERE     a.columnpk = b.columnfk(+);
```

Outer join

- e.g. List the customer name, ID number, and order number for all customers listed in the CUSTOMER table. Include customer information even if there is no order available for that customer

```
SELECT CUSTOMER_T.CUSTOMER_ID, CUSTOMER_NAME, ORDER_ID
       FROM CUSTOMER_T LEFT OUTER JOIN ORDER_T
       ON CUSTOMER_T.CUSTOMER_ID =
                          ORDER_T.CUSTOMER_ID
```

- The syntax LEFT OUTER JOIN was selected because the CUSTOMER_T table was named first, and it is the table from which we wish all rows returned (regardless of whether there is a matching order in the ORDER_T table)

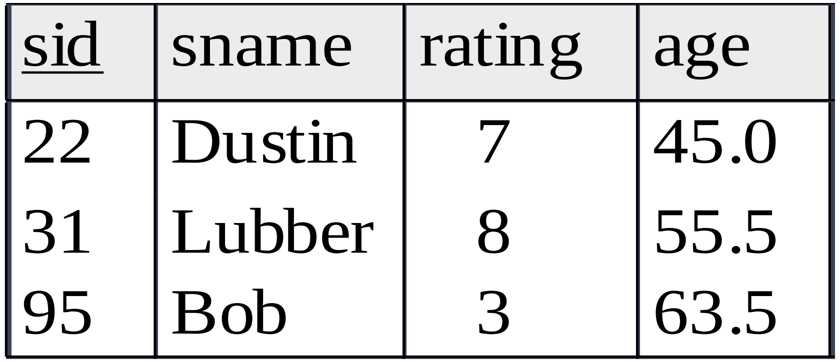
Outer join

- e.g. List the customer name, ID number, and order number for all orders listed in the ORDER table. Include order number even if there is no customer name and identification number available

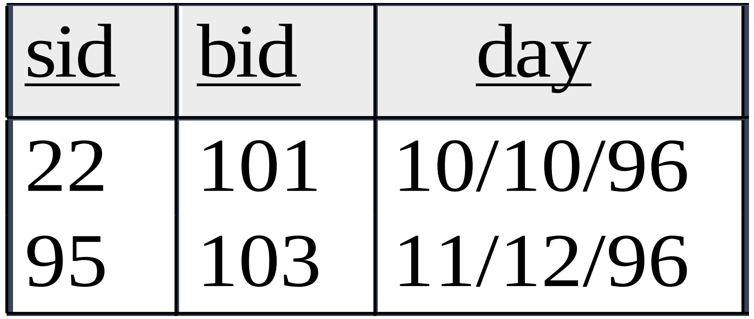
```
SELECT CUSTOMER_T.CUSTOMER_ID, CUSTOMER_NAME, ORDER_ID
       FROM CUSTOMER_T RIGHT OUTER JOIN ORDER_T
       ON CUSTOMER_T.CUSTOMER_ID =
                                 ORDER_T.CUSTOMER_ID
```

LEFT OUTER JOIN

SELECT s.sid, s.name, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid



sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5



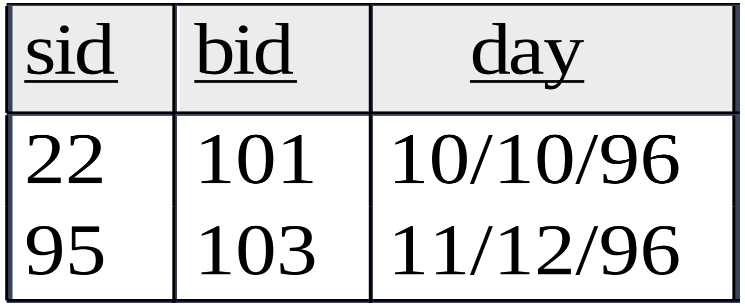
sid	bid	day
22	101	10/10/96
95	103	11/12/96

s.sid	s.name	r.bid
22	Dustin	101
95	Bob	103
31	Lubber	

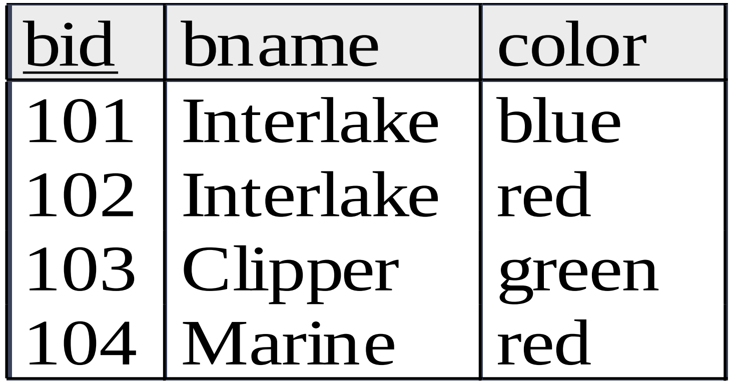
Returns all sailors & information on whether they have reserved boats

RIGHT OUTER JOIN

```
SELECT r.sid, b.bid, b.name
FROM Reserves r RIGHT OUTER JOIN Boats b
ON r.bid = b.bid
```



sid	bid	day
22	101	10/10/96
95	103	11/12/96



bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

r.sid	b.bid	b.name
22	101	Interlake
	102	Interlake
95	103	Clipper
	104	Marine

Returns all boats & information on which ones are reserved.

FULL OUTER JOIN

```
SELECT r.sid, b.bid, b.name
FROM Reserves r FULL OUTER JOIN Boats b
ON r.bid = b.bid
```

sid	bid	day
22	101	10/10/96
95	103	11/12/96

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

r.sid	b.bid	b.name
22	101	Interlake
	102	Interlake
95	103	Clipper
	104	Marine

Returns all boats & all information on reservations

SQL Joins

Defining Join Types: Self Join

- A SELF JOIN is another type of join in sql which is used to join a table to itself,
  - specially when the table has a FOREIGN KEY which references its own PRIMARY KEY.
- A recursive join internally within a single table based on a primary and foreign key residing in each row of data in a table.
- You must use table name aliases to create a ***SELF JOIN***.
- Self joins typically use two separate column names.

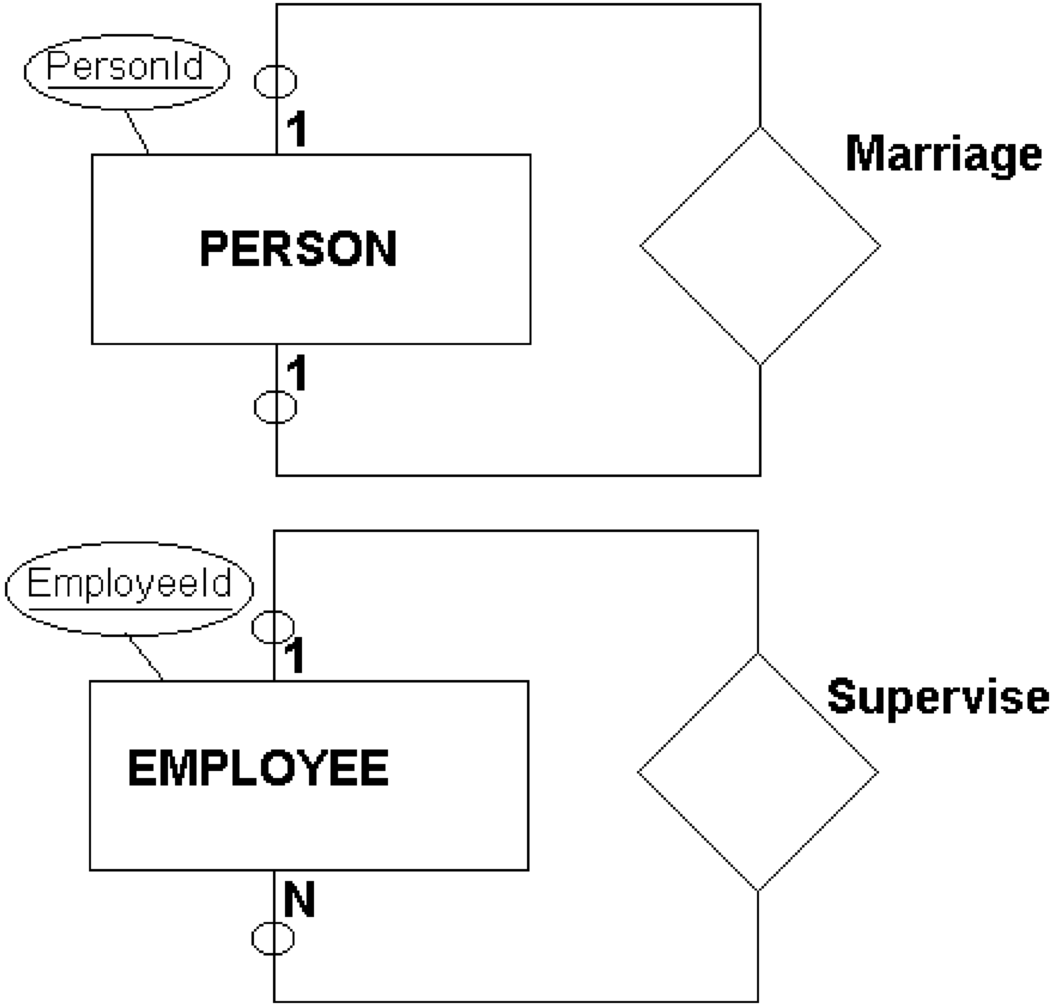
SQL Joins

Defining Join Types: Self Join

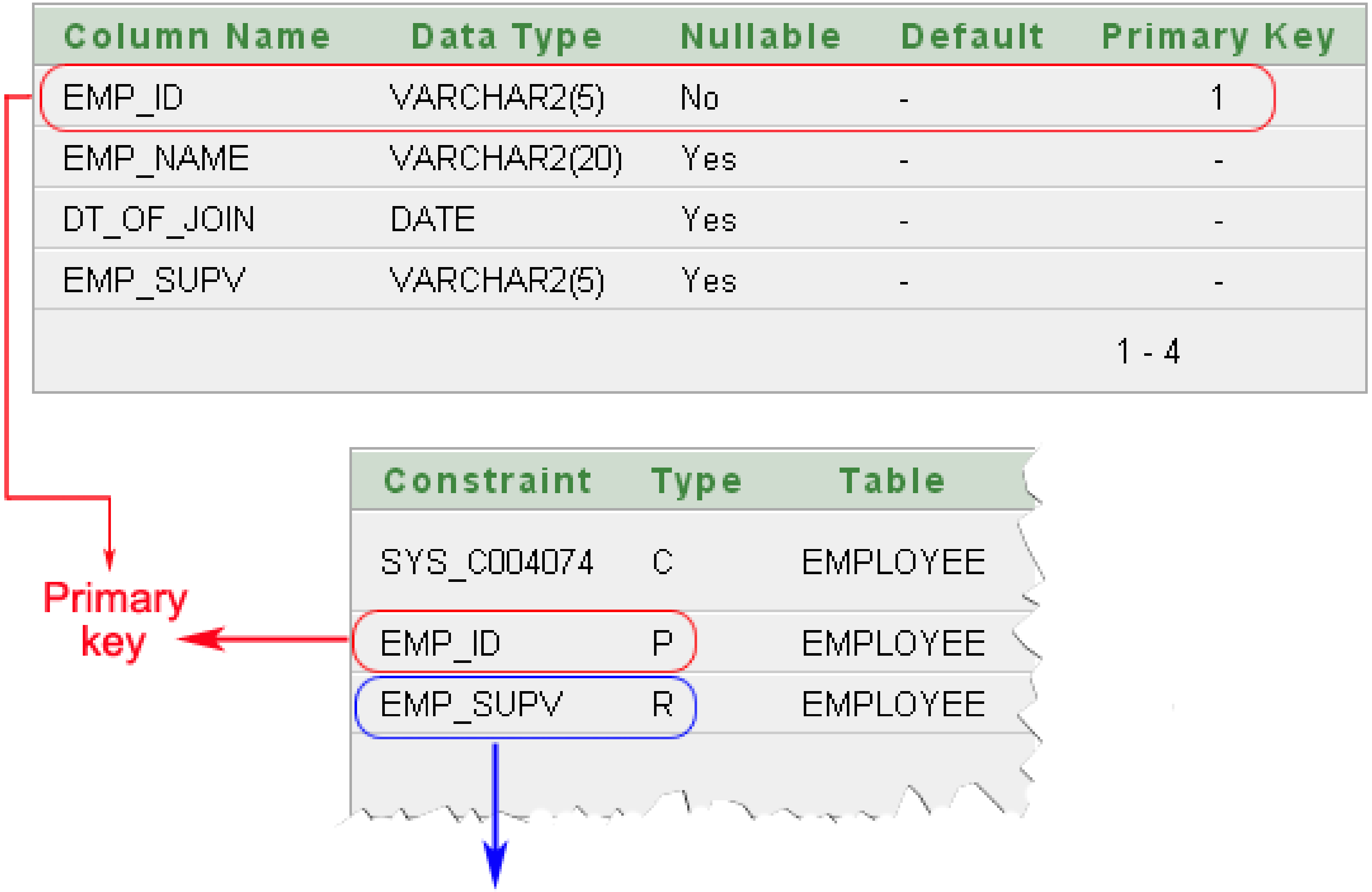
- Example:

```
SELECT    a.column1, b.column2
FROM      table1 a [INNER] JOIN table1 b
ON        a.columnpk = b.columnfk;
----------------------------------------------------------------------
                                --
SELECT    a.column1, b.column2
FROM      table1 a, table1 b
WHERE     a.columnpk = b.columnfk;
```

Self Join - Unary Relationship In Database



The structure of the table



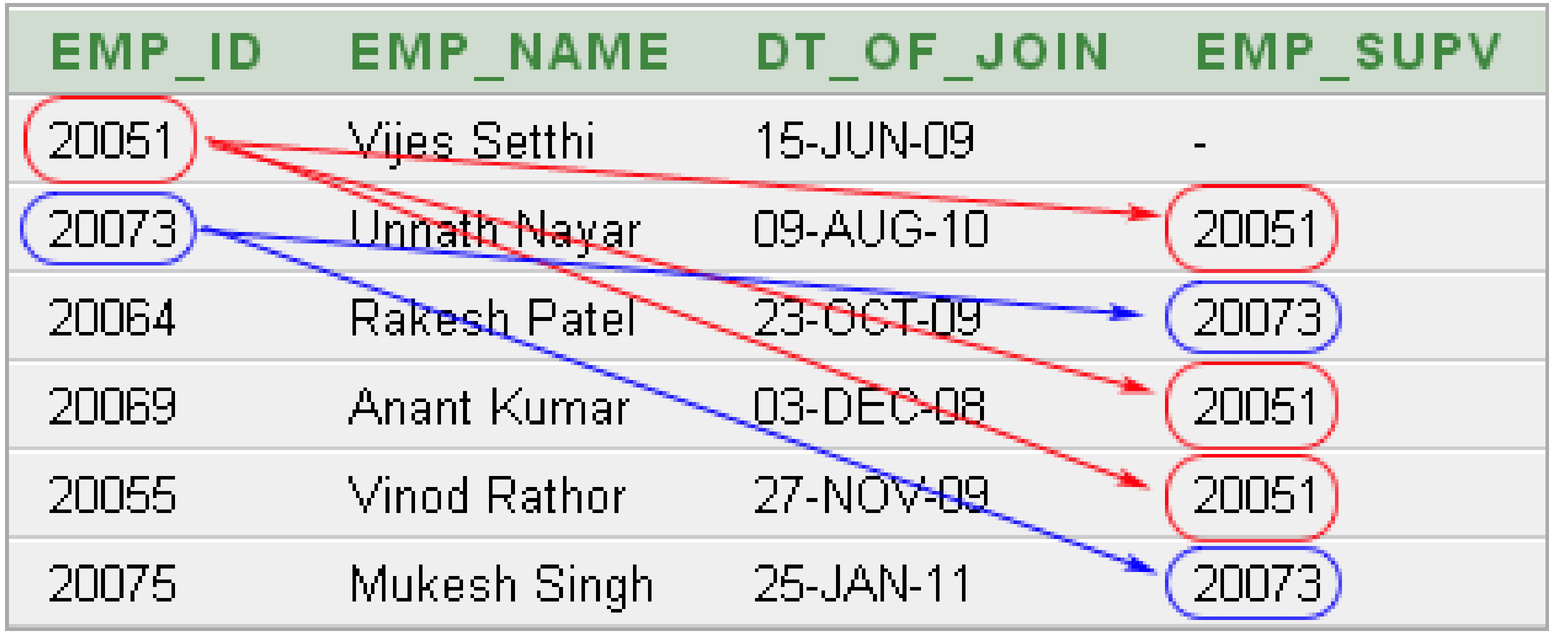
Column Name	Data Type	Nullable	Default	Primary Key
EMP_ID	VARCHAR2(5)	No	-	1
EMP_NAME	VARCHAR2(20)	Yes	-	-
DT_OF_JOIN	DATE	Yes	-	-
EMP_SUPV	VARCHAR2(5)	Yes	-	-
				1 - 4

Constraint	Type	Table
SYS_C004074	C	EMPLOYEE
EMP_ID	P	EMPLOYEE
EMP_SUPV	R	EMPLOYEE

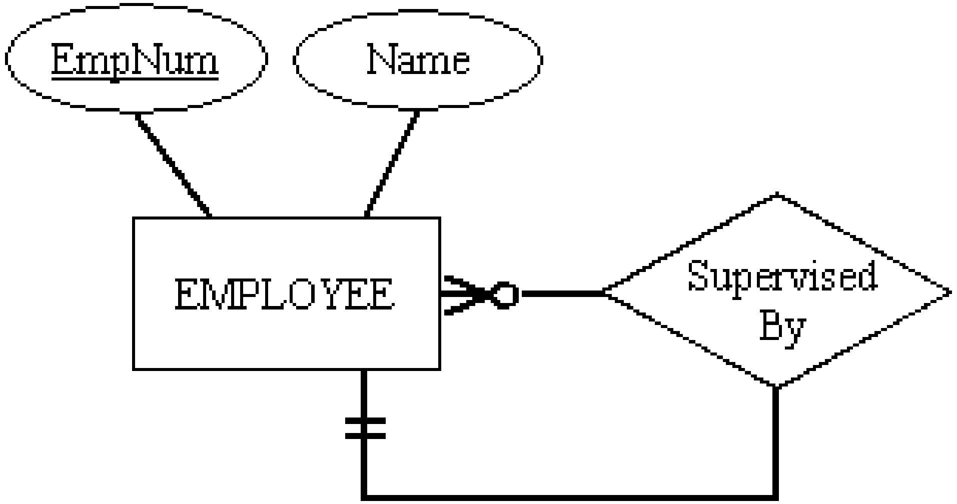
Primary key

Foreign key
Referencing EMP_ID of this table

Unary relationship to employee

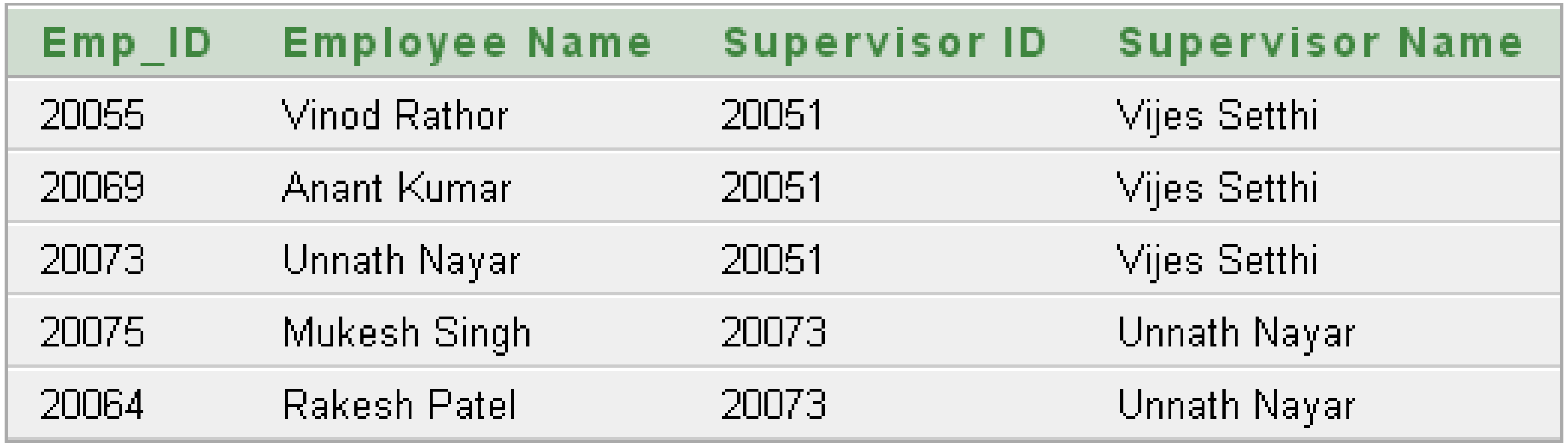


EMP_ID	EMP_NAME	DT_OF_JOIN	EMP_SUPV
20051	Vijes Setthi	15-JUN-09	-
20073	Unnath Nayar	09-AUG-10	20051
20064	Rakesh Patel	23-OCT-09	20073
20069	Anant Kumar	03-DEC-08	20051
20055	Vinod Rathor	27-NOV-09	20051
20075	Mukesh Singh	25-JAN-11	20073



Self Join - Example

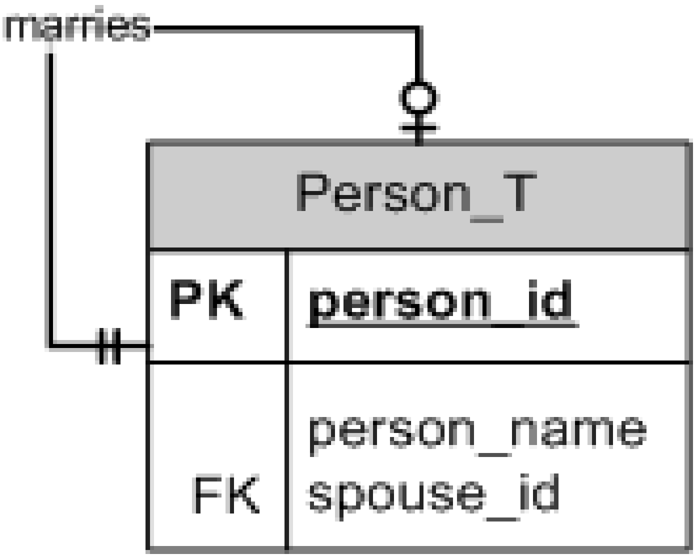
```
• SELECT    a.emp_id AS "Emp_ID",
         a.emp_name AS "Employee Name",
             b.emp_id AS "Supervisor ID",
              b.emp_name AS "Supervisor Name"
        FROM employee a, employee b
                        WHERE a.emp_id = b. emp_supv;
```



Emp_ID	Employee Name	Supervisor ID	Supervisor Name
20055	Vinod Rathor	20051	Vijes Setthi
20069	Anant Kumar	20051	Vijes Setthi
20073	Unnath Nayar	20051	Vijes Setthi
20075	Mukesh Singh	20073	Unnath Nayar
20064	Rakesh Patel	20073	Unnath Nayar

Self Join - Example

- Display the persons’ name along with their spouse name.



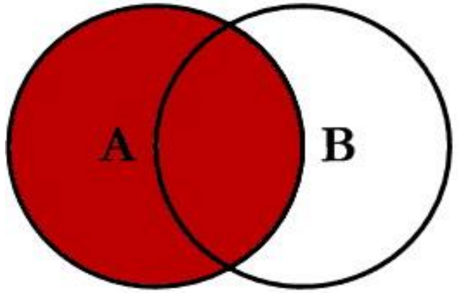
marries

Person_T

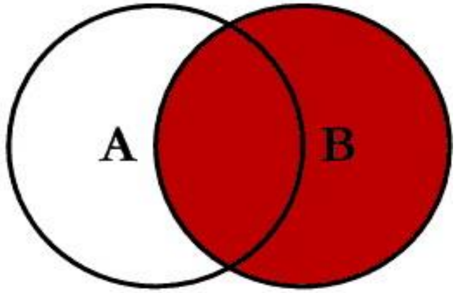
PK	person_id
 FK	person_name spouse_id

```
SELECT   p.person_name as “Person Name”,
          s.person_name as “Spouse Name”
FROM     Person p, Person s
WHERE  p.person_id = s.spouse_id
```

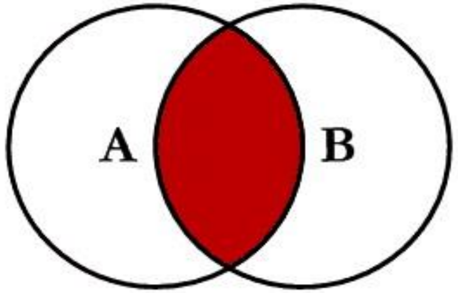
SQL JOINS



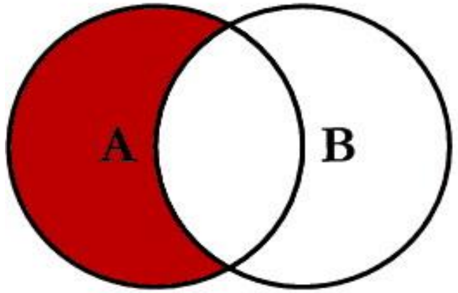
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



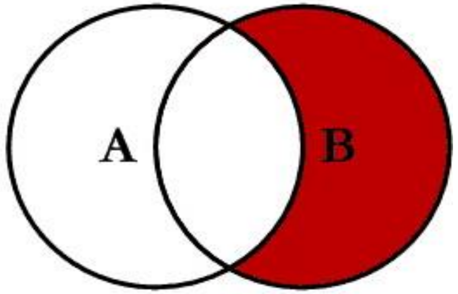
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



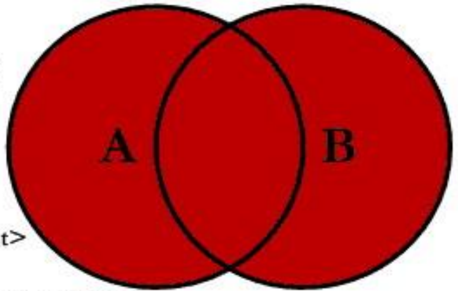
```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



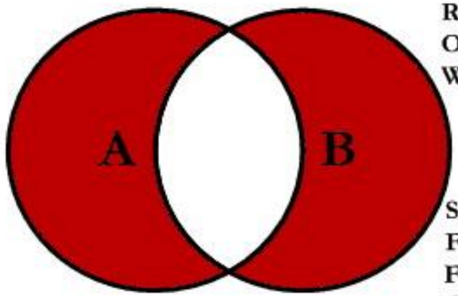
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Introduction

- Querying one table already done & practiced!
- Real power of relational database
  - Storage of data in multiple tables
  - Necessitates creating queries to use multiple tables
- Two Basic approaches for processing multiple tables
  - Sub-queries
  - Join

Processing Multiple Tables Using Sub-queries

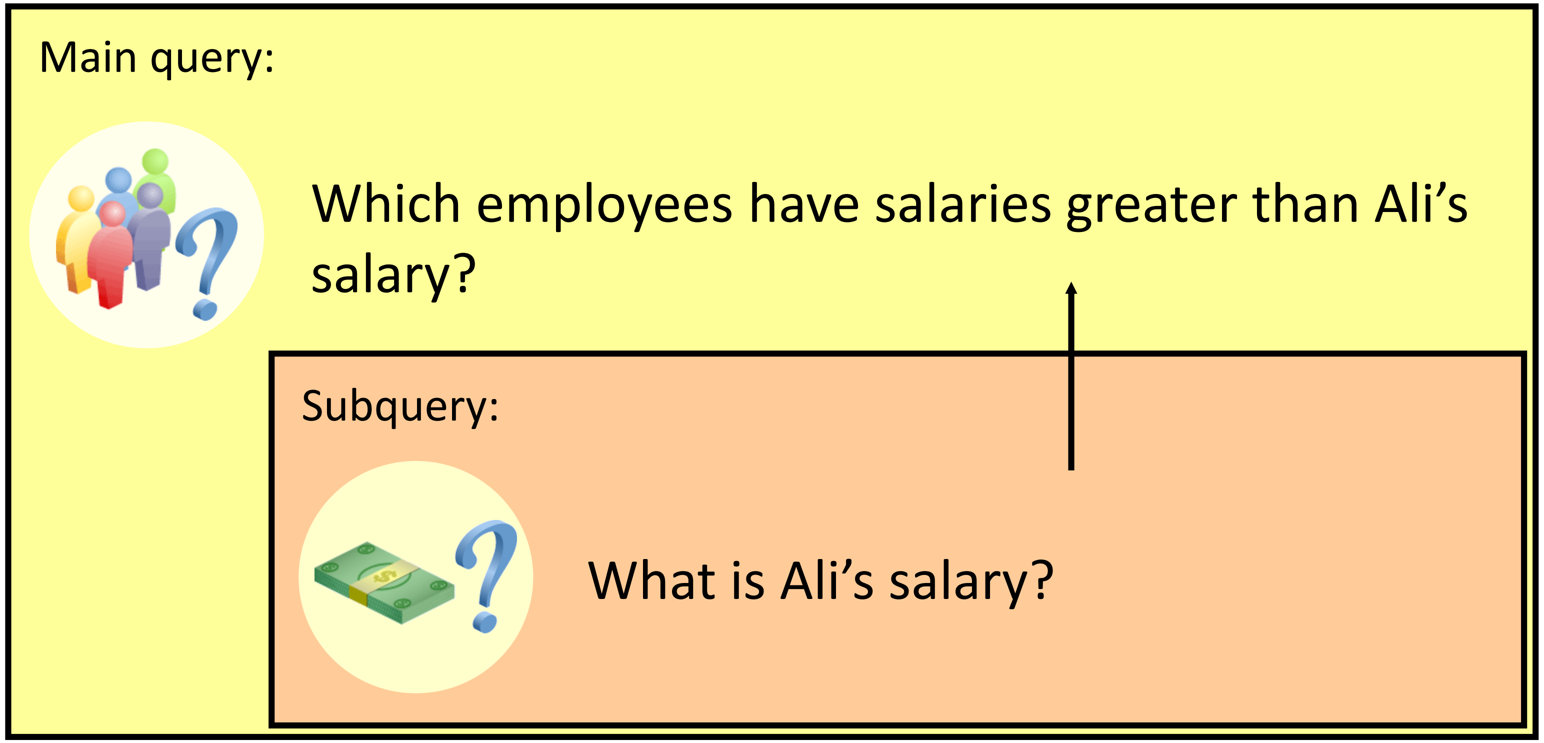
- A *subquery* is a query within a query.
- Subqueries enable you to write queries that select data rows for criteria that are actually developed while the query is executing at *run time*.
- Subquery – placing an inner query (SELECT statement) inside an outer query
  - Inner query provides a set of one or more values for outer query

Processing Multiple Tables Using Sub-queries

- One of the two basic approaches to process multiple tables
  - Different people will have different preferences about which technique to use
  - Joining is useful when data from several tables are to be retrieved and displayed
  - Subquery when data from tables in outer query are to be displayed only

Using a Subquery to Solve a Problem

- Who has a salary greater than Ali's?



Main query:

Which employees have salaries greater than Ali's salary?

Subquery:

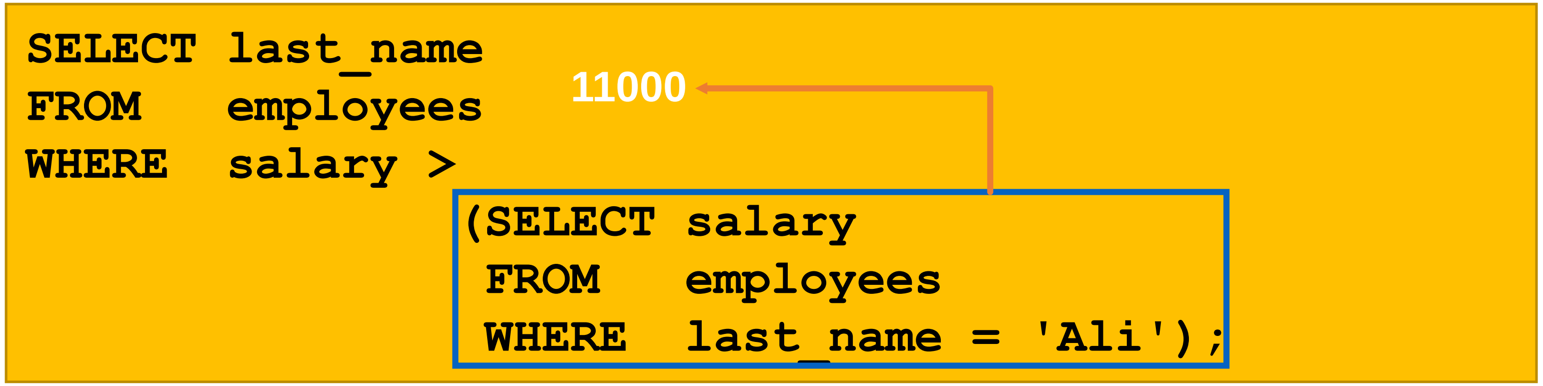
What is Ali's salary?

Subquery Syntax

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

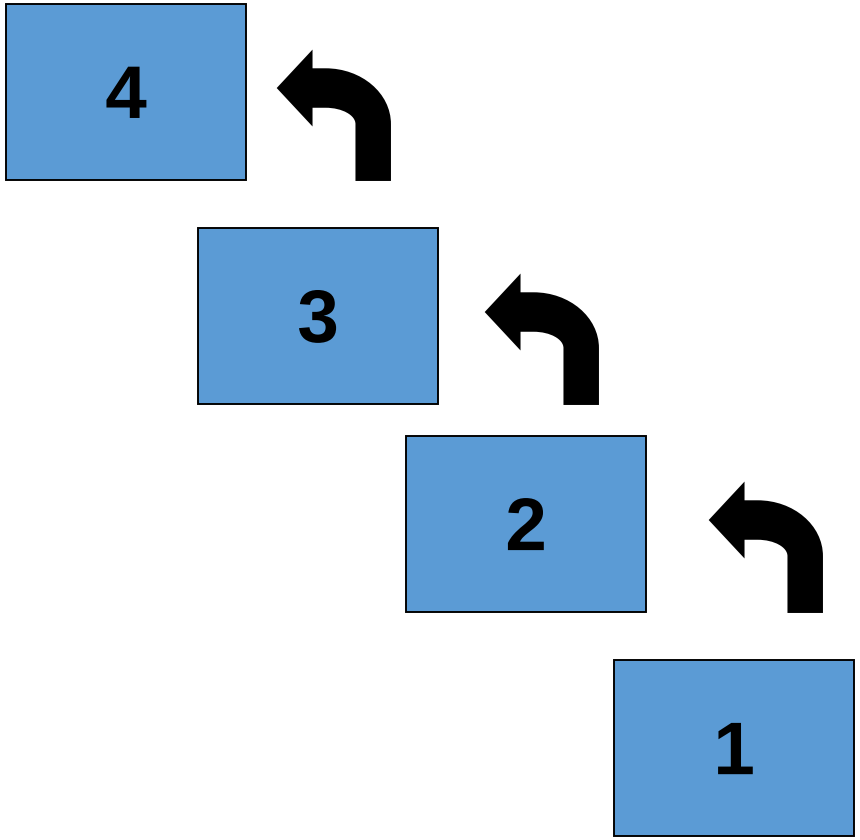
- The subquery (inner query) executes once before the main query (outer query).
- The result of the subquery is used by the main query.

Using a Sub-query



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

The basic concept is to pass a single value or many values from the subquery to the next query and so on.



When reading or writing SQL subqueries, you should start from the bottom upwards, working out which data is to be passed to the next query up.

Subquery Types

- There are three basic types of subqueries.

1. Subqueries that operate on lists by use of the IN operator or with a comparison operator.
    - These subqueries can return a group of values, but the values must be from a single column of a table.

SUBQUERY TYPES

2. Subqueries that use an unmodified comparison operator (=, <, >, <>)
   - these subqueries must return only a single, *scalar* value.
3. Subqueries that use the EXISTS operator to test the *existence* of data rows satisfying specified criteria.

Guidelines for Using Subqueries

- Enclose subqueries in parentheses.
- Place subqueries on the right side of the comparison condition.
- The `ORDER BY` clause in the subquery is not needed.
  - Subqueries cannot manipulate their results internally.
- Use single-row operators with single-row subqueries, and use multiple-row operators with multiple-row subqueries.

Sub-Queries Example

-
  ```
  SELECT CUSTOMER_NAME  FROM CUSTOMER_T, ORDER_T
   WHERE CUSTOMER_T.CUSTOMER_ID = ORDER_T.CUSTOMER_ID
                          AND      ORDER_ID = 1008;
  ```

-
  ```
  SELECT CUSTOMER_NAME  FROM CUSTOMER_T
     WHERE  CUSTOMER_ID =
          (SELECT CUSTOMER_ID  FROM ORDER_T
                         WHERE ORDER_ID = 1008);
  ```

SUBQUERIES AND THE IN Operator

- Subqueries that are introduced with the keyword **IN** take the general form:
    - WHERE expression [NOT] IN (subquery)
- The only difference in the use of the IN operator with subqueries is that the list does not consist of *hard-coded* values.

SUBQUERIES AND COMPARISON OPERATORS

- The general form of the WHERE clause with a comparison operator is similar to that used thus far in the text.
- Note that the subquery is again enclosed by parentheses.

```
WHERE <expression> <comparison_operator> (subquery)
```

SUBQUERIES AND COMPARISON OPERATORS

- The most important point to remember when using a subquery with a comparison operator is that the subquery can only return a single or *scalar* value.
- This is also termed a *scalar subquery* because a single column of a single row is returned by the subquery.

To identify the students who have failed in course CSC273

```
Select student_id
From marks
Where course_id = ‘CSC273’
And grade < 40;
```

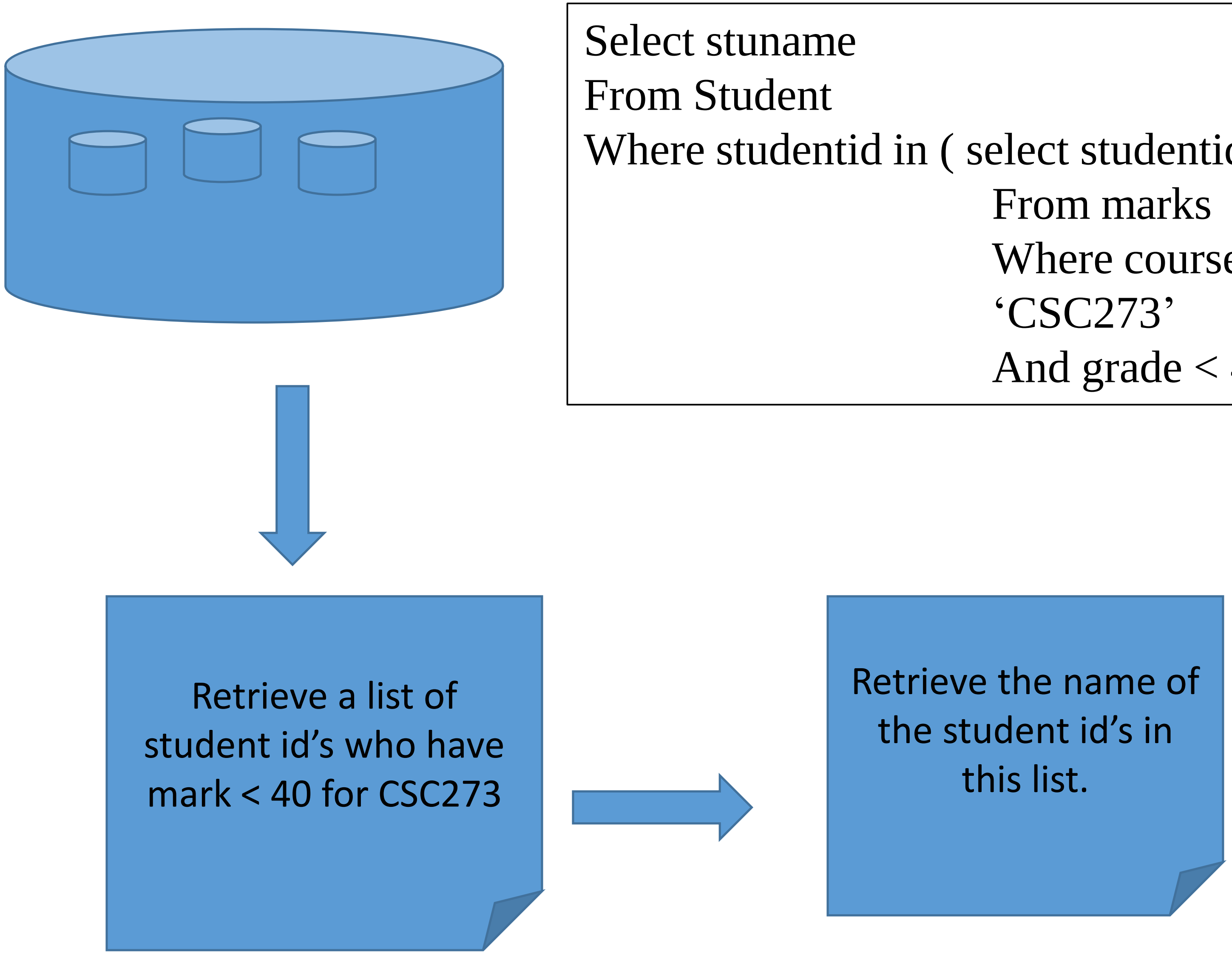
If we want to retrieve a name based on a student id

```
Select stu_name
From student
Where student_id = 9292145;
```

```
Select stu_name
From Student
Where student_id in ( select student_id
                        From marks
                        Where course_id = ‘CSC273’
                        And grade < 40);
```

Why use IN?

```
Select stuname
From Student
Where studentid in ( select studentid
                     From marks
                     Where courseid =
                     'CSC273'
                     And grade < 40);
```



Retrieve a list of student id’s who have mark < 40 for CSC273

Retrieve the name of the student id’s in this list.

Subquery Example

- Show all customers who have placed an order

Many programmers simply use IN even if equal sign (=) would also work

The IN operator will test to see if the CUSTOMER_ID value of a row is included in the list returned from the subquery

```
SELECT CUSTOMER_NAME    FROM CUSTOMER_T
WHERE CUSTOMER_ID  IN
          (SELECT DISTINCT CUSTOMER_ID FROM ORDER_T);
```

Subquery is embedded in parentheses. In this case it returns a list that will be used in the WHERE clause of the outer query

SUBQUERIES AND COMPARISON OPERATORS

- If we substitute this query as a subquery in another SELECT statement, then that SELECT statement will fail.
- This is demonstrated in the next SELECT statement. Here the SQL code will fail because the subquery uses the greater than (>) comparison operator and the subquery returns multiple values.

```
SELECT emp_ssn
FROM employee
   WHERE emp_salary >
   (SELECT emp_salary
    FROM employee
       WHERE emp_salary > 40000);
```

Aggregate Functions and Comparison Operators

- The aggregate functions (AVG, SUM, MAX, MIN, and COUNT) always return a *scalar* result table.
- Thus, a subquery with an aggregate function as the object of a comparison operator will always execute provided you have formulated the query properly.

Aggregate Functions and Comparison Operators

```
SELECT emp_last_name "Last Name",
   emp_first_name "First Name",
   emp_salary "Salary"
FROM employee
WHERE emp_salary >
       (SELECT AVG(emp_salary)
        FROM employee);
```

```
Last Name    First Name   Salary
--------------- --------------- ----------
Bordoloi      Bijoy          $55,000
Joyner        Suzanne       $43,000
Zhu           Waiman       $43,000
Joshi         Dinesh        $38,000
```

Exercise

1. *Write a query that will list the names of who is older than the average student.*

TIP the sub-query needs to select the average age of students this should be used then as a filter.

```
SELECT stu_name
        FROM student
                WHERE age >
                        (SELECT avg(age) FROM student);
```

This will return 25 students of the 74 who are enrolled as being older than the average age.

Comparison Operators Modified with the ALL or ANY Keywords

- The ALL and ANY keywords can modify a comparison operator to allow an outer query to accept multiple values from a subquery.
- The general form of the WHERE clause for this type of query is shown here.

```
WHERE <expression> <comparison_operator> [ALL |
ANY] (subquery)
```

- Subqueries that use these keywords may also include GROUP BY and HAVING clauses.

The ALL Keyword

- The ALL keyword modifies the greater than comparison operator to mean greater than all values.

```
SELECT emp_ssn
FROM employee
  WHERE emp_salary >
   (SELECT emp_salary
    FROM employee
      WHERE emp_salary > 40000);
```

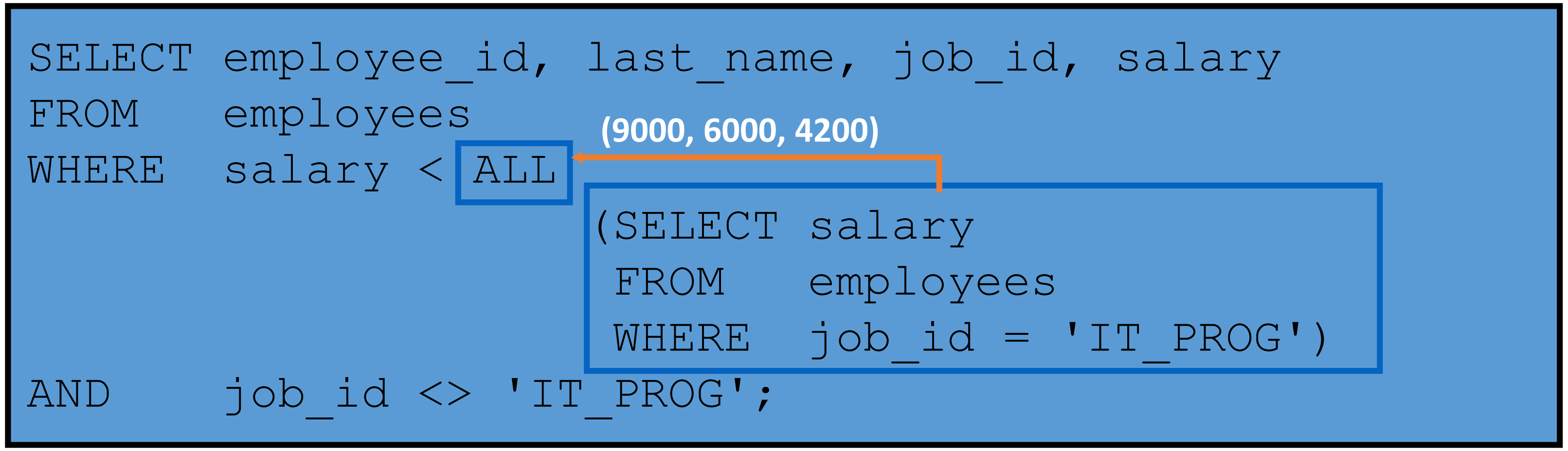
```
SELECT emp_ssn
FROM employee
WHERE emp_salary > ALL
   (SELECT emp_salary
      FROM employee
   WHERE emp_salary > 40000);
```

Using the **ALL** Operator in Multiple-Row Subqueries

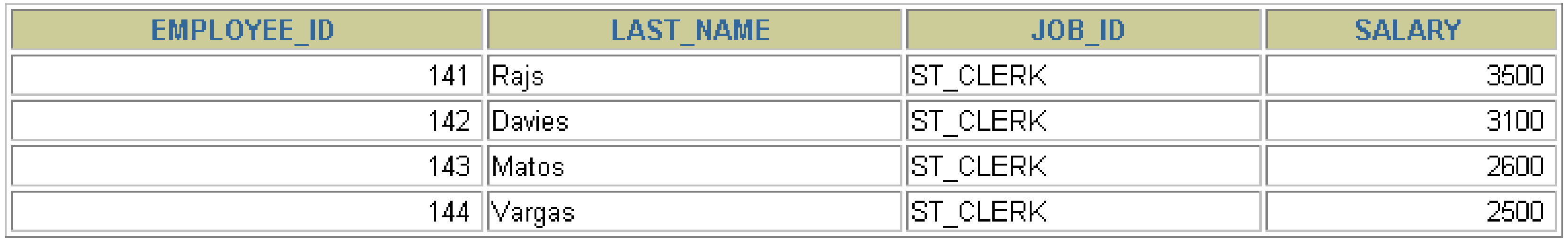
The slide example displays employees whose salary is less than the salary of all employees with a job ID of IT_PROG and whose job is not IT_PROG.

>ALL means more than the maximum, and <ALL means less than the minimum.

The NOT operator can be used with IN, ANY, and ALL operators.



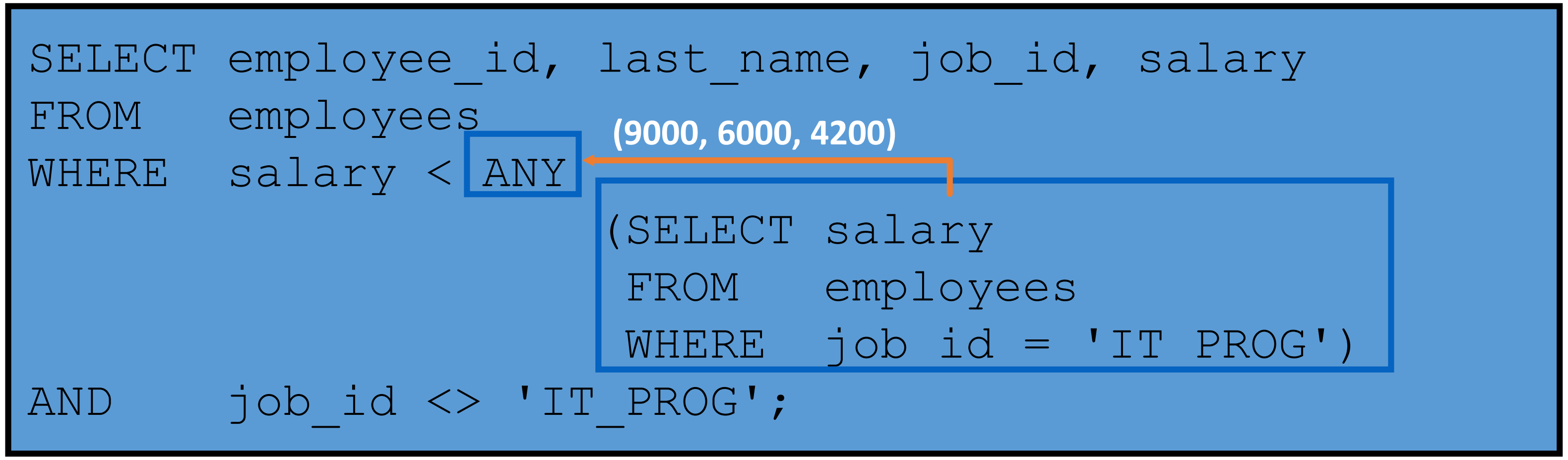
```
SELECT employee_id, last_name, job_id, salary
FROM   employees
WHERE  salary < ALL
                    (SELECT salary
                     FROM   employees
                     WHERE  job_id = 'IT_PROG')
AND    job_id <> 'IT_PROG';
```



EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

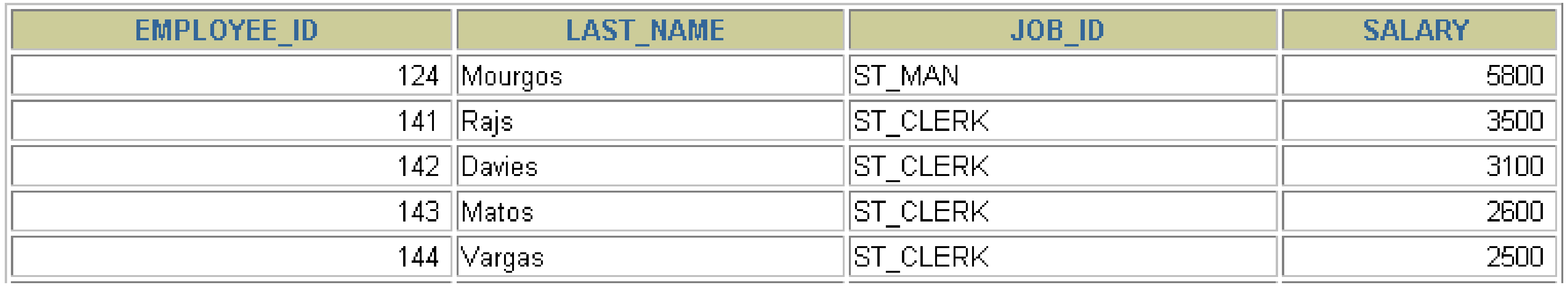
Using the ANY Operator in Multiple-Row Subqueries

The slide example displays employees who are not IT programmers and whose salary is less than that of any IT programmer.
The maximum salary that a programmer earns is $9,000.
<ANY means less than the maximum. >ANY means more than the minimum.



```
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)



EMPLOYEE_ID	LAST_NAME	JOB_ID	SALARY
124	Mourgos	ST_MAN	5800
141	Rajs	ST_CLERK	3500
142	Davies	ST_CLERK	3100
143	Matos	ST_CLERK	2600
144	Vargas	ST_CLERK	2500

An "= ANY" (Equal Any) Example

- The "= ANY" operator is exactly equivalent to the IN operator.
- For example, to find the names of employees that have male dependents, you can use either IN or "= ANY" – both of the queries shown below will produce an identical result table.

```
SELECT emp_last_name "Last Name", emp_first_name "First Name"
FROM employee
WHERE emp_ssn IN
   (SELECT dep_emp_ssn
    FROM dependent
    WHERE dep_gender = 'M');

SELECT emp_last_name "Last Name", emp_first_name "First Name"
FROM employee
WHERE emp_ssn = ANY
   (SELECT dep_emp_ssn
    FROM dependent
    WHERE dep_gender = 'M');
```

A "!= ANY" (Not Equal Any) Example

- The "= ANY" is identical to the IN operator.
- However, the "!= ANY" (not equal any) is **not** equivalent to the NOT IN operator.
- If a subquery of employee salaries produces an intermediate result table with the salaries
  - $38,000, $43,000, and $55,000,
- then the WHERE clause shown here means
  - "NOT $38,000" AND "NOT $43,000" AND "NOT $55,000".

```
WHERE NOT IN (38000, 43000, 55000);
```

- However, the "!= ANY" comparison operator and keyword combination shown in this next WHERE clause means
  - "NOT $38,000" OR "NOT $43,000" OR "NOT $55,000".

MULTIPLE LEVELS OF NESTING

- Subqueries may themselves contain subqueries.
- When the WHERE clause of a subquery has as its object another subquery, these are termed *nested subqueries*.
- Consider the problem of producing a listing of employees that worked more than 10 hours on the project named *Order Entry*.

  - employee,
  - assignment,
  - project

emp_ssn	**last_name**	**first_name**

emp_ssn	**pro_no**	**work_hours**

pro_no	**pro_name**	

Example

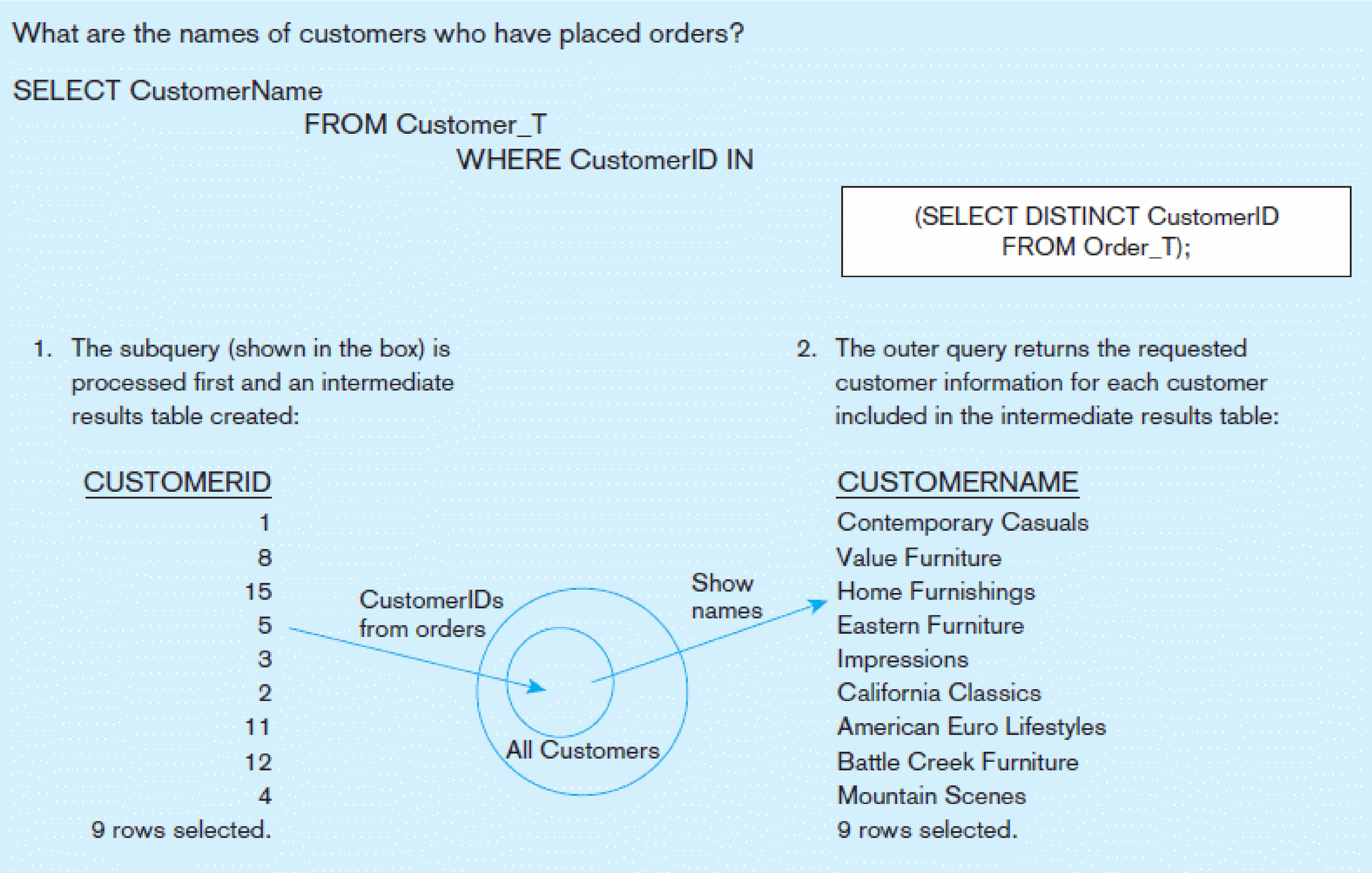
```
SELECT emp_last_name "Last Name",  emp_first_name "First Name"
   FROM employee  WHERE emp_ssn IN
            (SELECT work_emp_ssn
                  FROM assignment
                        WHERE work_hours > 10 AND work_pro_number IN
                                 (SELECT pro_number
                                     FROM project
                                           WHERE pro_name = 'Order Entry') );
```

```
Last Name    First Name
--------------- ---------------
Bock          Douglas
Prescott       Sherri
```

Correlated vs. Non-correlated Subqueries

- Subqueries can be:
  - Noncorrelated–executed once for the entire outer query
  - Correlated–executed once for each row returned by the outer query
- **Non-correlated** subqueries:
  - Do not depend on data from the outer query
  - Execute once for the entire outer query
- **Correlated** subqueries:
  - Make use of data from the outer query
  - Execute once for each row of the outer query
  - Usually use the EXISTS operator

Processing a noncorrelated subquery



What are the names of customers who have placed orders?

```
SELECT CustomerName
        FROM Customer_T
            WHERE CustomerID IN
                (SELECT DISTINCT CustomerID
                 FROM Order_T);
```

1. The subquery (shown in the box) is processed first and an intermediate results table created:

```
CUSTOMERID
         1
         8
        15
         5
         3
         2
        11
        12
         4
9 rows selected.
```

2. The outer query returns the requested customer information for each customer included in the intermediate results table:

```
CUSTOMERNAME
Contemporary Casuals
Value Furniture
Home Furnishings
Eastern Furniture
Impressions
California Classics
American Euro Lifestyles
Battle Creek Furniture
Mountain Scenes
9 rows selected.
```

A noncorrelated subquery processes completely before the outer query begins

Correlated Subquery Example

- Show all orders that include furniture finished in natural ash

The EXISTS operator will return a TRUE value if the subquery resulted in a non-empty set, otherwise it returns a FALSE

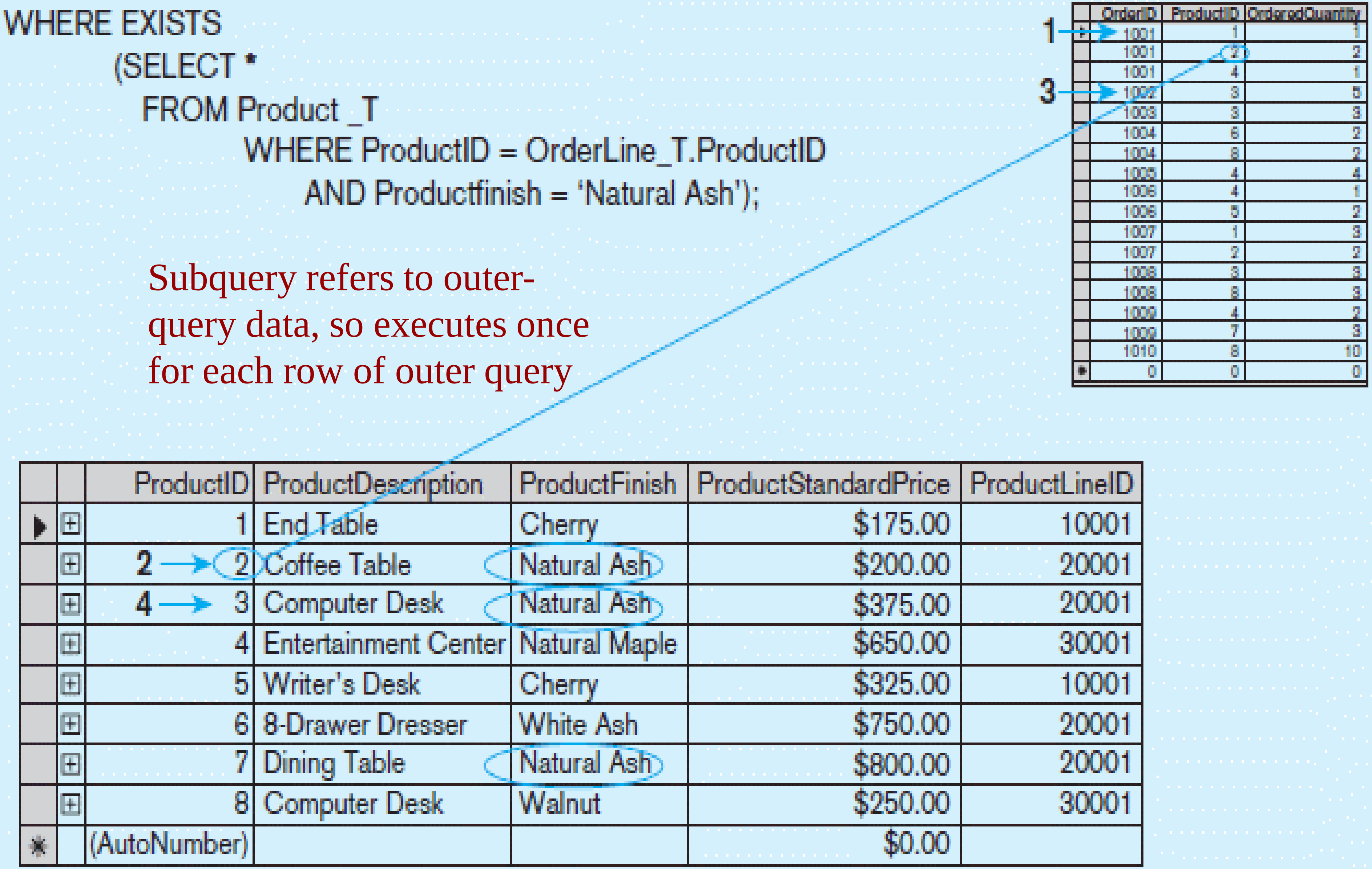
```
SELECT DISTINCT ORDER_ID FROM ORDER_LINE_T
WHERE  EXISTS

   (SELECT * FROM PRODUCT_T
      WHERE PRODUCT_ID = ORDER_LINE_T.PRODUCT_ID

      AND PRODUCT_FINISH = ‘Natural ash’);
```

The subquery is testing for a value that comes from the outer query

What are the order IDs for all orders that have included furniture finished in natural ash?

```
SELECT DISTINCT OrderID FROM OrderLine_T
WHERE EXISTS
        (SELECT *
          FROM Product _T
                WHERE ProductID = OrderLine_T.ProductID
                    AND Productfinish = 'Natural Ash');
```

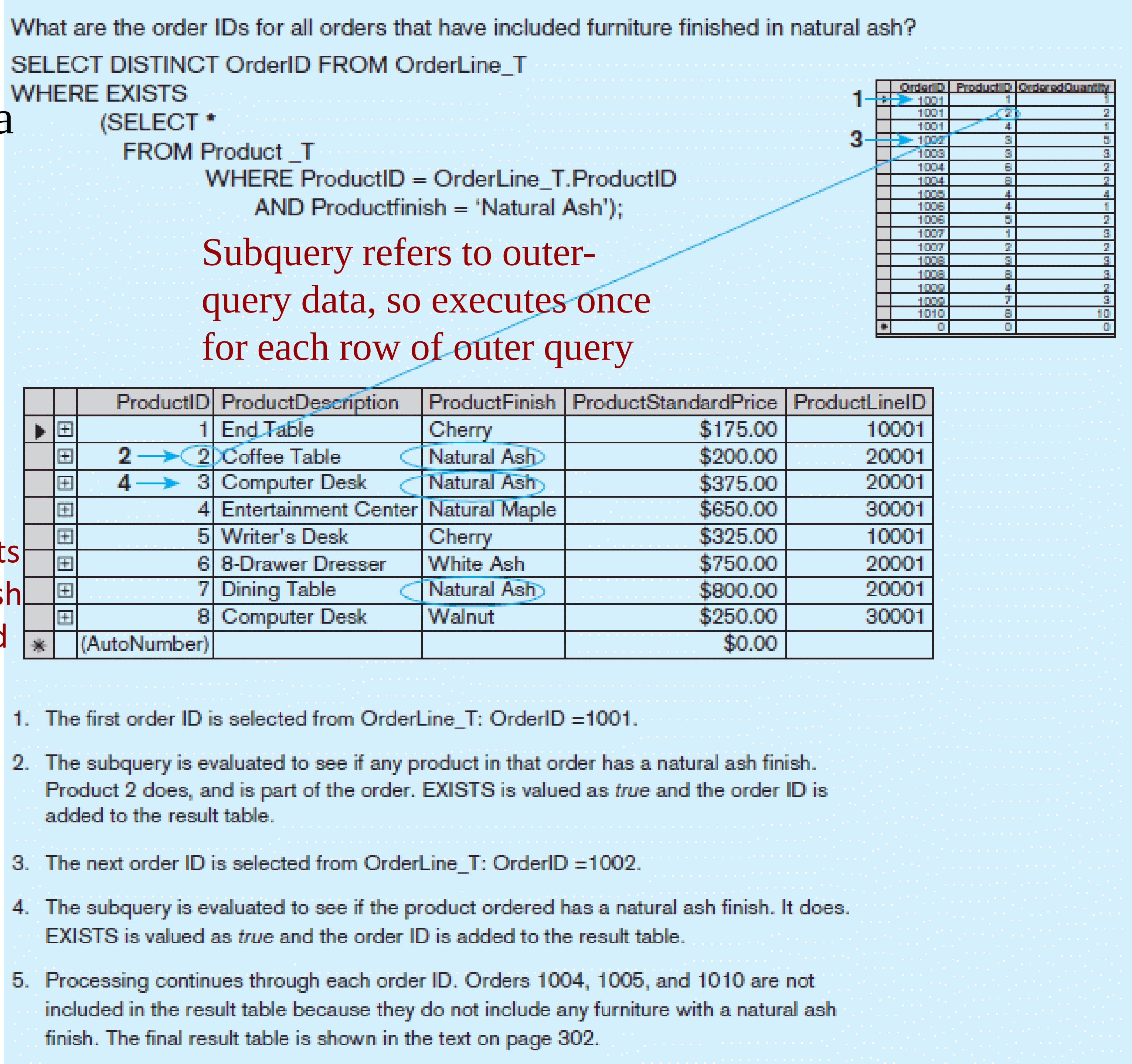
Subquery refers to outer-query data, so executes once for each row of outer query



	OrderID	ProductID	OrderedQuantity
▸	1001	1	1
	1001	2	2
	1001	4	1
	1002	3	5
	1003	3	3
	1004	6	2
	1004	8	2
	1005	4	4
	1006	4	1
	1006	5	2
	1007	1	3
	1007	2	2
	1008	3	3
	1008	8	3
	1009	4	2
	1009	7	3
	1010	8	10
*	0	0	0

		ProductID	ProductDescription	ProductFinish	ProductStandardPrice	ProductLineID
▶	⊞	1	End Table	Cherry	$175.00	10001
	⊞	2	Coffee Table	Natural Ash	$200.00	20001
	⊞	3	Computer Desk	Natural Ash	$375.00	20001
	⊞	4	Entertainment Center	Natural Maple	$650.00	30001
	⊞	5	Writer's Desk	Cherry	$325.00	10001
	⊞	6	8-Drawer Dresser	White Ash	$750.00	20001
	⊞	7	Dining Table	Natural Ash	$800.00	20001
	⊞	8	Computer Desk	Walnut	$250.00	30001
*		(AutoNumber)			$0.00	

Processing a correlated subquery



What are the order IDs for all orders that have included furniture finished in natural ash?

```
SELECT DISTINCT OrderID FROM OrderLine_T
WHERE EXISTS
      (SELECT *
        FROM Product _T
              WHERE ProductID = OrderLine_T.ProductID
                  AND Productfinish = 'Natural Ash');
```

Subquery refers to outer-query data, so executes once for each row of outer query

OrderID	ProductID	OrderedQuantity
1001	1	1
1001	2	2
1001	4	1
1002	3	5
1003	3	3
1004	6	2
1004	8	2
1005	4	4
1006	4	1
1006	5	2
1007	1	3
1007	2	2
1008	3	3
1008	8	3
1009	4	2
1009	7	3
1010	8	10
0	0	0

ProductID	ProductDescription	ProductFinish	ProductStandardPrice	ProductLineID
1	End Table	Cherry	$175.00	10001
2	Coffee Table	Natural Ash	$200.00	20001
3	Computer Desk	Natural Ash	$375.00	20001
4	Entertainment Center	Natural Maple	$650.00	30001
5	Writer's Desk	Cherry	$325.00	10001
6	8-Drawer Dresser	White Ash	$750.00	20001
7	Dining Table	Natural Ash	$800.00	20001
8	Computer Desk	Walnut	$250.00	30001
(AutoNumber)			$0.00	

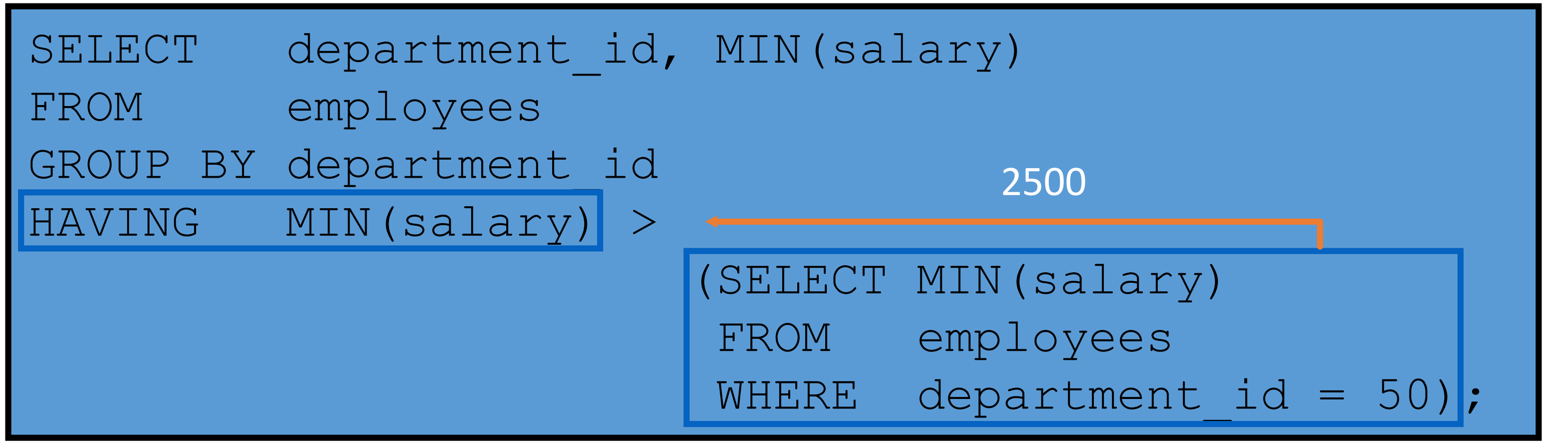
Note: only the orders that involve products with Natural Ash will be included in the final results

1. The first order ID is selected from OrderLine_T: OrderID =1001.
2. The subquery is evaluated to see if any product in that order has a natural ash finish. Product 2 does, and is part of the order. EXISTS is valued as *true* and the order ID is added to the result table.
3. The next order ID is selected from OrderLine_T: OrderID =1002.
4. The subquery is evaluated to see if the product ordered has a natural ash finish. It does. EXISTS is valued as *true* and the order ID is added to the result table.
5. Processing continues through each order ID. Orders 1004, 1005, and 1010 are not included in the result table because they do not include any furniture with a natural ash finish. The final result table is shown in the text on page 302.

The **HAVING** Clause with Subqueries

- Display all the departments that have a minimum salary greater than that of department 50

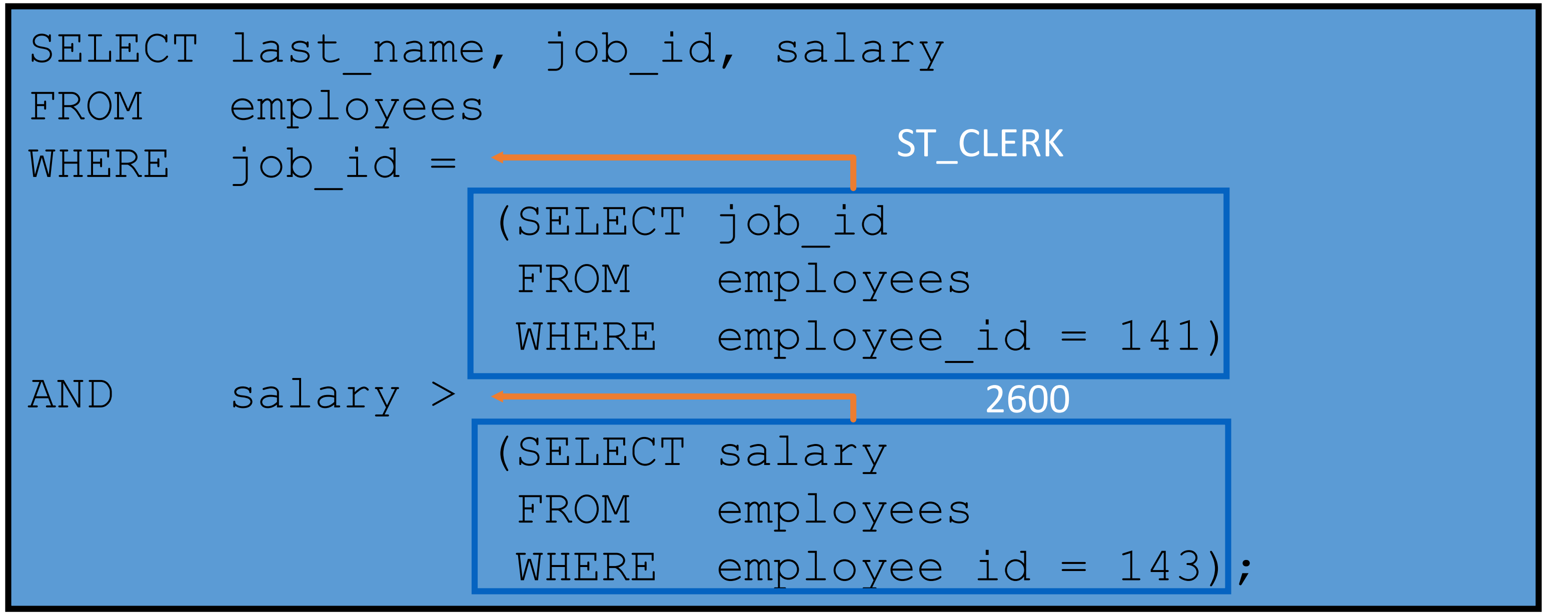
emp_id	dept_id	salary
1001	40	5000
1002	30	4500
1003	50	2500
1004	50	4000
1005	30	3700
1006	40	3500



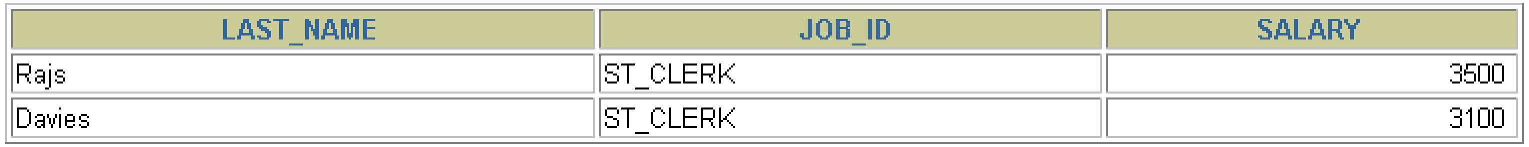
```
SELECT    department_id, MIN(salary)
FROM      employees
GROUP BY  department_id
HAVING    MIN(salary)  >
                        (SELECT MIN(salary)
                         FROM   employees
                         WHERE  department_id = 50);
```

Exercise: Executing Single-Row Subqueries

display employees whose job ID is the same as that of employee 141 and whose salary is greater than that of employee 143.



```
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);
```



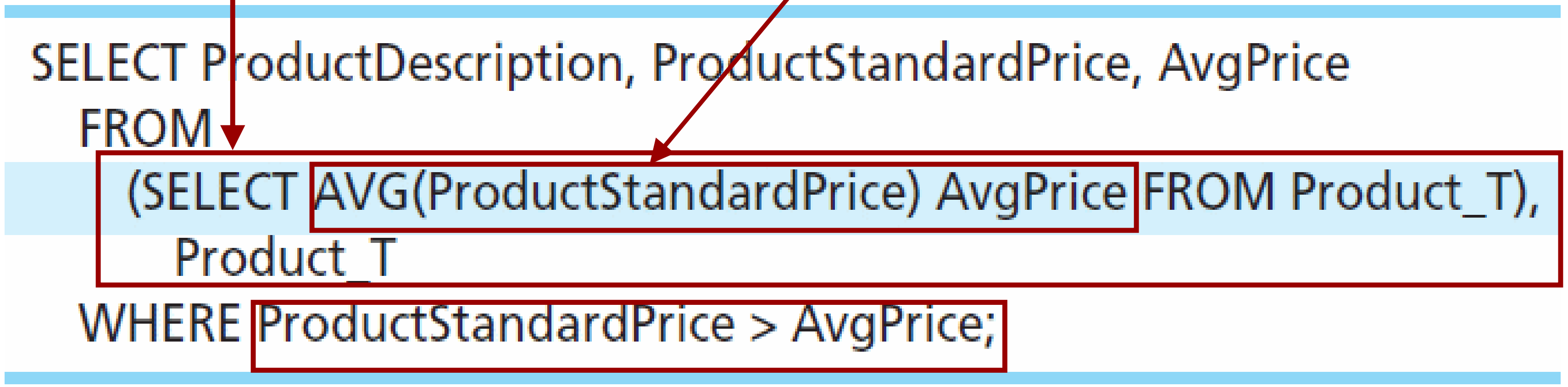
LAST_NAME	JOB_ID	SALARY
Rajs	ST_CLERK	3500
Davies	ST_CLERK	3100

Subquery – Derived Table Example

- Show all products whose standard price is higher than the average price

Subquery forms the derived table used in the FROM clause of the outer query

One column of the subquery is an aggregate function that has an alias name. That alias can then be referred to in the outer query



```
SELECT ProductDescription, ProductStandardPrice, AvgPrice
  FROM
     (SELECT AVG(ProductStandardPrice) AvgPrice FROM Product_T),
       Product_T
  WHERE ProductStandardPrice > AvgPrice;
```

The WHERE clause normally cannot include aggregate functions, but because the aggregate is performed in the subquery its result can be used in the outer query's WHERE clause.
Derived table is required when we want to display information from subquery e.g here we want to show both the standard price and the average standard price

SELECT Sub-query Examples

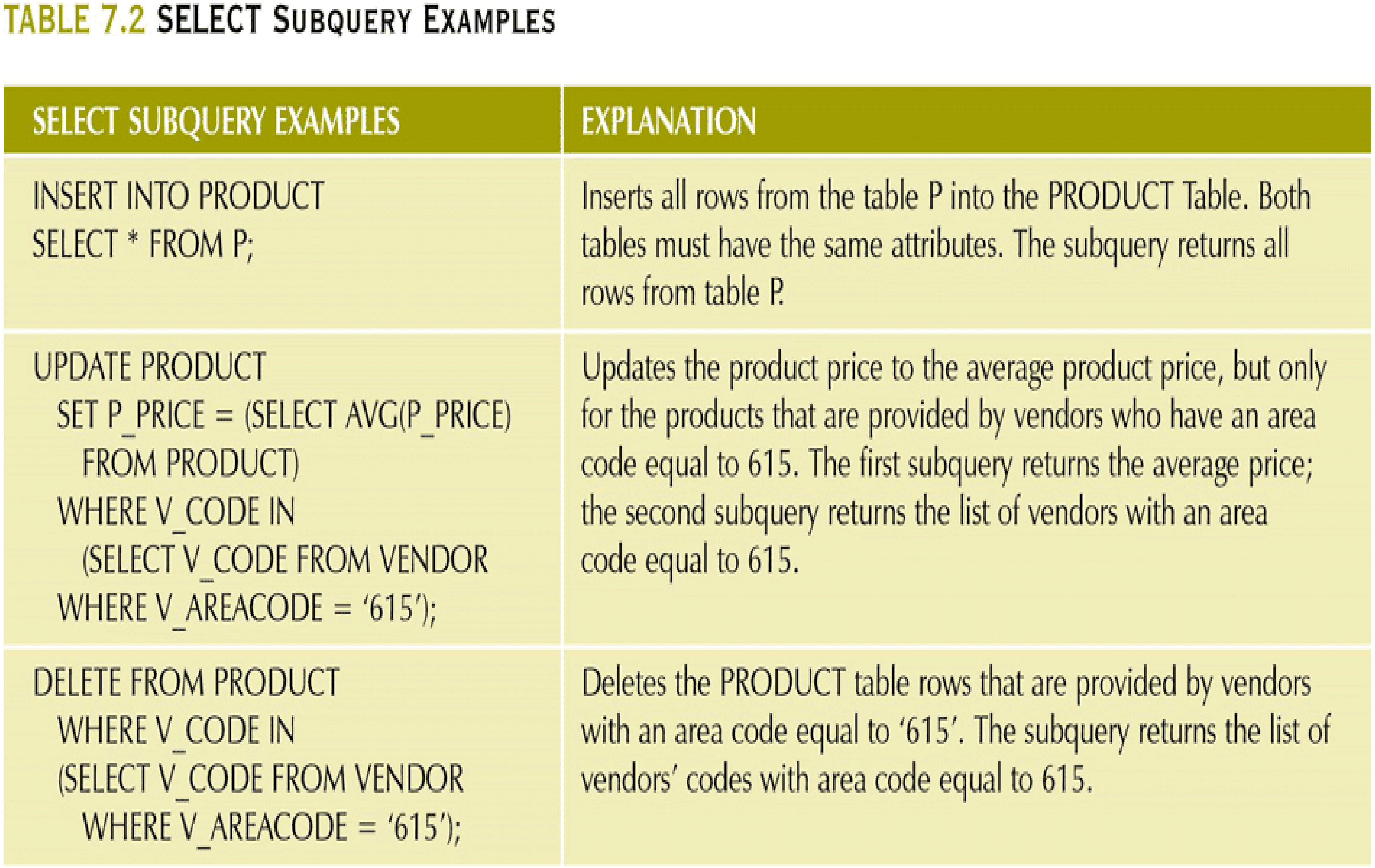


TABLE 7.2 SELECT SUBQUERY EXAMPLES

SELECT SUBQUERY EXAMPLES	EXPLANATION

INSERT INTO PRODUCT
SELECT * FROM P;
	Inserts all rows from the table P into the PRODUCT Table. Both tables must have the same attributes. The subquery returns all rows from table P.

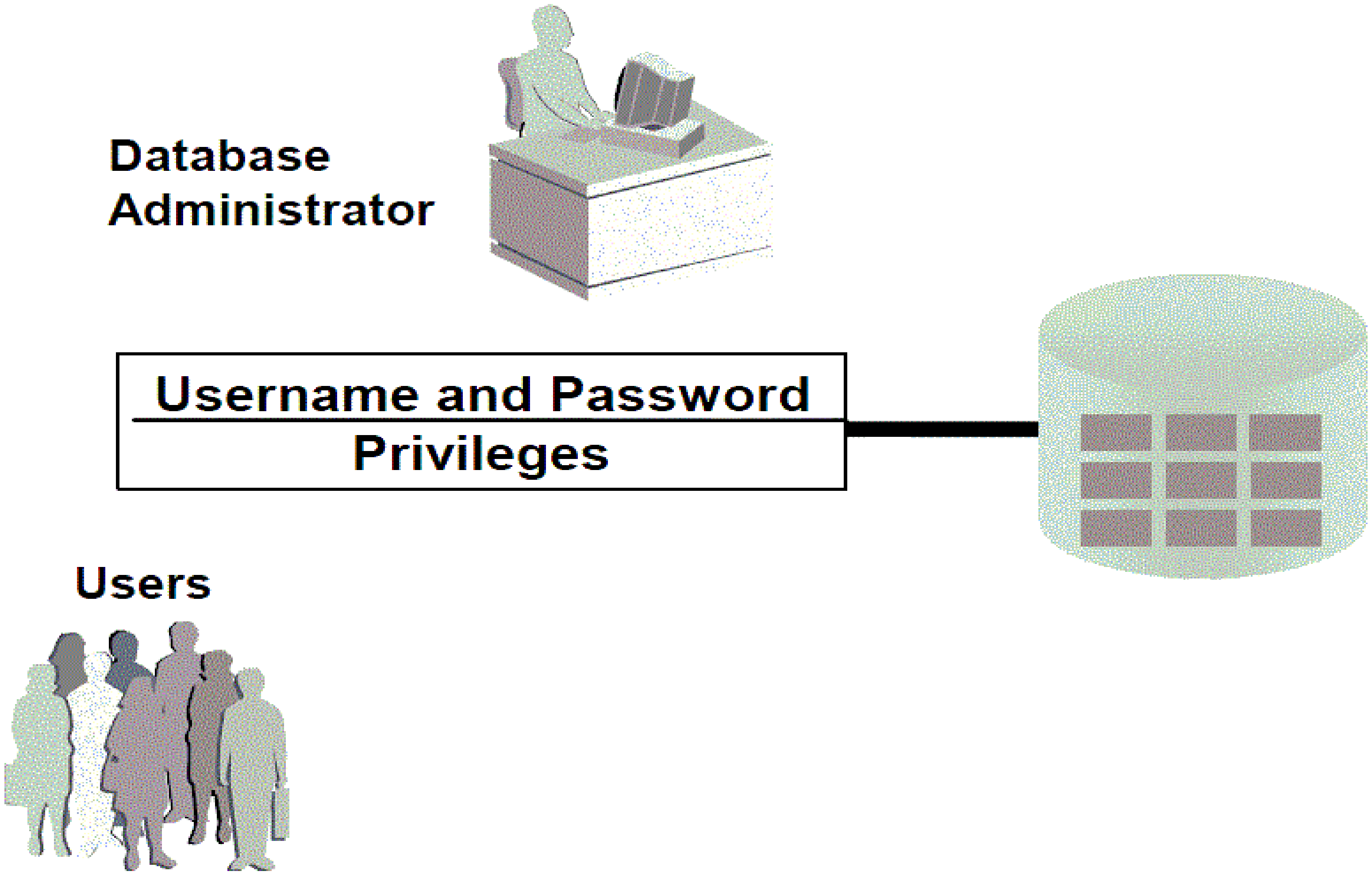
UPDATE PRODUCT
SET P_PRICE = (SELECT AVG(P_PRICE)
FROM PRODUCT)
WHERE V_CODE IN
(SELECT V_CODE FROM VENDOR
WHERE V_AREACODE = '615');
	Updates the product price to the average product price, but only for the products that are provided by vendors who have an area code equal to 615. The first subquery returns the average price; the second subquery returns the list of vendors with an area code equal to 615.

DELETE FROM PRODUCT
WHERE V_CODE IN
(SELECT V_CODE FROM VENDOR
WHERE V_AREACODE = '615');
	Deletes the PRODUCT table rows that are provided by vendors with an area code equal to '615'. The subquery returns the list of vendors' codes with area code equal to 615.

Data Control Language (DCL)

- Create users
- Create roles
- Use GRANT and REVOKE statements
- Create and use database link

User Access



Database Administrator

Username and Password

Privileges

Users

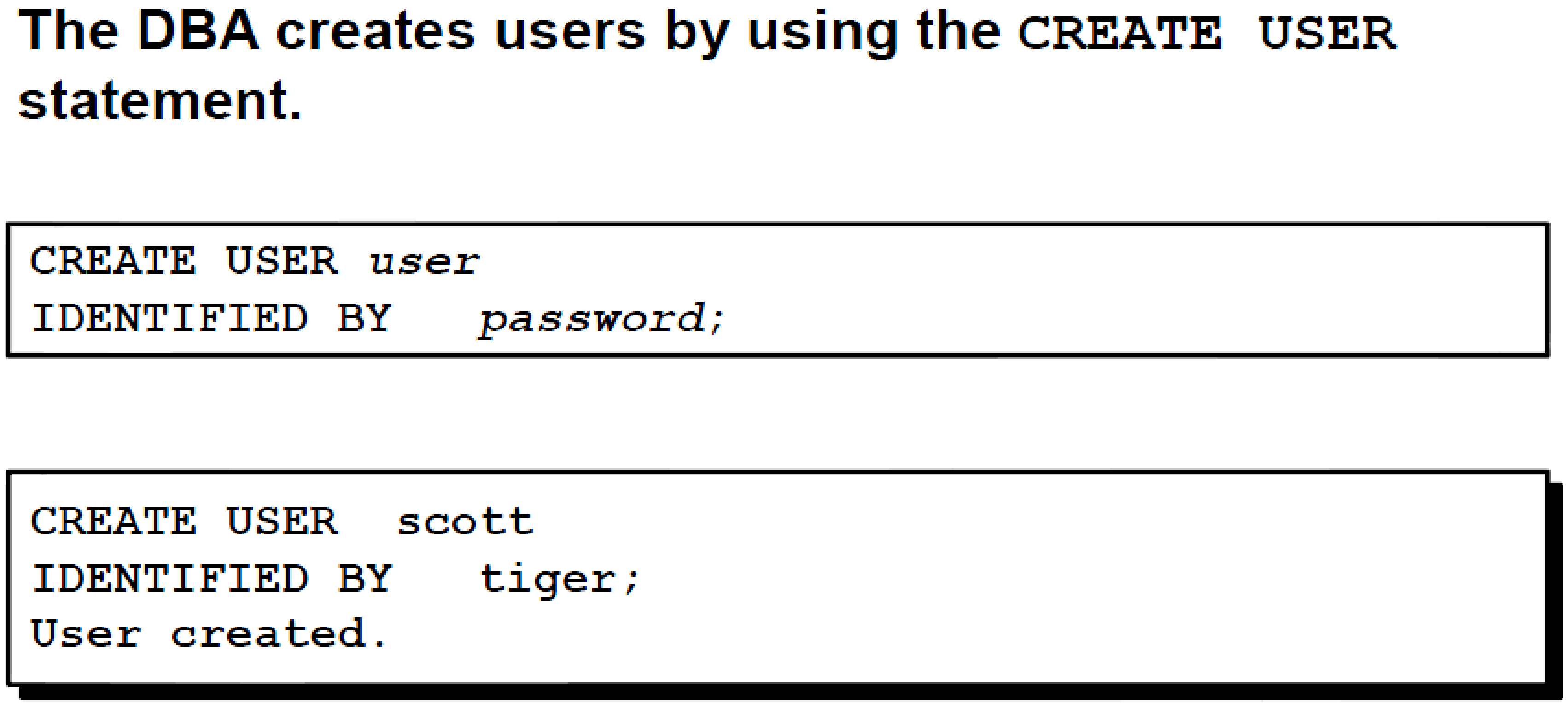
Privileges

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

System Privileges

- **More than 100 privileges are available.**
- **The database administrator has high-level system privileges for tasks such as:**
  - **Creating new users**
  - **Removing users**
  - **Removing tables**
  - **Backing up tables**

Creating users

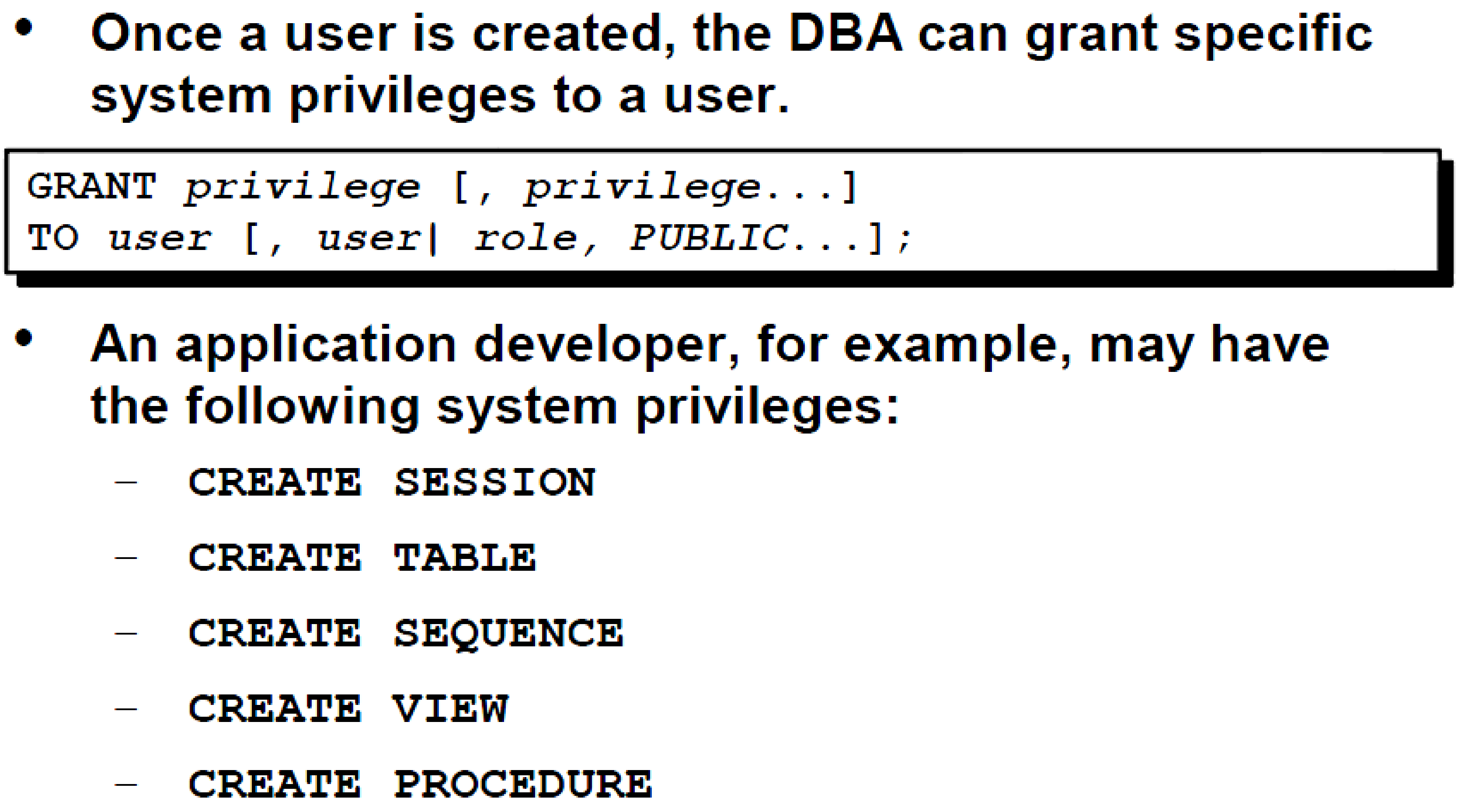


The DBA creates users by using the `CREATE USER` statement.

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

```
CREATE USER  scott
IDENTIFIED BY   tiger;
User created.
```

User System Privileges



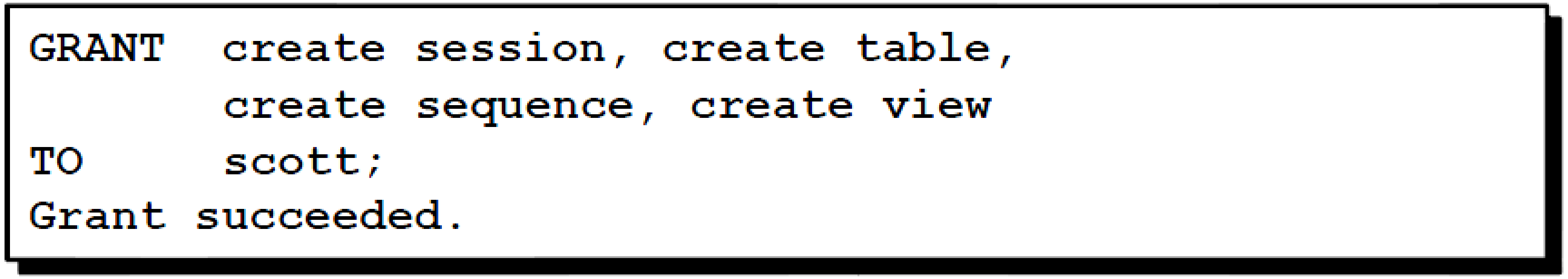
- **Once a user is created, the DBA can grant specific system privileges to a user.**

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

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

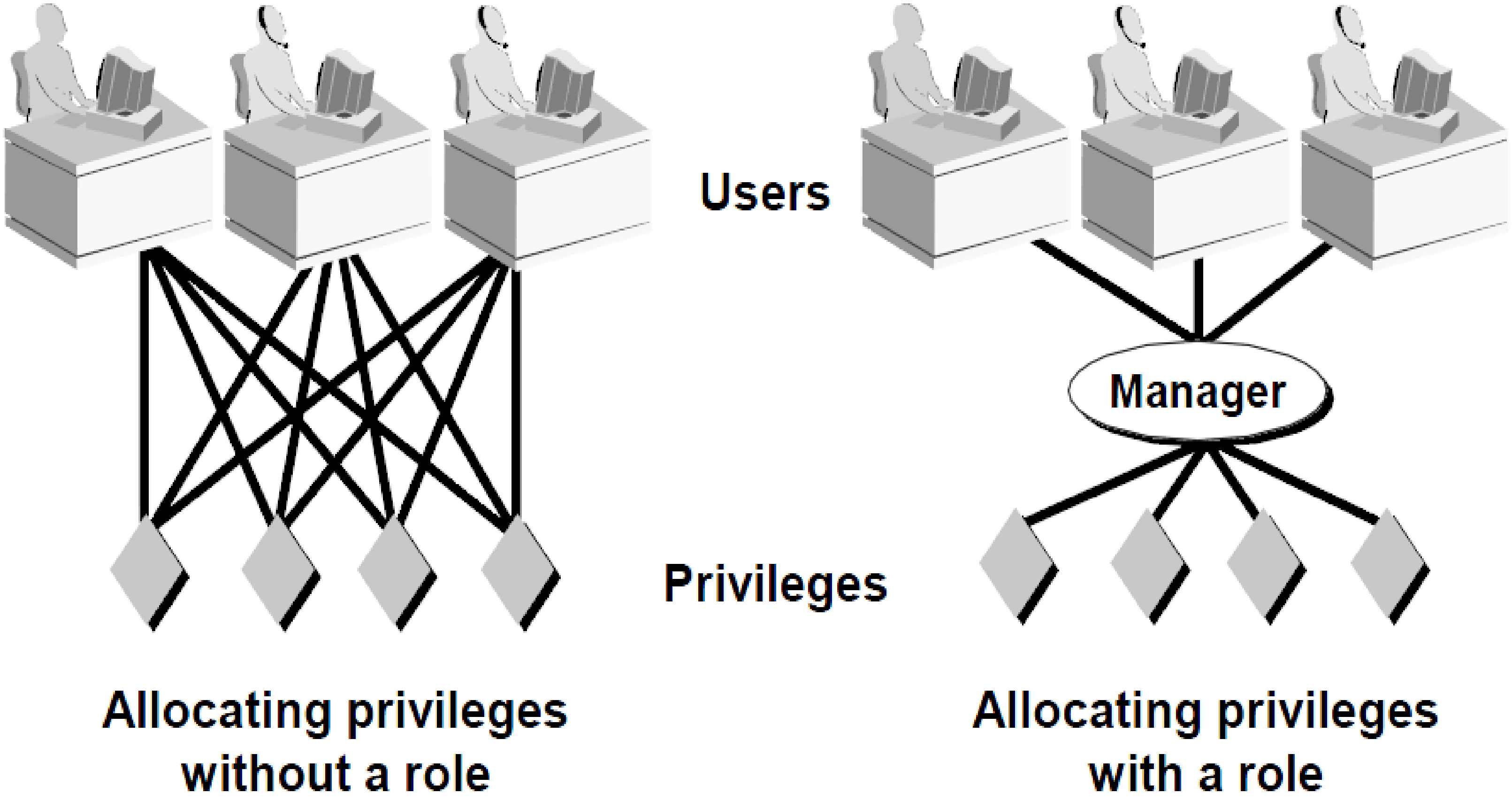
Granting System Privileges

The DBA can grant a user specific system privileges.

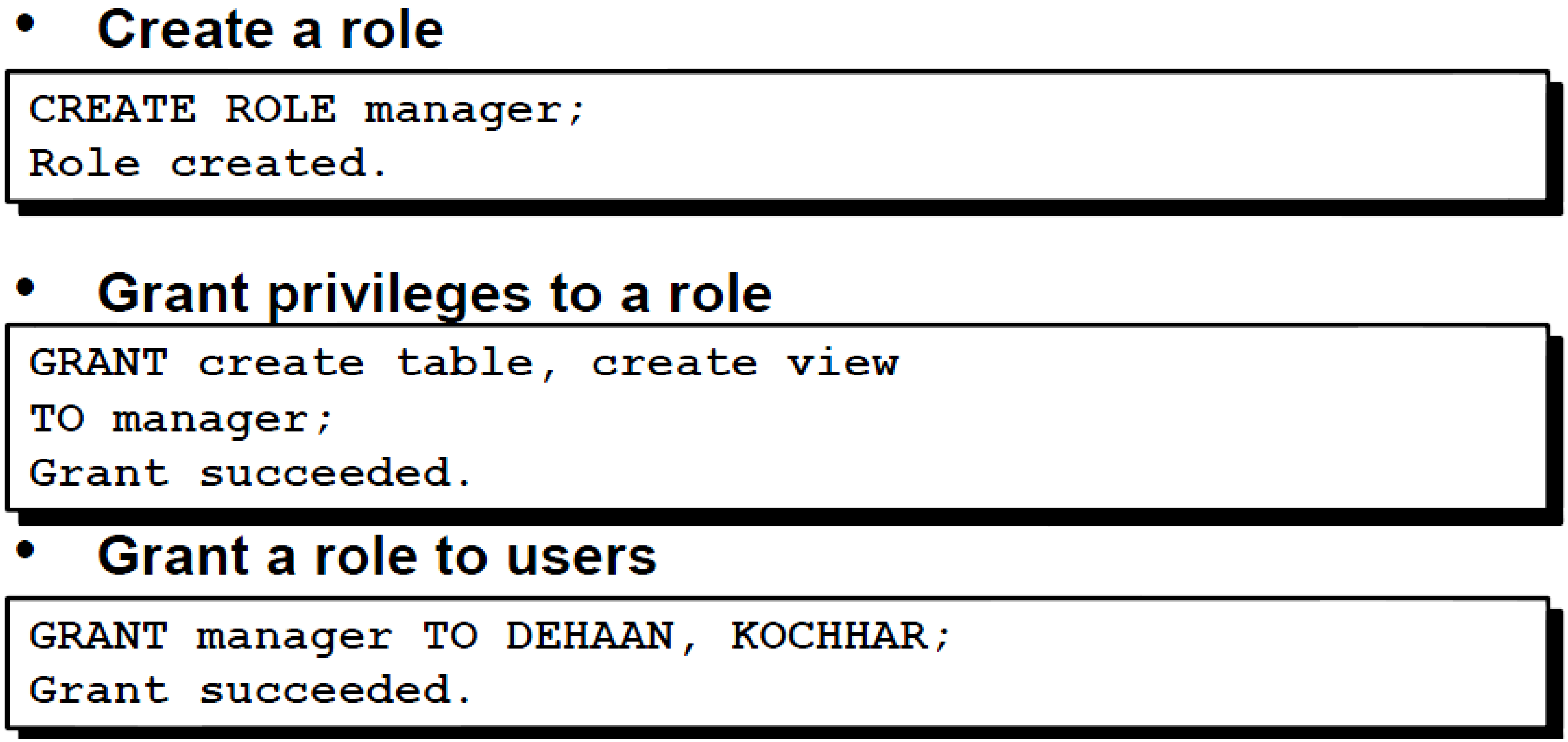


```
GRANT  create session, create table,
       create sequence, create view
TO     scott;
Grant succeeded.
```

What is a Role?


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

Creating and Granting Privileges to a Role



- **Create a role**

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

- **Grant privileges to a role**

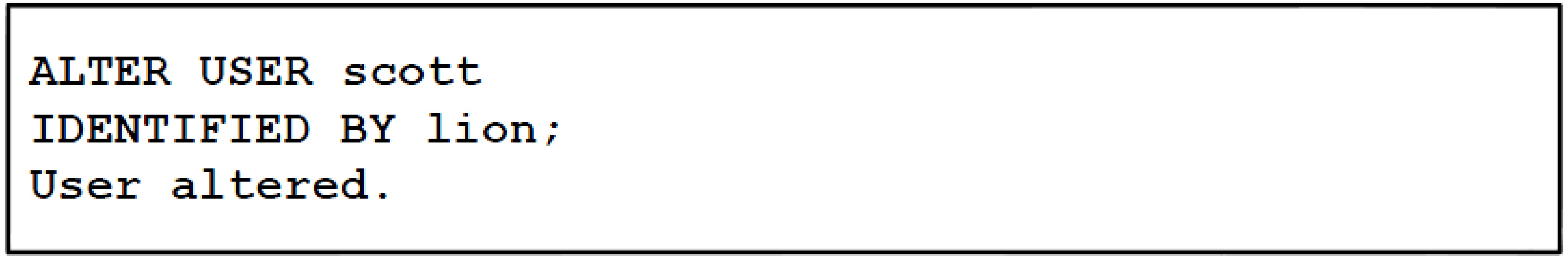
```
GRANT create table, create view
TO manager;
Grant succeeded.
```

- **Grant a role to users**

```
GRANT manager TO DEHAAN, KOCHHAR;
Grant succeeded.
```

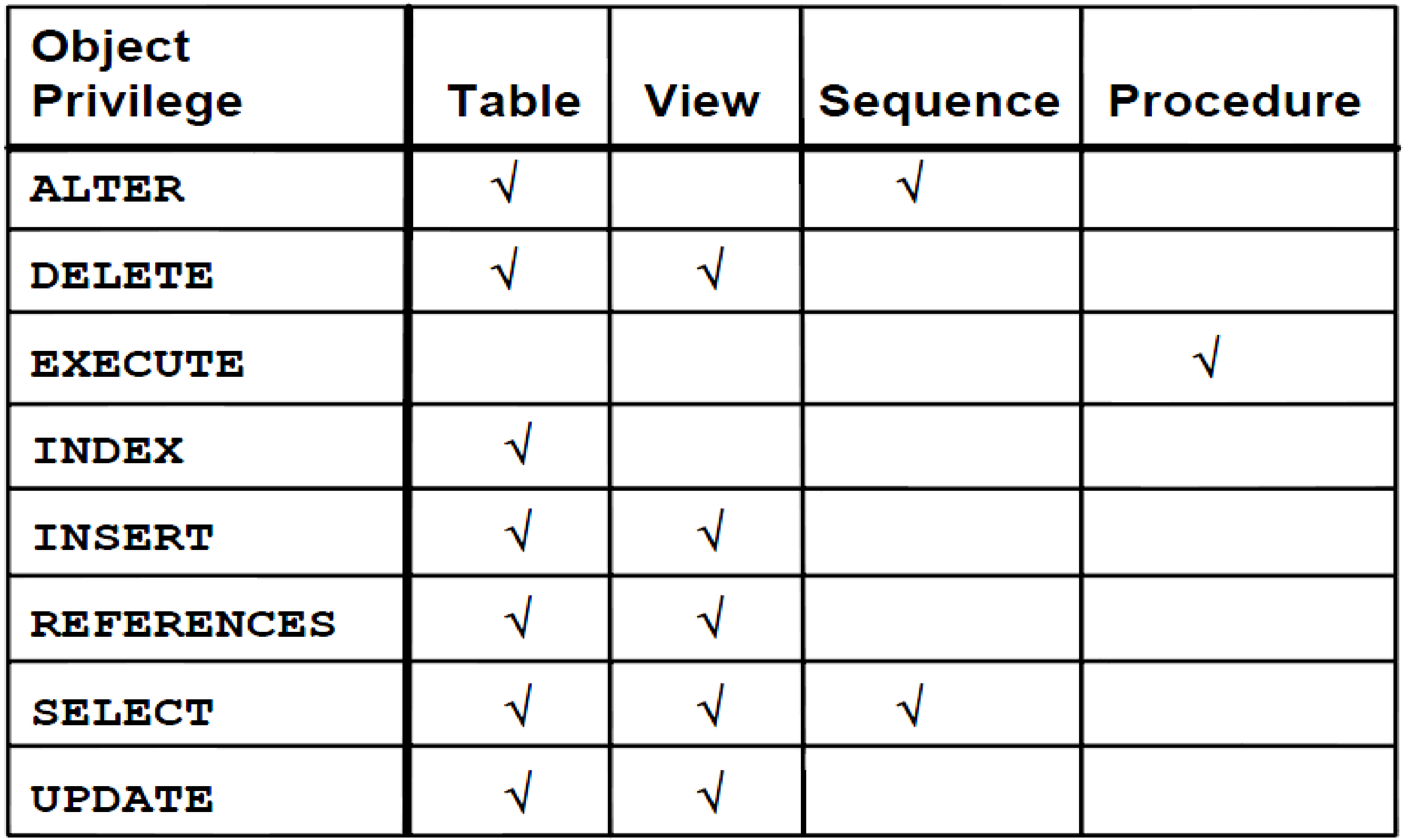
Changing Password

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



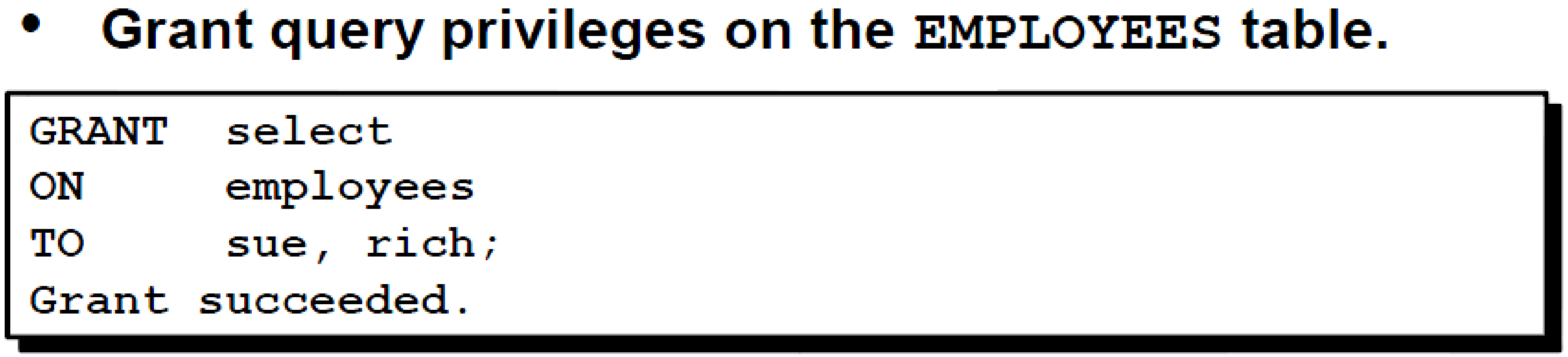
```
ALTER USER scott
IDENTIFIED BY lion;
User altered.
```

Object Privileges



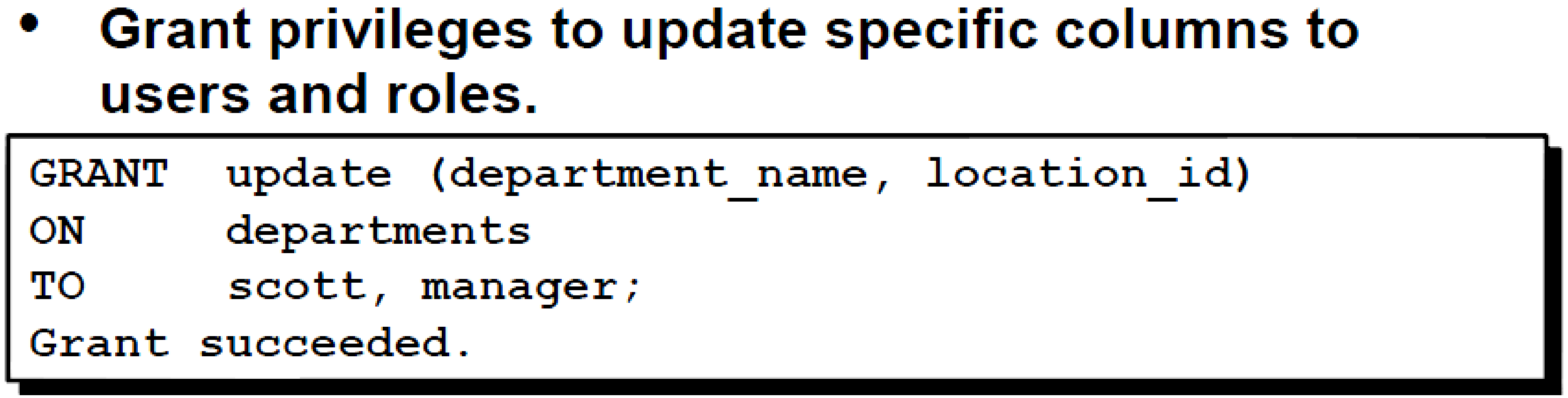
Object Privilege	Table	View	Sequence	Procedure
ALTER	√		√	
DELETE	√	√		
EXECUTE				√
INDEX	√			
INSERT	√	√		
REFERENCES	√	√		
SELECT	√	√	√	
UPDATE	√	√		

Granting Object Privileges



- **Grant query privileges on the `EMPLOYEES` table.**

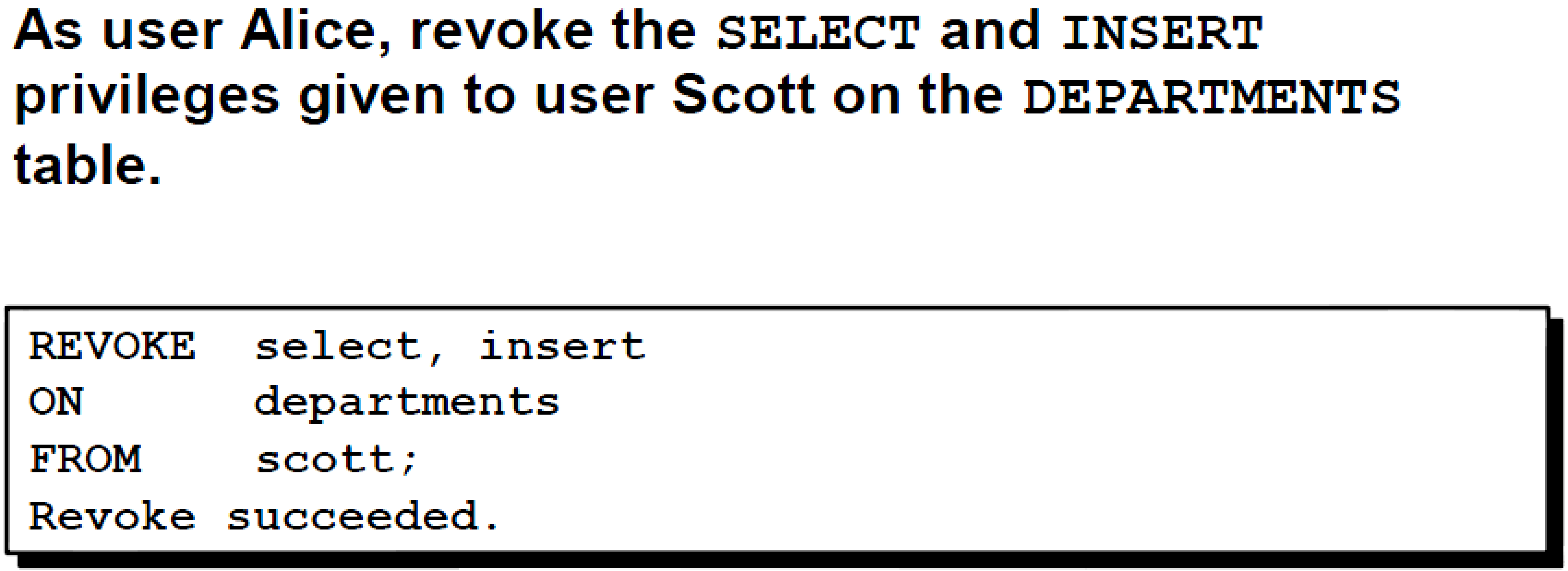
```
GRANT  select
ON     employees
TO     sue, rich;
Grant succeeded.
```



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

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

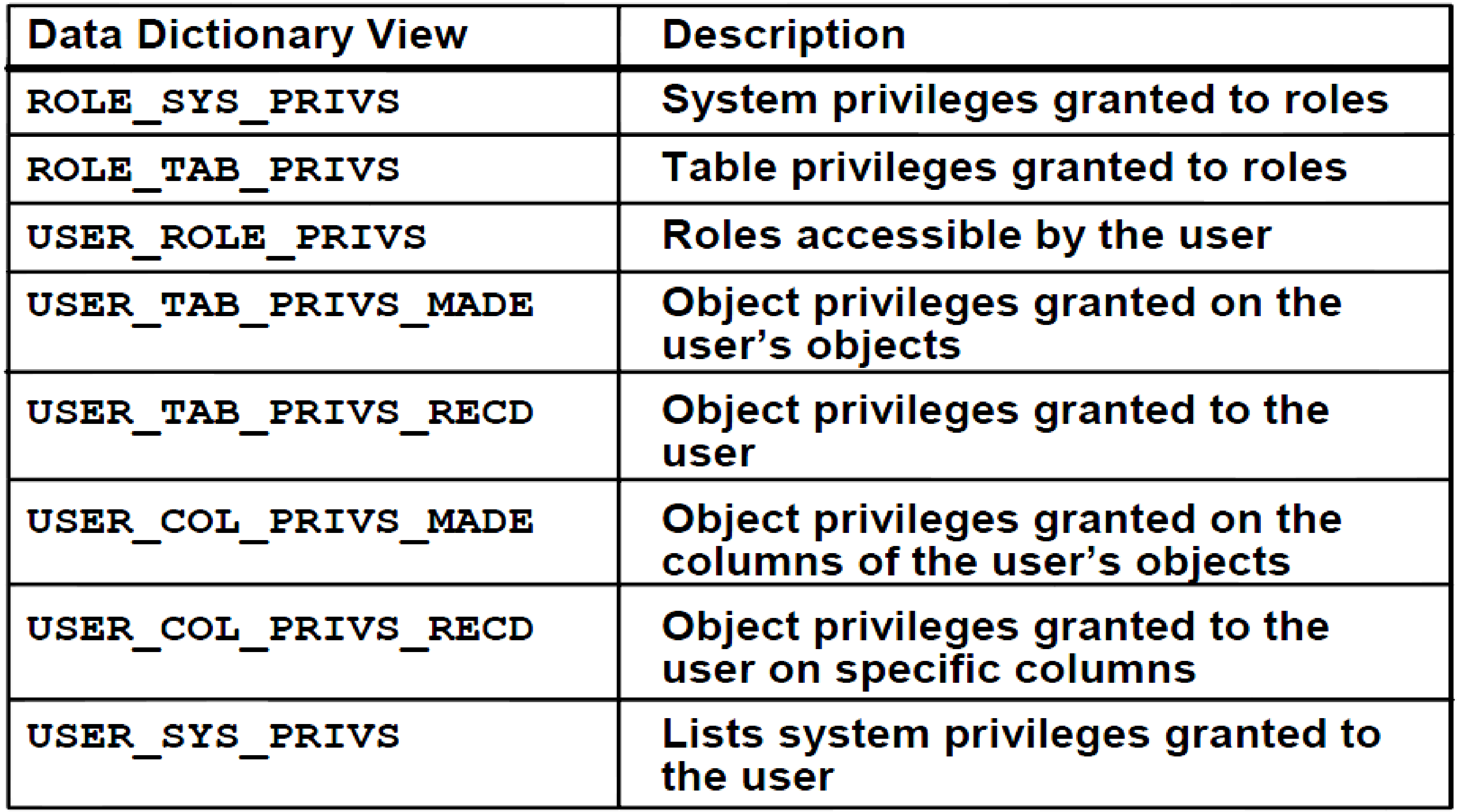
Revoking Privileges



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

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

Data Dictionary for Privileges



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

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

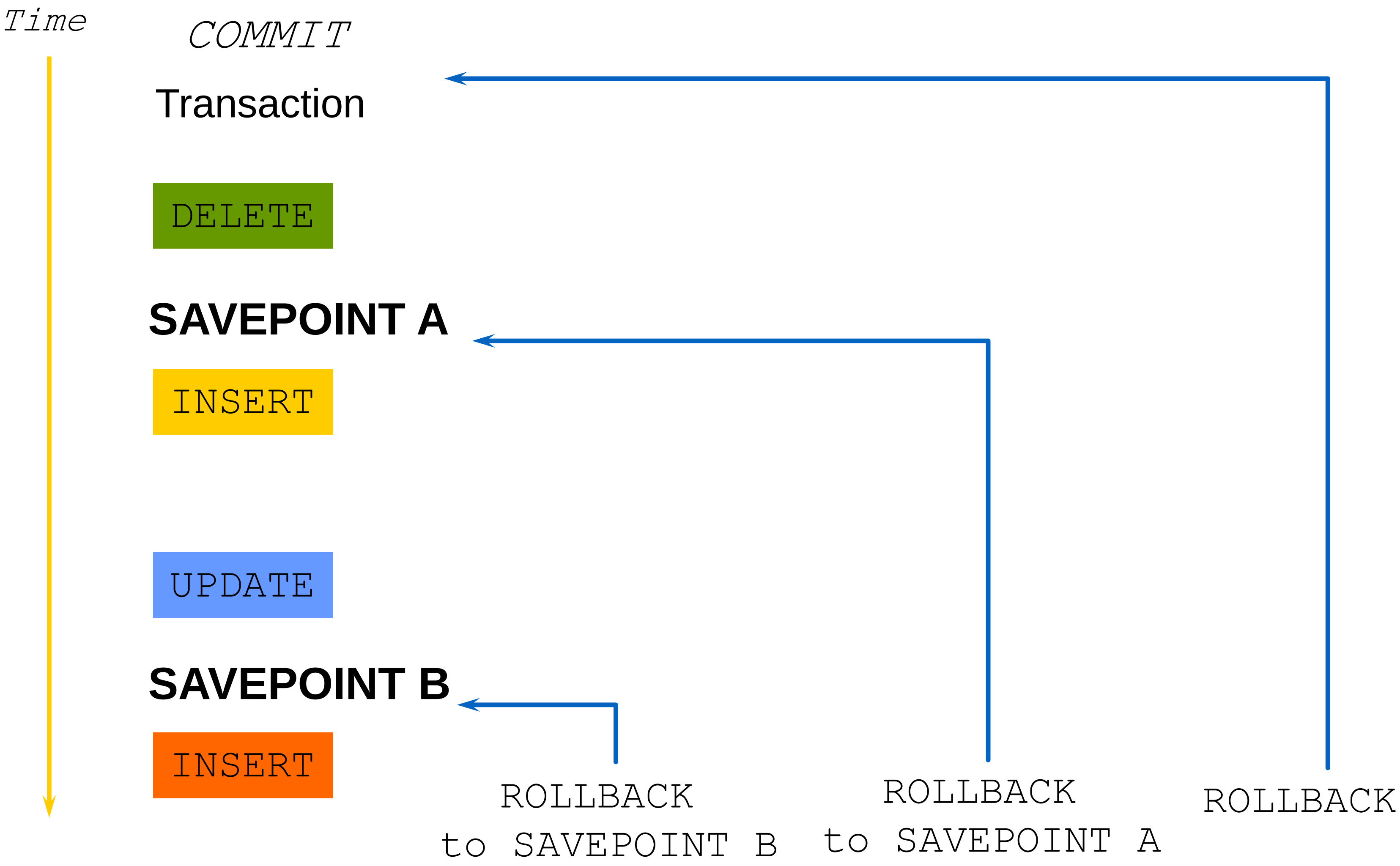
    (automatic commit)
  - The system crashes

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*Plus, without explicitly issuing `COMMIT` or `ROLLBACK` statements
- An automatic rollback occurs under an abnormal termination of SQL*Plus 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 save points 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.
```

State of the Data After ROLLBACK: Example

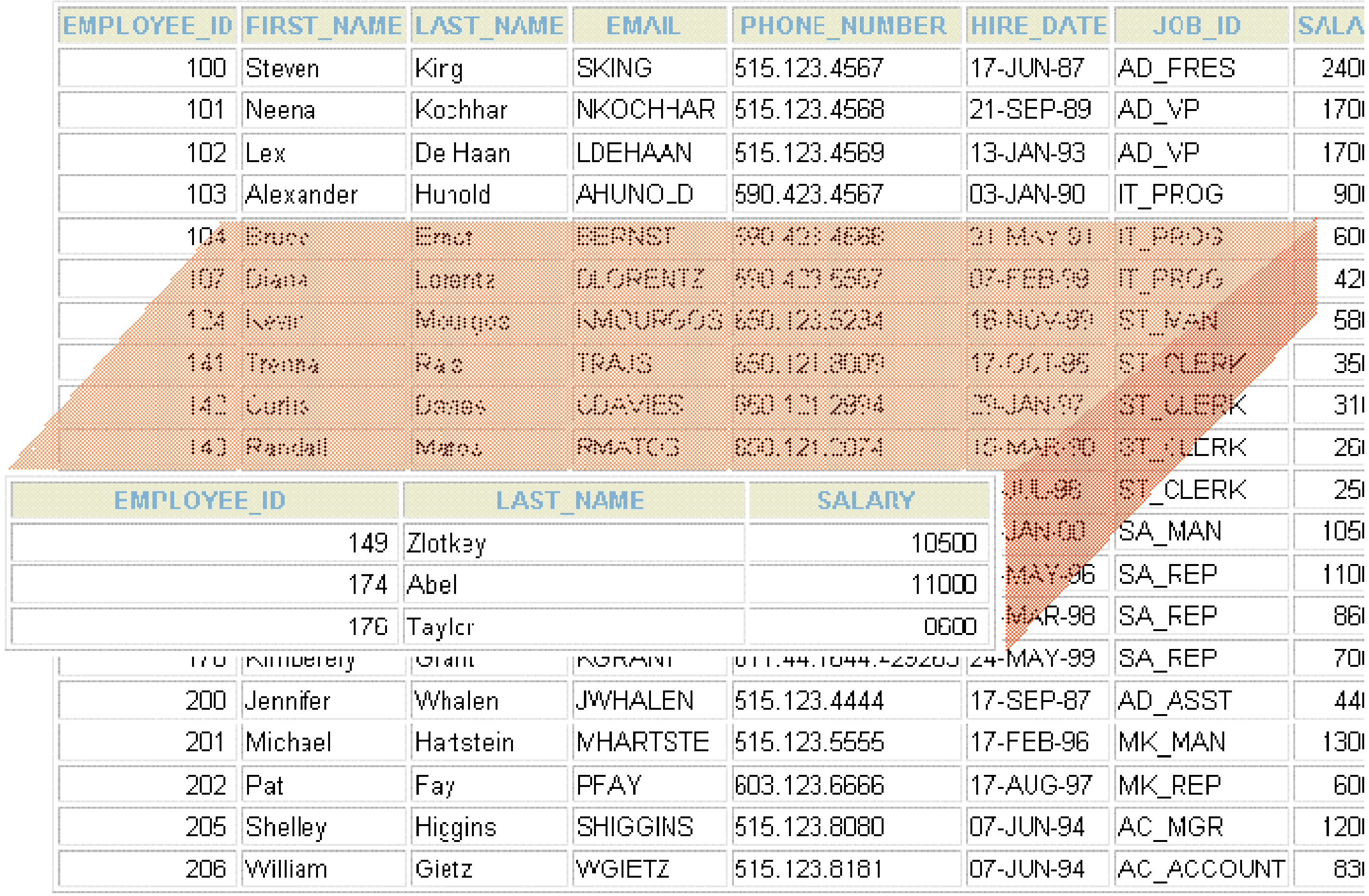
```
DELETE FROM test;
25,000 rows deleted.

ROLLBACK;
Rollback complete.

DELETE FROM test WHERE  id = 100;
1 row deleted.

SELECT * FROM   test WHERE  id = 100;
No rows selected.

COMMIT;
Commit complete.
```

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	Kirg	SKING	515.123.4567	17-JUN-87	AD_FRES	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?

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

Simple and Complex Views

Feature	Simple Views	Complex Views
Number of tables	One	One or more
Contain functions	No	Yes
Contain groups of data	No	Yes
DML operations through a view	Yes	Not always

Creating a View

- You embed a subquery within the `CREATE VIEW` statement.

```
CREATE [OR REPLACE] [FORCE|NOFORCE] VIEW view
  [(alias[, alias]...)]
 AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY [CONSTRAINT constraint]];
```

- The subquery can contain complex `SELECT` syntax.

Creating a View

- 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
```

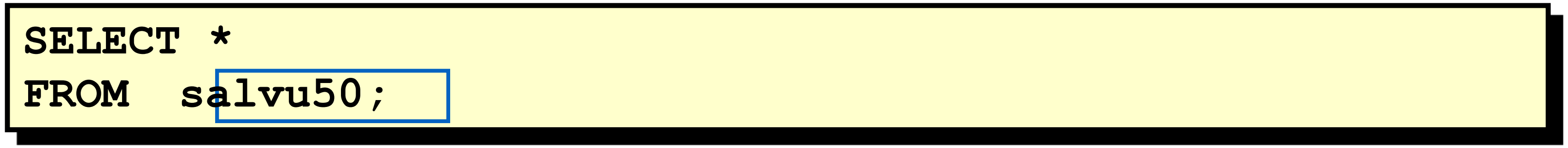
Creating a View

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

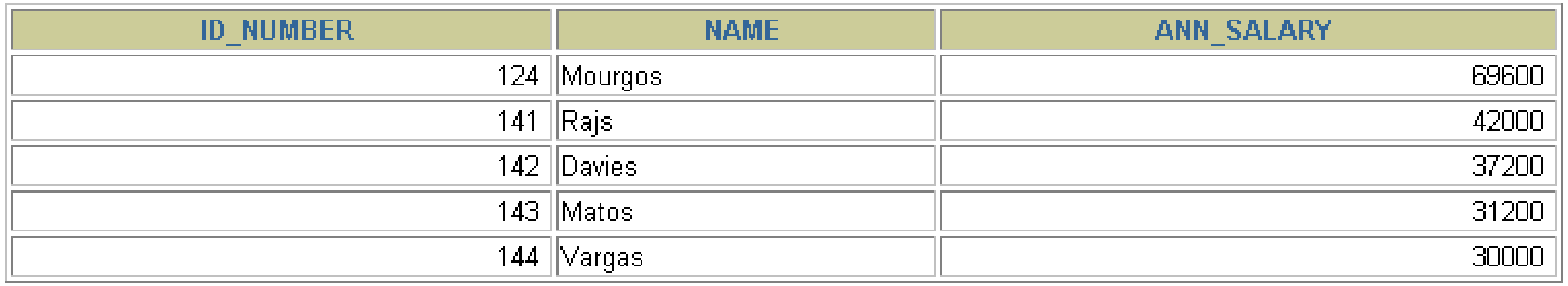
```
CREATE VIEW  salvu50
 AS SELECT   employee_id ID_NUMBER, last_name NAME,
             salary*12 ANN_SALARY
    FROM     employees
    WHERE    department_id = 50;
View created.
```

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

Retrieving data from view

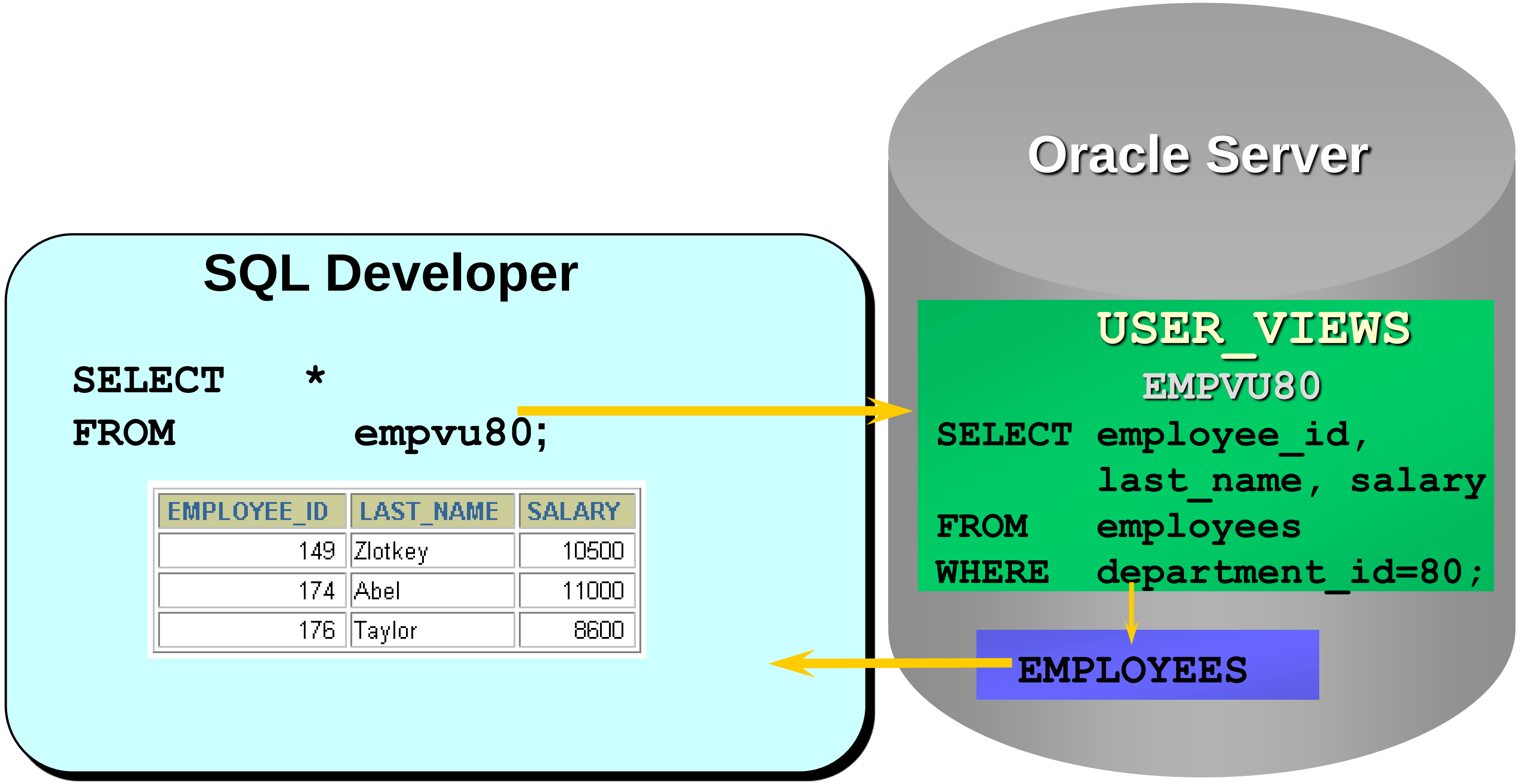


```
SELECT *
FROM  salvu50;
```



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

Querying a View



```
SELECT  *
FROM        empvu80;
```

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

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

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 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.

```
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.

Denying DML Operations

```
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.
```

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 and smallest values sets are considered Top-N queries.

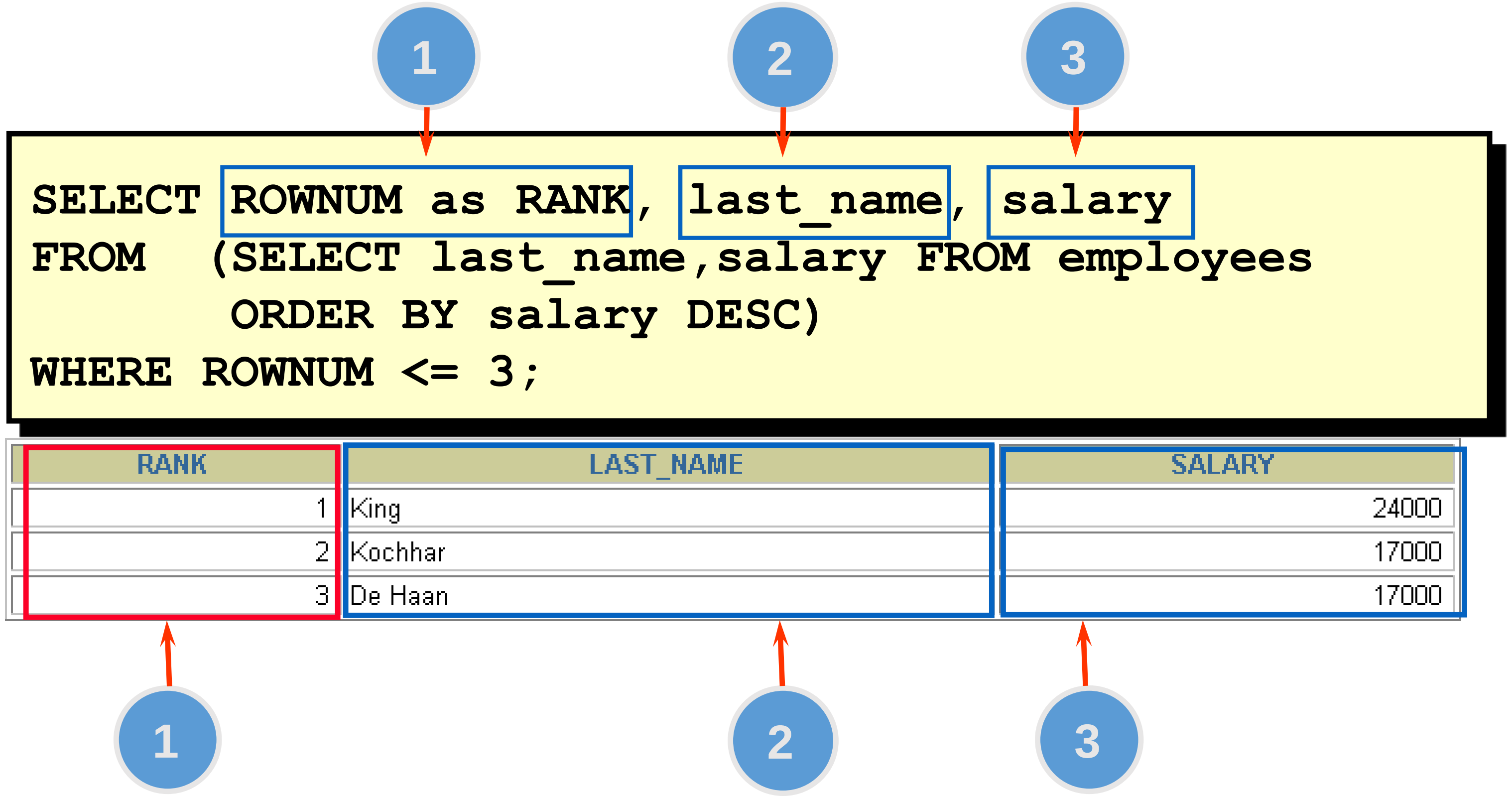
Performing Top-N Analysis

The high-level structure of a Top-N analysis query is:

```
SELECT [column_list], ROWNUM
FROM   (SELECT [column_list]
        FROM table
        ORDER  BY Top-N_column)
WHERE  ROWNUM <=  N;
```

Example of Top-N Analysis

- To display the top three earner names and salaries from the EMPLOYEES table:



```sql
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
```

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

The CREATE SEQUENCE Statement Syntax

Define a sequence to generate sequential numbers automatically:

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

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

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 table is updated frequently
- 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';
```

Function-Based Indexes

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

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

Index created.

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

Removing an Index

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

```
DROP INDEX index;
```

- Remove the UPPER_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.

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.
```