More SQL: Complex Queries, Views, and Schema Modification

Dr. Sumeet Kaur Sehra

More Complex SQL Retrieval Queries

- Additional features allow users to specify more complex retrievals from database:
  - Nested queries, joined tables, and outer joins (in the FROM clause), aggregate functions, and grouping

Schema Change Statements in SQL

- **Schema evolution commands**
  - DBA may want to change the schema while the database is operational
  - Does not require recompilation of the database schema

The DROP Command

- `DROP` command
  - Used to drop named schema elements, such as tables, domains, or constraint
- Drop behavior options:
  - `CASCADE` and `RESTRICT`
- Example:
  - `DROP SCHEMA COMPANY CASCADE;`
  - This removes the schema and all its elements including tables,views, constraints, etc.

The ALTER table command

- **Alter table actions** include:
  - Adding or dropping a column (attribute)
  - Changing a column definition
  - Adding or dropping table constraints
- Example:
  - `ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job VARCHAR(12);`

Alter Statement

Alter Statement:

used to make changes to the schema of the table. Columns can be added and the data type of the columns changed as long as the data in those columns conforms to the data type specified.

Syntax:

```
ALTER TABLE table_name
ADD (column datatype [Default Expression])
[REFERENCES table_name (column_name)’
[CHECK condition]
```

- **Example:**

```
ALTER TABLE studios
ADD (revenue Number int )
```

Alter Statement

Add table level constraints:

Syntax:

```
ALTER TABLE table_name
ADD ([CONSTRAINT constraint_name CHECK comparison]
[columns REFERENCES table_name (columns)]
```

Example:

```
ALTER TABLE studios
ADD (CONSTRAINT check_state CHECK (studio_state in (‘TX’, ‘CA’, ‘WA’))
```

Modify Columns:

Syntax:

```
ALTER TABLE table_name
MODIFY column [data type]
[Default Expression]
[REFERENCES table_name (column_name)’
[CHECK condition]
```

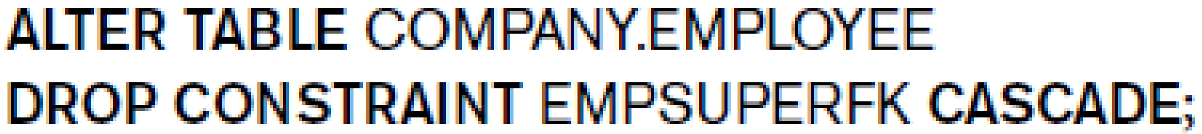
Example:

```
ALTER TABLE People
MODIFY person_union varchar(10)
```

- **Notes1:** Columns can not be removed from the table using alter. If you want to remove columns you have to drop the table and then recreate it without the column that you want to discard

Adding and Dropping Constraints

- Change constraints specified on a table
  - Add or drop a named constraint



```
ALTER TABLE COMPANY.EMPLOYEE
DROP CONSTRAINT EMPSUPERFK CASCADE;
```

Adapted from Navathe 7th Edition

Dropping Columns, Default Values

- To drop a column
  - Choose either `CASCADE` or `RESTRICT`
  - `CASCADE` would drop the column from views etc. `RESTRICT` is possible if no views refer to it.

ALTER TABLE COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE**;

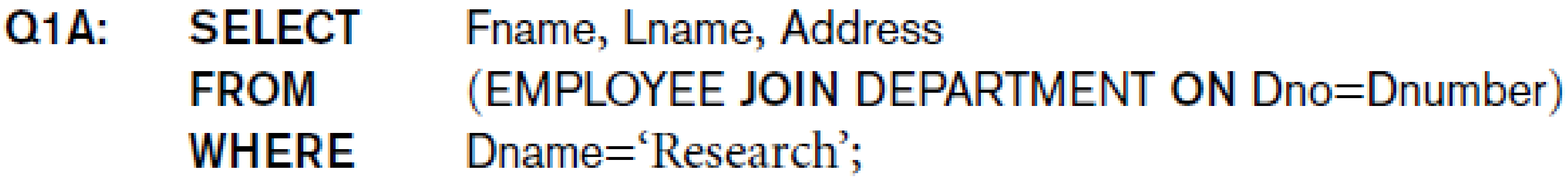
- Default values can be dropped and altered :

  ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn **DROP DEFAULT**;

  ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn **SET DEFAULT** ‘333445555’;

Specifying Joined Tables in the FROM Clause of SQL

- **Joined table**
  - Permits users to specify a table resulting from a join operation in the FROM clause of a query
- The FROM clause in Q1A
  - Contains a single joined table. JOIN may also be called INNER JOIN



```
Q1A:    SELECT    Fname, Lname, Address
        FROM      (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
        WHERE     Dname='Research';
```

Adapted from Navathe 7th Edition

Different Types of JOINed Tables in SQL

- Specify different types of join
  - NATURAL JOIN
  - Various types of OUTER JOIN (LEFT, RIGHT, FULL)
- NATURAL JOIN on two relations R and S
  - No join condition specified
  - Is equivalent to an implicit EQUIJOIN condition for each pair of attributes with same name from R and S

NATURAL JOIN

- Rename attributes of one relation so it can be joined with another using NATURAL JOIN:

```
Q1B:    SELECT    Fname, Lname, Address
        FROM      (EMPLOYEE NATURAL JOIN
                  (DEPARTMENT AS DEPT (Dname, Dno, Mssn,
                     Msdate)))
        WHERE     Dname='Research';
```

The above works with EMPLOYEE.Dno = DEPT.Dno as an implicit join condition

Adapted from Navathe 7th Edition

INNER and OUTER Joins

- INNER JOIN **(versus** OUTER JOIN**)**
  - Default type of join in a joined table
  - Tuple is included in the result only if a matching tuple exists in the other relation
- LEFT OUTER JOIN
  - Every tuple in left table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of right table
- RIGHT OUTER JOIN
  - Every tuple in right table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of left table

Example: LEFT OUTER JOIN

```
SELECT E.Lname AS Employee_Name
              S.Lname AS Supervisor_Name

FROM Employee AS E LEFT OUTER JOIN EMPLOYEE AS S
                          ON E.Super_ssn = S.Ssn)
```

ALTERNATE SYNTAX:

```
SELECT E.Lname , S.Lname
FROM  EMPLOYEE E, EMPLOYEE S
WHERE E.Super_ssn + = S.Ssn
```

Adapted from Navathe 7th Edition

Multiway JOIN in the FROM clause

- FULL OUTER JOIN – combines result if LEFT and RIGHT OUTER JOIN
- Can nest JOIN specifications for a multiway join:

```
        Q2A:  SELECT Pnumber, Dnum, Lname, Address, Bdate
              FROM   ((PROJECT JOIN DEPARTMENT ON
          Dnum=Dnumber)  JOIN EMPLOYEE ON
Mgr_ssn=Ssn)
              WHERE   Plocation='Stafford';
```

Adapted from Navathe 7th Edition

Aggregate Functions in SQL

- Used to summarize information from multiple tuples into a single-tuple summary
- Built-in aggregate functions
  - **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**
- **Grouping**
  - Create subgroups of tuples before summarizing
- To select entire groups, HAVING clause is used
- Aggregate functions can be used in the SELECT clause or in a HAVING clause

Renaming Results of Aggregation

- Following query returns a single row of computed values from EMPLOYEE table:

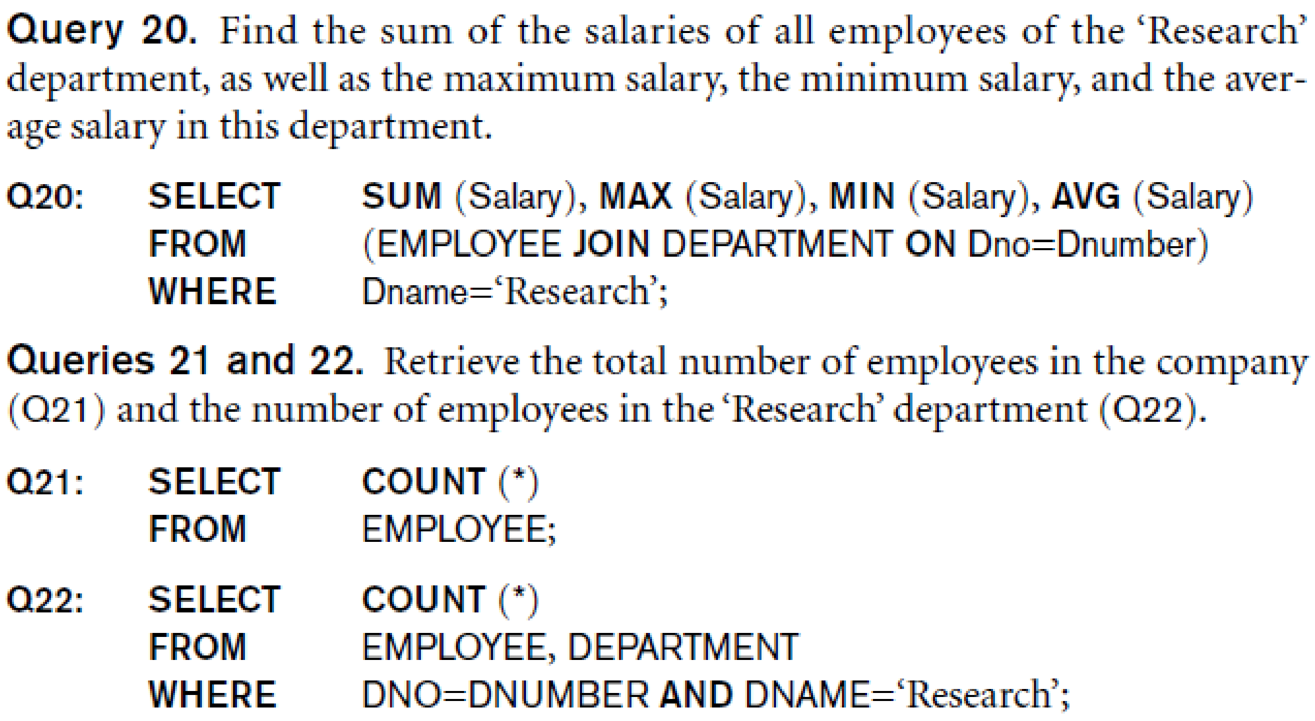
```
Q19:        SELECT   SUM (Salary), MAX (Salary), MIN (Salary), AVG
                 (Salary)
             FROM     EMPLOYEE;
```

- The result can be presented with new names:

```
Q19A:        SELECT  SUM (Salary) AS Total_Sal, MAX (Salary) AS
        Highest_Sal, MIN (Salary) AS Lowest_Sal, AVG                   (Salary)
AS Average_Sal
             FROM     EMPLOYEE;
```

Aggregate Functions in SQL (cont’d.)

- NULL values are discarded when aggregate functions are applied to a particular column



Query 20. Find the sum of the salaries of all employees of the ‘Research’ department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20:    SELECT    SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
        FROM      (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
        WHERE     Dname=‘Research’;
```

Queries 21 and 22. Retrieve the total number of employees in the company (Q21) and the number of employees in the ‘Research’ department (Q22).

```
Q21:    SELECT    COUNT (*)
        FROM      EMPLOYEE;

Q22:    SELECT    COUNT (*)
        FROM      EMPLOYEE, DEPARTMENT
        WHERE     DNO=DNUMBER AND DNAME=‘Research’;
```

Adapted from Navathe 7th Edition

Aggregate Functions on Booleans

- SOME and ALL may be applied as functions on Boolean Values.
- SOME returns true if at least one element in the collection is TRUE (similar to OR)
- ALL returns true if all of the elements in the collection are TRUE (similar to AND)

Grouping: The GROUP BY Clause

- **Partition** relation into subsets of tuples
  - Based on **grouping attribute(s)**
  - Apply function to each such group independently
- **`GROUP BY`** clause
  - Specifies grouping attributes
- COUNT (*) counts the number of rows in the group

Examples of GROUP BY

- The grouping attribute must appear in the SELECT clause:

```
Q24:            SELECT      Dno, COUNT (*), AVG (Salary)
        FROM        EMPLOYEE
        GROUP BY    Dno;
```

- If the grouping attribute has NULL as a possible value, then a separate group is created for the null value (e.g., null Dno in the above query)
- GROUP BY may be applied to the result of a JOIN:

```
Q25:    SELECT      Pnumber, Pname, COUNT (*)
        FROM        PROJECT, WORKS_ON
        WHERE           Pnumber=Pno
        GROUP BY    Pnumber, Pname;
```

Grouping: The GROUP BY and HAVING Clauses (cont'd.)

- **`HAVING`** clause
  - Provides a condition to select or reject an entire group:
- **Query 26.** For each project *on which more than two employees work,* retrieve the project number, the project name, and the number of employees who work on the project.

```
Q26:            SELECT      Pnumber, Pname, COUNT (*)
        FROM        PROJECT, WORKS_ON
        WHERE       Pnumber=Pno
        GROUP BY    Pnumber, Pname
        HAVING      COUNT (*) > 2;
```

Combining the WHERE and the HAVING Clause

- Consider the query: we want to count the *total* number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work.

- INCORRECT QUERY:

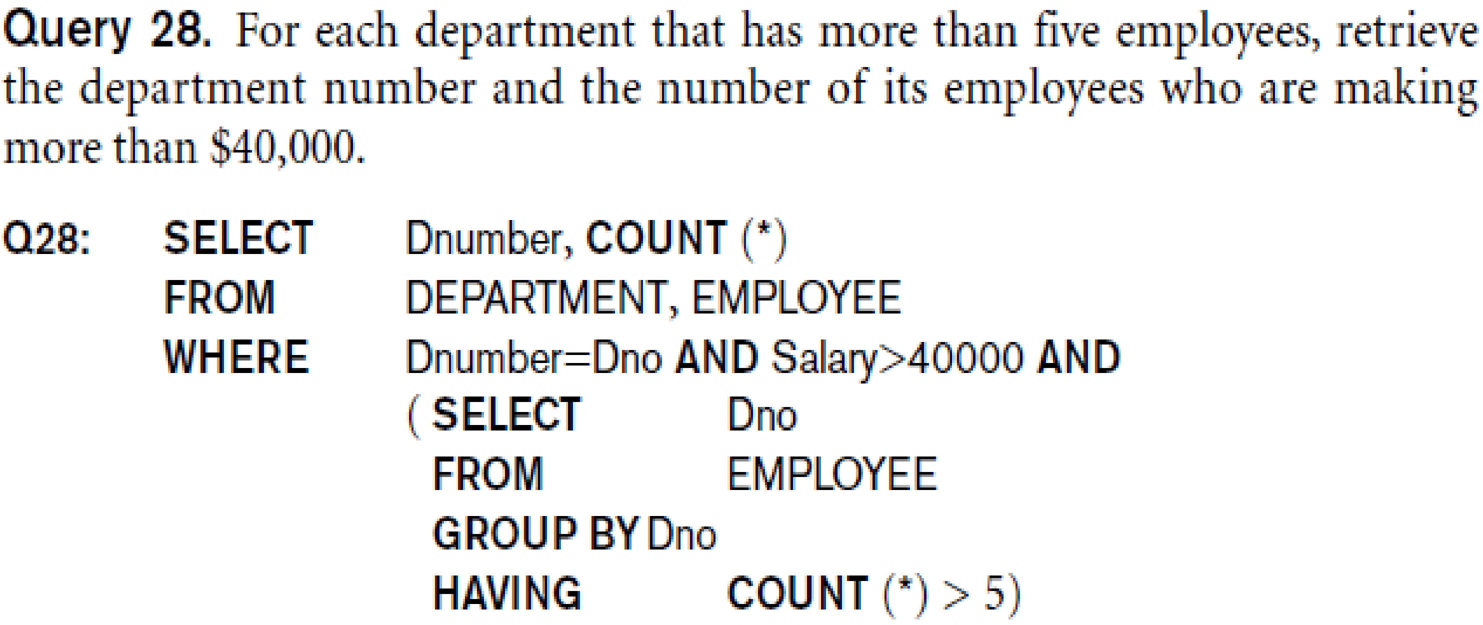
```
SELECT      Dno, COUNT (*)
FROM        EMPLOYEE
WHERE               Salary>40000
GROUP BY    Dno
HAVING              COUNT (*) > 5;
```

Adapted from Navathe 7th Edition

Combining the WHERE and the HAVING Clause (continued)

Correct Specification of the Query:

- Note: the WHERE clause applies tuple by tuple whereas HAVING applies to entire group of tuples



Query 28. For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

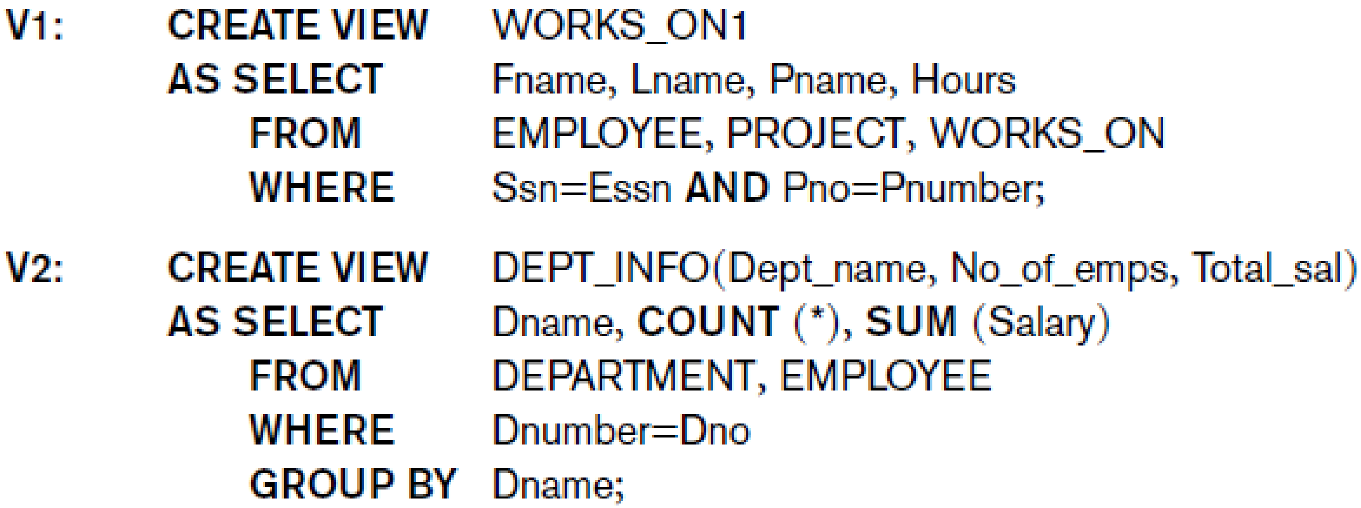
```
Q28:    SELECT    Dnumber, COUNT (*)
        FROM      DEPARTMENT, EMPLOYEE
        WHERE     Dnumber=Dno AND Salary>40000 AND
                  ( SELECT       Dno
                    FROM         EMPLOYEE
                    GROUP BY Dno
                    HAVING       COUNT (*) > 5)
```

Views (Virtual Tables) in SQL

- Concept of a view in SQL
  - Single table derived from other tables called the **defining tables**
  - Considered to be a virtual table that is not necessarily populated

Specification of Views in SQL

- **CREATE VIEW** command
  - Give table name, list of attribute names, and a query to specify the contents of the view
  - In V1, attributes retain the names from base tables. In V2, attributes are assigned names



```
V1:    CREATE VIEW   WORKS_ON1
       AS SELECT     Fname, Lname, Pname, Hours
          FROM       EMPLOYEE, PROJECT, WORKS_ON
          WHERE      Ssn=Essn AND Pno=Pnumber;

V2:    CREATE VIEW   DEPT_INFO(Dept_name, No_of_emps, Total_sal)
       AS SELECT     Dname, COUNT (*), SUM (Salary)
          FROM       DEPARTMENT, EMPLOYEE
          WHERE      Dnumber=Dno
          GROUP BY   Dname;
```

Adapted from Navathe 7th Edition

SUBQUERY

- 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*.

Example

```
SELECT emp_last_name "Last Name", emp_first_name "First
  Name",
   emp_salary "Salary"
FROM employee
WHERE emp_salary =
   (SELECT MIN(emp_salary)
    FROM employee);
Last Name    First Name   Salary
--------------- --------------- --------
Markis        Marcia        $25,000
Amin          Hyder         $25,000
Prescott      Sherri        $25,000
```

SUBQUERY TYPES

- by use of the IN operator or with a comparison operator modified by the ANY or ALL optional keywords.
- 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.

SUBQUERY – General Rules

A subquery SELECT statement is very similar to the SELECT statement used to begin a regular or outer query. The complete syntax of a subquery is shown below.

```
( SELECT subquery_select_argument
  FROM {table_name | view_name}
           {table_name | view_name} ...
  [WHERE search_conditions]
  [GROUP BY aggregate_expression [, aggregate_expression] ...]
  [HAVING search_conditions] )
```

Rules Cont’d

- The SELECT clause of a subquery must contain only one expression, only one aggregate function, or only one column name.
- The value(s) returned by a subquery must be *join-compatible* with the WHERE clause of the outer query.

Example

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

```
Last Name First Name
------------- ---------------
Bock         Douglas
Zhu          Waiman
Joyner        Suzanne
```

Rules Cont’d

Subqueries cannot manipulate their resultS internally. This means that a subquery cannot include the ORDER BY clause, the COMPUTE clause, or the INTO keyword.

SUBQUERIES AND THE IN Operator

- Subqueries that are introduced with the keyword **IN** take the general form:
  - WHERE expression [NOT] IN (subquery)

```
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');
```

```
Last Name   First Name
--------------- ---------------
Bock           Douglas
Zhu            Waiman
Joyner          Suzanne
```

SUBQUERIES AND THE IN Operator

- Conceptually, this statement is evaluated in two steps.
- First, the inner query returns the identification numbers of those employees that have male dependents.

```
SELECT dep_emp_ssn
FROM dependent
WHERE dep_gender = 'M';
DEP_EMP_S
---------
999444444
999555555
999111111
```

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.
- If a subquery returns more than one value, the Oracle Server will generate the " ORA-01427: *single-row subquery returns more than one row* " error message, and the query will fail to execute.

SUBQUERIES AND COMPARISON OPERATORS

- Let's examine a subquery that will not execute because it violates the "single value" rule.
- The query shown below returns multiple values for the *emp_salary* column.

```
SELECT emp_salary
FROM employee
  WHERE emp_salary > 40000;
EMP_SALARY
-------------------
                  55000
          43000
          43000
```

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);
ERROR at line 4:
ORA-01427: single-row subquery returns more than one row
```

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
```

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_last_name "Last Name",
   emp_first_name "First Name",
   emp_salary "Salary"
FROM employee
WHERE emp_salary > ALL
   (SELECT emp_salary
      FROM employee
      WHERE emp_dpt_number = 7);
Last Name    First Name   Salary
--------------- --------------- --------
Bordoloi      Bijoy        $55,000
```

The ANY Keyword

- The ANY keyword is not as restrictive as the ALL keyword.
- When used with the greater than comparison operator, "> ANY" means greater than some value.

```
SELECT emp_last_name "Last Name",
   emp_first_name "First Name",
   emp_salary "Salary"
FROM employee
WHERE emp_salary > ANY
   (SELECT emp_salary
       FROM employee
       WHERE emp_salary > 30000);
```

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

Subqueries and the EXISTS operator

- When a subquery uses the EXISTS operator, the subquery functions as an *existence test*.
- The WHERE clause of the outer query tests for the existence of rows returned by the inner query.
- The subquery does not actually produce any data; rather, it returns a value of TRUE or FALSE.

Subqueries and the EXISTS operator

- The general format of a subquery WHERE clause with an EXISTS operator is shown here.
- Note that the NOT operator can also be used to negate the result of the EXISTS operator.

```
WHERE [NOT] EXISTS (subquery)
```

Example

```
SELECT emp_last_name "Last Name", emp_first_name "First
 Name"
FROM employee
WHERE EXISTS
    (SELECT *
     FROM dependent
     WHERE emp_ssn = dep_emp_ssn);
```

```
Last Name  First Name
---------- ---------------
Joyner     Suzanne
Zhu        Waiman
Bock       Douglas
```

Subqueries and the EXISTS operator

- The EXISTS operator is very important, because there is often no alternative to its use.
- All queries that use the IN operator or a modified comparison operator (=, <, >, etc. modified by ANY or ALL) can be expressed with the EXISTS operator.
- However, some queries formulated with EXISTS cannot be expressed in any other way!

Subqueries and the EXISTS operator

```
SELECT emp_last_name
FROM employee
WHERE emp_ssn = ANY
    (SELECT dep_emp_ssn
     FROM dependent);



EMP_LAST_NAME
-------------
Bock
Zhu
Joyner
```

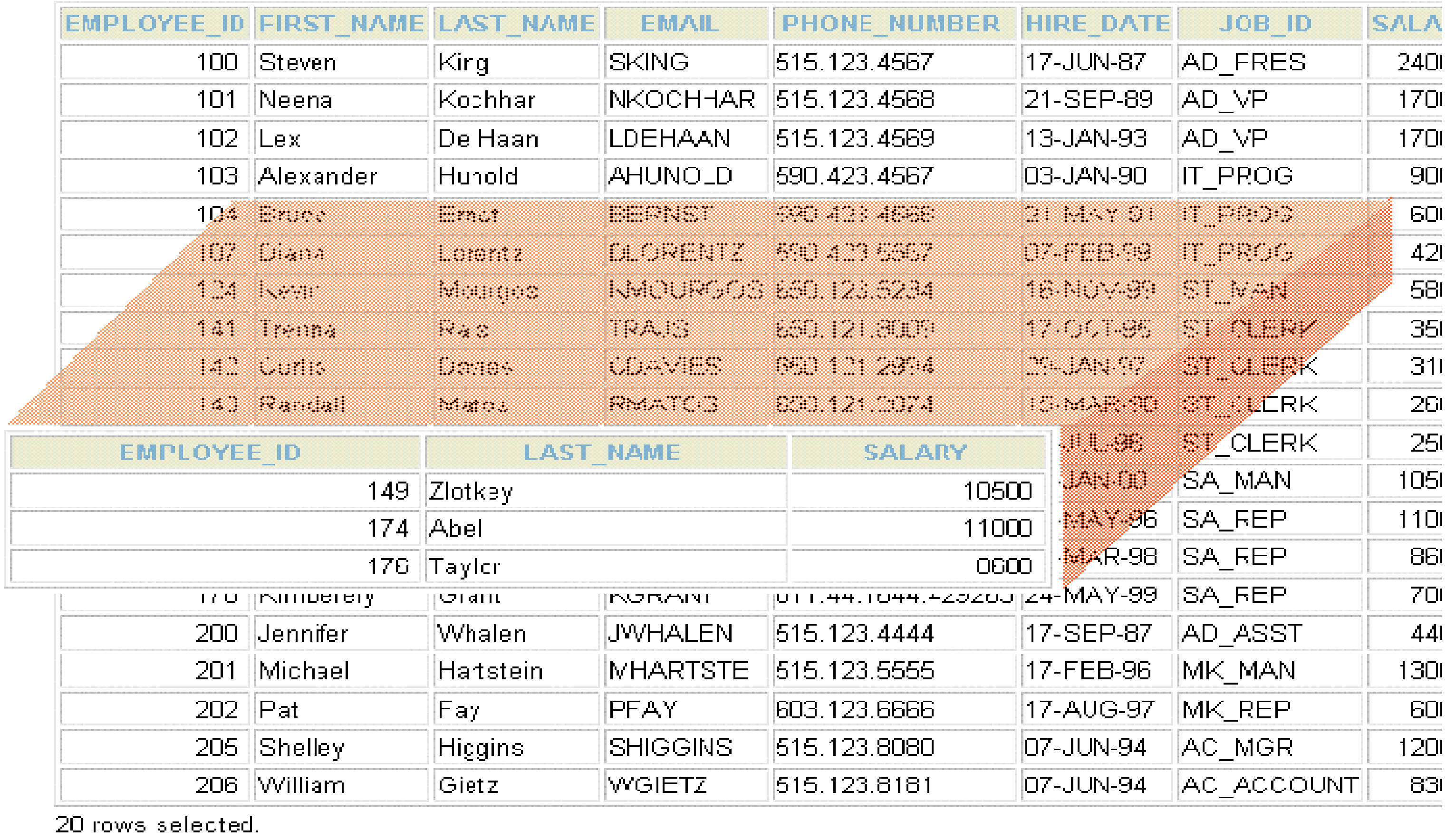
```
SELECT
   emp_last_name
FROM employee
WHERE EXISTS
    (SELECT *
     FROM dependent
     WHERE emp_ssn
   = dep_emp_ssn);

EMP_LAST_NAME
----------------
Bock
Zhu
Joyner
```

Subqueries and the EXISTS operator

The NOT EXISTS operator is the mirror-image of the EXISTS operator.

A query that uses NOT EXISTS in the WHERE clause is satisfied if the subquery returns no rows.

VIEWS



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

20 rows selected.

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

Why Use Views?

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

Creating a View

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

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

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 *i*SQL*Plus DESCRIBE command.

```
DESCRIBE empvu80
```

Creating a View

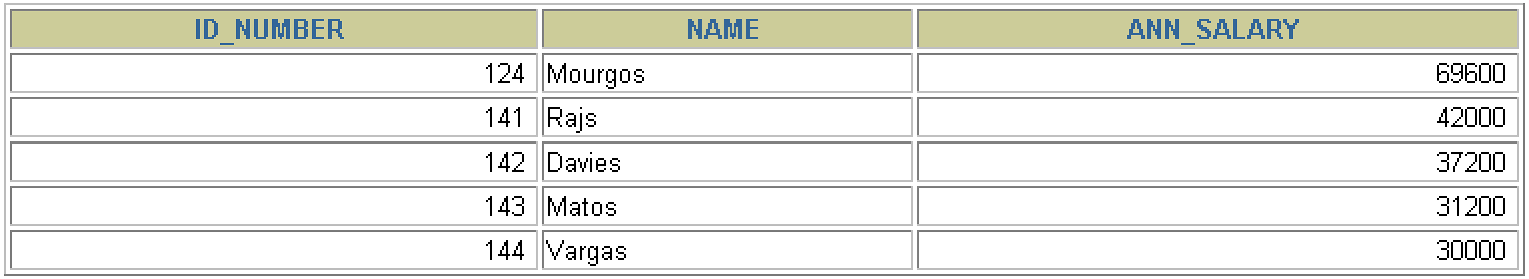
- Create a view by using column aliases in the subquery.

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

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

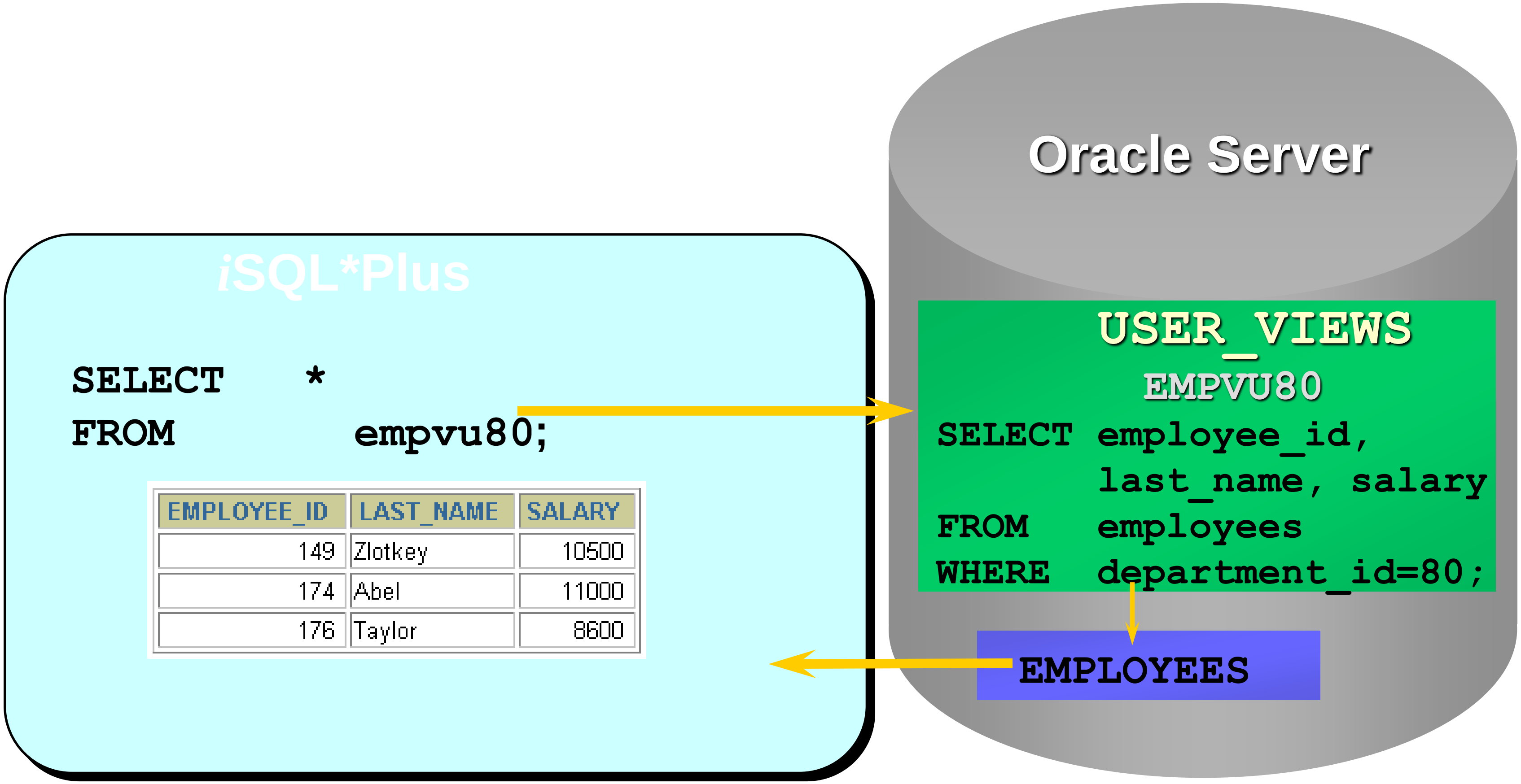
Retrieving Data from a View

```
SELECT *
FROM  salvu50;
```



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

Querying a View



```
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;
```

Summary of SQL Syntax

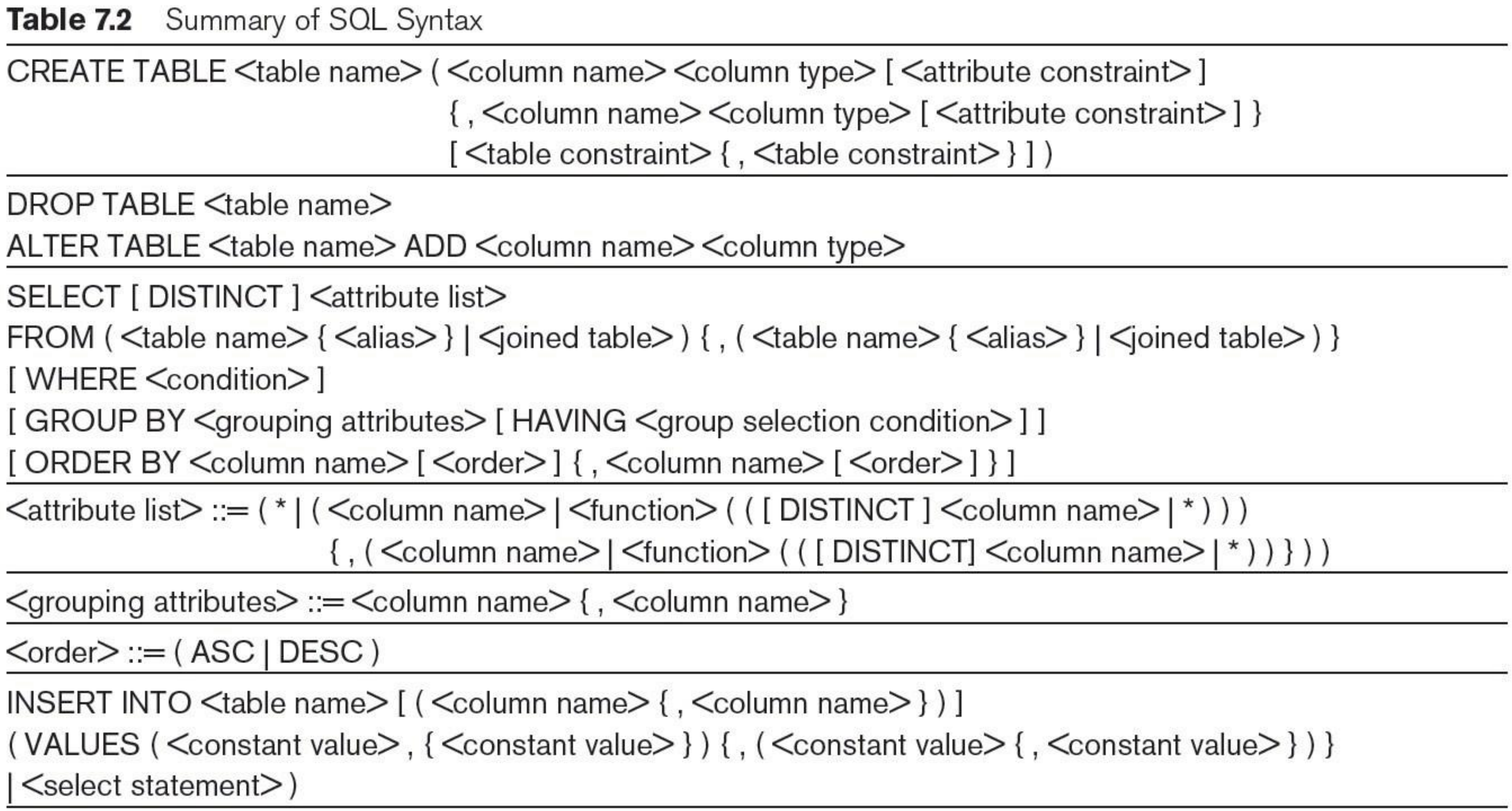


Table 7.2 Summary of SQL Syntax

```
CREATE TABLE <table name> ( <column name> <column type> [ <attribute constraint> ]
                            { , <column name> <column type> [ <attribute constraint> ] }
                            [ <table constraint> { , <table constraint> } ] )
--------------------------------------------------------------------------------
DROP TABLE <table name>
ALTER TABLE <table name> ADD <column name> <column type>
--------------------------------------------------------------------------------
SELECT [ DISTINCT ] <attribute list>
FROM ( <table name> { <alias> } | <joined table> ) { , ( <table name> { <alias> } | <joined table> ) }
[ WHERE <condition> ]
[ GROUP BY <grouping attributes> [ HAVING <group selection condition> ] ]
[ ORDER BY <column name> [ <order> ] { , <column name> [ <order> ] } ]
--------------------------------------------------------------------------------
<attribute list> ::= ( * | ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) )
                     { , ( <column name> | <function> ( ( [ DISTINCT] <column name> | * ) ) } ) )
--------------------------------------------------------------------------------
<grouping attributes> ::= <column name> { , <column name> }
--------------------------------------------------------------------------------
<order> ::= ( ASC | DESC )
--------------------------------------------------------------------------------
INSERT INTO <table name> [ ( <column name> { , <column name> } ) ]
( VALUES ( <constant value> , { <constant value> } ) { , ( <constant value> { , <constant value> } ) }
| <select statement> )
```

continued on next slide

Adapted from Navathe 7th Edition

Summary of SQL Syntax

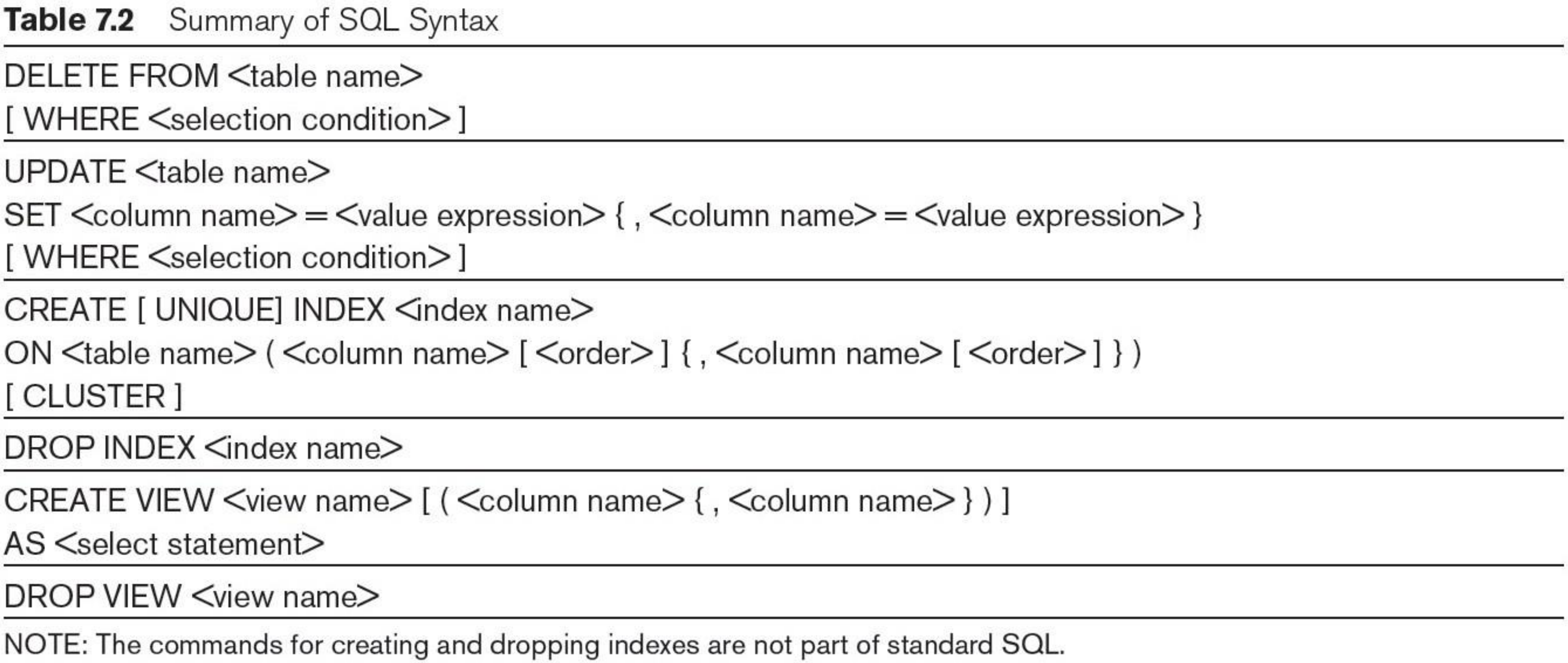


Table 7.2 Summary of SQL Syntax

DELETE FROM <table name> [WHERE <selection condition>]

UPDATE <table name>
SET <column name> = <value expression> { , <column name> = <value expression> }
[WHERE <selection condition>]

CREATE [UNIQUE] INDEX <index name> ON <table name> (<column name> [<order>] { , <column name> [<order>] }) [CLUSTER]
DROP INDEX <index name>
CREATE VIEW <view name> [(<column name> { , <column name> })] AS <select statement>
DROP VIEW <view name>

NOTE: The commands for creating and dropping indexes are not part of standard SQL.

Adapted from Navathe 7th Edition