Complex SQL Queries

- Elmisri Navathe, 6th Edition

More Complex SQL Retrieval Queries

- **5.1.1 Comparisons Involving NULL and Three-Valued Logic**
- **5.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons**
- **5.1.3 Correlated Nested Queries**
- **5.1.4 The EXISTS and UNIQUE Functions in SQL**
- **5.1.5 Explicit Sets and Renaming of Attributes in SQL**
- **5.1.6 Joined Tables in SQL and Outer Joins**
- **5.1.7 Aggregate Functions in SQL**
- **5.1.8 Grouping: The GROUP BY and HAVING Clauses**
- **5.1.9 Discussion and Summary of SQL Queries**

5.1.1 Comparisons Involving NULL and Three-Valued Logic

- SQL has various rules for dealing with NULL values.
- NULL is used to represent a missing value, but that it usually has one of three different interpretations—value *unknown*, value *not available*, or value *not applicable* .
- Consider the following examples to illustrate each of the 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.
- Hence, SQL uses a three-valued logic with values TRUE, FALSE, and UNKNOWN.

5.1.1 Comparisons Involving NULL and Three-Valued Logic

- Table 5.1 shows the resulting values.

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

5.1.1 Comparisons Involving NULL and Three-Valued Logic

- In select-project-join queries, the general rule is that only those combinations of tuples that evaluate the logical expression in the WHERE clause of the query to TRUE are selected. Tuple combinations that evaluate to FALSE or UNKNOWN are not selected.

- SQL allows queries that check whether an attribute value is **NULL**. Rather than using = or <> to compare an attribute value to NULL, SQL uses the comparison operators **IS** or **IS NOT**.

- It follows that when a join condition is specified, tuples with NULL values for the join attributes are not included in the result.

- **Query 18.** Retrieve the names of all employees who do not have supervisors.

Q18: SELECT Fname, Lname **FROM** EMPLOYEE **WHERE** Super_ssn **IS** NULL;

5.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

- **Nested queries**, are complete select-from-where blocks within the WHERE clause of another query.
- That other query is called the **outer query**.
- Q4A introduces the comparison operator **IN**, which compares a value *v* with a set (or multiset) of values *V* and evaluates to **TRUE** if *v* is one of the elements in *V*.

5.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

- Make a list of all project numbers for projects that involve an employee whose last name is ‘Smith’, either as a worker or as a manager of the department that controls the project.

```
SELECT DISTINCT Pnumber
FROM PROJECT
WHERE Pnumber IN
( SELECT Pnumber
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE Dnum=Dnumber AND
Mgr_ssn=Ssn AND Lname=‘Smith’ )
OR
Pnumber IN
( SELECT Pno
FROM WORKS_ON, EMPLOYEE
WHERE Essn=Ssn AND Lname=‘Smith’ );
```

5.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

- In general, the nested query will return a **table** (relation), which is a set or multiset of tuples.
- SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

SELECT DISTINCT Essn

FROM WORKS_ON

WHERE (Pno, Hours) **IN** (**SELECT** Pno, Hours

FROM WORKS_ON

WHERE Essn='123456789');

5.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

- Query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT Lname, Fname

FROM EMPLOYEE

WHERE Salary > ALL ( SELECT Salary

FROM EMPLOYEE

WHERE Dno=5 );
```

5.1.2 Nested Queries, Tuples, and Set/Multiset Comparisons

- **Query 16.** Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

Q16: SELECT E.Fname, E.Lname

FROM EMPLOYEE **AS** E

WHERE E.Ssn **IN** (**SELECT** Essn

FROM DEPENDENT **AS** D

WHERE E.Fname=D.Dependent_name

AND E.Sex=D.Sex);

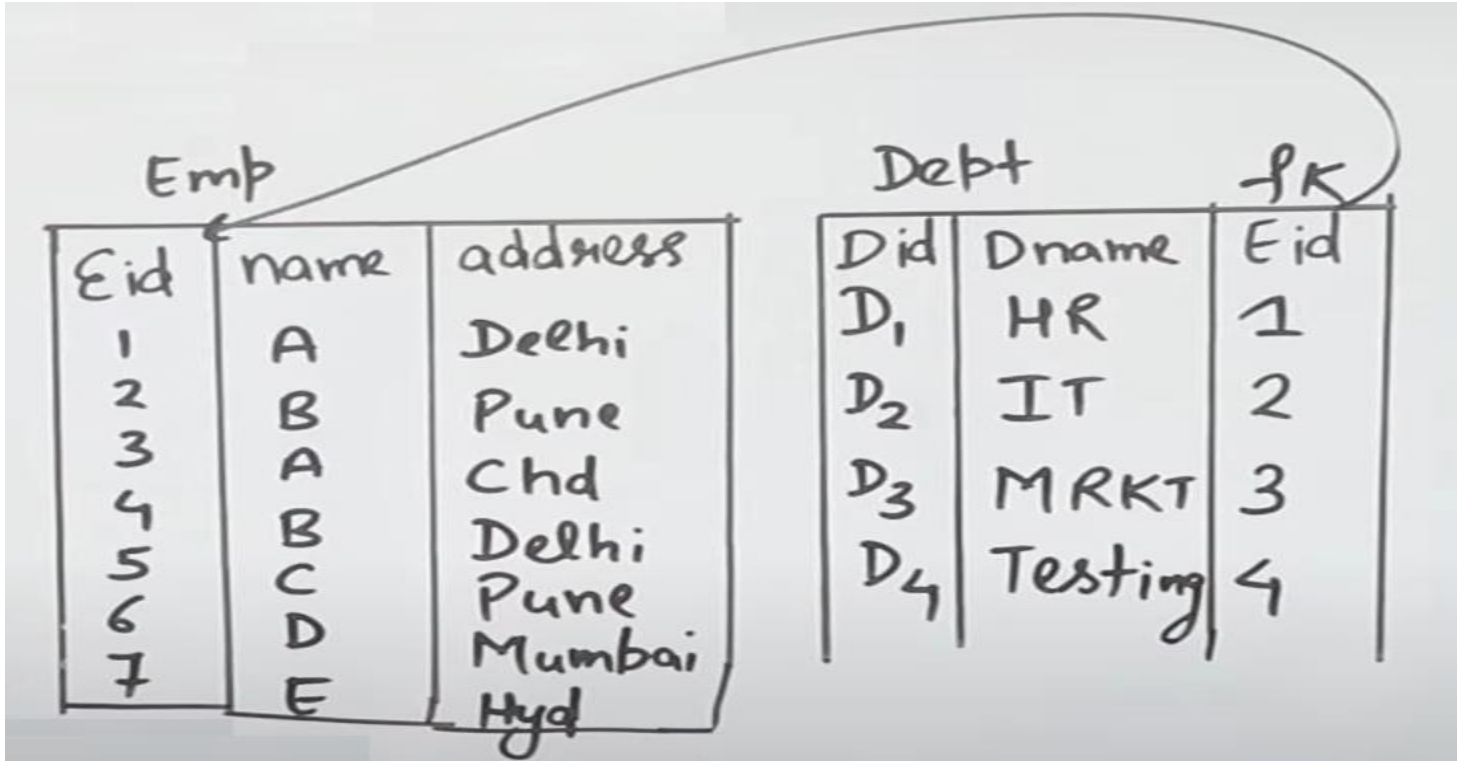
5.1.3 Correlated Nested Queries

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be **correlated**.

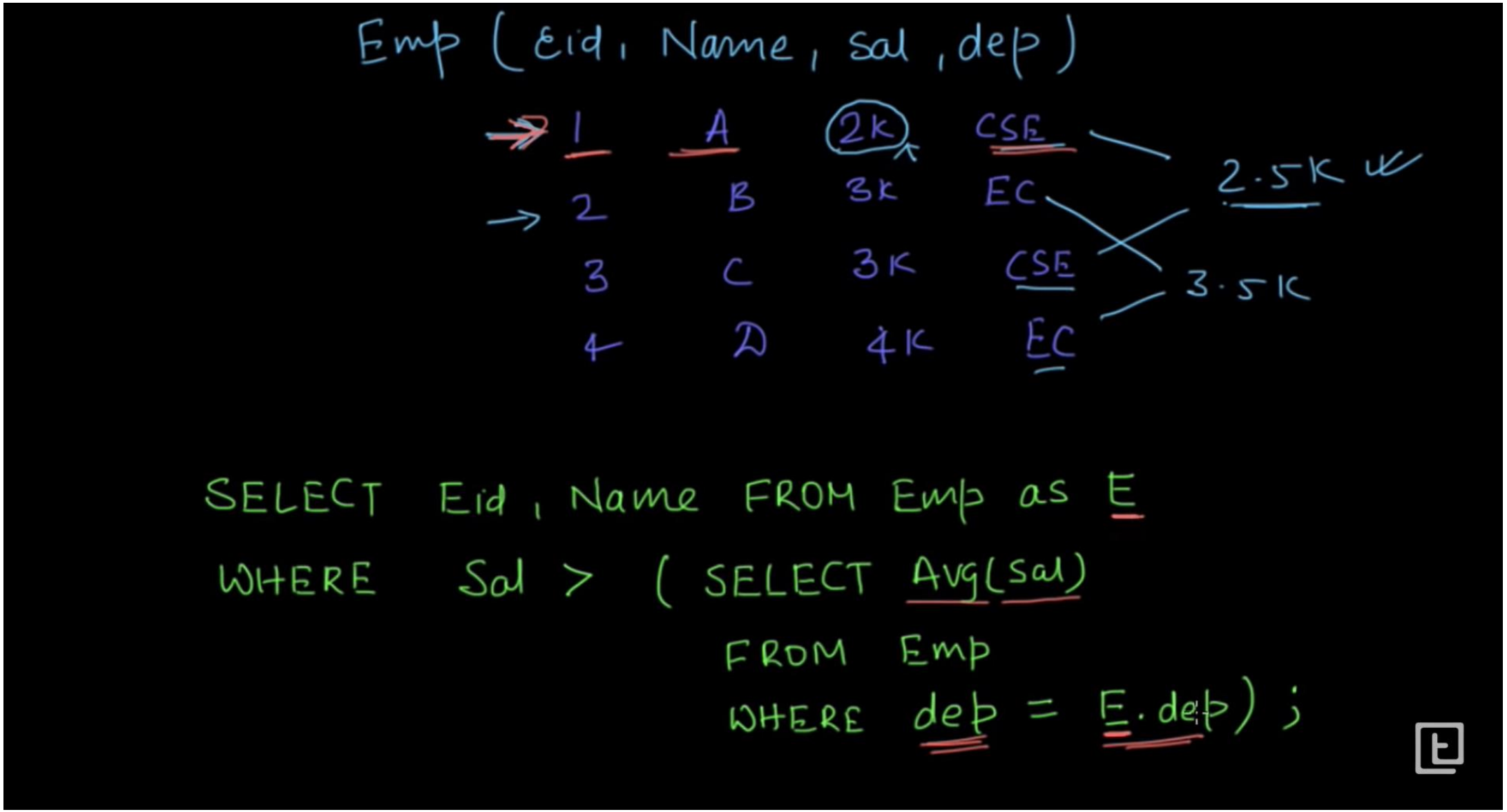
Find all employees detail who work in a department.

SELECT * from EMP where EXISTS

(SELECT * FROM DEPT where DEPT.eid = EMP.eid);



5.1.3 Correlated Nested Queries



5.1.4 The EXISTS and UNIQUE Functions in SQL

- The EXISTS function in SQL is used to check whether the result of a correlated nested query is *empty* (contains no tuples) or not.
- The result of EXISTS is a Boolean value **TRUE** if the nested query result contains at least one tuple, or **FALSE** if the nested query result contains no tuples.
- Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

Q16B: SELECT E.Fname, E.Lname

FROM EMPLOYEE **AS** E

WHERE EXISTS (**SELECT** *

FROM DEPENDENT **AS** D

WHERE E.Ssn=D.Essn **AND** E.Sex=D.Sex

AND E.Fname=D.Dependent_name);

5.1.4 The EXISTS and UNIQUE Functions in SQL

- EXISTS and NOT EXISTS are typically used in conjunction with a correlated nested query.

- In general, EXISTS(Q) returns **TRUE** if there is *at least one tuple* in the result of the nested query Q, and it returns **FALSE** otherwise. On the other hand, NOT EXISTS(Q) returns **TRUE** if there are *no tuples* in the result of nested query Q, and it returns **FALSE** otherwise.

- **Query 6.** Retrieve the names of employees who have no dependents.

Q6: SELECT Fname, Lname **FROM** EMPLOYEE

WHERE NOT EXISTS

(**SELECT** * **FROM** DEPENDENT **WHERE** Ssn=Essn);

5.1.4 The EXISTS and UNIQUE Functions in SQL

Query 7. List the names of managers who have at least one dependent.

Q7: SELECT Fname, Lname **FROM** EMPLOYEE **WHERE EXISTS**

(**SELECT** * **FROM** DEPENDENT **WHERE** Ssn=Essn)

AND

EXISTS (**SELECT** * **FROM** DEPARTMENT **WHERE** Ssn=Mgr_ssn);

There is another SQL function, UNIQUE(Q), which returns TRUE if there are no duplicate tuples in the result of query Q; otherwise, it returns FALSE. This can be used to test whether the result of a nested query is a set or a multiset.

5.1.5 Explicit Sets and Renaming of Attributes in SQL

- It is also possible to use an **explicit set of values** in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.

- **Query 17.** Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

Q17: SELECT DISTINCT Essn

FROM WORKS_ON

WHERE Pno **IN** (1, 2, 3);

5.1.5 Explicit Sets and Renaming of Attributes in SQL

- In SQL, it is possible to rename any attribute that appears in the result of a query by adding the qualifier **AS** followed by the desired new name.

Q8A: SELECT E.Lname **AS** Employee_name, S.Lname **AS** Supervisor_name

FROM EMPLOYEE **AS** E, EMPLOYEE **AS** S

WHERE E.Super_ssn=S.Ssn;

5.1.6 Joined Tables in SQL and Outer Joins

- The concept of a **joined table** (or **joined relation**) was incorporated into SQL to permit users to specify a table resulting from a join operation *in the* FROM *clause* of a query.
- This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause.
- For example, to retrieves the name and address of every employee who works for the ‘Research’ department.

Q1A: SELECT Fname, Lname, Address

FROM (EMPLOYEE **JOIN** DEPARTMENT **ON** Dno=Dnumber)

WHERE Dname=‘Research’;

5.1.6 Joined Tables in SQL and Outer Joins

Q1B: SELECT Fname, Lname, Address

FROM (EMPLOYEE **NATURAL JOIN**

(DEPARTMENT **AS** DEPT (Dname, Dno, Mssn, Msdate)))

WHERE Dname=‘Research’;

- The default type of join in a joined table is called an **inner join**, where a tuple is included in the result only if a matching tuple exists in the other relation.

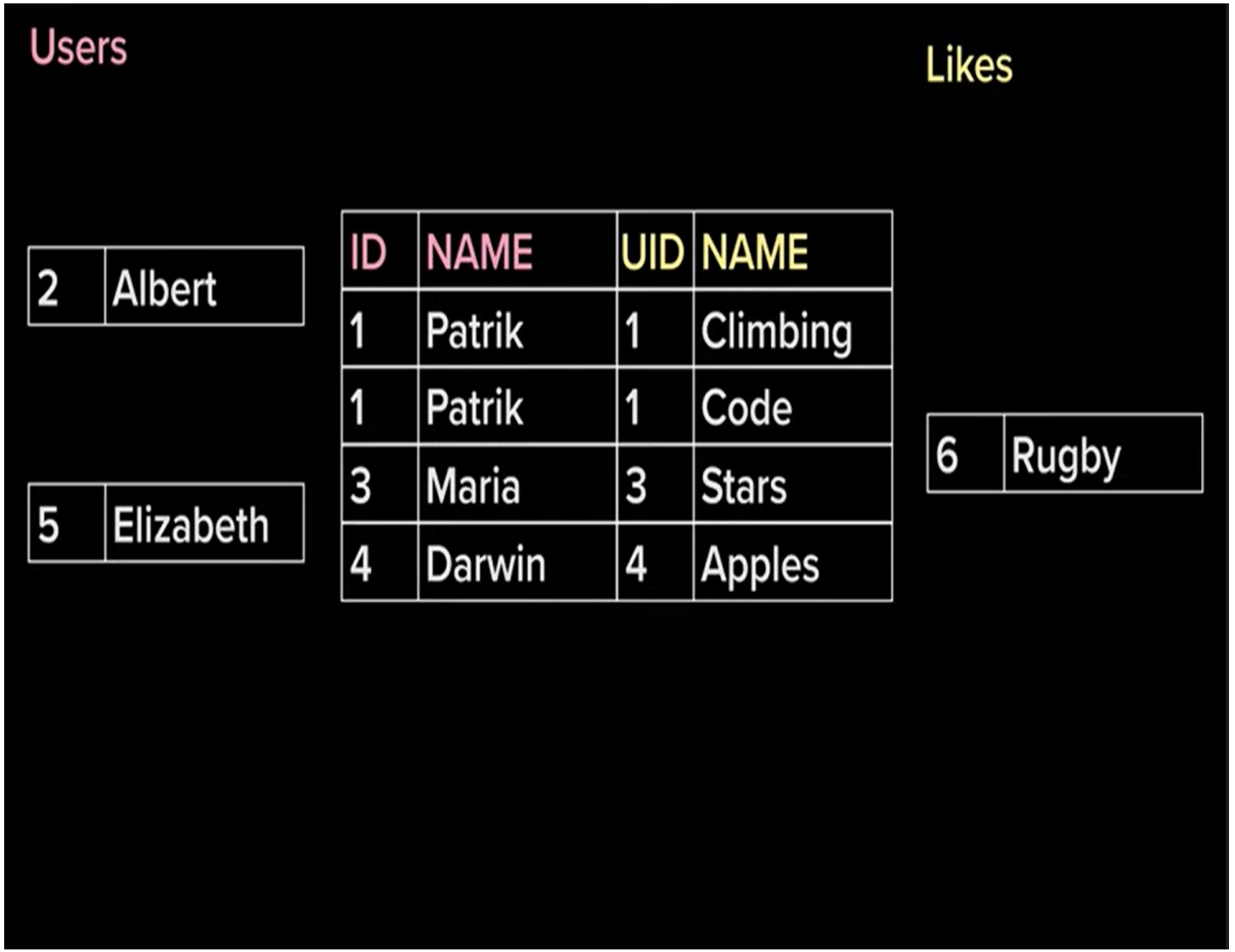
5.1.6 Joined Tables in SQL and Outer Joins

Users

ID	NAME
1	Patrik
2	Albert
3	Maria
4	Darwin
5	Elizabeth

Likes

UID	NAME
3	Stars
1	Climbing
1	Code
6	Rugby
4	Apples

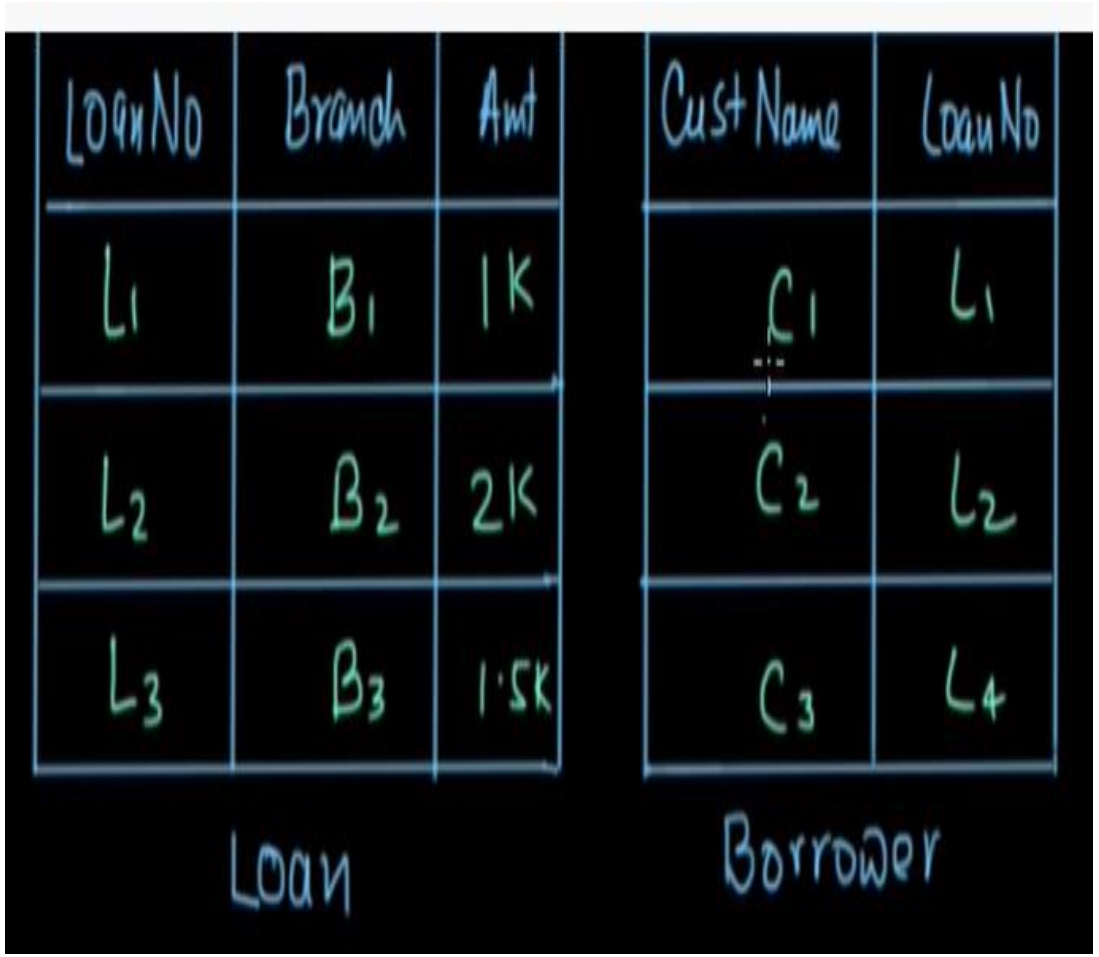


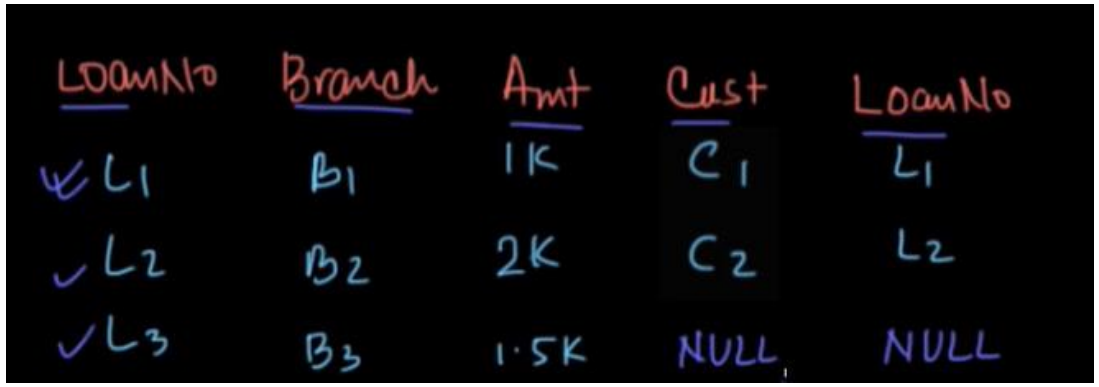
Example: inner join

5.1.6 Joined Tables in SQL and Outer Joins

- INNER JOIN - only pairs of tuples that match the join condition are retrieved, same as JOIN
- LEFT OUTER JOIN - every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the right table
- RIGHT OUTER JOIN - every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the left table), and FULL OUTER JOIN.
- If the join attributes have the same name, one can also specify the natural join variation of outer joins by using the keyword NATURAL before the operation (for example, NATURAL LEFT OUTER JOIN).
- The keyword CROSS JOIN is used to specify the CARTESIAN PRODUCT operation.
- It is also possible to *nest* join specifications; that is, one of the tables in a join may itself be a joined table. This allows the specification of the join of three or more tables as a single joined table, which is called a **multiway join**.

Left and Right outer join





Left outer Join

Right outer join

Full outer join

LoanNo	Branch	Amt
L_1	B_1	1K
L_2	B_2	2K
L_3	B_3	1·5K

Loan

Cust Name	LoanNo
C_1	L_1
C_2	L_2
C_3	L_4

Borrower

LoanNo	Branch	Amt	Cust	LoanNo
L_1	B_1	1K	C_1	L_1
L_2	B_2	2K	C_2	L_2
NULL	NULL	NULL	C_3	L_4
L_3	B_3	1·5K	NULL	NULL

Full outer join

5.1.6 Joined Tables in SQL and Outer Joins

- In an outer join, unmatched rows in one or both tables can be returned.

Q8B: 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);

5.1.6 Joined Tables in SQL and Outer Joins

- For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, address, and birth date.

Q2A: SELECT Pnumber, Dnum, Lname, Address, Bdate

FROM ((PROJECT **JOIN** DEPARTMENT **ON** Dnum=Dnumber)

JOIN EMPLOYEE **ON** Mgr_ssn=Ssn)

WHERE Plocation=‘Stafford’;

5.1.6 Joined Tables in SQL and Outer Joins

- In some systems, a different syntax was used to specify outer joins by using the comparison operators +=, =+, and +=+ for left, right, and full outer join, respectively
- For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

Q8C: SELECT E.Lname, S.Lname

FROM EMPLOYEE E, EMPLOYEE S

WHERE E.Super_ssn += S.Ssn;

5.1.7 Aggregate Functions in SQL

- **Aggregate functions** are used to summarize information from multiple tuples into a single-tuple summary.
- **Grouping** is used to create subgroups of tuples before summarization.
- A number of built-in aggregate functions exist: **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**.

Query 19. Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

Q19: SELECT SUM (Salary), **MAX** (Salary), **MIN** (Salary), **AVG** (Salary)

FROM EMPLOYEE;

5.1.7 Aggregate Functions in SQL

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;

5.1.7 Aggregate Functions in SQL

Q22: SELECT COUNT (*)

FROM EMPLOYEE, DEPARTMENT

WHERE DNO=DNUMBER **AND** DNAME='Research';

Query 23. Count the number of distinct salary values in the database.

Q23: SELECT COUNT (**DISTINCT** Salary)

FROM EMPLOYEE;

- In general, NULL values are **discarded** when aggregate functions are applied to a particular column (attribute).

5.1.7 Aggregate Functions in SQL

- We can specify a correlated nested query with an aggregate function.
- To retrieve the names of all employees who have two or more dependents

Q5: SELECT Lname, Fname

FROM EMPLOYEE

WHERE (**SELECT COUNT** (*)

FROM DEPENDENT

WHERE Ssn=Essn) >= 2;

5.1.8 Grouping: The GROUP BY and HAVING Clauses

- SQL has a **GROUP BY** clause to apply the aggregate functions *to subgroups of tuples in a relation,*

Query 24. For each department, retrieve the department number, the number of employees in the department, and their average salary.

Q24: SELECT Dno, **COUNT** (*), **AVG** (Salary)

FROM EMPLOYEE

GROUP BY Dno;

- If NULLs exist in the grouping attribute, then a **separate group** is created for all tuples with a *NULL value in the grouping attribute*.

5.1.8 Grouping: The GROUP BY and HAVING Clauses

- **Query 25.** For each project, retrieve the project number, the project name, and the number of employees who work on that project.

Q25: SELECT Pnumber, Pname, **COUNT** (*)

FROM PROJECT, WORKS_ON

WHERE Pnumber=Pno

GROUP BY Pnumber, Pname;

5.1.8 Grouping: The GROUP BY and HAVING Clauses

- SQL provides a **HAVING** clause, which can appear in conjunction with a GROUP BY clause to retrieve the values of these functions only for *groups that satisfy certain conditions.*

- **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;

5.1.8 Grouping: The GROUP BY and HAVING Clauses

Query 27. For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

Q27: SELECT Pnumber, Pname, **COUNT** (*)

FROM PROJECT, WORKS_ON, EMPLOYEE

WHERE Pnumber=Pno **AND** Ssn=Essn **AND** Dno=5

GROUP BY Pnumber, Pname;

- Notice that we must be extra careful when two different conditions apply (one to the aggregate function in the SELECT clause and another to the function in the HAVING clause).

5.1.8 Grouping: The GROUP BY and HAVING Clauses

- For example, suppose that 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.
- Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT clause.
- Suppose that we write the following *incorrect* query:

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

- This is incorrect because it will select only departments that have more than five employees *who each earn more than $40,000.*

5.1.8 Grouping: The GROUP BY and HAVING Clauses

- One way to write this query correctly is to use a nested query, as shown in Query 28.
- **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)

5.1.9 Discussion and Summary of SQL Queries

- A retrieval query in SQL can consist of up to six clauses, but only the first two— SELECT and FROM—are mandatory.
- The query can span several lines, and is ended by a semicolon.
- Query terms are separated by spaces, and parentheses can be used to group relevant parts of a query in the standard way.
- The clauses are specified in the following order, with the clauses between square brackets [...] being optional.
- The SELECT clause lists the attributes or functions to be retrieved.
- The FROM clause specifies all relations (tables) needed in the query.
- The WHERE clause specifies the conditions for selecting the tuples.

5.1.9 Discussion and Summary of SQL Queries

- GROUP BY specifies grouping attributes,
- HAVING specifies a condition on the groups being selected.
- The built-in aggregate functions are COUNT, SUM, MIN, MAX, and AVG
- ORDER BY specifies an order for displaying the result of a query.

SELECT <attribute and function list>

FROM <table list>

[**WHERE** <condition>]

[**GROUP BY** <grouping attribute(s)>]

[**HAVING** <group condition>]

[**ORDER BY** <attribute list>];

5.2 Specifying Constraints as Assertions and Actions as Triggers

- **5.2.1 Specifying General Constraints as Assertions in SQL**
- **5.2.2 Introduction to Triggers in SQL**

5.2.1 Specifying General Constraints as Assertions in SQL

- In SQL, users can specify general constraints via **declarative assertions**, using the **CREATE ASSERTION** statement of the DDL.
- Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.
- For example, to specify the constraint that *the salary of an employee must not be greater than the salary of the manager of the department that the employee works for* in SQL, we can write the following assertion:

CREATE ASSERTION SALARY_CONSTRAINT

CHECK (**NOT EXISTS** (**SELECT** *

FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D

WHERE E.Salary>M.Salary

AND E.Dno=D.Dnumber

AND D.Mgr_ssn=M.Ssn));

5.2.1 Specifying General Constraints as Assertions in SQL

- Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is **violated**.
- The constraint is **satisfied** by a database state if *no combination of tuples* in that database state violates the constraint.
- A major difference between CREATE ASSERTION and the individual domain constraints and tuple constraints is that the CHECK clauses on individual attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated*.
- On the other hand, the schema designer should use CREATE ASSERTION only in cases where it is not possible to use CHECK on attributes, domains, or tuples, so that simple checks are implemented more efficiently by the DBMS.

5.2.2 Introduction to Triggers in SQL

- In many cases it is convenient to specify the type of action to be taken when certain events occur and when certain conditions are satisfied.
- A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs.
- The condition is thus used to **monitor** the database.
- Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database.

5.2.2 Introduction to Triggers in SQL

R5: CREATE TRIGGER SALARY_VIOLATION

BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN

ON EMPLOYEE

FOR EACH ROW

WHEN (**NEW**.SALARY > (**SELECT** SALARY **FROM** EMPLOYEE

WHERE SSN = **NEW**.SUPERVISOR_SSN))

INFORM_SUPERVISOR(**NEW**.Supervisor_ssn, **NEW**.Ssn);

5.2.2 Introduction to Triggers in SQL

- A typical trigger has three components:

- 1. The **event(s)**: These are usually database update operations that are explicitly applied to the database. In this example the events are: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. The person who writes the trigger must make sure that all possible events are accounted for. These events are specified after the keyword **BEFORE** in our example, which means that the trigger should be executed before the triggering operation is executed. An alternative is to use the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.

5.2.2 Introduction to Triggers in SQL

- 2. The **condition** that determines whether the rule action should be executed: Once the triggering event has occurred, an *optional* condition may be evaluated. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the WHEN clause of the trigger.

- 3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed. In this example, the action is to execute the stored procedure INFORM_SUPERVISOR. Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates, and updating derived data automatically.

5.3 Views (Virtual Tables) in SQL

- **5.3.1 Concept of a View in SQL**
- **5.3.2 Specification of Views in SQL**
- **5.3.3 View Implementation, View Update, and Inline Views**

5.3.1 Concept of a View in SQL

- A **view** in SQL terminology is a single table that is derived from other tables. These other tables can be *base tables* or previously defined views.
- A view does not necessarily exist in physical form; it is considered to be a **virtual table**, in contrast to **base tables**, whose tuples are always physically stored in the database.
- This limits the possible update operations that can be applied to views, but it does not provide any limitations on querying a view.
- We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.

5.3.1 Concept of a View in SQL

- For example, referring to the COMPANY database we may frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the three tables EMPLOYEE,WORKS_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins. Then we can issue queries on the view, which are specified as singletable retrievals rather than as retrievals involving two joins on three tables.

- We call the EMPLOYEE,WORKS_ON, and PROJECT tables the **defining tables** of the view.

5.3.2 Specification of Views in SQL

- In SQL, the command to specify a view is **CREATE VIEW**. The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view.
- If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify new attribute names for the view, since they would be the same as the names of the attributes of the defining tables in the default case.
- The views in V1 and V2 create virtual tables

V1: CREATE VIEW WORKS_ON1

AS SELECT Fname, Lname, Pname, Hours

FROM EMPLOYEE, PROJECT, WORKS_ON

WHERE Ssn=Essn **AND** Pno=Pnumber;

5.3.1 Concept of a View in SQL

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;

- An efficient strategy for automatically updating the view table when the base tables are updated must be developed in order to keep the view up-to-date. Techniques using the concept of **incremental update** have been developed for this purpose, where the DBMS can determine what new tuples must be inserted, deleted, or modified in a *materialized view table* when a database update is applied *to one of the defining base tables*.

5.3.1 Concept of a View in SQL

- The view is generally kept as a materialized (physically stored) table as long as it is being queried.
- If the view is not queried for a certain period of time, the system may then automatically remove the physical table and recompute it from scratch when future queries reference the view.
- Updating of views is complicated and can be ambiguous. In general, an update on a view defined on a *single table* without any *aggregate functions* can be mapped to an update on the underlying base table under certain conditions.
- For a view involving joins, an update operation may be mapped to update operations on the underlying base relations in *multiple ways*. Hence, it is often not possible for the DBMS to determine which of the updates is intended.

5.3.1 Concept of a View in SQL

- To illustrate potential problems with updating a view defined on multiple tables, consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of ‘John Smith’ from ‘ProductX’ to ‘ProductY’. This view update is shown in UV1:

UV1: UPDATEWORKS_ON1

SET Pname = ‘ProductY’

WHERE Lname=‘Smith’ **AND** Fname=‘John’

AND Pname=‘ProductX’;

- This query can be mapped into several updates on the base relations to give the desired update effect on the view.

- In addition, some of these updates will create additional side effects that affect the result of other queries.

5.3.1 Concept of a View in SQL

- For example, here are two possible updates, (a) and (b), on the base relations corresponding to the view update operation in UV1:

(a): UPDATEWORKS_ON

SET Pno= (**SELECT** Pnumber

FROM PROJECT

WHERE Pname='ProductY')

WHERE Essn **IN** (**SELECT** Ssn

FROM EMPLOYEE

- **WHERE** Lname='Smith' **AND** Fname='John') would also give the desired update effect on the view, but it accomplishes this by changing the name of the 'ProductX' tuple in the PROJECT relation to 'ProductY'.

5.3.1 Concept of a View in SQL

- It is quite unlikely that the user who specified the view update UV1 wants the update to be interpreted as in (b), since it also has the side effect of changing all the view tuples with Pname = ‘ProductX’.

- Some view updates may not make much sense; for example, modifying the Total_sal attribute of the DEPT_INFO view does not make sense because Total_sal is defined to be the sum of the individual employee salaries. This request is shown as UV2:

UV2: UPDATE DEPT_INFO

SET Total_sal=100000

WHERE Dname=‘Research’;

5.3.1 Concept of a View in SQL

- A large number of updates on the underlying base relations can satisfy this view update.
- Generally, a view update is feasible when only *one possible update* on the base relations can accomplish the desired update effect on the view.
- Whenever an update on the view can be mapped to *more than one update* on the underlying base relations, we must have a certain procedure for choosing one of the possible updates as the most likely one.
- In summary, we can make the following observations:
- ■ A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint *that do not have* default values specified.

5.3.1 Concept of a View in SQL

- ■ Views defined on multiple tables using joins are generally not updatable.
- ■ Views defined using grouping and aggregate functions are not updatable.
- In SQL, the clause **WITH CHECK OPTION** must be added at the end of the view definition if a view *is to be updated*. This allows the system to check for view updatability and to plan an execution strategy for view updates.
- It is also possible to define a view table in the **FROM clause** of an SQL query. This is known as an **in-line view**. In this case, the view is defined within the query itself.

5.4 Schema Change Statements in SQL

- **5.4.1 The DROP Command**
- **5.4.2 The ALTER Command**

5.4.1 The DROP Command

- The DROP command can be used to drop *named* schema elements, such as tables, domains, or constraints.
- One can also drop a schema using the DROP SCHEMA command.
- There are two *drop behavior* options: CASCADE and RESTRICT.
- For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:
- **DROP SCHEMA** COMPANY **CASCADE**;
- If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has *no elements* in it; otherwise, the DROP command will not be executed.
- To use the RESTRICT option, the user must first individually drop each element in the schema, then drop the schema itself.

5.4.1 The DROP Command

- The relation and its definition can be deleted by using the DROP TABLE command.
- For example, if we no longer wish to keep track of dependents of employees in the COMPANY database, we can get rid of the DEPENDENT relation by issuing the following command:
- **DROP TABLE** DEPENDENT **CASCADE**;
- If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is *not referenced* in any constraints (for example, by foreign key definitions in another relation) or views or by any other elements.
- With the CASCADE option, all such constraints, views, and other elements that reference the table being dropped are also dropped automatically from the schema, along with the table itself.
- Notice that the DROP TABLE command not only deletes all the records in the table if successful, but also removes the *table definition* from the catalog.
- If it is desired to delete only the records but to leave the table definition for future use, then the
- DELETE command should be used instead of DROP TABLE.
- The DROP command can also be used to drop other types of named schema elements, such as constraints or domains.

5.4.2 The ALTER Command

- The definition of a base table or of other named schema elements can be changed by using the ALTER command.
- For base tables, the possible **alter table actions** include adding or dropping a column, changing a column definition, and adding or dropping table constraints.
- For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relation in the COMPANY schema, we can use the command
- **ALTER TABLE** COMPANY.EMPLOYEE **ADD COLUMN** Job VARCHAR(12);
- We must still enter a value for the new attribute Job for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command individually on each tuple.
- If no default clause is specified, the new attribute will have NULLs in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is *not allowed* in this case.

5.4.2 The ALTER Command

- To drop a column, we must choose either CASCADE or RESTRICT for drop behavior.
- If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column.
- If RESTRICT is chosen, the command is successful only if no views or constraints reference the column.
- For example, the following command removes the attribute Address from the EMPLOYEE base table:
- **ALTER TABLE** COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE**;
- It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause.
- The following examples illustrate this clause:
- **ALTER TABLE** COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn **DROP DEFAULT**;
- **ALTER TABLE** COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn **SET DEFAULT** '333445555';

5.4.2 The ALTER Command

- One can also change the constraints specified on a table by adding or dropping a named constraint.
- To be dropped, a constraint must have been given a name when it was specified.
- For example, to drop the constraint named EMPSUPERFK from the EMPLOYEE relation, we write:
- **ALTER TABLE** COMPANY.EMPLOYEE **DROP CONSTRAINT** EMPSUPERFK **CASCADE**;
- Once this is done, we can redefine a replacement constraint by adding a new constraint to the relation, if needed.
- This is specified by using the **ADD** keyword in the ALTER TABLE statement followed by the new constraint.

END of chapter 5