DBMS IAT -2 (AY 2021-22)-5 SEMESTER-CSE

SCHEME AND SOLUTION

Q1: Consider the following relation for a database of the company: Employee(name,Eno,sex,salary,super no, no.) Department(Dname, Dnumber, Mgr No) Dept location(Dnumber,Dlocation) Project(Pname, Pnumber, Plocation, Dnumber) WorksOn(EEno, Pno, hours) Dependent(EEno,Dependent_name,sex, relationship) Write SQL queries for the following:

(2 MARKS EACH QUERY)

i. Retrieve the names of all employees who work in the department that has the employee with the highest salary among all employees.

```
select Fname, Lname
from EMPLOYEE
where Dno = (select Dno
               from EMPLOYEE
               where Salary = (select MAX(Salary) FROM
employee));
```

ii. Retrieve the names of employees who makes at least 10000 more than the employee who is paid least in the company.

```
select Fname, Lname
from EMPLOYEE
where Salary > 10000 + (select MIN(Salary)
                        from EMPLOYEE)
```

iii. Create a view that has the employee name, supervisor name and employee salary for each employee who works in the research department.

CREATE VIEW EMP_NAME AS SELECT EMPLOYEE_NAME,SALARY,EMP_NAME AS SUPERVISOR_NAME FROM EMPLOYEE E, DEPARTMENT D WHERE DNAME=" RESEARCH" AND D.DNO=E.DNO AND E.SSN= D.MGRSSN;

iv. Retrieve project name, number of employees and total hours worked on the project for each project with more than one employee.

```
select Pname, Dname, (select COUNT(*)
                      from WORKS_ON W1
```

```
                                where W1.Pno = P1.Pnumber) as
Num_Employee,


                                (select SUM(W2.Hours)
                                from WORKS_ON W2
                                where W2.Pno = P1.Pnumber
                                group by Pno) as Total_Hours
```

v. Retrieve the names of employee who have no dependents

SELECT FNAME, LNAME FROM EMPLOYEE WHERE NOT EXISTS (SELECT * FROM DEPENDENT WHERE SSN = ESSN);

Q2:Explain the schema based constraints in relational model. (10 MARKS)

Constraints **that are directly applied in the schemas of the data model**, by specifying them in the DDL(Data Definition Language). These are called schema-based constraints or Explicit constraints. Constraints that cannot be directly applied in the schemas of the data model.

the schema-based constraints include domain constraints, key constraints, constraints on NULLs, entity integrity constraints, and referential integrity constraints.

1. Domain Constraints

Domain constraints specify that within each tuple, the value of each attribute *A* must be an atomic value from the domain dom(*A*). We have already discussed the ways in which domains can be specified in Section 3.1.1. The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and double-precision float). Characters, Booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money, or other special data types. Other possible domains may be described by a subrange of values from a data type or as an enumerated data type in which all possible values are explicitly listed. Rather than describe these in detail here, we discuss the data types offered by the SQL relational standard.

2. Key Constraints and Constraints on NULL Values

In the formal relational model, a *relation* is defined as a *set of tuples*. By definition, all elements of a set are distinct; hence, all tuples in a relation must also be distinct. This means that no two tuples can have the same combination of values for *all* their attributes. Usually, there are other **subsets of attributes** of a relation schema *R* with the property that no two tuples in any relation state *r* of *R* should have the same combination of values for these attributes. Suppose that we denote one such subset of attributes by SK; then for any two *distinct* tuples *t*1 and *t*2 in a relation state *r* of *R*, we have the constraint that:

$$t_1[SK] \neq t_2[SK]$$

Any such set of attributes SK is called a **superkey** of the relation schema *R*. A superkey SK specifies a *uniqueness constraint* that no two distinct tuples in any state *r* of *R* can have the same value for SK. Every relation has at least one default superkey—the set of all its attributes. A superkey can have redundant attributes, however, so a more useful concept is that of a *key*, which has no redundancy. A **key** *K* of a relation schema *R* is a superkey of *R* with the additional property that removing any attribute *A* from *K* leaves a set of attributes *K* that is not a superkey of *R* any more. Hence, a key satisfies two properties:

Two distinct tuples in any state of the relation cannot have identical values for (all) the attributes in the key. This first property also applies to a superkey.

It is a *minimal superkey*—that is, a superkey from which we cannot remove any attributes and still have the uniqueness constraint in condition 1 hold. This property is not required by a superkey.

Whereas the first property applies to both keys and superkeys, the second property is required only for keys. Hence, a key is also a superkey but not vice versa. Consider the STUDENT relation of Figure 3.1. The attribute set {Ssn} is a key of STUDENT because no two student tuples can have the same value for Ssn.8 Any set of attrib-utes that includes Ssn—for example, {Ssn, Name, Age}—is a superkey. However, the superkey {Ssn, Name, Age} is not a key of STUDENT because removing Name or Age or both from the set still leaves us with a superkey. In general, any superkey formed from a single attribute is also a key. A key with multiple attributes must require *all* its attributes together to have the uniqueness property.

The value of a key attribute can be used to identify uniquely each tuple in the rela-tion. For example, the Ssn value 305-61-2435 identifies uniquely the tuple corresponding to Benjamin Bayer in the STUDENT relation. Notice that a set of attributes constituting a key is a property of the relation schema; it is a constraint that should hold on *every* valid relation state of the schema. A key is determined from the mean-ing of the attributes, and the property is *time-invariant:* It must continue to hold when we insert new tuples in the relation. For example, we cannot and should not designate the Name attribute of the STUDENT relation in Figure 3.1 as a key because it is possible that two students with identical names will exist at some point in a valid state.9

In general, a relation schema may have more than one key. In this case, each of the keys is called a **candidate key**. For example, the CAR relation in Figure 3.4 has two candidate keys: License_number and Engine_serial_number. It is common to designate one of the candidate keys as the **primary key** of the relation. This is the candidate key whose values are used to *identify* tuples in the relation. We use the convention that the attributes that form the primary key of a relation schema are underlined, as shown in Figure 3.4. Notice that when a relation schema has several candidate keys,

CAR

License_number	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Figure 3.4
The CAR relation, with two candidate keys: License_number and Engine_serial_number.

the choice of one to become the primary key is somewhat arbitrary; however, it is usually better to choose a primary key with a single attribute or a small number of attributes. The other candidate keys are designated as **unique keys**, and are not underlined.

Another constraint on attributes specifies whether NULL values are or are not permitted. For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL.

3. Relational Databases and Relational Database Schemas

The definitions and constraints we have discussed so far apply to single relations and their attributes. A relational database usually contains many relations, with tuples in relations that are related in various ways. In this section we define a rela-tional database and a relational database schema.

A **relational database schema** *S* is a set of relation schemas S = {*R*1, *R*2, ..., *Rm*} and a set of **integrity constraints** IC. A **relational database state** DB of *S* is a set of relation states DB = {*r*1, *r*2, ..., *rm*} such that each *ri* is a state of *Ri* and such that the *ri* relation states satisfy the integrity constraints specified in IC. Figure 3.5 shows a relational database schema that we call COMPANY = {EMPLOYEE, DEPARTMENT,

DEPT_LOCATIONS, PROJECT, WORKS_ON, DEPENDENT}. The underlined attributes represent primary keys. Figure 3.6 shows a relational database state corresponding to the COMPANY schema. We will use this schema and database state in this chapter and in Chapters 4 through 6 for developing sample queries in different relational languages. (The data shown here is expanded and available for loading as a populated database from the Companion Website for the book, and can be used for the hands-on project exercises at the end of the chapters.)

When we refer to a relational database, we implicitly include both its schema and its current state. A database state that does not obey all the integrity constraints is

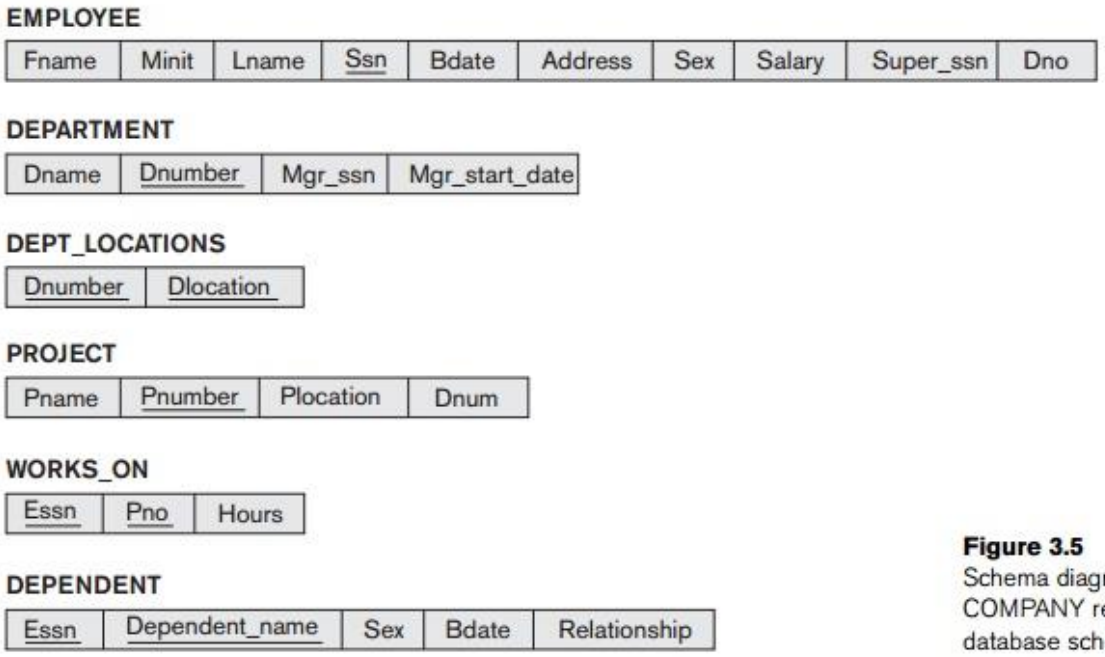


Figure 3.5
Schema diagram for the COMPANY relational database schema.

called an **invalid state**, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state.**

In Figure 3.5, the Dnumber attribute in both DEPARTMENT and DEPT_LOCATIONS stands for the same real-world concept—the number given to a department. That same concept is called Dno in EMPLOYEE and Dnum in PROJECT. Attributes that represent the same real-world concept may or may not have identical names in different relations. Alternatively, attributes that represent different concepts may have the same name in different relations. For example, we could have used the attribute name Name for both Pname of PROJECT and Dname of DEPARTMENT; in this case, we would have two attributes that share the same name but represent different real-world concepts—project names and department names.

In some early versions of the relational model, an assumption was made that the same real-world concept, when represented by an attribute, would have *identical* attribute names in all relations. This creates problems when the same real-world concept is used in different roles (meanings) in the same relation. For example, the concept of Social Security number appears twice in the EMPLOYEE relation of Figure 3.5: once in the role of the employee's SSN, and once in the role of the super-visor's SSN. We are required to give them distinct attribute names—Ssn and Super_ssn, respectively—because they appear in the same relation and in order to distinguish their meaning.

Each relational DBMS must have a data definition language (DDL) for defining a relational database schema. Current relational DBMSs are mostly using SQL for this purpose. We present the SQL DDL in Sections 4.1 and 4.2.

Figure 3.6
One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Integrity constraints are specified on a database schema and are expected to hold on every valid database state of that schema. In addition to domain, key, and NOT NULL constraints, two other types of constraints are considered part of the relational model: entity integrity and referential integrity.

4. Integrity, Referential Integrity, and Foreign Keys

The **entity integrity constraint** states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation. Having NULL values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

Key constraints and entity integrity constraints are specified on individual relations. The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation. For example, in Figure 3.6, the attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.

To define referential integrity more formally, first we define the concept of a *foreign key*. The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas $R1$ and $R2$. A set of attributes FK in rela-tion schema $R1$ is a **foreign key** of $R1$ that **references** relation $R2$ if it satisfies the following rules:

The attributes in FK have the same domain(s) as the primary key attributes PK of $R2$; the attributes FK are said to **reference** or **refer to** the relation $R2$.

A value of FK in a tuple $t1$ of the current state $r1(R1)$ either occurs as a value of PK for some tuple $t2$ in the current state $r2(R2)$ *or is NULL*. In the former case, we have $t1[FK] = t2[PK]$, and we say that the tuple $t1$ **references** or **refers to** the tuple $t2$.

In this definition, $R1$ is called the **referencing relation** and $R2$ is the **referenced rela-tion**. If these two conditions hold, a **referential integrity constraint** from $R1$ to $R2$ is said to hold. In a database of many relations, there are usually many referential integrity constraints.

To specify these constraints, first we must have a clear understanding of the mean-ing or role that each attribute or set of attributes plays in the various relation schemas of the database. Referential integrity constraints typically arise from the *relationships among the entities* represented by the relation schemas. For example, consider the database shown in Figure 3.6. In the EMPLOYEE relation, the attribute Dno refers to the department for which an employee works; hence, we designate Dno to be a foreign key of

EMPLOYEE referencing the DEPARTMENT relation. This means that a value of Dno in any tuple *t*1 of the EMPLOYEE relation must match a value of

the primary key of DEPARTMENT—the Dnumber attribute—in some tuple *t*2 of the DEPARTMENT relation, or the value of Dno *can be NULL* if the employee does not belong to a department or will be assigned to a department later. For example, in Figure 3.6 the tuple for employee 'John Smith' references the tuple for the 'Research' department, indicating that 'John Smith' works for this department.

Notice that a foreign key can *refer to its own relation.* For example, the attribute Super_ssn in EMPLOYEE refers to the supervisor of an employee; this is another employee, represented by a tuple in the EMPLOYEE relation. Hence, Super_ssn is a foreign key that references the EMPLOYEE relation itself. In Figure 3.6 the tuple for employee 'John Smith' references the tuple for employee 'Franklin Wong,' indicating that 'Franklin Wong' is the supervisor of 'John Smith.'

We can *diagrammatically display referential integrity constraints* by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation. Figure 3.7 shows the schema in Figure 3.5 with the referential integrity constraints displayed in this manner.

All integrity constraints should be specified on the relational database schema (i.e., defined as part of its definition) if we want to enforce these constraints on the data-base states. Hence, the DDL includes provisions for specifying the various types of constraints so that the DBMS can automatically enforce them. Most relational DBMSs support key, entity integrity, and referential integrity constraints. These constraints are specified as a part of data definition in the DDL.

Q3:How trigger and assertion can be defined in SQL? Explain with example.

(TRIGGER- 3(EXPLANATION)+2 MARKS(EXAMPLE)+ ASSERTION- 3 MARKS
(EXPLANATION)+ 2MARKS(EXAMPLE))

1. Assertions:-

When a constraint involves 2 (or) more tables, the table constraint mechanism is sometimes hard and results may not come as expected. To cover such situation SQL supports the creation of assertions that are constraints not associated with only one table. And an assertion statement should ensure a certain condition will always exist in the database. DBMS always checks the assertion whenever modifications are done in the corresponding table.

```
CREATE ASSERTION  [ assertion_name ]
CHECK ( [ condition ] );
```

Example –

```
CREATE TABLE sailors (sid int,sname varchar(20), rating int,primary
key(sid),
```

```
CHECK(rating >= 1 AND rating <=10)
CHECK((select count(s.sid) from sailors s) + (select count(b.bid)from
boats b)<100) );
```

In the above example, we enforcing CHECK constraint that the number of boats and sailors should be less than 100. So here we are able to CHECK constraints of two tablets simultaneously.

. Triggers:

A trigger is a database object that is associated with the table, it will be activated when a defined action is executed for the table. The trigger can be executed when we run the following statements:

1. INSERT
2. UPDATE
3. DELETE

And it can be invoked before or after the event. Syntax –

```
create trigger [trigger_name]
[before | after]
{insert | update | delete}
on [table_name]
[for each row]
[trigger_body]
```

Example –

```
create trigger t1  before  UPDATE on sailors
for each row
begin
   if new.age>60 then
      set new.age=old.age;
   else
      set new.age=new.age;
   end if;
end;
$
```

In the above example, we are creating triggers before updates. so, if the new age is greater than 60 we should not update else we should update. We can call this trigger by using “$” symbol.

Difference between Assertions and Triggers :

S.No	Assertions	Triggers
1.	We can use Assertions when we know that the given particular condition is always true.	We can use Triggers even particular condition may or may not be true.
2.	When the SQL condition is not met then there are chances to an entire table or even Database to get locked up.	Triggers can catch errors if the condition of the query is not true.
3.	Assertions are not linked to specific table or event. It performs task specified or defined by the user.	It helps in maintaining the integrity constraints in the database tables, especially when the primary key and foreign key constraint are not defined.
4.	Assertions do not maintain any track of changes made in table.	Triggers maintain track of all changes occurred in table.
5.	Assertions have small syntax compared to Triggers.	They have large Syntax to indicate each and every specific of the created trigger.

6.	Modern databases do not use Assertions.	Triggers are very well used in modern databases.

Assertions can't modify the data and they are not linked to any specific tables or events in the database but Triggers are more powerful because they can check conditions and also modify the data within the tables inside a database, unlike assertions.

Q4:How is a view created and dropped? What problems are associated with updating views?

(VIEW CREATION- 3 MARKS, DROP- 3 MARKS,PROBLEMS ASSOCIATED WITH UPDATING VIEWS- 4 MARKS)

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.

Creating Views

Database views are created using the CREATE VIEW statement. Views can be created from a single table, multiple tables or another view. To create a view, a user must have the appropriate system privilege according to the specific implementation. CREATE VIEW view_name AS SELECT column1, column2…

How do I drop a view?

Use the DROP VIEW statement to remove a view or an object view from the database. You can change the definition of a view by dropping and re-creating it. The view must be in your own schema or you must have the DROP ANY VIEW system privilege. Specify the schema containing the view.

CREATING VIEWS

We can create View using CREATE VIEW statement. A View can be created from a single table or multiple tables.

Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE condition;

view_name: Name for the View
table_name: Name of the table
condition: Condition to select rows
```

Examples:

- Creating View from a single table:

In this example we will create a View named DetailsView from the table StudentDetails.
Query:

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5;
```

To see the data in the View, we can query the view in the same manner as we query a table.

```
SELECT * FROM DetailsView;
```

- Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

In this example, we will create a view named StudentNames from the table StudentDetails.
Query:

```
CREATE VIEW StudentNames AS
SELECT S_ID, NAME
FROM StudentDetails
ORDER BY NAME;
```

If we now query the view as,

```
SELECT * FROM StudentNames;
```

- Output:

S_ID	NAMES
2	Ashish
4	Dhanraj
1	Harsh
3	Pratik
5	Ram

Creating View from multiple tables: In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement. Query:

```
CREATE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
StudentMarks.MARKS
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

To display data of View MarksView:

```
SELECT * FROM MarksView;
```

- Output:

NAME	ADDRESS	MARKS
Harsh	Kolkata	90
Pratik	Delhi	80
Dhanraj	Bihar	95
Ram	Rajasthan	85

DELETING VIEWS

We have learned about creating a View, but what if a created View is not needed any more? Obviously we will want to delete it. SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

Syntax:

```
DROP VIEW view_name;

view_name: Name of the View which we want to delete.
```

For example, if we want to delete the View MarksView, we can do this as:

```
DROP VIEW MarksView;
```

UPDATING VIEWS

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is not met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

We can use the CREATE OR REPLACE VIEW statement to add or remove fields from a view.
Syntax:

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1,coulmn2,..
FROM table_name
WHERE condition;
```

For example, if we want to update the view MarksView and add the field AGE to this View from StudentMarks Table, we can do this as:

```
CREATE OR REPLACE VIEW MarksView AS
SELECT StudentDetails.NAME, StudentDetails.ADDRESS,
StudentMarks.MARKS, StudentMarks.AGE
FROM StudentDetails, StudentMarks
WHERE StudentDetails.NAME = StudentMarks.NAME;
```

If we fetch all the data from MarksView now as:

```
SELECT * FROM MarksView;
```

- Output:

NAME	ADDRESS	MARKS	AGE
Harsh	Kolkata	90	19
Pratik	Delhi	80	19
Dhanraj	Bihar	95	21
Ram	Rajasthan	85	18

Inserting a row in a view:
We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.Syntax:

```
INSERT INTO view_name(column1, column2 , column3,..)
VALUES(value1, value2, value3..);

view_name: Name of the View
```

Example:
In the below example we will insert a new row in the View DetailsView which we have created above in the example of "creating views from a single table".

```
INSERT INTO DetailsView(NAME, ADDRESS)
VALUES("Suresh","Gurgaon");
```

If we fetch all the data from DetailsView now as,

```
SELECT * FROM DetailsView;
```

- Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar
Suresh	Gurgaon

Deleting a row from a View:
Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.Syntax:

```
DELETE FROM view_name
WHERE condition;

view_name:Name of view from where we want to delete rows
condition: Condition to select rows
```

Example:
In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.

```
DELETE FROM DetailsView
```

```
WHERE NAME="Suresh";
```

If we fetch all the data from DetailsView now as,

```
SELECT * FROM DetailsView;
```

- Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

Q5:Discuss how each ER model can be mapped to the relational model. (ALGORITHM- 7 MARKS+EXPLANATION- 3 MARKS)

Database Design

Goal of design is to generate a formal specification of the database schema

Methodology:

1. Use E-R model to get a high-level graphical view of essential components of enterprise and how they are related
2. Then convert E-R diagram to SQL Data Definition Language (DDL), or whatever database model you are using

E-R Model is not SQL based.

The E-R Model: The database represented is viewed as a graphical drawing of

- Entities and attributes
- Relationships among those entities
- --not tables!

Relational Model: The database is viewed as a

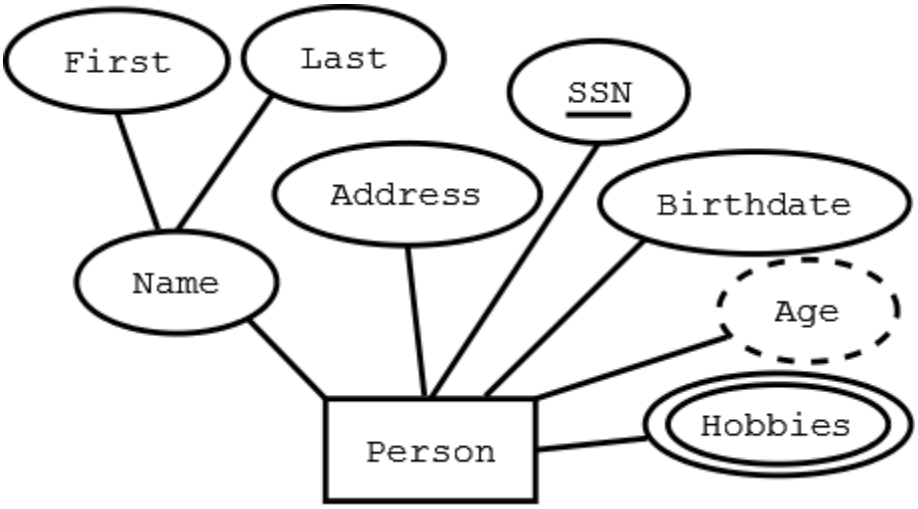
- Tables
- and their attributes (keys)
- --we could include constraints but will not at this stage.

Note on attributes:

- not all attributes in the relational model are attributes on an entity; we may have attributes that are establishing relationships.

Representation of Entity Type in Relational Model

Mapping #1 (ENTITIES): Each entity type always corresponds to a relation



---> Person(....)

Mapping #2 (ATTRIBUTES): The attributes of a relation contains at least the simple attributes of an entity type

- Attributes are single valued
- There may be additional attributes (foreign keys) in the table, not found in the ER diagram
  Persons(SSN, FirstName, LastName, Address, Birthdate)

Problem: Recall that the entity type can have multi-valued attributes.

Possible solution: Use several rows to represent a single entity

- (111111, John, 123 Main St, stamps)
- (111111, John, 123 Main St, coins)

  Problems with this solution:

  - Redundancy of the other attributes (never good)
  - Key of entity type no longer can be key of relation

so, the resulting relation must be further transformed--> *Normalization* is this decomposition transformation process we will study to help deal with this and would result in:

Mapping #2-m (MULTIVALUED ATTRIBUTES): The multivalued attributes of a relation and the entity key become their own relation.

Persons(SSN, FirstName, LastName, Address, Birthdate)

Hobbies(SSN, Hobby)

Derived attributes

Coded separately in SQL as a view. They are not an attribute in a basic relation table.

Relationship mapping

Relationship: connects two or more entities into an association/relationship

- John majors in Computer Science

Relationship Type: set of similar relationships

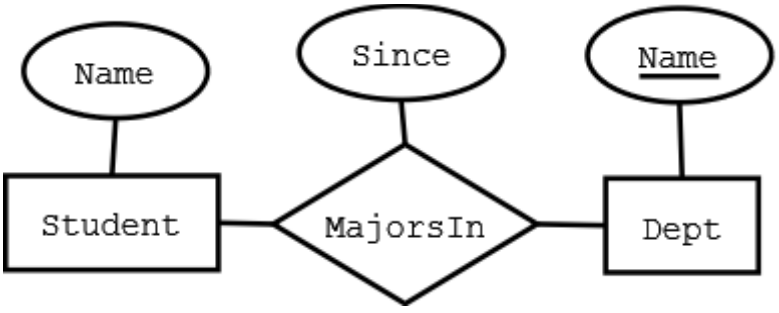
- Student (entity type) related to Department (entity type) by MajorsIn (relationship type).

Distinction -

- relation (relational model) - set of tuples
- relationship (E-R Model) – describes relationship between entities of an enterprise

Entity types and most relationship types in the E-R model are mapped to relations (relational model)

- Mapping #3 (1-MANY RELATIONSHIP): *1-1 and 1-many* relationships between separate entitites need not be mapped to a relation; the primary key attributes of the "1" relation become foreign key attributes of the "many" relation

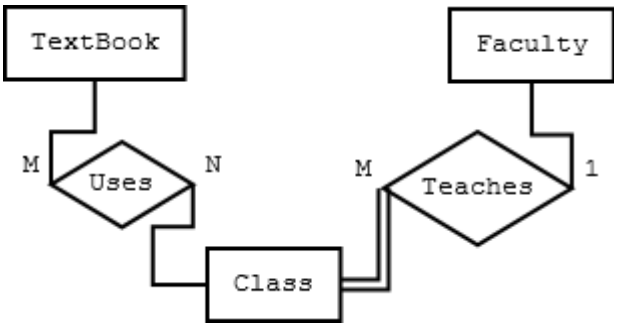


If no "Since" attribute, the relations could be (with some appropriate attribute renaming and additions)

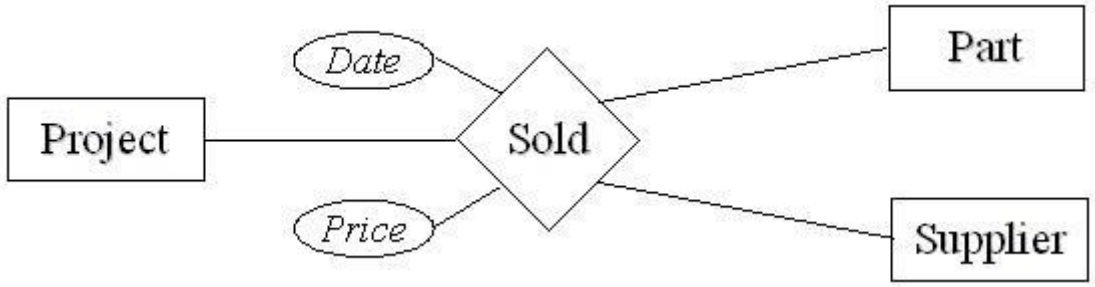
Students(StudId, Name, Dept)
Departments(Dept, Chair)

Relationship Types may also have attributes in the E-R model.

- Mapping #4 (RELATIONSHIP ATTRIBUTES): Any attributes of the *1-1 or 1-many* relationship may be attached to the "many" relation. Students(StudId, Name, Dept, Since) Departments(Dept, Chair)
- Mapping #5 (MANY-MANY RELATIONSHIP): *many-many* relationships are always mapped to a separate relation.



- Textbooks(ISBN, Title, Author, Copyright, Edition, Price) Class(ClassNo, Name, Room, Days, Time) TextUses(ISBN, ClassNo)Mapping #6: The attributes of *many-many* relationships become part of the relationship type relation, as well as the primary key attributes of the related entity types TextUses(ISBN, ClassNo, Optional)



Projects(ProjId, Name, TotalCost, StartDate)

Parts(UPC, PartName, Weight, WSPrice)

Suppliers(SupId, Name, Address)

Sold(ProjId, UPC, SupId, Date, Price)

Relationships tend to be verbs; attributes of relationships are nouns or adverbs

Q6:For the relations provided in company database in Q1, write relational algebra queries for the following:(2 MARKS EACH)

i. List the name of all employees with at least two dependents.

T1(SSN, NO OF DEPTS) ←− ESSN =COUNT DEPENDENT NAME(DEPENDENT) T2 ←− σ NO OF DEPTS≥2(T1) RESULT ←− π LNAME,F NAME(T2 ∗ EMPLOY EE)

ii. Find the employee number of employee who do not work on project Comp2.

π(EMPLOYEE_NAME(σ WORKSON.PROJECT!=COMP2(EMPLOYEE * WORKS_ON_PROJECT)

iii. Retrieve the name of manager who do not have female dependents.

ALL EMPS ←− π SSN (EMPLOY EE) EMPS WITH DEPS(SSN) ←− π ESSN (DEPENDENT) EMPS WITHOUT DEPS ←− (ALL EMPS − EMPS WITH DEPS) RESULT ←− π LNAME,F NAME(EMPS WITHOUT DEPS ∗ EMPLOYEE)

iv. List the names of all employees who have a dependent with the same name as themselves.

EMPS DEPS ← (EMPLOY EE /SSN=ESSN AND F NAME=DEPENDENT NAME DEPENDENT) RESULT ← π LNAME,F NAME(EMPS DEPS)

v. Retrieve details of employee working on both comp3 and comp4 project Numbers

π(EMPLOYEE_NAME(σ WORKS ON.PROJECT=COMP2 AND WORKS ON.PROJECT=COMP3(EMPLOYEE * WORKS_ON_PROJECT)

Q7(A)Explain how the different update operations deal with constraint violations

(4 MARKS)

Update Operations, Transactions, and Dealing with Constraint Violations

The operations of the relational model can be categorized into *retrievals* and *updates.* The relational algebra operations, which can be used to specify retrievals, are discussed in detail in Chapter 6. A relational algebra expression forms a new relation after applying a number of algebraic operators to an existing set of relations; its main use is for querying a database to retrieve information. The user formulates a query that specifies the data of interest, and a new relation is formed by applying relational operators to retrieve this data. That result relation becomes the answer to (or result of) the user's query.

In this section, we concentrate on the database modification or update operations. There are three basic operations that can change the states of relations in the data-base: Insert, Delete, and Update (or Modify). They insert new data, delete old data, or modify existing data records. Insert is used to insert one or more new tuples in a relation, Delete is used to delete tuples, and Update (or Modify) is used to change the values of some attributes in existing tuples. Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated. In this section we discuss

the types of constraints that may be violated by each of these operations and the types of actions that may be taken if an operation causes a violation. We use the database shown in Figure 3.6 for examples and discuss only key constraints, entity integrity constraints, and the referential integrity constraints shown in Figure 3.7. For each type of operation, we give some examples and discuss any constraints that each operation may violate.

1. The Insert Operation

The Insert operation provides a list of attribute values for a new tuple *t* that is to be inserted into a relation *R*. Insert can violate any of the four types of constraints dis-cussed in the previous section. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type. Key constraints can be violated if a key value in the new tuple *t* already exists in another tuple in the relation *r*(*R*). Entity integrity can be violated if any part of the primary key of the new tuple *t* is NULL. Referential integrity can be violated if the value of any foreign key in *t* refers to a tuple that does not exist in the referenced relation. Here are some examples to illustrate this discussion.

Operation:

Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, NULL, ‘1960-04-05’, ‘6357 Windy Lane, Katy, TX’, F, 28000, NULL, 4> into EMPLOYEE.

Result: This insertion violates the entity integrity constraint (NULL for the primary key Ssn), so it is rejected.

Operation:

Insert <‘Alicia’, ‘J’, ‘Zelaya’, ‘999887777’, ‘1960-04-05’, ‘6357 Windy Lane, Katy, TX’, F, 28000, ‘987654321’, 4> into EMPLOYEE.

Result: This insertion violates the key constraint because another tuple with the same Ssn value already exists in the EMPLOYEE relation, and so it is rejected.

Operation:

Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, ‘677678989’, ‘1960-04-05’, ‘6357 Windswept, Katy, TX’, F, 28000, ‘987654321’, 7> into EMPLOYEE.

Result: This insertion violates the referential integrity constraint specified on Dno in EMPLOYEE because no corresponding referenced tuple exists in

DEPARTMENT with Dnumber = 7.

Operation:

Insert <‘Cecilia’, ‘F’, ‘Kolonsky’, ‘677678989’, ‘1960-04-05’, ‘6357 Windy Lane, Katy, TX’, F, 28000, NULL, 4> into EMPLOYEE.

Result: This insertion satisfies all constraints, so it is acceptable.

If an insertion violates one or more constraints, the default option is to *reject the insertion.* In this case, it would be useful if the DBMS could provide a reason to the user as to why the insertion was rejected. Another option is to attempt to *correct the reason for rejecting the insertion,* but this is typically not used for violations caused by Insert; rather, it is used more often in correcting violations for Delete and Update. In the first operation, the DBMS could ask the user to provide a value for Ssn, and could then accept the insertion if a valid Ssn value is provided. In opera-tion 3, the DBMS could either ask the user to change the value of Dno to some valid value (or set it to NULL), or it could ask the user to insert a DEPARTMENT tuple with Dnumber = 7 and could accept the original insertion only after such an operation was accepted. Notice that in the latter case the insertion violation can cascade back to the EMPLOYEE relation if the user attempts to insert a tuple for department 7 with a value for Mgr_ssn that does not exist in the EMPLOYEE relation.

2. The Delete Operation

The Delete operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database. To specify deletion, a condition on the attributes of the relation selects the tuple (or tuples) to be deleted. Here are some examples.

Operation:

Delete the WORKS_ON tuple with Essn = ‘999887777’ and Pno = 10. *Result*: This deletion is acceptable and deletes exactly one tuple.

Operation:

Delete the EMPLOYEE tuple with Ssn = ‘999887777’.

Result: This deletion is not acceptable, because there are tuples in WORKS_ON that refer to this tuple. Hence, if the tuple in EMPLOYEE is

deleted, referential integrity violations will result.

Operation:

Delete the EMPLOYEE tuple with Ssn = ‘333445555’.

Result: This deletion will result in even worse referential integrity violations, because the tuple involved is referenced by tuples from the EMPLOYEE,

DEPARTMENT, WORKS_ON, and DEPENDENT relations.

Several options are available if a deletion operation causes a violation. The first option, called restrict, is to *reject the deletion.* The second option, called cascade, is to *attempt to cascade (or propagate) the deletion* by deleting tuples that reference the tuple that is being deleted. For example, in operation 2, the DBMS could automati-cally delete the offending tuples from WORKS_ON with Essn = '999887777'. A third option, called set null or set default, is to *modify the referencing attribute values* that cause the violation; each such value is either set to NULL or changed to reference another default valid tuple. Notice that if a referencing attribute that causes a viola-tion is *part of the primary key,* it *cannot* be set to NULL; otherwise, it would violate entity integrity.

Combinations of these three options are also possible. For example, to avoid having operation 3 cause a violation, the DBMS may automatically delete all tuples from WORKS_ON and DEPENDENT with Essn = '333445555'. Tuples in EMPLOYEE with Super_ssn = '333445555' and the tuple in DEPARTMENT with Mgr_ssn = '333445555' can have their Super_ssn and Mgr_ssn values changed to other valid values or to NULL. Although it may make sense to delete automatically the WORKS_ON and DEPENDENT tuples that refer to an EMPLOYEE tuple, it may not make sense to delete other EMPLOYEE tuples or a DEPARTMENT tuple.

In general, when a referential integrity constraint is specified in the DDL, the DBMS will allow the database designer to *specify which of the options* applies in case of a violation of the constraint. We discuss how to specify these options in the SQL DDL in Chapter 4.

3. The Update Operation

The Update (or Modify) operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation *R*. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified. Here are some examples.

Operation:

Update the salary of the EMPLOYEE tuple with Ssn = '999887777' to 28000. *Result*: Acceptable.

Operation:

Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 1. *Result*: Acceptable.

Operation:

Update the Dno of the EMPLOYEE tuple with Ssn = '999887777' to 7. *Result*: Unacceptable, because it violates referential integrity.

Operation:

Update the Ssn of the EMPLOYEE tuple with Ssn = '999887777' to '987654321'.

Result: Unacceptable, because it violates primary key constraint by repeating a value that already exists as a primary key in another tuple; it violates refer-ential integrity constraints because there are other relations that refer to the existing value of Ssn.

Updating an attribute that is *neither part of a primary key nor of a foreign key* usually causes no problems; the DBMS need only check to confirm that the new value is of the correct data type and domain. Modifying a primary key value is similar to delet-ing one tuple and inserting another in its place because we use the primary key to identify tuples. Hence, the issues discussed earlier in both Sections 3.3.1 (Insert) and 3.3.2 (Delete) come into play. If a foreign key attribute is modified, the DBMS must make sure that the new value refers to an existing tuple in the referenced relation (or is set to NULL). Similar options exist to deal with referential integrity violations caused by Update as those options discussed for the Delete operation. In fact, when a referential integrity constraint is specified in the DDL, the DBMS will allow the user to choose separate options to deal with a violation caused by Delete and a vio-lation caused by Update (see Section 4.2).

4. The Transaction Concept

A database application program running against a relational database typically executes one or more *transactions*. A transaction is an executing program that includes some database operations, such as reading from the database, or applying insertions, deletions, or updates to the database. At the end of the transaction, it must leave the database in a valid or consistent state that satisfies all the constraints spec-ified on the database schema. A single transaction may involve any number of retrieval operations (to be discussed as part of relational algebra and calculus in Chapter 6, and as a part of the language SQL in Chapters 4 and 5), and any number of update operations. These retrievals and updates will together form an atomic unit of work against the database. For example, a transaction to apply a bank with-drawal will typically read the user account record, check if there is a sufficient bal-ance, and then update the record by the withdrawal amount.

Q7(B)Write the SQL query for the following relation algebra expression. (2 MARKS EACH)

i. πBdate, Address(σFname='John' and Lname='Smith'(EMP))

SELECT BDATE,ADDRESS FROM EMP WHERE FNAME="JOHN" AND LNAME="SMITH";

ii. πSSN, Name((σEmp.Dno=Dept.Dno and Dname='Research'(EMP X DEPT)))

SELECT SSN,NAME FROM EMP, DEPT WHERE E.DNO=D.DNO AND DNAME="RESEARCH";