**Module II**

**Relational Model:** Structure of relational Databases, Integrity Constraints, synthesizing ER diagram to relational schema (Reading: ElmasriNavathe, Ch. 3 and 8.1, Additional Reading: Silbershatz, Korth, Ch. 2.1-2.4) **Database Languages:** Concept of DDL and DML relational algebra (Reading: Silbershatz, Korth, Ch 2.5-2.6 and 6.1-6.2, ElmasriNavathe, Ch. 6.1-6.5)

## Relational Model Concepts

**Q) Define a Relation**

The Relational Model represents the database as a collection of **relations**.

* The relational Model of Data is based on the concept of a **Relation.**
* A **Relation** is a mathematical concept based on the idea of **sets.**
* RELATION: A table of values
* A relation may be thought of as a **set of row**s.
* A relation may alternately be thought of as a **set of columns**.
* Each row represents a fact that corresponds to a real-world **entity** or **relationship**.
* Each **row** has a value of an item or set of items that uniquely identifies that row in the table.
* Sometimes **row-ids** or **sequential numbers** are assigned to identify the rows in the table.
* Each **column** typically is called by its column name or column header or **attribute** name.

**Q) Define Domain, Attribute, Tuple**

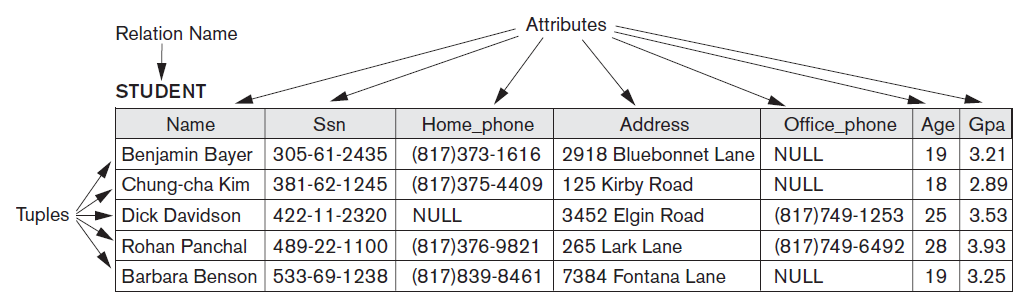
A **domain** *D* is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned. Examples:

* Social\_security\_numbers: The set of valid nine-digit Social Security numbers
* Names: The set of character strings that represent names of persons
* A **data type** or **format** is also specified for each domain
* A domain is thus given a name, data type, and format

**Relation Schema** – It is a description of a **relation** which is a collection of **attributes** A1, A2, A3, ..An. Each attribute Ai is the name of a role played by the domain D in the relation R (D is called the domain of Ai and denoted as dom(Ai)). R is the name of the relation. The **degree** of a relation is the number of attributes **n** of its relation schema.

Eg- Relation schema for a relation of degree 7 describes Engineering students

STUDENT(Name, Ssn, Home\_phone, Address, Office\_phone, Age, Gpa)



**Figure 3.1**

**Terminology**

|  |  |
| --- | --- |
| **Informal Terms** | **Formal Terms** |
| Table | Relation |
| Column | Attribute |
| Row | Tuple |
| Values in a Column | Domain |
| Table Definition | Schema of Relation |
| Populated Table | Extension |

A **relation** (or **relation state**) *r* of the relation schema *R*(*A*1, *A*2, ..., *An*), also denoted by *r*(*R*), is a set of *n*-tuples *r* = {*t*1, *t*2, ..., *tm*}. Each ***n*-tuple** *t* is an ordered list of *n* values *t =<v*1*, v*2*, ..., vn>*, where each value *vi,* 1 ≤*i* ≤ *n*, is an element of dom *(Ai*) or is a special NULL value. A relation state at a given time is called the **current relation state**

**Q) List and explain the Characteristics/Features of Relations**

**Q) Why are the tuples in a relation not ordered?**

1. **Ordering of Tuples in a Relation -** A relation is defined as a *set* of tuples. Mathematically, elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order. Thus, a relation is not sensitive to the ordering of tuples. However, in a file, records are physically stored on disk so there always is an order among the records
2. **Ordering of Values within a Tuple and an Alternative Definition of a Relation -** an *n*-tuple is an *ordered list* of *n* values, so the ordering of values in a tuple and hence of attributes in a relation schema is important. However, at a more abstract level, the order of attributes and their values is *not* that important as long as the correspondence between attributes and values is maintained.

An **alternative definition** of a relation - a **tuple** can be considered as a **set** of (<attribute>, <value>) pairs, where each pair gives the value of the mappingfrom an attribute *Ai* to a value *vi* from dom(*Ai*). The ordering of attributes is *not* important, because the *attribute name* appears with its *value*.

1. **Values and NULLs in the Tuples.** Each value in a tuple is an **atomic** value; that is, it is not divisible into components. Hence, composite and multivalued attributes are not allowed. This model is sometimes called the **flat relational model.**

NULL values, are used to represent the valuesof attributes that may be unknown or may not apply to a tuple. A special value,called NULL, is used in these cases.

There are several meanings for NULL values, such as ***value unknown****,* ***value*** exists but is ***not available***, or ***attribute*** ***does not apply*** to this tuple.

1. **Interpretation (Meaning) of a Relation.** The relation schema can be interpreted as a declaration or a type of **assertion**. Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion. Example in Figure 3.1 asserts the fact that there is a STUDENT whose Name is Benjamin Bayer, Ssn is 305-61-2435, Age is 19, and so on Some relations may represent facts about *entities,* whereas other relations may represent facts about *relationships.*

**Q) Explain various relational model constraints**

**Q) Which constraint deals with two relation (referential integrity constraint)**

**Q) Explain entity integrity and referential integrity**

**Q) Explain concept of foreign key**

**Relational Model Constraints**

There are generally many restrictions or **constraints** on the actual values in a database state. These constraints are derived from the rules in the miniworld that the database represents.

Constraints on databases can generally be divided into three main categories:

1. Constraints that are inherent in the data model. We call these **inherent model-based constraints** or **implicit constraints**.- ( explanation points same as characteristics of relations listed above ), example, the constraint that a relation cannot have duplicate tuples is an inherent constraint
2. Constraints that can be directly expressed in schemas of the data model, called as **schema-based constraints** or **explicit constraints**

* Domain Constraints
* Key constraints and constraints on NULL values
* Entity Integrity, Referential Integrity and Foreign key Constraint

1. Constraints that *cannot* be directly expressed in the schemas of the data model, and hence must be expressed and enforced by the application programs. We call these **application-based** or **semantic constraints** or **business rules**.

**2.1 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*).

**2.2 Key Constraints and Constraints on NULL Values** (mention about super key, uniqueness constraint, key, primary and candidate key)

No two tuples can have the same combination of values for *all* their attributes. Usually, there are **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]≠*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.

A superkey can have redundant attribute values, 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.

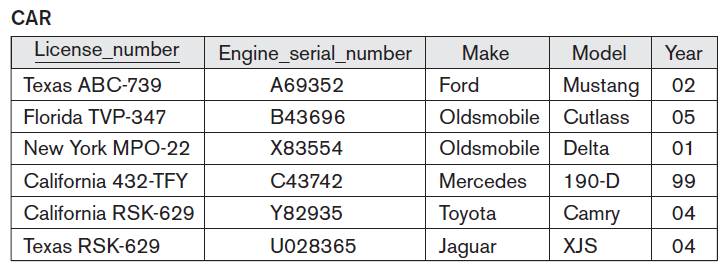
Hence, a key satisfies two properties:

**1.** 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.

**2.** 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.

For example, {Ssn, Name, Age}—is a superkey. The attribute set {Ssn} obtained by removing Name and Age is a key of STUDENT because no two student tuples can have the same value for Ssn, neither the same attribute value.

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. Attributes that form the primary key of a relation schema are underlined, as shown in Figure 3.2. The other candidate keys are designated as **unique keys**, and are not underlined.



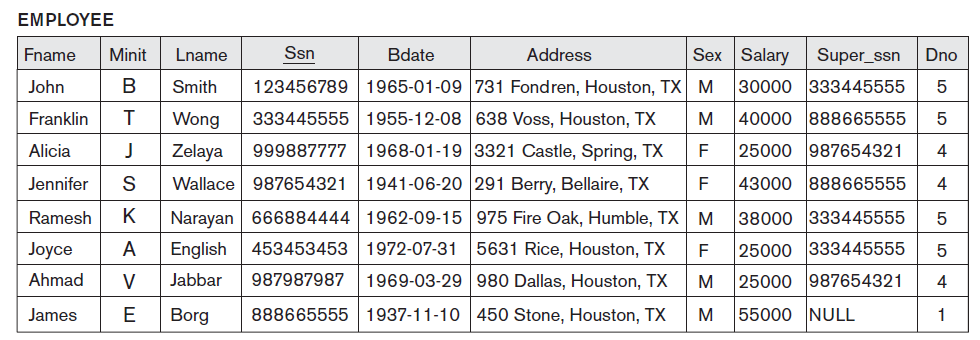
**Figure 3.2**

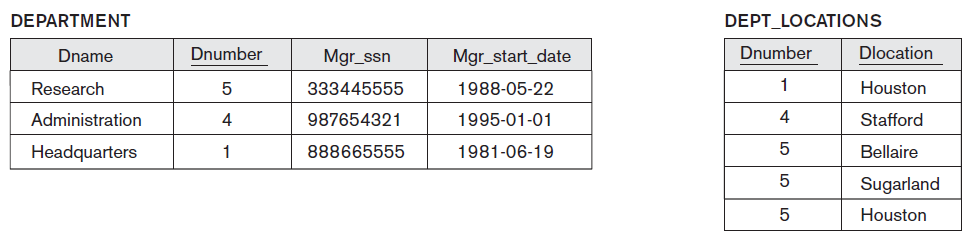
**Another constraint on attributes specifies whether NULL values are or are not permitted**. For example, if every STUDENT tuple must have a valid, non-NULL value for the Name attribute, then Name of STUDENT is constrained to be NOT NULL

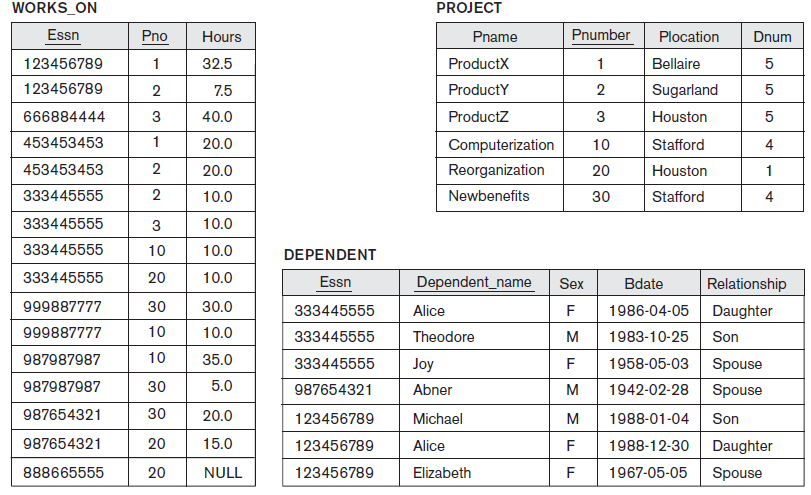
A **relational database schema** *S* is a set of relation schemas S = {*R*1, *R*2, ..., *Rm*} and a set of **integrity constraints** IC. A database state that does not obey all the integrity constraints is called an **invalid state**, and a state that satisfies all the constraints in the defined set of integrity constraints IC is called a **valid state.**

**2.3 Entity 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.







**Figure 3.3 (for explaining referential integrity and foreign key concept)**

**Referential Integrity Constraint** (can be used to explain foreign key)

The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations.

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 3.3 (above), 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 constraintbetween the two relation schemas *R*1 and *R*2. A set of attributes FK in relationschema *R*1 is a **foreign key** of *R*1 that **references** relation *R*2 if it satisfies thefollowing rules:

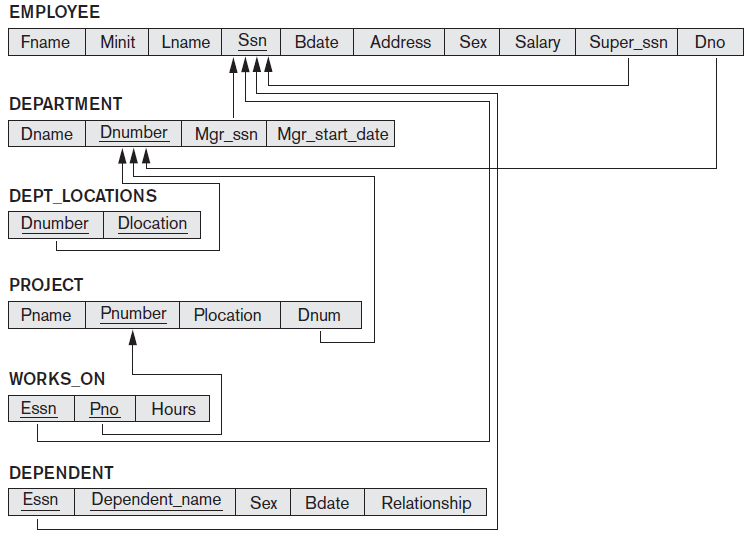
**1.** The attributes in FK have the same domain(s) as the primary key attributes PK of *R*2; the attributes FK are said to **reference** or **refer to** the relation *R*2.

**2.** A value of FK in a tuple *t*1 of the current state *r*1(*R*1) either occurs as a value of PK for some tuple *t*2 in the current state *r*2(*R*2) *or is NULL*. In the former case, we have *t*1[FK] = *t*2[PK], and we say that the tuple *t*1 **references** or **refers to** the tuple *t*2.

In this definition, *R*1 is called the **referencing relation** and *R*2 is the **referenced relation**. If these two conditions hold, a **referential integrity constraint** from *R*1 to *R*2 is said to hold

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

We can *diagrammatically display referential integrity constraints* by drawing a directed arc from each foreign key to the relation it references



**Figure 3.4**

**Q)Expalin Self refrencing Foriegn key with example**

**Self Referencing Foreign Key** : Foreign key referencing a primary key in the same table. Ex: Super\_ssn(foreign key) of table EMPLOYEE referencing Ssn(Primary Key) of same table EMPLOYEE(draw the above fig showing Super\_ssn refrencing Ssn)

**3. application-based** or **semantic constraints** or **business rules**.

***3.1 semantic integrity constraints ( application based constraint)***

They are specified and enforced on a relational database. Examples of such constraints are *the salary of an employee should not exceed the salary of the* *employee’s supervisor* and *the maximum number of hours an employee can work on all* *projects per week is 56*. Such constraints can be specified and enforced within the application programs that update the database.

**Other constraints…**

***functional dependency* constraint**, which establishes a functional relationship among two sets of attributes *X* and *Y*. This constraint specifies that the value of *X* determines a unique value of *Y* in all states of a relation; it is denoted as a functional dependency *X* -> *Y*

**state constraints** (all constraints discussed so far)- define the constraints that a *valid state* of the database must satisfy. Another type of constraint, called **transition constraints**, can be defined to deal with state changes in the database

**Q) Explain different Update operation, how do Insertion, deletion violates relational model constraints also explain how are the constraint violations handled are.**

**UPDATE operations and dealing with Constraint Violations**

The operations of the relational model can be categorized into *retrievals* and *Updates*. There are three basic operations that can change the states of relations in the database: Insert, Delete, and Update (or Modify).

1. **Insert** is used to insert one or more new tuples in a relation,
2. **Delete** is used to delete tuples, and
3. **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.

**Constraint violation made by 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.*

* 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.

**How the constraint violation made by Insert operation is handled**..

* If an insertion violates one or more constraints, the default option is to *reject the insertion*

**Constraint violation made by 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

**How the constraint violation made by Delete operation is handled**..

* 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.
* 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 violation is *part of the primary key,* it *cannot* be set to NULL; otherwise, it would violate entity integrity.

**Dealing with Constraint violation made by 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*.

* Updating an attribute that is *neither part of a primary key nor of a foreign key* usually causes no problems;
* Modifying a primary key value is similar to deleting one tuple and inserting another in its place because we use the primary key to identify tuples
* 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)

**Converting (Mapping) E-R model (for Company) to Relational Schema and writing SQL Data Definition for creating the schema.**

## 

We assume that the mapping will create tables with simple single-valued attributes. The relational model constraints like primary keys, unique keys (if any), and referential integrity constraints on the relations, will also be specified in the mapping results.

**Step 1: Mapping of Regular Entity Types.** For each regular (strong) entity type *E* in the ER schema, create a relation *R* that includes all the simple attributes of *E*. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of *E* as the primary key for *R*. If the chosen key of *E* is a composite, then the set of simple attributes that form it will together form the primary key of *R*.

If multiple keys were identified for *E* during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify secondary (unique) keys of relation *R*.

**Step 2: Mapping of Weak Entity Types.** For each weak entity type *W* in the ER schema with owner entity type *E*, create a relation *R* and include all simple attributes (or simple components of composite attributes) of *W* as attributes of *R*. In addition, include as foreign key attributes of *R*, the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of mapping the identifying relationship type of *W*. The primary key of *R* is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type *W*, if any.

**Step 3: Mapping of Binary 1:1 Relationship Types.** For each binary 1:1 relationship type *R* in the ER schema, identify the relations *S* and *T* that correspond to the entity types participating in *R.*

**Foreign key approach:** Choose one of the relations—*S*, say—and include as a foreign key in *S* the primary key of *T*. It is better to choose an entity type with *total participation* in *R* in the role of *S*. Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type *R* as attributes of *S*.

**Step 4: Mapping of Binary 1:N Relationship Types.** For each regular binary 1:N relationship type *R*, identify the relation *S* that represents the participating entity type at the *N-side* of the relationship type. Include as foreign key in *S* the primary key of the relation *T* that represents the other entity type participating in *R*; we do this because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type. Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of *S*.

**Step 5: Mapping of Binary M:N Relationship Types.** For each binary M:N relationship type *R*, create a new relation *S* to represent *R*. Include as foreign key attributes in *S* the primary keys of the relations that represent the participating entity types; their *combination* will form the primary key of *S*. Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of *S*. Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations (as we did for 1:1 or 1:N relationship types) because of the M:N cardinality ratio; we must create a separate *relationship relation S*.

**Step 6: Mapping of Multivalued Attributes.** For each multivalued attribute *A*, create a new relation *R*. This relation *R* will include an attribute corresponding to *A*, plus the primary key attribute *K—*as a foreign key in *R—*of the relation that represents the entity type or relationship type that has *A* as a multivalued attribute. The primary key of *R* is the combination of *A* and *K*. If the multivalued attribute is composite, we include its simple components.

Following the steps we get the relational database schema as shown below (next page).

## 

**The database Definitions to create tables corresponding to the database schema generated.**

**CREATE TABLE** EMPLOYEE

( Fname VARCHAR(15) **NOT NULL**,

Minit CHAR,

Lname VARCHAR(15) **NOT NULL**,

Ssn CHAR(9) **NOT NULL**,

Bdate DATE,

Address VARCHAR(30),

Sex CHAR,

Salary DECIMAL(10,2),

Super\_ssn CHAR(9),

Dno INT **NOT NULL**,

**PRIMARY KEY** (Ssn),

**FOREIGN KEY** (Super\_ssn) **REFERENCES** EMPLOYEE(Ssn),

**FOREIGN KEY** (Dno) **REFERENCES** DEPARTMENT(Dnumber) );

**CREATE TABLE** DEPARTMENT

( Dname VARCHAR(15) **NOT NULL**,

Dnumber INT **NOT NULL**,

Mgr\_ssn CHAR(9) **NOT NULL**,

Mgr\_start\_date DATE,

**PRIMARY KEY** (Dnumber),

**UNIQUE** (Dname),

**FOREIGN KEY** (Mgr\_ssn) **REFERENCES** EMPLOYEE(Ssn) );

**CREATE TABLE** DEPT\_LOCATIONS

( Dnumber INT **NOT NULL**,

Dlocation VARCHAR(15) **NOT NULL**,

**PRIMARY KEY** (Dnumber, Dlocation),

**FOREIGN KEY** (Dnumber) **REFERENCES** DEPARTMENT(Dnumber) );

**CREATE TABLE** PROJECT

( Pname VARCHAR(15) **NOT NULL**,

Pnumber INT **NOT NULL**,

Plocation VARCHAR(15),

Dnum INT **NOT NULL**,

**PRIMARY KEY** (Pnumber),

**UNIQUE** (Pname),

**FOREIGN KEY** (Dnum) **REFERENCES** DEPARTMENT(Dnumber) );

**CREATE TABLE** WORKS\_ON

( Essn CHAR(9) **NOT NULL**,

Pno INT **NOT NULL**,

Hours DECIMAL(3,1) **NOT NULL**,

**PRIMARY KEY** (Essn, Pno),

**FOREIGN KEY** (Essn) **REFERENCES** EMPLOYEE(Ssn),

**FOREIGN KEY** (Pno) **REFERENCES** PROJECT(Pnumber) );

**CREATE TABLE** DEPENDENT

( Essn CHAR(9) **NOT NULL**,

Dependent\_name VARCHAR(15) **NOT NULL**,

Sex CHAR,

Bdate DATE,

Relationship VARCHAR(8),

**PRIMARY KEY** (Essn, Dependent\_name),

**FOREIGN KEY** (Essn) **REFERENCES** EMPLOYEE(Ssn) );

## Database Design Guidelines

**Relational Database Design** –The grouping of attributes to form “good” relation schemas.

There are 4 measures of quality for a relation schema design

1. **Semantics of the attributes**.- Design a schema that can be explained easily by the relation. The semantics of attributes should be easy to interpret. Attributes of different entities (EMPLOYEEs, DEPARTMENTs, PROJECTs) should not be mixed in the same relation. Only foreign keys should be used to refer to other entities. Entity and relationship attributes should be kept apart as much as possible.
2. **Reducing the redundant values in tuples and Update Anomalies** - Mixing attributes of multiple entities may cause problems. Information is stored redundantly wasting storage. Problems with update anomalies- Insertion anomalies, Deletion anomalies, and Modification anomalies. Example- Consider the relation: EMP\_PROJ ( **Emp#, Proj#**, Ename, Pname, No\_hours) 🡪 **Update Anomaly:** Changing the name of project number P1 from “Billing” to “Customer Accounting” may cause this update to be made for all 100 employees working on project P1. **Insert Anomaly:** Cannot insert a project unless an employee is assigned to. ***Inversely*** - Cannot insert an employee unless he/she is assigned to a project. **Delete Anomaly:** When a project is deleted, it will result in deleting all the employees who work on that project. Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.
3. **Reducing null values in tuples** - Attributes that are NULL frequently could be placed in separate relations (with the primary key) - Reasons for nulls: attribute not applicable or invalid, attribute value unknown (may exist), value known to exist, but unavailable.
4. **Disallowing generation of spurious tuples** - Bad designs for a relational database may result in erroneous results for certain JOIN operations

## Q) Explain the concept of Relational Algebra, differentiate between relational algebra and relational calculus

**Q) List and explain the fundamental operations of relational algebra with example**

Relation Algebra constitutes a basic set of operations that can be carried out on a relational model.

The relational algebra is very important for several reasons.

* it provides a formal foundation for relational model operations
* it is used as a basis for implementing and optimizing queries.
* some of its concepts are incorporated into the SQL standard query language for RDBMSs.

Whereas the algebra defines a set of operations for the relational model, the **relational calculus** provides a higher-level *declarative* language for specifying relational queries. A relational calculus expression creates a new relation. In a relational calculus expression, there is *no order of operations* to specify how to retrieve the query result—only what information the result should contain. This is the main distinguishing feature between relational algebra and relational calculus

**Operations in Relational Algebra**

1. Unary Relational Operations:

**SELECT** Operation

**PROJECT** Operation

2. Relational Algebra Operations from Set Theory

**UNION, INTERSECTION,** and **MINUS** Operations

**CARTESIAN PRODUCT** (or **CROSS PRODUCT**) Operation.

3. Binary Relational Operations

**JOIN** Operation

**EQUIJOIN** and **NATURAL JOIN (**Variations of JOIN)

**DIVISION** Operation.

4. Additional Relational Operations

**Aggregate Functions** and **Grouping**

**Recursive Closure** Operations

**OUTER JOIN** Operations

##### **SELECT Operation**

This unary operator is used to select a subset of the tuples from a relation that satisfies a **selection condition** (a sort of a filter) and is given by

σ <selection condition> (R)

Example: σ <salary>4000> (EMPLOYEE)

(The operation would select those tuples from the EMPLOYEE relation/table whose salary is greater than 4000)

The Boolean expression specified in <selection condition> is made up of a number

of **clauses** of the form

<attribute name> <comparison op> <constant value>

or

<attribute name> <comparison op> <attribute name>

Comparison operators possible are( =, <, ≤, >, ≥, ≠))

The relation resulting from the SELECT operation has the same attributes as R. The selection condition is applied independently to each tuple in a relation.

Clauses can be connected by Boolean operators AND, OR and NOT.

Eg- σ <salary>4000 AND deptno=2> (EMPLOYEE)

The SELECT operation can also be visualized as a *horizontal partition* of the relation into two sets of tuples.

In SQL, the SELECT condition is typically specified in the WHERE clause of a query.

For example, the following operation:

σ Dno=4 **AND** Salary>25000 (EMPLOYEE),

would correspond to the following SQL query:

**SELECT** \*

**FROM** EMPLOYEE

**WHERE** Dno=4 **AND** Salary>25000

##### **PROJECT Operation**

The PROJECT operation selects certain attributes/columns from the relation/table. The PROJECT operation is given by the general form

**Π** <attribute list> (R)

Example: **Π** <Name, Salary> (EMPLOYEE). The operation would yield all tuples in the EMPLOYEE Relation for the columns of Name and Salary.

The number of tuples resulting from the PROJECT operation is equal to the number of tuples in R.

If the attribute list includes only nonkey attributes of *R*, duplicate tuples are likely to occur. The PROJECT operation *removes any duplicate tuples*, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**

In SQL, the PROJECT attribute list is specified in the SELECT clause of a query. For example, the following operation:

**Π**Sex, Salary(EMPLOYEE)

would correspond to the following SQL query:

**SELECT DISTINCT** Sex, Salary

**FROM** EMPLOYEE

Notice that if we remove the keyword **DISTINCT** from this SQL query, then duplicates will not be eliminated

**Sequences of Operations and the RENAME Operation**

When we need to apply several relational algebra operations one after the other. Either we can write the operations as a single **relational algebra expression** by nesting the operations, or we can apply one operation at a time and create intermediate result relations

For example, to retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a SELECT and a PROJECT operation. We can write a single relational algebra expression, also known as an **in-line expression**, as follows:

**Π**Fname, Lname, Salary(σDno=5(EMPLOYEE))

Alternatively, we can explicitly show the sequence of operations, giving a name to

each intermediate relation, as follows:

DEP5\_EMPS <- σDno=5(EMPLOYEE)

RESULT <- **Π** Fname, Lname, Salary(DEP5\_EMPS)

We can also use this technique to **rename** the attributes in the intermediate and result relations.

TEMP <- σ Dno=5(EMPLOYEE)

*E(*First\_name, Last\_name, Salary) <- **Π** Fname, Lname, Salary(TEMP)

The attributes Fname, Lname , Salary in TEMP would be renamed as First\_name, Last\_name, Salary in E.

The **rename operator** is also represented as **ρ** and can be applied as shown.

**ρ** *E*(First\_name, Last\_name, Salary)(TEMP)

Renaming in SQL is accomplished by aliasing using **AS**, as in the following example:

**SELECT** E.Fname **AS** First\_name, E.Lname **AS** Last\_name, E.Salary **AS** Salary

**FROM** EMPLOYEE **AS** E

**WHERE** E.Dno=5

##### **UNION, INTERSECTION, MINUS Operations**

These are **binary operations** from set theory adapted to relational databases. Let R & S be two Relations, R(A1, A2, A3 ..An) and S(B1, B2, B3 ..Bn). When these operations are adapted to relational databases, the two relations on which any of these three operations are applied must have the same **type of tuples**; this condition has been called *union compatibility* or *type compatibility. Ie:-* the two relations have the same number of attributes and each corresponding pair of attributes has the same domain

**UNION**- **R ∪ S,** the result of this operation includes all tuples that are either in R or S. Duplicate tuples are eliminated.

**INTERSECTION** – **R ∩ S,** the result of this operation is a relation that includes all tuples that are both in R and S.

**SET DIFFERENCE (minus)**- **R-S**, the result of the operation is a relation that includes all tuples that are in R but not in S.

**The CARTESIAN PRODUCT (CROSS PRODUCT) Operation**

It is denoted by X. This is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible. The set operation produces a new element by combining every member (tuple) from one relation (set) with every member (tuple) from the other relation (set).

In general, the result of *R*(*A*1, *A*2, ..., *An*) × *S*(*B*1, *B*2, ..., *Bm*) is a relation *Q* with degree *n* + *m* attributes *Q*(*A*1, *A*2, ..., *An*, *B*1, *B2*, ..., *Bm*), in that order.

The resulting relation *Q* has one tuple for each combination of tuples—one from *R* and one from *S*. Hence, if *R* has *nR* tuples and *S* has *nS* tuples,then *R* × *S* will have *nR* \* *nS* tuples.

For example, suppose that we want to retrieve a list of names of each female employee’s dependents. We can do this as follows ( look table(names) in figure 3.3 and 3.5 to understand)

FEMALE\_EMPS <- σ Sex=‘F’(EMPLOYEE)

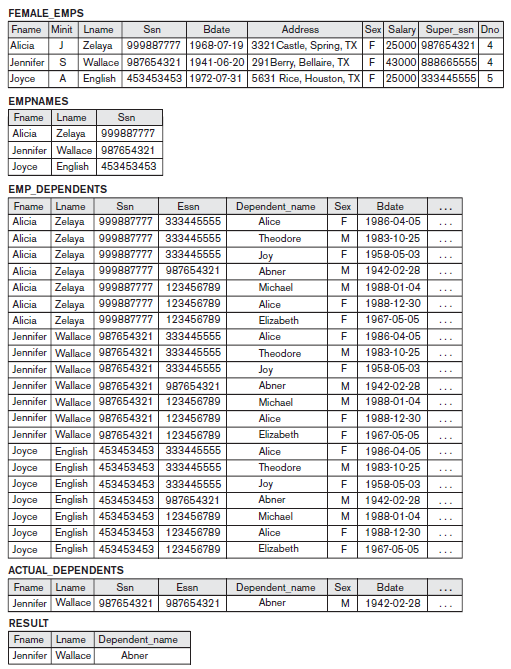
EMPNAMES <-**Π** Fname, Lname, Ssn(FEMALE\_EMPS)

EMP\_DEPENDENTS <- EMPNAMES X DEPENDENT

ACTUAL\_DEPENDENTS <- σ Ssn=Essn(EMP\_DEPENDENTS)

RESULT <-**Π** Fname, Lname, Dependent\_name(ACTUAL\_DEPENDENTS)

The results of the above operation are shown in the figure below(next page).

**

***Figure 3.5, example for Cartesian Product***

In SQL, CARTESIAN PRODUCT can be realized by using the CROSS JOIN option in joined tables. Alternatively, if there are two tables in the WHERE clause and there is no corresponding join condition in the query, the result will also be the CARTESIAN PRODUCT of the two tables

**Q) Explain different types of Join operations with example** ( 4 different types are Theta join, Equi Join, Natural Join and Outer join, among these Theta, Equi and Natural are classified under Inner Join)

**Q) Distinguish between Inner and Outer Join Operation**

**Q) Explain different types of Outer Join Operations.**

**Q) Explain Different types of Inner Join Operations.**

**Q) Distinguish between Cartesian Product and Join operations with example.**

**JOIN OPERATOR**

The JOIN operation is a binary operator. It is used to combine **related tuples** from two relations into a single tuple. The general form of a JOIN operation on two relations R1 and R2 with a resulting Relation R3 is

**R3**🡨**R1 ⋈<join condition> R2**

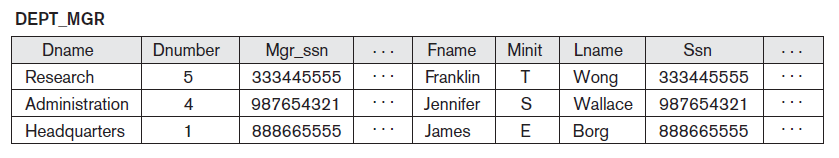
Example:-

Suppose that we want to retrieve the name of the manager of each department. To get the manager’s name, we need to combine each department tuple with the employee tuple whose Ssn value matches the Mgr\_ssn value in the department tuple.(look figure 3.3)

Dept\_MGR 🡨 DEPARTMENT ⋈<Mgr\_ssn=Ssn> EMPLOYEE

Result 🡨 **Π**<Dname, Lname, Fname > (DEPT\_MGR)

We first JOIN the relations EMPLOYEE and DEPARTMENT to get the tuples where the join condition is Mgr\_Ssn in DEPARTMENT matches Ssn in EMPLOYEE (look figure 3.3 and 3.6 to understand).



**Figure 3.6, Result of Dept\_MGR 🡨 DEPARTMENT ⋈<Mgr\_ssn=Ssn> EMPLOYEE**

Next we project the desired attributes namely Dname, Lname, Fname for each Department Manager.

The JOIN operation can be specified as a CARTESIAN PRODUCT operation followed by a SELECT operation. The result of the JOIN is a relation *Q* with *n* + *m* attributes *Q*(*A*1, *A*2, ..., *An*, *B*1, *B*2,... , *Bm*) in that order; *Q* has one tuple for each combination of tuples—one from *R* and one from *S—whenever the combination satisfies the join condition*. This is the main **difference between CARTESIAN PRODUCT and JOIN**. In JOIN, only combinations of tuples *satisfying the join condition* appear in the result, whereas in the CARTESIAN PRODUCT *all* combinations of tuples are included in the result.

A general join condition is of the form

<condition> **AND** <condition> **AND**...**AND** <condition>

where each <condition> is of the form *Ai* Θ*Bj*, where *Ai* is an attribute of *R*, *Bj* is an attribute of *S*, *Ai* and *Bj* have the same domain, and Θ (theta) is one of the comparison operators {=, <, ≤, >, ≥, ≠}. A JOIN operation with such a general join condition is called a **THETA JOIN**. Tuples whose join attributes are NULL or for which the join condition is FALSE *do not* appear in the result.

**EQUIJOIN** – It is a JOIN operation where only the comparison operator (**=**) is used. Example above shows an EQUIJOIN. (The result of an Equijoin will have always one or more pairs of attributes that have identical values in every tuple because the value of Mgr\_Ssn and Ssn are identical in every tuple of DEPT\_MGR. This is overcome using NATURAL JOIN.

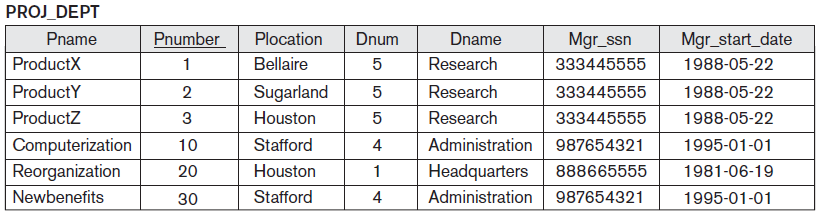
**NATURAL JOIN** – This operator was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition. In general, NATURAL JOIN is performed by equating all attribute pairs that have the same name in the two Relations. If this is not the case, a renaming operation is applied first.

Suppose we want to combine each PROJECT tuple with the DEPARTMENT tuple that controls the project. In the following example, first we rename the Dnumber attribute of DEPARTMENT to Dnum—so that it has the same name as the Dnum attribute in PROJECT—and then we apply NATURAL JOIN:

DEPT <- **ρ** (Dname, Dnum, Mgr\_ssn, Mgr\_start\_date)(DEPARTMENT)

PROJ\_DEPT <- PROJECT \* DEPT

The attribute Dnum is called the **join attribute** for the NATURAL JOIN operation, because it is the only attribute with the same name in both relations. The resulting relation is illustrated in Figure 3.7. (look figure 3.3 and 3.7 together to understand)



**Figure 3.7, Result of Natural Join**

A single JOIN operation is used to combine data from two relations so that related information can be presented in a single table. These operations are also known as **inner joins** (Theta, Equi and Natural Joins are Inner joins)**.** An *inner join* is a type of match and combine operation defined formally as a combination of CARTESIAN PRODUCT and SELECTION. In inner joins tuples with no match are eliminated. This amounts to the loss of information if the user wants the result of the JOIN to include all the tuples in one or more of the component relations.

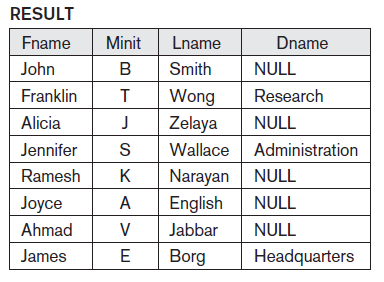
**Outer joins**, were developed for the case where the user wants to keep all the tuples in *R*, or all those in *S*, or all those in both relations in the result of the JOIN, regardless of whether or not they have matching tuples in the other relation. This satisfies the need of queries in which tuples from two tables are to be combined by matching corresponding rows, but without losing any tuples for lack of matching values.

For example, suppose that we want a list of all employee names as well as the name of the departments they manage *if they happen to manage* *a department*; if they do not manage one, we can indicate it with a NULL value. We can apply an operation **LEFT OUTER JOIN**, denoted by , to retrieve the result as follows:

TEMP <- (EMPLOYEE Ssn=Mgr\_ssn DEPARTMENT)

RESULT ← πFname, Minit, Lname, Dname(TEMP)

The LEFT OUTER JOIN operation keeps every tuple in the *first*, or *left*, relation *R* in *R S*; if no matching tuple is found in *S*, then the attributes of *S* in the join result arefilled or *padded* with NULL values. The result of these operations is shown in Figure 3.8



**Figure 3.8, example of Left Outer Join**

A similar operation, **RIGHT OUTER JOIN**, denoted by , keeps every tuple in the *second*, or right, relation *S* in the result of*R S*.

A third operation, **FULL OUTER** **JOIN**, denoted by , keeps all tuples in both the left and the right relations when no matching tuples are found, padding them with NULL values as needed.

**OUTER UNION** operation was developed to take the union of tuples from 2 relations if the relations are not Union Compactable.

**DIVISION OPERATOR**

The DIVISION operation, denoted by ÷, is useful for queries like, example - *Retrieve the names of* *employees who work on* ***all*** *the projects that ‘John Smith’ works on*. To express this query using the DIVISION operation, proceed as follows. First, retrieve the list of project numbers that ‘John Smith’ works on in the intermediate relation SMITH\_PNOS:

SMITH <- σFname=‘John’ **AND** Lname=‘Smith’(EMPLOYEE)

SMITH\_PNOS <- **Π** Pno(WORKS\_ON **⋈** Essn=Ssn SMITH)

Next, create a relation that includes a tuple <Pno, Essn> whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN\_PNOS:

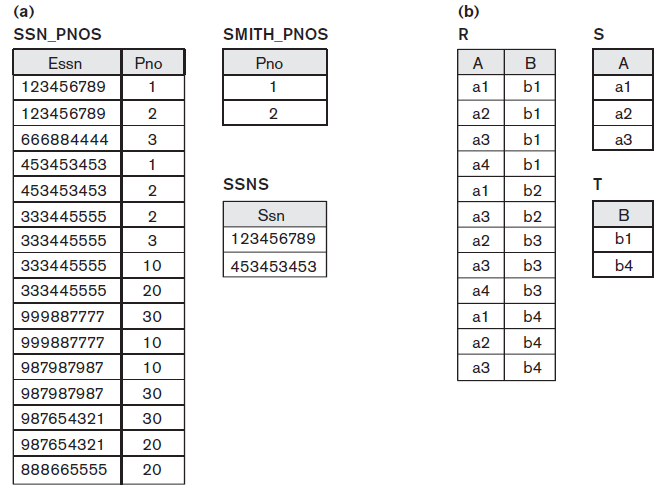
SSN\_PNOS <- **Π**Essn, Pno(WORKS\_ON)

Finally, apply the DIVISION operation to the two relations, which gives the desired employees’ Social Security numbers:

SSNS(Ssn) <- SSN\_PNOS ÷ SMITH\_PNOS

RESULT <- **Π**Fname, Lname(SSNS \* EMPLOYEE)

The above operations are shown in Figure 3.9(a).



**Figure 3.9, Division operation**

Figure 3.9(b) shows the result of T<- R÷S

##### **Q) Explain the Aggregate Functions and Grouping in relational algebera**

Aggregating functions (SUM, AVERAGE, MAXIMUM, MINIMUM, COUNT..) are applied on collection of values. It is denoted with the symbol **ℑ** (pronounced as “script F”) and has the general form

<grouping attributes> **ℑ** <function list> (R)

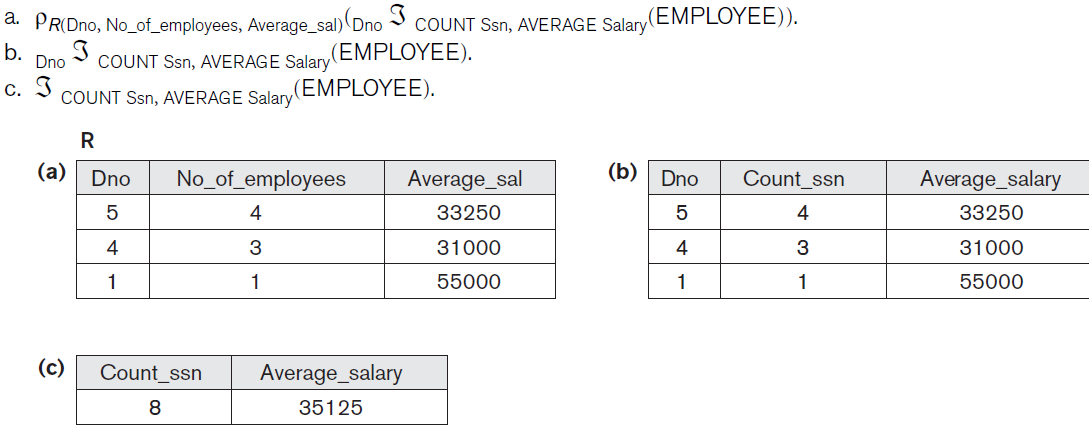
where <grouping attributes> is a list of attributes of the relation specified in *R*, and <function list> is a list of (<function> <attribute>) pairs. In each such pair, <function> is one of the allowed functions—such as SUM, AVERAGE, MAXIMUM, MINIMUM,COUNT—and <attribute> is an attribute of the relation specified by *R. The* resulting relation has the grouping attributes plus one attribute for each element in the function list.

For example, to retrieve each department number, the number of employees in the department, and their average salary, while renaming the resulting attributes – example figure 3.10(a)

In the above example(ie:3.10(a)), we specified a list of attribute names—between parentheses in the RENAME operation—for the resulting relation *R*. If no renaming is applied, then the attributes of the resulting relation that correspond to the function list will each be the concatenation of the function name with the attribute name in the form <function>\_<attribute> - example figure 3.10(b)

If no grouping attributes are specified, the functions are applied to *all the tuples* in

the relation, so the resulting relation has a *single tuple only,*  example figure 3.10(c)



**Figure 3.10, Example of Aggregate Functions**

**Recursive Closure Operations**

This is applied to a recursive relationship between tupes of the same type. Example:- To retrieve all Supervisees of an employee at all levels. (Employees e’ directly supervised by e, all e” directly supervised by e’ and so on).

For example, to specify the Ssns of all employees *e*’ directly supervised by the employee *e*

whose name is ‘James Borg’ (see Figure 3.3), we can apply the following operation

BORG\_SSN <- **Π**Ssn(σ Fname=‘James’ **AND** Lname=‘Borg’(EMPLOYEE))

SUPERVISION(Ssn1, Ssn2) <- **Π**Ssn,Super\_ssn(EMPLOYEE)

RESULT1(Ssn) <- **Π**Ssn1(SUPERVISION **⋈**Ssn2=Ssn BORG\_SSN)

To retrieve all employees supervised by Borg at level 2—that is, all employees *e*’’ supervised by some employee *e*’ who is directly supervised by Borg—we can apply

another **JOIN** to the result of the first query, as follows:

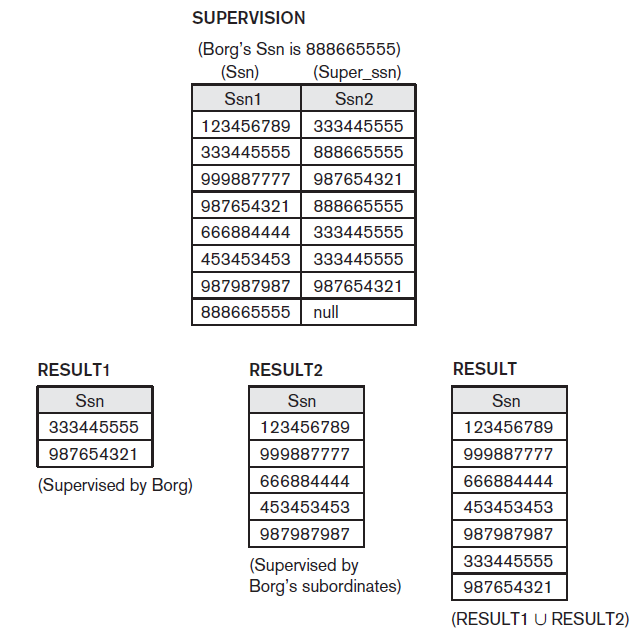
RESULT2(Ssn) <- **Π**Ssn1(SUPERVISION **⋈**Ssn2=Ssn RESULT1)

To get both sets of employees supervised at levels 1 and 2 by ‘James Borg’, we can

apply the UNION operation to the two results, as follows:

RESULT <-RESULT2 ∪ RESULT1

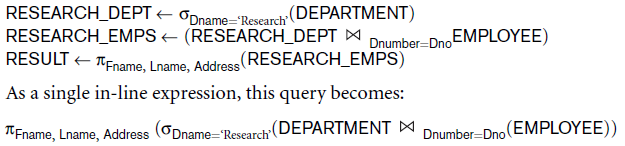
The results of these queries are illustrated in Figure 6.11

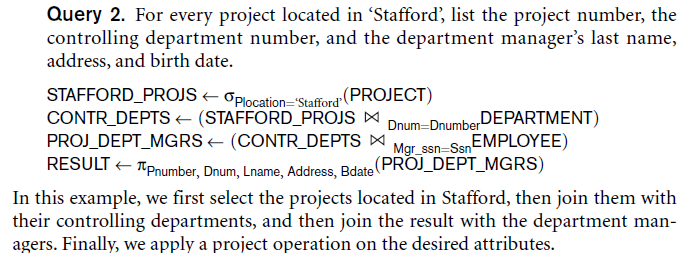
****

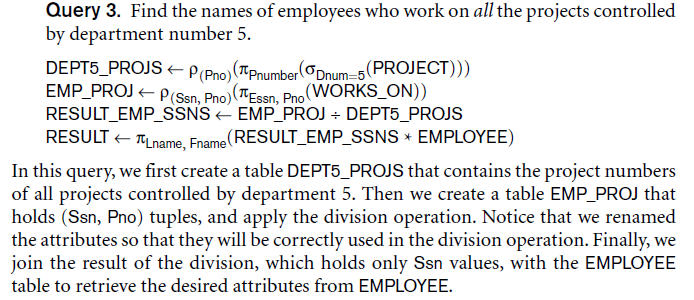
**Figure 3.11**

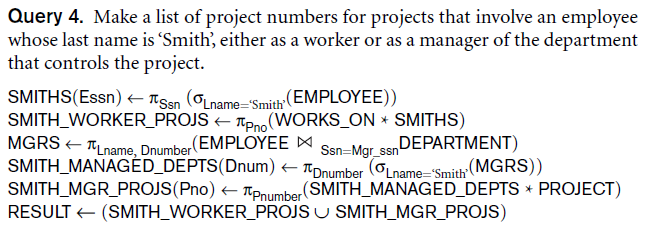
**Examples of Relational Algebra Queries**

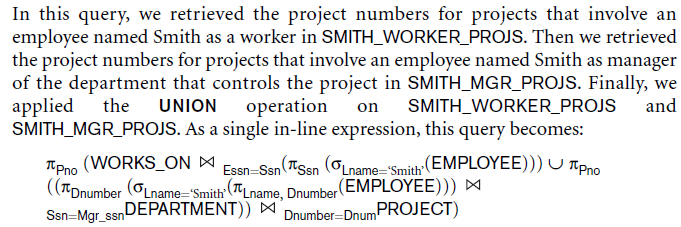
****

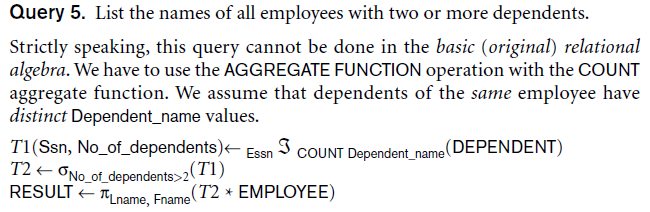
****

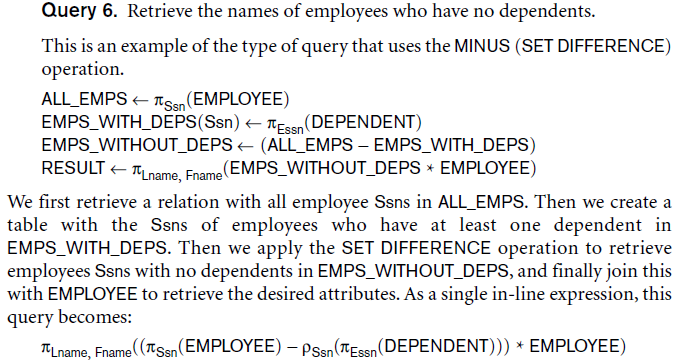
****

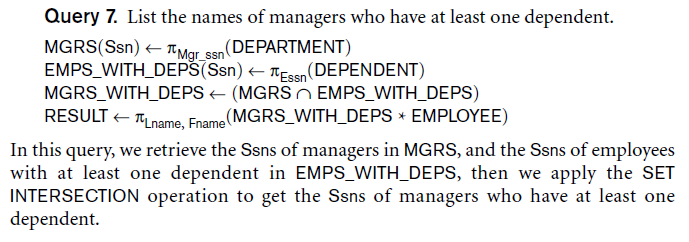
****

****

****

****

****

****