Q. What is a data model? Explain in detail about different data models used in database management systems?

Ans: Data Model – Model is an **abstraction process that hides irrelevant details while highlighting details relevant to the applications** at hand.
Similarly, a data model is **a collection of concepts that can be used to describe structure of a database** and provides the necessary means to achieve this abstraction.

Structure of database means the data types, relationships, and constraints that should hold for the data.
In general a data model consists of two elements:
• A mathematical notation for expressing data and relationships.
• Operations on the data that serve to express queries and other manipulations of the data.

Data Models used in DBMSs:

- **Hierarchical Model -** It was developed to model many types of hierarchical organizations that exist in the real world. It uses tree structures to represent relationship among records.
  In hierarchical model, "no dependent record can occur without its parent record occurrence" and no "dependent record occurrence may be connected to more than one parent record occurrence."
- **Network Model -** It was formalized in the late 1960s by the Database Task Group of the Conference on Data System Language (DBTG/CODASYL).
  It uses two different data structures to represent the database entities and relationships between the entities, namely ***record type*** and ***set type*.**
  In the network model, the relationships as well as the navigation through the database are predefined at database creation time.
- **Relational Model -** The relational model was first introduced by E.F. Codd of the IBM Research in 1970.
  The model uses the concept of a **mathematical relation** (like a table of values) as its basic building block, and has its theoretical basis in set theory and first-order predicate logic.

  The relational model represents the database **as a collection of *relations*.**
- **Object Oriented Model –** This model is based on the object-oriented programming language paradigm.
  It includes the features of OOP like inheritance, object-identity, encapsulation, etc.
  It also supports a rich type system, including structured and collection types.
- **Object Relational Model –** This model combines the features of both **relational model** and object oriented model.
- It extends the traditional relational model with a variety of features such as structured and collection types.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Q. What is a database? Describe the advantages and disadvantages of using of DBMS over File System.

Ans:

- **Database**– A database is a collection of related data and/or information stored so that it is available to many users for different purposes.

Advantages of DBMS

1. **Centralized Management and Control**- One of the main advantages of using a database system is that the organization can exert, via the DBA, centralized management and control over the data.
2. **Reduction of Redundancies and Inconsistencies** - Centralized control avoids unnecessary duplication of data and effectively reduces the total amount of data storage required.
   - **Removing redundancy eliminates inconsistencies.**
3. **Data Sharing** - A database allows the sharing of data under its control by any number of application programs or users.
4. **Data Integrity** - Data integrity means that the data contained in the database is both accurate and consistent.
   - Centralized control can also ensure that adequate checks are incorporated in the DBMS to provide data integrity.
5. **Data Security**- Data is of vital importance to an organization and may be confidential.
   - Such confidential data must not be accessed by unauthorized persons.
   - The DBA who has the ultimate responsibility for the data in the DBMS can ensure that proper access procedures are followed.
   - Different levels of security could be implemented for various types of data and operations.
6. **Data Independence**- Data independence is the capacity to change the schema at one level of a database system without having to change the schema at the next level.
   It is usually considered from two points of view: Physical data independence and logical data independence.

- Physical data independence is the capacity to change the internal schema without having to change conceptual schema.
- Logical data independence is the capacity to change the conceptual schema without having to change external schemas or application programs.

7. **Providing Storage Structures for Efficient Query Processing** -Database systems provide capabilities for efficiently executing queries and updates. Auxiliary files called *indexes* are used for this purpose.
8. **Backup and Recovery** -These facilities are provided to recover databases from hardware and/or software failures.

Some other advantages are:

_ Reduced Application Development Time
_ Flexibility
_ Availability of up-to-date Information

Disadvantages of DBMS

1. **Cost of Software/Hardware and Migration**- A significant disadvantage of the DBMS system is cost.
2. **Reduced Response and Throughput**- The processing overhead introduced by the DBMS to implement security, integrity, and sharing of the data causes a degradation of the response and throughput times.
3. **Problem with Centralization**–

- Centralization also means that the data is accessible from a single source namely the database.
- This increases the potential of security breaches and disruption of the operation of the organization because of downtimes and failures.

Some other disadvantages are:

• Database systems are complex, difficult, and time-consuming to design.
• Substantial hardware and software start-up costs.
• Damage to database affects virtually all applications programs.
• Extensive conversion costs in moving form a file-based system to a database system.
• Initial training required for all programmers and users.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Q. Explain different types of database users and write the functions of DBA?

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

Database Users and User Interfaces

- There are four different types of database-system users, differentiated by the way they expect to interact with the system.
- Different types of user interfaces have been designed for the different types of users.

1. **Naive users** are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously.
- The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naive users may also simply read *reports* generated from the database.

2. **Application programmers** are computer professionals who write application programs.
- Application programmers can choose from many tools to develop user interfaces.
- **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program.
- There are also special types of programming languages that combine imperative control structures (for example, for loops, while loops and if-then-else statements) with statements of the data manipulation language.
- These languages, sometimes called *fourth-generation languages*, often include special features to facilitate the generation of forms and the display of data on the screen. Most major commercial database systems include a fourth generation language.

3. **Sophisticated users** interact with the system without writing programs.
- Instead, they form their requests in a database query language.
- They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the "**storage manager**" understands.
- Analysts who submit queries to explore data in the database fall in this category.

4. **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework.

Among these applications are computer-aided design systems, knowledgebase and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

Functions of Database Administrator:

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrator (DBA)**. The functions of a DBA include:

1. **Schema definition**. The DBA creates the original database schema by executing a set of data definition statements in the DDL.
2. **Storage structure and access-method definition**.
3. **Schema and physical-organization modification**. The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.
4. **Granting of authorization for data access**. By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
5. **Routine maintenance**. Examples of the database administrator's routine maintenance activities are:
   - Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
   - Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
   - Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Q. Classify various features of the ER-Models? How to represent the strong entity and week entity set through ER-diagrams.

The E-R data model employs three basic notions:

- entity sets,
- relationship sets, and
- Attributes.

An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects.
For example, each person in an enterprise is an entity.

An entity has a set of properties, and the values for some set of properties may uniquely identify an entity.
For instance, a person may have a *person-id* property whose value uniquely identifies that person.

An **entity set** is a set of entities of the same type that share the same properties, or attributes.
The set of all persons who are customers at a given bank, for example, can be defined as the entity set *customer*.

An entity is represented by a set of **attributes**.
Attributes are descriptive properties possessed by each member of an entity set.

An entity set may not have sufficient attributes to form a primary key.
Such an entity set is termed a **weak entity set**.
An entity set that has a primary key is termed a **strong entity set**.

For a weak entity set to be meaningful, it must be associated with another entity set, called the **identifying** or **owner entity set**.
Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set.
The identifying entity set is said to **own** the weak entity set that it identifies.
The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.
The identifying relationship is many to one from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is total.

In E-R diagrams, a doubly outlined box indicates a weak entity set, and a doubly outlined diamond indicates the corresponding identifying relationship.

In Figure 2.16, the weak entity set *payment* depends on the strong entity set *loan* via the relationship set *loan-payment*.

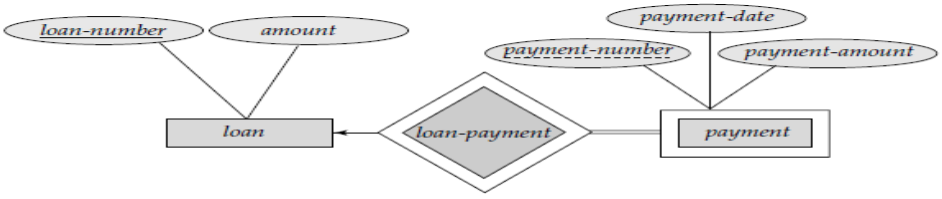


Figure 2.16 E-R diagram with a weak entity set.

The figure also illustrates the use of double lines to indicate *total participation*—the participation of the (weak) entity set *payment* in the relationship *loan-payment* is total, meaning that every payment must be related via *loan-payment* to some loan.

Finally, the arrow from *loan-payment* to *loan* indicates that each payment is for a single loan.

The discriminator of a weak entity set also is underlined, but with a dashed, rather than a solid, line.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Explain the concept of Specialization, generalization and aggregation in E_R diagrams. Give one example for each one of them.

Specialization: An entity set may include sub groupings of entities that are distinct in some way from other entities in the set. For instance, a subset of entities within an entity set may have attributes that are not shared by all the entities in the entity set.

Consider an entity set *person*, with attributes *name*, *street*, and *city*.

A person may be further classified as one of the following:
- *customer*
- *employee*

Each of these person types is described by a set of attributes that includes all the attributes of entity set *person* plus possibly additional attributes.
For example, *customer* entities may be described further by the attribute *customer-id*, whereas *employee* entities may be described further by the attributes *employee-id* and *salary*.

The process of designating sub groupings within an entity set is called **specialization**.
The specialization of *person* allows us to distinguish among persons according to whether they are employees or customers.

An entity set may be specialized by more than one distinguishing feature.
In our example, the distinguishing feature among employee entities is the job the employee performs.

When more than one specialization is formed on an entity set, a particular entity may belong to multiple specializations.
For instance, a given employee may be a temporary employee who is a secretary.
In terms of an E-R diagram, specialization is depicted by a *triangle* component labeled **ISA**, as Figure 2.17 shows.

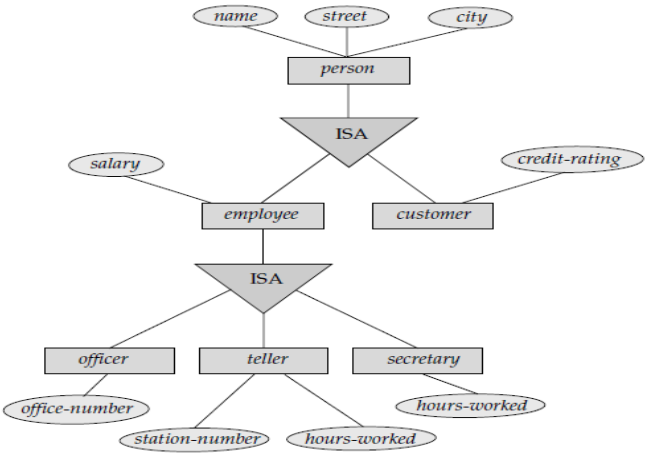


Figure 2.17 Specialization and generalization.

The label ISA stands for "is a" and represents, for example, that a customer "is a" person.
The ISA relationship may also be referred to as a **superclass-subclass** relationship.

Generalization: The refinement from an initial entity set into successive levels of entity subgroupings represents a **top-down** design process in which distinctions are made explicit.

The design process may also proceed in a **bottom-up** manner, in which multiple entity sets are synthesized into a higher-level entity set on the basis of common features.

The database designer may have first identified a
Customer entity set with the attributes *name*, *street*, *city*, and *customer-id*, and an
Employee entity set with the attributes *name*, *street*, *city*, *employee-id*, and *salary*.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

There are similarities between the *customer* entity set and the *employee* entity set in the sense that they have several attributes in common.
This commonality can be expressed by **generalization**, which is a containment relationship that exists between a *higher-level* entity set and one or more *lower-level* entity sets.
In our example, *person* is the higher-level entity set and *customer* and *employee* are lower-level entity sets.

Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively.
The *person* entity set is the superclass of the *customer* and *employee* subclasses.

For all practical purposes, generalization is a simple inversion of specialization.

Aggregation: One limitation of the E-R model is that it cannot express relationships among relationships. To illustrate the need for such a construct, consider the ternary relationship *works-on*, which we saw earlier, between a *employee*, *branch*, and *job*.

One alternative for representing this relationship is to create a quaternary relationship *manages* between *employee*, *branch*, *job*, and *manager*. (A quaternary relationship is required—a binary relationship between *manager* and *employee* **would not permit us to represent** which (*branch*, *job*) combinations of an employee are managed by which manager.)
Using the basic E-R modeling constructs, we obtain the E-R diagram of Figure.

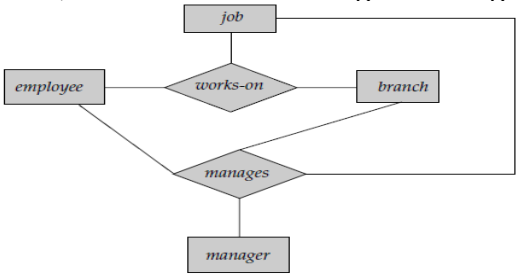


Figure 2.18 E-R diagram with redundant relationships.

It appears that the relationship sets *works-on* and *manages* can be combined into one single relationship set. Nevertheless, we should not combine them into a single relationship, since some *employee*, *branch*, *job* combinations many not have a manager.

The best way to model a situation such as the one just described is to use aggregation.

Aggregation is an abstraction through which relationships are treated as higher level entities.

Thus, for our example, we regard the relationship set *works-on* (relating the entity sets *employee*, *branch*, and *job*) as a higher-level entity set called *works-on*.
Such an entity set is treated in the same manner as is any other entity set.

We can then create a binary relationship *manages* between *works-on* and *manager* to represent who manages what tasks.

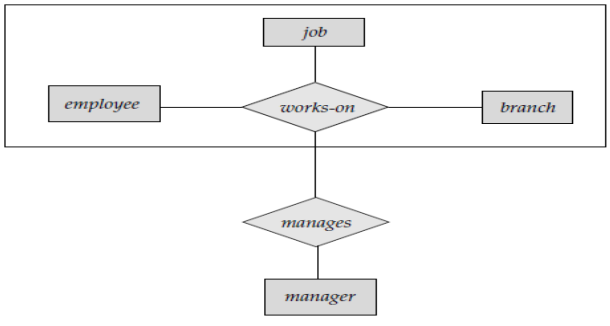


Figure 2.19 E-R diagram with aggregation.

Figure 2.19 shows a notation for aggregation commonly used to represent the above situation.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Q. What is view of data? Explain the three levels of data independence

View of data in DBMS narrate how the data is visualized at each level of data abstraction? **Data abstraction** allow developers to keep complex data structures away from the users. The developers achieve this by hiding the complex data structures through **levels of abstraction**.

For the system to be usable, it must retrieve data efficiently.

- The need for efficiency has led designers to use complex data structures to represent data in the database.
- Since many database-systems users are not computer trained, developers hide the complexity from users through several levels of abstraction or data independence, to simplify users' interactions with the system:

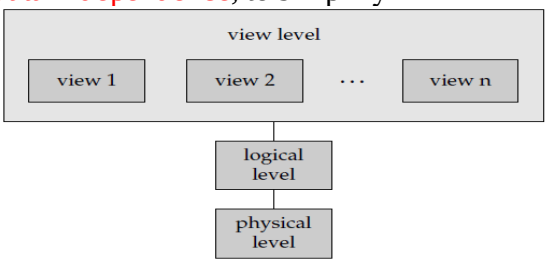


Figure 1.1 The three levels of data abstraction.

1. **Physical level.** The lowest level of data independence describes ***how* the data are actually stored.**
- The physical level describes complex low-level data structures in detail.

2. **Logical level**. The next-higher level of data independence describes ***what* data are stored in the database,** and what relationships exist among those data.
- The logical level thus describes the entire database in terms of a small number of relatively simple structures.
- Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity.
- Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

3. **View level**. The highest level of data independence describes **only part of the entire database.**
- Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database.
- Many users of the database system do not need all this information; instead, they need to access only a part of the database.
- The view level of abstraction exists to simplify their interaction with the system.
- The system may provide many views for the same database.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Explain the structure of RDBMS with a neat sketch?

The main construct for representing data in the relational model or RDBMS is a **relation**.
A relation consists of a **relation schema** and a **relation instance**.
The relation instance is a table, and the relation schema describes the column heads for the table.

The schema specifies the relation's name, the name of each **field** (or **column**, or **attribute**), and the **domain** of each field.

A domain is referred to in a relation schema by the **domain name** and has a set of associated **values**.

Let a relation

Students (*sid:* string, *name:* string, *login:* string, *age:* integer, *gpa:* real)

This says, for instance, that the field named *sid* has a domain named string.
The set of values associated with domain string is the set of all character strings.

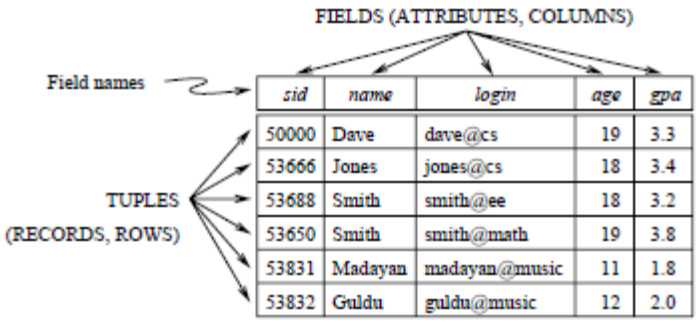
We now turn to the instances of a relation.
An **instance** of a relation is a set of **tuples**, also called **records**, in which each tuple has the same number of fields as the relation schema.

A relation instance can be thought of as a *table* in which each tuple is a *row*, and all rows have the same number of fields.

(The term *relation instance* is often abbreviated to just *relation*, when there is no confusion with other aspects of a relation such as its schema.)

An instance of the Students relation appears in Figure 3.1.
The instance *S*1 contains six tuples and has, as we expect from the schema, five fields.

Note that no two rows are identical.



sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.1 An Instance *S*1 of the Students Relation

This is a requirement of the relational model| each relation is defined to be a *set* of unique tuples or rows.

The order in which the rows are listed is not important.

Figure 3.2 shows the same relation instance.

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53666	Jones	jones@cs	18	3.4
50000	Dave	dave@cs	19	3.3

Figure 3.2 An Alternative Representation of Instance *S*1 of Students

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

If the fields are named, as in our schema definitions and figures depicting relation instances, the order of fields does not matter either.

For example, in SQL the named fields convention is used in statements that retrieve tuples, and the ordered fields convention is commonly used when inserting tuples.

A relation schema specifies the domain of each field or column in the relation instance.
These **domain constraints** in the schema specify an important condition that we want each instance of the relation to satisfy: The values that appear in a column must be drawn from the domain associated with that column.

An instance of R is defined as a set of tuples.

The fields of each tuple must correspond to the fields in the relation schema.

Domain constraints are so fundamental in the relational model that we will henceforth consider only relation instances that satisfy them; therefore, *relation instance* means *relation instance that satisfies the domain constraints in the relation schema*.

The **degree**, also called **arity**, of a relation is the **number of fields**.

The **cardinality** of a relation instance is the **number of tuples in it.**

In the students relation above, the degree of the relation (the number of columns) is five, and the cardinality of this instance is six.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

What are the Integrity constraints? Explain in detail different types of integrity constraints over relations?

An **integrity constraint (IC)** is a *The Relational Model* condition that is specified on a database schema, and restricts the data that can be stored in an instance of the database.
If a database instance satisfies all the integrity constraints specified on the database schema, it is a **legal** instance.
A DBMS **enforces** integrity constraints, in that it permits only legal instances to be stored in the database.

Integrity constraints are specified and enforced at different times:

1. When the DBA or end user defines a database schema, he or she specifies the ICs that must hold on any instance of this database.
2. When a database application is run, the DBMS checks for violations and disallows changes to the data that violate the specified ICs.

(In some situations, rather than disallow the change, the DBMS might instead make some compensating changes to the data to ensure that the database instance satisfies all ICs.

In any case, changes to the database are not allowed to create an instance that violates any IC.)

Many kinds of **integrity constraints** can be specified in the relational model. We have already seen one example of an integrity constraint in the *domain constraints* associated with a relation schema.

In general, other kinds of constraints can be specified as well; for example, no two students have the same *sid* value.

Key Constraints

Consider the Students relation and the constraint that no two students have the same student id.
This IC is an example of a key constraint.

A **key constraint** is a statement that a certain *minimal* subset of the fields of a relation is a unique identifier for a tuple.

A set of fields that uniquely identifies a tuple according to a key constraint is called a **candidate key** for the relation; we often abbreviate this to just *key*.

In the case of the Students relation, the (set of fields containing just the) *sid field* is a candidate key.

Let us take a closer look at the above definition of a (candidate) key.

There are two parts to the definition:

1. Two distinct tuples in a legal instance (an instance that satisfies all ICs, including the key constraint) cannot have identical values in all the fields of a key.
2. No subset of the set of fields in a key is a unique identifier for a tuple.

The first part of the definition means that in *any* legal instance, the values in the key fields uniquely identify a tuple in the instance.

When specifying a key constraint, the **DBA or user must be sure that** this constraint will not prevent them from storing a `correct' set of tuples.

The notion of `correctness' here depends upon the nature of the data being stored.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

For example, several students may have the same name, although each student has a unique student id.

If the *name field* is declared to be a key, the DBMS will not allow the Students relation to contain two tuples describing different students with the same name!

The second part of the definition means, for example, that the set of fields *sid, name* is not a key for Students, because this set properly contains the key *(sid)*.

The set *fsid, name* is an example of a **superkey**, which is a set of fields that contains a key.

Look again at the instance of the Students relation in Figure 3.1.

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53666	Jones	jones@cs	18	3.4
50000	Dave	dave@cs	19	3.3

Observe that two different rows always have different *sid* values; *sid* is a key and uniquely identifies a tuple.

However, this does not hold for non-key fields. For ex, the relation contains two rows with *Smith* in the *name field*.

Note that every relation is guaranteed to have a key. Since a relation is a set of tuples, **the set of *all fields* is always a superkey.**

If other constraints hold, some subset of the fields may form a key, but if not, the set of all fields is a key.

A relation may have several candidate keys.

For example, the *login* and *age fields* of the Students relation may, taken together, also identify students uniquely. That is, *(login, age)* is also a key. It may seem that *login* is a key, since no two rows in the example instance have the same *login* value.

However, the key must identify tuples uniquely in all possible legal instances of the relation.

By stating that *(login, age)* is a key, the user is declaring that two students may have the same login or age, but not both. Out of all the available candidate keys, a database designer can identify a **primary** key.

Intuitively, a tuple can be referred to from elsewhere in the database by storing the values of its primary key fields. For example, we can refer to a Students tuple by storing its *sid* value.
As a consequence of referring to student tuples in this manner, tuples are frequently accessed by specifying their *sid* value. In principle, we can use any key, not just the primary key, to refer to a tuple.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Discuss about different operations in relational algebra with example.

Relational algebra is one of the two formal query languages associated with the relational model.
Queries in algebra are composed using a collection of operators.

A fundamental property is that every operator in the algebra accepts (one or two) relation instances as arguments and returns a relation instance as the result. This property makes it easy to *compose* operators to form a complex query a **relational algebra expression** is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions.

Selection and Projection

Relational algebra includes operators to *select* rows from a relation (σ) and to *project* columns (π).

These operations allow us to manipulate data in a single relation.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Figure 4.1 Instance $S1$ of Sailors

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

Figure 4.2 Instance $S2$ of Sailors

sid	*bid*	*day*
22	101	10/10/96
58	103	11/12/96

Figure 4.3 Instance $R1$ of Reserves

Consider the instance of the Sailors relation shown in Figure 4.2, denoted as *S2*. We can retrieve rows corresponding to expert sailors by using the σ operator. The expression

$$\sigma_{rating>8}(S2)$$

evaluates to the relation shown in Figure 4.4.
The subscript *rating>8* specifies the selection criterion to be applied while retrieving tuples.

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
58	Rusty	10	35.0

Figure 4.4 $\sigma_{rating>8}(S2)$

sname	*rating*
yuppy	9
Lubber	8
guppy	5
Rusty	10

Figure 4.5 $\pi_{sname, rating}(S2)$

The selection operator σ specifies the tuples to retain through a *selection condition*.

The schema of the result of a selection is the schema of the input relation instance. The projection operator π allows us to extract columns from a relation;
for example, we can find out all sailor names and ratings by using π.
The expression evaluates to the relation shown in Figure 4.5.

$$\pi_{sname,\, rating}(S2)$$

The subscript *sname, rating* specifies the fields to be retained; the other fields are `projected out.'

Suppose that we wanted to find out only the ages of sailors. The expression

$$\pi_{age}(S2)$$

Renaming

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

It is therefore convenient to be able to give names explicitly to the fields of a relation instance that is defined by a relational algebra expression. In fact, it is often convenient to give the instance itself a name so that we can break a large algebra expression into smaller pieces by giving names to the results of subexpressions.

The **renaming** operator ρ for this purpose.
The expression $\rho(R(\overline{F});E)$ takes an arbitrary relational algebra expression E and returns an instance of a (new) relation called R.

R contains the same tuples as the result of E, and has the same schema as E, but some fields are renamed.

The field names in relation R are the same as in E, except for fields renamed in the *renaming list* $\overline{F}$, which is a list of terms having the form *oldname*→ *newname* or *position* → *newname*.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Identify various operations in relational calculus in creating SQL query with suitable example.

Relational calculus is an alternative to relational algebra.

In contrast to the algebra, which is procedural,
The calculus is nonprocedural, or *declarative*, in that it allows us to describe the set of answers without being explicit about how they should be computed.

The variant of the calculus that we present in detail is called the **tuple relational calculus (TRC).**

Variables in TRC take on tuples as values.

In another variant, called the domain relational calculus (DRC), the variables range over field values.

Tuple Relational Calculus

A **tuple variable** is a variable that takes on tuples of a particular relation schema as values.

That is, every value assigned to a given tuple variable has the same number and type of fields.

A tuple relational calculus query has the form
{*T*|*p*(*T*)},
Where *T* is a tuple variable and *p*(*T*) denotes a *formula* that describes *T*;
(set of all tuples *T* such that predicate p is true for T.)

The result of this query is the set of all tuples *t* for which the formula *p*(*T*) evaluates to true with *T* = *t*.

As a simple example, consider the following query.

Ex: Find all sailors with a rating above 7.

$$\{S \mid S \in Sailors \wedge S.rating > 7\}$$

When this query is evaluated on an instance of the Sailors relation, the tuple variable *S* is instantiated successively with each tuple, and the test *S.rating*>*7* is applied.

Domain Relational Calculus

- A **domain variable** is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers).
- A DRC query has the form {<*x*1, *x*2, ... , *xn*>|
- *p*(<*x*1, *x*2, ... , *xn*>)}, where each *xi* is either a *domain variable* or a constant and *p*(<*x*1, *x*2, ... , *xn*>) denotes a **DRC formula** whose only free variables are the variables among the *xi*, 1 ≤ *n*.
- The result of this query is the set of all tuples <*x*1, *x*2, ... , *xn*> for which the formula evaluates to true.

Ex: *Find all sailors with a rating above 7*

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in Sailors \wedge T > 7\}$$

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Discuss about Nested queries with an example.

One of the most powerful features of SQL is nested queries.
A **nested query** is a query that has another query embedded within it; the embedded query is called a **subquery**.

When writing a query, we sometimes need to express a condition that refers to a table that must itself be computed. The query used to compute this subsidiary table is a subquery and appears as part of the main query.

A subquery typically appears within the WHERE clause of a query.
Subqueries can sometimes appear in the FROM clause or the HAVING clause.

As an example, let us rewrite the following query, which we discussed earlier, using a nested subquery:

Ex: Find the names of sailors who have reserved boat 103.

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (    SELECT R.sid
                    FROM Reserves R
                    WHERE R.bid = 103 )
```

The nested subquery computes the (multi)set of *sid*s for sailors who have reserved boat 103 (the set contains 22, 31, and 74 on instances *R*2 and *S*3), and the top-level query retrieves the names of sailors whose *sid* is in this set.

Correlated Nested Queries

In the nested queries that we have seen thus far, the inner subquery has been completely independent of the outer query.

In general the inner subquery could depend on the row that is currently being examined in the outer query (in terms of our conceptual evaluation strategy).

Let us rewrite the following query once more:

Ex: Find the names of sailors who have reserved boat number 103.

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (    SELECT *
                  FROM Reserves R
                  WHERE R.bid = 103
                        AND R.sid = S.sid )
```

The EXISTS operator is another set comparison operator, such as IN.
It allows us to test whether a set is nonempty.
Thus, for each Sailor row *S*, we test whether the set of Reserves rows *R* such that *R.bid = 103* AND *S.sid = R.sid* is nonempty.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Discuss about different types of aggregate operators in SQL with examples?

- SQL supports a powerful class of constructs for computing *aggregate*
- *values such as MIN and SUM.*
- *These features represent a significant extension of relational* algebra.
- SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
   Count the number of different sailor names.

```
SELECT COUNT ( DISTINCT S.sname )
FROM Sailors S
```

2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
   Ex: find sum of salary received by sailors

```
SELECT SUM (S.salary )
FROM Sailors S
```

3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
   Ex: *Find the average age of all sailors.*

```
SELECT AVG (S.age)
FROM Sailors S
```

4. MAX (A): The maximum value in the A column.
   Ex: *Find the name and age of the oldest sailor. Consider the following attempt to* answer this query:

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age = ( SELECT MAX (S2.age)
FROM Sailors S2 )
```

5. MIN (A): The minimum value in the A column.
   Ex: *Find the name and age of the youngest sailor. Consider the following attempt to* answer this query:

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age = ( SELECT MIN (S2.age)
FROM Sailors S2 )
```


P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Classify different join operations (Relational Algebra & SQL) and explain with example

SQL supports several other ways in which information from two or more relations can be **joined** together.
There are a total of five JOINs. They are :

1. JOIN or INNER JOIN
2. OUTER JOIN

   2.1 LEFT OUTER JOIN or LEFT JOIN
   2.2 RIGHT OUTER JOIN or RIGHT JOIN
   2.3 FULL OUTER JOIN or FULL JOIN

3. NATURAL JOIN
4. CROSS JOIN
5. SELF JOIN

1. JOIN or INNER JOIN :
In this kind of a JOIN, we get all records that match the condition in both the tables, and records in both the tables that do not match are not reported. In other words, INNER JOIN is based on the single fact that: ONLY the matching entries in BOTH the tables SHOULD be listed. Note that a JOIN without any other JOIN keywords (like INNER, OUTER, LEFT, etc) is an INNER JOIN. In other words, JOIN is a Syntactic sugar for INNER JOIN

2. OUTER JOIN :
OUTER JOIN retrieves Either, the matched rows from one table and all rows in the other table Or, all rows in all tables (it doesn't matter whether or not there is a match). There are three kinds of Outer Join :

2.1 LEFT OUTER JOIN or LEFT JOIN
This join returns all the rows from the left table in conjunction with the matching rows from the right table. If there are no columns matching in the right table, it returns NULL values.

2.2 RIGHT OUTER JOIN or RIGHT JOIN
This JOIN returns all the rows from the right table in conjunction with the matching rows from the left table. If there are no columns matching in the left table, it returns NULL values.

2.3 FULL OUTER JOIN or FULL JOIN
This JOIN combines LEFT OUTER JOIN and RIGHT OUTER JOIN. It returns row from either table when the conditions are met and returns NULL value when there is no match. In other words, OUTER JOIN is based on the fact that : ONLY the matching entries in ONE OF the tables (RIGHT or LEFT) or BOTH of the tables(FULL) SHOULD be listed. Note that `OUTER JOIN` is a loosened form of `INNER JOIN`.

3. NATURAL JOIN :
It is based on the two conditions :
1. the JOIN is made on all the columns with the same name for equality.
2. Removes duplicate columns from the result.

This seems to be more of theoretical in nature and as a result (probably) most DBMS don't even bother supporting this.

4. CROSS JOIN (Cartesian product) :
It is the Cartesian product of the two tables involved. The result of a CROSS JOIN will not make sense in most of the situations. Moreover, we wont need this at all (or needs the least, to be precise).

5. SELF JOIN :
It is not a different form of JOIN, rather it is a JOIN (INNER, OUTER, etc) of a table to itself.
JOINs based on Operators:

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Depending on the operator used for a JOIN clause, there can be two types of JOINs. They are

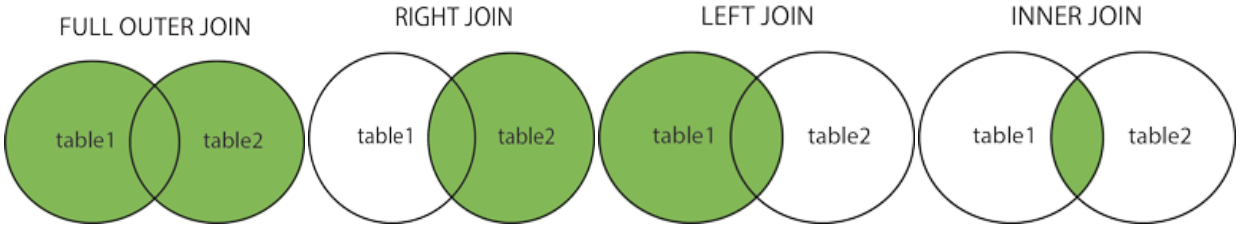
1. Equi JOIN
2. Theta JOIN (Non Equi JOIN)
3.

1. Equi JOIN :

For whatever JOIN type (INNER, OUTER, etc), if we use ONLY the equality operator (=), then we say that the JOIN is an EQUI JOIN.

2. Theta JOIN (Non Equi JOIN) :

This is same as EQUI JOIN but it allows all other operators like >, <, >= etc.



P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Explain Active Databases and designing Active Databases with suitable example.

- A **trigger** is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA.
- A database that has a set of associated triggers is called an **active database**
- Triggers offer a powerful mechanism for dealing with changes to a database, but they must be used with caution.
- The effect of a collection of triggers can be very complex, and maintaining an active database can become very difficult.
- Often, a judicious use of integrity constraints can replace the use of triggers.
- In an active database system, when the DBMS is about to execute a statement that modifies the database, it checks whether some trigger is activated by the statement.
- If so, the DBMS processes the trigger by evaluating its condition part, and then (if the condition evaluates to true) executing its action part.
- If a statement activates more than one trigger, the DBMS typically processes all of them, in some arbitrary order.
- An important point is that the execution of the action part of a trigger could in turn activate another trigger.
- In particular, the execution of the action part of a trigger could again activate the same trigger; such triggers are called **recursive triggers**.

Ex: Set-Oriented Trigger:

```
CREATE TRIGGER set count AFTER INSERT ON Students /* Event */
REFERENCING NEW TABLE AS InsertedTuples
FOR EACH STATEMENT
   INSERT
   /* Action */
INTO StatisticsTable(ModiedTable, ModicationType, Count)
SELECT `Students', `Insert', COUNT *
FROM InsertedTuples
WHERE I.age < 18
```


P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

List various set operations and explain with suitable examples.

The following standard operations on sets are also available in relational algebra:

- *Union* (υ),
- *intersection* ($\cap$),
- *set-difference* ($-$), and
- *cross-product* (x).

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Figure 4.1 Instance $S1$ of Sailors

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

Figure 4.2 Instance $S2$ of Sailors

Union:

- $R \upsilon S$ returns a relation instance containing all tuples that occur in *either* relation instance R or relation instance S (or both).
- R and S must be *union compatible*, and the schema of the result is defined to be identical to the schema of R.
- Two relation instances are said to be **union-compatible** if the following conditions hold:
  - they have the same number of the fields, and
  - corresponding fields, taken in order from left to right, have the same *domains*.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0
28	yuppy	9	35.0
44	guppy	5	35.0

Figure 4.8 $S1 \cup S2$

Intersection: $R \cap S$ returns a relation instance containing all tuples that occur in *both* R and S.

sid	*sname*	*rating*	*age*
31	Lubber	8	55.5
58	Rusty	10	35.0

Figure 4.9 $S1 \cap S2$

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0

Figure 4.10 $S1 - S2$

Set-difference: $R-S$ returns a relation instance containing all tuples that occur in R but not in S.

Cross-product: RS returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S).
The result of R S contains one tuple $<r, s>$ (the concatenation of tuples r and s) for each pair of tuples $r \epsilon R$; $s \epsilon S$.
The cross-product operation is sometimes called **Cartesian product**.

(*sid*)	*sname*	*rating*	*age*	(*sid*)	*bid*	*day*
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

Figure 4.11 $S1 \times R1$

Discuss about trigger with syntax and example.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

- A **trigger** is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA.
- A trigger description contains three parts:
- **Event**: A change to the database that **activates** the trigger.
- **Condition**: A query or test that is run when the trigger is activated.
- **Action**: A procedure that is executed when the trigger is activated and its condition is true.
- A trigger can be thought of as a `daemon' that monitors a database, and is executed when the database is modified in a way that matches the *event* specification.
- An insert, delete or update statement could activate a trigger, regardless of which user or application invoked the activating statement;
- users may not even be aware that a trigger was executed as a side effect of their program.
- A *condition* in a trigger can be a true/false statement (e.g., all employee salaries are less than $100,000) or a query.
- A query is interpreted as *true* if the answer set is nonempty, and *false* if the query has no answers.
- If the condition part evaluates to true, the action associated with the trigger is executed.

Ex:
```
CREATE TRIGGER init_count BEFORE INSERT ON Students        /* Event */
    DECLARE
        count INTEGER;
    BEGIN
        /* Action */
        count := 0;
    END
```

Summarize key terms and Rules for functional dependency.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

- A **functional dependency** (FD) is a kind of IC that generalizes the concept of a *key*.
- Let *R* be a relation schema and let *X* and *Y* be nonempty sets of attributes in *R*.
- We say that an instance *r* of *R* satisfies the

  FD $X \rightarrow Y$

if the following holds for every pair of tuples *t*1 and *t*2 in *r*:

- If $t1.X = t2.X$, then $t1.Y = t2.Y$.

- The set of all FDs implied by a given set *F* of FDs is called the **closure of F** and is denoted as *F*+.
- An important question is how we can **infer**, or compute, the closure of a given set *F* of FDs.
- The answer is simple and elegant.
- The following three rules, called **Armstrong's Axioms**, can be applied repeatedly to infer all FDs implied by a set *F* of FDs.
- We use *X*, *Y*, and *Z* to denote *sets* of attributes over a relation schema *R*:

  - **Reflexivity:** If $X \supseteq Y$, then $X \rightarrow Y$.
  - **Augmentation:** If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z.
  - **Transitivity:** If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

- Armstrong's Axioms are **sound** in that they generate only FDs in *F*+ when applied to a set *F* of FDs.
- They are **complete** in that repeated application of these rules will generate all FDs in the closure *F*+.
- It is convenient to use some additional rules while reasoning about *F*+:
- **Union:** If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.
- **Decomposition:** If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$.

Demonstrate functional dependencies. How are primary keys related to FD's?

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

- A **functional dependency** (FD) is a kind of IC that generalizes the concept of a *key*.
- Let *R* be a relation schema and let *X* and *Y* be nonempty sets of attributes in *R*.
- We say that an instance *r* of *R* satisfies the

  FD $X \rightarrow Y$

if the following holds for every

- pair of tuples *t*1 and *t*2 in *r*:
- If *t*1*:X* = *t*2*:X*, then *t*1*:Y* = *t*2*:Y* .

- We use the notation *t*1.*X* to refer to the projection of tuple *t*1 onto the attributes in *X*, in a natural extension of our TRC notation *t.a* for referring to attribute *a* of tuple *t*.
- An FD $X \rightarrow Y$ essentially says that if two tuples agree on the values in attributes *X*, they must also agree on the values in attributes *Y*.

- primary key constraint is a special case of an FD.
- The attributes in the key play the role of *X*, and the set of all attributes in the relation plays the role of *Y*.
- Note, however, that the definition of an FD does not require that the set *X* be minimal;
- The additional minimality condition must be met for *X* to be a key.
- If $X \rightarrow Y$ holds, where *Y* is the set of all attributes, and there is some subset *V* of *X* such that $V \rightarrow Y$ holds, then *X* is a *superkey*; if *V* is a strict subset of *X*, then *X* is not a key.

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

Explain trivial and non-trivial dependencies.

1. **Trivial Functional dependency:**

Functional dependency which also known as a trivial dependency occurs when A->B holds true where B is a subset of A.

- A → B has trivial functional dependency if B is a subset of A.
- The following dependencies are also trivial like: A → A, B → B

Example:

Consider a table with two columns Employee_Id and Employee_Name.

{Employee_id, Employee_Name} → Employee_Id is a trivial functional dependency as Employee_Id is a subset of {Employee_Id, Employee_Name}.

2. **Non trivial functional dependency in DBMS**

Functional dependency which also known as a nontrivial dependency occurs when A->B holds true where B is not a subset of A.

In a relationship, if attribute B is not a subset of attribute A, then it is considered as a non-trivial dependency.

- A → B has a non-trivial functional dependency if B is not a subset of A.
- When A intersection B is NULL, then A → B is called as complete non-trivial.

Example:

ID → Name,

Name → DOB

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)

What is decomposition and how does it address redundancy? What problem may be caused by the use of decompositions?

A **decomposition of a relation schema** *R* consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of *R* and together include all attributes in *R*.

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural. Functional dependencies (and, for that matter, other ICs) can be used to identify such situations and to suggest refinements to the schema. The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of `smaller' relations. Each of the smaller relations contains a (strict) subset of the attributes of the original relation. We refer to this process as *decomposition* of the larger relation into the smaller relations.

We can deal with the redundancy in Hourly Emps by decomposing it into two relations:

Hourly_Emps2(*ssn*, *name*, *lot*, *rating*, *hours_worked*)

Wages(*rating*, *hourly_wages*)

Unless we are careful, decomposing a relation schema can create more problems than it solves. Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problems (if any) does a given decomposition cause?

To help with the first question, several *normal forms* have been proposed for relations.

If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. Considering the normal form of a given relation schema can help us to decide whether or not to decompose it further. If we decide that a relation schema must be decomposed further, we must choose a particular decomposition (i.e., a particular collection of smaller relations to replace the given relation). With respect to the second question, two properties of decompositions are of particular interest.

The ***lossless-join* property** enables us to recover any instance of the decomposed relation from corresponding instances of the smaller relations.

The ***dependency preservation*** property enables us to enforce any constraint on the original relation by simply enforcing some constraints on each of the smaller relations. That is, we need not perform joins of the smaller relations to check whether a constraint on the original relation is violated.

A serious drawback of decompositions is that queries over the original relation may require us to join the decomposed relations. If such queries are common, the performance penalty of decomposing the relation may not be acceptable. In this case we may choose to live with some of the problems of redundancy and not decompose the relation. It is important to be aware of the potential problems caused by such residual redundancy in the design and to take steps to avoid them (e.g., by adding some checks to application code).

P. V Ramana Murthy
LEE; B.E(Comp); M.Tech(CS); (Ph.D(CSE));
Malla Reddy Engineering College (Autonomous)