**Sardar Patel University Balaghat(M.P)**

**School of Engineering & Technology**

**Department of CSE**

**Subject Name: Database Management System**

**Subject Code: CSE503**

**Semester: 5th**

**UNIT-I**

## INTRODUCTION

**Data:-**Data is a plural of datum, which is originally a Latin noun meaning "something given." Today, data is used in English both as a plural noun meaning "facts or pieces of information" and as a singular mass noun meaning "information".

Data can be defined as a representation of facts, concepts or instructions in a formalized manner which should be suitable for communication, interpretation, or processing by human or electronic machine.

Data is represented with the help of characters like alphabets (A-Z,a-z), digits (0-9) or special characters(+,-, /,*, <,>, = etc.).

Types of Data: Text, Numbers & Multimedia

**Information: -**

Information is organized or classified data which has some meaningful values for the receiver.

Information is the processed data on which decisions and actions are based.

For the decision to be meaningful, the processed data must qualify for the following characteristics:

- **Timely -** Information should be available when required.
- **Accuracy -** Information should be accurate.
- **Completeness -** Information should be complete.

**File System**

Before DBMS invented, information was stored in file processing system. A file processing system is a collection of files and programs that access/modify these files. Typically, new files and programs are added over time (by different programmers) as new information needs to be stored and new ways to access information are needed.

Problems with file processing systems:

**File System Verses Database**

Most explicit and major disadvantages of file system when compared to database management system are as follows:

**Data Redundancy**- The files are created in the file system as and when required by an enterprise over its growth path. So, in that case the repetition of information about an entity cannot be avoided. E.g. The addresses of customers will be present in the file maintaining information about customers holding savings account and also the address of the customers will be present in file maintaining the current account. Even when same customers have a saving account and current account his address will be present at two places.

**Data Inconsistency**: Data redundancy leads to greater problem than just wasting the storage i.e. it may lead to inconsistent data. Same data which has been repeated at several places may not match after it has been updated at some places.

For example: Suppose the customer requests to change the address for his account in the Bank and the Program is executed to update the saving bank account file only but his current bank account file is not updated. Afterwards the addresses of the same customer present in saving bank account file and current bank account file will not match.

Moreover, there will be no way to find out which address is latest out of these two.

**Difficulty in Accessing Data**: For generating ad hoc reports the programs will not already be present and only options present will to write a new program to generate requested report or to work manually. This is going to take impractical time and will be more expensive.

For example: Suppose all of sudden the administrator gets a request to generate a list of all the customers holding the saving banks account who lives in particular locality of the city. Administrator will not have any program already written to generate that list but say he has a program which can generate a list of all the

customers holding the savings account. Then he can either provide the information by going thru the list manually to select the customers living in the particular locality or he can write a new program to generate the new list. Both of these ways will take large time which would generally be impractical.

**Data Isolation**: Since the data files are created at different times and supposedly by different people the structures of different files generally will not match. The data will be scattered in different files for a particular entity. So, it will be difficult to obtain appropriate data.

For example: Suppose the Address in Saving Account file have fields: Add line1, add line2, City, State, Pin while the fields in address of Current account are: House No., Street No., Locality, City, State, and Pin. Administrator is asked to provide the list of customers living in a particular locality. Providing consolidated list of all the customers will require looking in both files. But they both have different way of storing the address.

Writing a program to generate such a list will be difficult.

**Integrity Problems**: All the consistency constraints have to be applied to database through appropriate checks in the coded programs. This is very difficult when number such constraint is very large.

For example: An account should not have balance less than Rs. 500. To enforce this constraint appropriate check should be added in the program which add a record and the program which withdraw from an account. Suppose later on this amount limit is increased then all those checks should be updated to avoid inconsistency. These time to time changes in the programs will be great headache for the administrator.

Security and access control: Database should be protected from unauthorized users. Every user should not be allowed to access every data. Since application programs are added to the system. For example: The Payroll Personnel in a bank should not be allowed to access accounts information of the customers.

**Concurrency Problems**: When more than one user is allowed to process the database.

If in that environment two or more users try to update a shared data element at about the same time then it may result into inconsistent data. For example, Suppose Balance of an account is Rs. 500. And User A and B try to withdraw Rs. 100 and Rs. 50 respectively at almost the same time using the Update process.

Update:

1. Read the balance amount.
2. Subtract the withdrawn amount from balance.
3. Write updated Balance value.

Suppose A performs Step 1 and 2 on the balance amount i.e. it reads 500 and subtracts 100 from it. But at the same time B withdraws Rs 50 and he performs the Update process and he also reads the balance as 500 subtract 50 and writes back 450. User A will also write his updated Balance amount as 400. They may update the Balance value in any order depending on various reasons concerning to system being used by both of the users. So finally, the balance will be either equal to 400 or 450. Both of these values are wrong for the updated balance and so now the balance amount is having inconsistent value forever.

**Advantages of database systems**

- **Data Independence**: Application programs should not, ideally, be exposed to details of data representation and storage, The DBMS provides an abstract view of the data that hides such details.
- **Efficient Data Access:** A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.
- **Data Integrity and Security:** If data is always accessed through the DBMS, the DBMS can enforce integrity constraints. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, it can enforce access controls that govern what data is visible to different classes of users.
- **Data Administration:** When several users share the data, centralizing the administration of data can offer significant improvements. Experienced professionals, who understand the nature of the data being managed, and how different groups of users use it, can be responsible for organizing the data representation to minimize redundancy and for fine-tuning the storage of the data to make retrieval efficient.

- **Concurrent Access and Crash Recovery:** A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.
- **Reduced Application Development Time:** Clearly, the DBMS supports important functions that are common to many applications accessing data in the DBMS. This, in conjunction with the high-level interface to the data, facilitates quick application development. DBMS applications are also likely to be more robust than similar stand-alone applications because many important tasks are handled by the DBMS (and do not have to be debugged and tested in the application).

**What Is Database:**

A database consists of an organized collection of interrelated data for one or more uses, typically in digital form. Examples of databases could be: Database for Educational Institute or a Bank, Library, Railway Reservation system etc.

- Consists of two things- a Database and a set of programs.
- Database is a very large, integrated collection of data.
- The set of programs are used to Access and Process the database.

So, DBMS can be defined as the software package designed to store and manage or process the database.

**Management of data involves**

- Definition of structures for the storage of information
- Methods to manipulate information
- Safety of the information stored despite system crashes.

Database model's real-world enterprise by entities and relationships.
Entities (e.g., students, courses, class, subject)

**CHARACTERISTICS OF DATABASE APPROACH**

A number of characteristics distinguish the database approach from the much older approach of programming with files. In traditional file processing, each user defines and implements the files needed for a specific software application as part of programming the application.

For example, one user, the grade reporting office, may keep files on students and their grades. Programs to print a student's transcript and to enter new grades are implemented as part of the application.

A second user, the accounting office, may keep track of students' fees and their payments. Although both users are interested in data about students, each user maintains separate files—and programs to manipulate these files—because each requires some data not available from the other user's files. This redundancy in defining and storing data results in wasted storage space and in redundant efforts to maintain common up-to-date data. In the database approach, a single repository maintains data that is defined once and then accessed by various users. In file systems, each application is free to name data elements independently. In contrast, in a database, the names or labels of data are defined once, and used repeatedly by queries, transactions, and applications.

The main characteristics of the database approach versus the file-processing approach are the following:

- Self-describing nature of a database system
- Insulation between programs and data, and data abstraction
- Support of multiple views of the data
- Sharing of data and multiuser transaction processing.

**Data models**

A data model—a collection of concepts that can be used to describe the structure of a database—provides the necessary means to achieve the abstraction. By structure of a database we mean the data types, relationships,

and constraints that apply to the data. Most data models also include a set of basic operations for specifying retrievals and updates on the database.

In addition to the basic operations provided by the data model, it is becoming more common to include concepts in the data model to specify the dynamic aspect or behavior of a database application. This allows the database designer to specify a set of valid user-defined operations that are allowed on the database objects.

Data models define how the logical structure of a database is modeled. Data Models are fundamental entities to introduce abstraction in a DBMS. Data models define how data is connected to each other and how they are processed and stored inside the system.

**Types of Database Users:**

Users are differentiated by the way they expect to interact with the system:

1. **Application programmers** - interact with system through DML calls.
2. **Sophisticated users** - form requests in a database query language.
3. **Specialized users** - write specialized database applications that do not fit into the traditional data processing framework.
4. **Naive users** - invoke one of the permanent application programs that have been written previously.

**Three Level Architecture of DBMS-**

An early proposal for a standard terminology and general architecture database a system was produced in 1971 by the DBTG (Data Base Task Group) appointed by the Conference on data Systems and Languages. The DBTG recognized the need for a two-level approach with a system view called the schema and user view called subschema. The American National Standard Institute terminology and architecture in 1975.ANSI-SPARC recognized the need for a three-level approach with a system catalog.

**There are following three levels or layers of DBMS architecture:**

1. External Level
2. Conceptual Level
3. Internal Level

**1. External Level**: - External Level is described by a schema i.e. it consists of definition of logical records and relationship in the external view. It also contains the method of deriving the objects in the external view from the objects in the conceptual view.

**2. Conceptual Level**: - Conceptual Level represents the entire database. Conceptual schema describes the records and relationship included in the Conceptual view. It also contains the method of deriving the objects in the conceptual view from the objects in the internal view.

**3. Internal Level: -** Internal level indicates hoe the data will be stored and described the data structures and access method to be used by the database. It contains the definition of stored record and method of representing the data fields and access aid used.

A mapping between external and conceptual views gives the correspondence among the records and relationship of the conceptual and external view. The external view is the abstraction of conceptual view which in turns is the abstraction of internal view. It describes the contents of the database as perceived by the user or application program of that view.

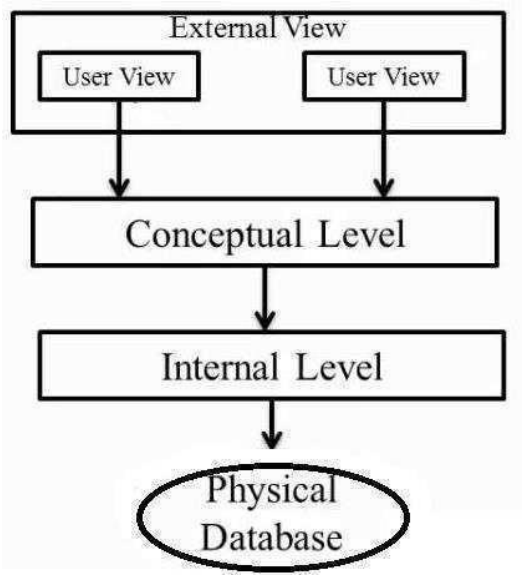



**Fig-1.1**

**Difference between Two Tier and Three Tier Architecture**

| Sr. No | Two-tier Architecture | Three -tier Architecture |
|---|---|---|
| 1 | Client -Server Architecture | Web -based application |
| 2 | Client will hit request directly to server and client will get response directly from server | Here in between client and server middle ware will be there, if client hits a request it will go to the middle ware and middle ware will send to server and vice versa. |
| 3 | 2-tier means 1) Design layer 2) Data layer | 3-tier means 1) Design layer 2) Business layer or Logic layer 3) Data layer |

**Three-Tier Architecture:**

**Three-tier architecture** typically comprises a presentation tier, a business or data access tier, and a data tier. Three layers in the three-tier architecture are as follows:

**1) Client layer**
**2) Business layer**
**3) Data layer**

**1) Client layer:**

It is also called as Presentation *layer* which contains UI part of our application. This layer is used for the design purpose where data is presented to the user or input is taken from the user.

Example-Designing registration form which contains text box, label, button etc.

**2) Business layer:**

In this layer all business logic written likes validation of data, calculations, data insertion etc. This acts as an interface between Client layer and Data Access Layer. This layer is also called the intermediary layer helps to

make communication faster between client and data layer.

**3) Data layer:**

In this layer, actual database is coming in the picture. Data Access Layer contains methods to connect with database and to perform insert, update, delete, get data from database based on our input data.

**Advantages**

1. High performance, lightweight persistent objects
2. Scalability – Each tier can scale horizontally
3. Performance – Because the Presentation tier can cache requests, network utilization is minimized, and the load is reduced on the Application and Data tiers.
4. High degree of flexibility in deployment platform and configuration
5. Better Re-use
6. Improve Data Integrity
7. Improved Security – Client is not direct access to database.
8. Easy to maintain and modification is bit easy, won't affect other modules
9. In three tier architecture application performance is good.

**Disadvantages**

1. Increase Complexity/Effort

**RDBMS-It stands for Relational Database Management System. It organizes data into related rows and columns.**

**Features:**

It stores data in tables.
Tables have rows and column.
These tables are created using SQL.

**Difference between DBMS and RDBMS**

| **No.** | **DBMS** | **RDBMS** |
|---|---|---|
| 1) | DBMS applications store **data asfile**. | RDBMS applications store **data in a tabular form**. |
| 2) | In DBMS, data is generally stored in either a hierarchical form or a navigational form. | In RDBMS, the tables have an identifier called primary key and the data values are stored in the form of tables. |
| 3) | **Normalization is not** present inDBMS. | **Normalization is** present in RDBMS. |
| 4) | DBMS does **not apply any security** with regards to data manipulation. | RDBMS **defines the integrity constraint** for the purpose of ACID (Atomicity, Consistency, Isolation and Durability) property. |
| 5) | DBMS uses file system to store data, so there will be **no relation between the tables**. | In RDBMS, data values are stored in the form of tables, so a **relationship** between these data values will be stored in the form of a table as well. |
| 6) | DBMS has to provide some uniformmethods to information. | RDBMS system supports a tabular structure of thedata and a relationship between them to access the stored information. |

| | | |
|---|---|---|
| 7) | DBMS **does not support distributeddatabase**. | RDBMS **supports distributed database**. |
| 8) | DBMS is meant to be for small organization and **deal with smalldata**. It supports **single user**. | RDBMS is designed to **handle large amount of data**. It supports **multiple users**. |
| 9) | Examples of DBMS are file systems, **xml** etc. | Example of RDBMS are **MySQL**, **postgrad**, **sql server**, **oracle** etc. |

**Types of data models are:**

- Entity relationship model
- Relational model
- Hierarchical model
- Network model
- Object oriented model
- Object relational model

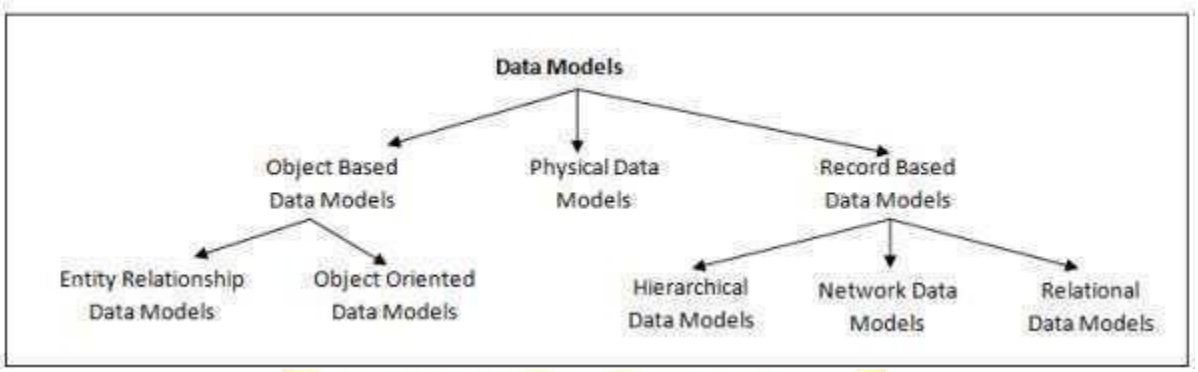



**Fig. 1.2**

**Entity relationship model**

**Entity**

An entity can be real world object, either animate or inanimate, that can be easily identifiable. For example, in a school database, students, teachers, classes and courses offered can be considered as entities. Entities are represented by means of rectangles.

**Relationship**

A relationship is an association among several entities. For example, an employee works at a department, a student enrolls in a course. Here, **works at** and **enrolls** are called relationship. Relationships are represented by diamond-shaped box.

**Attributes**

Entities are represented by means of their properties, called **attributes**. All attributes have values. For example, a student entity may have name, class, and age as attributes. Attributes are represented by means of ellipses. Every ellipse represents one attribute and is directly connected to its entity (rectangle).

- **Strong Entity**: Entities having its own attribute as primary keys are called strong entity. For example, STUDENT has STUDENT_ID as primary key. Hence it is a strong entity.
- **Weak Entity**: Entities which cannot form their own attribute as primary key are known weak entities. These entities will derive their primary keys from the combination of its attribute and primary key from its mapping entity.

- **Simple Attribute**

These kinds of attributes have values which cannot be divided further. For example, STUDENT_ID attribute which cannot be divided further. Passport Number is unique value and it cannot be divided.

- **Composite Attribute**

This kind of attribute can be divided further to more than one simple attribute. For example, address of a person. Here address can be further divided as Door#, street, city, state and pin which are simple attributes.

- **Derived Attribute**

Derived attributes are the one whose value can be obtained from other attributes of entities in the database. For example, Age of a person can be obtained from date of birth and current date. Average salary, annual salary, total marks of a student etc. are few examples of derived attribute.

- **Stored Attribute**

The attribute which gives the value to get the derived attribute are called Stored Attribute. In example above, age is derived using Date of Birth. Hence Date of Birth is a stored attribute.

- **Single Valued Attribute**

These attributes will have only one value. For example, EMPLOYEE_ID, passport#, driving license#, SSN etc have only single value for a person.

- **Multi-Valued Attribute**

These attributes can have more than one value at any point of time. Manager can have more than one employee working for him, a person can have more than one email address, and more than one house etc is the examples.

- **Simple Single Valued Attribute**

This is the combination of above four types of attributes. An attribute can have single value at any point of time, which cannot be divided further. For example, EMPLOYEE_ID – it is single value as well as it cannot be divided further.

- **Simple Multi-Valued Attribute**

Phone number of a person, which is simple as well as he can have multiple phone numbers is an example of this attribute.

- **Composite Single Valued Attribute**

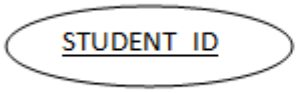
Date of Birth can be a composite single valued attribute. Any person can have only one DOB and it can be further divided into date, month and year attributes.

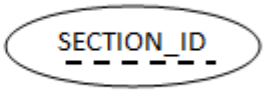
- **Composite Multi-Valued Attribute**

Shop address which is located two different locations can be considered as example of this attribute.

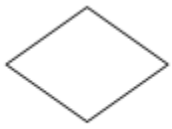
- **Descriptive Attribute**

Attributes of the relationship is called descriptive attribute. For example, employee works for department. Here 'works for' is the relation between employee and department entities. The relation 'works for' can have attribute DATE_OF_JOIN which is a descriptive attribute.

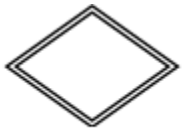



**Relationship**: A diamond shape is used to show the relationship between the entities. A mapping with weak entity is shown using double diamond. Relationship name will be written inside them.



Normal Relations

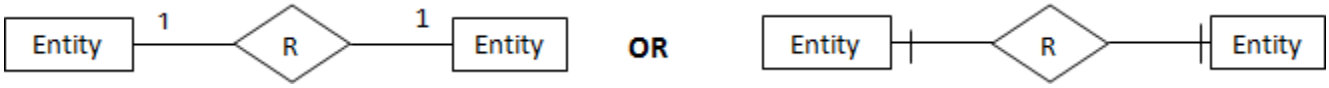


Identifying or weak Relations

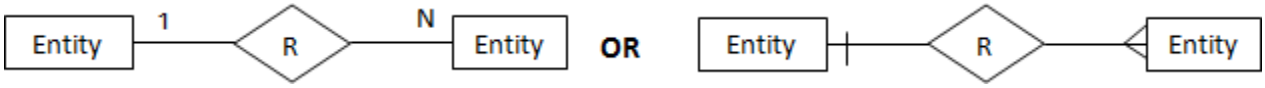
**Cardinality of Relationship**: Different developers use different notation to represent the cardinality of the relationship. Not only for cardinality, but for other objects in ER diagram will have slightly different notations. But main difference is noticed in the cardinality. For not to get confused with many, let us see two types of

notations for each.

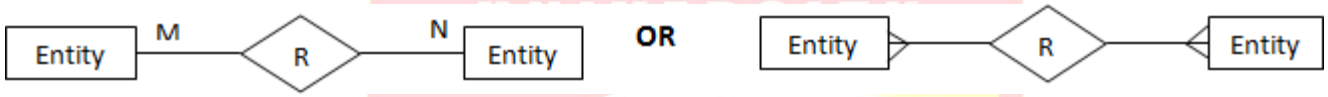
**One-to-one relation: -** A one-to-one relationship is represented by adding '1' near the entities on the line joining the relation. In another type of notation one dash is added to the relationship line at both ends.



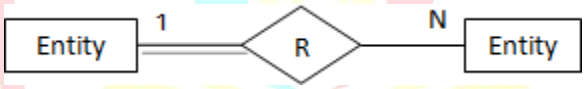

**One-to-Many relation**: A one-to-many relationship is represented by adding '1' near the entity at left hand side of relation and 'N' is written near the entity at right side. Other type of notation will have dash at LHS of relation and three arrow kind of lines at the RHS of relation as shown below.




**Many-to-Many relation**: A one-to-many relationship is represented by adding 'M' near the entity at left hand side of relation and 'N' is written near the entity at right side. Other type of notation will have three arrow kinds of lines at both sides of relation as shown below.




**Participation Constraints**: Total participation constraints are shown by double lines and partial participations are shown as single line.




**Data-model**

In the below diagram, Entities or real world objects are represented in a rectangular box. Their attributes are represented in ovals. Primary keys of entities are underlined. All the entities are mapped using diamonds. This is one of the methods of representing ER model. There are many different forms of representation. More details of this model are described in ER data model article.

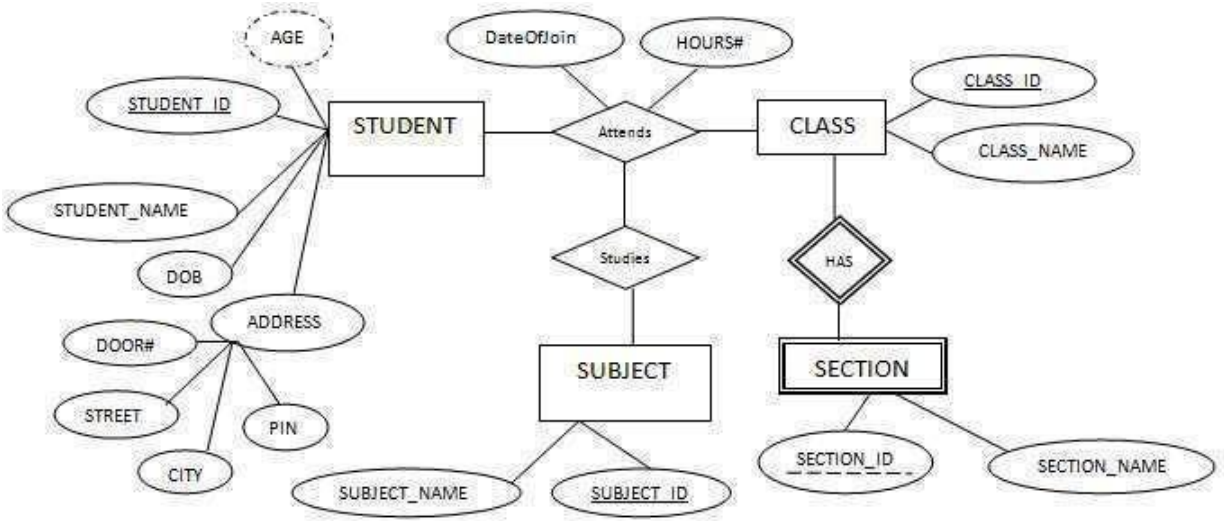


**Fig 1.3**

Basically, ER model is a graphical representation of real world objects with their attributes and relationship. It makes the system easily understandable. This model is considered as a top down approach of designing a requirement.

**Advantages:**

- It makes the requirement simple and easily understandable by representing simple diagrams.
- One can covert ER diagrams into record based data model easily.
- Easy to understand ER diagrams

**Disadvantages:**

- No standard notations are available for E‘ diagram. There is great flexibility in the notation. It’s all depends upon the designer, how he draws it.
- It is meant for high level designs. We cannot simplify for low level design like coding.

**Object Oriented Data Model**

This data model is another method of representing real world objects. It considers each object in the world as objects and isolates it from each other. It groups its related functionalities together and allows inheriting its functionality to other related sub-groups.

Let us consider an Employee database to understand this model better. In this database, we have different types of employees – Engineer, Accountant, Manager, Clark. But all these employees belong to Person group. Person can have different attributes like name, address, age and phone. What do we do if we want to get a person’s address and phone number? We write two separate procedure sp_getAddress and sp_getPhone.

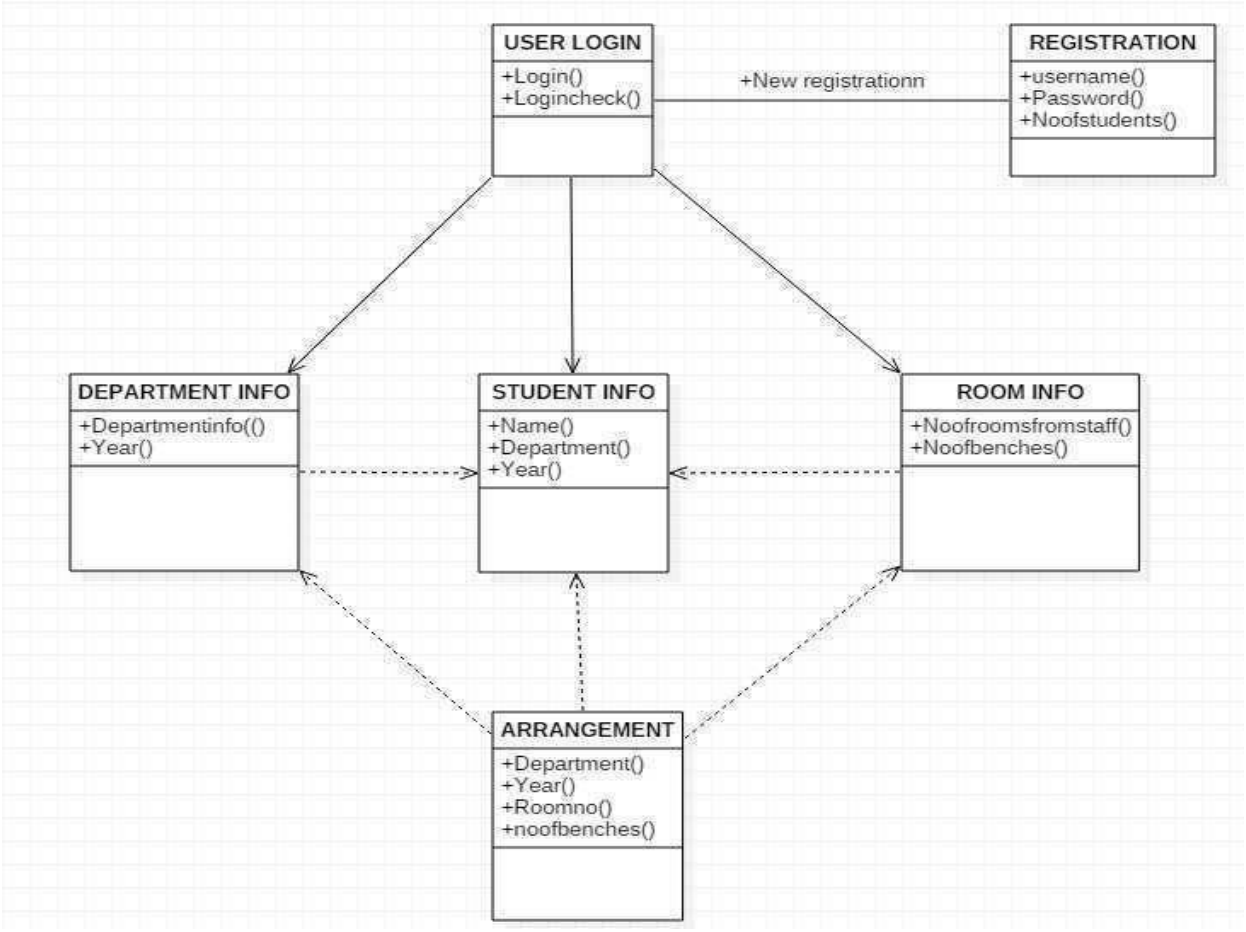



**Fig 1.4**

**Advantages:**

• Because of its inheritance property, we can re-use the attributes and functionalities. It reduces the cost of maintaining the same data multiple times. Also, these information’s are encapsulated and, there is no fear being misused by other objects. If we need any new feature we can easily add new class inherited from parent class and adds new features. Hence it reduces the overhead and maintenance costs.

- Because of the above feature, it becomes more flexible in the case of any changes.
- Codes are re-used because of inheritance.
- Since each class binds its attributes and its functionality, it is same as representing the real-world object. We can see each object as a real entity. Hence it is more understandable.

**Disadvantages:**

- It is not widely developed and complete to use it in the database systems. Hence it is not accepted by the users.
- It is an approach for solving the requirement. It is not a technology. Hence it fails to put it in the database management systems.

Imagine we have to create a database for a company. What are the entities involved in it? Company, its department, its supplier, its employees, different projects of the company etc are the different entities we need to take care of. If we observe each of the entity they have parent –child relationship. We can design them like we do ancestral hierarchy. In our case, Company is the parent and rests of them are its children. Department has employees and project as its children and so on. This type of data modeling is called hierarchical data model.

In this data model, the entities are represented in a hierarchical fashion. Here we identify a parent entity, and its child entity. Again, we drill down to identify next level of child entity and so on. This model can be imagined as folders inside a folder!

In our example above, it is diagrammatically represented as below:

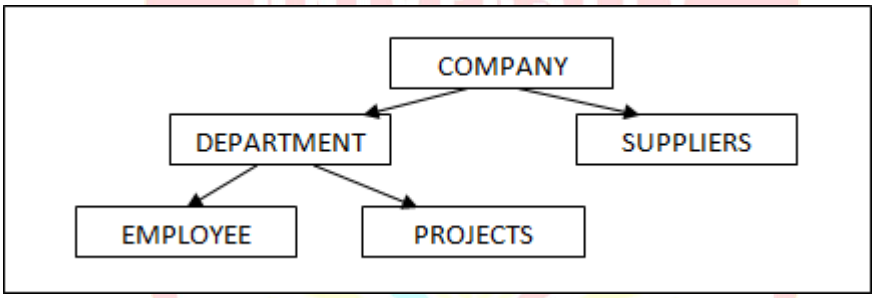



**Fig 1.5**

It can also be imagined as root like structure. This model will have only one main root. It then branches into sub-roots, each of which will branch again. This type of relationship is best defined for 1:N type of relationships. E.g.; One company has multiple departments (1:N), one company has multiple suppliers (1:N),one department has multiple employees (1:N), each department has multiple projects(1:N) . If we have M:N relationships, then we have to duplicate the entities and show it in the diagram. For example, if a project in the company involves multiple departments, then our hierarchical representation changes as below:

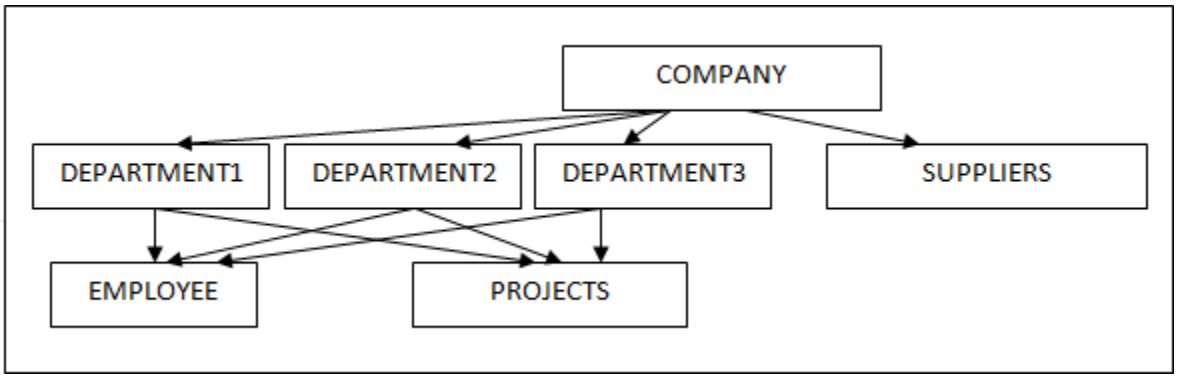



**Fig 1.6**

**Advantages**

It helps to address the issues of flat file data storage. In flat files, data will be scattered and there will not be any proper structuring of the data. This model groups the related data into tables and defines the relationship between the tables, which is not addressed in flat files.

**Disadvantages**

- Redundancy: - When data is stored in a flat file, there might be repetition of same data multiple times and any changes required for the data will need to change in all the places in the flat file. Missing to update at any one place will cause incorrect data. This kind redundancy is solved by hierarchical model to some extent. Since records are grouped under related table, it solves the flat file redundancy issue. But look at the many to many relationship examples given above. In such case, we have to store same project information for more than one department. This is duplication of data and hence a redundancy. So, this model does not reduce the redundancy issue to a significant level.
- As we have seen above, it fails to handle many to many relationships efficiently. It results in redundancy and confusion. It can handle only parent-child kind of relationship.
- If we need to fetch any data in this model, we have to start from the root of the model and traverse through its child till we get the result. In order to perform the traversing, either we should know well in advance the layout of model or we should be very good programmer. Hence fetching through this model becomes bit difficult.
- Imagine company has got some new project details, but it did not assign it to any department yet. In this case, we cannot store project information in the PROJECT table, till company assigns it to some department. That means, in order to enter any child information, its parent information should be already known / entered.



**Network Data Models**

This is the enhanced version of hierarchical data model. It is designed to address the drawbacks of the hierarchical model. It helps to address M: N relationship. This data model is also represented as hierarchical, but this model will not have single parent concept. Any child in the tree can have multiple parents here.

Let us revisit our company example. A company has different projects and departments in the company own those projects. Even suppliers of the company give input for the project. Here Project has multiple parents and each department and supplier have multiple projects. This is represented as shown below. Basically, it forms a network like structure between the entities, hence the name.

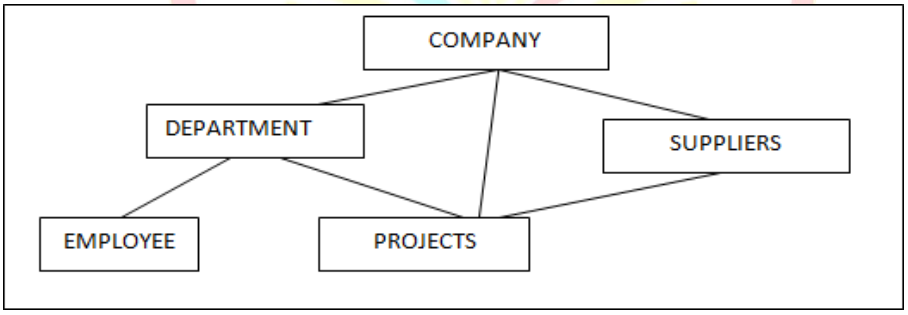



**Fig 1.7**

One more example:

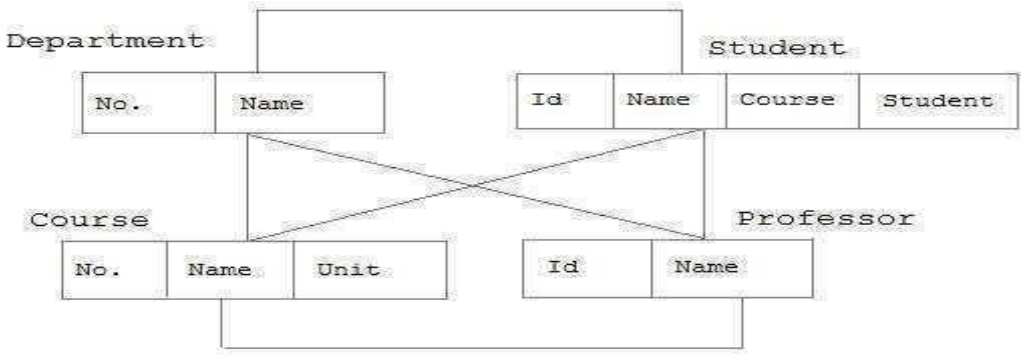



**Fig 1.8**

**Advantages**

- Accessing the records in the tables is easy since it addresses many to many relationships. Because of this kind of relationship, any records can be easily pulled by using any tables. For example, if we want to know the department of project X and if we know SUPPLIER table, we can pull this information. i.e.; SUPPLIER has the information about project X which includes the departments involved in the projects too. Hence makes the accessibility to any data easier, and even any complex data can be retrieved easily and quickly.
- Because of the same feature above, one can easily navigate among the tables and get any data.
- It is designed based on database standards – ANSI/SP ARC.

**Disadvantages**

- If there is any requirement for the changes to the entities, it requires entire changes to the database. There is no independence between any objects. Hence any changes to the any of the object will need changes to the whole model. Hence difficult to manage.
- It would be little difficult to design the relationship between the entities, since all the entities are related in some way. It requires thorough practice and knowledge about the designing.

**Relational Data Models**

This model is designed to overcome the drawbacks of hierarchical and network models. It is designed completely different from those two models. Those models define how they are structured in the database physically and how they are inter-related. But in the relational model, we are least bothered about how they are structured. It purely based on how **the records in each table are related**. It purely isolates physical structure from the logical structure. Logical structure is defining records are grouped and distributed.

Let us try to understand it by an example. Let us consider department and employee from our previous examples above. In this model, we look at employee with its data. When we say an employee what all comes into our mind? His employee id, name, address, age, salary, department that he is working etc. are attributes of employee. That means these details about the employee forms columns in employee table and value set of each employee for these attribute forms a row/record for an employee. Similarly, department has its id, name.

Now, in the employee table, we have column which uniquely identifies each employee – that is employee Id column. This column has unique value and we are able to differentiate each employee from each other by using this column. Such column is called as primary key of the table. Similarly, department table has DEPT_ID as primary key. In the employee table, instead of storing whole information about his department, we have DEPT_ID from department table stored. i.e.; by using the data from the department table, we have established the relation between employee and department tables.

**EMPLOYEE**

| EMP_ID | EMP_NAME | ADDRESS | DEPT_ID |
|---|---|---|---|
| 100 | Joseph | Clinton Town | 10 |
| 101 | Rose | Fraser Town | 20 |
| 102 | Mathew | Lakeside Village | 10 |
| 103 | Stewart | Troy | 30 |
| 104 | William | Holland | 30 |

**DEPARTMENT**

| DEPT_ID | DEPT_NAME |
|---|---|
| 10 | Accounting |
| 20 | Quality |
| 30 | Design |

**Fig 1.9**

A relational data model revolves around 5 important rules.

1. Order of rows / records in the table is not important. For example, displaying the records for Joseph is independent of displaying the records for Rose or Mathew in Employee table. It does not change the meaning or level of them. Each record in the table is independent of other. Similarly, order of columns in the table is not important. That means, the value in each column for a record is independent of other. For example, representing DEPT_ID at the end or at the beginning in the employee table does not have any affect.

2. Each record in the table is unique. That is there is no duplicate record exists in the table. This is achieved by the use of primary key or unique constraint.
3. Each column/attribute will have single value in a row. For example, in Department table, DEPT_NAME column cannot have 'Accounting' and 'Quality' together in a single cell. Both has to be in two different rows as shown above.
4. All attributes should be from same domain. That means each column should have meaningful value. For example, Age column cannot have dates in it. It should contain only valid numbers to represent individual's age. Similarly, name columns should have valid names, Date columns should have proper dates.
5. Table names in the database should be unique. In the database, same schema cannot contain two or more tables with same name. But two tables with different names can have same column names. But same column name is not allowed in the same table.
Examine below table structure for Employee, Department and Project and see if it satisfies relational data model rules.

**Advantages**

- Structural independence: - Any changes to the database structure, does not the way we are accessing the data. For example, Age is added to Employee table. But it does not change the relationship between the other tables nor changes the existing data. Hence it provides the total independence from its structure.
- Simplicity: - This model is designed based on the logical data. It does not consider how data are stored physically in the memory. Hence when the designer designs the database, he concentrates on how he sees the data. This reduces the burden on the designer.
- Because of simplicity and data independence, this kind of data model is easy to maintain and access.
- This model supports structured query language – SQL. Hence it helps the user to retrieve and modify the data in the database. By the use of SQL, user can get any specific information from the database.

**Disadvantages**

Compared to the advantages above, the disadvantages of this model can be ignored.

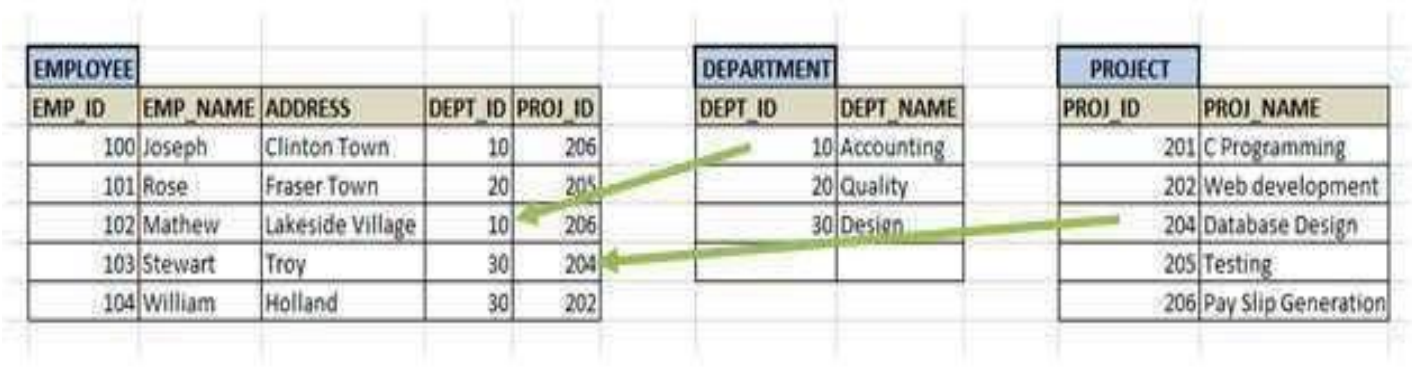
- High hardware cost: - In order to separate the physical data information from the logical data, more powerful system hardware's – memory is required. This makes the cost of database high.
- Sometimes, design will be designed till the minute level, which will lead to complexity in the database.

Observe the table structures above. They are very simple to understand. There is no redundant data as well. It addressed major drawback of earlier data models. This type of data model is called relational data model.
This model is based on the mathematical concepts of set theory. It considers the tables as a two-dimensional table with rows and columns. It is least bothered about the physical storage of structure and data in the memory. It considers only the data and how it can be represented in the form of rows and columns, and the way it can establish the relation between other tables.
A relational data model revolves around 5 important rules.
1. Order of rows / records in the table is not important. For example, displaying the records for Joseph is independent of displaying the records for Rose or Mathew in Employee table. It does not change the meaning or level of them. Each record in the table is independent of other. Similarly, order of columns in the table is not important. That means, the value in each column for a record is independent of other. For example, representing DEPT_ID at the end or at the beginning in the employee table does not have any affect.
2. Each record in the table is unique. That is there is no duplicate record exists in the table. This is achieved by the use of primary key or unique constraint.
3. Each column/attribute will have single value in a row. For example, in Department table, DEPT_NAME column cannot have 'Accounting' and 'Quality' together in a single cell. Both has to be in two different rows as shown above.
4. All attributes should be from same domain. That means each column should have meaningful value. For

example, Age column cannot have dates in it. It should contain only valid numbers to represent individual's age. Similarly, name columns should have valid names, Date columns should have proper dates.

5. Table names in the database should be unique. In the database, same schema cannot contain two or more tables with same name. But two tables with different names can have same column names. But same column name is not allowed in the same table.

Examine below table structure for Employee, Department and Project and see if it satifies relational data model rules.



EMPLOYEE

| EMP_ID | EMP_NAME | ADDRESS | DEPT_ID | PROJ_ID |
|---|---|---|---|---|
| 100 | Joseph | Clinton Town | 10 | 206 |
| 101 | Rose | Fraser Town | 20 | 205 |
| 102 | Mathew | Lakeside Village | 10 | 206 |
| 103 | Stewart | Troy | 30 | 204 |
| 104 | William | Holland | 30 | 202 |

DEPARTMENT

| DEPT_ID | DEPT_NAME |
|---|---|
| 10 | Accounting |
| 20 | Quality |
| 30 | Design |

PROJECT

| PROJ_ID | PROJ_NAME |
|---|---|
| 201 | C Programming |
| 202 | Web development |
| 204 | Database Design |
| 205 | Testing |
| 206 | Pay Slip Generation |

**Fig 1.11**

**Advantages**

- Structural independence: - Any changes to the database structure, does not the way we are accessing the data. For example, Age is added to Employee table. But it does not change the relationship between the other tables nor changes the existing data. Hence it provides the total independence from its structure.
- Simplicity: - This model is designed based on the logical data. It does not consider how data are stored physically in the memory. Hence when the designer designs the database, he concentrates on how he sees the data. This reduces the burden on the designer.
- Because of simplicity and data independence, this kind of data model is easy to maintain and access.
- This model supports structured query language – SQL. Hence it helps the user to retrieve and modify the data in the database. By the use of SQL, user can get any specific information from the database.

**Disadvantages**

Compared to the advantages above, the disadvantages of this model can be ignored.

- High hardware cost: - In order to separate the physical data information from the logical data, more powerful system hardware's – memory is required. This makes the cost of database high.
- Sometimes, design will be designed till the minute level, which will lead to complexity in the database.

Below table provides the comparison among the three models:

| Hierarchical Data Model | Network Data Models | Relational Data Models |
|---|---|---|
| Supports One-Many Relationship | Supports both one to many and Many to Many relationship | Supports both one to many and Many to Many relationship |
| Because of single parent-child relationship, difficult to navigate through the child | It establishes the relationship between most of the objects, hence easy to access compared to hierarchical model | It provides SQL, which makes the access to the data simpler and quicker. |
| Flexibility among the different object is restricted to the child. | Because of the mapping among the sub level tables, flexibility is more | Primary and foreign key constraint makes the flexibility much simpler than other models. |

| | | |
|---|---|---|
| Based on the physical storage details | Based on the physical storage details | Based on the logical data view |

**Relationship**

A relationship defines how two or more entities are inter-related. For example, STUDENT and CLASS entities are related as 'Student X **studies** in a Class Y'. Here 'Studies' defines the relationship between Student and Class. Similarly, Teacher and Subject are related as 'Teacher A **teaches** Subject B'. Here 'teaches' forms the relationship between both Teacher and Subject.

**Degrees of Relationship**

In a relationship two or more number of entities can participate. The number of entities who are part of a particular relationship is called degrees of relationship. If only two entities participate in the mapping, then degree of relation is 2 or binary. If three entities are involved, then degree of relation is 3 or ternary. If more than 3 entities are involved then the degree of relation is called n-degree or n-nary.

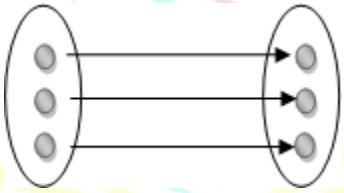
**Cardinality of Relationship**

How many number of instances of one entity is mapped to how many number of instances of another entity is known as cardinality of a relationship. In a 'studies' relationship above, what we observe is only one Student X is studying in on Class Y. i.e.; single instance of entity student mapped to a single instance of entity Class. This means the cardinality between Student and Class is 1:1.

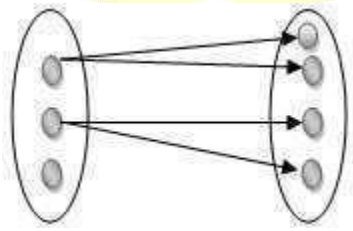
**Based on the cardinality, there are 3 types of relationship.**

- **One-to-One (1:1)**: As we saw in above example, one instance of the entity is mapped to only one instance of another entity.

Consider, HOD of the Department. There is only one HOD in one department. That is there is 1:1 relationship between the entity HOD and Department.

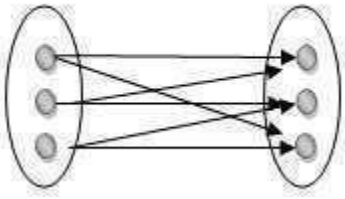


- **One-to-Many (1: M)**: As we can guess now, one to many relationships has one instance of entity related to multiple instances of another entity. One manager manages multiple employees in his department. Here Manager and Employee are entities, and the relationship is one to many. Similarly, one multiple classes is also a 1: M relationship.



- **Many-to-Many (M: N)**: This is a relationship where multiple instances of entities are related to multiple instances of another entity. A relationship between TEACHER and STUDENT is many to many. How? Multiple Teachers teach multiple numbers of Students.

Similarly, above example of 1:1 can be M:N !! Surprised?? Yes, it can be M:N relationship, provided, how we relate these two entities. Multiple Students enroll for multiple classes/courses makes this relationship M:N. The relationship 'studies' and 'enroll' made the difference here. That means, it all depends on the requirement and how we are relating the entities.



**DBA Responsibilities**

- Maintaining all databases required for development, testing, training and production usage
- Installation and configuration of DBMS server software and related products.
- Upgrading and patching/hot-fixing of DBMS server software and related products.
- Migrate database to another server.
- Evaluate DBMS features and DBMS related products.
- Establish and maintain sound backup and recovery policies and procedures.
- Implement and maintain database security (create and maintain logins, users and roles, assign privileges).
- Performance tuning and health monitoring on DBMS, OS and application.
- Plan growth and changes (capacity planning).
- Do general technical troubleshooting and give consultation to development teams.
- Troubleshooting on DBMS and Operating System performance issue
- Documentation of any implementation and changes (database changes, reference data changes and application UI changes etc.)

**Types of DBA**

1. **Administrative DBA –** Work on maintaining the server and keeping it running. Concerned with installation, backups, security, patches, replication, OS configuration and tuning, storage management etc. Things that concern the actual server software.
2. **Development DBA** - works on building queries, stored procedures, etc. that meet business needs. This is the equivalent of the programmer. You primarily write T-SQL or PL-SQL.
3. **Database Architect** – Design schemas. Build tables, FKs, PKs, etc. Work to build a structure that meets the business needs in general. The design is then used by developers and development DBAs to implement the actual application.
4. **Data Warehouse DBA** - responsible for merging data from multiple sources into a data warehouse. May have to design warehouse, but cleans, standardizes, and scrubs data before loading. In SQL Server, this DBA would use SSIS heavily.
5. **OLAP DBA** – Builds multi-dimensional cubes for decision support or OLAP systems. The primary language in SQL Server is MDX, not SQL here
6. **Application DBA-** Application DBAs straddle the fence between the DBMS and the application software and are responsible for ensuring that the application is fully optimized for the database and vice versa. They usually manage all the application components that interact with the database and carry out activities such as application installation and patching, application upgrades, database cloning, building and running data cleanup routines, data load process management, etc.

**Generalization**

- It is a bottom-up approach in which two lower level entities combine to form higher entity. In generalization, the higher-level entity can also combine with other lower level entity to make further higher-level entity.
- Generalization proceeds from the recognition that a number of entity sets share some common features. On the basis of the commonalities, generalization synthesizes these entity sets into a single, higher-level entity set.
- Generalization is used to emphasize the similarities among lower-level entity sets and to hide the differences in the schema.

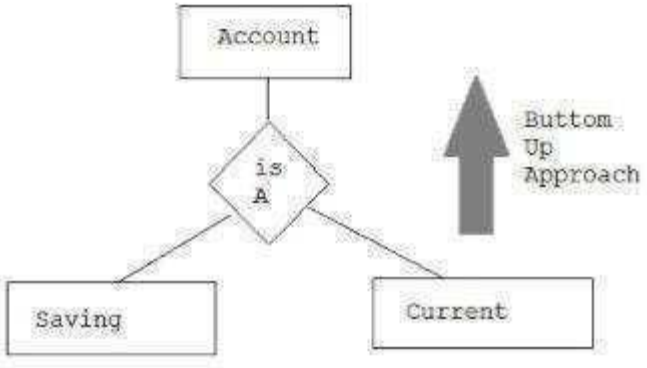



**Fig 1.13**

**Specialization**

- It is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into lower level entity.
- Example: The specialization of student allows us to distinguish among students according to whether they are Ex-Student or Current Student.
- Specialization can be repeatedly applied to refine a design schema.

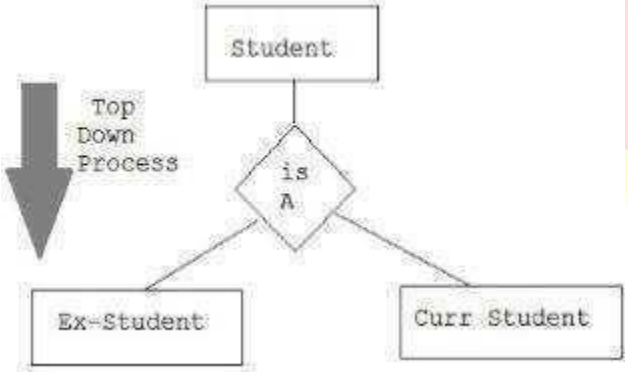



**Fig 1.14**

**Aggregation**

- One limitation of the E-R model is that it cannot express relationships among relationships. To illustrate the need for such a construct, quaternary relationships are used which lead to redundancy in data storage.
- The best way to mode such situations is to use aggregation.
- Aggregation is an abstraction through which relationships are treated as higher-level entities.
- Below is the example of aggregation relation between offer (which is binary relation between center and course) and visitor.

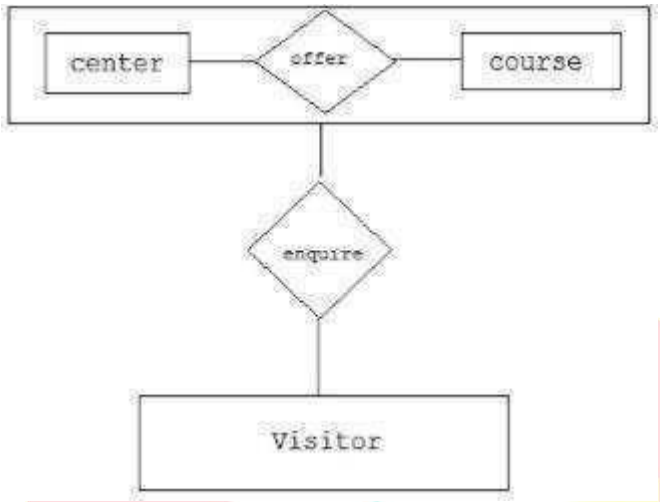



**Fig 1.15**

**Data Independence**

1. **Logical data independence** is the capacity to change the conceptual schema without having to change
2. external schemas or application programs. We may change the conceptual schema to expand the database (byadding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item). In the last case, external schemas that refer only to the remaining data should not be affected. Only the view definition and the mappings need to be changed in a DBMS that supports logical **data independence**. After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before. Changes to constraints can be applied to the conceptual schema without affecting the external schemas or application programs.
3. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files were reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema. Generally, physical **data independence** exists in most databases and file environments where physical details such as the exact location of data on disk, and hardware details of storage encoding, placement, compression, splitting, merging of records, and so on are hidden from the user. Applications remain unaware of these details. On the other hand, logical **data independence** is harder to achieve because it allows structural and constraint changes without affecting application programs a much stricter requirement.

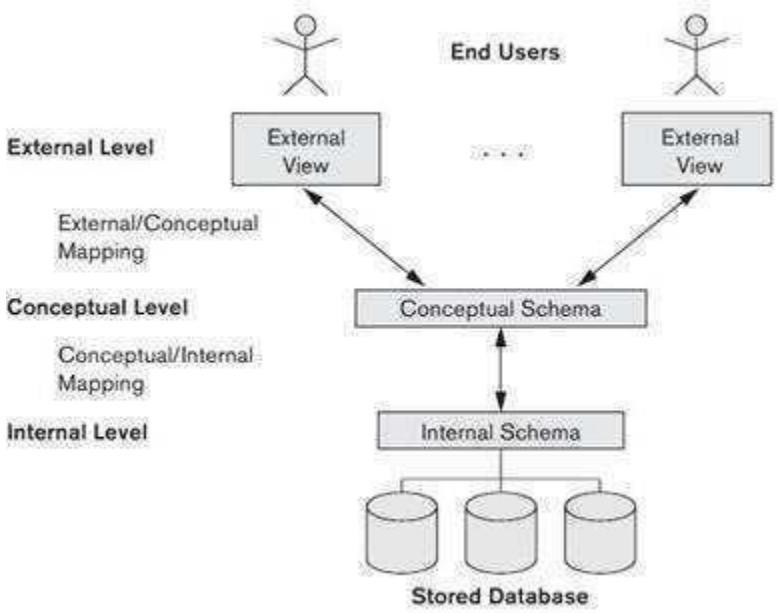



Fig 1.16

**Database Schema**

A database schema is the skeleton structure that represents the logical view of the entire database. It defines how the data is organized and how the relations among them are associated. It formulates all the constraints that are to be applied on the data.

A database schema defines its entities and the relationship among them. It contains a descriptive detail of the database, which can be depicted by means of schema diagrams. It's the database designers who design the schema to help programmers understand the database and make it useful.

A database schema can be divided broadly into two categories –

- **Physical Database Schema** − This schema pertains to the actual storage of data and its form of storage like files, indices, etc. It defines how the data will be stored in a secondary storage.
- **Logical Database Schema** − This schema defines all the logical constraints that need to be applied on the data stored. It defines tables, views, and integrity constraints.

**Database Instance**

It is important that we distinguish these two terms individually. Database schema is the skeleton of database. It is designed when the database doesn't exist at all. Once the database is operational, it is very difficult to make any changes to it. A database schema does not contain any data or information.

A database instance is a state of operational database with data at any given time. It contains a snapshot of the database. Database instances tend to change with time. A DBMS ensures that its every instance (state) is in a valid state, by diligently following all the validations, constraints, and conditions that the database designers have imposed.

**Relational database**

A relational database (RDB) is a collective set of multiple data sets organized by tables, records and columns. RDBs establish a well-defined relationship between database tables. Tables communicate and share information, which facilitates data searchability, organization and reporting.

RDBs use Structured Query Language (SQL), which is a standard user application that provides an easy programming interface for database interaction.

RDB is derived from the mathematical function concept of mapping data sets and was developed by Edgar F. Codd.
RDBs organize data in different ways. Each table is known as a relation, which contains one or more data category columns. Each table record (or row) contains a unique data instance defined for a corresponding column category. One or more data or record characteristics relate to one or many records to form functional dependencies. These are classified as follows:

- One to One: One table record relates to another record in another table.
- One to Many: One table record relates to many records in another table.
- Many to One: More than one table record relates to another table record.
- Many to Many: More than one table record relates to more than one record in another table.
- RDB performs "select", "project" and "join" database operations, where select is used for data retrieval, project identifies data attributes, and join combines relations.

RDBs advantages:
- Easy extendibility, as new data may be added without modifying existing records. This is also known as scalability.
- New technology performance, power and flexibility with multiple data requirement capabilities.
- Data security, which is critical when data sharing is based on privacy. For example, management may share certain data privileges and access and block employees from other data, such as confidential salary or benefit information.

**Database Schema**
A database schema is the skeleton structure that represents the logical view of the entire database. It defines how the data is organized and how the relations among them are associated. It formulates all the constraints that are to be applied on the data.

A database schema defines its entities and the relationship among them. It contains a descriptive detail of the database, which can be depicted by means of schema diagrams. It’s the database designers who design the schema to help programmers understand the database and make it useful.

A database schema can be divided broadly into two categories −

**Physical Database Schema** − This schema pertains to the actual storage of data and its form of storage like files, indices, etc. It defines how the data will be tored in a secondary storage.

**Logical Database Schema** − This schema defines all the logical constraints that need to be applied on the data stored. It defines tables, views, and integrity constraints.

**Keys** are the attributes of the entity, which uniquely identifies the record of the entity. For example, STUDENT_ID identifies individual students, passport#, license # etc.
As we have seen already, there are different types of keys in the database.
**Super Key** is the one or more attributes of the entity, which uniquely identifies the record in the database.
**Candidate Key** is one or more set of keys of the entity. For a person entity, his SSN, passport#, license# etc can be a super key.
**Primary Key** is the candidate key, which will be used to uniquely identify a record by the query. Though a person can be identified using his SSN, passport# or license#, one can choose any one of them as primary key to uniquely identify a person. Rest of them will act as a candidate key.

**Comparison of primary key and foreign key**

- Primary key is unique but foreign key need not be unique.
- Primary key is not null and foreign key can be null, foreign key references a primary key in another table.
- Primary key is used to identify a row; where as foreign key refers to a column or combination of columns.
- Primary key is the parent table and foreign key is a child table.

**Constraints**
Every relation has some conditions that must hold for it to be a valid relation. These conditions are called **Relational Integrity Constraints**. There are three main integrity constraints −

- Key constraints
- Domain constraints
- Referential integrity constraints

**Key Constraints**
There must be at least one minimal subset of attributes in the relation, which can identify a tuple uniquely. This minimal subset of attributes is called **key** for that relation. If there is more than one such minimal subset, these are called **candidate keys**.
Key constraints force that −

- In a relation with a key attribute, no two tuples can have identical values for key attributes.
- A key attribute cannot have NULL values.

Key constraints are also referred to as Entity Constraints.
**Domain Constraints**
Attributes have specific values in real-world scenario. For example, age can only be a positive integer. The same constraints have been tried to employ on the attributes of a relation. Every attribute is bound to have a specific range of values. For example, age cannot be less than zero and telephone numbers cannot contain a digit outside 0-9.
**Referential integrity Constraints**
Referential integrity constraints work on the concept of Foreign Keys. A foreign key is a key attribute of a relation that can be referred in other relation.
Referential integrity constraint states that if a relation refers to a key attribute of a different or same relation, then that key element must exist.
**Intension and Extension-**

**Extension**

The set of tuples appearing at any instant in a relation is called the extension of that relation. In other words, instance of Schema is the extension of a relation. The extension varies with time as instance of schema or the value in the database will change with time.

**Intension**

The intension is the schema of the relation and thus is independent of the time as it does not change once created. So, it is the permanent part of the relation and consists of –

1. **Naming Structure –** Naming Structure includes the name of the relation and the attributes of the relation.
2. **Set of Integrity Constraints –** The Integrity Constraints are divided into Integrity Rule 1 (or entity integrity rule), Integrity Rule 2 (or referential integrity rule), key constraints, domain constraints etc.

For example:

**Employee (EmpNo Number (4) NOT NULL, EName Char(20), Age Number(2), Dept Char(4)**

**Relational Query languages-** Relational query languages use relational algebra to break the user requests and instruct the DBMS to execute the requests. It is the language by which user communicates with the database. These relational query languages can be procedural or non-procedural.

**Procedural query language** will have set of queries instructing the DBMS to perform various transactions in the sequence to meet the user request. For example, *get_CGPA* procedure will have various queries to get the marks of student in each subject, calculate the total marks, and then decide the CGPA based on his total marks. This procedural query language tells the database what is required from the database and how to get them from the database. Relational algebra is a procedural query language.

**Non-procedural queries** will have single query on one or more tables to get result from the database. For example, get the name and address of the student with particular ID will have single query on STUDENT table. Relational Calculus is a non-procedural language which informs what to do with the tables, but doesn't inform how to accomplish.

**SQL**

SQL language is divided into four types of primary language statements: DML, DDL, DCL and TCL. Using these statements, we can define the structure of a database by creating and altering database objects, and we can manipulate data in a table through updates or deletions. We also can control which user can read/write data or manage transactions to create a single unit of work.

The four main categories of SQL statements are as follows:

I. DDL (Data Definition Language)
II. DML (Data Manipulation Language)
III. DCL (Data Control Language)
IV. TCL (Transaction Control Language)

**DDL (Data Definition Language)**

DDL statements are used to alter/modify a database or table structure and schema. These statements handle the design and storage of database objects.

**CREATE** – create a new Table, database, schema

**Example:**

```
create table vendor_master(vencode varchar(5) unique, venname varchar(7) not null);
```

**ALTER** – alter existing table, column description

**Example:**

```
alter table vendor_master add(productprice int(10) check(productprice<10000));
```

**DROP** – delete existing objects from database
**Example:**
drop table vendor_master;

**DML (Data Manipulation Language)**
DML statements affect records in a table. These are basic operations we perform on data such as selecting a few records from a table, inserting new records, deleting unnecessary records, and updating/modifying existing records.
DML statements include the following:
**SELECT** – select records from a table

**Example:**
select * from order_master
**INSERT** – insert new records

**Example:**
insert into order_master values('&oredrno','&odate','&vencode', '&o_status','&deldate');

**UPDATE** – update/Modify existing records

**Example:**
update person set address='United States'where pid=5;

**DELETE** – delete existing records

**Example:**

delete from person where lastname='Kumar';

**DCL (Data Control Language)**
DCL statements control the level of access that users have on database objects.
**GRANT** – allows users to read/write on certain database objects

**Example:**
Grant select, update on dept to ABC;

**REVOKE** – keeps users from read/write permission on database objects

**Example:**
Reovek delete on emp form admin;

TCL (Transaction Control Language)
TCL statements allow you to control and manage transactions to maintain the integrity of data within SQL statements.
**BEGIN Transaction** – opens a transaction
**COMMIT Transaction** – commits a transaction

**Example:**
Set autocommit=0;

**ROLLBACK Transaction** – ROLLBACK a transaction in case of any error.

```
BEGIN TRAN @TransactionName
    INSERT INTO ValueTable VALUES(1), (2);
ROLLBACK TRAN @TransactionName;
```

**Complex queries-**

Complex SQL is the use of SQL queries which go beyond the standard SQL of using the SELECT and WHERE commands. Complex SQL often involves using complex joins and sub-queries, where queries are nested in WHERE clauses. Complex queries frequently involve heavy use of AND clauses and OR clauses. These queries make it possible for perform more accurate searches of a database.

**Various Joins: -**

Joins are operations that "cross reference" the data. That is, tuples from one relation are somehow matched with tuples from another relation to form a third relation. There are several types of joins, but the most basic type is the Cartesian join, sometimes called a Cartesian product or cross product. Other joins, including the natural join, the Equi-join and the theta join, are variations of the Cartesian join in which special rules are applied. Each of these four types of joins in described below.

**Indexing -**

Indexing is a way to optimize performance of a database by minimizing the number of disk accesses required when a query is processed. An index or database index is a data structure which is used to quickly locate and access the data in a database table.

Indexes are created using some database columns.

- The first column is the Search key that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly (Note that the data may or may not be stored in sorted order).
- The second column is the Data Reference which contains a set of pointers holding the address of the disk block where that particular key value can be found.

There are two kinds of indices:

1) **Ordered indices:** Indices are based on a sorted ordering of the values.
2) **Hash indices:** Indices are based on the values being distributed uniformly across a range of buckets. The bucket to which a value is assigned is determined by function called a hash function.

There is no comparison between both the techniques; it depends on the database application on which it is being applied.

- **Access Types**: e.g. value based search, range access, etc.
- **Access Time**: Time to find particular data element or set of elements.
- **Insertion Time**: Time taken to find the appropriate space and insert a new data time.
- **Deletion Time**: Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead**: Additional space required by the index.

Indexing Methods

**Ordered Indices**

The indices are usually sorted so that the searching is faster. The indices which are sorted are known as ordered indices.

- If the search key of any index specifies same order as the sequential order of the file, it is known as primary index or clustering index.

**Note:** The search key of a primary index is usually the primary key, but it is not necessarily so.

▪ If the search key of any index specifies an order different from the sequential order of the file, it is called the secondary index or non-clustering index.

**Clustered Indexing**

Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups.

**Primary Index**

In this case, the data is sorted according to the search key. It induces sequential file organization.
In this case, the primary key of the database table is used to create the index. As primary keys are unique and are stored in sorted manner, the performance of searching operation is quite efficient. The primary index is classified into two types: **Dense Index** and **Sparse Index**.

*(I) Dense Index:*

▪ For every search key value in the data file, there is an index record.
▪ This record contains the search key and also a reference to the first data record with that search key value.

*(II) Sparse Index:*

▪ The index record appears only for a few items in the data file. Each item points to a block as shown.
▪ To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
▪ We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.

**Non-Clustered Indexing**

A non-clustered index just tells us where the data lies, i.e. it gives us a list of virtual pointers or references to the location where the data is actually stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For example, the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here (information on each page of book) is not organized but we have an ordered reference (contents page) to where the data points actually lie.

**Secondary Index**

It is used to optimize query processing and access records in a database with some information other than the usual search key (primary key). In these two levels of indexing are used in order to reduce the mapping size of the first level and in general. Initially, for the first level, a large range of numbers is selected so that the mapping size is small. Further, each range is divided into further sub ranges.
In order for quick memory access, first level is stored in the primary memory. Actual physical location of the data is determined by the second mapping level.

**Trigger**

A **database trigger** is procedural code that is automatically executed in response to certain events on a particular table or view in a database. The trigger is mostly used for maintaining the integrity of the information on the database.

***Schema-level triggers.***

- After Creation
- Before Alter
- After Alter
- Before Drop
- After Drop

- Before Insert

The four main types of triggers are:

1. Row Level Trigger: This gets executed before or after *any column value of a row* changes
2. Column Level Trigger: This gets executed before or after the *specified column* changes
3. For Each Row Type: This trigger gets executed once for each row of the result set affected by an insert/update/delete
4. For Each Statement Type: This trigger gets executed only once for the entire result set, but also fires each time the statement is executed.

**Syntax**

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
  Declaration-statements
BEGIN
  Executable-statements
EXCEPTION
  Exception-handling-statements
END;
```

**Details:**

C‘EATE [O‘ ‘EPLACE] T‘IGGE‘ trigger_name – Creates or replaces an existing trigger with the trigger_name.

{BEFO‘E | AFTE‘ | INSTEAD OF} – This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.

{INSE‘T [O‘] | UPDATE [O‘] | DELETE} – This specifies the DML operation.

[OF col_name] – This specifies the column name that will be updated.

[ON table_name] – This specifies the name of the table associated with the trigger.

[‘EFE‘ENCING OLD AS o NEW AS n] – This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.

[FO‘ EACH ‘OW] – This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.

WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

**Assertion**

An **assertion** is a statement in SQL that ensures a certain condition will always exist in the database. Assertions are like column and table constraints, except that they are specified separately from table definitions. An example of a column constraint is NOT NULL, and an example of a table constraint is a compound foreign key, which, because it's compound, cannot be declared with column constraints.

Assertions are similar to check constraints, but unlike check constraints they are not defined on table or column level but are defined on schema level. (i.e., assertions are database objects of their own right and are not defined within a create table or alter table statement.)

**Relational Algebra**

Relational Algebra is a procedural language used for manipulating relations. The relational model gives the structure for relations so that data can be stored in that format but relational algebra enables us to retrieve information from relations. Some advanced SQL queries requires explicit relational algebra operations, most commonly outer join.

Relations are seen as sets of tuples, which means that no duplicates are allowed. SQL behaves differently in some cases. Remember the SQL keyword distinct. SQL is declarative, which means that you tell the DBMS what you want.

**Set operations**

Relations in relational algebra are seen as sets of tuples, so we can use basic set operations.

Review of concepts and operations from set theory

- Set Element
- No duplicate elements
- No order among the elements
- Subset
- Proper subset (with fewer elements)
- Superset
- Union
- Intersection
- Set Difference
- Cartesian product
- Relational Algebra
- Relational Algebra consists of several groups of operations

**Unary Relational Operations**

SELECT (symbol: s (sigma))

PROJECT (symbol: (pi))

RENAME (symbol: r (rho))

**Relational Algebra Operations from Set Theory**

UNION (U ), INTERSECTION ( ), DIFFERENCE (or MINUS, – )

CARTESIAN PRODUCT ( x )

**Binary Relational Operations**

JOIN (several variations of JOIN exist)

**Additional Relational Operations**

OUTER JOINS, OUTER UNION

AGGREGATE FUNCTIONS

**Unary Relational Operations**

SELECT (symbol: s (sigma))

**Selection**

Selection is another unary operation. It is performed on a single relation and returns a set of tuples with the same schema as the original relation. The selection operation must include a condition, as follows:

A selection of Relation A is a new relation with all of the tuples from Relation A that meet a specified condition. The condition must be a logical comparison of one or more of the attributes of Relation A and their possible values. Each tuple in the original relation must be checked one at a time to see if it meets the condition. If it does, it is included in the result set, if not, it is not included in the result set. The logical comparisons are the same as those used in Boolean conditions in most computer programming languages.
The symbol for selection is $\sigma$, a lowercase Greek letter sigma.
A selection operation is written as $B = \sigma_c(A)$ , where c is the condition.

**Example:**

We wish to find all members of the U.S. Supreme Court born before 1935 in Arizona

Let A = the relation containing data on members of the Supreme Court, as defined above.

$B = \sigma_{((\text{year of birth} < 1935) \wedge (\text{state of birth} = \text{“Arizona”}))}(A)$.

B = {(William, Rehnquist, Arizona, 1924), (Sandra, O'Connor, Arizona, 1930)}

The allowable operators for the logical conditions are the six standard logical comparison operators: equality, inequality, less than, greater than, not less than (equality or greater than), not greater than (equality or less than) Simple logical comparisons may be connected via conjunction, disjunction and negation (*and*, *or*, and *not*).

The symbols for the six logical comparison operators are:

| Comparison Operation | Symbol |
|---|---|
| equal to | = |
| not equal to | ≠ or <> |
| less than | < |
| greater than | > |
| not less than (greater than or equal to) | ≥ or >= |
| not greater than (less than or equal to) | ≤ or <= |

The symbols for the three logical modifiers used to build complex conditions are:

| Modifier | Symbol |
|---|---|
| And | V |
| or | V |
| not | ¬ or ~ |

Some definition or collating sequence must exist to determine how values compare to one another for each of the data types used, just as in computer programming languages. For example, 2 comes before 11 if the values are integers, but “11” comes before “2” if the values are character strings.

Each time we perform a selection operation, the result set will have the same number or fewer tuples. Most often, the result set gets smaller with each selection operation.
σsubject = "database"(Books)
Output − Selects tuples from books where subject is 'database'.

σsubject = "database" and price = "450"(Books)
Output − Selects tuples from books where subject is 'database' and 'price' is 450.

σsubject = "database" and price = "450" or year > "2010"(Books)
Output − Selects tuples from books where subject is 'database' and 'price' is 450 or those books published after 2010.

**Projection**

Projection is another unary operation, performed on a single relation with a result set that is a single relation, defined as follows:
The projection operation returns a result set with all rows from the original relation, but only those attributes that are specified in the projection.

Projection is shown by $\prod$ , the upper case Greek letter Pi. A selection operation is written as $B = \prod_{attributes} (A)$, where $_{attributes}$ is a list of the attributes to be included in the result set.

Example:
We wish to show only the names of the U. S. Supreme Court Justices from the example above.

Let A = the relation containing data on members of the Supreme Court, as defined above.

$B = \prod_{first\ name,\ last\ name} (A)$

B = { (William, Rehnquist), (John, Stevens), (Sandra, O'Connor), (Antonin, Scalia), (Anthony, Kennedy), (David, Souter), (Clarence, Thomas), (Ruth, Ginsburg), (Stephen, Breyer) }

∏subject, author (Books)
Selects and projects columns named as subject and author from the relation Books.

A projection operation will return the same number of tuples, but with a new schema that will include either the same columns or fewer columns. Most often, the number of columns shrinks with each projection.

**Combining Selection and Projection**
Often the selection and projection operations are combined to select certain data from a single relation. We can nest the operation with parenthesis, just like in ordinary algebra.

Example:
We wish to show only the names of the U.S. Supreme Court justices born in Arizona before 1935.

Let A = the relation containing data on members of the U. S. Supreme Court, as defined above.

B = $\prod$ first name, last name ($\sigma$((year of birth < 1935) $\wedge$ (state of birth = "Arizona")) (A) )

B = { (William, Rehnquist), (Sandra, O'Connor) }

When performing both a projection and a selection, which would be more efficient to do first, a projection or a selection? Imagine that we have a database of 100,000 student records. We wish to find the student #, name, and GPA for student # 111-11-1111. Should we find the student's record first and then pull out the name and GPA, or should we pull out the name and GPA for all students, then search that set for the student we are seeking? Usually it is best to do the selection first, thereby limiting the number of tuples, but this is not always the case. Fortunately, most good database management systems have optimizing compilers that will perform the operations in the most efficient way possible.

The most common queries on single tables in modern data base management systems are equivalent to a combination of the selection and projection operations.

**Union**

In simple set theory, the union of Set A and Set B includes all of the elements of Set A and all of the elements of Set B. In relational algebra, the union operation is similar:
The Union of relation A and relation B is a new relation containing all of the tuples contained in either relation A or relation B. Union can only be performed on two relations that have the same schema.
The symbol for union is U In relational algebra we would write something like R3 = R1 U R2.

**Example:**

The set of students majoring in Communications includes all of the Acting majors and all of the Journalism majors.

Let A = the relation with data for all Acting majors
Let J = the relation with data for all Journalism majors
Let C = the relation with data for all Communications majors

C = A U J

The union operation is both commutative and associative.

Commutative law of union: A U B = B U A
Associative law of union: (A U B) U C = A U (B U C)

**Intersection**

Like union, intersection means pretty much the same thing in relational algebra that it does in simple set theory:

The Intersection of relation A and relation B is a new relation containing all of the tuples that are contained in both relation A and relation B. Intersection can only be performed on two relations that have the same schema.
The symbol for intersection is ∩ **In** relational algebra we would write something like R3 = R1 ∩ R2.
Example:
We might want to define the set of all students registered for both Database Management and Linear Algebra.

Let D = the relation with data for all students registered for Database Management
Let L = the relation with data for all students registered for Linear Algebra
Let B = the relation with data for all students registered for both Database Management and Linear Algebra

$B = D \cap L$

The intersection operation is both commutative and associative.

Commutative law of intersection: $A \cap B = B \cap A$
Associative law of intersection: $(A \cap B) \cap C = A \cap (B \cap C)$
Unlike union, however, intersection is not considered a basic operation, but a derived operation, because it can be derived from the basic operations. We will look at difference next, but the derivation looks like this:

$R1 \cap R2 = R1 - (R1 - R2)$

For our purposes, however, it really doesn't matter that intersection is a derived operation.

**Difference**
The difference operation also means pretty much the same thing in relational algebra that it does in simple set theory:
The difference between relation A and relation B is a new relation containing all of the tuples that are contained in relation A but not in relation B. Difference can only be performed on two relations that have the same schema.
The symbol for difference is the same as the minus sign - We would write $R3 = R1 - R2$.

**Example:**
We might want to define the set of all players sitting on the bench during a basketball game.

Let T = the relation with data for all players currently on the team
Let G = the relation with data for all players currently in the game
Let B = the relation with data for all players on the bench; that is, on the team but not playing

$B = T - G$

The difference operation is neither commutative nor associative.

$A - B \neq B - A$
$(A - B) - C \neq A - (B - C)$

**Complement**
Union, Intersection, and difference were *binary* operations; that is, they were performed on two relations with the result being a third relation. Complement is a *unary* operation; it is performed on a single relation to form a new relation, as follows:
The complement of relation A is a relation composed all possible tuples not in A, which have the same schema as A, derived from the same range of values for each attribute of A.
Complement is sometimes shown by using a superscripted *C*, like this: $B = A^C$ .

Example:

Let A = the relation containing data on members of the U.S. Supreme Court, with the schema (first name, last name, state of birth, year of birth).

A = { (William, Rehnquist, Arizona, 1924), (John, Stevens, Illinois, 1920), (Sandra, O'Connor, Arizona, 1930), (Antonin, Scalia, New Jersey, 1936), (Anthony, Kennedy, California, 1936), (David, Souter, Massachusetts, 1939), (Clarence, Thomas, Georgia, 1948), (Ruth, Ginsburg, New York, 1933), (Stephen, Breyer, California, 1938) }

$A^C$ would be the set of all possible tuples with the same schema as A but not in A, that are derived from the same domains for the attributes. It would be very large and include tuples like (William, Rehnquist, Arizona, 1920), (William, Rehnquist, Arizona, 1930), (Ruth, Rehnquist, Illinois, 1924), (William, Stevens, Illinois, 1930), (John, O'Connor, Georgia, 1938), (Clarence, Ginsberg, California, 1936), and so on.

If we perform the complement operation twice, we get back the original relation, just like we would when using the negation operation on numbers in simple arithmetic. $(A^C)^C = A$, which means that if $B = A^C$, then $A = B^C$.

**Joins**

Joins are operations that "cross reference" the data. That is, tuples from one relation are somehow matched with tuples from another relation to form a third relation. There are several types of joins, but the most basic type is the Cartesian join, sometimes called a Cartesian product or cross product. Other joins, including the natural join, the equi-join and the theta join, are variations of the Cartesian join in which special rules are applied. Each of these four types of joins in described below.

**Cartesian Join**

Imagine that we have two sets, one composed of letters and one composed of numbers, as follows:

S1 = { a, b, c, d} and S2 = { 1,2,3}

The cross product of the two sets is a set of ordered pairs, matching each value from S1 with each value from S2.

S1 x S2 = { (a,1), (a2), (a3), (b1), (b2), (b3), (c,1), (c2), (c3), (d1), (d2), (d3) }

The cross product of two sets is also called the Cartesian product, after René Descartes, the French mathematician and philosopher, who among other things, developed the Cartesian Coordinates used in quantifying geometry. In Cartesian coordinates, we have an X-axis and a Y-axis, which means we have a set of X values and a set of Y values. Each point on the Cartesian plane can be referenced by its coordinates, with an X-Y ordered pair: (x,y). The set of all possible X and Y coordinates is the cross product of the set of all X coordinates with the set of all Y coordinates.

In relational algebra, the cross product of two relations is also called the Cartesian Product or Cartesian Join, and is defined as follows:

The Cartesian Join of relation A and relation B is composed by matching each tuple from relation A one at time, with each tuple from relation B one at a time to form a new relation containing tuples with all of the attributes from tuple A and all of the attributes from tuple B. If tuple A has $A_T$ tuples and AA attributes and tuple B has $B_T$ tuples and $B_A$ attributes, then the new relation will have $(A_T * B_T)$ tuples, and $(A_A + B_A)$ attributes.

The symbol for a Cartesian Join is × We would write C = A × B.

**Example:**

We wish to match a group of drivers with a group of trucks.

Let D = the relation with data for all of the drivers
D has the schema D(name, years of service)
D = { (Joe Smith, 12), (Mary Jones, 4), (Sam Wilson, 20), Bob Johnson, 8) }

Let T = the relation with data for all of the trucks
T has the schema D (make, model, year purchased)
T = {(White, Freightliner, 1996), (Ford, Econoline, 2002), (Mack, CHN 602, 2004)}

Let A = the relation with data for all possible assignments of driver to trucks
A = D × T
A has the schema A (name, years of service, make, model, year purchased)

A = { (Joe Smith, 12, White, Freightliner, 1996), (Joe Smith, 12, Ford, Econoline, 2002), (Joe Smith, 12, Mack, CHN 602, 2004 ), (Mary Jones, 4, White, Freightliner, 1996), (Mary Jones, 4, Ford, Econoline, 2002), (Mary Jones, 4, Mack, CHN 602, 2004 ), (Sam Wilson, 20, White, Freightliner, 1996), (Sam Wilson, 20, Ford, Econoline, 2002), (Sam Wilson, 20, Mack, CHN 602, 2004 ), (Bob Johnson, 8, White, Freightliner, 1996), (Bob Johnson, 8, Ford, Econoline, 2002), (Bob Johnson, 8, Mack, CHN 602, 2004 ) }

Although Cartesian Joins form the conceptual basis for all other joins, they are rarely used in actual database management systems because they often result in a relation with a large amount of data. Consider the case of a table with data for 40,000 students, with each row needing 300 bytes of storage space, and a table for 2,000 advisors, with each row needing 200 bytes. The two original tables would need about 12,000,000 and 400,000 bytes of storage space (12 megabytes and 400 kilobytes). The Cartesian join of these two would have 80,000,000 records, each with nearly 600 bytes of storage space for a total of 48,000,000,000 bytes (48 gigabytes).

Another reason that Cartesian joins are not used often is this: What is the value of a Cartesian join? How often do we really need to create such a table?

The other types of joins, which are based on the Cartesian join, are used more often, and are commonly applied in combination with projection and selection operations.

**Natural Join**

A natural join is performed on two relations that share at least one attribute, and is defined as follows:

The natural join of relation A with relation B is a new relation formed by matching all tuples from relation A one by one with all tuples from relation B one by one, but only where the value of the shared attributes are the same. Each shared attribute is only included once in the schema of the result set. A natural join can only be performed on two relations that have at least one shared attribute.

The symbol for a natural join is ⋈ We would write C = A ⋈ B.

**Theta Join**

A theta join is similar to a Cartesian join, except that only those tuples are included that meet a specified condition, as follows:

The theta join of relation A with relation B is a new relation formed by matching all tuples from relation A one by one with all tuples from relation B one by one, but only where the tuples meet a specified condition, called the theta predicate. If the relations share any attributes, then each shared attribute is only included once in the schema of the result set.

The general symbol for a theta join is composite symbol, similar to the symbol for a natural join subscripted with the Greek letter theta: $\bowtie_\theta$ We would write $C = A \bowtie_\theta B$. In practice, the theta is replaced with the actual condition.

**Equi-Join**

An equi-join, which is similar to both a theta joins and a natural join, is defined as follows:
The equi-join of relation A with relation B is a new relation formed by matching all tuples from relation A one by one with all tuples from relation B one by one, but only where the tuples meet a specified condition of equality, called the equi-join predicate. If the relations share any attributes, then each shared attribute is only included once in the schema of the result set.

The symbolism for an equi-join is similar to the symbol for a natural join subscripted with the equal sign: $\bowtie_=$ We would write $C = A \bowtie_= B$. Just as with the theta join, in practice the equal sign is replaced with the actual condition.

The difference between an equi-join and a theta join is that the condition must be one of equality in an equi-join. The difference between an equi-join and a natural join is that the two relations do not need to have a common attribute in an equi-join.

**Example:**

We wish to match groups of people waiting for tables at a restaurant with the available tables, on the condition that the number of people in the group equals the number of seats at the table.

Let W = the relation with data for all the groups waiting for tables
Let T = the relation with data for all of the available tables
Let M = the relation with data assigning groups to tables

$M = W \bowtie_{group.size = table.seats} T$

In this notation the attribute names are shown in their more complex form, *relation.attribute*, so that *group.size* refers to the *size* attribute of the *group* relation, and *table.seats* refers to the *seats* attribute of the *table* relation.

**Summary of Relation Algebra:**

| OPERATION | PURPOSE | NOTATION |
|---|---|---|
| SELECT | Selects all tuples that satisfy the selection condition from a relation R. | σ<selection condition>(‘) |
| PROJECT | Produces a new relation with only some of the attributes of R, and removes duplicate tuples. | π<attribute list>(‘) |

| | | |
|---|---|---|
| THETA JOIN | Produces all combinations of tuples from R and R1 2 that satisfy the join condition. | R1 <join condition> R2 |
| EQUIJOIN | Produces all the combinations of tuples from R1 and R2 that satisfy a join condition with only equality comparisons. | R1 <join condition> R2, OR (<joinattributes 1>), (<join attributes 2>) R2 |
| NATURAL JOIN | Same as EQUIJOIN except that the join attributes of R2 are not included in the resulting relation; if the join attributes have the same names, they do not have to be | R1*<join condition> R2, OR R1*(<join attributes 1>), (<join attributes 2>) R2 OR R1 * R2 |
| UNION | Produces a relation that includes all the tuples in R1 or R2 or both R1 and R2; R1 and R2 must be union-compatible. | R1 U R2 |
| INTERSECTION | Produces a relation that includes all the tuples in both R1 and R2; R1 and R2 must be union-compatible. | ‘1 ∩ ‘2 |
| DIFFERENCE | Produces a relation that includes all the tuples in R1 that are not in R2; R1 and R2 must be union-compatible. | R1 – R2 |
| CARTESIAN PRODUCT | Produces a relation that has the attributes of R1 and R2 and includes as tuples all possible combinations of tuples from R1 and R2. | R1 × R2 |
| DIVISION | Produces a relation R(X) that includes all tuples t[X] in R1(Z) that appear in R1 in combination with every tuple from R(Y), where Z = X U Y. | R1(Z) ÷ R2(Y) |

**Different Types of constraints in DBMS with Example:**

- Domain Constraints
- Tuple Uniqueness Constraints
- Key Constraints
- Single Value Constraints

| SID | Name | Class (semester) | Age |
|---|---|---|---|
| 8001 | Ankit | 1st | 19 |
| 8002 | Srishti | 1st | 18 |
| 8003 | Somvir | 4th | 22 |
| 8004 | Sourabh | 6th | A |

Not Allowed. Because age is an integer attribute

***Tuple Uniqueness Constraints –***

A relation is defined as a set of tuples. All tuples or all rows in a relation must be unique or distinct. Suppose if in a relation, tuple uniqueness constraint is applied, then all the rows of that table must be unique i.e. it does not contain the duplicate values. For example,

| SID | Name | Class (semester) | Age |
|---|---|---|---|
| 8001 | Ankit | 1st | 19 |
| 8002 | Srishti | 2nd | 18 |
| 8003 | Somvir | 4th | 22 |
| 8004 | Sourabh | 6th | 19 |

Not Allowed, beacues all rows must be unique

***Key Constraints –***

Keys are attributes or sets of attributes that uniquely identify an entity within its entity set. An Entity set E can have multiple keys out of which one key will be designated as the primary key. Primary Key must have unique and not null values in the relational table. In an subclass hierarchy, only the root entity set has a key or primary key and that primary key must serve as the key for all entities in the hierarchy.

Example of Key Constraints in a simple relational table –

| SID | Name | Class (semester) | Age |
|---|---|---|---|
| 8001 | Ankit | 1st | 19 |
| 8002 | Srishti | 1st | 18 |
| 8003 | Somvir | 4th | 22 |
| 8004 | Sourabh | 6th | 45 |
| 8002 | Tony | 5th | 23 |

Not allowed as Primary key values must be unique

***Single Value Constraints –***

Single value constraints refer that each attribute of an entity set has a single value. If the value of an attribute is missing in a tuple, then we can fill it with a "null" value. The null value for a attribute will specify that either the value is not known or the value is not applicable. Consider the below example-

| SID | Name | Class (semester) | Age | Driving License Number |
|---|---|---|---|---|
| 8001 | Ankit | 1st | 19 | DL-45698 |
| 8002 | Srishti | 2nd | 18 | DL-45871, DL-89740 |
| 8003 | Somvir | 4th | 22 | DL-95687 |
| 8004 | Sourabh | 6th | 19 | |

Not allowed as a person does not have two driving licenses.

allowed as person may or may not have driving license

***Integrity Rule 1 (Entity Integrity Rule or Constraint)***

The Integrity Rule 1 is also called Entity Integrity Rule or Constraint. This rule states that no attribute of primary key will contain a null value. If a relation has a null value in the primary key attribute, then uniqueness property of the primary key cannot be maintained. Consider the example below-

***Integrity Rule 2 (Referential Integrity Rule or Constraint) –***

The integrity Rule 2 is also called the Referential Integrity Constraints. This rule states that if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2. For example,

Some more Features of Foreign Key –

Let the table in which the foreign key is defined is Foreign Table or details table i.e. Table 1 in above example and the table that defines the primary key and is referenced by the foreign key is master table or primary table i.e. Table 2 in above example. Then the following properties must be hold:

- Records cannot be **inserted** into a F**oreign table** if corresponding records in the master table do not exist.
- Records of the **master table or Primary Table** cannot be **deleted** or **updated** if corresponding records in the detail table actually exist.

**Relational calculus**

**Relational calculus** is a non-procedural query language. It uses mathematical predicate calculus instead of algebra. It provides the description about the query to get the result whereas relational algebra gives the method to get the result. It informs the system what to do with the relation, but does not inform how to perform it.

For example, steps involved in listing all the students who attend 'Database' Course in relational algebra would be

- SELECT the tuples from COU'SE relation with COU'SE_NAME = 'DATABASE'
- PROJECT the COURSE_ID from above result
- SELECT the tuples from STUDENT relation with COUSE_ID resulted above.

In the case of relational calculus, it is described as below:

Get all the details of the students such that each student has course as 'Database'.

See the difference between relational algebra and relational calculus here. From the first one, we are clear on how to query and which relations to be queried. But the second tells what needs to be done to get the students with 'database' course. But it does tell us how we need to proceed to achieve this. 'elational calculus is just the explanative way of telling the query.

There are two types of relational calculus - Tuple Relational Calculus (TRC) and Domain Relational Calculus (DRC).

**Tuple Relational Calculus**

A tuple relational calculus is a non-procedural query language which specifies to select the tuples in a relation. It can select the tuples with range of values or tuples for certain attribute values etc. The resulting relation can have one or more tuples. It is denoted as below:

**{t | P (t)}** or **{t | condition (t)} --** this is also known as expression of relational calculus

Where t is the resulting tuples, P(t) is the condition used to fetch t.

**{t | EMPLOYEE (t) and t.SALARY>10000}** - implies that it selects the tuples from EMPLOYEE relation such that resulting employee tuples will have salary greater than 10000. It is example of selecting a range of values.

**{t | EMPLOYEE (t) AND t.DEPT_ID = 10}** – this select all the tuples of employee name who work for Department 10.

The variable which is used in the condition is called **tuple variable**. In above example t.SALARY and t.DEPT_ID are tuple variables. In the first example above, we have specified the condition t.SALARY >10000. What is the meaning of it? For all the SALARY>10000, display the employees. Here the SALARY is called as bound variable. Any tuple variable with '*For All*' (?) or '*there exists*' (?) condition is **called bound variable**. Here, for any range of values of SALARY greater than 10000, the meaning of the condition remains the same. Bound variables are those ranges of tuple variables whose meaning will not change if the tuple variable is replaced by another tuple variable.

In the second example, we have used DEPT_ID= 10. That means only for DEPT_ID = 10 display employee details. Such variable is called free variable. Any tuple variable without any '*For All*' or '*there exists*' condition is called **Free Variable**. If we change DEPT_ID in this condition to some other variable, say EMP_ID, the meaning of the query changes. For example, if we change EMP_ID = 10, then above it will result in different result set. Free variables are those ranges of tuple variables whose meaning will change if the tuple variable is replaced by another tuple variable.

All the conditions used in the tuple expression are called as well formed formula – WFF. All the conditions in the expression are combined by using logical operators like AND, O' and NOT, and qualifiers like 'For All' (?) or 'there exists' (?). If the tuple variables are all bound variables in a WFF is called **closed WFF**. In an **open WFF**, we will have at least one free variable.

**Domain Relational Calculus**

In contrast to tuple relational calculus, domain relational calculus uses list of attributes to be selected from the relation based on the condition. It is same as TRC, but differs by selecting the attributes rather than selecting whole tuples. It is denoted as below:

$\{< a_1, a_2, a_3, \ldots a_n > \mid P(a_1, a_2, a_3, \ldots a_n)\}$

Where $a_1, a_2, a_3, \ldots a_n$ are attributes of the relation and P is the condition.

For example, select EMP_ID and EMP_NAME of employees who work for department 10

**{<EMP_ID, EMP_NAME> | <EMP_ID, EMP_NAME> ? EMPLOYEE Λ DEPT_ID = 10}**

Get name of the department name that Alex works for.

**{DEPT_NAME |< DEPT_NAME > ? DEPT Λ ? DEPT_ID (<DEPT_ID> ? EMPLOYEE Λ EMP_NAME = Alex)}**

Here green color expression is evaluated to get the department Id of Alex and then it is used to get the department name form DEPT relation.

Let us consider another example where select EMP_ID, EMP_NAME and ADDRESS the employees from the department where Alex works. What will be done here?

**{<EMP_ID, EMP_NAME, ADDRESS, DEPT_ID > | <EMP_ID, EMP_NAME, ADDRESS, DEPT_ID> ? EMPLOYEE Λ ? DEPT_ID (<DEPT_ID> ? EMPLOYEE Λ EMP_NAME = Alex)}**

First, formula is evaluated to get the department ID of Alex (green color), and then all the employees with that department is searched (red color).

Other concepts of TRC like free variable, bound variable, WFF etc. remains same in DRC too. Its only difference is DRC is based on attributes of relation.

## Unit-III

## Normalization

Normalization is a database design technique which organizes tables in a manner that reduces redundancy and dependency of data. It divides larger tables to smaller tables and link them using relationships.

If a database design is not perfect, it may contain anomalies, which are like a bad dream for any database administrator. Managing a database with anomalies is next to impossible.

- **Update anomalies** − If data items are scattered and are not linked to each other properly, then it could lead to strange situations. For example, when we try to update one data item having its copies scattered over several places, a few instances get updated properly while a few others are left with old values. Such instances leave the database in an inconsistent state.
- **Deletion anomalies** − We tried to delete a record, but parts of it was left undeleted because of unawareness, the data is also saved somewhere else.
- **Insert anomalies** − We tried to insert data in a record that does not exist at all.

Normalization is a method to remove all these anomalies and bring the database to a consistent state.

Here are the most commonly used normal forms:

- First normal form(1NF)
- Second normal form(2NF)
- Third normal form(3NF)
- Boyce & Codd normal form (BCNF)

**Normal forms:**

**First normal form (1NF)**

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Example: Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

| emp_id | emp_name | emp_address | emp_mobile |
|---|---|---|---|
| 101 | Harish | New Delhi | 8912312390 |
| 102 | Jonny | Kanpur | 8812121212<br>9900012222 |
| 103 | Ronny | Chennai | 7778881212 |
| 104 | Lokesh | Bangalore | 9990000123<br>8123450987 |

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says “each attribute of a table must have atomic (single) values”, the emp_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

| emp_id | emp_name | emp_address | emp_mobile |
|---|---|---|---|
| 101 | Harish | New Delhi | 8912312390 |
| 102 | Jonny | Kanpur | 8812121212 |
| 102 | Jonny | Kanpur | 9900012222 |
| 103 | Ronny | Chennai | 7778881212 |
| 104 | Lokesh | Bangalore | 9990000123 |
| 104 | Lokesh | Bangalore | 8123450987 |

**Second normal form (2NF)**

A table is said to be in 2NF if both the following conditions hold:

- Table is in 1NF (First normal form)
- No non-prime attribute is dependent on the proper subset of any candidate key of table.

An attribute that is not part of any candidate key is known as non-prime attribute.

**Example**: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

| teacher_id | subject | teacher_age |
|---|---|---|
| 111 | Maths | 38 |
| 111 | Physics | 38 |
| 222 | Biology | 38 |
| 333 | Physics | 40 |
| 333 | Chemistry | 40 |

**Candidate Keys**: {teacher_id,subject}

**Non-prime attribute**: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non-prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this: **teacher_details table:**

| teacher_id | teacher_age |
|---|---|
| 111 | 38 |
| 222 | 38 |
| 333 | 40 |

**teacher_subject table:**

| teacher_id | subject |
|---|---|
| 111 | Maths |
| 111 | Physics |
| 222 | Biology |
| 333 | Physics |
| 333 | Chemistry |

Now the tables comply with Second normal form (2NF).

**Third Normal form (3NF)**

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- **Transitive functional dependency** of non-prime attribute on any super key should be removed.

An attribute that is not part of any **candidate key** is known as non-prime attribute.

In other words, 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency X-> Y at least one of the following conditions hold:

- X is a **super key** of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

**Example**: Suppose a company wants to store the complete address of each employee, they create a table named employee_details that looks like this:

| emp_id | emp_name | emp_zip | emp_state | emp_city | emp_district |
|---|---|---|---|---|---|
| 1001 | John | 282005 | UP | Agra | Dayal Bagh |
| 1002 | Ajeet | 222008 | TN | Chennai | M-City |
| 1006 | Lora | 282007 | TN | Chennai | Urrapakkam |
| 1101 | Lilly | 292008 | UK | Pauri | Bhagwan |
| 1201 | Steve | 222999 | MP | Gwalior | Ratan |

**Super keys**: {emp_id}, {emp_id, emp_name}, {emp_id, emp_name, emp_zip}…so on

**Candidate Keys**: {emp_id}

**Non-prime attributes**: all attributes except emp_id are non-prime as they are not part of any candidate keys.

Here, emp_state, emp_city & emp_district dependent on emp_zip. And, emp_zip is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

**Employee table:**

| emp_id | emp_name | emp_zip |
|---|---|---|
| 1001 | John | 282005 |
| 1002 | Ajeet | 222008 |
| 1006 | Lora | 282007 |
| 1101 | Lilly | 292008 |
| 1201 | Steve | 222999 |

**Employee_zip table:**

| emp_zip | emp_state | emp_city | emp_district |
|---|---|---|---|
| 282005 | UP | Agra | Dayal Bagh |
| 222008 | TN | Chennai | M-City |
| 282007 | TN | Chennai | Urrapakkam |
| 292008 | UK | Pauri | Bhagwan |
| 222999 | MP | Gwalior | Ratan |

**Boyce Codd normal form (BCNF)**

It is an advance version of 3NF that's why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every **functional dependency** X->Y, X should be the super key of the table.

**Example**: Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

| emp_id | emp_nationality | emp_dept | dept_type | dept_no_of_emp |
|---|---|---|---|---|
| 1001 | Indian | Production and planning | D001 | 200 |
| 1001 | Indian | stores | D001 | 250 |
| 1002 | American | design and technical support | D134 | 100 |
| 1002 | American | Purchasing department | D134 | 600 |

**Functional dependencies in the table above**:

emp_id -> emp_nationality

emp_dept -> {dept_type, dept_no_of_emp}

**Candidate key**: {emp_id, emp_dept}

The table is not in BCNF as neither emp_id nor emp_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:

**Emp_nationality table:**

| emp_id | emp_nationality |
|---|---|
| 1001 | Indian |
| 1002 | American |

**Emp_dept table:**

| emp_dept | dept_type | dept_no_of_emp |
|---|---|---|
| Production and planning | D001 | 200 |
| Stores | D001 | 250 |
| design and technical support | D134 | 100 |
| Purchasing department | D134 | 600 |

**Emp_dept_mapping table:**

| emp_id | emp_dept |
|---|---|
| 1001 | Production and planning |
| 1001 | stores |
| 1002 | design and technical support |

Functional dependencies:
emp_id -> emp_nationality
emp_dept -> {dept_type, dept_no_of_emp}
**Candidate keys**:
For first table: emp_id
For second table: emp_dept
For third table: {emp_id, emp_dept}
This is now in BCNF as in both the functional dependencies left side part is a key.

**Functional Dependency:**
Functional dependency (FD) is a set of constraints between two attributes in a relation. Functional dependency says that if two tuples have same values for attributes A1, A2..., An, then those two tuples must have to have same values for attributes B1, B2, ..., Bn.
Functional dependency is represented by an arrow sign (→) that is, X→Y, where X functionally determines Y. The left-hand side attributes determine the values of attributes on the right-hand side.

**Fully Functional Dependency**

A functional dependency P → Q is fully functional dependency if removal of any attribute A from P means that the dependency does not hold any more. or

n a relation R, an attribute Q is said to be fully functional dependent on attribute P, if it is functionally dependent on P and not functionally dependent on any proper subset of P. The dependency P → Q is left reduced, there being no extraneous attributes in the left-hand side of the dependency.

If AD → C, is fully functional dependency, then we cannot remove A or D. i.e. C is fully functional dependent on AD. If we are able to remove A or D, then it is not fully functional dependency.

**EMPLOYEE**

| | | | | FK |
|---|---|---|---|---|
| ENAME | SSN | BDATE | ADDREDD | DNUMBER |

**DEPARTMENT**

| | | FK |
|---|---|---|
| DNAME | DNUMBER | DMGRESSN |

**DEPT_LOCATIONS**

FK

| DNUMBER | DLOCATION |
|---|---|

PK

**PROJECT**

| | | | FK |
|---|---|---|---|
| PNAME | PNUMBER | PLOCATION | DNUM |

PK

**WORKS_ON**

| FK | FK | |
|---|---|---|
| SSN | PNUMBER | HOURS |

{SSN, PNUMBE‘} → HOU‘S is a full FD since neither SSN → HOU‘S
nor PNUMBE‘ → HOU‘S hold

{SSN, PNUMBE‘} → ENAME is not a full FD (it is called a partial
dependency ) since SSN → ENAME also holds

**Partial Functional Dependency –**

A Functional Dependency in which one or more non key attributes are functionally depending on a part of the primary key is called partial functional dependency. or

where the determinant consists of key attributes, but not the entire primary key, and the determined consist of non-key attributes.

For example, Consider a Relation R(A,B,C,D,E) having
FD : AB → CDE where PK is AB.

Then, { A → C; A → D; A → E; B → C; B → D; B → E }
all are Partial Dependencies.

**Transitive Dependency –**

Given a relation R(A,B,C) then dependency like A–>B, B–>C is a transitive dependency, since A–>C is implied .

In the above Fig,
SSN --> DMGRSSN is a transitive FD
{since SSN --> DNUMBER and DNUMBER --> DMGRSSN hold}

SSN --> ENAME is non-transitive FD since there is no set of attributes X
where SSN --> X and X --> ENAME.

**Trivial Functional Dependency:**

- **Trivial** – If a functional dependency (FD) X → Y holds, where Y is a subset of X, then it is called a trivial FD. Trivial FDs always hold.
- **Non-trivial** – If an FD X → Y holds, where Y is not a subset of X, then it is called a non-trivial FD.

**FD Axioms**

Understanding: Functional Dependencies are recognized by analysis of the real world; no automation or algorithm. Finding or recognizing them are the database designer's task.FD manipulations:

**Soundness** -- no incorrect FD's are generated

**Completeness** -- all FD's can be generated

| Axiom Name | Axiom | Example |
|---|---|---|
| **Reflexivity** | if a is set of attributes, b ⊆ a, then a →b | *SSN,Name → SSN* |
| **Augmentation** | if a→ b holds and c is a set of attributes, then ca→cb | *SSN → Name* then *SSN,Phone → Name, Phone* |
| **Transitivity** | if a →b holds and b→c holds, then a→ c holds | *SSN →Zip* and *Zip → City* then *SSN →City* |
| **Union or Additivity *** | if a → b and a → c holds then a→ bc holds | *SSN→Name* and *SSN→Zip* then *SSN→Name,Zip* |
| **Decomposition or Projectivity*** | if a → bc holds then a → b and a → c holds | *SSN→Name,Zip* then *SSN→Name* and *SSN→Zip* |
| **Pseudotransitivity*** | if a → b and cb → d hold then ac → d holds | *Address → Project* and *Project,Date →Amount* then *Address,Date → Amount* |

**Decomposition:**

Decomposition in DBMS is nothing but another name for Normalization.....Normal forms in a database or the concept of Normalization makes a Relation or Table free from insert/update/delete anomalies and saves space by removing duplicate data.

- Decomposition is the process of breaking down in parts or elements.
- It replaces a relation with a collection of smaller relations.
- It breaks the table into multiple tables in a database.
- It should always be lossless, because it confirms that the information in the original relation can be accurately reconstructed based on the decomposed relations.
- If there is no proper decomposition of the relation, then it may lead to problems like loss of information.

**Properties of Decomposition**

1. Lossless Decomposition
2. Dependency Preservation
3. Lack of Data Redundancy

**1. Lossless Decomposition**

- Decomposition must be lossless. It means that the information should not get lost from the relation that is decomposed.
- It gives a guarantee that the join will result in the same relation as it was decomposed.

**Example:**

Let's take 'E' is the Relational Schema, With instance 'e'; is decomposed into: E1, E2, E3,..... En; with instance: e1, e2, e3, . . . . en, If e1 ⋈ e2 ⋈ e3..... ⋈ en, then it is called as **'Lossless Join Decomposition'.**

- In the above example, it means that, if natural joins of all the decomposition give the original relation, then it is said to be lossless join decomposition.

**Example: <Employee_Department> Table**

| Eid | Ename | Age | City | Salary | Deptid | DeptName |
|---|---|---|---|---|---|---|
| E001 | ABC | 29 | Pune | 20000 | D001 | Finance |
| E002 | PQR | 30 | Pune | 30000 | D002 | Production |
| E003 | LMN | 25 | Mumbai | 5000 | D003 | Sales |
| E004 | XYZ | 24 | Mumbai | 4000 | D004 | Marketing |
| E005 | STU | 32 | Bangalore | 25000 | D005 | Human Resource |

- Decompose the above relation into two relations to check whether a decomposition is lossless or lossy.
- Now, we have decomposed the relation that is Employee and Department.

**Relation 1 : <Employee> Table**

| Eid | Ename | Age | City | Salary |
|---|---|---|---|---|
| | | | | |

| | | | | |
|---|---|---|---|---|
| E001 | ABC | 29 | Pune | 20000 |
| E002 | PQR | 30 | Pune | 30000 |
| E003 | LMN | 25 | Mumbai | 5000 |
| E004 | XYZ | 24 | Mumbai | 4000 |
| E005 | STU | 32 | Bangalore | 25000 |

- Employee Schema contains (Eid, Ename, Age, City, Salary).

**Relation 2: <Department> Table**

| Deptid | Eid | DeptName |
|---|---|---|
| D001 | E001 | Finance |
| D002 | E002 | Production |
| D003 | E003 | Sales |
| D004 | E004 | Marketing |
| D005 | E005 | Human Resource |

- Department Schema contains (Deptid, Eid, DeptName).
- So, the above decomposition is a Lossless Join Decomposition, because the two relations contain one common field that is 'Eid' and therefore join is possible.
- Now apply natural join on the decomposed relations.

**Employee ⋈ Department**

| Eid | Ename | Age | City | Salary | Deptid | DeptName |
|---|---|---|---|---|---|---|
| E001 | ABC | 29 | Pune | 20000 | D001 | Finance |
| E002 | PQR | 30 | Pune | 30000 | D002 | Production |
| E003 | LMN | 25 | Mumbai | 5000 | D003 | Sales |
| E004 | XYZ | 24 | Mumbai | 4000 | D004 | Marketing |
| E005 | STU | 32 | Bangalore | 25000 | D005 | Human Resource |

Hence, the decomposition is Lossless Join Decomposition.

- If the <Employee> table contains (Eid, Ename, Age, City, Salary) and <Department> table contains (Deptid and DeptName), then it is not possible to join the two tables or relations, because there is no common column between them. And it becomes Lossy Join Decomposition.

**2. Dependency Preservation**

- Dependency is an important constraint on the database.
- Every dependency must be satisfied by at least one decomposed table.
- If {A → B} holds, then two sets are functional dependent. And, it becomes more useful for checking the dependency easily if both sets in a same relation.
- This decomposition property can only be done by maintaining the functional dependency.
- In this property, it allows to check the updates without computing the natural join of the database structure.

**3. Lack of Data Redundancy**

- Lack of Data Redundancy is also known as a Repetition of Information.
- The proper decomposition should not suffer from any data redundancy.
- The careless decomposition may cause a problem with the data.
- The lack of data redundancy property may be achieved by Normalization process.

**Null values and dangling tuples:**

**Null Values:** The SQL NULL is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank. A field with a NULL value is a field with no value.

**Dangling tuples:** An attribute of the join operator is that it is possible for certain tuples to be "dangling"; that is, they fail to match any tuple of the other relation in the common attributes. Dangling tuples do not have any trace in the result of the join, so the join may not represent the data of the original relations completely.

**Multivalued dependency:**

Multivalued dependency occurs when there are more than one independent multivalued attributes in a table.

, a **multivalued dependency** is a full constraint between two sets of attributes in a relation.

In contrast to the functional dependency, the **multivalued dependency** requires that certain tuples be present in a relation. Therefore, a multivalued dependency is a special case of tuple-generating dependency. The multivalued dependency plays a role in the 4NF database normalization.

A multivalued dependency is a special case of a join dependency, with only two sets of values involved, i.e. it is a binary join dependency.

**For example:** Consider a bike manufacture company, which produces two colors (Black and white) in each model every year.

| bike_model | manuf_year | color |
|---|---|---|
| M1001 | 2007 | Black |
| M1001 | 2007 | Red |
| M2012 | 2008 | Black |
| M2012 | 2008 | Red |
| M2222 | 2009 | Black |
| M2222 | 2009 | Red |

Here columns manuf_year and color are independent of each other and dependent on bike_model. In this case these two columns are said to be multivalued dependent on bike_model. These dependencies can be represented like this:

bike_model ->> manuf_year

bike_model ->> color

**Query optimization:**

Query optimization is a function of many relational database management systems. The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans. Alternative ways of evaluating a given query

- Equivalent expressions
- Different algorithms for each operation

**What is Query Optimization?**

Suppose you were given a chance to visit 15 pre-selected different cities in Europe. The only constraint would be Time. Would you have a plan to visit the cities in any order? Place the 15 cities in different groups based on their proximity to each other. Start with one group and move on to the next group. Important point made over here is that you would have visited the cities in a more organized manner, and the 'Time' constraint mentioned earlier would have been dealt with efficiently.

Query Optimization works in a similar way: There can be many different ways to get an answer from a given query. The result would be same in all scenarios. DBMS strive to process the query in the most efficient way (in terms of 'Time') to produce the answer.

**Steps in a Cost-based query optimization:**

1. Parsing
2. Transformation
3. Implementation
4. Plan selection based on cost estimates

**Query Flow:**

- **Query Parser** – Verify validity of the SQL statement. Translate query into an internal structure using relational calculus.
- **Query Optimizer** – Find the best expression from various different algebraic expressions. Criteria used is 'Cheapness'
- **Code Generator/Interpreter** – Make calls for the Query processor as a result of the work done by the optimizer.
- **Query Processor** – Execute the calls obtained from the code generator.

**There are two main techniques for implementing Query Optimization:**

The **first** technique is based on ***Heuristic Rules*** for ordering the operations in a query execution strategy.

The **second** technique involves the ***systematic estimation*** of the cost of the different execution strategies and choosing the execution plan with the *lowest cost*.

- Semantic query optimization is used with the combination with the heuristic query transformation rules.
- It uses constraints specified on the database schema such as unique attributes and other more complex constraints, in order to modify one query into another query that is more efficient to execute.

**1. Heuristic Rules: -**

- The heuristic rules are used as an optimization technique to modify the internal representation of the query.
- Usually, heuristic rules are used in the form of query tree of query graph data structure, to improve its performance.
- One of the main heuristic rules is to apply SELECT operation before applying the JOIN or other BINARY operations.
- This is because the size of the file resulting from a binary operation such as JOIN is usually a multi-value function of the sizes of the input files.
- The SELECT and PROJECT reduced the size of the file and hence, should be applied before the JOIN or other binary operation.
- Heuristic query optimizer transforms the initial (canonical) query tree into final query tree using equivalence transformation rules.
- This final query tree is efficient to execute.

For example, consider the following relations:

Employee (EName, EID, DOB, EAdd, Sex, ESalary, EDeptNo)

Department (DeptNo, DeptName, DeptMgrID, Mgr_S_date)

DeptLoc (DeptNo, Dept_Loc)

Project (ProjName, ProjNo, ProjLoc, ProjDeptNo)

WorksOn (E-ID, P-No, Hours)

Dependent (E-ID, DependName, Sex, DDOB, Relation)

- Now let us consider the query in the above database to find the name of employees born after 1970 who work on a project named 'Growth'.

SELECT EName

FROM Employee, WorksOn, Project

WHE'E ProjName = 'Growth' AND ProjNo = P-No

AND EID = E-ID AND DOB > '31-12-1970';



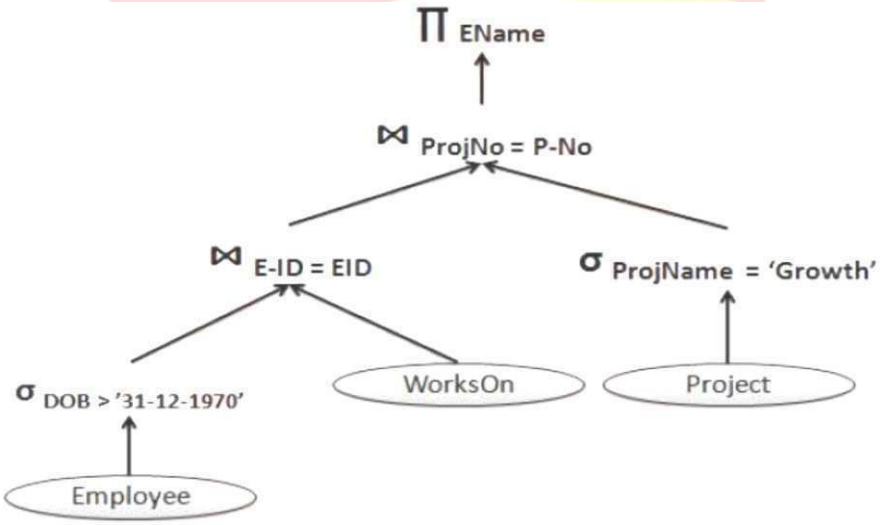



Fig 3.1

**General Transformation Rules: -**

Transformation rules are used by the query optimizer to transform one relational algebra expression into an equivalent expression that is more efficient to execute.

- A relation is considering as equivalent of another relation if two relations have the same set of attributes in a different order but representing the same information.
- These transformation rules are used to restructure the initial (canonical) relational algebra query tree attributes during query decomposition.

**1. Cascade of σ :-**

σ c1 AND c2 AND …AND cn (') = σ c1 (σ c2 (…(σ cn ('))…))

**2. Commutativity of σ :-**

σ C1 (σ C2 (‘)) = σ C2 (σ C1 (‘))

**3. Cascade of Л :-**

Л List1 (Л List2 (…(Л List n (‘))…)) = Л List1 (‘)

**4. Commuting σ with Л :-**

Л A1,A2,A3…An (σ C (‘) ) = σ C (Л A1,A2,A3…An (‘))

**5. Commutativity of ⋈ AND x :-**

R ⋈ c S = S ⋈ c R

R x S = S x R

**6. Commuting σ with ⋈ or x :-**

If all attributes in selection condition c involved only attributes of one of the relation schemas (R).

σ c (‘ ⋈ S) = (σ c (‘) ) ⋈ S

Alternatively, selection condition c can be written as (c1 AND c2) where condition c1 involves only attributes of R and condition c2 involves only attributes of S then:

σ c (‘ ⋈ S) = (σ c1 (‘) ) ⋈ (σ c2 (S) )

**7. Commuting Л with ⋈ or x :-**

- he projection list L = {A1,A2,..An,B1,B2,…Bm}.

-A1…An attribute of ‘ and B1…Bm attributes of S.

-Join condition C involves only attributes in L then :

ЛL ( ‘ ⋈ c S ) = ( ЛA1,…An (‘) ) ⋈c ( ЛB1,…Bm(S) )

**8. Commutativity of SET Operation: -**

- R ∪ S = S ∪ R

- R ß S = S ß R

Minus (R-S) is not commutative.

**9. Associatively of ⋈, x, ß, and ∪ :-**

- If Ø stands for any one of these operation throughout the expression then :

(R Ø S) Ø T = R Ø (S Ø T)

**10. Commutativity of σ with SET Operation: -**

- If Ø stands for any one of three operations (∪,ß,and-) then :

σ c (‘ Ø S) = (σ c (‘)) ∪ (σ c (S))

Л c (‘ Ø S) = (Л c (‘)) ∪ (Лc (S))

**11. The Л operation comute with ∪ :-**

Л L (‘ ∪ S) = (Л L(‘)) ∪ (Л L(S))

**12. Converting a (σ,x) sequence with ∪**

(σ c (‘ x S)) = (‘ ⋈ c S)

**Heuristic Optimization Algorithm: -**

- The Database Management System Use Heuristic Optimization Algorithm that utilizes some of the transformation rules to transform an initial query tree into an optimized and efficient executable query tree.
- The steps of the heuristic optimization algorithm that could be applied during query processing and optimization are:

**Step-1: -**

- Perform SELECT operation to reduce the subsequent processing of the relation:
- Use the transformation rule 1 to break up any SELECT operation with conjunctive condition into a cascade of SELECT operation.

**Step-2: -**

- Perform commutativity of SELECT operation with other operation at the earliest to move each SELECT operation down to query tree.
- Use transformation rules 2, 4, 6 and 10 concerning the commutativity of SELECT with other operation such
- as unary and binary operations and move each select operation as far down the tree as is permitted by the attributes involved in the SELECT condition. Keep selection predicates on the same relation together.

**Step-3: -**

- Combine the Cartesian product with subsequent SELECT operation whose predicates represents the join condition into a JOIN operation.
- Use the transformation rule 12 to combine the Cartesian product operation with subsequent SELECT operation.

**Step-4: -**

- Use the commutativity and associativity of the binary operations.
- Use transformation rules 5 and 9 concerning commutativity and associativity to rearrange the leaf nodes of the tree so that the leaf node with the most restrictive selection operation is executed first in the query tree representation. The most restrictive SELECT operation means:
- Either the one that produce a relation with a fewest tuples or with smallest size.
- The one with the smallest selectivity.

**Step-5: -**

Perform the projection operation as early as possible to reduce the cardinality of the relation and the subsequent processing of the relation, and move the Projection operations as far down the query tree as possible.

o Use transformation rules 3, 4, 7 and 10 concerning the cascading and commuting of projection operations with other binary operation. Break down and move the projection attributes down the tree as far as needed. Keep the projection attributes in the same relation together.

**Step-6: -**

- Compute common expression once.
- Identify sub-tree that represent group of operations that can be executed by a single algorithm.

**Implementing the SELECT Operation**

Simple selection
**S1: Linear search**
**S2: Binary search**
**S3a: Primary index**
**S3b: Hash key**
**S4: Primary index, multiple records**
(For a comparative clause like >, ≤) Use the index to retrieve the bounding = condition, then iterate.
**S5: Clustering index, multiple records**
For an equality clause that's nonkey.
**S6: Secondary index ($B^+$-Tree)**
Complex selection
A conjunctive selection combines subclauses with logical And.
**S7: Conjunctive selection, individual index**
Retrieve records based on the indexed attribute that satisfy its associated condition. Check each one whether it satisfies the remaining conditions.
**S8: Conjunctive selection, composite index**
If the attributes participating the composite index have equality conditions, use it directly.
**S9: Conjunctive selection, intersection of result sets**
Retrieve records satisfying the clause of their indexed attributes, separately, to form result sets. Compute the intersection of these sets.

**Condition selectivity**
The *selectivity* sl is the ratio of tuples satisfying the selection condition to the total tuples in the relation. Low values of sl are usually desirable. More importantly, a DBMS will tend to keep an estimate of the distribution of values among the attributes of the rows of a table, as a *histogram*, to be able to estimate the selectivity of query operations.

**Disjunctive selection conditions**
These have the form $\sigma_{k1\ v\ k2\ v\ \ldots}$(Relation). About the only hope to optimize is to use a separate index on each sub condition $k_i$ and compute the union of their results.

**PROJECT operation**

If the < attribute list > of the P‘OJECT operation π **(‘**) includes a key of **‘**, then the number of tuples in the projection result is equal to the number of tuples in R, but only with the values for the attributes in < attribute list > in each tuple.

If the < attribute list > does not contain a key of R, duplicate tuples must be eliminated. The following methods can be used to eliminate duplication.

- Sorting the result of the operation and then eliminating duplicate tuples.

- Hashing the result of the operation into hash file in memory and check each hashed record against those in the same bucket; if it is a duplicate, it is not inserted.

**CARTESIAN PRODUCT operation:** – It is an extremely expensive operation. Hence, it is important to avoid this operation and to substitute other equivalent operations during optimization.

**Translating SQL Queries into Relation Algebra**

An SQL query is first translated into an equivalent extended relation algebra expression (as a query tree) that is then optimized.

Query block in SQL: the basic unit that can be translated into the algebraic operators and optimized. A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses.

Consider the following SQL query.

*SELECT LNAME, FNAME FROM EMPLOYEE WHERE SALARY > ( SELECT MAX (SALARY) FROM EMPLOYEE WHERE DNO=5);*

We can decompose it into two blocks: Inner block: Outer block:

*(SELECT MAX (SALARY) SELECT LNAME, FNAME FROM EMPLOYEE FROM EMPLOYEE WHERE DNO=5) WHERE SALARY > c*

Then translate to algebra expressions: ∗ Inner block: =

MAX SALA‘Y (σDNO=5 EMPLOY EE) ∗ Outer block: πLNAME, F NAME(σSALA‘Y >c EMPLOY EE)

**Optimization methods:**

**Optimization: Heuristical Processing Strategies**

- Perform selection operations as early as possible.
- Keep predicates on same relation together.
- Combine Cartesian product with subsequent selection whose predicate represents join condition into a join operation.
- Use associativity of binary operations to rearrange leaf nodes so leaf nodes with most restrictive selection operations executed first.
- Perform projection as early as possible.
- Keep projection attributes on same relation together.
- Compute common expressions once.
    - If common expression appears more than once, and result not too large, store result and reuse it when required.
    - Useful when querying views, as same expression is used to construct view each time.

One of the main heuristic rules is to apply SELECT and PROJECT operations *before* applying the JOIN or other binary operations. This is because the size of the file resulting from a binary operation—such as JOIN—is usually a multiplicative function of the sizes of the input files. The SELECT and PROJECT operations reduce the size of a file and hence should be applied *before* a join or other binary operation.

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
    - Perform selection early (reduces the number of tuples)
    - Perform projection early (reduces the number of attributes)

- Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.

Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Pcustomer_name((sbranch_city = “Brooklyn” (branch) account) depositor)

1) When we compute (sbranch_city = “Brooklyn” (branch) account)
we obtain a relation whose schema is:
(branch_name, branch_city, assets, account_number, balance)
2) Push projections using equivalence rules; eliminate unneeded attributes from intermediate results to get: Pcustomer_name ((Paccount_number ((sbranch_city = “Brooklyn” (branch) account)) depositor)
3) Performing the projection as early as possible reduces the size of the relation to be joined.

**Optimization: Cost Estimation for RA Operations:**

- Many different ways of implementing RA operations.
- Aim of QO is to choose most efficient one.
- Use formulae that estimate costs for a number of options, and select one with lowest cost.
- Consider only cost of disk access, which is usually dominant cost in QP.
- Many estimates are based on cardinality of the relation, so need to be able to estimate this.

**Unit-IV**

**Transaction System**

**Transaction Processing Concepts:**

A transaction is a unit of program execution that accesses and possibly updates various data items.

- A transaction must see a consistent database.
- During transaction execution, the database may be inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further.

Let's take an example of a simple transaction. Suppose a bank employee transfers Rs 700 from A's account to B's account. This very simple and small transaction involves several low-level tasks.

**A's Account**

Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 700
A.balance = New_Balance
Close_Account(A)

**B's Account**

Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 700
B.balance = New_Balance
Close_Account(B)

**ACID Properties:**

A transaction is a very small unit of a program and it may contain several low-level tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

- **Atomicity** − This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency** − The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.
- **Durability** − The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action.
- **Isolation** − In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

**Example of Fund Transfer**

Transaction to transfer $50 from account A to account B:

- read(A)
- A := A – 50
- write(A)
- read(B)
- B := B + 50
- write(B)

- **Consistency requirement –** the sum of A and B is unchanged by the execution of the transaction.
- **Atomicity requirement —** if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- **Durability requirement —** once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.
- **Isolation requirement —** if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum A + B will be less than it should be). Can be ensured trivially by running transactions serially, that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.

**States of Transactions:**

A transaction in a database can be in one of the following states –

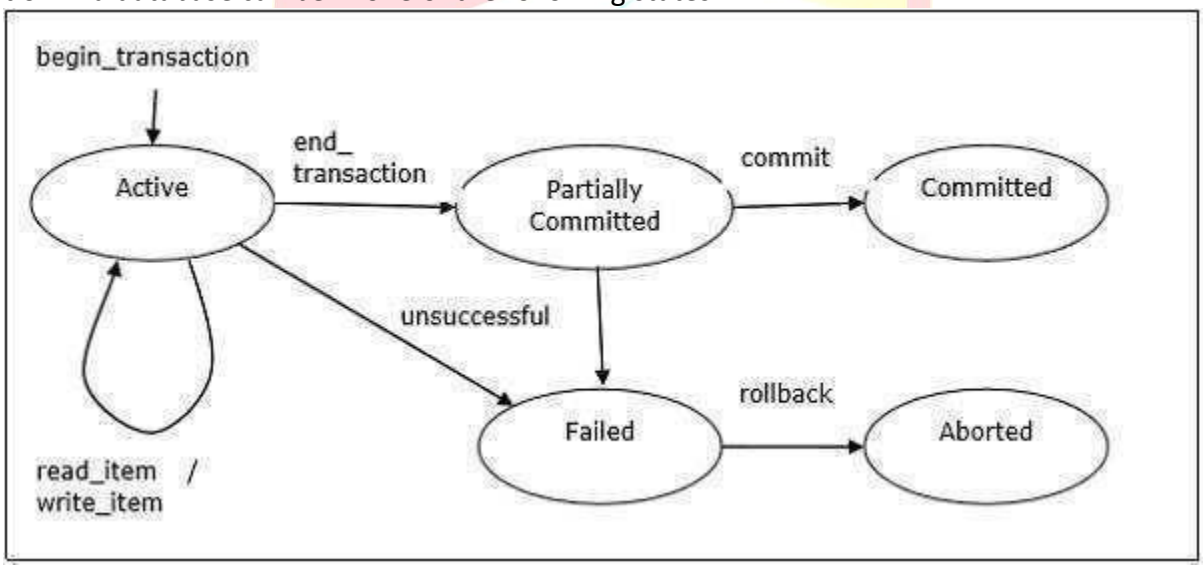



**Fig-4.1**

- **Active –** in this state, the transaction is being executed. This is the initial state of every transaction.
- **Partially Committed –** When a transaction executes its final operation, it is said to be in a partially committed state.
- **Failed –** A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
- **Aborted –** If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts
  - Re-start the transaction
  - Kill the transaction

- **Committed –** If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

**Serializabilty in Database-**

1) A schedule is called as a serial schedule when the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction.
2) Formally, a schedule S is serial if, for every transaction T participating in the schedule, all the operations of T is executed consecutively in the schedule; otherwise, the schedule is called no serial.

**Testing serializability**

When designing concurrency control schemes, we must show that schedules generated by the scheme are serializable. To do that, we must first understand how to determine, given a particular schedule *S*, whether the schedule is serializable.

We now present a simple and efficient method for determining conflict serializ- ability of a schedule. Consider a schedule *S*. We construct a directed graph, called a precedence graph, from *S*. This graph consists of a pair $G = (V, E)$, where *V* is a set of vertices and *E* is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $Ti \rightarrow Tj$ for which
one of three conditions hold:

1. *Ti* executes write(*Q*) before *Tj* executes read(*Q*).
2. *Ti* executes read(*Q*) before *Tj* executes write(*Q*).
3. *Ti* executes write(*Q*) before *Tj* executes write(*Q*).

**Serializable schedules**

To process transactions concurrently, the database server must execute some component statements of one transaction, then some from other transactions, before continuing to process further operations from the first. The order in which the component operations of the various transactions are interleaved is called the schedule. Applying transactions concurrently in this manner can result in many possible outcomes, including the three particular inconsistencies described in the previous section. Sometimes, the final state of the database also could have been achieved had the transactions been executed sequentially, meaning that one transaction was always completed in its entirety before the next was started. A schedule is called serializable whenever executing the transactions sequentially, in some order, could have left the database in the same state as the actual schedule.

Serializabilty is the commonly accepted criterion for correctness. A serializable schedule is accepted as correct because the database is not influenced by the concurrent execution of the transactions.

The isolation level affects a transaction's Serializabilty. At isolation level 3, all schedules are serializable. The default setting is 0.

**Serializable means that concurrency has added no effect**

Even when transactions are executed sequentially, the final state of the database can depend upon the order in which these transactions are executed. For example, if one transaction sets a particular cell to the value 5 and other sets it to the number 6, then the final value of the cell is determined by which transaction executes last.

Knowing a schedule is serializable does not settle which order transactions would best be executed, but rather states that concurrency has added no effect. Outcomes which may be achieved by executing the set of transactions sequentially in some order are all assumed correct.

**Un- serializable schedules introduce inconsistencies**

The inconsistencies introduced in typical types of inconsistency are typical of the types of problems that appear when the schedule is not serializable. In each case, the inconsistency appeared because of the way the statements were interleaved; the result produced would not be possible if all transactions were executed sequentially. For example, a dirty read can only occur if one transaction can select rows while another transaction is in the middle of inserting or updating data in the same row.

**Conflict Serializable:** A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.

**Conflicting operations:** Two operations are said to be conflicting if all conditions satisfy:

- They belong to different transaction
- They operation on same data item
- At Least one of them is a write operation

Example: –

- **Conflicting** operations pair ($R_1$ (A), $W_2$ (A)) because they belong to two different transactions on same data item A and one of them is write operation.
- Similarly, ($W_1$ (A), $W_2$ (A)) and ($W_1$ (A), $R_2$ (A)) pairs are also **conflicting**.
- On the other hand, ($R_1$ (A), $W_2$ (B)) pair is **non-conflicting** because they operate on different data item.
- Similarly, (($W_1$ (A), $W_2$ (B)) pair is **non-conflicting.**

**View Serializable Schedule-**

A schedule is view serializable if it is view equivalent to any serial schedule.

**How to test if two schedules are View Equal or not?**

Two schedules S1 and S2 are said to be view equal if following below conditions are satisfied:

**1) Initial Read**

If a transaction T1 reading data item A from initial database in S1 then in S2 also T1 should read A from initial database.

| **T1** | **T2** | **T3** |
|---|---|---|
| | R(A) | |
| W(A) | | |
| | | R(A) |
| | R(B) | |

Transaction T2 is reading A form initial database.

**2)Updated Read**

If Ti is reading A which is updated by Tj in S1 then in S2 also Ti should read A which is updated by Tj.

| **T1** | **T2** | **T3** | **T1** | **T2** | **T3** |
|---|---|---|---|---|---|
| W(A) | | | W(A) | | |
| | W(A) | | | | R(A) |
| | | R(A) | | W(A) | |

Above two schedules are not view equal as in S1:T3 is reading A updated by T2, in S2 T3 is reading A updated by T1.

**3) Final Write operation**

if a transaction T1 updated A at last in S1, then in S2 also T1 should perform final write operations.

| **T1** | **T2** | **T1** | **T2** |
|---|---|---|---|
| R(A) | | R(A) | |
| | W(A) | W(A) | |
| W(A) | | | W(A) |

Above two schedules is not view as Final write operation in S1 is done by T1 while in S2 done by T2.

**Equivalence Schedules**

An equivalence schedule can be of the following types −

**Result Equivalence**

If two schedules produce the same result after execution, they are said to be result equivalent. They may yield the same result for some value and different results for another set of values. That's why this equivalence is not generally considered significant.

**View Equivalence**

Two schedules would be view equivalence if the transactions in both the schedules perform similar actions in a similar manner.

For example −

- If T reads the initial data in S1, then it also reads the initial data in S2.
- If T reads the value written by J in S1, then it also reads the value written by J in S2.
- If T performs the final write on the data value in S1, then it also performs the final write on the data value in S2.

**Conflict Equivalence**

Two schedules would be conflicting if they have the following properties −

- Both belong to separate transactions.
- Both accesses the same data item.
- At least one of them is "write" operation.

Two schedules having multiple transactions with conflicting operations are said to be conflict equivalent if and only if −

- Both the schedules contain the same set of Transactions.
- The order of conflicting pairs of operation is maintained in both the schedules.

**Recoverability: -**

DBMS is a highly complex system with hundreds of transactions being executed every second. The durability and robustness of a DBMS depends on its complex architecture and its underlying hardware and system software. If it fails or crashes amid transactions, it is expected that the system would follow some sort of algorithm or techniques to recover lost data.

**Failure Classification**

To see where the problem has occurred, we generalize a failure into various categories, as follows −

**Transaction failure**

A transaction has to abort when it fails to execute or when it reaches a point from where it can't go any further. This is called transaction failure where only a few transactions or processes are hurt.

'easons for a transaction failure could be −

- **Logical errors –** Where a transaction cannot complete because it has some code error or any internal error condition.
- **System errors –** Where the database system itself terminates an active transaction because the DBMS is not able to execute it, or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability, the system aborts an active transaction.

**System Crash**

There are problems − external to the system – that may cause the system to stop abruptly and cause the system to crash. For example, interruptions in power supply may cause the failure of underlying hardware or software failure.

Examples may include operating system errors.

**Disk Failure**

In early days of technology evolution, it was a common problem where hard-disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or a part of disk storage.

**Recovery from transaction failures**

When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.

- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

- Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
- Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.

**Log-based Recovery**

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe. Log-based recovery works as follows −

- The log file is kept on a stable storage media.
- When a transaction enters the system and starts execution, it writes a log about it.

  $<T_n$, Start>
  - When the transaction modifies an item X, it writes logs as follows −

  $<T_n, X, V_1, V_2>$

  It reads $T_n$ has changed the value of X, from $V_1$ to $V_2$.
  - When the transaction finishes, it logs −

  $<T_n$, commit>

  The database can be modified using two approaches −
- **Deferred database modification** − All logs are written on to the stable storage and the database is updated when a transaction commits.
- **Immediate database modification** − each log follows an actual database modification. That is, the database is modified immediately after every operation.

**Checkpoint**

Keeping and maintaining logs in real time and in real environment may fill out all the memory space available in the system. As time passes, the log file may grow too big to be handled at all. Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in consistent state, and all the transactions were committed.

**Recovery**

When a system with concurrent transactions crashes and recovers, it behaves in the following manner −

- The recovery system reads the logs backwards from the end to the last checkpoint.
- It maintains two lists, an undo-list and a redo-list.
- If the recovery system sees a log with $<T_n$, Start> and $<T_n$, Commit> or just $<T_n$, Commit>, it puts the transaction in the redo-list.
- If the recovery system sees a log with $<T_n$, Start> but no commit or abort log found, it puts the transaction in undo-list.

All the transactions in the undo-list are then undone and their logs are removed. All the transactions in the redo-list and their previous logs are removed and then redone before saving their logs.

**Deadlock handling**

In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.

For example, assume a set of transactions $\{T_0, T_1, T_2, ...,T_n\}$. $T_0$ needs a resource X to complete its task. Resource X is held by $T_1$, and $T_1$ is waiting for a resource Y, which is held by $T_2$. $T_2$ is waiting for resource Z, which is held by $T_0$. Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

**Deadlock Prevention**

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed. There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

**Wait-Die Scheme**

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$ − that is $T_i$, which is requesting a conflicting lock, is older than $T_j$ − then $T_i$ is allowed to wait until the data-item is available.
- If $TS(T_i) > TS(t_j)$ − that is $T_i$ is younger than $T_j$ − then $T_i$ dies. $T_i$ is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.

**Wound-Wait Scheme**

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur −

- If $TS(T_i) < TS(T_j)$, then $T_i$ forces $T_j$ to be rolled back – that is $T_i$ wounds $T_j$. $T_j$ is restarted later with a random delay but with the same timestamp.
- If $TS(T_i) > TS(T_j)$, then $T_i$ is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

**Problem of starvation**

Whenever transactions are rolled back, it is important to ensure that there is no starvation that is, no transaction gets rolled back repeatedly and is never allowed to make progress.

Both the wound-wait and the wait-die schemes avoid starvation. At any time, there is a transaction with the smallest timestamp. This transaction cannot be required to roll back in either scheme. Since timestamps always increase, and since transactions are not assigned new timestamps when they are rolled back, a transaction that is rolled back will eventually have the smallest timestamp. Thus it will not be rolled back again.

**Timeout-Based Schemes**

Another simple approach to deadlock handling is based on lock timeouts. In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts. If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed. This scheme falls somewhere between deadlock prevention, where a deadlock will never occur and deadlock detection and recovery.

**Uses of Timeout-Based Schemes**

The timeout scheme is particularly easy to implement, and works well if transactions· are short, and if long waits are likely to be due to deadlocks.

**Limitations**

• It is hard to decide how long a transaction must wait before timing out. Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources.

• Starvation is also a possibility with this scheme.

Hence the timeout-based scheme has limited applicability.

**Concurrency Control Techniques:**

In a multiprogramming environment where multiple transactions can be executed simultaneously, it is highly important to control the concurrency of transactions. We have concurrency control protocols to ensure atomicity, isolation, and serializability of concurrent transactions. Concurrency control protocols can be broadly divided into two categories −

- Lock based protocols
- Time stamp based protocols

**Lock-based Protocols**

Database systems equipped with lock-based protocols use a mechanism by which any transaction cannot read or write data until it acquires an appropriate lock on it. Locks are of two kinds −

- **Binary Locks –** A lock on a data item can be in two states; it is either locked or unlocked.
- **Shared/exclusive –** This type of locking mechanism differentiates the locks based on their uses. If a lock is acquired on a data item to perform a write operation, it is an exclusive lock. Allowing more than one transaction to write on the same data item would lead the database into an inconsistent state. Read locks are shared because no data value is being changed.

**There are four types of lock protocols available (Locking Technique) –**

**Simplistic Lock Protocol**

Simplistic lock-based protocols allow transactions to obtain a lock on every object before a 'write' operation is performed. Transactions may unlock the data item after completing the ‘write’ operation**.**

**Pre-claiming Lock Protocol**

Pre-claiming protocols evaluate their operations and create a list of data items on which they need locks. Before initiating an execution, the transaction requests the system for all the locks it needs beforehand. If all the locks are granted, the transaction executes and releases all the locks when all its operations are over. If all the locks are not granted, the transaction rolls back and waits until all the locks are granted.

**Two-Phase Locking 2PL**

This locking protocol divides the execution phase of a transaction into three parts. In the first part, when the transaction starts executing, it seeks permission for the locks it requires. The second part is where the transaction acquires all the locks. As soon as the transaction releases its first lock, the third phase starts. In this phase, the transaction cannot demand any new locks; it only releases the acquired locks.

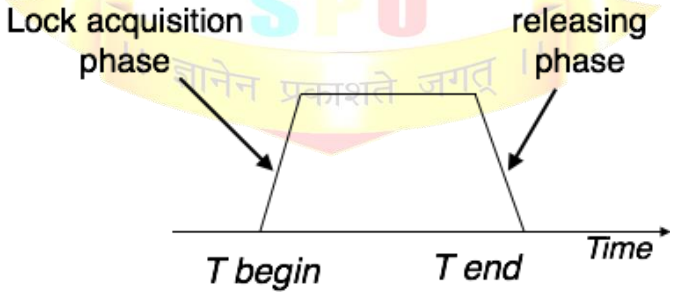



**Fig-4.4**

Two-phase locking has two phases, one is growing, where all the locks are being acquired by the transaction; and the second phase is shrinking, where the locks held by the transaction are being released.
To claim an exclusive (write) lock, a transaction must first acquire a shared (read) lock and then upgrade it to an exclusive lock.

**Strict Two-Phase Locking**

The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.

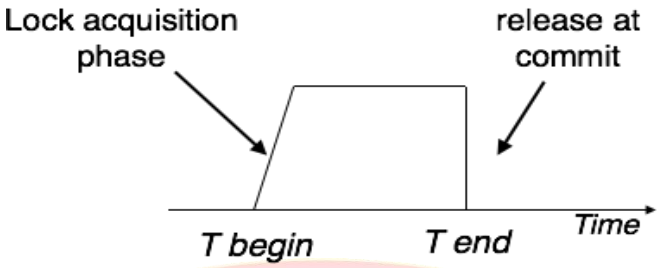



**Fig-4.5**

Strict-2PL does not have cascading abort as 2PL does.

**Timestamp-based Protocols**

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.
Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.
Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.
In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last 'read and write' operation was performed on the data item.

**Timestamp Ordering Protocol**

The timestamp-ordering protocol ensures serializability among transactions in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

- The timestamp of transaction $T_i$ is denoted as $TS(T_i)$.
- Read time-stamp of data-item X is denoted by R-timestamp(X).
- Write time-stamp of data-item X is denoted by W-timestamp(X).

Timestamp ordering protocol works as follows −

- If a transaction Ti issues a read(X) operation −
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected.
  - If TS(Ti) >= W-timestamp(X)
    - Operation executed.
  - All data-item timestamps updated.
- If a transaction Ti issues a write(X) operation −
  - If TS(Ti) < R-timestamp(X)
    - Operation rejected.
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected and Ti rolled back.
  - Otherwise, operation executed.

**Validation-Based Protocol:**

Execution of transaction Ti is done in three phases.

1. Read and execution phase: Transaction Ti writes only to temporary local variables
2. Validation phase: Transaction Ti performs a "validation test" to determine if local variables can be written without violating serializability.
3. Write phase: If Ti is validated, the updates are applied to the database; otherwise, Ti is rolled back.

- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order
- Each transaction Ti has 3 timestamps
  - Start(Ti): the time when Ti started its execution
  - Validation(Ti): the time when Ti entered its validation phase
  - Finish(Ti): the time when Ti finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency. Thus TS(Ti) is given the value of Validation(Ti).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low. That is because the serializability order is not pre-decided and relatively less transactions will have to be rolled back.
- If for all Ti with TS (Ti) < TS (Tj) either one of the following condition holds:
  - finish(Ti) < start(Tj)
  - start(Tj) < finish(Ti) < validation(Tj) and the set of data items written by Ti does not intersect with the set of data items read by Tj. then validation succeeds and Tj can be committed. Otherwise, validation fails and Tj is aborted.

  • Justification: Either first condition is satisfied, and there is no overlapped execution, or second condition is satisfied and
    1. the writes of Tj do not affect reads of Ti since they occur after Ti has finished its reads.
    2. the writes of Ti do not affect reads of Tj since Tj does not read any item written by Ti.

**Multiple Granularities:**

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- when a transaction locks a node in the tree explicitly, it implicitly locks all the node's descendants in the same mode.
- Granularity of locking (level in tree where locking is done):
  - fine granularity (lower in tree): high concurrency, high locking overhead
  - coarse granularity (higher in tree): low locking overhead, low concurrency

**Example of Granularity Hierarchy:**

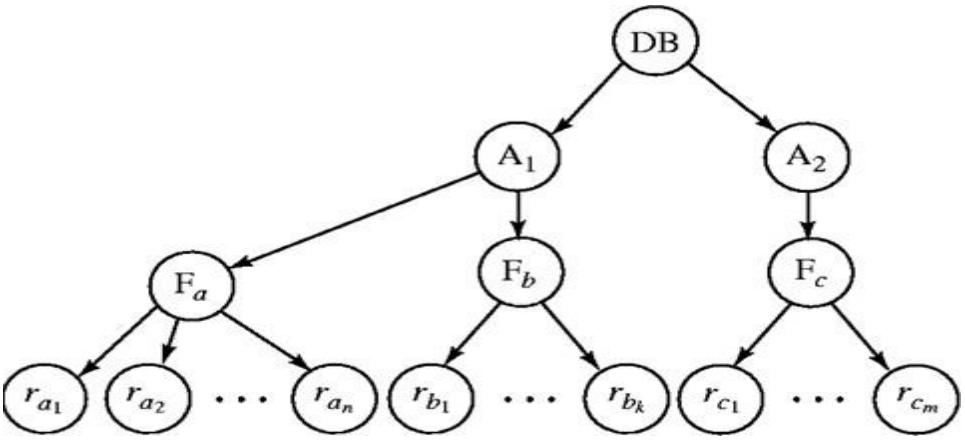



**Fig-4.6**

The highest level in the example hierarchy is the entire database. The levels below are of type area, file and record in that order.

**Multiversion Schemes:**

**Multiversion** concurrency control (MCC or MVCC), is a concurrency control method commonly used by database management systems to provide concurrent access to the database and in programming languages to implement transactional memory. If someone is reading from a database at the same time as someone else is writing to it, it is possible that the reader will see a half-written or inconsistent piece of data. There are several ways of solving this problem, known as concurrency control methods. The simplest way is to make all readers wait until the writer is done, which is known as a lock. This can be very slow, so MVCC takes a different approach: each user connected to the database sees a *snapshot* of the database at a particular instant in time. Any changes made by a writer will not be seen by other users of the database until the changes have been completed

**Recovery with Concurrent Transactions**

When more than one transaction is being executed in parallel, the logs are interleaved. At the time of recovery, it would become hard for the recovery system to backtrack all logs, and then start recovering. To ease this situation, most modern DBMS use the concept of 'checkpoints'.

**Distributed Databases:**

A distributed database appears to a user as a single database but is, in fact, a set of databases stored on multiple computers. The data on several computers can be simultaneously accessed and modified using a network. Each database server in the distributed database is controlled by its local DBMS, and each cooperates to maintain the consistency of the global database. Figure illustrates a representative distributed database system.

The following sections outline some of the general terminology and concepts used to discuss distributed database systems.

**Clients, Servers, and Nodes**

A database server is the software managing a database, and a client is an application that requests information from a server. Each computer in a system is a node. A node in a distributed database system can be a client, a server, or both.

**Data mining:**

Data mining is the computing process of discovering patterns in large data sets involving methods at the intersection of artificial intelligence, machine learning, statistics, and database systems. It is an interdisciplinary subfield of computer science. The overall goal of the data mining process is to extract information from a data set and transform it into an understandable structure for further use.

Aside from the raw analysis step, it involves database and data management aspects, data pre-processing, model and inference considerations, interestingness metrics, complexity considerations, post-processing of discovered structures, visualization, and online updating. Data mining is the analysis step of the "knowledge discovery in databases" process, or KDD.

The knowledge discovery in databases (KDD) process is commonly defined with the stages:

(1) Selection
(2) Pre-processing
(3) Transformation
(4) Data mining
(5) Interpretation/evaluation

Data mining involves six common classes of tasks:

- **Anomaly detection (outlier/change/deviation detection) –** The identification of unusual data records, that might be interesting or data errors that require further investigation.
- **Association rule learning (dependency modelling) –** Searches for relationships between variables. For example, a supermarket might gather data on customer purchasing habits. Using association rule learning, the supermarket can determine which products are frequently bought together and use this information for marketing purposes. This is sometimes referred to as market basket analysis.

- **Clustering –** is the task of discovering groups and structures in the data that are in some way or another "similar", without using known structures in the data.
- **Classification –** is the task of generalizing known structure to apply to new data. For example, an e-mail program might attempt to classify an e-mail as "legitimate" or as "spam".
- **Regression –** attempts to find a function which models the data with the least error that is,for estimating the relationships among data or datasets.
- **Summarization –** providing a more compact representation of the data set, including visualization and report generation.

**Data Warehouse:**

In computing, a data warehouse (DW or DWH), also known as an enterprise data warehouse (EDW), is a system used for reporting and data analysis, and is considered a core component of business intelligence. DWs are central repositories of integrated data from one or more disparate sources. They store current and historical data in one single place and are used for creating analytical reports for knowledge workers throughout the enterprise.

The data stored in the warehouse is uploaded from the operational systems (such as marketing or sales). The data may pass through an operational data store and may require data cleansing for additional operations to ensure data quality before it is used in the DW for reporting.

**Benefits:**

A data warehouse maintains a copy of information from the source transaction systems. This architectural complexity provides the opportunity to:

- Integrate data from multiple sources into a single database and data model. Mere congregation of data to single database so a single query engine can be used to present data is an ODS.
- Mitigate the problem of database isolation level lock contention in transaction processing systems caused by attempts to run large, long running, analysis queries in transaction processing databases.Maintain data history, even if the source transaction systems do not.
- Integrate data from multiple source systems, enabling a central view across the enterprise. This benefit is always valuable, but particularly so when the organization has grown by merger.
- Improve data quality, by providing consistent codes and descriptions, flagging or even fixing bad data.
- Present the organization's information consistently.
- Provide a single common data model for all data of interest regardless of the data's source.
- Restructure the data so that it makes sense to the business users.
- Restructure the data so that it delivers excellent query performance, even for complex analytic queries, without impacting the operational systems.
- Add value to operational business applications, notably customer relationship management (CRM) systems.
- Make decision–support queries easier to write.
- Optimized data warehouse architectures allow data scientists to organize and disambiguate repetitive data.

**Object Technology and DBMS**

An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages. The first criterion translates into five features: persistence, secondary storage management, concurrency, recovery and an ad hoc query facility. The second one translates into eight features: complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility and computational completeness.

An object-oriented database system must satisfy two criteria: it should be a DBMS, and it should be an object-oriented system, i.e., to the extent possible, it should be consistent with the current crop of object-oriented programming languages. The first criterion translates into five features: persistence, secondary storage management, concurrency, recovery and an ad hoc query facility. The second one translates into eight

features: complex objects, object identity, encapsulation, types or classes, inheritance, overridin combined with late binding, extensibility and computational completeness.

**Comparative study of OODBMS Vs DBMS:**

DBMS hides the complexity around structure of data by de-coupling the memory storage for manipulating the data and persistent disk storage structure. Of different DBMS systems evolved, the Relational model survived with commercial implementations with SQL as the interface to manage them calling them RDBMS.A system that can store objects (not real objects but the programmable objects of object oriented systems).We can't store the objects as they are, because they contain some memory references connecting different structures. One method of dealing with this problem is the serialization using Object - Relational mapping i.e, ORM. Commercial databases like Oracle provide methods to store objects in relational structure.

**Temporal Databases:**

Temporal database stores data relating to time instances. It offers temporal data types and stores information relating to past, present and future time.

More specifically the temporal aspects usually include valid time and transaction time. These attributes can be combined to form bitemporal data.

- **Valid time** is the time period during which a fact is true in the real world.
- **Transaction time** is the time period during which a fact stored in the database was known.
- **Bitemporal data** combines both Valid and Transaction Time.

It is possible to have timelines other than Valid Time and Transaction Time, such as Decision Time, in the database. In that case the database is called a multitemporal database as opposed to a bitemporal database. However, this approach introduces additional complexities such as dealing with the validity of (foreign) keys.

Temporal databases are in contrast to current databases (a term that doesn't mean currently available databases, some do have temporal features, see also below), which store only facts which are believed to be true at the current time.

**Deductive Database:**

A **deductive database** is a database system that can make deductions (i.e., conclude additional facts) based on rules and facts stored in the (deductive) database. Data log is the language typically used to specify facts, rules and queries in deductive databases. Deductive databases have grown out of the desire to combine logic programming with relational databases to construct systems that support a powerful formalism and are still fast and able to deal with very large datasets. Deductive databases are more expressive than relational databases but less expressive than logic programming systems. In recent years, deductive databases such as Data log have found new application in data integration, information extraction, networking, program analysis, security, and cloud computing.

**Multimedia Database:-**

Multimedia data typically means digital images, audio, video, animation and graphics together with text data. The acquisition, generation, storage and processing of multimedia data in computers and transmission over networks have grown tremendously in the recent past. Multimedia data are blessed with a number of exciting features. They can provide more effective dissemination of information in science, engineering, medicine, modern biology, and social sciences. It also facilitates the development of new paradigms in distance learning, and interactive personal and group entertainment.

The huge amount of data in different multimedia-related applications warranted to have databases as databases provide consistency, concurrency, integrity, security and availability of data. From an user perspective, databases provide functionalities for the easy manipulation, query and retrieval of highly relevant information from huge collections of stored data.

A Multimedia Database Management System (MMDBMS) is a framework that manages different types of data potentially represented in a wide diversity of formats on a wide array of media sources. It provides support for multimedia data types, and facilitate for creation, storage, access, query and control of a multimedia database

**Web Database**

A Web database is a database application designed to be managed and accessed through the Internet. Website operators can manage this collection of data and present analytical results based on the data in the Web database application. Databases first appeared in the 1990s, and have been an asset for businesses, allowing the collection of seemingly infinite amounts of data from infinite amounts of customers.

**Mobile Database-**

A mobile database uses wireless technology to allow mobile computers to connect to its system. The database consists of a client and server that connect to each other over a wireless network. Due to the vulnerability of wireless network signals, a cache of activity is maintained to ensure that sensitive information can be recovered. A mobile database is used to provide remote access to information that authorized users may need to obtain on a moment's notice.

Mobile or cloud computing usually consists of three components. Within a wireless network, a mobile database will have one or more base stations. These stations are responsible for controlling the communication signals that need to be passed from one host to another. A base station receives and sends information and often comes in the form of some type of wireless router.

Hosts are responsible for handling the actual transactions that occur within a mobile database. They are sometimes referred to as "fixed," since hosts do not typically change locations within the network. In order to process database requests, hosts use servers or software applications in order to access the data that is needed.

A third factor in mobile databases is the mobile unit. These are the portable computers, phones and other devices that request information from the database. Rather than communicating directly with the database server, mobile units route their requests through the base stations.

**UNIT-V**

**DBMS Vs RDBMS**

| | DBMS | RDBMS |
|---|---|---|
| 1) | DBMS applications store data as file. | RDBMS applications store data in a tabular form. |
| 2) | In DBMS, data is generally stored in either a hierarchical form or a navigational form. | In RDBMS, the tables have an identifier called primary key and the data values are stored in the form of tables. |
| 3) | Normalization is not present in DBMS. | Normalization is present in RDBMS. |
| 4) | DBMS does not apply any security with regards to data manipulation. | RDBMS defines the integrity constraint for the purpose of ACID (Atomicity, Consistency, Isolation and Durability) property. |
| 5) | DBMS uses file system to store data, so there will be no relation between the tables. | in RDBMS, data values are stored in the form of tables, so a relationship between these data values will be stored in the form of a table as well. |
| 6) | DBMS has to provide some uniform methods to access the stored information. | RDBMS system supports a tabular structure of the data and a relationship between them to access the stored information. |
| 7) | DBMS does not support distributed database. | RDBMS supports distributed database. |
| 8) | DBMS is meant to be for small organization and deal with small data. it supports single user. | RDBMS is designed to handle large amount of data. it supports multiple users. |
| 9) | Examples of DBMS are file systems, xml etc. | Example of RDBMS are MySQL, postage, SQL server, oracle etc. |

**Study of Relational Database Management Systems through SQL/MySQL: Architecture segments**

**MySQL**

MySQL is very different from other database servers, and its **architectural** characteristics make it useful for a wide range of purposes as well as making it a poor choice for others. MySQL is not perfect, but it is flexible enough to work well in very demanding environments, such as web applications. At the same time, MySQL can power embedded applications, data warehouses, content indexing and delivery software, highly available redundant systems, online transaction processing (OLTP), and much more.

To get the most from MySQL, you need to understand its design so that you can work with it, not against it. MySQL is flexible in many ways. For example, you can configure it to run well on a wide range of hardware, and it supports a variety of data types. However, MySQL's most unusual and important feature is its storage-engine architecture, whose design separates query processing and other server tasks from data storage and retrieval. This separation of concerns lets you choose how your data is stored and what performance, features, and other characteristics you want.

This provides a high-level overview of the MySQL server architecture, the major differences between the storage engines, and why those differences are important. We'll finish with some historical context and benchmarks. We've tried to explain MySQL by simplifying the details and showing examples. This discussion will be useful for those new to database servers as well as readers who are experts with other database servers.

MySQL's Logical Architecture

A figure show of how MySQL's components work together will help you understand the server

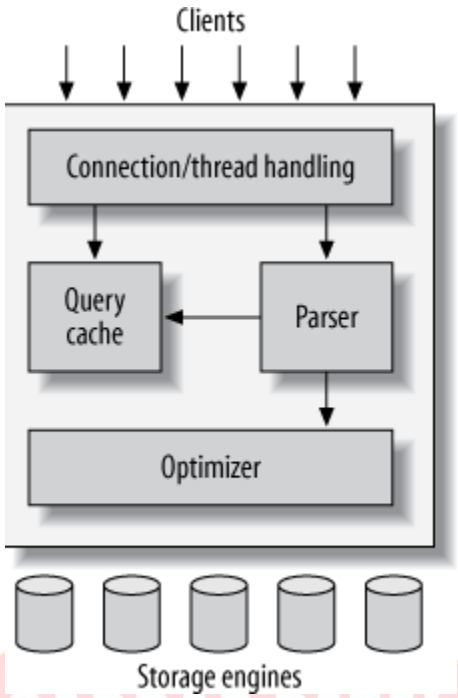



Figure 5-1. A logical view of the MySQL server architecture

The **topmost layer** contains the services that aren't unique to MySQL. They're services most network-based client/server tools or servers need: connection handling, authentication, security, and so forth.

The **second layer** is where things get interesting. Much of MySQL's brains are here, including the code for query parsing, analysis, optimization, caching, and all the built-in functions (e.g., dates, times, math, and encryption). Any functionality provided across storage engines lives at this level: stored procedures, triggers, and views, for example.

The **third layer** contains the storage engines. They are responsible for storing and retrieving all data stored "in" MySQL. Like the various filesystems available for GNU/Linux, each storage engine has its own benefits and drawbacks. The server communicates with them through the *storage engine API*. This interface hides differences between storage engines and makes them largely transparent at the query layer. The API contains a couple of dozen low-level functions that perform operations such as "begin a transaction" or "fetch the row that has this primary key." The storage engines don't parse SQL or communicate with each other; they simply respond to requests from the server.

**Physical files and memory structures**

**Introduction**:

The collection of data that makes up a computerized database must be stored physically on some computer storage medium. The DBMS software that can then retrieve, updates, and processes this data as needed. Computer storage media from a storage hierarchy that includes two main categories.

**Primary Storage**

The category includes storage media that can be operated on directly by the computer Central Processing Unit(CPU), such as the computer main memory and smaller but faster cache memories. Primary storage usually provides fast access to data but is of limited storage capacity.

**Secondary Storage**

This category includes magnetic disks, optical disks, and tapes. These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices. Data in secondary storage cannot be processed directly by the CPU: it must first be copied into

**Primary storage.**

The storage media are classified by the speed with which data can be accessed, by the cost per unit of data to buy the medium, and by the medium's reliability. Let's look into the media that are typically available.

Cache: The cache is the fastest and most costly form of storage. Cache memory is small. The computer hardware manages its use

**Main Memory**: The general-purpose machine instructions operate on main memory. If a power failure or system crash occurs, the contents of main memory are usually lost. (i.e.) volatile memory type.

**Flash Memory:** Also, known as Electrically Erasable programmable read only memory (EEPROM). Data Survive power failure in flash memory (Non-volatile Type). Reading data from flash memory takes less than 100 nanoseconds, which is as fast as reading data from main memory. However, writing data to flash memory is more complicated. Data can be written once, but cannot be overwritten directly. To overwrite memory that has been written already, we have to erase an entire bank of memory once.
It is used for storing small volumes of data (5-10 MB) in hand-held computers, digital cameras.

**Background process**

SHOW PROCESSLIST shows you which threads are running. You can also get this information from the INFORMATION_SCHEMAPROCESSLIST table or the mysqladmin process list command. If you have the PROCESS privilege, you can see all threads. Otherwise, you can see only your own threads (that is, threads associated with the MySQL account that you are using). If you do not use the FULL keyword, only the first 100 characters of each statement are shown in the Info field.
Process information is also available from the performance_schema threads table. However, access to threads does not require a mutex and has minimal impact on server performance. INFORMATION_SCHEMA.PROCESSLIST and SHOW PROCESSLIST have negative performance consequences because they require a mutex. threads also show information about background threads, which INFORMATION_SCHEMA.PROCESSLIST and SHOW PROCESSLIST do not. This means that threads can be used to monitor activity the other thread information sources cannot.

**Table Spaces**

A data file that can hold data for one or more InnoDB tables and associated indexes.
There are many types of tablespaces based on the configuration w.r.t the information clubbing per table. These are:
a. System tablespace b. File per tablespace c. General tablespace
System tablespace contains,

1. InnoDB data dictionary.
2. DoubleWrite Buffer.
3. Change buffer
4. Undo Logs.

Apart from this it also contains,

1. Tables &
2. Index data

Associated file is. idbdata1
The innodb_file_per_table option, which is enabled by default in MySQL 5.6 and higher, allows tables to be created in file-per-table tablespaces, with a separate data file for each table. Enabling the innodb_file_per_table option makes available other MySQL features such as table compression and transportable tablespaces.

Associated file is .idbd
InnoDB introduced general tablespaces in MySQL 5.7.6. General tablespaces are shared tablespaces created using CREATE TABLESPACE syntax. They can be created outside of the MySQL data directory, are capable of holding multiple tables, and support tables of all row formats.

**Introduction to Data Blocks, Extents, and Segments**

My sql allocates logical database space for all data in a database. The units of database space allocation are data blocks, extents, and segments. Figure shows the relationships among these data structures:

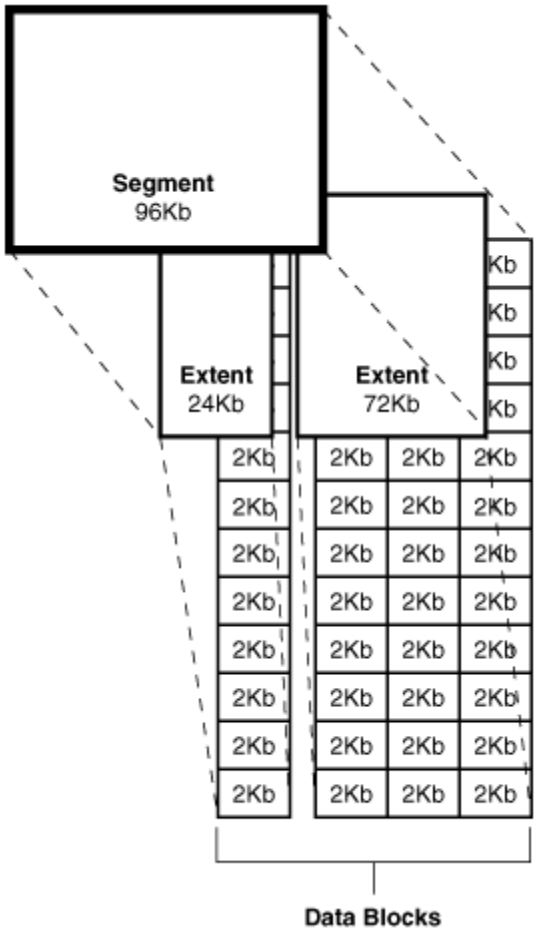



Figure 5.1 The Relationships Among Segments, Extents, and Data Blocks"

At the finest level of granularity, Mysql stores data in data blocks (also called logical blocks, My sql blocks, or pages). One data block corresponds to a specific number of bytes of physical database space on disk.

The next level of logical database space is an extent. An extent is a specific number of contiguous data blocks allocated for storing a specific type of information.

The level of logical database storage greater than an extent is called a segment. A segment is a set of extents, each of which has been allocated for a specific data structure and all of which are stored in the same tablespace. For example, each table's data is stored in its own data segment, while each index's data is stored in its own index segment. If the table or index is partitioned, each partition is stored in its own segment.

My sql allocates space for segments in units of one extent. When the existing extents of a segment are full, My sql allocates another extent for that segment. Because extents are allocated as needed, the extents of a segment may or may not be contiguous on disk.

A segment and all its extents are stored in one tablespace. Within a tablespace, a segment can include extents from more than one file; that is, the segment can span datafiles. However, each extent can contain data from only one datafile.

Although you can allocate additional extents, the blocks themselves are allocated separately. If you allocate an extent to a specific instance, the blocks are immediately allocated to the free list. However, if the extent is not allocated to a specific instance, then the blocks themselves are allocated only when the high-water mark moves. The high-water mark is the boundary between used and unused space in a segment.

**Profiles**

MySQL query profiling is a useful technique when trying to analyze the overall performance of a database driven application. When developing a mid to large size application, there tends to be hundreds of queries distributed throughout a large code base and potentially numerous queries ran against the database per second. Without some sort of query profiling techniques, it becomes very difficult to determine locations and causes of bottlenecks and applications slow down. This article will demonstrate some useful query profiling techniques using tools that are built into MySQL server.

**Dedicated Server**

A dedicated server is a single computer in a network reserved for serving the needs of the network. For example, some networks require that one computer be set aside to manage communications between all the other computers. A dedicated server could also be a computer that manages printer resources. Note, however, that not all servers are dedicated. In some networks, it is possible for a computer to act as a server and perform other functions as well.

In the Web hosting business, a dedicated server is typically a rented service. The user rents the server, software and an Internet connection from the Web host.

**Multithreaded server**

Multithreaded server, also known as a shared server, allows many user processes to share a few shared server processes to connect to the database. Without MTS, each user process spawns its own dedicated server process, consuming OS memory. A dedicated server process remains associated to the user process for the remainder of the connection.

Multithreading will allow a program or an OS to handle multiple users and requests simultaneously by utilizing multiple threads, as opposed to running the program multiple times. It uses the program resources more effectively by handling multiple queries or commands and tracking them with multiple threads until completed.

**Advantages of Multithreading**

- Increase performance
- Concurrency
- Reduce number of servers needed reducing cost and maintenance
- Compatibility with applications that create client threads such as API
- Parallel tasks
- More efficient use of system resources

**Disadvantages of Multithreading**

- Synchronizing resources that are being share

- Potential deadlocks

Increased complexity in programming as well as troubleshooting problems

**Distributed Database System-**

A distributed database (DDB) is a collection of multiple, *logically interrelated* databases distributed over a *computer network.*

A distributed database management system (D-DBMS) is the software that manages the DDB and provides an access mechanism that makes this distribution transparent to the users.

Distributed database system (DDBS) = DDB + D– DBMS

What is not a DDBS?

A timesharing computer system.

A loosely or tightly coupled multiprocessor system.

A database system which resides at one of the nodes of a network of computers - this is a centralized database on a network nod.

**Database links, and snapshot**

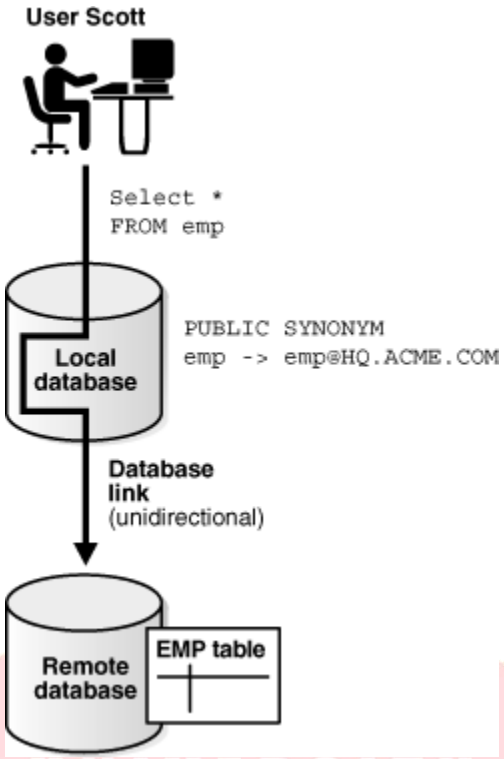
**Database Links?**

A database link is a pointer that defines a one-way communication path from an Oracle Database server to another database server. The link pointer is actually defined as an entry in a data dictionary table. To access the link, you must be connected to the local database that contains the data dictionary entry.

A database link connection is one-way in the sense that a client connected to local database A can use a link stored in database A to access information in remote database B, but users connected to database B cannot use the same link to access data in database A. If local users on database B want to access data on database A, then they must define a link that is stored in the data dictionary of database B.

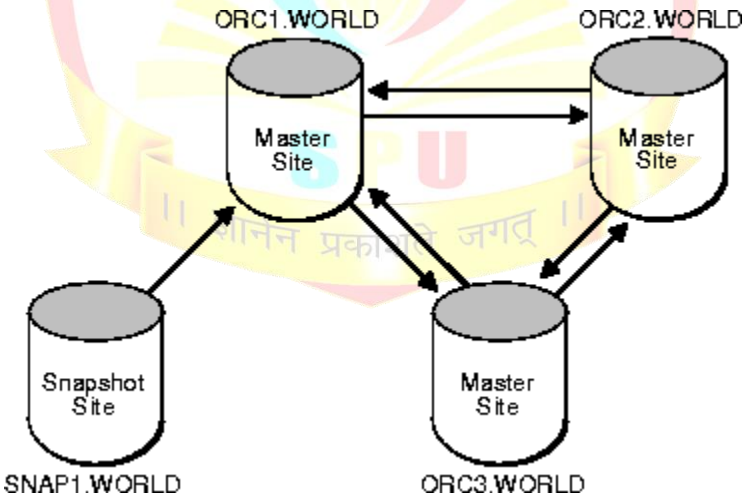
A database link connection allows local users to access data on a remote database. For this connection to occur, each database in the distributed system must have a unique **global database name** in the network domain. The global database name uniquely identifies a database server in a distributed system.

Figure shows an example of user scott accessing the emp table on the remote database with the global name hq.acme.com:




**Snapshot**

A snapshot is a replica of a target master table from a single point-in-time. Whereas in multimaster replication tables are continuously being updated by other master sites, snapshots are updated by one or more master tables via individual batch updates, known as a *refresh*, from a single master site.




Snapshots also have the option of containing a WHERE clause so that snapshot sites can contain custom data sets, which can be very helpful for regional offices or sales forces that don't require the complete corporate data set.

**Why use Snapshots?**

Oracle offers a variety of snapshots to meet the needs of many different replication (and non-replication) situations; each of these snapshots will be discussed in detail in following sections. You might use a snapshot to achieve one or more of the following:

- Ease Network Loads
- Mass Deployment

- Data Subsetting
- Disconnected Computing

**Data dictionary**

A **metadata** (also called the **data dictionary**) is the data about the data. It is the self describing nature of the database that provides program-data independence. It is also called as the System Catalog. It holds the following information about each data element in the databases, it normally includes:

- Name
- Type
- Range of values
- Source
- Access authorization
- Indicates which application programs use the data so that, when a change in a data structure is contemplated, a list of the affected programs can be generated.

**Active and Passive Data Dictionaries**

An **active data dictionary** (also called integrated data dictionary) is managed automatically by the database management software.

**The passive data dictionary** (also called non-integrated data dictionary) is the one used only for documentation purposes. Data about fields, files, people and so on, in the data processing environment are.

***Dynamic Performance Views***

Throughout its operation, Oracle Database maintains a set of virtual tables that record current database activity. These views are dynamic because they are continuously updated while a database is open and in use. The views are sometimes called V$ views because their names begin with V$.

**These views contain information such as the following:**

- System and session parameters
- Memory usage and allocation
- File states (including RMAN backup files)
- Progress of jobs and tasks
- SQL execution
- Statistics and metrics

**The dynamic performance views have the following primary uses:**

- Oracle Enterprise Manager Uses the views to obtain information about the database (see "Oracle Enterprise Manager").
- Administrators can use the views for performance monitoring and debugging.

**Privilege and Security**

A **privilege** is a right to execute a particular type of SQL statement or to access another user's object. Some examples of privileges include the right to:

- Connect to the database (create a session)
- Create a table
- Select rows from another user's table
- Execute another user's stored procedure

There are two distinct categories of privileges:

- System privileges
- Schema object privileges

**Database security** concerns the use of a broad range of information security controls to protect databases (potentially including the data, the database applications or stored functions, the database systems, the database servers and the associated network links) against compromises of their confidentiality, integrity and availability. It involves various types or categories of controls, such as technical, procedural/administrative and

physical. *Database security* is a specialist topic within the broader realms of computer security, information security and risk management.

**Privileges**

Authorization includes primarily two processes:

- Permitting only certain users to access, process, or alter data.
- Applying varying limitations on user access or actions. The limitations placed on (or removed from) users can apply to objects such as schemas, tables, or rows or to resources such as time (CPU, connect, or idle times).

A user privilege is the right to run a particular type of SQL statement, or the right to access an object that belongs to another user, run a PL/SQL package, and so on. The types of privileges are defined by Oracle Database.

**Managing User Role**

This section contains:

- About User Roles
- Predefined Roles in an Oracle Database Installation
- Creating a Role
- Specifying the Type of Role Authorization
- Dropping Roles
- Restricting SQL*Plus Users from Using Database Roles

***Uses of Roles***

In general, you create a role to serve one of two purposes:

- To manage the privileges for a database application (see "Common Uses of Application Roles")
- To manage the privileges for a user group (see "Common Uses of User Roles")

**Introduction to SQL**

The history of SQL begins in an IBM laboratory in San Jose, California, where SQL was developed in the late 1970s. The initials stand for Structured Query Language, and the language itself is often referred to as "sequel." It was originally developed for IBM's DB2 product (a relational database management system, or RDBMS, that can still be bought today for various platforms and environments). In fact, SQL makes an RDBMS possible. SQL is a nonprocedural language, in contrast to the procedural or third-generation languages (3GLs) such as COBOL and C that had been created up to that time.

**Dr. Codd's 12 Rules for a Relational Database Model**

The most popular data storage model is the relational database, which grew from the seminal paper "A Relational Model of Data for Large Shared Data Banks," written by Dr. E. F. Codd in 1970. SQL evolved to service the concepts of the relational database model. Dr. Codd defined 13 rules, oddly enough referred to as Codd's 12 Rules, for the relational model:

A relational DBMS must be able to manage databases entirely through its relational capabilities.

**1.** Information rule-- All information in a relational database (including table and column names) is represented explicitly as values in tables.

**2.** Guaranteed access--Every value in a relational database is guaranteed to be accessible by using a combination of the table name, primary key value, and column name.

**3.** Systematic null value support--The DBMS provides systematic support for the treatment of null values (unknown or inapplicable data), distinct from default values, and independent of any domain.

**4.** Active, online relational catalog--The description of the database and its contents is represented at the logical level as tables and can therefore be queried using the database language.

**5.** Comprehensive data sublanguage--At least one supported language must have a well-defined syntax and be comprehensive. It must support data definition, manipulation, integrity rules, authorization, and transactions.

**6.** View updating rule--All views that are theoretically updatable can be updated through the system.
**7.** Set-level insertion, update, and deletion--The DBMS supports not only set-level retrievals but also set-level inserts, updates, and deletes.
**8.** Physical data independence--Application programs and ad hoc programs are logically unaffected when physical access methods or storage structures are altered.
**9.** Logical data independence--Application programs and ad hoc programs are logically unaffected, to the extent possible, when changes are made to the table structures.
**10.** Integrity independence--The database language must be capable of defining integrity rules. They must be stored in the online catalog, and they cannot be bypassed.
**11.** Distribution independence--Application programs and ad hoc requests are logically unaffected when data is first distributed or when it is redistributed.
**12.** No subversion--It must not be possible to bypass the integrity rules defined through the database language by using lower-level

**SQL in Application Programming**
SQL was originally made an ANSI standard in 1986. The ANSI 1989 standard (often called SQL-89) defines three types of interfacing to SQL within an application program:

Module Language-Uses procedures within programs. These procedures can be called by the application program and can return values to the program via parameter passing.

Embedded SQL-Uses SQL statements embedded with actual program code. This method often requires the use of a precompile to process the SQL statements. The standard defines statements for Pascal, FORTRAN, COBOL, and PL/1.

Direct Invocation-Left up to the implementer. Before the concept of dynamic SQL evolved, embedded SQL was the most popular way to use SQL within a program. Embedded SQL, which is still used, uses *static* SQL--meaning that the SQL statement is compiled into the application and cannot be changed at runtime. The principle is much the same as a compiler versus an interpreter. The performance for this type of SQL is good; however, it is not flexible--and cannot always meet the needs of today's changing business environments. Dynamic SQL is discussed shortly.

The ANSI 1992 standard (SQL-92) extended the language and became an international standard. It defines three levels of SQL compliance: entry, intermediate, and full. The new features introduced include the following:

Connections to databases
Scrollable cursors
Dynamic SQL
Outer joins
**SQL queries**
Structure Query Language(SQL) is a programming language used for storing and managing data in RDBMS. SQL was the first commercial language introduced for E.F Codd's **Relational** model. Today almost all RDBMS (MySql, Oracle, Infomix, Sybase, MS Access) uses **SQL** as the standard database language. SQL is used to perform all type of data operations in RDBMS.
SQL Joins
SQL Joins are used to relate information in different tables. A Join condition is a part of the sql query that retrieves rows from two or more tables. A SQL Join condition is used in the SQL *WHERE Clause* of select, update, delete statements.

**Joins in SQL**
The **SQL Syntax** for joining two tables is:

**Syntax:**
*SELECT column_list*
*FROM table1, table2....*
*WHERE table1.column_name =*
*table2.column_name;*
*or*
*SELECT **
*FROM table1*
*JOIN table2*
*[ON (join_condition)]*

If a sql join condition is omitted or if it is invalid the join operation will result in a Cartesian product. The Cartesian product returns a number of rows equal to the product of all rows in all the tables being joined. For example, if the first table has 20 rows and the second table has 10 rows, the result will be 20 * 10, or 200 rows. This query takes a long time to execute.
SQL Joins Example
Lets use the below two tables to explain the sql join conditions.

**Database table "product";**

| product_id | product_name | supplier_name | unit_price |
|---|---|---|---|
| 100 | Camera | Nikon | 300 |
| 101 | Television | Onida | 100 |
| 102 | Refrigerator | Vediocon | 150 |
| 103 | Ipod | Apple | 75 |
| 104 | Mobile | Nokia | 50 |

**Database table "order_items";**

| order_id | product_id | total_units | customer |
|---|---|---|---|
| 5100 | 104 | 30 | Infosys |
| 5101 | 102 | 5 | Satyam |
| 5102 | 103 | 25 | Wipro |
| 5103 | 101 | 10 | TCS |

SQL Joins can be classified into Equi join and Non Equi join.
**1) SQL Equi joins**
It is a simple sql join condition which uses the equal sign as the comparison operator. Two types of equi joins are SQL Outer join and SQL Inner join.
**For example:** You can get the information about a customer who purchased a product and the quantity of product.
**2) SQL Non equi joins**
It is a sql join condition which makes use of some comparison operator other than the equal sign like >, <, >=, <=

1) SQL Equi Joins:
An equi-join is further classified into two categories:

a) SQL Inner Join
b) SQL Outer Join
a) SQL Inner Join:
All the rows returned by the sql query satisfy the sql join condition specified.
**SQL Inner Join Example:**
If you want to display the product information for each order the query will be as given below. Since you are retrieving the data from two tables, you need to identify the common column between these two tables, which is the product_id.
The query for this type of sql joins would be like,

```
SELECT order_id, product_name, unit_price, supplier_name, total_units
FROM product, order_items
WHERE order_items.product_id = product.product_id;
```

The columns must be referenced by the table name in the join condition, because product_id is a column in both the tables and needs a way to be identified. This avoids ambiguity in using the columns in the SQL SELECT statement.
The number of join conditions is (n-1), if there are more than two tables joined in a query where 'n' is the number of tables involved. The rule must be true to avoid Cartesian product.
We can also use aliases to reference the column name, then the above query would be like,

```
SELECT o.order_id, p.product_name, p.unit_price, p.supplier_name, o.total_units
FROM product p, order_items o
WHERE o.product_id = p.product_id;
```

SQL Self Join:
A Self Join is a type of sql join which is used to join a table to itself, particularly when the table has a FOREIGN KEY that references its own PRIMARY KEY. It is necessary to ensure that the join statement defines an alias for both copies of the table to avoid column ambiguity.
The below query is an example of a self-join,

```
SELECT a.sales_person_id, a.name, a.manager_id, b.sales_person_id, b.name
FROM sales_person a, sales_person b
WHERE a.manager_id = b.sales_person_id;
```

2) SQL Non Equi Join:
A Non Equi Join is a SQL Join whose condition is established using all comparison operators except the equal (=) operator. Like >=, <=, <, >

**SQL Non Equi Join Example:**
If you want to find the names of students who are not studying either Economics, the sql query would be like, (lets use student_details table defined earlier).

```
SELECT first_name, last_name, subject
FROM student_details
WHERE subject != 'Economics'
```

| first_name | last_name | subject |
|---|---|---|
| Anajali | Bhagwat | Maths |
| Shekar | Gowda | Maths |
| Rahul | Sharma | Science |
| Stephen | Fleming | Science |

**Outer Join**

This type of join is needed when we need to select all the rows from the table on the left (or right or both) regardless of whether the other table has common values or not and it usually enter null values for the data which is missing.

The Outer join can be of three types

1. **Left Outer Join**
2. **Right Outer Join**
3. **Full Outer Join**

**Left Outer Join**

If we want to get employee id, employee first name, employes last name and their department name for all the employees regardless of whether they belong to any department or not,then we can use the left outer join. In this case we keep the Employee table on the left side of the join clause. It will insert NULL values for the data which is missing in the right table.

**Query for Left Outer Join**

*SELECT Emp.Empid, Emp.EmpFirstName, Emp.EmpLastName, Dept.DepartmentName,*
*FROM Employee Emp LEFT OUTER JOIN Department dept*
*ON Emp.Departmentid=Dept.Departmenttid*

**Result**

| Empid | EmpFirstName | EmpLastName | DepartmentName |
|---|---|---|---|
| 1 | samir | singh | admin |
| 2 | amit | kumar | accounts |
| 3 | neha | sharma | admin |
| 4 | vivek | kumar | NULL |

**Right Outer Join**

If we want to get all the departments name and employee id, employee first name, and employees last name of all the employees belonging to the department regardless of whether a department have employees or not, then we can use the right outer join. In this case we keep the Department table on the right side of the join clause. It will insert NULL values for the data which is missing in the left table (Employee).

*SELECT Dept.DepartmentName, Emp.Empid, Emp.EmpFirstName, Emp.EmpLastName FROM Employee Emp*
*RIGHT OUTER JOIN Department dept*
*ON Emp.Departmentid=Dept.Departmentid*

| DepartmentName | Empid | EmpFirstName | EmpLastName |
|---|---|---|---|
| accounts | 2 | amit | kumar |
| admin | 1 | samir | singh |
| admin | 3 | neha | sharma |
| HR | NULL | NULL | NULL |
| Technology | NULL | NULL | NULL |

**Full Outer Join**

If we want to get all the departments name and the employee id, employee first name, employes last name of all the employees regardless of whether a department have employees or not, or whether a employee belong to a department or not, then we can use the full outer join. It will insert null values for the data which is missing in both the tables.

**Query for Full Outer Join**

*SELECT Emp.Empid, Emp.EmpFirstName, Emp.EmpLastName, Emp.EmpLastName,Dept.DepartmentName, FROM Employee Emp FULL OUTER JOIN Department dept ON Emp.Departmentid=Dept.Departmenttid*

| Empid | EmpFirstName | EmpLastName | DepartmentName |
|---|---|---|---|
| 1 | samir | singh | admin |
| 2 | amit | kumar | accounts |
| 3 | neha | sharma | admin |
| 4 | vivek | kumar | NULL |
| NULL | NULL | NULL | HR |
| NULL | NULL | NULL | Technology |

**Special Select operator**

**ANY**

The ANY operator returns true if any of the subquery values meet the condition.

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name operator* ANY
(SELECT *column_name* FROM *table_name* WHERE *condition*);

**ALL**

The ALL operator returns true if all of the subquery values meet the condition.

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name operator* ALL
(SELECT *column_name* FROM *table_name* WHERE *condition*);

**LIKE Operator**

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

- % The percent sign represents zero, one, or multiple characters
- _ The underscore represents a single character

*SELECT column1, column2, ...*
*FROM table_name*
*WHERE columnN LIKE pattern;*

**The SQL IN Operator**

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

SELECT *column_name(s)*
FROM *table_name*
WHERE *column_name* IN (*value1*, *value2*, ...);

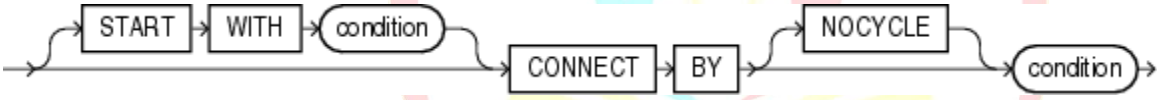
**The SQL EXISTS Operator**

The EXISTS operator is used to test for the existence of any record in a subquery.

The EXISTS operator returns true if the subquery returns one or more records.

SELECT *column_name(s)*
FROM *table_name*
WHERE EXISTS
(SELECT *column_name* FROM *table_name* WHERE *condition*);

A hierarchy is built upon a parent-child relationship within the same table or view.




START WITH specifies the root row(s) of the hierarchy.

CONNECT BY specifies the relationship between parent rows and child rows of the hierarchy.

The NOCYCLE parameter instructs Oracle Database to return rows from a query even if a CONNECT BY LOOP exists in the data. Use this parameter along with the CONNECT_BY_ISCYCLE pseudocolumn to see which rows contain the loop. Please refer to CONNECT_BY_ISCYCLE Pseudocolumn for more information.

In a hierarchical query, one expression in condition must be qualified with the PRIOR operator to refer to the parent row.

hierarchical queries as follows:
A join, if present, is evaluated first, whether the join is specified in the FROM clause or with WHERE clause predicates.
The CONNECT BY condition is evaluated.
Any remaining WHERE clause predicates are evaluated.

```
SELECT employee_id, last_name, manager_id
  FROM employees
  CONNECT BY PRIOR employee_id = manager_id;
```

**Inline Query**

When you use SQL Subquery in From clause of the select statement it is called **inline** view.

A common use for **inline views** in Oracle SQL is to simplify complex queries by removing join operations and condensing several separate queries into a single query. A **subquery** which is enclosed in parenthesis in the FROM clause may be given an alias name. The columns selected in the **subquery** can be referenced in the parent query, just as you would select from any normal table or view.

**Example**

Display the top five earner names and salaries from the EMPLOYEES table:

```
SELECT ROWNUM as RANK, last_name, salary
FROM (SELECT last_name, salary
    FROM employees
    ORDER BY salary DESC)
WHERE ROWNUM <= 5;
```

**Flashback Query**

It is useful to recover from accidental statement failures. For example, suppose a user accidently deletes rows from a table and commits it also then, using flash back query he can get back the rows.

Flashback feature depends upon on how much undo retention time you have specified. If you have set the UNDO_RETENTION parameter to 2 hours then, Oracle will not overwrite the data in undo tablespace even after committing until 2 Hours have passed. Users can recover from their mistakes made since last 2 hours only.

For example, suppose John gives a delete statement at 10 AM and commits it. After 1 hour he realizes that delete statement is mistakenly performed. Now he can give a flashback AS.. OF query to get back the deleted rows like this.

**Flashback Query**

```
SQL>select * from emp as of timestamp sysdate-1/24;
Or
SQL> SELECT * FROM emp AS OF TIMESTAMP
    TO_TIMESTAMP('2007-06-07 10:00:00', 'YYYY-MM-DD HH:MI:SS')
```

To insert the accidently deleted rows again in the table he can type

```
SQL> insert into emp (select * from emp as of timestamp sysdate-1/24)
```

**Introduction of ANSI SQL**

SQL is an ANSI (American National Standards Institute) standard

Although SQL is an ANSI (American National Standards Institute) standard, there are different versions of the SQL language.

However, to be compliant with the ANSI standard, they all support at least the major commands (such as SELECT, UPDATE, DELETE, INSERT, WHERE) in a similar manner.

An anonymous block is an unnamed sequence of actions. Since they are unnamed, anonymous blocks cannot be referenced by other program units.

In contrast to anonymous blocks, stored/ named code blocks include Packages, Procedures, and Functions.

Here are some example anonymous blocks written in PL/SQL:

**Description**

**DECLARE**
An optional keyword that starts the DECLARE statement, which can be used to declare data types, variables, or cursors. The use of this keyword depends upon the context in which the block appears.
declaration
Specifies a variable, cursor, or type declaration whose scope is local to the block. Each declaration must be terminated by a semicolon.
**BEGIN**
A mandatory keyword that introduces the executable section, which can include one or more SQL or PL/SQL statements. A BEGIN-END block can contain nested BEGIN-END blocks.
statement
Specifies a PL/SQL or SQL statement. Each statement must be terminated by a semicolon.
**EXCEPTION**
An optional keyword that introduces the exception section.
WHEN exception-condition
Specifies a conditional expression that tests for one or more types of exceptions.
THEN handler-statement
Specifies a PL/SQL or SQL statement that is executed if a thrown exception matches an exception in exception-condition. Each statement must be terminated by a semicolon.
**END**
A mandatory keyword that ends the block.
Examples
The following example shows the simplest possible anonymous block statement that the DB2 data server can compile:

```
BEGIN
   NULL;
END;
```

**Example:**

```
DECLARE
  current_date DATE := SYSDATE;
BEGIN
   dbms_output.put_line( current_date );
END;
```

**Cursor**
**Database cursor** is a control structure that enables traversal over the records in a database. Cursors facilitate subsequent processing in conjunction with the traversal, such as retrieval, addition and removal of database records.
There are two types of cursors −

1) **Implicit Cursors**

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

2) **Explicit Cursors**

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.
Creating an explicit cursor is –

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

```
DECLARE
  c_id customers.id%type;
  c_name customerS.No.ame%type;
  c_addr customers.address%type;
  CURSOR c_customers is
    SELECT id, name, address FROM customers;
BEGIN
  OPEN c_customers;
  LOOP
  FETCH c_customers into c_id, c_name, c_addr;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE c_customers;
END;
```

**Stored procedure**

A procedure (often called a stored procedure) is a subroutine like a subprogram in a regular computing language, stored in database. A procedure has a name, a parameter list, and SQL statement(s).

**Why Stored Procedures?**

- Stored procedures are fast. MySQL server takes some advantage of caching, just as prepared statements do. The main speed gain comes from reduction of network traffic. If you have a repetitive task that requires checking, looping, multiple statements, and no user interaction, do it with a single call to a procedure that's stored on the server.
- Stored procedures are portable. When you write your stored procedure in SQL, you know that it will run on every platform that MySQL runs on, without obliging you to install an additional runtime-environment package, or set permissions for program execution in the operating system, or deploy different packages if you have different computer types. That's the advantage of writing in SQL rather than in an external language like Java or C or PHP.
- Stored procedures are always available as 'source code' in the database itself. And it makes sense to link the data with the processes that operate on the data.
- Stored procedures are migratory! MySQL adheres fairly closely to the SQL:2003 standard. Others (DB2, Mimer) also adhere.

**Create Procedure**

Following statements create a stored procedure. By default, a procedure is associated with the default database (currently used database). To associate the procedure with a given database, specify the name as database_name**.** stored_procedure_name when you create it. Here is the complete syntax:

**Syntax:**

```
CREATE [DEFINER = { user | CURRENT_USER }]
PROCEDURE sp_name ([proc_parameter[,...]])
[characteristic ...] routine_body
proc_parameter: [ IN | OUT | INOUT ] param_name type
type:
Any valid MySQL data type
```

characteristic:
COMMENT 'string'
| LANGUAGE SQL
| [NOT] DETERMINISTIC
| {CONTAINS SQL | NO SQL | READS SQL DATA
| MODIFIES SQL DATA}
| SQL SECURITY {DEFINER | INVOKER}
routine_body:

**Function**

A stored function is a special kind stored program that returns a single value. You use stored functions to encapsulate common formulas or business rules that are reusable among SQL statements or stored programs. Different from a stored procedure, you can use a stored function in SQL statements wherever an expression is used. This helps improve the readability and maintainability of the procedural code.

**MySQL stored function syntax**

```
DELIMITER $$

CREATE FUNCTION CustomerLevel(p_creditLimit double) RETURNS VARCHAR(10)
   DETERMINISTIC
BEGIN
   DECLARE lvl varchar(10);

   IF p_creditLimit > 50000 THEN
SET lvl = 'PLATINUM';
   ELSEIF (p_creditLimit <= 50000 AND p_creditLimit >= 10000) THEN
      SET lvl = 'GOLD';
   ELSEIF p_creditLimit < 10000 THEN
      SET lvl = 'SILVER';
   END IF;

RETURN (lvl);
END
```

**Trigger**

In a DBMS, a *trigger* is a SQL procedure that initiates an action (i.e., *fires* an action) when an event (INSERT, DELETE or UPDATE) occurs. Since triggers are event-driven specialized procedures, they are stored in and managed by the DBMS. A trigger cannot be called or executed; the DBMS automatically fires the trigger as a result of a data modification to the associated table. Triggers are used to maintain the referential integrity of data by changing the data in a systematic fashion.

**Syntax** CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{INSERT [OR]| UPDATE [OR]| DELETE}
[OF col_name]
ON table_name

*Require INSTEAD OF Triggers*

A view cannot be modified by UPDATE, INSERT, or DELETE statements if the view query contains any of the following constructs:

- A set operator

- A DISTINCT operator
- An aggregate or analytic function
- A GROUP BY, ORDER BY, MODEL, CONNECT BY, or START WITH clause
- A collection expression in a SELECT list
- A subquery in a SELECT list
- A subquery designated WITH READ ONLY
- Joins, with some exceptions, as documented in Oracle Database Administrator's Guide

If a view contains pseudocolumns or expressions, then you can only update the view with an UPDATE statement that does not refer to any of the pseudocolumns or expressions.
INSTEAD OF triggers provide the means to modify object view instances on the client-side through OCI calls. To modify an object materialized by an object view in the client-side object cache and flush it back to the persistent store, you must specify INSTEAD OFtriggers, unless the object view is modifiable. If the object is read only, then it is not necessary to define triggers to pin it.

**Mutating Errors**

Mutating table errors occur when a trigger attempts to access a "mutating" table, like the table from which the trigger has been called. In MySQL, triggers are not initiated by cascading foreign keys, and they cannot modify the table from which they are called, so neither of those issues can cause a mutating table error. However, foreign keys appear to operate before 'AFTER' triggers, so it is possible to create a trigger that will cause a statement to fail.

**Instead of Triggers**

INSTEAD OF triggers can be defined on either tables or views; however, INSTEAD OF triggers are most useful for extending the types of updates a view can support. For example, INSTEAD OF triggers can provide the logic to modify multiple base tables through a view or to modify base tables.

**Example using an INSTEAD-OF-trigger:**

```
CREATE VIEW VIEW_TEST
AS SELECT A.NAME , A.ID_LOC , B.CITY , B.STATE , B.COUNTRY , A.DEPARTMENT FROM A, B
where A.ID_LOC = B.ID_LOC;

CREATE TRIGGER TRG_VIEW_TEST_INST_OF
INSTEAD OF INSERT ON VIEW_TEST
FOR EACH ROW
BEGIN
insert into B (ID_LOC, CITY, state, country)
values (varID, varCity, varState, varCountry);

INSERT INTO A (ID, NAME, ID_LOC, DEPARTMENT)
VALUES(varID, varName, varLoc, varDept);

END TRG_TEST_INST_OF;
```

**DDL**

**Data Definition Language** (DDL) statements are used to define the database structure or schema. Some examples:

- CREATE - to create objects in the database
- ALTER - alters the structure of the database
- DROP - delete objects from the database

- TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
- COMMENT - add comments to the data dictionary
- RENAME - rename an object

**DML**

**Data Manipulation Language** (DML) statements are used for managing data within schema objects. Someexamples:

- SELECT - retrieve data from the database
- INSERT - insert data into a table
- UPDATE - updates existing data within a table
- DELETE - deletes all records from a table, the space for the records remain
- MERGE - UPSERT operation (insert or update)
- CALL - call a PL/SQL or Java subprogram
- EXPLAIN PLAN - explain access path to data
- LOCK TABLE - control concurrency

**DCL**

**Data Control Language** (DCL) statements. Some examples:

- GRANT - gives user's access privileges to database
- REVOKE - withdraw access privileges given with the GRANT command

**TCL**

**Transaction Control** (TCL) statements are used to manage the changes made by DML statements. It allowsstatements to be grouped together into logical transactions.

- COMMIT - save work done
- SAVEPOINT - identify a point in a transaction to which you can later roll back
- ROLLBACK - restore database to original since the last COMMIT
- SET TRANSACTION - Change transaction options like isolation level and what rollback segment to use