Data:

* Facts that can be recorded.

**UNIT - I**

* + It is raw or unprocessed form.
  + You cannot take decision based on data.
  + Data has no meaning.

Example(s):

* Text, numbers, images, videos etc….
* 25, Karthik, Karimnagar.

**Information:** When data is organized, processed in a given context. Example(s):

Data

Data

Data

**Information**

Processing

Example(s):

* The age of a karthik is 25.
* 505467  pincode
* Marks/result of a class is a data.

Pass percentage of that class is information.

Has a meaning/ meaningfull

**Database (db):** Collection of similar or related data or

The related information when placed in an organized form makes a database or organized collection of related information is known as database.

Example:

* A Book can be treated as a database, we will pen to manipulate (Manually).
* Dictionary, University (System).

Various types of databases can be Traditional databases (text & numbers), Multimedia databases (audio, video, movies, speech), Geographically Information System (satellite images), Real time databases (store).

**Data ware house:** Data ware house is a kind of database in which the data is going to be very huge and historical.

**Example:** Data stored about past 100 years of a company .stock market rates.

**\*\*\*Good decisions require good information that is derived from raw facts (Data).**

**Database Management systems:** It is the software system that allows the user to define and maintain the database and provide control access to the database or

DBMS is a collection of interrelated data and a set of programs to access data. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.

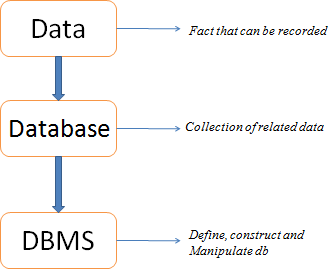
\*\* DBMS is a software or set of programs to define, Construct and Manipulate database

i.e it helps us to add, modify, delete data in the database.

 DBMS is an intermediate layer between programs and data.

**Database(Book) + Database Management Systems(Pen)=Database Systems**

Summary:



**Some of the Database Management systems are:**

* + Oracle, MySQL, MongoDB, Microsoft SQL server, PostgreSQL, NoSQL, Informix, DB2, Sybase etc….

Functionalities:

1. **Define:** Specifying the data type, structure and constraints for the data to be stored.
2. **Construct:** Process of storing data on some storage medium.
3. **Manipulate:** Querying the database to retrieve specific data, updating database and generating reports.
4. **Share:** Allows multiple users and programs to access the database concurrently.

* Others functionalities can be protection and maintenance.

**Database System Application:**

1. **Banking:** For customer information, accounts, loans, and banking transactions.
2. **Airlines:** For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
3. **Universities:** For student information, course registration and grades.
4. **Credit card transactions:** For purchase on credit cards and generations of monthly statements.
5. **Telecommunications:** For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.
6. **Finance:** For storing information about sales, and purchases of financial instruments such as stocks and bonds also for storing real-time market data to enable on-line trading by customers and automated trading by firm.
7. **Railway Reservation System:** Database is required to keep record of ticket booking, train’s departure and arrival status. Also if trains get late then people get to know it through database update.
8. **Social media sites:** We all are on social media websites to share our views and connect with our friends. Daily millions of users signed up for these social media accounts like facebook, whatsapp, etc. But how all the information of users are stored and how we become able to connect to other people , yes this is all because DBMS.

A Historical Perspective:

**Manual System – 1950s:**

* + Data was stored as paper records.
  + Huge man power involved.
  + Unnecessary time was wasted like when searching for a particular record.
  + This was inefficient.

1950s and Early 1960s:

* + Data processing using magnetic tapes for storage.
  + Tapes provided only sequential access.
  + Data stored in files known as File based System.

1960s:

* + Charles Bachman designed first general – purpose DBMS, was called integrated data store.
  + IDS formed the basis for the network data model.

Late 1960s:

* + IBM (International Business Machines) developed the Information Management System (IMS).
  + IMS formed the basis for an alternative data representation framework called the hierarchical data model.

1970:

* + In 1970s, Edgar Codd, at IBM research Laboratory, proposed a new data representation framework called relational data model

1980s:

* + The relational model consolidated its position as the dominant DBMS paradigm, and database systems continued to gain widespread use,
  + The SQL query language for relational databases, developed as part of IBM’s project, is now the standard query language.

late 1980s and 1990s:

* + Several vendors (e.g IBMS’s DB2, Oracle 8, Informix) extended their systems with the ability to store new data types such as images, videos and to ask more complex queries.
  + Specialized systems have been developed by numerous vendors for creating data warehouses.

Present era:

* + DBMS are entered the Internet age. While the first generation of websites stored their data in operating system files, the use of a database accessed through a web browser is becoming wide spread.
  + Queries are generated through web-accessible forms and answers are formatted using a markup language such as HTML.

File Systems vs a DBMS Drawbacks of File Systems:

1. **Data Redundancy:**

There are no methods to validate the insertion of duplicate data in the system. Any user can enter any data. File system doesn’t validate for the kind of data being entered nor doesn’t validate for previous existence of the same data in the same file. Duplicate data in the system is not appreciated as it is a waste of space and always lead to confusion and misleading of data. When there are duplicate data in the file and if we need to update or delete the record, we might end up in updating / deleting one of the records leaving the other record in the file.

1. Data inconsistency:

For Example student and student\_report files have student’s address in it, and there was a change request for one particular student’s address. The program search only student file for the address and updated it correctly. There is another program which prints the student’s report and mails it to the address mentioned in the student\_report file. There is a mismatch in the actual address and his report is sent to his old address. This mismatch in different copies of same data is called data inconsistency.

1. Data Isolation:

Imagine we have to generate a single report of student, who is studying in particular class, his study report, his library book details and hostel information. All these information are stored in different files. How do we get all these details in one report? We have to write a program. But before writing the program , the programmer should find out which all files have the information needed, what is the format of each file, how to search data in each file etc. Once all these analysis is done, he writes a program. If there is 2-3 files involved, programming would be bit simple. Imagine if there is lot many files involved in it? It would require lot of effort for the programmer. Since all the data are isolated from each other in different files, programming becomes difficult.

1. Security:

Each file can be password protected. But what if we have to give access to only few records in the file? For Example, user has to be given access to view only their bank account details in the file. This is very difficult in the file system.

1. Integrity:

If we need to check for certain insertion criteria while entering the data in to file, it is not possible directly. We can do it writing program, say, if we have to restrict the students above age 18 then it is by means program alone. There is no direct checking facility in the file system. Hence these of integrity checks are not easy in file system.

1. Atomicity:

If there is any failure to insert, update, or delete in the file system, there is no mechanism to switch back to the previous state. Consider a program to transfer 500 from the account of A to the account of B. If a system failure occur during the execution of the program, it is possible that the 500 as removed from the account A but was not credited to the account B, resulting in an inconsistent database state. It is essential to database consistency that either both the credit and debit occur, or that neither occur i.e the funds transfer must be atomic-it must happen in its entirely or not at all. It is difficult to ensure atomicity in a file processing system.

1. Limited data sharing:

Data are scattered in various files also different files may have different formats and these files may be stored in different folders may be of different departments. So, due to this, it is difficult to share data among different applications.

1. Data Mapping and Access:

Although all the related information is grouped and stored in different files, there is no mapping between any two files. i.e.; any two dependent files are not linked. Even though Student files and Student\_Report files are related, they are two different files and they are not linked by any means. Hence if we need to display student details along with his report, we cannot directly pick from those two files. We have to write a lengthy program to search the Student file first, get all details, then go Student\_Report file and search for his report. When there is a very huge amount of data, it is always a time-consuming task to search for particular information from the file system. It is always an inefficient method to search for the data.

|  |  |  |
| --- | --- | --- |
| **S.No** | **File System** | **DBMS** |
| 1 | Suitable for small organization. | Suitable for both small and large organization. |
| 2 | It can have unstructured data | Can have structured data. |
| 3 | Storing and retrieving of data can't be done efficiently in a file system. | DBMS is efficient to use as there are a wide variety of methods to store and retrieve data. |
| 4 | The file system doesn't have a crash recovery mechanism. | DBMS provides a crash recovery mechanism |
| 5 | Protecting a file system is very difficult. | DBMS offers good protection mechanism. |
| 6 | In a file management system, the redundancy of data is greater. | The redundancy of data is low in the DBMS system. |
| 7 | Data inconsistency is higher in the file system. | Data inconsistency is low in a database management system. |
| 8 | It doesn't offer backup and recovery of data if it is lost. | DBMS system provides backup and recovery of data even if it is lost. |
| 9 | There is no efficient query processing in the file system. | You can easily query data in a database using the SQL language. |
| 10 | This system doesn't offer concurrency. | DBMS system provides a concurrency facility. |
| 11 | Data cannot be shared | Data can be shared. |

Data Models:

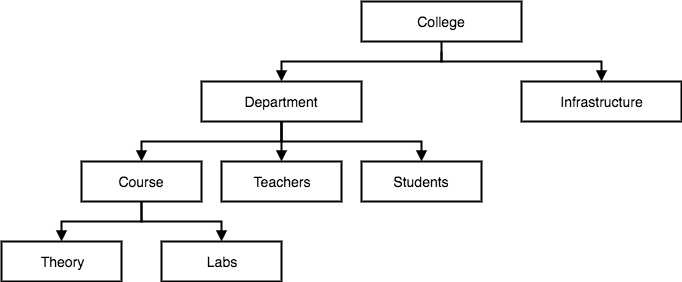
* + Planning the structure of data in database is called Data Models.
  + A data model is a collection of high level data description construct that hide many low-level storage details. A DBMS allows a user to define the data to be stored in terms of a data model.

Data Models are:

1. Hierarchical Model.
2. Network Model
3. Entity – Relationship Model
4. Relational Model

Hierarchical Model:

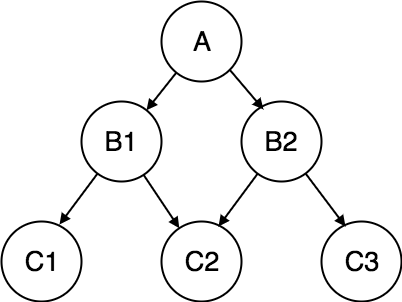
This database model organizes data into a tree-like-structure, with a single root, to which all the other data is linked. The hierarchy starts from the **Root** data, and expands like a tree, adding child nodes to the parent nodes. In this model, a child node will only have a single parent node. This model efficiently describes many real-world relationships like index of a book, recipes etc. It has one-to-many relationship between two different types of data, for example, one department can have many courses, many professors and many students.



Network Model:

This is an extension of the Hierarchical model. In this model data is organized more like a graph, and are allowed to have more than one parent node.

In this database model data is more related as more relationships are established in this database model. Also, as the data is more related, hence accessing the data is also easier and fast. This database model was used to map many-to-many data relationships.

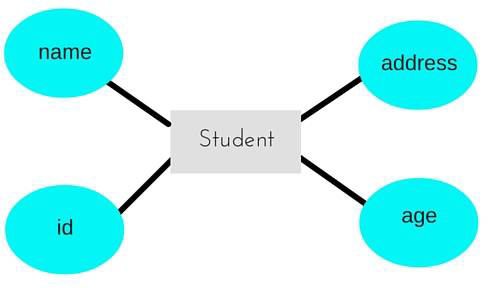


Entity – relationship Model:

E-R Models are defined to represent the relationships into pictorial form to make it easier for different stakeholders to understand.

This model is good to design a database, which can then be turned into tables in relational model.

Let's take an example, If we have to design a School Database, then **Student** will be an **entity** with **attributes** name, age, address etc. As **Address** is generally complex, it can be another **entity** with **attributes** street name, pincode, city etc, and there will be a relationship between them.



Relational Model:

In this model, data is organised in two-dimensional **tables** and the relationship is maintained by storing a common field.

This model was introduced by E.F Codd in 1970, and since then it has been the most widely used database model, infact, we can say the only database model used around the world.

The basic structure of data in the relational model is tables. All the information related to a particular type is stored in rows of that table.

Hence, tables are also known as **relations** in relational model.

|  |  |  |
| --- | --- | --- |
| ***Student\_id*** | ***Name of the student*** | ***Subject\_name*** |
| 501 | Raju | Java |
| 502 | Ravi | dbms |

Demerits of database Systems:

1. **Complex design:** Database design is complex, difficult and time consuming.
2. **Hardware and software cost:** Large amount of invest is needed to setup the required hardware or to repair software failure.
3. **Damaged part:** If one part of database is corrupted or damaged, then entire database may get affected.
4. **Conversion cost:** If the current system is in conventional file system and if we need to convert it to database system then large amount of cost is incurred in purchasing different tools, and adopting different techniques as per the requirement.
5. **Training:** For designing and maintaining the database systems, the people need to be trained.

Levels of Abstraction in a DBMS

Data abstraction is a process of hiding the implementation details (such as how the data are stored and maintained) and representing only the essential features to simplify user's interaction with the system.

To simplify user's interaction with the system, the complexity is hidden from the database users through several levels of abstraction.

**Three levels of data abstraction are:**

1. View Level / External Schema
2. Conceptual Level / Logical Schema
3. Physical Level / Internal Schema



View Level:

* Highest level of abstraction.
* Describes only part of the database for a particular group of users.
* Can be many different views of a database.

***Example:*** If we have a login-id and password in a university system, then as a student, we can view our marks, attendance, fee structure, etc. But the faculty of the university will have a

different view. He will have options like salary, edit marks of a student, enter attendance of the students, etc. So, both the student and the faculty have a different view. By doing so, the security of the system also increases. In this example, the student can't edit his marks but the faculty who is authorized to edit the marks can edit the student's marks. Similarly, the dean of the college or university will have some more authorization and accordingly, he will have his view. So, different users will have a different view according to the authorization they have.

Logical / Conceptual Level:

* Next highest level of abstraction.
* Describes **what** data are stored and what relationships exit among those data.
* Database administrator level.

***Example***: Let us take an example where we use the relational model for storing the data. We have to store the data of a student, the columns in the student table will be student\_name, age, mail\_id, roll\_no etc. We have to define all these at this level while we are creating the database. Though the data is stored in the database but the structure of the tables like the student table, teacher table, books table, etc are defined here in the conceptual level or logical level. Also, how the tables are related to each other is defined here. Overall, we can say that we are creating a blueprint of the data at the conceptual level.

Physical Level:

* Lowest level of abstraction.
* Describes **how** the data are stored.
* Complex low-level data structures described in detail.
* E.g: index, B-tree, hashing.

It tells the actual location of the data that is being stored by the user. The Database Administrators (DBA) decides that which data should be kept at which particular disk drive, how the data has to be fragmented, where it has to be stored etc. They decide if the data has to be centralized or distributed. Though we see the data in the form of tables at view level the data

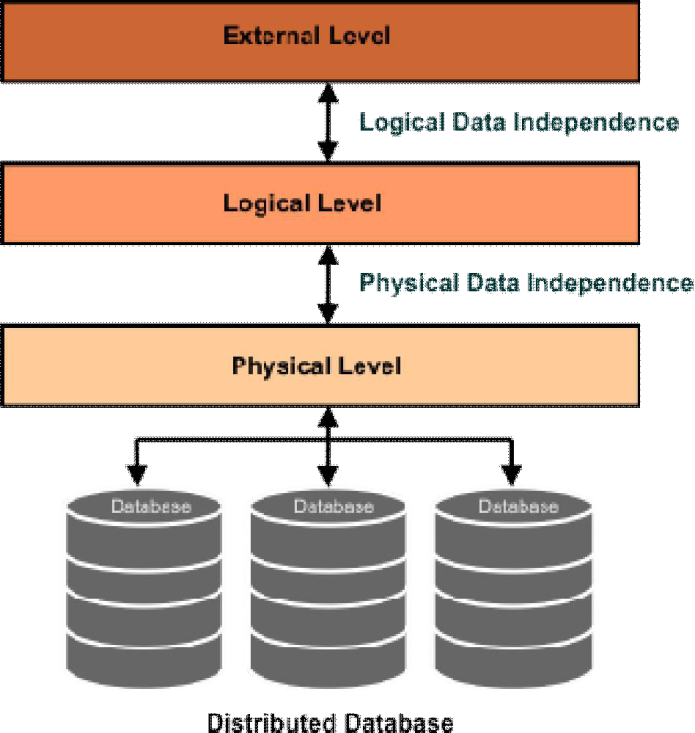
here is actually stored in the form of files only. It totally depends on the DBA, how he/she manages the database at the physical level.

Data Independence

Data independence is ability to modify a schema definition in one level without affecting a schema definition in the next higher level. or Capacity of changing the schema at one level without effecting the another level.

**There are two levels of data independence:**

1. Physical Data Independence
2. Logical Data Independence



1. **Physical Data Independence:**

Physical Data Independence refers to the characteristic of changing the physical level without affecting the logical level or conceptual level. Using this property we can easily change the **storage device** of the database without affecting the logical schema.

**Examples of changes under Physical Data Independence**

Due to Physical independence, any of the below change will not affect the conceptual layer.

* + Using a new storage device like Hard Drive or Magnetic Tapes
  + Switching to different data structures.
  + Changes to compression techniques or hashing algorithms.
  + Change of Location of Database from say C drive to D Drive

1. **Logical Data Independence:**

It refers to the characteristics of changing the logical level without affecting the external or view level. This also helps in separating the logical level from the view level. If we do any changes in the logical level then the **user view** of the data remains **unaffected**. The changes in the logical level are required whenever there is a change in the logical structure of the database.

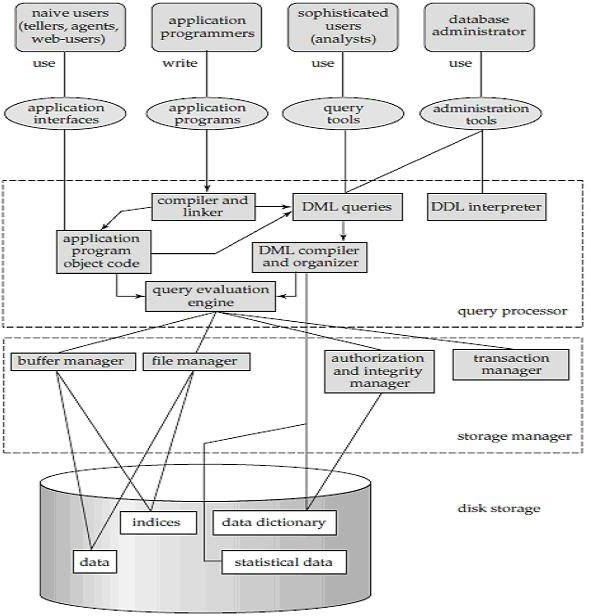
**Examples of changes under Logical Data Independence**

Due to Logical independence, any of the below change will not affect the external layer.

1. Add/Modify/Delete a new attribute, entity or relationship is possible without a rewrite of existing application programs
2. Merging two tables into one
3. Breaking an existing table into two or more tables

Structure of a DBMS

* DBMS (Database Management System) acts as an interface between the user and the database. The user requests the DBMS to perform various operations such as insert, delete, update and retrieval on the database.
* The components of DBMS perform these requested operations on the database and provide necessary data to the users.



Naive users:

Any user who does not have any knowledge about database can be in this category. There task is to just use the developed application and get the desired results. For example: Clerical staff in any bank is a naive user. They don’t have any dbms knowledge but they still use the database and perform their given task.

Application programmers:

As its name shows, application programmers are the one who writes application programs that uses the database. These application programs are written in programming languages like PHP, Java etc. These programs meet the user requirement and made according to user requirements. Retrieving information, creating new information and changing existing information is done by these application programs.

Sophisticated Users:

They are database developers, who write SQL queries to select/insert/delete/update data. They do not use any application or programs to request the database. They directly interact with the database by means of query language like SQL. These users will be scientists, engineers, analysts who thoroughly study SQL and DBMS to apply the concepts in their requirement.

Database Administrators:

The life cycle of database starts from designing, implementing to administration of it. A database for any kind of requirement needs to be designed perfectly so that it should work without any issues. Once all the design is complete, it needs to be installed. Once this step is complete, users start using the database. The database grows as the data grows in the database. When the database becomes huge, its performance comes down. Also accessing the data from the database becomes challenge. There will be unused memory in database, making the memory inevitably huge. These administration and maintenance of database is taken care by database Administrator (DBA).A DBA has many responsibilities. A good performing database is in the hands of DBA.

* **Installing and upgrading the DBMS Servers:**

DBA is responsible for installing a new DBMS server for the new projects. He is also responsible for upgrading these servers as there are new versions comes in the market or requirement. If there is any failure in up gradation of the existing servers, he should be able revert the new changes back to the older version, thus maintaining the DBMS working.

* **Design and implementation:**

Designing the database and implementing is also DBA’s responsibility. He should be able to decide proper memory management, file organizations, error handling, log maintenance etc for the database.

* **Performance tuning:**

Since database is huge and it will have lots of tables, data, constraints and indices, there will be variations in the performance from time to time. Also, because of some designing issues or data growth, the database will not work as expected. It is responsibility of the DBA to tune the database performance. He is responsible to make sure all the queries and programs works in fraction of seconds.

* **Migrate database servers:**

Sometimes, users using oracle would like to shift to SQL server or MySQL. It is the responsibility of DBA to make sure that migration happens without any failure, and there is no data loss.

* **Backup and Recovery:**

Proper backup and recovery programs needs to be developed by DBA and has to be maintained him. This is one of the main responsibilities of DBA. Data/objects should be backed up regularly so that if there is any crash, it should be recovered without much effort and data loss.

* **Security:**

DBA is responsible for creating various database users and roles, and giving them different levels of access rights.

* **Documentation:**

DBA should be properly documenting all his activities so that if he quits or any new DBA comes in, he should be able to understand the database without any effort. He should basically maintain all his installation, backup, recovery, security methods. He should keep various reports about database performance.

**Data Interpreter:** It interprets DDL statements and records them in tables containing metadata.

**DML Compiler:** The DML commands such as insert, update, delete, retrieve from the application program are sent to the DML compiler for compilation into object code for database access. The object code is then optimized in the best way to execute a query by the query optimizer and then send to the data manager.

**Query Evaluation Engine:** It executes low-level instructions generated by the DML compiler.

**Buffer Manager:** It is responsible for retrieving data from disk storage into main memory. It enables the database to handle data sizes that are much larger than the size of main memory.

**File Manager:** It manages the allocation of space on the disk storage and the data structure used to represent information stored on disk.

**Authorization and Integrity Manager:** Checks the authority of users to access data and satisfaction of the integrity constraints.

**Transaction Manager:** It ensures that the database remains in a consistent state despite the system failures and that concurrent transaction execution proceeds without conflicting.

**Data Files:** Which store the database itself.

**Data dictionary:** Data Dictionary, which stores metadata about the database, in particular the schema of the database such as names of the tables, names of attributes of each table, length of attributes, and number of rows in each table.

**Indices:** An index is a small table having two columns in which the 1st column contains a copy of the primary or candidate key of a table and the second column a set of pointers holding the address of the disk block.

**Statistical data:**

These statistics provide the optimizer with information about the state of the tables that will be accessed by the SQL statement that is being optimized. The types of statistical information stored in the system catalog include:

* Information about tables including the total number of rows, information about compression, and total number of pages.
* Information about columns including number of discrete values for the column and the distribution range of values stored in the column.
* Information about table spaces including the number of active pages.
* Current status of the index including whether an index exists or not, the organization of the index (number of leaf pages and number of levels), the number of discrete values for the index key, and whether the index is clustered.
* Information about the table space and partitions.

ER Diagrams

ER diagram or **Entity Relationship diagram** is a conceptual model that gives the graphical representation of data or it is a visual representation of data that describes how data is related to each other.

An ER diagram is mainly composed of following three components

1. Entity sets
2. Attributes
3. Relationship sets

**Entity:** An entity is a thing or an object in the real world that is distinguishable from other based on the value of the attribute is posses. (Any noun can be called as Entity)

Types of Entities:

**Tangible / concrete:** Entities which physically exist in real world. Example: car, pen, book etc.

**Intangible / abstract:** Entities which exists logically. Example: Account.

**Entity set:** Entity set is a group of similar entities that share the same properties i.e it represents schema / structure.

PERSON (name, age, address) Entity set

(Raju, 26, knr) Entity

* Entity cannot be represented in an ER diagram as it is instance / data.
* Entity set is represented by rectangle in ER diagram.

Symbol:

**Student**

**Attributes:** Attributes are the units that describe the characteristics / properties of entities.

* For each attribute there is a set of permitted values called domains.
* In ER Diagram, represented by ellipse or oval.

Types of Attributes:

**Simple Attributes:** Simple attribute cannot be divided further, represented by simple oval.

Example:

**Name**

DOB

**Student**

**Composite Attribute:** Composite attribute can be further divided in simple attribute, represented by oval connected to an oval.

Example:

**L\_name**

**F\_name**

**Name**

**Roll\_No**

**Student**

**Single attribute:** Single attribute can have only one value at an instance of time. Represented by oval

Example:

**DOB**

**Student**

**Multi-valued attribute:** Multi-valued attribute can have more than one value at an instance of time.

Represented by double oval.

Example:

**Phne\_No**

**Student**

**Stored attribute:** Stored attribute is an attribute which are physically stored in the database.

Example:

**DOB**

**Student**

**Derived Attribute:** Derived attribute are the attributes that do not exist in the physical database, but their values are derived from other attributes present in the database.

Represented by dotted oval. Example:

**DOB**

age

**Student**

**Complex Attribute:** Complex Attribute is a type of attribute in database. It is formed by nesting composite attributes and multi-valued attributes.

**Example:** A Person can have more than one address (multivalued) and address can have city, pin code, country (composite).

**Relationship:** Relationship is an association between two or more entities of same or different entity set. (Any verb can be treated as Relationship)

* No representation in ER diagram as it is an instance or data.

T1

T2 T3

.

.

.

.

tn

S1

S2 S3

.

.

.

sn

**Relationship type / set:** Relationship set is a set similar type of relationship.

* In ER Diagram it is represented using diamond symbol.

Example:

**Student**

**teaches**

**Teacher**

**A Relationship may also have attributes called descriptive attributes. Example:**



**Since**

**works**

**for**

**Employee**

**Department**

**Case Study:**

**Requirement Analysis:** Every employee works for exactly a department and a department can have many employees. New department need not have any employee.

employee Works for department

E1**.**

E2**.**

E3**.**

E4**.**

E5**.**

E6**.**

.

.

.

.

.

.

.

D1

D2

D3 D4

**Degree:** Number of entities that are participating in a relationship. In the above example degree is 2.(binary relationship)

**Cardinality ratio / Mapping cardinalities:** Cardinality ratio is the max number of relationships in which an entity can participate.

In the above diagram e1 can participate only one relationship i.e 1. Department is participating more than one relationship i.e N.

**Participation or existence:** Minimum number of relationships in which an entity can participate, sometimes it is also called as min – cardinality.

In the above figure e1 can participate min 1. Department can participate 0.

**Converting above set theory diagram into ER diagram.**

**Works**

**for**

1

**Department**

|  |  |  |
| --- | --- | --- |
| **employe** | N | |
|  |  |
|  | |

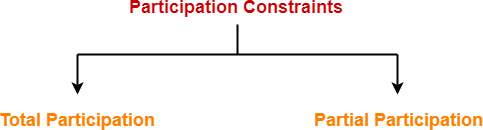
**Figure - 1**

|  |  |  |
| --- | --- | --- |
| Rectangles | Represents entity sets |  |
| Ellipses | Represents attributes |  |
| Diamonds | Represents relationships |  |
| Lines | Attributes to entity sets & entity sets to relationships |  |
| Double ellipses | Represents multivalued |  |
| Dashed ellipses | Denotes derived attributes |  |
| Doubles lines | Which indicates total participation of an entity in a relationship sets |  |
| Double rectangles | Represents weak entity sets |  |

Additional Features of ER Model:

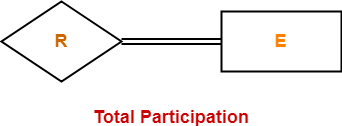
1. **Participation constraints**

There are two types of participation constraints-



**Total participation:** If all the entities are participating in a relationship then it is called total participation. It is also called as mandatory participation.

Total participation is represented using a double line between the entity set and relationship set.



**Example:**

**Works**

**for**

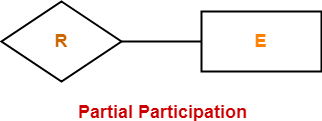
1

**Department**

**Employee**

**Partial participation:** It specifies that each entity in the entity set may or may not participate in the relationship instance in that relationship set. It is also called as optional participation.

Partial participation is represented using a single line between the entity set and relationship set.



**Example:**

N

**Works**

**for**

1

**Department**

**employee**

1. **Cardinality ratio / Mapping Cardinalities / Key constraints: It** defines the maximum number of relationship instances in which an entity can participate.

There are 4 types of key constraints

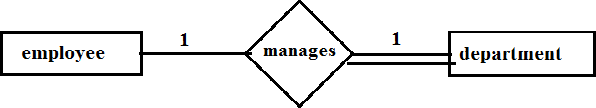
* + **One to one**
  + One to many
  + **Many to one**
  + Many to many

1. **One to one:**

An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.

**Example:** Every dept should have a hod and only one employee manages a dept and an employee can manage only one dept.





1. **Many to one:** An entity in A is associated with at most one entity in B. An entity in B, however can be associated with any number (zero or more) of entities in A.

**Example:** Every employee works for exactly a department and a department can have many employees. New department need not have any employee.

Employee works for department

E1**.**

E2**.**

E3**.**

E4**.**

E5**.**

E6**.**

.

.

.

.

.

.

.

D1

D2

D3 D4

N

**Works**

**for**

1

**Department**

**employee**

1. **One to many:** An entity in A is associated with any number (zero or more) of entities in B. An entity in B, can be associated with at most one entity in A.

**1**

**N**

**Has**

**Dept**

**Employee**

1. **Many to many:** An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A.

**Example:** Every emp is supported to work at least one project and he can work on many projects, a project is supported to have many emp and a project should have at least one emp.

Employee works for project

**E1 .**

**E2 .**

**E3 .**

**E4 .**

**E5 .**

**E6 .**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**.**

**P1**

**P2** **P3**

**P4**

**N**

**M**

**Works**

**on**

**Employee**

**project**

**Key:** Key is an attribute or collection of attributes that uniquely indentifies an entity among entity set. For example the roll number of a student makes him or her identifiable among students.

**Name**

**Roll No**

Student

**Strong and weak entities:**

**Strong entity:** An entity type is strong if its existence does not depend on some other entity type. Such entity is called Strong Entity.

* A strong entity set is an entity that contains sufficient attributes to uniquely identify all its entities.
* An entity set which has a key is called strong entity set.

Representation

* The strong entity is represented by a single rectangle.
* Relationship between two strong entities is represented by a single diamond.

Example:



* Consider the ER diagram which consists of two entities student and course
* Student entity is a strong entity because it consists of a key called student id which is enough for accessing each record uniquely
* The same way, course entity contains of course ID attribute which is capable of uniquely accessing each row it is each course details

Weak Entity:

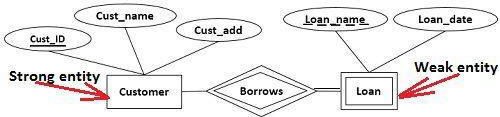
A weak entity is an entity set that does not have sufficient attributes for Unique Identification of its records. Simply a weak entity is nothing but an entity which does not have a key attribute.

* It contains a partial key called as discriminator which helps in identifying a group of entities from the entity set**.**
* Discriminator is represented by underlining with a dashed line.

Representation

* A double rectangle is used for representing a weak entity set.
* The double diamond symbol (Identifying relationship) is used for representing the relationship between a strong entity and weak entity which is known as identifying relationship.
* Double lines are used for presenting the connection with a weak entity set with relationship.

Example:



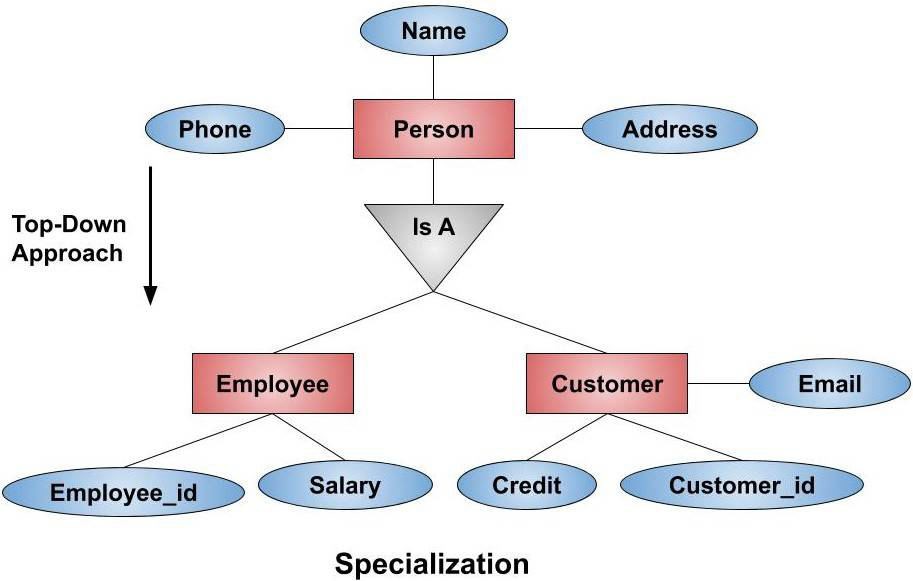
**Note:**

* Without strong entity there is no weak entity.
* If it is weak entity then it must be total participation.
* But total participation doesn’t make an entity weak.

Class Hierarchies / Extended features of ER Model.

**Specialization:** Specialization means creating new subclass from an existing class. If it turns out that certain attributes only apply to some of the object class sub class can be created.

* Top – down design approach.
* Converting high – level entity into low – level entity by converting some adding some attributes.
* Low – level entity inherits all the attributes from high – level entity.



Example:

If we have a person entity type who has attributes such as *Name, Phone\_no, Address*. Now, suppose we are making software for a shop then the person can be

of two types. This Person entity can further be divided into two entity i.e. Employee and Customer. How we will specialize it? We will specialize the Person entity type to Employee entity type by adding attributes like *Emp\_no and Salary*. Also, we can specialize the Person entity type to Customer entity type by adding attributes like *Customer\_no, Email, and Credit.* These lower-level attributes will inherit all the properties of the higher-level attribute. The Customer entity type will also have attributes like *Name, Phone\_no, and Address* which was present in the higher-level entity.

Generalization:

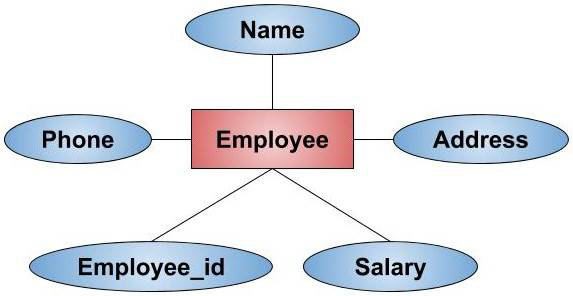
It is the process of extracting shared characteristics from two or more classes and combining them into a generalized super class, shared characteristics can be attributes.

* Reverse of Specialization
* Bottom – up approach.

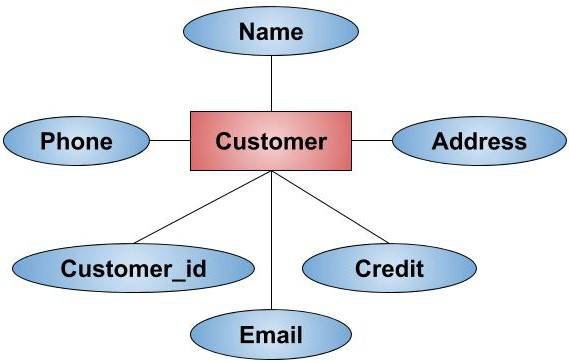
***Example:*** Suppose we have two entity types, Employee and Customer.

The attributes of Employee entity type are *Name, Phone, Salary, Employee\_id,* and

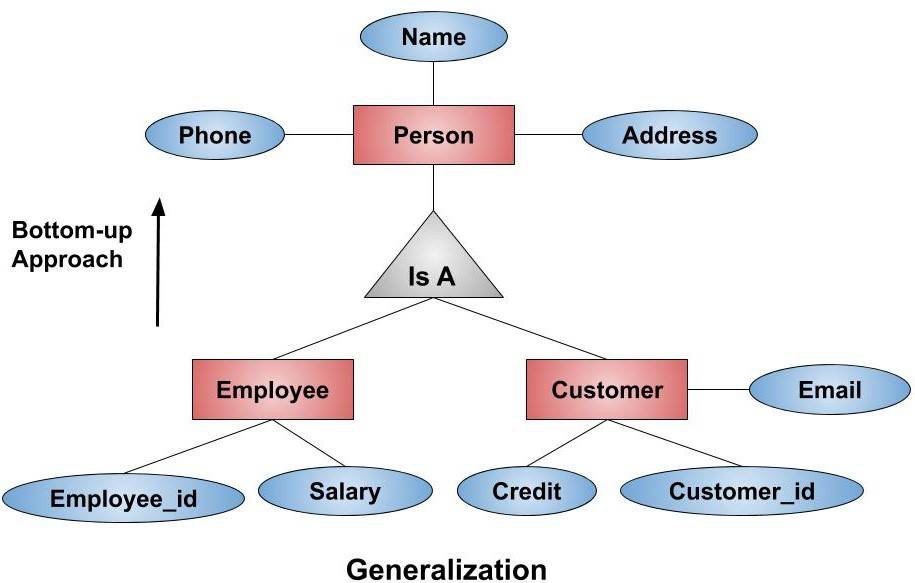
*Address*.



The attributes of Customer entity type are *Name, Phone, Address, Credit, Customer\_id,* and *Email.*



We can see that the three attributes i.e. *Name, Phone,* and *Address* are common here. When we generalize these two entities, we form a new higher-level entity type Person. The new entity type formed is a generalized entity. We can see in the below E-R diagram that after generalization the new generalized entity Person contains only the common attributes i.e. *Name, Phone,* and *Address*. Employee entity contains only the specialized attribute like *Employee\_id* and *Salary*. Similarly, the Customer entity type contains only specialized attributes like *Customer\_id, Credit,* and *Email.* So from this example, it is also clear that when we go from bottom to top, it is a Generalization and when we go from top to bottom it is Specialization. Hence, we can say that Generalization is the reverse of Specialization.

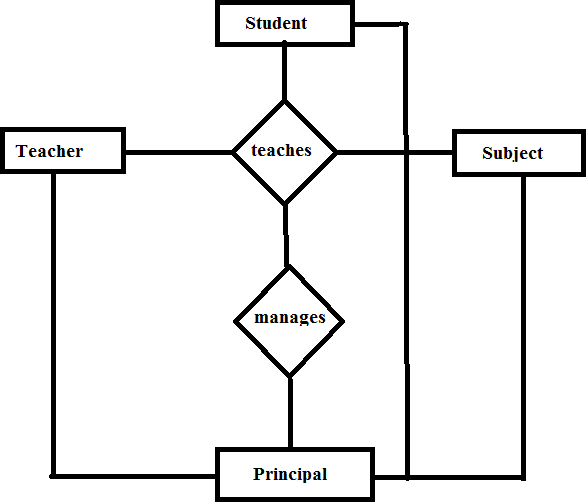


**Aggregation:** Aggregation is the process of merging. It completely different from generation. In generation, we merge entities of same domain into one entity; In this case we merge related entities into one entity.

* We cannot express relationship among relationship Representation:

This is indicated on an ER Diagram by drawing a dashed box around the aggregation.

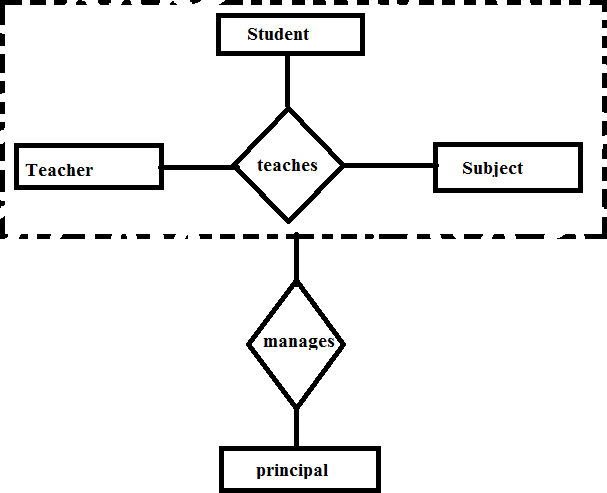
Example:



**Before aggregation**

Drawbacks:

1. Relationship among relation not possible.
2. Redundant data.



After aggregation

**Conceptual Design with ER Model**

Developing the ER diagram presents several choices, including the following:

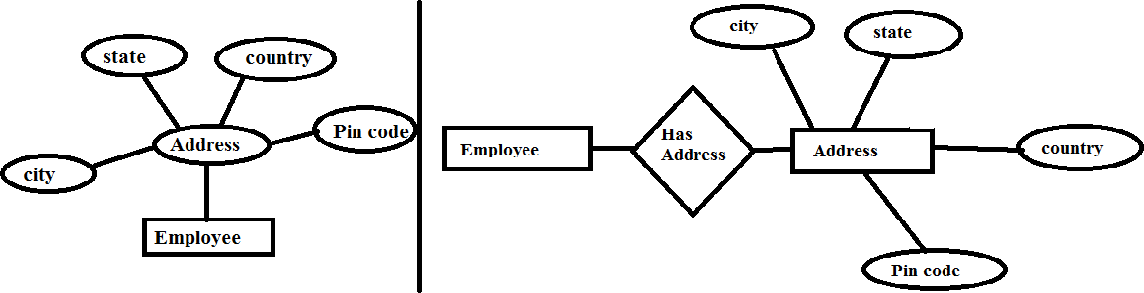
* Should a concept be modeled as an entity or an attribute?
* Should a concept be modeled as an entity or a relationship?
* What are the relationship sets and their participating entity sets? Should we use binary or ternary relationships?
* Should we use aggregation?

Entity versus Attribute

While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as an attribute or as an entity set. For example, consider adding address information to the Employees entity set. One option is to use an attribute address. This option is appropriate if we need to record only one address per employee, an alternate is to create an entity set called Addresses and to record associations between employees and address using a relationship (say, Has\_address). This more complex alternative is necessary in two situations.

 We have to record more than one address for an employee.

 By representing an address as an entity with these attributes (such as city, state, country, and Pin code) we can support queries such as “Find all employees with an address in Hyderabad”.



Address as attribute Address as entity set

UNIT - II

**Relational Model**

Relational Model was proposed by E.F Codd in 1970. At that time, most database systems were based on one of two older data models (hierarchical mode and network model).

Relational model were designed to model data in the form of relations or tables. After designing the conceptual model of database using ER diagram, we need to convert the conceptual model in the relational model which can be implemented using any RDBMS languages like MySQL, Oracle etc.

A database is a collection of one or more relations, where each relation is a table with rows and columns.

A relational model represents how data is stored in relational database. A relational database stores data in the form of relations (tables).

**Advantages:**

* The relational model is very simple and elegant.
* This simple tabular representation enables even novice users to understand the contents of a database and it permits the use of simple, high-level languages to query the data.
* The major advantages of the relational model over the older data models are its simple data representation and the ease with which even complex queries can be expressed.

**Relation:** The main construct for representing data in the relational model is a relation. A Relation consists of a *relation schema* and *relation instance*.

**Relation schema:** Relation schema specifies / represents the name of the relation, name of each field (or column or attribute), and the domain of each field.

The set of permitted values for each attribute is called domain” or “A domain is referred to in a relation schema by the attribute name and has a set of associated values”

**Example:**

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

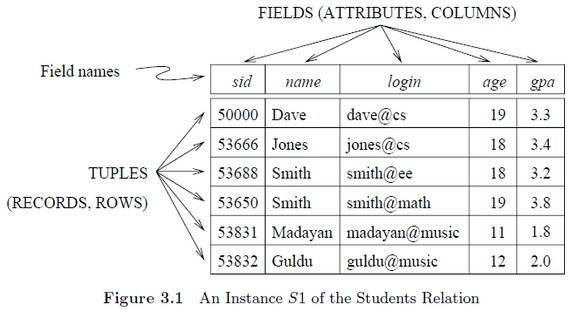
Here field name *sid* has a domain named *string.* The set of values associated with domain string is the set of all character strings.

**Relation instance:**

An instance of a relation is a set of tuples at a particular instance of time, also called records, in which each tuple has the same number of fields as the relation schema. A relation instance can be thought of as a table in which each tuple is a row, and all rows have the same number of fields. (The term relation instance is often abbreviated to just relation).

It can change whenever there is insertion, deletion, updation in the database.

**Example:**



**Degree:** Number of attributes in the relation in known as degree of relation. Student relation defined above as degree 5.

*Degree is also known as arity.*

**Cardinality:** Cardinality of a relation instance is the number of tuples. In the above student relation, the cardinality is six.

**Integrity constraints over Relations:**

An integrity constraint (IC) is a condition on a database schema and restricts the data that can be stored in an instance of the database.

Integrity constraints are a set of rules. It is used to maintain the quality of information.

Integrity constraints ensure that changes (update deletion, insertion) made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database.

Various types of integrity constraints are –

1. Domain constraints
2. General constraints
3. Key constraints
4. Foreign key constraints.

**Domain constraints:** Domain integrity means the definition of a valid set of values for an attribute. The values that appear in a column must be drawn from the domain associated with that column. The domain of a field is essentially the type of that filed.

Example:

|  |  |  |
| --- | --- | --- |
| **Roll Number** | **Name** | **age** |
| **101** | Raju | 50 |
| **102** | Ravi | 30 |
| **104** | Ramu | A |

Not allowed because age is integer

General Constraints:

Constraints are the rules that are to be followed while entering data into columns of the database table.

Constraints ensure that data entered by the user into columns must be within the criteria specified by the condition

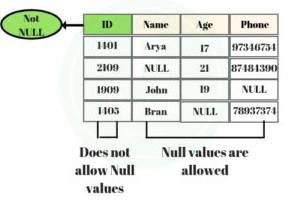
For example, if you want to maintain only unique IDs in the employee table or if you want to enter only age less than 18 in the student table etc.

Some of the constraints are:

* NOT NULL
* DEFAULT
* CHECK

1. **NOT NULL:** Once not null is applied to a particular column, you cannot enter null values to that column and restricted to maintain only some proper value other than null.

Example:



Example:

* + CREATE TABLE test (ID int NOT NULL, name char(50),age int, phone varchar (50));
  + DESC test;
  + INSERT INTO test values(100,"Deepak",30,"9966554744");
  + SELECT \* FROM test;
  + INSERT INTO test VALUES (NULL,"Raju",40,"9966554744");**//Error**

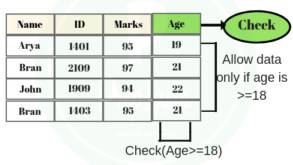
1. DEFAULT:
   * Default clause in SQL is used to add default data to the columns.
   * When a column is specified as default with some value then all the rows will use the same value i.e each and every time while entering the data we need not enter that value.
   * But ***default column value can be customized*** i.e it can be overridden when inserting a data for that row based on the requirement.

Example:

* + DROP TABLE test;
  + CREATE TABLE test(rollno int,name varchar(50) DEFAULT "KITS");
  + INSERT INTO test values(102,"Rajitha");
  + SELECT \* FROM test;
  + INSERT INTO test(rollno) values(103);
  + SELECT \* FROM test;

1. CHECK:
   * Suppose in real-time if you want to give access to an application only if the age entered by the user is greater than 18 this is done at the back-end by using a check constraint.
   * Check constraint ensures that the data entered by the user for that column is within the range of values or possible values specified.

Example:



**Example:**

Drop table test;

CREATE TABLE test(sno int,age int,CHECK (age>=18)); DESC test;

INSERT INTO test values(101,52); SELECT \* FROM test;

INSERT INTO test values(102,10); SELECT \* FROM test;

Key Constraints

**SUPER KEY:** A super key is combination of single or multiple keys which uniquely identifies rows in a table.

**Note:** For any table we have at least one super key which is the combination of all the attributes of the table.

Example:

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Name** | **Phone** | **Age** |
| 101 | Mahesh | 9618810821 | 30 |
| 102 | Ravi | 9160600572 | 28 |
| 103 | Rajesh | 9618749101 | 32 |

* **{ID}:** As no two students will have the same Id, it will help to uniquely access the student details; hence it is a super key.
* **{NAME,ID}:** Even if more than one student has the same name then the combination of name and id will help to recognize the record, as student id will break the tie and this combination is a super key.
* **{PHONE}:** As no two students will have the same phone number**,** a phone number will help to uniquely access the student details and hence phone number is a super key.

No Physical representation in database.



Candidate key:

A set of minimal attribute(s) that can identify each tuple uniquely in the given relation is called as a candidate key. Or

A minimal super key is called as a candidate key.

* It is possible to have multiple candidate keys.
* It is not reducible further.
* A candidate key can never be NULL or empty.
* No physical representation in database.
* Every candidate key is a super key but every super key is not a candidate key.

**Example:**

|  |  |  |
| --- | --- | --- |
| **sid** | **Name** | **marks** |
| s1 | A | 40 |
| s2 | A | 40 |
| s3 | B | 50 |
| s2 | B | 50 |

|  |  |  |
| --- | --- | --- |
| **sid** | **name** | **marks** |
| s1 | A | 40 |
| s2 | B | 20 |
| s3 | A | 20 |
| s4 | C | 30 |

{sid}---- With the help of sid column, we cannot identify a row uniquely, because in sid column, S2 is repeated.

{sid,name}--- With the help of sid and name we can identity a row uniquely So,

{sid,name} is a candidate key.

Explanation: {sid,name} is a super key and proper subset of {sid,name} is not a super key, so {sid,name} is a candidate key.

{sid}--- With the help of sid column, we can identify a row uniquely because all the values are distinct. So, {sid} is a candidate key.

Primary Key:

A primary key is a candidate key that the database designer selects while designing the database. Or

Primary key is a candidate key with no NULL values. Or

A primary key is a column / field or a set of columns / fields that uniquely identifies each row in the table. The primary key follows these rules:

* A primary key must contain unique values. If the primary key consists of multiple columns, the combination of values in these columns must be unique.
* A primary key column cannot have [NULL](https://www.mysqltutorial.org/mysql-null/) values. Any attempt to i[nsert](https://www.mysqltutorial.org/mysql-insert-statement.aspx) or [update](https://www.mysqltutorial.org/mysql-update-data.aspx) NULL to primary key columns will result in an error.
* A table can have one only one primary key.
* Primary key column often has the [AUTO\_INCREMENT](https://www.mysqltutorial.org/mysql-sequence/) attribute that automatically generates a sequential integer whenever you [insert a new row](https://www.mysqltutorial.org/mysql-insert-statement.aspx) into the table.
* It is recommended to use INT or BIGINT data type for the primary key column.

Creating Primary key:

**First way of creating primary key:**

**Syntax:**

CREATE TABLE table\_name (column1 data\_type PRIMARY KEY,

column2 data\_type….);

**Example:**

* CREATE TABLE student (rollno int PRIMARY KEY, name char(50));
* DESC student;
* INSERT INTO student VALUES (101,"Sagar");
* SELECT \* FROM student;
* INSERT INTO student VALUES (102,"Sagar");
* SELECT \* FROM student;
* INSERT INTO student VALUES (101,"Sravanthi"); ***// Error (Duplicate Entry)***
* INSERT INTO student VALUES (NULL, "Santhosh"); ***// Error (NULL cannot be allowed)***

**Second way of creating primary key:**

**Syntax:**

CREATE TABLE table\_name (column1 data\_type, column2 data\_type, PRMARY KEY (column1));

**Example:**

* CREATE TABLE student1 (sid int, name varchar(50),address varchar (50) , PRIMARY KEY(sid));
* DESC student1;
* INSERT INTO student1 VALUES (1,"Deepak","HZB");
* SELECT \* FROM student1;

**Third way of creating primary key:**

**Syntax:**

CREATE TABLE table\_name (column1 data\_type, column2 data\_type, CONSTRAINT constraint\_name PRIMARY KEY (column1));

**Example:**

* CREATE TABLE student2 (rollno int, name char (100), CONSTRAINT roll\_pk PRIMARY KEY (rollno));
* DESC student2;
* INSERT INTO student2 VALUES (1,"Deepak");
* SELECT \* FROM student2;

**Fourth way of creating primary key (ALTER): Syntax:**

ALTER TABLE table\_name ADD PRIMARY KEY (column\_name);

**Example:**

* CREATE TABLE student4 (sid int, name char(50));
* DESC student4;
* ALTER TABLE student4 ADD PRIMARY KEY (sid);
* DESC student4;

**Fifth way of creating primary key(ALTER):**

**Syntax:**

ALTER TABLE table\_name ADD CONSTRAINT constraint\_name PRIMARY KEY (column\_name);

**Example:**

* CREATE TABLE student5 (rollno int, name char (100));
* DESC student5;
* ALTER TABLE student5 ADD CONSTRAINT std\_pk PRIMARY KEY (rollno);
* DESC student5;

**Defining multiple fields as one PRIMARY KEY:**

**Syntax:**

CREATE TABLE table\_name (column1 data\_type, column2 data\_type, PRIMARY KEY (column1, column2));

**Example:**

* CREATE TABLE student6(name char(50),fname char(50),PRIMARY KEY (name,fname));
* DESC student6;
* INSERT INTO student6 VALUES (“Raju”, ”Rajireddy”);
* SELECT \* FROM student6;
* INSERT INTO student6 VALUES ("Raju","Rajaiah");
* SELECT \* FROM student6;
* INSERT INTO student6 VALUES ("Karthik","Rajireddy");
* SELECT \* FROM student6;
* INSERT INTO student6 VALUES ("Raju","Rajaiah");***// Error(Duplicate Entry)***

**Note:**

***CREATE TABLE student7(rollno int PRIMARY KEY, aadhar int PRIMARY KEY);***

***(Not possible)***

**DROP PRIMARY KEY:**

**Syntax:**

ALTER TABLE table\_name DROP PRIMARY KEY;

**Example:**

* ALTER TABLE student1 DROP PRIMARY KEY;
* DESC student1;
* ALTER TABLE student1 MODIFY sid int NULL;
* DESC student1;

Alternate key / Secondary key:

Candidate keys that are left unimplemented or unused after implementing the primary key are called as alternate keys. Or

Unimplemented candidate keys are called as alternate keys. Example:

|  |  |  |
| --- | --- | --- |
| Emp\_id | Emp\_name | Emp\_mailid |
| 101 | Raju | [raju123@gmail.com](mailto:raju123@gmail.com) |
| 102 | Ravi | [ravi@gmail.com](mailto:ravi@gmail.com) |
| 103 | Raju | [raju111@gmail.com](mailto:raju111@gmail.com) |

Candidate keys:

{Emp\_id, Emp\_mailid}

These are the candidate keys we concluded from the above attributes. Now, we have to choose one primary key, which is the most appropriate out of the two, and it is Emp\_Id. *So, the primary key is Emp\_Id.* Now, the remaining one candidate key is Emp\_emailid**.** Therefore, Emp\_emailid is the alternate key.

Composite key:

A key comprising of multiple attributes and not just a single attribute is called as a composite key. It is also known as *compound key.*

Any key such as [super key](https://beginnersbook.com/2015/04/super-key-in-dbms/), [primary key](https://beginnersbook.com/2015/04/primary-key-in-dbms/), [candidate key](https://beginnersbook.com/2015/04/candidate-key-in-dbms/) etc. can be called composite key if it has more than one attributes.

If a single column alone fails to be served as a primary key then combination columns would help to uniquely access a record from table such type of keys or nothing but composite keys.

**Example:**

|  |  |  |  |
| --- | --- | --- | --- |
| cust\_Id | order\_Id | product\_code | product\_count |
| C01 | O001 | P007 | 23 |
| C02 | O123 | P007 | 19 |
| C02 | O123 | P230 | 82 |
| C01 | O001 | P890 | 42 |

None of these columns alone can play a role of key in this table.

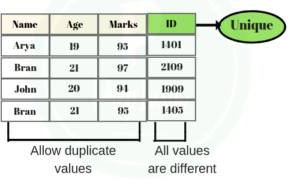
Based on this, it is safe to assume that the key should be having more than one attributes: Key in above table: {cust\_id, product\_code}.

This is a composite key as it is made up of more than one attributes.

UNIQUE Constraint:

* Sometimes we need to maintain only unique data in the column of a database table, this is possible by using a unique constraint.
* Unique constraint ensures that all values in a column are unique.
* Unique key accepts NULL.
* Unique key is a single field or combination of fields that ensure all values going to store into the column will be unique. It means a column cannot stores duplicate values.
* It allows us to use more than one column with UNIQUE constraint in a table.

Example:



**Syntax:**

Case 1:

CREATE TABLE table\_name ( Column1 data\_type,

Column2 data\_type UNIQUE KEY,

...

);

Example:

* CREATE TABLE test (sno int UNIQUE KEY,name char(50),age int);
* DESC test;
* INSERT INTO test values (101,"Deepak",30);
* SELECT \* FROM test;
* INSERT INTO test values (101,"Raju",32); ***// Error(Duplicate Value)***
* INSERT INTO test values (NULL,"Deepak",30); ***// NULL Accepted***
* SELECT \* FROM test;
* INSERT INTO test(name, age) values ("Ravinder",50);***// NULL Accepted***
* SELECT \* FROM test;

Case 2:

**Syntax:**

CREATE TABLE table\_name ( Column1 datatype,

Column2 datatype,UNIQUE KEY(Column1);

...

);

Example:

* CREATE TABLE student (sid int,name varchar(50),address varchar (50) , UNIQUE KEY(sid));
* DESC student;
* INSERT INTO student VALUES (1,"Raju","Hyd");
* SELECT \* FROM student;
* INSERT INTO student VALUES (1,"Rakesh","knr"); ***//Error(Duplicate value)***

Case 3:

CREATE TABLE table\_name ( Column1 datatype,

Column2 datatype, CONSTRAINT Column\_uk UNIQUE KEY(Column1);

...

);

Example:

* CREATE TABLE student1 (sid int, name char(50),address char(50),CONSTRAINT std\_uk UNIQUE KEY(sid));
* DESC student1;
* INSERT INTO student1 VALUES (101,"srinu","knr");
* SELECT \* FROM student1;
* INSERT INTO student1 VALUES (NULL,"SRINIVAS","HZB");
* SELECT \* FROM student1;
* INSERT INTO student1 VALUES (101,"Ravi","Parkal"); ***// Duplicate entry***

**Defining UNIQUE KEY on multiple columns:**

**Example:**

* CREATE TABLE student2 (sid int UNIQUE KEY, aadhar int UNIQUE KEY, name varchar(50));
* DESC student2;
* INSERT INTO student2 VALUES (101,998563256,"Mahesh");
* SELECT \* FROM student2;
* INSERT INTO student2 values (101,986532147,"Karthik");***// Error (Duplicate sid)***
* INSERT INTO student2 values(102,998563256,"Karthik");***// Error (Duplicate aadhar)***
* INSERT INTO student2 values(NULL,998513256,"Karthik");***// NULL Accepted for sid***
* SELECT \* FROM student2;
* INSERT INTO student2 values(105,NULL,"Karun"); ***//NULL Accepted for Aadhar***
* SELECT \* FROM student2;
* INSERT INTO student2 values (NULL,NULL,"Kamal"); ***//NULL Accepted for sid and Aadhar***
* SELECT \* FROM student2;

**Defining multiple fields as one UNIQUE KEY:**

**Example:**

* CREATE TABLE student3 (sid int,aadhar int,UNIQUE(sid,aadhar));
* DESC student1;
* INSERT INTO student3 VALUES (101,1234);
* SELECT \* FROM student3;
* INSERT INTO student3 VALUES(101,1235);
* SELECT \* FROM student3;
* INSERT INTO student3 VALUES(102,1234);
* SELECT \* FROM student3;
* INSERT INTO student3 VALUES(101,1234); ***// Duplicate entry***

**Unique Key Using ALTER TABLE Statement Syntax:**

ALTER TABLE table\_name ADD CONSTRAINT constraint\_name UNIQUE(column\_list

);

**Example:**

* CREATE TABLE student4(rollno int,name char(50));
* DESC student4;
* ALTER TABLE student4 ADD CONSTRAINT st4\_uni UNIQUE(rollno);
* DESC student4;

DROP Unique Key Syntax:

ALTER TABLE table\_name DROP INDEX constraint\_name;

**Example:**

* ALTER TABLE student4 DROP INDEX st4\_uni;
* DESC student4;
* ALTER TABLE test DROP INDEX sno;
* DESC test;

Foreign key constraint / Referential Integrity constraints:

In the relational databases, a foreign key is a field or a column that is used to establish a link between two tables. It is also known as the referencing key

In simple words you can say that, a foreign key in one table used to point primary key in another table.

Foreign key is used to make relation between two or more than two tables.

**Example:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Rollno | Name | Branch | Address | Activity\_name | Date |
| 101 | Rajesh | CSE | Hzb | Chess | 12/04/21 |
| 102 | Ravi | EEE | Knr | Tennis | 13/04/21 |
| 103 | Rakesh | MEC | Knr | Carrom | 13/04/21 |
| 101 | Rajesh | CSE | Hzb | Cricket | 13/04/21 |
| 101 | Rajesh | CSE | Hzb | Table tennis | 15/04/21 |

**Drawbacks of above table are:**



* + Redundant data (Name, Branch, address).
  + Difficult to implement primary key.

Student table

|  |  |  |  |
| --- | --- | --- | --- |
| Rollno | Name | Branch | Address |
| 101 | Rajesh | CSE | Hzb |
| 102 | Ravi | EEE | Knr |
| 103 | Rakesh | MEC | Knr |



Primary key

Activity table

|  |  |  |
| --- | --- | --- |
| Rollno | Activity\_name | Date |
| 101 | Chess | 12/04/21 |
| 102 | Tennis | 13/04/21 |
| 103 | Carrom | 13/04/21 |
| 101 | Cricket | 13/04/21 |
| 101 | Table tennis | 15/04/21 |

Foreign key



**Conditions to apply foreign key:**

* + - One table should contain primary key and other table contains foreign key.
    - A common column in both tables.
    - The common column data type must be same in both tables.

There are two ways to define foreign key.

1. Using CREATE statement.
2. Using ALTER statement.

**Defining foreign key CREATE statement:**

**Syntax:** Following is the basic syntax used for defining a foreign key using CREATE TABLE OR ALTER TABLE statement

[CONSTRAINT constraint\_name] FOREIGN KEY [foreign\_key\_name] (col\_nam) REFERENCES parent\_tbl\_name (col\_name,...)

ON DELETE referenceOption ON UPDATE referenceOption

ReferenceOption

RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT

**Example:**

* + CREATE TABLE department (deptno int PRIMARY KEY,dname varchar

1. , location varchar(50));
   * The table in which primary key is applied is called parent table or referenced table.
   * DESC department;
   * INSERT INTO department VALUES (05,"cse","main block"), (04,"ECE","Block 2"),

(01,"civil","Block 3");

* + - SELECT \* FROM department;
  + create table employee(empid int,ename varchar(40),DOJ date, deptno int, FOREIGN KEY(deptno) references department(deptno));
* The table in which foreign key is applied is called child table or referencing table.

***Test:***

Here the values 1, 4, 5 from parent table only applied by child table, if we give any other value it don’t accept. If is true (accepting) then we can say that there is no relation between the tables.

* + INSERT INTO employee VALUES (101,"karun",'2017-04-01',1);
  + SELECT \* FROM employee;
  + INSERT INTO employee VALUES (102,"kamal",'2018-05-30',4);
  + SELECT \* FROM employee;
  + INSERT INTO employee VALUES (103,"karthik","2019-04-28",10); ***// not accepted because 10 not present in parent (department) table.***
  + DELETE from department where deptno=1; *// Error*

**Creating foreign key using ALTER command:**

**Example:**

* + create table student(rollno int PRIMARY KEY,name varchar(30),branch char

(20) , address varchar(30));

* + DESC student;
  + CREATE table activity (rollno int,activity\_name char(50),DOA date);
  + DESC activity;
  + ALTER TABLE activity ADD CONSTRAINT fk\_ac FOREIGN KEY(rollno) references student(rollno);
  + DESC activity;

**Dropping the foreign key:**

**Syntax:**

ALTER TABLE table\_name DROP FOREIGN KEY fk\_constraint\_name;

**Example:**

Show create table activity;

ALTER TABLE activity DROP FOREIGN KEY fk\_ac; Show create table activity;

ENFORCING INTEGRITY CONSTRAINTS

ICs are specified when a relation is created and enforced when a relation is modified. The impact of domain, PRIMARY KEY, and UNIQUE constraints is straightforward: If an insert, delete, or update command causes a violation, it is rejected. Every potential IC violation is generally checked at the end of each SQL statement execution.

**Example:**

* + CREATE TABLE students (sid int PRIMARY KEY,name varchar(50),login varchar(50),age int);
  + INSERT INTO students VALUES ( 53899 ,"Deepak" , [gujjula@gmail.com](mailto:gujjula@gmail.com) , 30);

*The following insertion violates the primary key constraint because there is already a tuple with the s'id 53899, and it will be rejected by the DBMS:*

* + INSERT INTO students VALUES (53899,"Raju","[raju@gmail.com](mailto:raju@gmail.com)",28);

*The following insertion violates the constraint that the primary key cannot contain null:*

* + INSERT INTO students VALUES (NULL,"Raju","[raju@gmail.com](mailto:raju@gmail.com)",28);
  + INSERT INTO students VALUES(5000,"karthik","[karthi@gmail.com](mailto:karthi@gmail.com)",28);

*Deletion does not cause a violation of domain, primary key or unique key constraints. However, an update can cause violations, similar to an insertion.*

* + UPDATE students SET sid=5000 where sid=53899;

*This update violates the primary key constraint because there is already a tuple with sid 5000.*

The impact of foreign key constraints is more complex because SQL sometimes tries to rectify a foreign key constraint violation instead of simply rejecting the change.

We discuss the referential integrity enforcement steps taken by the DBMS in terms of our department and employee tables.

* + CREATE TABLE department (dno int PRIMARY KEY,dname varchar(50));
  + DESC department;
  + INSERT INTO department VALUES (01,"civil");
  + INSERT INTO department VALUES (02,"eee");
  + INSERT INTO department VALUES (03,"mec");
  + INSERT INTO department VALUES (04,"ece");
  + INSERT INTO department VALUES (05,"cse");
  + CREATE table employee(empid int PRIMARY KEY,ename varchar(50),dno int,FOREIGN KEY(dno) REFERENCES department(dno));
  + DESC employee;
  + INSERT INTO employee VALUES (101,"Deepak",5);
  + INSERT INTO employee VALUES (102,"karun",4);
  + SELECT \* FROM employee;

***Deletion of employee tuple does not violate referential integrity.***

* + DELETE FROM employee where empid=101;
  + SELECT \* FROM employee;

***Insertion of employee tuple violates the referential integrity because there is no department tuple with dno 10.***

* + INSERT INTO employee VALUES (105,"kamal",10);

***Insertion of department tuples does not violate referential integrity.***

* + INSERT INTO department VALUES (6,"MBA");
  + SELECT \* FROM deparment;

***Deletion of department tuples could cause violations.***

* + DELETE FROM department WHERE dno=4;

The FOREIGN KEY constraint provides an ability to control what action will be taken when the referenced value in the parent table is updated or deleted, using the ON UPDATE and ON DELETE clauses. The supported actions that can be taken when deleting or updating the parent table’s values include:

* **NO ACTION**: When the ON UPDATE or ON DELETE clauses are set to NO ACTION, the performed update or delete operation in the parent table will fail with an error.
* **CASCADE**: Setting the ON UPDATE or ON DELETE clauses to CASCADE, the same action performed on the referenced values of the parent table will be reflected to the related values in the child table. For example, if the referenced value is deleted in the parent table, all related rows in the child table are also deleted.
* **SET NULL**: With this ON UPDATE and ON DELETE clauses option, if the referenced values in the parent table are deleted or modified, all related values in the child table are set to NULL value.
* **SET DEFAULT**: Using the SET DEFAULT option of the ON UPDATE and ON DELETE clauses specifies that, if the referenced values in the parent table are updated or deleted, the related values in the child table with FOREIGN KEY columns will be set to its default value.

**ON DELETE CASCADE:**

* + DROP TABLE employee;
  + DROP TABLE department;
  + CREATE TABLE department (dno int PRIMARY KEY,dname varchar(50));
  + DESC department;
  + INSERT INTO department VALUES (01,"civil");
  + INSERT INTO department VALUES (02,"eee");
  + INSERT INTO department VALUES (03,"mec");
  + INSERT INTO department VALUES (04,"ece");
  + INSERT INTO department VALUES (05,"cse");
  + CREATE TABLE employee (empid int,ename varchar(50),dno int , FOREIGN key(dno) REFERENCES department(dno) ON DELETE CASCADE);
  + INSERT INTO employee VALUES(101,"karun",1);
  + INSERT INTO employee VALUES(102,"kamal",2);
  + INSERT INTO employee VALUES(103,"karthik",3);
  + DELETE from department where dno=1;

***Observation: In department table tuple with dno 1 is deleted and in employee table tuple with dno 1 is also deleted.***

**ON DELETE SET NULL:**

* + DROP TABLE employee;
  + CREATE TABLE employee (empid int ,ename varchar(50),dno int,FOREIGN KEY(dno) REFERENCES department(dno) ON DELETE SET NULL);
  + INSERT INTO employee VALUES ( 101, "raju", 2) , (102 ,"rajesh" ,3 ), (103,"ranjith",4);
  + DELETE FROM department WHERE dno=2;
  + SELECT \* FROM employee;

***Observation: In department table tuple with dno 2 is deleted and in employee table 2 is replaced with NULL.***

**ON DELETE NO ACTION:**

* + DROP TABLE employee;
  + CREATE TABLE employee (empid int ,ename varchar(50),dno int,FOREIGN KEY(dno) REFERENCES department(dno) ON DELETE NO ACTION);
  + INSERT INTO employee VALUES ( 101, "raju", 5) , (102 ,"rajesh" ,3 ), (103,"ranjith",4);
  + DELETE FROM department WHERE dno=5; ***// Error***

QUERYING RELATIONAL DATA

A relational database query (query, for short) is a question about the data, and the answer consists of a new relation containing the result. A query language is a specialized language for writing queries.

SQL is the most popular commercial query language for a relational DBMS. We now present some SQL examples that illustrate how easily relations can be queried.

* + CREATE TABLE employee(id int,name varchar(50),dept varchar(50),salary int,DOB date,gender char(1));
  + DESC employee;
  + INSERT INTO employee VALUES (1,'Praveen','cse',4000,'1990-05-06','M'), (2,'Raju','cse',6000,'1991-01-08','M'), (3,'Ravi','ece',8000,'1990-04-21','M'), (4,'Rajitha','eee',4000,'1990-05-18','F'), (5,'Kamal','cse',6000,'1990-04-17','M');

1. ***Write a query to display names of the employee table? Ans:*** SELECT name from employee;
2. ***Write a query to display id and names of the employee table? Ans:*** SELECT id, name from employee;
3. ***Write a query to display all the (details) columns of the employee table? Ans:*** SELECT \* FROM employee;
   * The symbol ‘\*’ means that we retain all fields of the relation.
4. ***Write a query to display details of employee who belongs to cse department? Ans:*** SELECT \* FROM employee where dept='cse';
   * *Where clause is used to filter records, to extract only those records that fulfill a specified condition.*
   * *SQL requires single quotes around text values, however numeric fields should not be enclosed in quotes.*
5. ***Write a query to display details of employee whose name is praveen? Ans:*** SELECT \* from employee where name='praveen';
6. ***Find the names of the employee who has salary greater than 4000. Ans:*** SELECT \* FROM employee where salary > 4000***;***
7. ***Write a query to find all departments except cse. Ans:*** SELECT dept from employee where dept!='cse';
8. ***Display the names of employee whose names contain upto 5 characters. Ans:*** SELECT name from employee where length(name)=5;

Logical Database Design

First step of any relational database design is to make ER Diagram for it and then convert it into relational Model.

It is very convenient to design the database using the ER Model by creating an ER diagram and later on converting it into relational model.

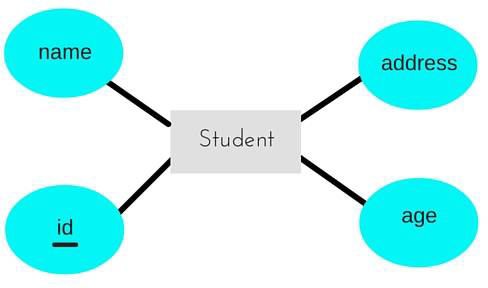
Relational Model represents how data is stored in database in the form of table.

Not all the ER Model constraints and components can be directly transformed into relational model, but an approximate schema can be derived.

We now describe how to translate an ER diagram into a collection of tables with associated constraints, that is, a relational database schema.

1. Entity Sets to Tables:

Consider we have entity STUDENT in ER diagram with attributes id, name, address and age.



To convert this entity set into relational schema

1. Entity is mapped as relation (table) in Relational schema
2. Attributes of Entity set are mapped as attributes (columns) for that Relation.
3. Key attribute of an Entity mapped as Primary key for that Relation.

A table with name **Student** will be created in relational model, which will have 4 columns, id, name, age, address and id will be the primary key for this table.

|  |  |  |  |
| --- | --- | --- | --- |
| id | name | age | address |

The following SQL statement captures the preceding information, including the domain constraints and key information.

CREATE TABLE student (id int PRIMARY KEY, name varchar (50),age int, address char(50));

1. Entity set with composite attributes:
   * In relational model don’t represent the composite attributes as it is, just take the compositions.
   * The attributes of a relation includes the simple attributes of an entity set.

**TA**

**salary**

**DA**

**HRA**

**basic**



eno

**ename**

**employee**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| eno | ename | TA | DA | HRA | basic |



CREATE TABLE employee (eno int PRIMARY KEY, ename varchar(50),TA int, DA int, HRA int, basic int);

1. Entity set with multivalued attributes:

If an entity contains multivalued attributes, it is represented with two relations, one with all simple attributes and other with key and all multivalued attributes.

**ename**

**Phone\_no**

**eno**

**Employee**

Employee employee\_phne



|  |  |
| --- | --- |
| **eno(PK)** | **ename** |
| 1 | A |
| 2 | B |

|  |  |
| --- | --- |
| **eno(FK)** | **Phone\_no** |
| 1 | 9966554744 |
| 1 | 9491474728 |
| 2 | 9966551122 |

CREATE TABLE employee (eno int PRIMARY KEY,ename varchar(40));



CREATE TABLE employee\_phne (eno int, phone\_no char(50),FOREIGN KEY(eno) REFERENCES employee(eno));

1. Relationship Sets (without Constraints) to Tables:

A Relationship set, like an entity set, is mapped to a relation in the relational model. The attributes of a relation includes.

* Key attributes of the participating relation are declared as foreign key to the respective relation.
* Descriptive attribute (if any).
* Set of non – descriptive attributes is the primary key of the relation.

**Example:**

**since**

**works**

**\_in**



**eno**

**Employee**

**Ename**

**did**

**Department**

**dname**

Employee Works\_in department

**salary**

|  |  |  |
| --- | --- | --- |
| eno | ename | salary |
| 1 | Raju | 50000 |
| 2 | Ravi | 60000 |

|  |  |  |
| --- | --- | --- |
| eno did | | since |
| 1 | 101 | 01/01/1990 |
| 1 | 102 | 02/05/1995 |
| 2 | 102 | 06/06/2000 |

|  |  |
| --- | --- |
| did | Dname |
| 101 | CSE |
| 102 | IT |

The following SQL statement captures the preceding information.



* + CREATE TABLE employee (eno int PRIMARY KEY, ename varchar(50), salary int);
  + CREATE TABLE department (did int PRIMARY KEY,dname varchar(50));
  + CREATE TABLE works\_in (eno int,

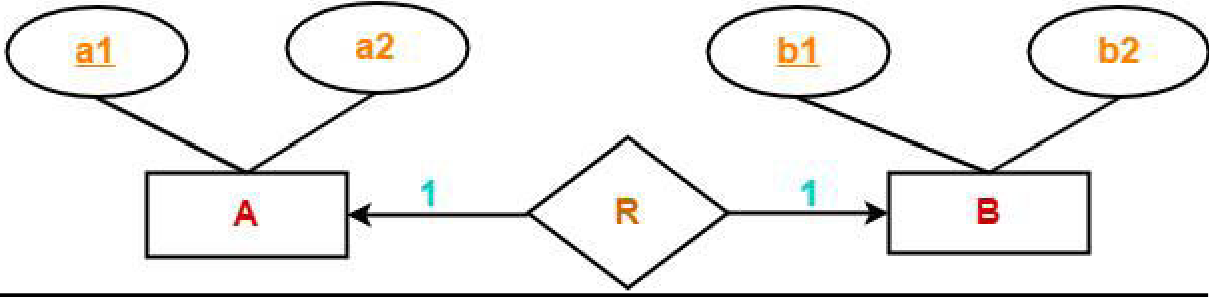
did int, since date,

PRIMARY KEY (eno, did),

FOREIGN KEY (eno) REFERENCES employee (eno), FOREIGN KEY (did) REFERENCES department (did));

1. Translating Relationship Sets with key Constraints: Relationship with cardinality ratio / key constraints 1:1

When a row in a table is related to only one row in another table and vice versa, we say that is a **one to one relationship**. This relationship can be created using *Primary key-Unique foreign key constraints*.



Here, two tables will be required. Either combines ‘R’ with ‘A’ or ‘B’

**Way-01:**

1. AR ( a1 , a2 , b1 )

2. B ( b1 , b2 )

**Way-02:**

1. A ( a1 , a2 )

2. BR ( a1 , b1 , b2 )

Example:

**sid**

**sname**

1

**sch\_id**

1

**sch\_type**

|  |  |
| --- | --- |
| **Student** |  |
|  |

**sch\_year**

**Scholarship**

*Database Management Systems*

**can have**



STUDENT TABLE SCHOLARSHIP TABLE

|  |  |  |
| --- | --- | --- |
| sid(PK) | Sname | sch\_id(FK  & UK) |
| 501 | Rakesh | 123555 |
| 502 | Raju | 556644 |
| 503 | Ravi | NULL |

|  |  |  |
| --- | --- | --- |
| sch\_id(PK) | sch\_type | sch\_year |
| 123555 | b.tech | 2020-05-21 |
| 556644 | b.tech | 2021-06-30 |

The following SQL statement captures the preceding information. Example:



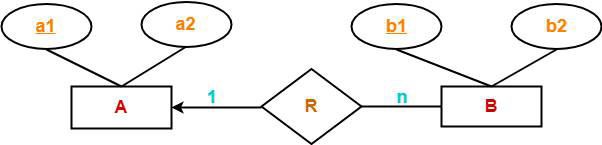
* + CREATE TABLE scholarship(schid int PRIMARY KEY,schtype varchar(50)

,schyear date);

* + DESC student;
  + CREATE TABLE student (sid int PRIMARY KEY, sname varchar(50),schid int UNIQUE KEY, FOREIGN KEY(schid) REFERENCES scholarship (schid));

**Relationship with cardinality ratio / key constraints 1: N**

This is where a row from one table can have multiple matching rows in another table this relationship is defined as a **one to many relationships**. This type of relationship can be created using *Primary key-Foreign key relationship.*



Here, two tables will be required-

1. A ( a1 , a2 )

2. BR ( a1 , b1 , b2 )

**Example:**

**cid**

**Customer**

**cname**

**Addr\_id**

**city**

**Address**

**1**

**Can**

**have**

**N**

**Country**

**customer** **address**



|  |  |
| --- | --- |
| **cid** | **cname** |
| 1 | Sravan |
| 2 | Sravanthi |

|  |  |  |  |
| --- | --- | --- | --- |
| **cid** | **Addr\_id** | **city** | **country** |
| 1 | 34 | Huzurabad | India |
| 1 | 46 | Karimnagar | India |
| 2 | 55 | Hyderabad | India |



The following SQL statement captures the preceding information.

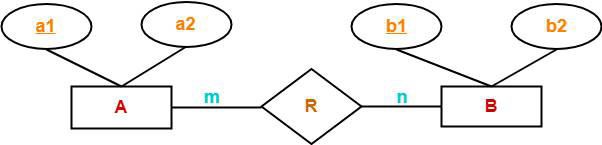
* + CREATE TABLE customer (cid int PRIMARY KEY, cname varchar(50));
  + CREATE TABLE address (cid int, addr\_id int PRIMARY KEY, city varchar

(50) , country varchar(50), FOREIGN KEY(cid) REFERENCES customer(cid));

**Relationship with cardinality ratio / key constraints M:N**

A row from one table can have multiple matching rows in another table, and a row in the other table can also have multiple matching rows in the first table this relationship is defined as a **many to many relationship**. This type of relationship can be created using a third table called “*Junction table*” or “*Bridging table*”. This Junction or Bridging table can be assumed as a place where attributes of the relationships between two lists of entities are stored.

This kind of Relationship, allows a junction or bridging table as a connection for the two tables.



Here, three tables will be required- 1. A ( a1 , a2 )

2. R ( a1 , b1 )

3. B ( b1 , b2 )

Example:

**sname**

**sid**

**student**

**cid**

**cname**

**Course**

**N**

**takes**

**M**

*Database Management Systems*



74

student std-takes-course course

|  |  |
| --- | --- |
| sid | sname |
| 1 | Raju |
| 2 | Ravi |

|  |  |
| --- | --- |
| eno did | |
| 1 | 101 |
| 1 | 102 |
| 2 | 102 |

|  |  |
| --- | --- |
| did | dname |
| 101 | Java |
| 102 | DBMS |



The following SQL statement captures the preceding information.

* + CREATE TABLE student (sid int PRIMARY KEY, sname int);
  + CREATE TABLE course (did int PRIMARY KEY, dname varchar (50));
  + CREATE TABLE std\_takes\_course ( eno int,

did int,

PRIMARY KEY (eno, did),

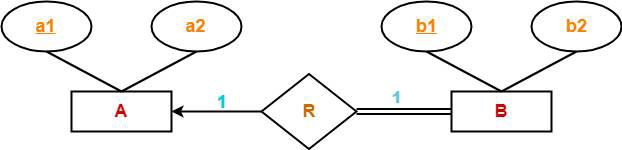
FOREIGN KEY (sid) REFERENCES student (sid), FOREIGN KEY (eno) REFERENCES course (eno));

1. Translating Relationship Sets with Participation Constraints:

Because of the total participation constraint, foreign key acquires **NOT NULL**

constraint i.e. now foreign key cannot be null.

**Example:**



Here, two tables will be required. Either combines ‘R’ with ‘A’ or ‘B’

**Way-01:**

1. AR ( a1 , a2 , b1 )

2. B ( b1 , b2 )

**Way-02:**

1. A ( a1 , a2 )

2. BR ( a1 , b1 , b2 )

Example:

**sid**

**sname**

1

**sch\_id**

**sch\_type**

|  |  |
| --- | --- |
| **Student** |  |
|  |

**sch\_year**

**Scholarship**

Student table Scholarship table

**Can have**

1



|  |  |
| --- | --- |
| sid(PK) | Sname |
| 501 | Rakesh |
| 502 | Raju |
| 503 | Ravi |

|  |  |  |  |
| --- | --- | --- | --- |
| sch\_id(PK) | sch\_type | sch\_year | Sid(FK&UK) |
| 123555 | b.tech | 2020-05-21 | 501 |
| 556644 | b.tech | 2021-06-30 | 502 |
| 225665 | MCA | 2021-02-02 | 503 |



The following SQL statement captures the preceding information.

* + CREATE TABLE student (sid int PRIMARY KEY, sname varchar(50));
  + CREATE TABLE scholarship(schid int PRIMARY KEY,schtype varchar(50)

,schyear date, sid int UNIQUE KEY NOT NULL, FOREIGN KEY(sid) REFERENCES student(sid) );

Example – 2:

**sch\_type**

**Can**

**have**

**sid**

**sname**

1

**sch\_id**

1

**sch\_year**

**Scholarship**

|  |  |
| --- | --- |
| **Student** |  |
|  |

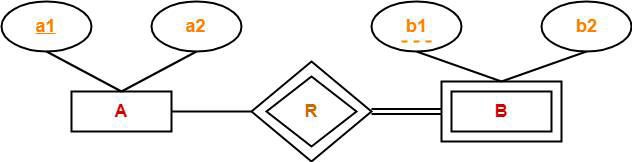


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sid | sname | schid | schtype | Schyear |
| 101 | Rakesh | 221154 | B.Tech | 2020 |
| 102 | Ravi | 332641 | B.Tech | 2021 |
| 103 | Rajesh | 548793 | MCA | 2021 |

CREATE TABLE student (sid int PRIMARY KEY, sname varchar(50),schid int UNIQUE KEY NOT NULL, schtype varchar(50),schyear date );

1. Translating Weak Entity Sets:

Weak entity set always appears in association with identifying relationship with total participation constraint.



Here, two tables will be required- 1. A ( a1 , a2 )

2. BR ( a1 , b1 , b2 )

Example:

**ename**

dname

**age**

**Dependent**

**eno**

**Employee**

**has**

Employee Dependent

|  |  |
| --- | --- |
| **eno** | **ename** |
| 101 | Raju |
| 102 | Ravi |
| 103 | Rakesh |

|  |  |  |
| --- | --- | --- |
| **dname** | **age** | **Eno** |
| Rajesham | 50 | 101 |
| Raji reddy | 55 | 102 |
| Venkataswamy | 58 | 103 |

* + CREATE TABLE employee (eno int PRIMARY KEY, ename varchar (50));
  + CREATE TABLE dependent (dname varchar (50), age int, eno int UNIQUE KEY NOT NULL, PRIMARY KEY (eno,pname),FOREIGN KEY(eno) REFERENCES employee(eno) ON DELETE CASCADE);

VIEWS

**View:** Views are Virtual Relations or virtual tables, through which a selective portion of the data from one or more relations (or tables) can be seen.

The tables on which the view is based are called base tables. Views do not exist physically.

Views are stored in data dictionary.

You can create a view by using the **CREATE VIEW** command: The CREATE VIEW statement is a data definition command.

Views provide a level of security in the database because the view can restrict users to only specified columns and specified rows in a table. For example, if you have a company with hundreds of employees in several Departments, you could give the secretary of each department a view of only certain attributes and for the employees that belong only to that secretary’s department

**Syntax:**

CREATE VIEW view\_name AS

SELECT column\_list

FROM table\_name [where condition];

**Example:**

* + CREATE table student (rollno int,sname varchar(50),gender char(50),gmail varchar(50),DOB date,password varchar(50));
  + DESC student;
  + INSERT INTO student VALUES (601,"karthik","M","[karthik@gmail.com](mailto:karthik@gmail.com)",'1990-05-18',"123456"), (602,"raju","M","[raju@gmail.com](mailto:raju@gmail.com)",'1990-04-21',"996655"), (603,"rajitha","F","[rajitha123@gmail.com](mailto:rajitha123@gmail.com)",'1990-02-09',"111111");
  + SELECT \* FROM student;

**CREATING SIMPLE VIEW:**

* + CREATE VIEW student\_details AS SELECT rollno,sname,gender FROM student;
  + SELECT \* FROM student\_details;

***Data modifications like insert, delete, and update operations on base table, it will reflect on views.***

* + INSERT INTO student VALUES (604,"Mahesh","M","[mahesh@hmail.com](mailto:mahesh@hmail.com)","1992-01-28",3333);
  + SELECT \* FROM student\_details;

***Data modifications like insert, delete, and update operations on views affects the actual relations in the database, upon which view is based.***

* + UPDATE student\_details SET sname="Narendher" WHERE rollno=601;
  + SELECT \* FROM student\_details; // Updated successfully
  + SELECT \* FROM student;
  + INSERT INTO student\_details VALUES (655,"srikanth","M");
  + SELECT \* FROM student\_details;
  + SELECT \* from student;
  + DELETE FROM student\_details WHERE rollno=601;
  + SELECT \* FROM student;
  + SELECT \* FROM student;
  + CREATE TABLE employee (eno int PRIMARY KEY,ename varchar(40));
  + CREATE TABLE employee\_phne (eno int, phone\_no char(50),FOREIGN KEY(eno) REFERENCES employee(eno));
  + INSERT INTO employee VALUES (101,"Mahipal"),(102,"Mahonar");
  + INSERT INTO employee\_phne VALUES(101,"9966554744"),(102,"9160600571");

**CREATING COMPLEX VIEWS:** view creation that involves multiple tables.

* + CREATE VIEW employe\_details AS SELECT employee.eno, employee.ename, employee\_phne.phone\_no FROM employee, employee\_phne WHERE employee.eno=employee\_phne.eno;
  + SELECT \* from employe\_details;

**UPDATE VIEWS:**

ALTER VIEW statement is used to modify or update the already created VIEW without dropping it.

**Syntax:**

ALTER VIEW view\_name AS SELECT columns FROM table WHERE conditions;

**Example:**

* + ALTER VIEW employe\_details AS SELECT employee.eno, employee\_phne.phone\_no FROM employee, employee\_phne WHERE employee.eno=employee\_phne.eno;
  + SELECT \* FROM employe\_details;

**DROP VIEW:**

We can drop the existing VIEW by using the DROP VIEW statement.

**Syntax:**

DROP VIEW view\_name:

**Example:**

* + DROP VIEW employe\_details;

**Advantages:**

Views improve security of the database by showing only intended data to authorized users. They hide sensitive data.

Views make life easy as you do not have write complex queries time and again.

**RELATIONAL ALGEBRA & RELATIONAL CALCULUS**

In the previous tutorials, we discussed the designing of database using [Relational](https://beginnersbook.com/2015/04/relational-model-in-dbms/) [model](https://beginnersbook.com/2015/04/relational-model-in-dbms/), [E-R diagram](https://beginnersbook.com/2015/04/e-r-model-in-dbms/). Now we have to design the database, we need to store and retrieve data from the database, for this purpose we need to understand the concept of Relational algebra and relational calculus.

Query Language:

In simple words, a Language which is used to store and retrieve data from database is known as query language. For example – **SQL**

There are two types of Query Languages:

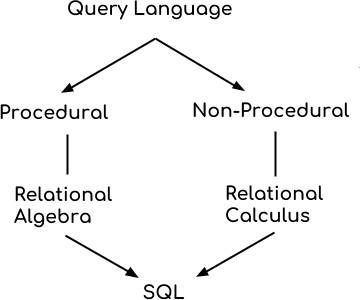
1. Procedural Query Language.
2. Non – Procedural Language.
3. Procedural Query language:

In procedural query language, user instructs the system to perform a series of operations to produce the desired results. Here users tell what data to be retrieved from database and how to retrieve it.

**For example –** Let’s take a real world example to understand the procedural language, you are asking your younger brother to make a cup of tea, if you are just telling him to make a tea and not telling the process then it is a non-procedural language, however if you are telling the step by step process like switch on the stove, boil the water, add milk etc. then it is a procedural language.

1. Non-procedural query language:

In Non-procedural query language, user instructs the system to produce the desired result without telling the step by step process. Here users tells what data to be retrieved from database but doesn’t tell how to retrieve it.



* + Relational algebra and calculus are the theoretical concepts used on relational model.
  + RDBMS is a practical implementation of relational model.
  + SQL is a practical implementation of relational algebra and calculus.

##### RELATIONAL ALGEBRA

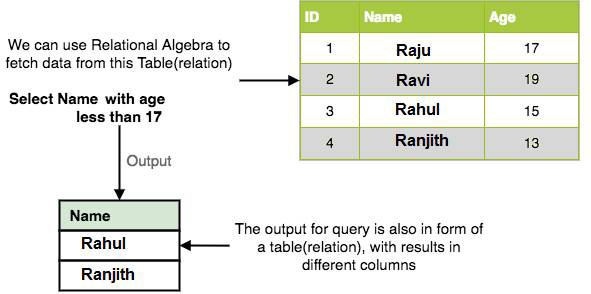
*Relational algebra is a family of algebras with a well-founded semantics used for modeling the data stored in relational databases, and defining queries on it.*

Relational algebra is a procedural Query Language (What is required and how) or Formal Query Language used to query the database tables to access data in different ways.

Introduced by E.F Codd in 1970’s.

Relational Algebra mainly provides theoretical foundation for relational database and SQL.

In relational algebra, input is a relation (table from which data has to be accessed) and output is also a relation (a temporary table holding the data asked for by the user). **Student table**



Operators in Relational Algebra:

Basic Relational Algebra Operations are:

* SELECT (σ)



Unary Operators

* PROJECT (π)
* RENAME (ρ)
* UNION (𝖴)
* INTERSECTION (∩ ),



Set Theory Operations

* DIFFERENCE (-)
* CROSS PRODUCT (X)

Binary Relational Operations:

* JOIN
* DIVISION.

1. **Select Operator (σ):** Select Operator is denoted by sigma (σ) and it is used to

find the tuples (or rows) in a relation (or table) which satisfy the given condition.

**Notation** − σ*p*(r)

Where, **σ** represents the Select Predicate.

**r** is the name of relation (table name in which you want to look for data).

**p** is the prepositional logic, where we specify the conditions that must be satisfied by the data.

Let's take an **example** of the Student table we specified above in the Introduction of relational algebra, and fetch data for **students** with **age** more than 15.

σ age > 15 (Student)

*The subscript age > 15 specifies the selection criterion to be applied while retrieving tuples.*

This will fetch the tuples (rows) from table **Student**, for which **age** will be greater than **15**.

**Output:**

|  |  |  |
| --- | --- | --- |
| **ID** | **NAME** | **AGE** |
| 1 | Raju | 17 |
| 2 | Ravi | 19 |

You can also use, *and, or* etc operators, to specify two conditions, for example,

**σ age > 17 and gender = 'Male' (Student)**

This will return tuples (rows) from table **Student** with information of male students, of age more than 17. (Consider the Student table has an attribute Gender too.)

1. **Projection Operator (π):** This operation shows the list of those attributes (columns) that we wish to appear in the result. Rest of the attributes is eliminated from the table (relation).

It is denoted by ∏ (pi).

***Note that projection removes duplicate data (tuples).***

Notation − ∏A1, A2 ….. An (r)

Where A1, A2, An are attribute (column) names of relation **r**. **For example,**

∏name, age(Student)

The subscript *name, age* specifies the fields to be retained; the other fields are projected out.

Above statement will show us only the **Name** and **Age** columns for all the rows of data in **Student** table.

Since the result of a relational algebra expression is always a relation, we can substitute an expression wherever a relation is expected. For example, we can compute the names of students whose age is greater than 15 by combining two of the preceding queries. The expression

∏**name** (**σ age > 15** (Student)

σ age > 15 (Student)



|  |  |  |
| --- | --- | --- |
| **ID** | **NAME** | **AGE** |
| 1 | Raju | 17 |
| 2 | Ravi | 19 |

∏**name**



|  |
| --- |
| **Name** |
| Raju |
| Ravi |

1. **Rename (ρ):** Rename is a unary operation used for renaming a relation name or renaming attributes of a relation.

'rename' operation is denoted with small Greek letter **rho** *ρ*.

Notation:

**First way:**

**ρ(C(oldname** **newname or position** **newname, relation\_name)) Second way: ρ S(B1,B2,….Bn)(R) or ρS(R)or ρ B1,B2,….Bn(R)**

Where the symbol P (rho) is used to denote the RENAME operator, S is the new relation name, and B1, *B2 …* Bn are the new attribute names. The first expression renames both the relation and its attributes, the second renames the relation only, and the third renames the attributes only. If the attributes of R are (A1, *A,* …. An) in that order, then each Ai is renamed as Bi.

Example:

ρ(C(1 sid1, 5 sid2),R)

1. **UNION (**𝖴**):** Let R and S are two relations; *R* 𝖴 S returns a relation instance containing all tuples that occur in *either* relation instance *R* or relation instance S (or both). R and S must be *union-compatible,* and the schema of the result is defined to be identical to the schema of *R.*

Two relation instances are said to be **union-compatible** if the following conditions hold:

* + they have the same number of the fields, and
  + Corresponding fields, taken in order from left to right, have the same *domains.*

Note that field names are not used in defining **union-compatible.**

It eliminates the duplicate tuples.

**Notation:** R 𝖴 S

**Example:**

**Student** **Employee**

|  |  |
| --- | --- |
| **Rollno** | **Name** |
| 101 | Sagar |
| 102 | Santhosh |
| 103 | Sagarika |

|  |  |
| --- | --- |
| **Empno** | **Name** |
| 103 | Sagarika |
| 104 | Sandeep |

Student 𝖴 Employee

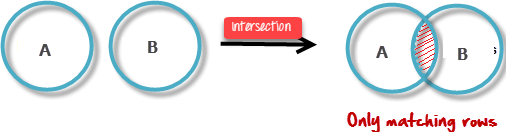
|  |  |
| --- | --- |
| **Rollno** | **Name** |
| 101 | Sagar |
| 102 | Santhosh |
| 103 | Sagarika |
| 104 | Sandeep |

1. **INTERSECTION (∩):**

*R* **∩** S returns a relation instance containing all tuples that occur in *both* R and *S.* The relations R and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R.*

It is denoted by intersection ∩.

It eliminates the duplicate tuples.



**Example:**

**Student** **Employee**

|  |  |
| --- | --- |
| **Rollno** | **Name** |
| 101 | Sagar |
| 102 | Santhosh |
| 103 | Sagarika |

|  |  |
| --- | --- |
| **Empno** | **Name** |
| 103 | Sagarika |
| 104 | Sandeep |

Student ∩ Employee



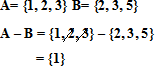
|  |  |
| --- | --- |
| **Rollno** | **Name** |
| 103 | Sagarika |

1. **Set Difference (**–**):**

R – S returns a relation instance containing all tuples that occur in *R* but not in S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of *R.*

It is denoted by intersection minus (-). R – S ≠ S – R

**Example:**



**Student\_list** **Failed\_list**

|  |  |
| --- | --- |
| **sid** | **Name** |
| 101 | Sagar |
| 102 | Santhosh |
| 103 | Sagarika |

|  |  |
| --- | --- |
| **sid** | **Name** |
| 101 | Sagar |
| 102 | Santhosh |

**Query:** Tell me the names of students who passed in the exams.

(Student\_list) – (Failed\_list) (Passed list)

|  |  |
| --- | --- |
| **Rollno** | **Name** |
| 103 | Sagarika |

1. **Cross Product (X):**

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

It is denoted by X. It is recommended to have one common column.

|  |  |  |  |
| --- | --- | --- | --- |
|  | R1 | R2 | R1 X R2 |
| Attributes | x | y | x + y number of attributes |
| tuples | n1 | n2 | n1 \* n2 no of tuples |

**Example:**

**Student** **Course\_details**

|  |  |
| --- | --- |
| **id** | **Name** |
| 101 | Ravi |
| 102 | Rajesh |
| 103 | Rathan |

|  |  |
| --- | --- |
| **id** | **course** |
| 102 | Java |
| 103 | DBMS |
| 105 | C |

Number of columns in Student table are 2 Number of column in course\_details are 2 Number of tuples in students are 3 Number of tuples in course\_details are 3

So Number of columns in Student X Course\_details are 2 + 2 = 4 Number of tuples (rows) in Student X Course\_details is 3 \* 3 = 9

Student X Course\_details

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **Name** | **id** | **Course** |
| 101 | Ravi | 102 | Java |
| 101 | Ravi | 103 | DBMS |
| 101 | Ravi | 105 | C |
| 102 | Rajesh | 102 | Java |
| 102 | Rajesh | 103 | DBMS |
| 102 | Rajesh | 105 | C |
| 103 | Rathan | 102 | Java |
| 103 | Rathan | 103 | DBMS |
| 103 | Rathan | 105 | C |

**JOINS:**

The *join* operation is one of the most useful operations in relational algebra and the most commonly used way to combine information from two or more relations.

Join can be defined as a cross-product followed by selections and projections. Types of Joins:

1. Inner Joins
   * Conditional Join / theta Join
   * Equi Join
   * Natural Join
2. Outer Joins
   * Left Outer Join
   * Right Outer Join
   * Full Outer Join

Conditional Join / theta Join:

**Theta join** allows you to merge two tables based on the condition represented by theta. Theta joins work for all comparison operators. It is denoted by symbol **θ**.

Syntax:

A ⋈θ or c B (A and B are two relations)

A ⋈θ or c B= σ c (A X B)

Thus ⋈ is defined to be a cross – product followed by a selection.

Example:

**Mobile** **Laptop**

|  |  |
| --- | --- |
| **Model** | **Price** |
| Nokia | 10000 |
| Samsung | 20000 |
| iphone | 50000 |

|  |  |
| --- | --- |
| **Model** | **Price** |
| Dell | 30000 |
| Acer | 20000 |
| Asus | 10000 |

**Query:** Purchase both mobile and laptop but mobile price should less than laptop price.

Mobile ⋈mobile.price < laptop.price Laptop

Answer:

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Price** | **Model** | **Price** |
| Nokia | 10000 | Dell | 30000 |
| Nokia | 10000 | Acer | 20000 |
| Samsung | 20000 | Dell | 30000 |

**Equi Join:**

Equi join is based on matched data as per the equality condition. The equi join uses the comparison operator (=).

Mobile Laptop

|  |  |
| --- | --- |
| **Model** | **Price** |
| Nokia | 10000 |
| Samsung | 20000 |
| iphone | 50000 |

|  |  |
| --- | --- |
| **Model** | **Price** |
| Dell | 30000 |
| Acer | 20000 |
| Asus | 10000 |

Mobile ⋈mobile.price = laptop.price Laptop

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Price** | **Model** | **Price** |
| Nokia | 10000 | Asus | 10000 |
| Samsung | 20000 | Acer | 20000 |

Natural Join (⋈):

* Natural join can only be performed if there is a common attribute (column) between the relations.
* The name and type (domain) of the attribute must be same.
* Natural join does not use any comparison operator. It does not concatenate the way a Cartesian product does.
* Natural join will also return the similar attributes only once as their value will be same in resulting relation.
* It is a special case of equijoin in which equality condition hold on all attributes which have same name in relations and S.

Example:

**Customer** **Order**

|  |  |  |
| --- | --- | --- |
| **Cid** | **Cname** | **Age** |
| 101 | Naresh | 30 |
| 102 | Narendher | 35 |
| 103 | Nagarjuna | 36 |
| 104 | Neeraj | 40 |

|  |  |  |
| --- | --- | --- |
| **Cid** | **Oname** | **Cost** |
| 101 | Pizza | 500 |
| 103 | Noodles | 200 |
| 104 | Burger | 400 |

Customer ⋈ Order

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Cid** | **Cname** | **Age** | **Oname** | **Cost** |
| 101 | Naresh | 30 | Pizza | 500 |
| 103 | Nagarjuna | 36 | Noodles | 200 |
| 104 | Neeraj | 40 | Burger | 400 |

**Drawbacks of Natural Join:**

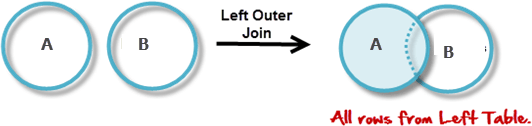
1. Natural join can only be performed if there is a common attribute (column) between the relations.

OUTER JOIN:

In an outer join, along with tuples that satisfy the matching criteria, we also include some or all tuples that do not match the criteria.

Left Outer Join (A  B)

In the left outer join, operation allows keeping all tuple in the left relation. However, if there is no matching tuple is found in right relation, then the attributes of right relation in the join result are filled with null values.



**Example:**

Student Marks Student Marks

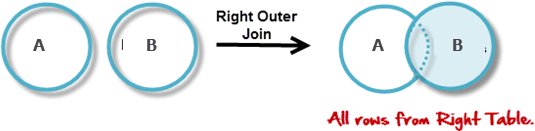
|  |  |
| --- | --- |
| id | Name |
| 101 | Raju |
| 102 | Rishi |
| 103 | Karthik |

|  |  |
| --- | --- |
| id | marks |
| 102 | 92 |
| 103 | 99 |

|  |  |  |  |
| --- | --- | --- | --- |
| id | Name | id | marks |
| 101 | Raju | NULL | NULL |
| 102 | Rishi | 102 | 92 |
| 103 | Karthik | 103 | 99 |

Right Outer Join: (A  B)

In the right outer join, operation allows keeping all tuple in the right relation. However, if there is no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.



Example:

Student Marks Student  Marks

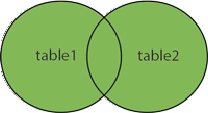
|  |  |
| --- | --- |
| Id | Name |
| 101 | Raju |
| 102 | Rishi |
| 103 | Karthik |

|  |  |
| --- | --- |
| id | marks |
| 102 | 92 |
| 103 | 99 |

|  |  |  |  |
| --- | --- | --- | --- |
| id | Name | id | marks |
| 102 | Rishi | 102 | 92 |
| 103 | Karthik | 103 | 99 |

Full Outer Join / Full Join: ( A B)

In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.





Student Marks Student Marks

|  |  |
| --- | --- |
| Id | Name |
| 101 | Raju |
| 102 | Rishi |
| 103 | Karthik |

|  |  |
| --- | --- |
| id | marks |
| 102 | 92 |
| 103 | 99 |

|  |  |  |  |
| --- | --- | --- | --- |
| id | Name | id | marks |
| 101 | Raju | NULL | NULL |
| 102 | Rishi | 102 | 92 |
| 103 | Karthik | 103 | 99 |
| 102 | Rishi | 102 | 92 |
| 103 | Karthik | 103 | 99 |

Division:

The division operator is used when we have to evaluate queries which contain the keyword ALL.

Some instances where division operator is used are:

* 1. Which person has account in all the banks of a particular city?
  2. Which students have taken all the courses required to graduate? Division operator R1**÷**R2 can be applied if and only if:
* Attributes of R2 is proper subset of Attributes of R1.
* The relation returned by division operator will have attributes = (All attributes of R1 – All Attributes of R2)

R1(x) = R1 (z) / R2(y) Result contains all tuples appear in R1 (z) in combination with every tuple from R2(y) where z=x U y.

Example:

**Students** **Subjects**

|  |  |
| --- | --- |
| **Name** | **Subjects\_failed** |
| Mahesh | DBMS |
| Mahesh | Java |
| Mahender | DBMS |
| Sagar | Java |

|  |
| --- |
| **Subjects\_failed** |
| DBMS |
| Java |

**Query:** Find the names of the students who failed in **all** subjects.

Students (name) = Students (name, Subjects\_failed) / Subjects (Subjects\_failed).

Mahesh

**Name**

Example – 2:

**Submission** **Project**

|  |  |
| --- | --- |
| **Student** | **Task** |
| S1 | DB - 1 |
| S1 | DB - 2 |
| S1 | CN - 1 |
| S2 | CN - 1 |
| S3 | DB - 1 |
| S2 | DB - 2 |
| S3 | DB - 2 |

|  |  |
| --- | --- |
| **Task** | **subject** |
| DB-1 | Database |
| DB-2 | Database |
| CN-1 | Computer Networks |
| CN-2 | Computer Networks |

**Query:** Find the names of the students who have submitted all the database projects.

**ρ** Temp1, ∏**task** (**σ subject=”database”** (Project)



|  |
| --- |
| **Temp1** |
| **Task** |
| DB – 1 |
| DB - 2 |

Submission Temp-1

|  |  |
| --- | --- |
| **Student** | **Task** |
| S1 | DB - 1 |
| S1 | DB - 2 |
| S1 | CN - 1 |
| S2 | CN - 1 |
| S3 | DB - 1 |
| S2 | DB - 2 |
| S3 | DB - 2 |

|  |
| --- |
| **Task** |
| DB-1 |
| DB-2 |

**Submission / Temp1**

|  |
| --- |
| Submission |
| S1 |
| S3 |

More Examples of Algebra Queries

**Sailors** **Reserves**

|  |  |  |  |
| --- | --- | --- | --- |
| **sid** | **sname** | **rating** | **age** |
| 22 | Dustin | 7 | 45.0 |
| 29 | Brutus | 1 | 33.0 |
| 31 | Lubber | 8 | 55.5 |
| 32 | Andy | 8 | 25.5 |
| 58 | Rusty | 10 | 35.0 |
| 64 | Horatio | 7 | 35.0 |
| 71 | Zorba | 10 | 16.0 |
| 74 | Horatio | 9 | 35.0 |
| 85 | Art | 3 | 25.5 |
| 95 | Bob | 3 | 63.5 |

|  |  |  |
| --- | --- | --- |
| **sid** | **bid** | **day** |
| 22 | 101 | 10/10/98 |
| 22 | 102 | 10/10/98 |
| 22 | 103 | 10/8/98 |
| 22 | 104 | 10/7/98 |
| 31 | 102 | 11/10/98 |
| 31 | 103 | 11/6/98 |
| 31 | 104 | 11/12/98 |
| 64 | 101 | 9/5/98 |
| 64 | 102 | 9/8/98 |
| 74 | 103 | 9/8/98 |

Boats

|  |  |  |
| --- | --- | --- |
| **bid** | **bname** | **color** |
| 101 | Interlake | Blue |
| 102 | Interlake | Red |
| 103 | Clipper | Green |
| 104 | Marine | red |

**1. Find the names of sailors who have reserved boat 103.**

***σbid=103 ( Reserves )***

***Temp1***

|  |  |  |
| --- | --- | --- |
| *sid* | *bid* | *day* |
| 22 | 103 | 10/8/98 |
| 31 | 103 | 11/6/98 |
| 74 | 103 | 9/8/98 |

###### Temp1 ⋈ Sailors

***Temp2***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *sid* | *sname* | *rating* | *Age* | *bid* | *Day* |
| 22 | Dustin | 7 | 45.0 | 103 | 10/8/98 |
| 31 | Lubber | 8 | 55.5 | 103 | 11/6/98 |
| 74 | Horatio | 9 | 35.0 | 103 | 9/8/98 |

***πsname Temp2***

|  |
| --- |
| ***sname*** |
| Dustin |
| Lubber |
| Horatio |

*Note: There are of course several ways to express Q1 in relational algebra. Here is another:*

***πsname(σbid=103(Reserves***⋈ ***Sailors))***

**(Q2) Find the names of sailors who have reserved a red boat.**

**Sailors** **Reserves**

|  |  |  |  |
| --- | --- | --- | --- |
| **sid** | **sname** | **rating** | **age** |
| 22 | Dustin | 7 | 45.0 |
| 29 | Brutus | 1 | 33.0 |
| 31 | Lubber | 8 | 55.5 |
| 32 | Andy | 8 | 25.5 |
| 58 | Rusty | 10 | 35.0 |
| 64 | Horatio | 7 | 35.0 |
| 71 | Zorba | 10 | 16.0 |
| 74 | Horatio | 9 | 35.0 |
| 85 | Art | 3 | 25.5 |
| 95 | Bob | 3 | 63.5 |

|  |  |  |
| --- | --- | --- |
| **sid** | **bid** | **day** |
| 22 | 101 | 10/10/98 |
| 22 | 102 | 10/10/98 |
| 22 | 103 | 10/8/98 |
| 22 | 104 | 10/7/98 |
| 31 | 102 | 11/10/98 |
| 31 | 103 | 11/6/98 |
| 31 | 104 | 11/12/98 |
| 64 | 101 | 9/5/98 |
| 64 | 102 | 9/8/98 |
| 74 | 103 | 9/8/98 |

Boats

|  |  |  |
| --- | --- | --- |
| **bid** | **bname** | **color** |
| 101 | Interlake | Blue |
| 102 | Interlake | Red |
| 103 | Clipper | Green |
| 104 | Marine | red |

**ρ(Temp1, (σ*color=‘red’* Boats))**

##### Temp1

|  |  |  |
| --- | --- | --- |
| bid | bname | color |
| 102 | Interlake | red |
| 104 | Marine | red |

**ρ(Temp2, Temp1** ⋈ ***Reserves)***

##### Temp2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bid | bname | color | sid | day |
| 102 | Interlake | red | 22 | 10/10/98 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 102 | Interlake | red | 31 | 11/10/98 |
| 102 | Interlake | red | 64 | 9/8/98 |
| 104 | Marine | red | 22 | 10/7/98 |
| 104 | Marine | red | 31 | 11/12/98 |

**π*sname* ( Temp2** ⋈ **Sailors )**

|  |
| --- |
| **sname** |
| Dustin |
| Horatio |
| Lubber |

###### (Q3) Find the colors of boats reserved by Lubber.

**π*color*((σ*sname=‘Lubber’* Sailors** ⋈ **Reserves** ⋈ **Boats)**

###### (Q4) Find the names of Sailors who have reserved at least one boat.

**π*sname*(Sailors** ⋈ **Reserves)**

###### (Q5) Find the names of sailors who have reserved a red or a green boat.

**ρ(*Tempred*, π*sid*((σ*color=‘red’Boats*)** ⋈ ***Reserves*)) ρ(*Tempgreen*, π*sid*((σ*color=‘green’Boats*)** ⋈ ***Reserves*)) π*sname* ((*Tempred* U *Tempgreen*)** ⋈ ***Sailors*)**

###### (Q6) Find the names of Sailors who have reserved a red and a green boat.

**ρ(*Tempred*, π*sid*((σ*color=‘red’Boats*)** ⋈ ***Reserves*))**

**ρ(*Tempgreen*, π*sid*((σ*color=‘green’Boats*)** ⋈ ***Reserves*))**

**π*sname* ((*Tempred* ∩ *Tempgreen*)** ⋈ ***Sailors*)**

###### (Q7) Find the names of sailors who have reserved at least two boats.

**Sailors** **Reserves**

|  |  |  |  |
| --- | --- | --- | --- |
| **sid** | **sname** | **rating** | **age** |
| 22 | Dustin | 7 | 45.0 |
| 29 | Brutus | 1 | 33.0 |
| 31 | Lubber | 8 | 55.5 |
| 32 | Andy | 8 | 25.5 |
| 58 | Rusty | 10 | 35.0 |
| 64 | Horatio | 7 | 35.0 |
| 71 | Zorba | 10 | 16.0 |
| 74 | Horatio | 9 | 35.0 |
| 85 | Art | 3 | 25.5 |
| 95 | Bob | 3 | 63.5 |

|  |  |  |
| --- | --- | --- |
| **sid** | **bid** | **day** |
| 22 | 101 | 10/10/98 |
| 22 | 102 | 10/10/98 |
| 22 | 103 | 10/8/98 |
| 22 | 104 | 10/7/98 |
| 31 | 102 | 11/10/98 |
| 31 | 103 | 11/6/98 |
| 31 | 104 | 11/12/98 |
| 64 | 101 | 9/5/98 |
| 64 | 102 | 9/8/98 |
| 74 | 103 | 9/8/98 |

###### ρ(Reservations, πsid,sname,bid(Sailors ⋈ Reserves))

|  |  |  |
| --- | --- | --- |
| **Reservations** | | |
| **sid** | **sname** | **bid** |
| 22 | Dustin | 101 |
| 22 | Dustin | 102 |
| 22 | Dustin | 103 |
| 22 | Dustin | 104 |
| 31 | Lubber | 102 |
| 31 | Lubber | 103 |
| 31 | Lubber | 104 |
| 64 | Horatio | 101 |
| 64 | Horatio | 102 |
| 74 | Horatio | 103 |

**ρ(*Reservationpairs*(1🠒*sid1*, 2🠒*sname1*, 3🠒*bid1*, 4🠒*sid2*, 5🠒*sname2*, 6🠒*bid2*), *Reservations*🞪*Reservations*) π*sname1*σ*(sid1=sid2)∧(bid1≠bid2)Reservationpairs*)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **sid1** | **sname1** | **bid1** | **sid2** | **sname2** | **bid2** |
| 22 | Dustin | 101 | 22 | Dustin | 101 |
| 22 | Dustin | 101 | 22 | Dustin | 102 |
| 22 | Dustin | 101 | 22 | Dustin | 103 |
| 22 | Dustin | 101 | 22 | Dustin | 104 |
| 22 | Dustin | 101 | 31 | Lubber | 102 |
| 22 | Dustin | 101 | 31 | Lubber | 103 |
| 22 | Dustin | 101 | 31 | Lubber | 104 |
| 22 | Dustin | 101 | 64 | Horatio | 101 |
| 22 | Dustin | 101 | 64 | Horatio | 102 |
| 22 | Dustin | 101 | 74 | Horatio | 103 |

.

.

.

.

###### (Q8) Find the sids of sailors with age over 20 who have not reserved a red boat.

Sailors Reserves

|  |  |  |  |
| --- | --- | --- | --- |
| **sid** | **sname** | **rating** | **age** |
| 22 | Dustin | 7 | 45.0 |
| 29 | Brutus | 1 | 33.0 |
| 31 | Lubber | 8 | 55.5 |
| 32 | Andy | 8 | 25.5 |
| 58 | Rusty | 10 | 35.0 |
| 64 | Horatio | 7 | 35.0 |
| 71 | Zorba | 10 | 16.0 |
| 74 | Horatio | 9 | 35.0 |
| 85 | Art | 3 | 25.5 |
| 95 | Bob | 3 | 63.5 |

|  |  |  |
| --- | --- | --- |
| **sid** | **bid** | **day** |
| 22 | 101 | 10/10/98 |
| 22 | 102 | 10/10/98 |
| 22 | 103 | 10/8/98 |
| 22 | 104 | 10/7/98 |
| 31 | 102 | 11/10/98 |
| 31 | 103 | 11/6/98 |
| 31 | 104 | 11/12/98 |
| 64 | 101 | 9/5/98 |
| 64 | 102 | 9/8/98 |
| 74 | 103 | 9/8/98 |

**Boats**

|  |  |  |
| --- | --- | --- |
| **bid** | **bname** | **color** |
| 101 | Interlake | Blue |
| 102 | Interlake | Red |
| 103 | Clipper | Green |
| 104 | Marine | red |

###### πsid(σage>20Sailors)

|  |
| --- |
| **sailors** |
| **sid** |
| 22 |
| 29 |
| 31 |
| 32 |
| 58 |
| 64 |
| 74 |
| 85 |
| 95 |

**π*sid*((σ*color=‘red’Boats)*** ⋈ ***Reserves*** ⋈ ***Sailors*)**

|  |
| --- |
| sid |
| 22 |
| 31 |
| 64 |

Set difference

|  |
| --- |
| **sailors** |
| **sid** |
| 22 |
| 29 |
| 31 |
| 32 |
| 58 |
| 64 |
| 74 |
| 85 |
| 95 |

|  |
| --- |
| sid |
| 22 |
| 31 |
| 64 |

|  |
| --- |
| **sailors** |
| **sid** |
| 29 |
| 32 |
| 58 |
| 74 |
| 85 |
| 95 |

RELATIONAL CALCULUS

Contrary to Relational Algebra which is a procedural query language to fetch data and which also explains how it is done, **Relational Calculus** in non-procedural query language and has no description about how the query will work or the data will be fetched. It only focuses on what to do, and not on how to do it.

Types of Relational Calculus

1. Tuple Relational Calculus
2. Domain Relational Calculus.

1. Tuple Relational Calculus

Tuple relational calculus is used for selecting those tuples that satisfy the given condition.

A tuple relational calculus query has the form

{T | P(t)}

Where t = resulting tuples,

P(t) = known as Predicate and these are the conditions that are used to fetch t Thus, it generates set of all tuples t, such that Predicate P(t) is true for t.

A **free variable** is a variable that has no limitations. For example, the *x* in this function is a free variable.

* f(*x*) = 3*x* - 1

Why is it a free variable? It's a free variable because you don't see any limitations put on it. It can equal a 1 or a -1, a 10 or even -1,000. Also, this function depends on this *x* variable. So, the *x* here is a free variable.

A **bound variable**, on the other hand, is a variable with limitations. An example of a bound variable is this one.



Here, the *x* is the bound variable. This expression specifies that the value of your *x* goes from 1 to 4 in this summation. Because the *x* is a bound variable with its values already chosen, the expression isn't dependent on this variable.

Syntax of TRC Queries:

We now define these concepts formally, beginning with the notion of a formula. Let *Rel* be a relation name, R and *S* be tuple variables, *a* be an attribute of *R,* and *b* be an attribute of *S.* Let op denote an operator in the set {<, >, =,<=,>=,!=}. An atomic (which cannot be further divided) formula is one of the following:

* R ϵ Rel
* R.a op S.b
* R.a op constant

A formula is recursively defined to be one of the following, where *p* and *q* are themselves formulas and *p(R)* denotes a formula in which the variable *R* appears: P(t) may have various conditions logically combined with OR (𝗏), AND (𝖠), NOT(¬). It also uses quantifiers:

∃ t ∈ r (Q(t)) = ”there exists” a tuple in t in relation r such that predicate Q(t) is true.

∀ t ∈ r (Q(t)) = Q(t) is true “for all” tuples in relation r.

Example:

1. **Find all sailors with a rating above 7.**

*{S* | *S* ϵ *Sailors* ᴧ *S.rating* > 7}

1. Find the names of sailors who’ve reserved boat number 103

{*T* | **∃**S **∈***Sailors* (T.name = S.name  **∃**R **∈** Reserves (S.sid = R.sid

 R.bid = 103))}

1. Find the names of sailors who’ve reserved any red boat

{*T*| **∃**S **∈***Sailors* (T.name = S.name  **∃**R **∈** Reserves (S.sid = R.sid  **∃**B **∈**

Boats (B.color = “Red”  B.bid = R.bid)))}

1. Find sailors who’ve reserved a red boat or a green boat

{*S* | S **∈***Sailors* 

(**∃**R **∈** Reserves

(S.sid = R.sid 

**∃**B **∈** Boats(B.bid = R.bid 

(B.color = “Red” 

B.color = “Green” ))))}

**Domain Relational Calculus:** In this, filtering is done based on the domain of the attributes and not based on the tuple values.

A DRC query has the form *{<X1,X2, ... ,Xn >* | *P(<XI,X2, ... ,Xn >)},* where each *Xi* is either a *domain variable* or a constant and *p(* <Xl, *X2,* ... *,xn >)* denotes a **DRC formula** whose only free variables are the variables among the *Xi,* 1 Sis *n.* The result of this query is the set of all tuples *(Xl, X2,* ... , *xn )* for which the formula evaluates to true.

A DRC formula is defined in a manner very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables. Let op denote an operator in the set {<, >, =, S,~, i-} and let X and *Y* be domain variables. An **atomic formula** in DRC is one of the following:

* *(X1,X2, ... ,Xn )* ϵ *Rel,* where *Rei* is a relation with *n* attributes; each *Xi,* 1 ≤ I

≤n is either a variable or a constant.

* X *op* Y
* X *op constant, or constant op* X*.*

A **formula** is recursively defined to be one of the following, where *p* and *q* are themselves formulas and *p(X)* denotes a formula in which the variable X appears:

* any atomic formula
* *¬p, P* /\ *q, P* V *q,* or *p* ⇒ *q*
* ∃ *(p(X)),* where *X* is a domain variable
* ∀*X(p(X)),* where *X* is a domain variable

***Examples:***

(Q1) Find all sailors with a rating above 7.

{〈1, *N, T, A*〉 | 〈*I, N, T, A*〉 ϵ *Sailors* /\ *T* > 7} (Q2) Find the names of sailors who have reserved boat 103.

{〈N〉| ∃I,T,A(〈I,N,T,A〉∈Sailors𝖠∃ Ir,Br,D(〈Ir,Br,D ∈Reserves 𝖠 Ir

=I 𝖠 Br=103))}

UNIT - III

**Form of basic SQL query**

We will present a number of sample queries using the following table definitions: Sailors (*sid:* integer, *sname:* string, *rating:* integer, *age:* real)

Boats (*bid:* integer, *bname:* string, *color:* string)

Reserves (*sid:* integer, *bid:* integer, *day:* date)

Queries:

* CREATE TABLE sailors (sid int PRIMARY KEY, sname varchar (50), rating integer ,age int)
* CREATE TABLE boats(bid int PRIMARY KEY,bname varchar(50),color varchar(50));
* CREATE TABLE reserves (sid int,bid int,day date,PRIMARY KEY (sid,bid,day) ,FOREIGN KEY(sid) REFERENCES sailors(sid), FOREIGN KEY(bid) REFERENCES boats(bid));
* INSERT INTO sailors VALUES (22,"Dustin",7,45),(29,"Brutus",1,33),(31,"Lubber",8,55),(32,"Andy",8,25), (58,"Rusty", 10, 35),(64,"Horatio",7,35),(71,"Zorba",10,16),

(74,"Horatio", 9, 35),(85,"Art",3,25),(95,"Bob",3,63);

* INSERT INTO boats VALUES (101,"Interlake","Blue"), (102,"Interlake","Red"), (103,"Clipper","Green"), (104,"Marine","red");
* INSERT INTO reserves VALUES

(22, 101, 10/10/98), (22, 102, 10/10/98), (22,103,10/8/98), (22,104,10/7/98),

(31, 102, 11/10/98), (31, 103, 11/6/98), (31,104, 11/12/98), (64,101,9/5/98),

(64, 102, 9/8/98), (74, 103, 9/8/98);

This section presents the syntax of a simple SQL query and explains its meaning through a *conceptual Evaluation strategy.* A conceptual evaluation strategy is a way to evaluate the query that is intended to be easy to understand rather than efficient. A DBMS would typically execute a query in a different and more efficient way.

The basic form of an SQL query is as follows:

SELECT [DISTINCT] select-list FROM from-list

WHERE qualification

Example:

SELECT DISTINCT S.sname, S.age FROM Sailors S

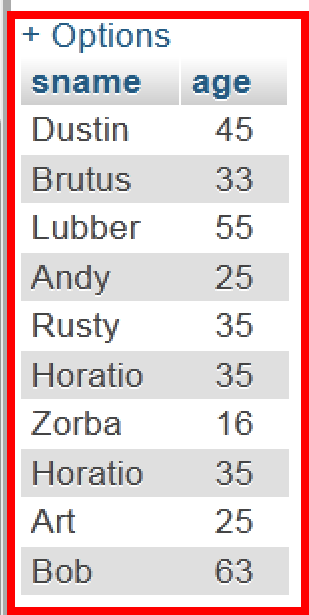
Output:

The DISTINCT keyword is optional. It indicates that the table computed as an answer to this query should not contain *duplicates.*

If two or more sailors have the same name and age, the answer still contains just one pair. This query is equivalent to the projection operator of relational algebra.

Example:

SELECT S.sname, S.age FROM Sailors S

Output:

**Example:**

***Q: Find all sailors with a rating above 7.***

*A*: SELECT S.sid, S.sname, S.rating, S.age FROM Sailors AS S WHERE S.rating > 7

Or

SELECT sid, sname, rating, age from sailors where rating > 7

Output:



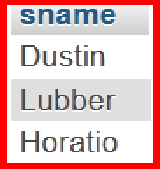
**AS** is an optional keyword used to assign temporary names to table or column name or both. This is known as creating **Alias in SQL.**

SELECT clause is actually used to do *projection,* whereas *selections* in the relational algebra sense are expressed using the WHERE clause.

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

***A:*** *SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid AND R.bid=103*

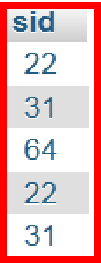
Output:



**Q: *Find the sids of sailors who have reserved a red boat.***

***A*:** SELECT R.sid FROM Boats B, Reserves R WHERE B.bid = R.bid AND B.color = 'red';

***Output:***



**Q: *Find the names of sailors who have reserved a red boat.***

***A:*** SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red';

***Output:***



**Q: *Find the colors of boats reserved by Lubber.***

**A:** SELECT B.color FROM Sailors S, Reserves R, Boats B WHERE R.bid = B.bid AND S.sid = R.sid AND S.sname = 'Lubber';

***Output:***



Like Operator:

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

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

* The percent sign (%) represents zero, one, or multiple characters
* The underscore sign (\_) represents one, single character Sailors

|  |  |  |  |
| --- | --- | --- | --- |
| **sid** | **sname** | **rating** | **age** |
| 22 | Dustin | 7 | 45.0 |
| 29 | Brutus | 1 | 33.0 |
| 31 | Lubber | 8 | 55.5 |
| 32 | Andy | 8 | 25.5 |
| 58 | Rusty | 10 | 35.0 |
| 64 | Horatio | 7 | 35.0 |
| 71 | Zorba | 10 | 16.0 |
| 74 | Horatio | 9 | 35.0 |
| 85 | Art | 3 | 25.5 |
| 95 | Bob | 3 | 63.5 |

Q: Find the sailors details whose name starts with ‘D’.

*A:* SELECT \* from sailors where sname like "D%";

**Q: Find the sailors details whose name ends with ‘Y’.** *A:* SELECT \* from sailors WHERE sname LIKE "%Y"; ***Q:* Find the sailors details whose name contains “bb”.** *A:* SELECT \* from sailors WHERE sname LIKE "%bb%";

*Q:* Find the sailors details whose name contain ‘o’ in second place.

***A:*** SELECT \* FROM sailors where sname LIKE "\_o%";

*Q:* Find the sailors details whose name contain “s” in the third place and name should contain total of five characters.

***A:*** SELECT \* FROM sailors WHERE sname like "\_\_s\_\_";

*Q:* Find the details of sailors whose names does not starts with “a”.

***A:*** SELECT \* FROM sailors WHERE sname NOT LIKE "a%";

SELECT \* FROM sailors WHERE sname LIKE '%a%' INTERSECT

SELECT \* FROM sailors WHERE sname NOT LIKE '%u%'

UNION, INTERSECT AND EXCEPT

**UNION:**

Union is an operator that allows us to combine two or more results from multiple SELECT queries into a single result set. It comes with a default feature that removes the **duplicate** rows from the result set. MySQL always uses the name of the column in the first SELECT statement will be the column names of the result set (output).

Union must follow these basic rules:

* The number and order of the columns should be the same in all tables that you are going to use.
* The data type must be compatible with the corresponding positions of each select query.
* The column name selected in the different SELECT queries must be in the same order.

Syntax:

SELECT expression1, expression2, expression\_n FROM tables

[WHERE conditions]

UNION

SELECT expression1, expression2, expression\_n FROM tables

[WHERE conditions];

**Q: *Find the sid’s and names of sailors who have reserved a red or a green boat.***

*A:* SELECT S.sid,S.sname

FROM Sailors S, Reserves R, Boats B

WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'

union

SELECT S2.sid,S2.sname

FROM Sailors S2, Boats B2, Reserves R2

WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green';

Output:



**INTERSECT:**

The INTERSECT operator is used to fetch the records that are in common between two SELECT statements or data sets. If a record exists in one query and not in the other, it will be omitted from the INTERSECT results.

Syntax:

SELECT expression1, expression2, ... expression\_n FROM tables

[WHERE conditions]

INTERSECT

SELECT expression1, expression2, ... expression\_n FROM tables

[WHERE conditions];

Intersection must follow these basic rules:

* The number and order of the columns should be the same in all tables that you are going to use.
* The data type must be compatible with the corresponding positions of each select query.
* The column name selected in the different SELECT queries must be in the same order.

**Q: *Find the sid’s and names of sailors who have reserved a red and a green boat. A:*** SELECT S.sid,S.sname

FROM Sailors S, Reserves R, Boats B

WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'

INTERSECT

SELECT S2.sid,S2.sname

FROM Sailors S2, Boats B2, Reserves R2

WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green';

Output:



**EXCEPT:**

The EXCEPT clause in [SQL](https://www.javatpoint.com/sql-tutorial) is widely used to filter records from more than one table. This statement first combines the two [SELECT statements](https://www.javatpoint.com/sql-select) and returns records from the first SELECT query, which isn’t present in the second SELECT query's result. In other words, it retrieves all rows from the first SELECT query while deleting redundant rows from the second.

This statement behaves the same as the minus (set difference) operator does in mathematics.

Rules for SQL EXCEPT

We should consider the following rules before using the EXCEPT statement in SQL:

* In all SELECT statements, the number of columns and orders in the tables must be the same.
* The corresponding column's data types should be either the same or compatible.
* The fields in the respective columns of two SELECT statements cannot be the same.

Syntax:

SELECT column\_lists from table\_name1 EXCEPT

SELECT column\_lists from table\_name2;

Q: Find the sids of all sailors who have reserved red boats but not green boats.

**A:** SELECT S.sid

FROM Sailors S, Reserves R, Boats B

WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'

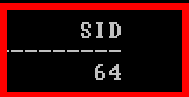
MINUS // minus in Oracle, Except in SQL server

SELECT S2.sid

FROM Sailors S2, Boats B2, Reserves R2

WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green';

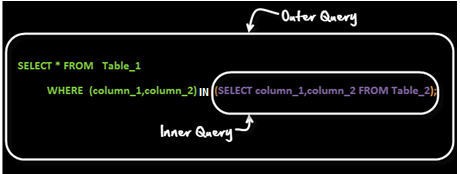
**Output:**



Nested Queries

A nested query is a query that has another query embedded within it; the embedded query is called a sub query. A sub query is known as the inner query**,** and the query that contains sub query is known as the outer query**.**

Syntax:



As per the execution process of sub query, it again classified in two ways.

1. Non- Correlated sub queries.

In this sub queries, First inner query is executed and later Outer query will execute. i.e Outer query always depends on the result of inner query.

1. Correlated Queries.

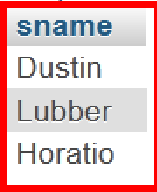
In correlated queries, First Outer query is executed and later Inner query will execute. i.e Inner query always depends on the result of outer query.

Example for Non – Correlated Queries:

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

***A:*** SELECT S.sname from sailors S WHERE S.sid IN (SELECT R.sid FROM reserves R WHERE R.bid=103);

Output:

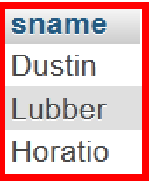


SQL IN condition used to allow multiple value in a WHERE clause condition. SQL IN condition you can use when you need to use multiple OR condition.

***(Q) Find the names of sailors who have reserved a red boat.***

***A:*** SELECT S.sname from sailors S WHERE S.sid IN

(SELECT R.sid FROM reserves R where R.bid IN (SELECT B.bid FROM boats B WHERE B.color="red"));

Output:

***(Q) Find the names of sailors who have* not *reserved a red boat.***

***A:*** SELECT S.sname from sailors S WHERE S.sid NOT IN

(SELECT R.sid FROM reserves R where R.bid IN (SELECT B.bid FROM boats B WHERE B.color="red"));

Output:



SQL NOT IN condition used to exclude the defined multiple value in a WHERE clause condition.

Correlated Sub Queries:

**Correlated sub queries** are the one in which **inner query or sub query reference outer query**. Outer query needs to be executed before inner query. One of the most common examples of correlated sub query is using keywords exits and not exits.

Example:

*(Q*) *Find the names of sailors who have reserved boat number 103*

SELECT S.sname FROM Sailors S WHERE

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

Aggregation Operators:

SQL aggregation function is used to perform the calculations on multiple rows of a single column of a table. It returns a single value. It is also used to summarize the data.

SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

Example:

CREATE TABLE employees(id int,name char(50),dob date,gender char(5),sal int,dept char(10),collegename char(15),age int);

DESC employees;

INSERT INTO employees VALUES (1,"Srikanth",'1990-05-05','M',25000,'cse','kits',25),

(2,"Kamal",'1992-02-02','M',65000,'ece','kitsw',29),

(3,"kavitha",'1993-01-01','F',45000,'cse','kits',30),

(4,"Karun",'1988-06-04','M',25000,'cse','kits',31),

(5,"Gouthami",'1980-02-09','F',10000,'civil','kits',29),

(6,"Narendher",'1985-09-18','M',30000,'civil','kits',35),

(7,"Mahesh",'1990-03-06','M',8000,'cse','kitsw',45);

(8, NULL,"1992-06-09",'F',65000,"ece",'kitsw',46),

(9,"Sravanthi","1991-07-04",'F',46000,"CSE",'kitsw',29),

(1,"Neeraja","1991-07-04",'F',50000,"CSE",'kitsw',26);

Queries:

SELECT COUNT(\*) from employees; **// Output=10** SELECT COUNT(name) FROM employees; **// Output=9** SELECT COUNT(id) FROM employees; **// Output=10**

SELECT COUNT(DISTINCT id) FROM employees; **// Output=9**

SELECT COUNT(\*) FROM employees WHERE collegename ="kitsw";

// Output=5

SELECT MAX(sal) FROM employees; **// Output=65000**

SELECT MAX(sal) as Maximum\_salary FROM employees; **// Output=65000**

SELECT MAX(sal) FROM employees WHERE collegename="kits";

// Output=65000

SELECT MIN(sal) FROM employees; **// Output=8000**

SELECT MIN(sal) FROM employees WHERE collegename='kits';

// Output=8000

SELECT SUM(sal) FROM employees; **// Output=369000**

SELECT sum(sal) as total\_salary FROM employees; **// Output=369000**

SELECT AVG(sal) FROM employees; **// Output=36900**

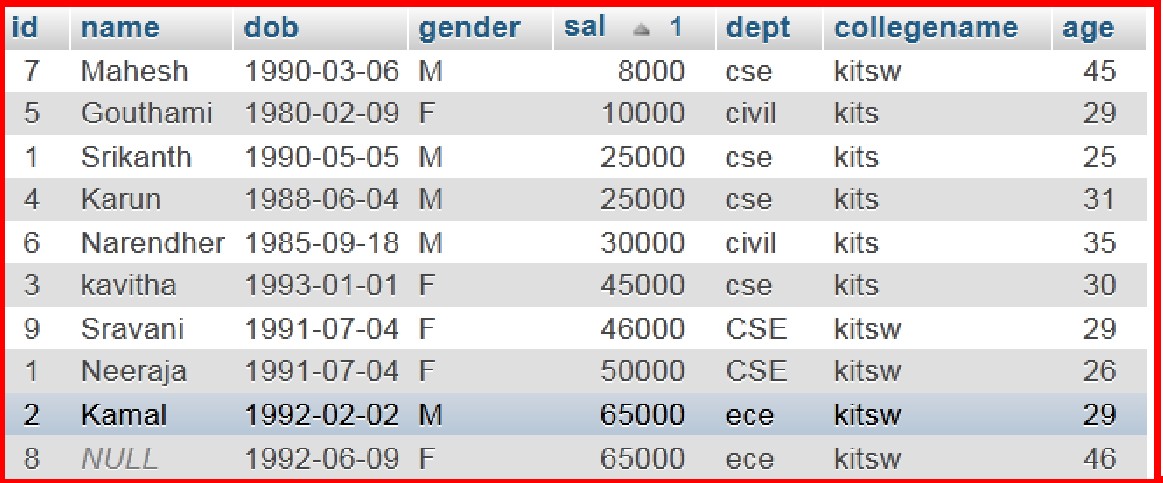
Order by Clause:

Order by clause is used sort the rows either in ascending order or descending order (default is ascending)

Example:

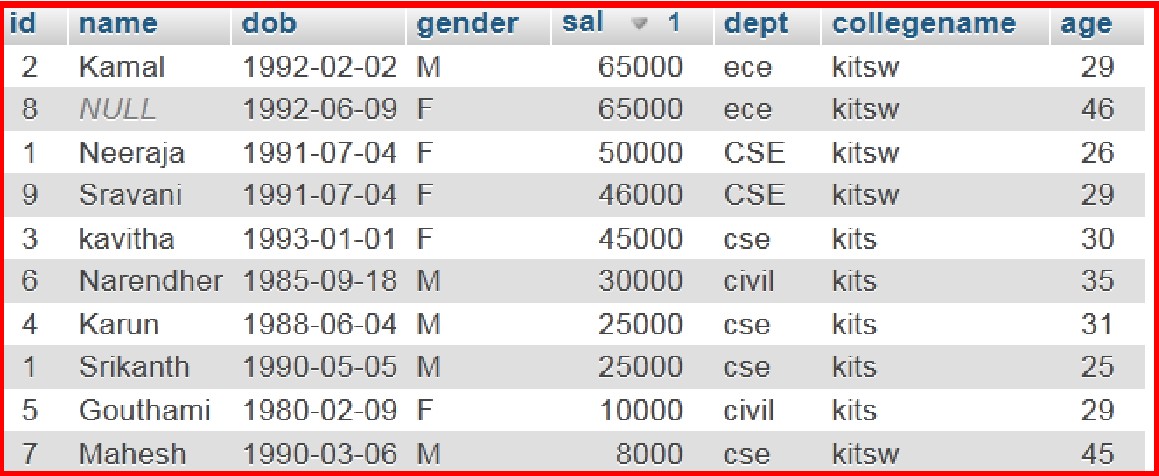
SELECT \* from employees ORDER BY sal ASC;

Output:



SELECT \* FROM employees ORDER BY sal DESC;

Output:



**Group By clause:**

The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e if a particular column has same values in different rows then it will arrange these rows in a group.

Important Points:

* GROUP BY clause is used with the SELECT statement.
* In the query, GROUP BY clause is placed after the WHERE clause.
* In the query, GROUP BY clause is placed before ORDER BY clause if used any.

Syntax:

SELECT column1, function\_name(column2) FROM table\_name

WHERE condition

GROUP BY column1, column2 ORDER BY column1, column2;

**function\_name**: Name of the function used for example, SUM() , AVG().

**table\_name**: Name of the table.

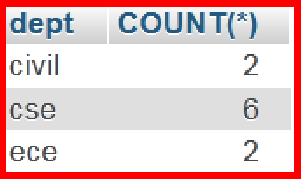
**condition**: Condition used.

Examples:

**Q: Display the Number of employees in each department.**

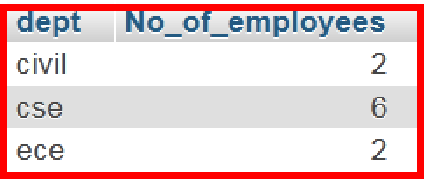
A: SELECT dept,COUNT(\*) FROM employees GROUP BY dept;

Output:



SELECT dept,COUNT(\*) AS No\_of\_employees FROM employees GROUP BY dept;

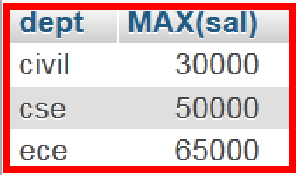
Output:



**Q: What is the highest salary in each department?**

***A:*** SELECT dept,MAX(sal) from employees GROUP BY dept;

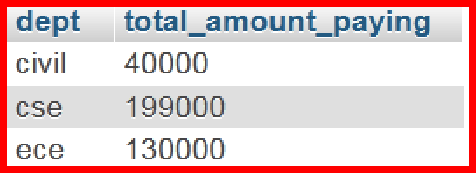
Output:



**Q: What is the total salary each department is paying?**

***A:*** SELECT dept,SUM(sal) AS total\_amount\_paying from employees GROUP BY dept;

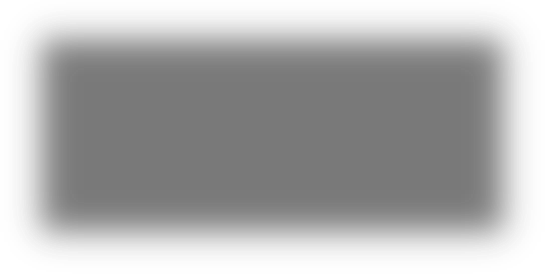
Output:



**Q: Display the Number of male and female faculty.**

**A:** SELECT gender,COUNT(\*) FROM employees GROUP BY gender;

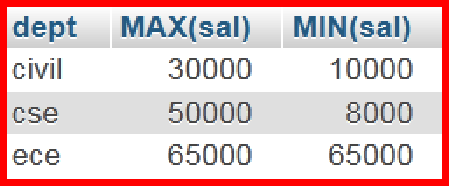
Output:



**Q: What is the highest and lowest salary where each dept is paying.**

***A:*** SELECT dept,MAX(sal),MIN(sal) FROM employees GROUP BY dept;

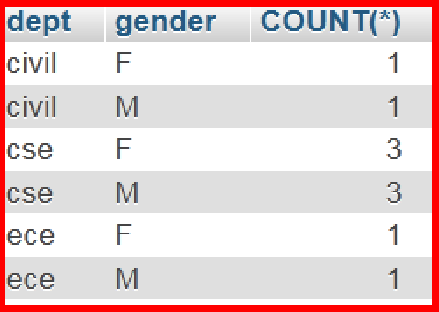
Output:



**Q: Display the number of male and female faculty from each Department.**

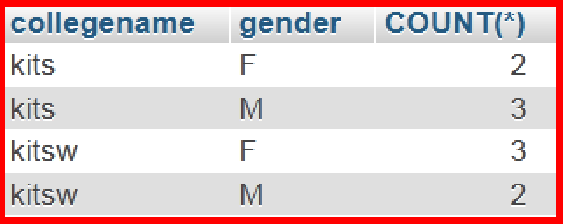
**A:** SELECT dept,gender,COUNT(\*) FROM employees GROUP BY dept, gender;

Output:



**Q: Display the number of male and female’s faculty from each college. *A:*** SELECT collegename,gender,COUNT(\*) from employees GROUP BY collegename , gender;

Output:



**Having Clause:**

The HAVING clause is like WHERE but operates on grouped records returned by a GROUP BY. HAVING applies to summarized group records, whereas WHERE applies to individual records. Only the groups that meet the HAVING criteria will be returned.

HAVING requires that a GROUP BY clause is present.

Both WHERE and HAVING can be used in the same query at the same time.

Syntax:

SELECT column-names FROM table-name WHERE condition GROUP BY column-names HAVING condition;

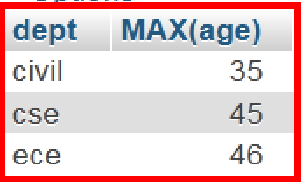
Example:

**Query:** SELECT dept,COUNT(\*) FROM employees GROUP BY dept HAVING COUNT(\*)>2;

Output:

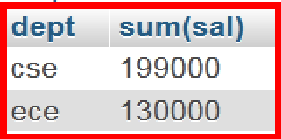
**Query:** SELECT dept,MAX(age) FROM employees GROUP BY dept HAVING MAX(sal)>30;

Output:



**Query:** SELECT dept,sum(sal) FROM employees GROUP BY dept HAVING SUM(sal) > 50000;

Output:



**NULL**

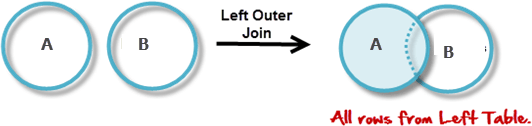
We have assumed that column values in a row are always known. In practice column values can be unknown. For example, when a sailor, say Dan, joins a yacht club, he may not yet have a rating assigned. Since the definition for the Sailors table has a rating column, what row should we insert for Dan? What is needed here is a special value that denotes unknown. Suppose the Sailor table definition was modified to include a maiden-name column. However, only married women who take their husband's last name have a maiden name. For women who do not take their husband's name and for men, the maiden-name column is inapplicable. Again, what value do we include in this column for the row representing Dan? SQL provides a special column value called null to use in such situations. We use null when the column value is either Unknown or inapplicable. Using our Sailor table definition, we might enter the row (98. Dan, null, 39) to represent Dan. The presence of null values complicates many issues, and we consider the impact of null values on SQL in this section.

OUTER JOIN:

In an outer join, along with tuples that satisfy the matching criteria, we also include some or all tuples that do not match the criteria.

Left Outer Join (A  B)

In the left outer join, operation allows keeping all tuple in the left relation. However, if there is no matching tuple is found in right relation, then the attributes of right relation in the join result are filled with null values.



**Example:**

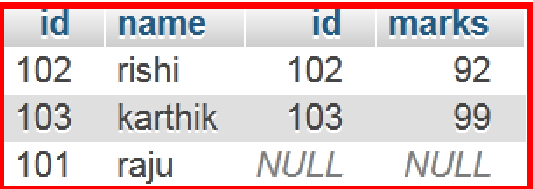
Student Marks Student Marks

|  |  |
| --- | --- |
| id | Name |
| 101 | Raju |
| 102 | Rishi |
| 103 | Karthik |

|  |  |
| --- | --- |
| id | marks |
| 102 | 92 |
| 103 | 99 |

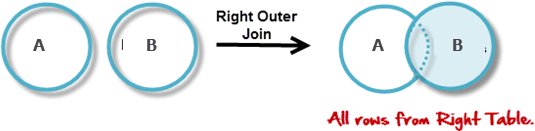
|  |  |  |  |
| --- | --- | --- | --- |
| id | Name | id | marks |
| 101 | Raju | NULL | NULL |
| 102 | Rishi | 102 | 92 |
| 103 | Karthik | 103 | 99 |

* CREATE TABLE student (id int, name char(50));
* INSERT INTO student VALUES (101,"raju"),(102,"rishi"),(103,"karthik");
* CREATE TABLE marks (id int,marks int);
* INSERT INTO marks values(102,92),(103,99);
* SELECT \* from student LEFT OUTER JOIN marks on student.id=marks.id;

Output:

**Right Outer Join: (A ** **B)**

In the right outer join, operation allows keeping all tuple in the right relation. However, if there is no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.



Example:

Student Marks Student  Marks

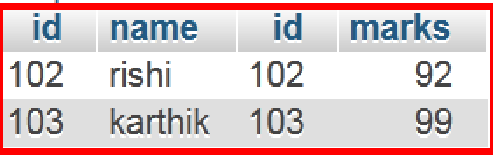
|  |  |
| --- | --- |
| Id | Name |
| 101 | Raju |
| 102 | Rishi |
| 103 | Karthik |

|  |  |
| --- | --- |
| id | marks |
| 102 | 92 |
| 103 | 99 |

|  |  |  |  |
| --- | --- | --- | --- |
| id | Name | id | marks |
| 102 | Rishi | 102 | 92 |
| 103 | Karthik | 103 | 99 |

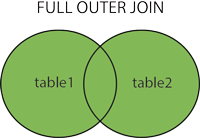
* SELECT \* from student RIGHT OUTER JOIN marks on student.id=marks.id

**Output:**



Full Outer Join / Full Join: ( A  B)

In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.



Student Marks Student  Marks

|  |  |
| --- | --- |
| Id | Name |
| 101 | Raju |
| 102 | Rishi |
| 103 | Karthik |

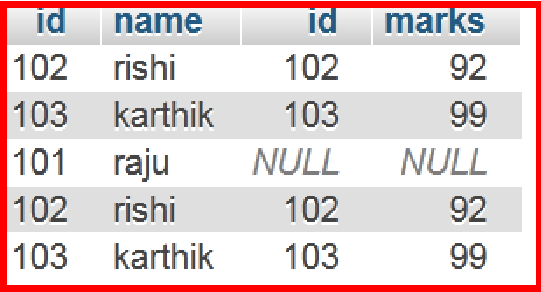
|  |  |
| --- | --- |
| id | marks |
| 102 | 92 |
| 103 | 99 |

|  |  |  |  |
| --- | --- | --- | --- |
| id | Name | id | marks |
| 101 | Raju | NULL | NULL |
| 102 | Rishi | 102 | 92 |
| 103 | Karthik | 103 | 99 |
| 102 | Rishi | 102 | 92 |

|  |  |  |  |
| --- | --- | --- | --- |
| 103 | Karthik | 103 | 99 |

* SELECT \* from student LEFT OUTER JOIN marks on student.id=marks.id UNION ALL

SELECT \* from student RIGHT OUTER JOIN marks on student.id=marks.id

Output:

**SELECT \* from student FULL JOIN marks on student.id=marks.id //sql server**

Triggers (MySQL)

Triggers are the [SQL](https://en.wikipedia.org/wiki/SQL) codes that are automatically executed in response to certain events on a particular table. These are used to maintain the integrity of the data.

It is name PL (procedural language) / SQL block.

To understand this let’s consider a hypothetical situation.

John is the marketing officer in a company. When a new customer data is entered into the company’s database he has to send the welcome message to each new customer. If it is one or two customers John can do it manually, but what if the count is more than a thousand? Well in such scenario triggers come in handy.

Thus, now John can easily create a trigger which will automatically send a welcome email to the new customers once their data is entered into the database.

Important points:

* To maintain Business data in Uniform case.
* To maintain audit information of any table data.

Triggers can be made to insert, update and delete statements in SQL. We have two types of triggers:

1. Row Level Triggers

In a row-level trigger, the changes are put on all the rows on which the insert, delete, or update transaction is performed.

If we have 1000 rows in a database and a delete operation is being run on them, then the trigger would also run 1000 times automatically.

Executed once for each row affected.

1. Statement Level Triggers

In the statement-level triggers, the operation is under execution only once no matter how many rows are involved in the operation.

Executed only once for the statement.

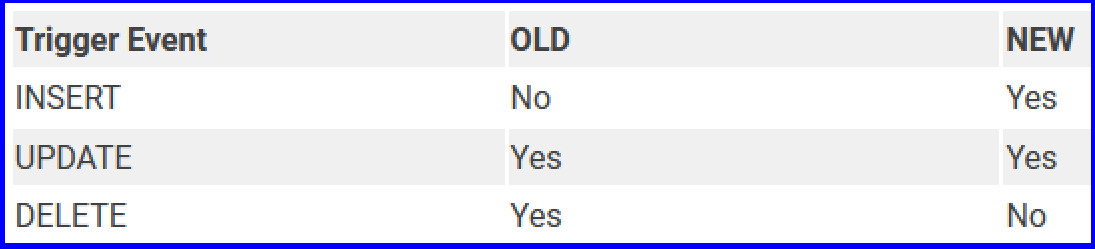
Parts of Triggers:

Trigger Event: Insert / Update / Delete Trigger action: Before / after

Trigger body: Logic / functionality (What Trigger is doing)

Triggers allow access to values from the table for comparison purposes using **NEW** and **OLD**. The availability of the modifiers depends on the trigger event you use:

New: get new value from column (Insert & after update). Old: get old value from the column (Delete & before update).



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;

Here,

* CREATE [OR REPLACE] TRIGGER trigger\_name: It creates or replaces an existing trigger with the trigger\_name.
* {BEFORE | AFTER | INSTEAD OF} : This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
* {INSERT [OR] | UPDATE [OR] | DELETE}: This specifies the DML operation.
* [OF col\_name]: This specifies the column name that would be updated.
* [ON table\_name]: This specifies the name of the table associated with the trigger.
* [REFERENCING OLD AS o NEW AS n]: This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
* [FOR EACH ROW]: This specifies a row level trigger, i.e., the trigger would 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.

Examples:

**BEFORE INSERT:**

* + CREATE TABLE student(rollno int , name char(30));
  + DELIMITER //

CREATE TRIGGER binsert BEFORE INSERT ON student for EACH ROW

BEGIN

SET new.name=ucase(new.name); END

//

* + INSERT INTO student values(101,"deepak");
  + SELECT \* FROM student;

**AFTER INSERT:**

* + DROP TABLE student;
  + CREATE TABLE student(id int,name char(30),gmail char(20),pwd char(20));
  + INSERT INTO student values(101,"Deepak","[gujjul@gmail.com](mailto:gujjul@gmail.com)","123456");
  + SELECT \* FROM student;



* + CREATE TABLE student\_copy(gmailcopy char(20),pwdcopy char(20));
  + DELIMITER //

CREATE TRIGGER ainsert AFTER INSERT ON student FOR EACH ROW

INSERT INTO student\_copy(gmailcopy,pwdcopy) VALUES(new.gmail,new.pwd);

//

* + INSERT INTO student VALUES(102,"Sagar","[sagar@gmail.com](mailto:sagar@gmail.com)","996655");
  + SELECT \* FROM student;



* + SELECT \* FROM student\_copy;



**BEFORE UPDATE:**

* + DROP TABLE emp;
  + CREATE TABLE emp(empno int, name char(30));
  + INSERT INTO emp VALUES(10, "Ravi")
  + Delimiter //
  + CREATE TRIGGER uexample BEFORE UPDATE ON emp FOR EACH ROW

BEGIN

IF NEW.empno <= 0 THEN SET NEW.empno= NULL; END IF;

END

//

* + INSERT INTO emp VALUES(40,"Raju");
  + SELECT \* FROM emp;



* + update emp set empno=0 where empno=40;
  + SELECT \* FROM emp;



**AFTER UPDATE:**

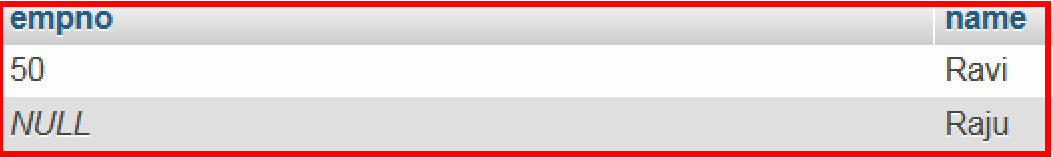
* + CREATE TABLE log(user char(30),status char(40));
  + CREATE TRIGGER auexample AFTER UPDATE ON emp FOR EACH ROW

INSERT INTO LOG VALUES(current\_user(),concat('Updated by ',old.name," ",now()));

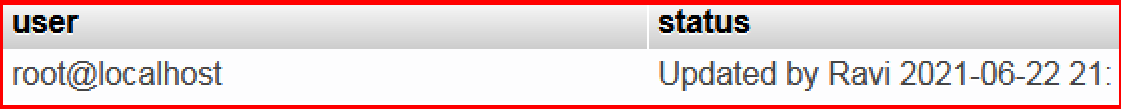
* + SELECT \* FROM emp;



* + update emp set empno=50 where empno=10;
  + SELECT \* FROM emp;



* + Select \* from log;



**BEFORE DELETE**

* + CREATE TABLE salaries(empno int,name varchar(20),salary int);
  + INSERT INTO salaries VALUES

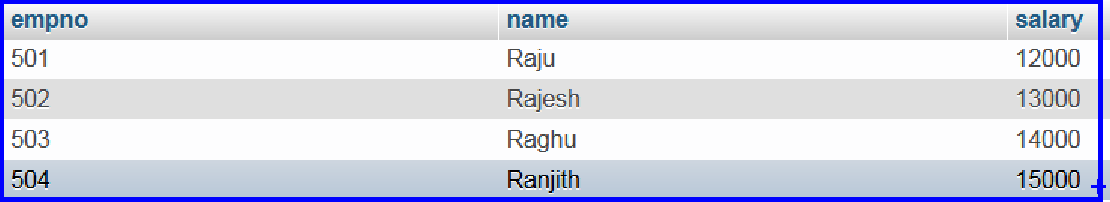
(501,"Raju",12000),

(502,"Rajesh",13000),

(503,"Raghu",14000),

(504,"Ranjith",15000);

* + SELECT \* FROM salaries;



* + CREATE TABLE salary\_deleted(empno int,name varchar(20),salary int,deleted\_time timestamp default now());
  + DELIMITER //
  + CREATE TRIGGER bdelete BEFORE DELETE ON salaries FOR EACH ROW

BEGIN

INSERT INTO salary\_deleted(empno,name,salary) VALUES

(old.empno,old.name,old.salary); END //

* + SELECT \* FROM salary\_deleted;//Empty
  + DELETE FROM salaries WHERE empno=501;
  + SELECT \* FROM salary\_deleted;



**AFTER DELETE:**

* + drop table salaries;
  + CREATE TABLE salaries(empno int,salary int);
  + INSERT INTO salaries values (501,3000),(502,2000),(503,1000);
  + CREATE table salary\_avg(sal int);
  + INSERT INTO salary\_avg (sal) SELECT SUM(salary) FROM salaries;
  + SELECT \* from salary\_avg;
  + DELIMITER //
  + CREATE TRIGGER adelete AFTER DELETE

ON salaries FOR EACH ROW BEGIN

UPDATE salary\_avg SET sal=sal-old.salary; END //

* + SELECT \* FROM salaries;
  + SELECT \* FROM salary\_avg;



* + DELETE FROM salaries where empno=501;
  + SELECT \* FROM salaries;
  + SELECT \* FROM salary\_avg;



**SCHEMA REFINEMENT AND NORMALISATION**

1. Schema Refinement:

The Schema Refinement is the process that re-defines the schema of a relation. The best technique of schema refinement is **decomposition.**

***Normalization or Schema Refinement is a technique of organizing the data in the database. It is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.***

***Redundancy*** refers to repetition of same data or duplicate copies of same data stored in different locations.

Three types of redundancy:

1. File level
2. Entire record level
3. Few attribute have redundancy.

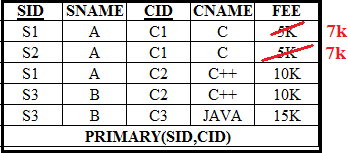
**Anomalies:** Anomalies refers to the problems occurred after poorly planned databases where all the data is stored in one table which is sometimes called a flat file database.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SID** | **SNAME** | **CID** | **CNAME** | **FEE** |
| S1 | A | C1 | C | 5K |
| S2 | A | C1 | C | 5K |
| S1 | A | C2 | C++ | 10K |
| S3 | B | C2 | C++ | 10K |
| S3 | B | C3 | JAVA | 15K |
| **PRIMARY(SID,CID)** | | | | |

Here all the data is stored in a single table which causes redundancy of data or say anomalies as SID and Sname are repeated once for same CID. Let us discuss anomalies one by one.

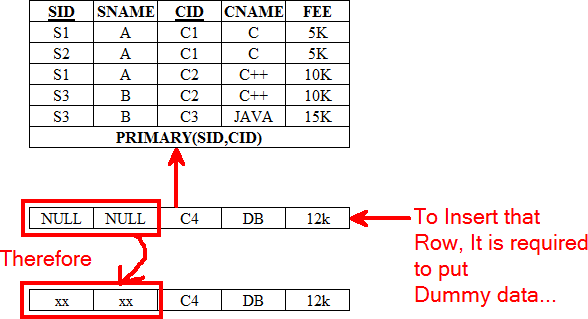
* 1. **Insertion anomalies:** It may not be possible to store some information unless some otherinformation is stored as well.
  2. **redundant storage:** some information is stored repeatedly
  3. **Update anomalies:** If one copy of redundant data is updated, then inconsistency is created unless all redundant copies of data are updated.
  4. **Deletion anomalies:** It may not be possible to delete some information without losing someother information as well.

**Problem in updation / updation anomaly** – If there is updation in the fee from 5000 to 7000, then we have to update FEE column in all the rows, else data will become inconsistent.



**Insertion Anomaly and Deletion Anomaly**- These anomalies exist only due to redundancy, otherwise they do not exist.

**Insertion Anomalies**: New course is introduced say c4 with course name DB, but no student is there who is having DB subject.



Because of insertion of some data, It is forced to insert some other dummy data.

Deletion Anomaly:

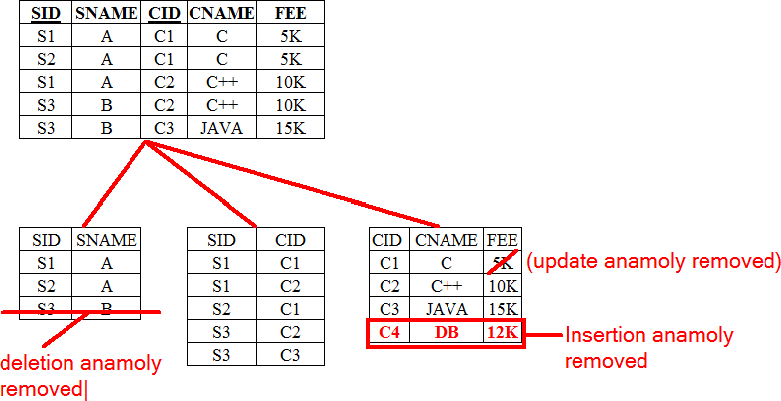
Deletion of S3 student causes the deletion of course.



Because of deletion of some data, It is forced to delete some other useful data. Solutions to Anomalies: Decomposition of Tables – Schema Refinement

To avoid redundancy and problems due to redundancy, we use refinement technique called DECOMPOSITION.

**Decomposition:** Process of decomposing a larger relation into smaller relations. Each of smaller relations contains subset of attributes of original relation.



Problems related to decomposition:

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

Properties of Decomposition:



* 1. **Lossless Join Decomposition:**
     + Consider there is a relation R which is decomposed into sub relations R1, R2 , …. , Rn.
     + This decomposition is called lossless join decomposition when the join of the sub relations results in the same relation R that was decomposed.
     + For lossless join decomposition, we always have

**R1** ⋈ **R2** ⋈ **R3 …….** ⋈ **Rn = R**

where ⋈ is a natural join operator

If a relation R is decomposed into two relations R1 & R2 then, it will be lossless iff

1. attribute (R1) U attribute (R2) = attribute(R)
2. attribute (R1) **∩** attribute (R2) ≠ ∅
3. attribute (R1) **∩** attribute (R2) → attribute (R1) or attribute (R1) **∩** attribute (R2) → attribute (R2)

Example:

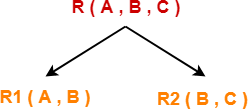
Consider the following relation R ( A , B , C ):

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |
| 1 | 2 | 1 |
| 2 | 5 | 3 |

|  |  |  |
| --- | --- | --- |
| 3 | 3 | 3 |

R(A, B, C)

Consider this relation is decomposed into two sub relations R1 (A, B) and R2 (B, C)



The two sub relations are:

|  |  |
| --- | --- |
| **A** | **B** |
| 1 | 2 |
| 2 | 5 |
| 3 | 3 |

|  |  |
| --- | --- |
| **B** | **C** |
| 2 | 1 |
| 5 | 3 |
| 3 | 3 |

R1(A,B) R2(B, C)

Now, let us check whether this decomposition is lossless or not. For lossless decomposition, we must have-

**R1** ⋈ **R2 = R**

Now, if we perform the natural join (⋈) of the sub relations R1 and R2 , we get

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |
| 1 | 2 | 1 |
| 2 | 5 | 3 |
| 3 | 3 | 3 |

R(A, B, C)

This relation is same as the original relation R.

Thus, we conclude that the above decomposition is lossless join decomposition.

Example - 2:

**R (A, B, C)**

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |

|  |  |  |
| --- | --- | --- |
| 1 | 1 | 1 |
| 2 | 1 | 2 |
| 3 | 2 | 1 |
| 4 | 3 | 2 |

The two sub relations are:

R1 (A, B)

**R2 (B, C)**

|  |  |
| --- | --- |
| **A** | **B** |
| 1 | 1 |
| 2 | 1 |
| 3 | 2 |
| 4 | 3 |

|  |  |
| --- | --- |
| **B** | **C** |
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |
| 3 | 2 |

Note: Above example is not lossless join decomposition. Dependency preserving decomposition:

Let a relation R(A,B,C,D) and set a FDs F = { A -> B , A -> C , C -> D} are given.

A relation R is decomposed into -

R1 = (A, B, C) with FDs F1 = {A -> B, A -> C}, and R2 = (C, D) with FDs F2 = {C -> D}.

F' = F1 𝖴 F2 = {A -> B, A -> C, C -> D} so, F' = F.

And so, F'+ = F+.

Functional dependencies

Functional dependency is a relationship that exists when one attribute uniquely determines another attribute.

A functional dependency XY in a relation holds true if two tuples having the same value ofattribute X also have the same value of attribute Y.

* 1. e if X and Y are the two sets of attributes in a relation R where X ⊆ R, Y ⊆ R

Then, for a functional dependency to exist from X to Y,

**if t1.X=t2.X then t1.Y=t2.Y** where t1,t2 are tuples and X,Y are attributes.



A functional dependency is denoted by an arrow "→".

XY can be called as X determines Y

Y is determined by X

Example:

|  |  |  |  |
| --- | --- | --- | --- |
| **eno** | **ename** | **sal** | **City** |
| 1 | Raju | 30000 | Knr |
| 2 | Ravi | 20000 | Hyd |
| 3 | Rakesh | 50000 | Hzb |

In this example, if we know the value of Employee number, we can obtain Employee Name, city, salary, etc. By this, we can say that the city, Employee Name, and salary are functionally depended on Employee number.

Example 2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **r.no** | **name** | **marks** | **dept** | **Course** |
| 1 | A | 78 | CS | C1 |
| ` | B | 60 | EE | C1 |
| 3 | A | 78 | CS | C2 |
| 4 | B | 60 | EE | C3 |
| 5 | C | 80 | IT | C3 |
| 6 | D | 80 | EC | C2 |

**Which of the following are functional dependencies**

r.no →name

name →r.no no functional dependency

r.no →marks functional dependency

dept →course course →dept marks →dept name →marks

[name, marks] →dept

[name, marks] →[dept, course]

Example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **D** | **E** |
| a | 2 | 3 | 4 | 5 |
| 2 | a | 3 | 4 | 5 |
| a | 2 | 3 | 6 | 5 |
| a | 2 | 3 | 6 | 6 |

**Which of the following are functional dependencies**

1. A → BC
2. DE → C
3. C → DE
4. BC → A

Types of Functional Dependencies

There are two types of functional dependencies

* + 1. Trivial Functional Dependencies
    2. Non – Trivial Functional Dependencies

**Trivial Functional Dependencies:**

A functional dependency X → Y is said to be trivial if and only if Y ⊆ X.

Thus, if RHS of a functional dependency is a subset of LHS, then it is called as a trivial functional dependency.

**Example:**

* + - 1. AB → A
      2. AB → B
      3. AB → AB

**Non - Trivial Functional Dependencies:**

A functional dependency X → Y is said to be non-trivial if and only if Y ⊄ X. Example:

AB → C

**Axioms / Inference rules:** It assumes that axioms are true without being able to prove them**.**

Developed by Armstrong in 1974, there are six rules (axioms) that all possible functional dependencies may be derived from them

1. Reflexivity Rule

In the reflexive rule, if Y is a subset of X, then X determines Y. If X ⊇ Y then X → Y

Example:

X= {A, B, C, D} Y= {B, C} ABC→BC

1. Augmentation Rule

The augmentation is also called as a partial dependency. In augmentation, if X determines Y, then XZ determines YZ for any Z

If X → Y then XZ → YZ

Example:

For R (ABCD), if A → B then AC → BC

1. Transitivity Rule

In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

If X → Y and Y → Z then X → Z

Additional rules:

1. **Union Rule**

Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.

If X→ Y and X→Z then X→YZ

1. Composition

If W → Z, X → Y, then WX → ZY.

1. Decomposition

If X→YZ then X→Y and X→Z

Closure of a Set of Attributes(X+) / Attribute closure:

X+ is the set of all attributes that can be determined using the given set X (attributes).

Algorithm: Determining X+, the closure of X under F.

**Input:** Let F be a set of FDs for relation R.

**Steps:**

1. **X+ = X** //initialize X+ to X

2. For each FD: Y -> Z in F Do

If **Y** ⊆ **X+** Then

**X+ = X+** 𝖴 **Z**

End If End For

//If Y is contained in X+

//add Z to X+

3. Return **X+** //Return closure of X

**Output:** Closure **X+** of **X** under **F**

**Example – 1:**

We are given the relation R(A, B, C, D, E). This means that the table R has five columns: A, B, C, D, and E. We are also given the set of functional dependencies:

{A->B, B->C, C->D, D->E}.

What is {A} +?

* + First, we add A to {A}+.
  + What columns can be determined given A? We have A -> B, so we can determine B. Therefore, {A}+ is now {A, B}.
  + What columns can be determined given A and B? We have B -> C in the functional dependencies, so we can determine C. Therefore, {A}+ is now

{A, B, C}.

* + Now, we have A, B, and C. What other columns can we determine? Well, we have C -> D, so we can add D to {A}+.
  + Now, we have A, B, C, and D. Can we add anything else to it? Yes, since D

-> E, we can add E to {A}+.

* + We have used all of the columns in R and we have all used all functional dependencies. {A}+ = {A, B, C, D, E}.

Example – 2:

Consider a relation R ( A , B , C , D , E , F , G ) with the functional dependencies-

A → BC BC → DE

D → F CF → G

Now, let us find the closure of some attributes and attribute sets Closure of attribute A

A+ = { A }

= { A , B , C } ( Using A → BC )

= { A , B , C , D , E } ( Using BC → DE )

= { A , B , C , D , E , F } ( Using D → F )

= { A , B , C , D , E , F , G } ( Using CF → G )

Thus,

A+ = { A , B , C , D , E , F , G }

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **R(ABCDEFG)**  A→B  BC→ DE AEG →G |  |  | **R(ABCDE)** A→ BC CD→ E  B→ D |  |
| (AC)+={AC}  ={ABC} | (Using Reflexivity)  (Using A→B) |  | E→ A B+={B} | (Using Reflexivity) |
| **={ABCDE}** | (Using BC→ DE) |  | **={BD}** | (Using B→ D) |

**R(ABCDEF)** AB→C BC→AD

D→E CF→B

(AB)+={AB} **(**Using Reflexivity)

={ABC} **(**Using AB→C**)**

={ABCD} **(**Using BC→AD**)**

=**{ABCDE} (**Using D→E**)**

**Super key:** attribute **/** set of attributes whose closure contains all attributes of given relation.

**How to find candidate key(s) in a Relation:**

**Procedure:**

* + Determine all essential attributes of the given relation.
  + Essential attributes are those attributes which are not present on RHS of any functional dependency.
  + Find closure of attribute / set of attributes for the essential attributes (Not present R.H.S attributes)

If closure contains all the attributes, then it will be the only key. Else find the combination of remaining attributes with the attributes not present in R.H.S

Example:

Let R(A, B, C, D, E, F) be a relation scheme with the following functional dependencies

C→F E→A EC→D

A→B

Step - 1

* + Determine all essential attributes of the given relation (attributes which are not present in R.H.S).
  + Essential attributes of the relation are ***C and E.***
  + So, attributes C and E will definitely be a part of every candidate key.

Step - 2

Now,

* + We will check if the essential attributes together can determine all remaining non-essential attributes.
  + To check, we find the closure of CE.

So, we have

{ CE }+ = { C , E }

= { C , E , F } ( Using C → F )

= { A , C , E , F } ( Using E → A )

= { A , C , D , E , F } ( Using EC → D )

= { A , B , C , D , E , F } ( Using A → B )

We conclude that CE can determine all the attributes of the given relation. So, CE is the only possible candidate key of the relation.

**More Examples:**

**R(ABCD) AB**→CD

**D**→A

Candidate keys are AB,BD

**R(ABCD) AB**→CD

**C**→A **D**→B

Candidate keys are AB, AD, BC,DC

**R(WXYZ)**

**Z**→W **Y**→XZ **WX**→Y

Candidate keys are Y, WX,ZX

**R(ABCD)**

**A**→B **B**→C **C**→A

Candidate keys are AD, BD, CD

**R(ABCDE) AB**→CD

**D**→A **BC**→DE

Candidate keys are AB, BC, BD

**R(ABCDE) AB**→C

**C**→D D→E **A**→B **C**→A

Candidate keys are A,C

**R(ABCDE)**

**A**→D **AB**→C

**B**→E **D**→C **E**→A

Candidate key is B

**R(ABCDEF) AB**→C **DC**→AE

**E**→F

Candidate key is ABD,BDC

**Normalization**

Normalization is a database design technique that reduces data redundancy and eliminates undesirable characteristics like Insertion, Update and Deletion Anomalies. Normalization rules divide larger tables into smaller tables and links them using relationships. The purpose of Normalization in SQL is to eliminate redundant (repetitive) data and ensure data is stored logically.

Normalization rules are divided into the following normal forms:

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. **First Normal Form:**

A given relation is called in First Normal Form (1NF) if each cell of the table contains only an atomic value (it states that an attribute of a table cannot hold multiple *values*). i.e it should not contain multivalued attribute and no composite attribute.

Example:

|  |  |  |
| --- | --- | --- |
| **sid** | **sname** | **cname** |
| 1 | Raju | C,C++ |
| 2 | Ravi | C,Java |
| 3 | Rakesh | Java |

**After Decomposition**

**Relation is not in 1NF**

|  |  |
| --- | --- |
| sid | sname |
| 1 | Raju |
| 2 | Ravi |
| 3 | Rakesh |

|  |  |
| --- | --- |
| sid | cname |
| 1 | C |
| 1 | C++ |
| 2 | C |
| 2 | JAVA |
| 2 | Java |

**Relation is in 1NF**

Note:

* By default, every relation is in 1NF.
* This is because formal definition of a relation states that value of all the attributes must be atomic.

1. **Second Normal form:**

A table is said to be in the second Normal Form if

* + It should be in the First Normal form.
  + And, it should not have Partial Dependency.

***Or***

A relation is in 2NF, if it is in 1NF and every non – prime attribute is fully functional dependent on candidate of the relation.

Partial Dependency

A partial dependency is a dependency where few attributes of the candidate key determines non-prime attribute(s).

***Or***

A partial dependency is a dependency where a portion of the candidate key or incomplete candidate key determines non-prime attribute(s).

In other words,

X → a is called a partial dependency if and only if-

1. ***X*** is a proper subset of some candidate key and
2. ***a*** is a non-prime attribute or non – key attribute

If any one condition fails, then it will not be a partial dependency.

Prime or Key Attributes:

Attributes of the relation which exist in at least one of the possible candidate keys, are called prime or key attributes.

Non Prime or Non Key Attributes:

Attributes of the relation which does not exist in any of the possible candidate keys of the relation, such attributes are called non prime or non key attributes.

**NOTE**

To avoid partial dependency, incomplete candidate key must not determine any non-prime attribute.

* ***However, incomplete candidate key can determine prime attributes.***

Example - 1:

Consider a relation- R (V, W, X, Y, Z) with functional dependencies-

VW → XY

Y → V WX → YZ

The possible candidate keys for this relation are

VW, WX, WY

From here,

* Prime attributes = { V , W , X , Y }
* Non-prime attributes = { Z }

Now, if we observe the given dependencies-

* There is no partial dependency.
* This is because there exists no dependency where incomplete candidate key determines any non-prime attribute.

Thus, we conclude that the given relation is in 2NF.

Example - 2:

Consider a relation- R (A, B, C, D, E, F) with functional dependencies-

A → B B → C C → D D → E

The possible candidate key for this relation is

AF

From here,

* Prime attributes = {A, F}
* Non-prime attributes = { B, C, D, E} Now, if we observe the given dependencies-
* There is partial dependency.
* This is because there is an incomplete candidate key (attribute A) determines non-prime attribute (B).

Thus, we conclude that the given relation is not in 2NF.

Example – 3:

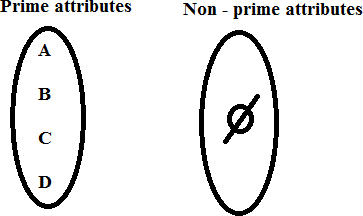
Consider a relation- R (A, B, C, D) with functional dependencies-

AB → CD

C → A D → B

The possible candidate key for this relation is

AB, AD, BC, CD



If there is no Non – prime attributes, then we can conclude that it is in 2NF

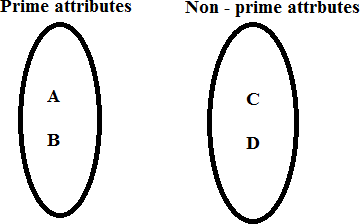
Example - 4:

Consider a relation- R (A, B, C, D) with functional dependencies-

AB → D

B → C

Possible candidate key is ***AB***



Now, if we observe the given dependencies-

* There is partial dependency.
* This is because there is an incomplete candidate key (attribute B) determines non-prime attribute (C).

Thus, we conclude that the given relation is not in 2NF.

1. **Third Normal Form(3 NF):**

A table is in 3NF If

* 1. It is in 2NF
  2. For every non trivial dependency

X → a, ***X*** is either super key or ***a*** is a prime attribute.

Or

A table is said to be in 3NF If

1. It should be in 2NF
2. It should not have any transitive dependency

Or

A table is said to be 3NF form when it don’t have partial dependency and transitive dependency.

**Transitive dependency:** When a non prime attribute finds another non prime attribute, it is called transitive dependency.

**Example-**

Consider a relation- R ( A , B , C , D , E ) with functional dependencies-

A → BC CD → E

B → D E → A

The possible candidate keys for this relation are-

A, E, CD, BC

From here,

* Prime attributes = { A , B , C , D , E }
* There are no non-prime attributes

Now,

* It is clear that there are no non-prime attributes in the relation.
* In other words, all the attributes of relation are prime attributes.
* Thus, all the attributes on RHS of each functional dependency are prime attributes.

Thus, we conclude that the given relation is in 3NF.

1. **Boyce-Codd Normal Form(BCNF)**

A given relation is called in BCNF if and only if-

* 1. Relation should be in 3NF.
  2. For each non-trivial functional dependency A → B, A is a super key of the

relation.

Example:

Consider a relation- R ( A , B , C ) with the functional dependencies-

A → B B → C C → A

The possible candidate keys for this relation are- A, B, C

Here A, B, C is super keys, so we can conclude that given relation is in BCNF

**Fourth Normal Form:**

A table is said to be in 4NF If

1. It should be in BCNF
2. It should not have any Multi – valued dependency.

***Multi – valued dependency:***

1. For a dependency A →→ B, if for a single value of A, multiple values of B exists.
2. Table should be at least 3 columns.
3. For a relation (A, B, C), If there is a Multi – valued dependency between A →→ B, then B and C should be Independent of each other.

Example:

|  |  |  |
| --- | --- | --- |
| **Sid** | **cname** | **hobby** |
| 1 | DBMS | Cricket |
| 1 | Java | Tennis |
| 2 | C | Dancing |
| 2 | C++ | Cricket |

|  |  |
| --- | --- |
| sid | cname |
| 1 | DBMS |
| 1 | Java |
| 2 | C |
| 2 | C++ |

|  |  |
| --- | --- |
| sid | hobby |
| 1 | Cricket |
| 1 | Tennis |
| 2 | Dancing |
| 2 | Cricket |

**Fifth normal form:**

A database is said to be in 5NF, if and only if,

* It should be in 4NF
* If we can decompose table further to eliminate redundancy and anomaly, and when we re-join the decomposed tables by means of candidate keys, we should not be losing the original data or any new record set should not arise. In simple words, joining two or more decomposed table should not lose records nor create new records. *Or*
* *It should not have join dependency*

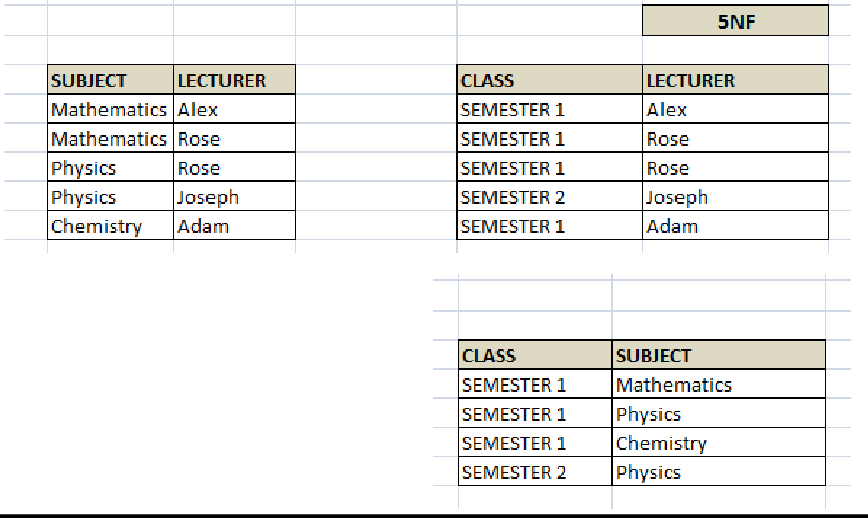
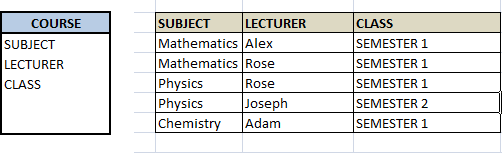
5NF is also known as Project-join normal form (PJ/NF).

Join dependency:

Let ‘R’ be a relation schema and R1, R2 ….Rn be the decompositions of R, R is said to satisfy the join dependency \*(R1, R2… Rn) if and only if

∏R1(R) ⋈∏R2(R) ⋈………∏RN(R) =R

Example:



Now, each of combinations is in three different tables. If we need to identify who is teaching which subject to which semester, we need join the keys of each table and get the result.

For example, who teaches Physics to Semester 1, we would be selecting Physics and Semester1 from table 3 above, join with table1 using Subject to filter out the lecturer names. Then join with table2 using Lecturer to get correct lecturer name. That is we joined key columns of each table to get the correct data. Hence there is no lose or new data – satisfying 5NF condition.

UNIT - IV

**Transaction Management**

Transaction Concept

**Transaction:**

Collections of operations that form a single logical unit of work are called transactions. Or

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

A transaction is a program unit whose execution may change the contents of a database. Or

Transaction can be taken as an event that occurs in the database. Usually a transaction reads a value from the database or writes a value to the database. A transaction is a group of tasks.

Operations in Transaction-

The main operations in a transaction are-

1. Read Operation
2. Write Operation
3. Read Operation-
   * Read operation reads the data from the database and then stores it in the buffer in main memory.
   * For example- **Read(A)** instruction will read the value of A from the database and will store it in the buffer in main memory.
4. Write Operation-
   * Write operation writes the updated data value back to the database from the buffer.
   * For example- **Write(A)** will write the updated value of A from the buffer to the database.

For instance, transfer of money from one account to another is a transaction

Simple Transaction Example

1. Read your account balance
2. Deduct the amount from your balance
3. Write the remaining balance to your account
4. Read your friend’s account balance
5. Add the amount to his account balance
6. Write the new updated balance to his account

This whole set of operations can be called a transaction.

***In DBMS, we write the above 6 steps transaction like this:***

Let’s say your account is A and your friend’s account is B, you are transferring 1000 from A

to B, the steps of the transaction are:

1. R(A);

2. A = A - 1000;

3. W(A);

4. R(B);

5. B = B + 1000;

6. W(B);

In the above transaction R refers to the Read operation and W refers to the write operation.

Transaction Management

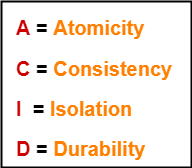
**Transaction Management** ensures that the database remains in a consistent (correct) state despite system failures (e.g power failures, and operating system crashes) and transaction failures.

ACID Properties-

* It is important to ensure that the database remains consistent before and after the transaction.
* To ensure the consistency of database, certain properties are followed by all

the transactions occurring in the system.

* These properties are called as **ACID Properties** of a transaction.



Atomicity:

* + This property ensures that either the transaction occurs completely or it does not occur at all.
  + In other words, it ensures that no transaction occurs partially.
  + That is why, it is also referred to as “**All or nothing rule**“.
  + It is the responsibility of Transaction Control Manager to ensure atomicity of the transactions.

Example:

|  |  |
| --- | --- |
| **Before: X: 500** | **Y: 200** |
| **Transaction T** | |
| **T1** | **T2** |
| **Read(X)** | **Read(Y)** |
| **X: = X – 100** | **Y: = Y+100** |
| **Write(X)** | **Write(Y)** |
| **After: X: 400** | **Y: 300** |

If the transaction fails after completion of T1 but before completion of T2.( say, after write(X) but before write(Y)), then amount has been deducted from X but not added to Y. This results in an inconsistent database state. Therefore, the transaction must be executed in totally in order to ensure correctness of database state.

Consistency

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above, the total amount before and after the transaction must be maintained.

Total before T occurs = 500 + 200 = 700. Total after T occurs = 400 + 300 = 700.

Therefore, database is consistent.

Inconsistency occurs in case T1 completes but T2 fails. As a result T is incomplete. It is the responsibility of DBMS and application programmer to ensure consistency of the database.

Isolation:

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

Let X= 500, Y = 500.

Consider two transactions T1 and T2.

|  |  |
| --- | --- |
| **T1** | **T2** |
| **Read(X) Read(X)**  **X: =X\*100 Read(Y)**  **Write(X) Z: = X + Y**  **Read(Y) Write(Z) Y: = Y – 50**  **Write(Y)** | |

Suppose T1 has been executed till Read (Y) and then T2 starts. As a result, interleaving of operations takes place due to which T2 reads correct value of X but incorrect value of Y and sum computed by

T2: (X+Y = 50, 000+500=50, 500)

is thus not consistent with the sum at end of transaction:

T1: (X+Y = 50, 000 + 450 = 50, 450).

This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory.

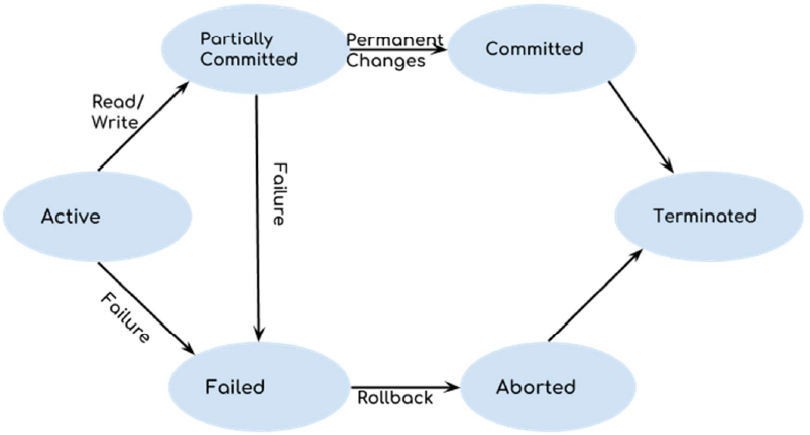
The effects of the transaction, thus, are never lost.

The ACID properties, in totality, provide a mechanism to ensure consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.



Transaction States

A transaction in DBMS can be in one of the following states.



Active State

As we have discussed in the DBMS transaction introduction that a transaction is a sequence of operations. If a transaction is in execution then it is said to be in active

state. It doesn’t matter which step is in execution, until unless the transaction is executing, it remains in active state.

Failed State

If a transaction is executing and a failure occurs, either a hardware failure or a software failure then the transaction goes into failed state from the active state.

Partially Committed State

As we can see in the above diagram that a transaction goes into “partially committed” state from the active state when there are read and write operations present in the transaction.

Once the whole transaction is successfully executed, the transaction goes into partially committed state where we have all the read and write operations performed on the main memory (local memory) instead of the actual database. The reason why we have this state is because a transaction can fail during execution so if we are making the changes in the actual database instead of local memory, database may be left in an inconsistent state in case of any failure. This state helps us to rollback the changes made to the database in case of a failure during execution.

Committed State

If a transaction completes the execution successfully then all the changes made in the local memory during partially committed state are permanently stored in the database. You can also see in the above diagram that a transaction goes from partially committed state to committed state when everything is successful.

Aborted State

As we have seen above, if a transaction fails during execution then the transaction goes into a failed state. The changes made into the local memory (or buffer) are rolled back to the previous consistent state and the transaction goes into aborted state from the failed state. Refer the diagram to see the interaction between failed and aborted state.

Terminated state

This is the last state in the life cycle of a transaction. After entering the committed state or aborted state, the transaction finally enters into a **terminated state** where its life cycle finally comes to an end.

NOTE

* + After a transaction has entered the committed state, it is not possible to roll back the transaction.
  + In other words, it is not possible to undo the changes that have been made by the transaction.
  + This is because the system is updated into a new consistent state.
  + The only way to undo the changes is by carrying out another transaction called as **compensating transaction** that performs the reverse operations.

Implementation of Atomicity and Durability

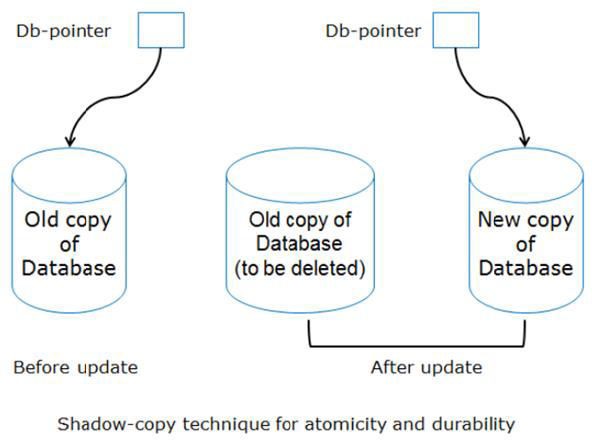
The recovery-management component of a database system can support atomicity and durability by a variety of schemes.

* In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched. If at any point the transaction has to be aborted, the system merely deletes the new copy. The old copy of the database has not been affected.
* This scheme is based on making copies of the database, called shadow copies, assumes that only one transaction is active at a time. The scheme also assumes that the database is simply a file on disk. A pointer called db- pointer is maintained on disk; it points to the current copy of the database.

If the transaction completes, it is committed as follows:

* First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk. (Unix systems use the flush command for this purpose.)
* After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted.

Figure below depicts the scheme, showing the database state before and after the update.



* If the transaction fails at any time before db-pointer is updated, the old contents of the database are not affected.
* We can abort the transaction by just deleting the new copy of the database.
* Once the transaction has been committed, all the updates that it performed are in the database pointed to by db pointer.
* Thus, either all updates of the transaction are reflected, or none of the effects are reflected, regardless of transaction failure.

**Note:** Unfortunately, this implementation is extremely inefficient in the context of large databases, since executing a single transaction requires copying the entire database. Furthermore, the implementation does not allow transactions to execute concurrently with one another.

Concurrent Execution

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data.

Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run serially—that is, one at a time, each starting only after the previous one has completed.

However, there are two good reasons for allowing concurrency:

Improved throughput and resource utilization:

* A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU.
* The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel.
* While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction.
* All of this increases the throughput of the system—that is, the number of transactions executed in a given amount of time.
* Correspondingly, the processor and disk utilization also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

Reduced waiting time:

* There may be a mix of transactions running on a system, some short and some long.
* If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction.
* If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them.
* Concurrent execution reduces the unpredictable delays in running transactions.
* Moreover, it also reduces the average response time: the average time for a transaction to be completed after it has been submitted.

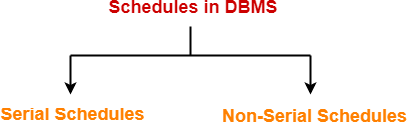
The idea behind using concurrent execution in a database is essentially the same as the idea behind using multi programming in an operating system.

The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It is achieved using **concurrency-control schemes**.

Schedule:

The order in which the operations of multiple transactions appear for execution is called as a schedule.

It represents the order in which instructions of a transaction are executed. Schedules may be classified as

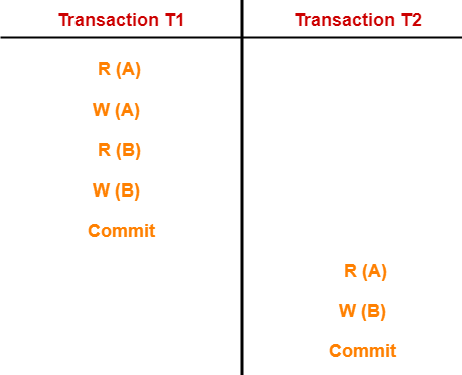


Serial Schedules-

In serial schedules,

* + All the transactions execute serially one after the other.
  + When one transaction executes, no other transaction is allowed to execute.
  + Serial schedules are always Consistent

Example:



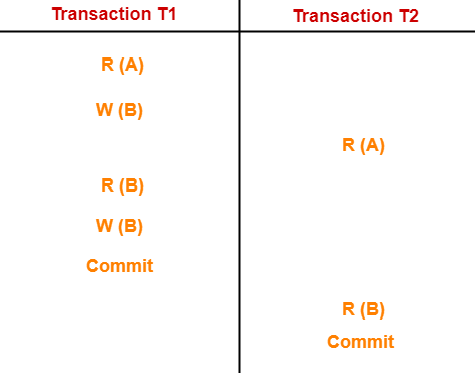
In this schedule,

* + There are two transactions T1 and T2 executing serially one after the other.
  + Transaction T1 executes first.
  + After T1 completes its execution, transaction T2 executes.
  + So, this schedule is an example of a **Serial Schedule**.

Non – serial / parallel schedule:

A non serial schedule is, before completing one transaction if it switches to another transaction.

Non-serial schedules are **NOT** always consistent.

Example:

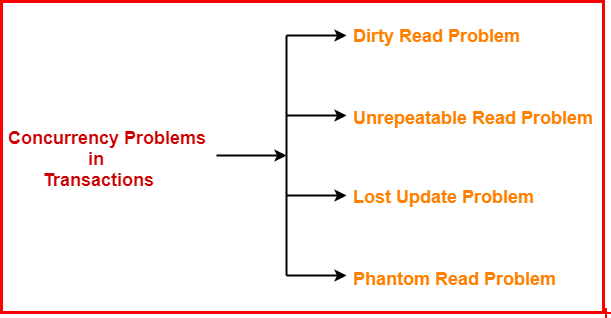
In this schedule,

* + There are two transactions T1 and T2 executing concurrently.
  + The operations of T1 and T2 are interleaved.
  + So, this schedule is an example of a **Non-Serial Schedule**.

Concurrency Problems in DBMS

* + When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems.
  + Such problems are called as **concurrency problems**.

The concurrency problems are:



1. **Dirty Read Problem (W – R Conflict)**

Reading the data written by an uncommitted transaction is called as dirty read. This read is called as dirty read because

* + There is always a chance that the uncommitted transaction might roll back

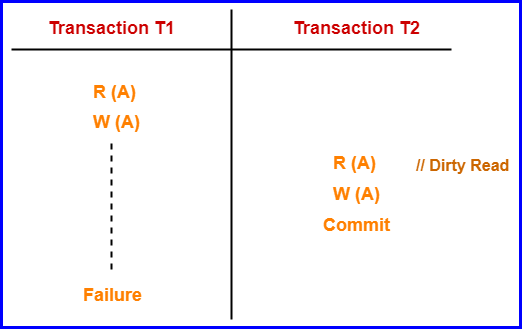
later.

* + Thus, uncommitted transaction might make other transactions read a value that does not even exist.
  + This leads to inconsistency of the database.

**Note:**

Dirty read does not lead to inconsistency always.

* + It becomes problematic only when the uncommitted transaction fails and roll backs later due to some reason.

Example:

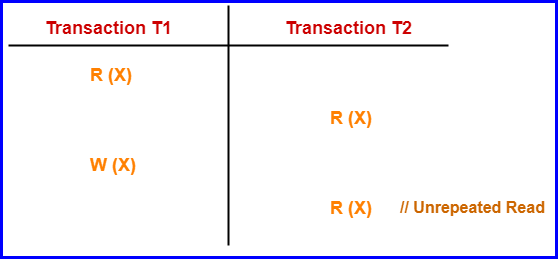
Here,

1. T1 reads the value of A.
2. T1 updates the value of A in the buffer.
3. T2 reads the value of A from the buffer.
4. T2 writes the updated the value of A.
5. T2 commits.
6. T1 fails in later stages and rolls back. In this example,

* T2 reads the dirty value of A written by the uncommitted transaction T1.
* T1 fails in later stages and roll backs.
* Thus, the value that T2 read now stands to be incorrect.
* Therefore, database becomes inconsistent.

1. Unrepeatable Read Problem:

This problem occurs when a transaction gets to read unrepeated i.e. different values of the same variable in its different read operations even when it has not updated its value.

Example:

Here,

* 1. T1 reads the value of X (= 10 say).
  2. T2 reads the value of X (= 10).
  3. T1 updates the value of X (from 10 to 15 say) in the buffer.
  4. T2 again reads the value of X (but = 15).

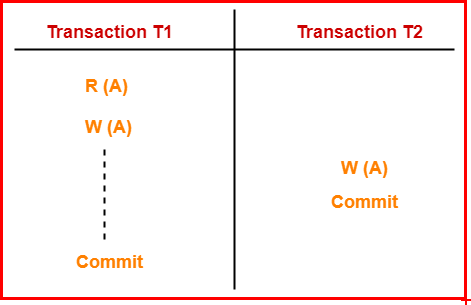
In this example,

* T2 gets to read a different value of X in its second reading.
* T2 wonders how the value of X got changed because according to it, it is running in isolation.

1. Lost Update Problem (W – W Conflict)

This problem occurs when multiple transactions execute concurrently and updates from one or more transactions get lost.

Example:



Here,

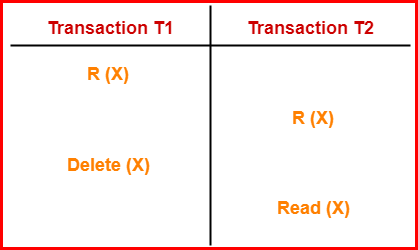
* 1. T1 reads the value of A (= 10 say).
  2. T2 updates the value to A (= 15 say) in the buffer.
  3. T2 does blind write A = 25 (write without read) in the buffer.
  4. T2 commits.
  5. When T1 commits, it writes A = 25 in the database. In this example,
* T1 writes the over written value of X in the database.
* Thus, update from T1 gets lost.

NOTE

* This problem occurs whenever there is a write-write conflict.
* In write-write conflict, there are two writes one by each transaction on the same data item without any read in the middle.

4. Phantom Read Problem

This problem occurs when a transaction reads some variable from the buffer and when it reads the same variable later, it finds that the variable does not exist.

Example:

Here,

1. T1 reads X.
2. T2 reads X.
3. T1 deletes X.
4. T2 tries reading X but does not find it.

In this example,

* T2 finds that there does not exist any variable X when it tries reading X again.
* T2 wonders who deleted the variable X because according to it, it is running in isolation.

Serializability

Some non-serial schedules may lead to inconsistency of the database. Serializability is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database.

Serializable Schedules

If a given non-serial schedule of ‘n’ transactions is equivalent to some serial schedule of ‘n’ transactions, then it is called as a **Serializable schedule**.

Types of Serializability

Serializability is mainly of two types-

1. Conflict Serializability
2. View Serializability

Conflict Serializability

If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a **conflict Serializable schedule**.

It is a type of Serializability that can be used to check whether the non-serial schedule is conflict Serializable or not.

Conflicting Operations-

Two operations are called as **conflicting operations** if all the following conditions hold true for them-

* Both the operations belong to different transactions
* Both the operations are on the same data item
* At least one of the two operations is a write operation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Case – 1**  **(No conflict)** | | **Case – 2**  **(No conflict)** | | **Case – 3**  **(Conflict operations)** | |
| **Ti** | **Tj** | **Ti** | **Tj** | **Ti** | **Tj** |
| R(A) |  | R(A) |  | R(A) |  |
|  | R(B) |  | R(A) |  | W(A) |
| **After Swapping** | | **After Swapping** | | **After Swapping** | |
|  | R(B) |  | R(A) |  | W(A) |
| R(A) |  | R(A) |  | R(A) |  |
|  |  |  |  | Ti is effecting | |
| **Case – 4**  **(Conflict operations)** | | **Case – 5**  **(Conflict operations)** | |  | |
| **Ti** | **Tj** | **Ti** | **Tj** |
| W(A) |  | W(A) |  |
|  | R(A) |  | W(A) |
| **After Swapping** | | **After Swapping** | |
|  | R(A) |  | W(A) |
| W(A) |  | W(A) |  |
| Tj is effecting | | DB is effecting | |

Checking Whether a Schedule is Conflict Serializable Or Not: Step-01:

Find and list all the conflicting operations.

Step-02:

Start creating a precedence graph by drawing one node for each transaction.

Step-03:

* Draw an edge for each conflict pair such that if Xi (V) and Yj (V) forms a conflict pair then draw an edge from Ti to Tj.
* This ensures that Ti gets executed before Tj.

Step-04:

* Check if there is any cycle formed in the graph.
* If there is no cycle found, then the schedule is conflict Serializable otherwise not.

NOTE

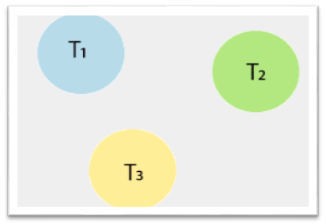
* By performing the [**Topological Sort**](https://www.gatevidyalay.com/topological-sort-topological-sorting/)of the [**Directed Acyclic Graph**](https://www.gatevidyalay.com/directed-acyclic-graphs/)so obtained, the corresponding serial schedule(s) can be found.
* Such schedules can be more than 1.

Example -1:

|  |  |  |  |
| --- | --- | --- | --- |
| S1 | | | |
|  | T1 | T2 | T3 |
| t1 | R(X) |  |  |
| t2 |  |  | R(Y) |
| t3 |  |  | R(X) |
| t4 |  | R(Y) |  |
| t5 |  | R(Z) |  |
| t6 |  |  | W(Y) |
| t7 |  | W(Z) |  |
| t8 | R(Z) |  |  |
| t9 | W(X) |  |  |
| t10 | W(Z) |  |  |

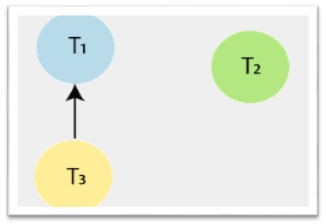
**Precedence graph for schedule S1:**

In the above schedule, there are three transactions: T1, T2, and T3. So, the precedence graph contains three vertices.



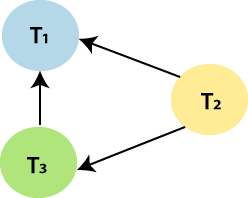
To draw the edges between these nodes or vertices, follow the below steps:

**Step1:** At time t1, there is no conflicting operation for **read(X)** of Transaction T1. **Step2:** At time t2, there is no conflicting operation for **read(Y)** of Transaction T3. **Step3:** At time t3, there exists a conflicting operation **Write(X)** in transaction T1 for **read(X)** of Transaction T3. So, draw an edge from T3 to T1.



**Step4:** At time t4, there exists a conflicting operation **Write(Y)** in transaction T3 for **read(Y)** of Transaction T2. So, draw an edge from T2 to T3.

**Step5:** At time t5, there exists a conflicting operation **Write (Z)** in transaction T1 for **read (Z)** of Transaction T2. So, draw an edge from T2 to T1.



**Step6:** At time t6, there is no conflicting operation for **Write(Y)** of Transaction T3.

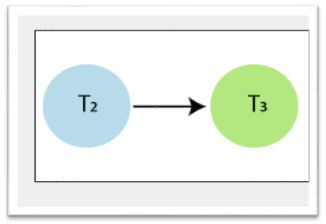
**Step7:** At time t7, there exists a conflicting operation **Write (Z)** in transaction T1 for **Write (Z)** of Transaction T2. So, draw an edge from T2 to T1, but it is already drawn.

After all the steps, the precedence graph will be ready, and it does not contain any cycle or loop, so the above schedule S1 is conflict Serializable. And it is equivalent to a serial schedule. Above schedule S1 is transformed into the serial schedule by using the following steps:

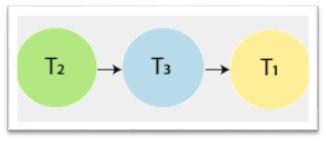
**Step1:** Check the vertex in the precedence graph where **indegree=0.** So, take the vertex T2 from the graph and remove it from the graph.



**Step 2:** Again check the vertex in the left precedence graph where **indegree=0.** So, take the vertex T3 from the graph and remove it from the graph. And draw the edge from T2 to T3.



**Step3:** And at last, take the vertex T1 and connect with T3.



Example – 2:

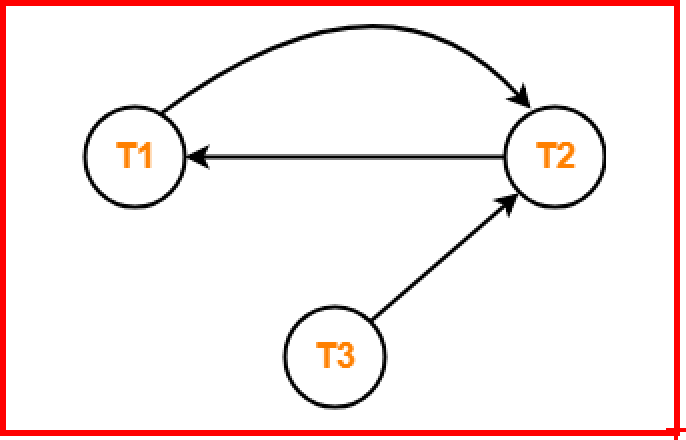
Check whether the given schedule S is conflict Serializable or not

S : R1(A) , R2(A) , R1(B) , R2(B) , R3(B) , W1(A) , W2(B)

List all the conflicting operations and determine the dependency between the transactions

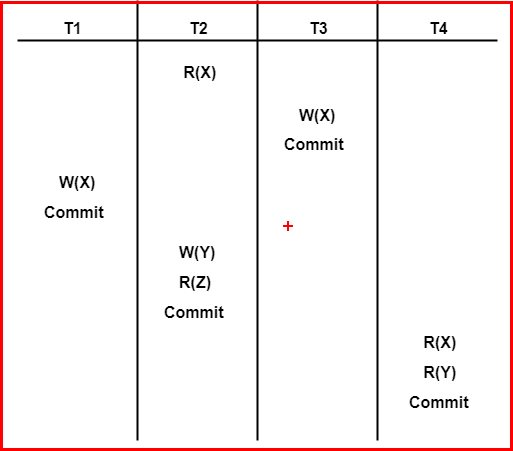
* R2(A) , W1(A) (T2 → T1)
* R1(B) , W2(B) (T1 → T2)
* R3(B) , W2(B) (T3 → T2)

Draw the precedence graph



* Clearly, there exists a cycle in the precedence graph.
* Therefore, the given schedule S is not conflict Serializable.

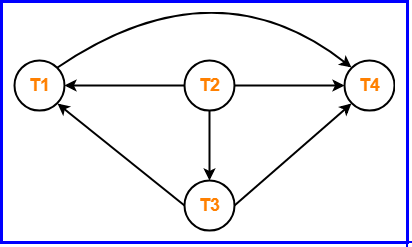
Example – 3:



List all the conflicting operations and determine the dependency between the transactions

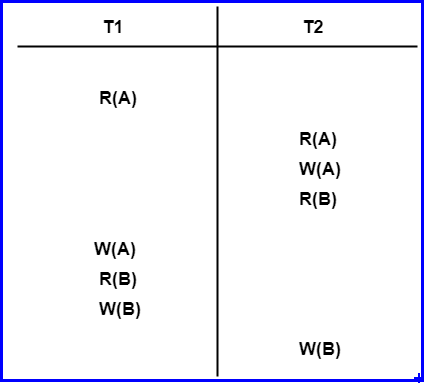
* R2(X) , W3(X) (T2 → T3)
* R2(X) , W1(X) (T2 → T1)
* W3(X) , W1(X) (T3 → T1)
* W3(X) , R4(X) (T3 → T4)
* W1(X) , R4(X) (T1 → T4)
* W2(Y) , R4(Y) (T2 → T4)

Draw the precedence graph



* + Clearly, there exists no cycle in the precedence graph.
  + Therefore, the given schedule S is conflict Serializable.

Practice Example:



**View Serializability**

If a given schedule is found to be view equivalent to some serial schedule, then it is called as a view Serializable schedule.

* If the given schedule is conflict serializable, then it is surely view serializable.
* If the given schedule is not conflict serializable, then it may or may not be view serializable.
* Check if there exists any blind write operation. (Writing without reading is called as a blind write).
* If there does not exist any blind write, then the schedule is surely not view serializable.
* If there exists any blind write, then the schedule may or may not be view serializable.

View Equivalent Schedules

Consider two schedules S1 and S2 each consisting of two transactions T1 and T2. Schedules S1 and S2 are called view equivalent if the following three conditions hold true for them-

Condition-01:

**Initial Read**

* The initial read of both the schedules must be in the same transaction.
* Suppose two schedule S1 and S2. In schedule S1, if a transaction **T1 is**

reading the data item A, then in S2, transaction T1 should also read A.

***Note: “Initial readers must be same for all the data items”.***

Condition-02:

**Updated Read**

In schedule S1, if the transaction Ti is reading the data item A which is updated by transaction Tj, then in schedule S2 also, Ti should read data item A which is updated by Tj.

***Note: “Intermediate Write-read sequence must be same.”***

Condition-03:

**Final Write**

* A final write must be the same in both the schedules.
* Suppose in schedule S1, if a transaction T1 updates A in the last, then in S2 final write operation should also be done by transaction T1.

***Note: “Final writers must be same for all the data items”.***

***Example:***

|  |  |  |
| --- | --- | --- |
| **Time** | **Transaction T1** | **Transaction T2** |
| t1 | Read(X) |  |
| t2 | Write(X) |  |
| t3 |  | Read(X) |
| t4 |  | Write(X) |
| t5 | Read(Y) |  |
| t6 | Write(Y) |  |
| t7 |  | Read(Y) |
| t8 |  | Write(Y) |

***Possible serial schedule is***

|  |  |  |
| --- | --- | --- |
| **Time** | **Transaction T1** | **Transaction T2** |
| t1 | Read(X) |  |
| t2 | Write(X) |  |
| t3 | Read(Y) |  |
| t4 | Write(Y) |  |
| t5 |  | Read(X) |
| t6 |  | Write(X) |
| t7 |  | Read(Y) |
| t8 |  | Write(Y) |

**Note:** S2 is the serial schedule of S1. If we can prove that both the schedule are view equivalent, then we can say that S1 schedule is a view serializable schedule.

**Now,** check the three conditions of view serializability for this example:

1. Initial Read

In S1 schedule, T1 transaction first reads the data item X. In Schedule S2 also transaction T1 first reads the data item X.

Now, check for Y. In schedule S1, T1 transaction first reads the data item Y. In schedule S2 also the first read operation on data item Y is performed by T1.

We checked for both data items X and Y, and the **initial read** condition is satisfied in schedule S1 & S2.

1. Updated Read

In Schedule S1, transaction T2 reads the value of X, which is written by transaction T1. In Schedule S2, the same transaction T2 reads the data item X after T1 updates it.

Now check for Y. In Schedule S1, transaction T2 reads the value of Y, which is written by T1. In S2, the same transaction T2 reads the value of data item Y after T1 writes it.

The **update read condition** is also satisfied for both the schedules S1 and S2.

1. Final Write

In schedule S1, the **final write operation** on data item X is done by transaction T2. In schedule S2 also transaction T2 performs the final write operation on X.

Now, check for data item Y. In schedule S1, the final write operation on Y is done by T2 transaction. In schedule S2, a final write operation on Y is done by T2.

We checked for both data items X and Y, and the **final write** condition is also satisfied for both the schedule S1 & S2.

**Conclusion:** Hence, all the three conditions are satisfied in this example, which means Schedule S1 and S2 are view equivalent. Also, it is proved that schedule S2 is the serial schedule of S1. Thus we can say that the S1 schedule is a view serializable schedule.

[Recoverability](https://www.gatevidyalay.com/recoverable-schedules-irrecoverable-schedules-non-serializable-schedules/)

So far, we have studied what schedules are acceptable from the viewpoint of consis

-tency of the database, assuming implicitly that there are no transaction failures. We now address the effect of transaction failures during concurrent execution.

If a transaction T₁, fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction Tj, that is dependent on Ti, (that is, Tj, has read data written by Ti.) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

Now, we address the issue of what schedules are acceptable from the viewpoint of recovery from transaction failure.

Irrecoverable Schedules:

If in a schedule,

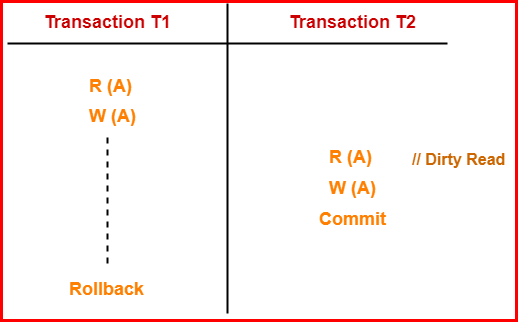
* A transaction performs a dirty read operation from an uncommitted transaction.

***\*\*Reading from an uncommitted transaction is called as a dirty read***

* And commits before the transaction from which it has read the value.
* Such a schedule is known as an **Irrecoverable Schedule**.

Example

Consider the following schedule



Here,

* T2 performs a dirty read operation.
* T2 commits before T1.
* T1 fails later and roll backs.
* The value that T2 read now stands to be incorrect.
* T2 cannot recover since it has already committed.

Recoverable Schedules

If in a schedule,

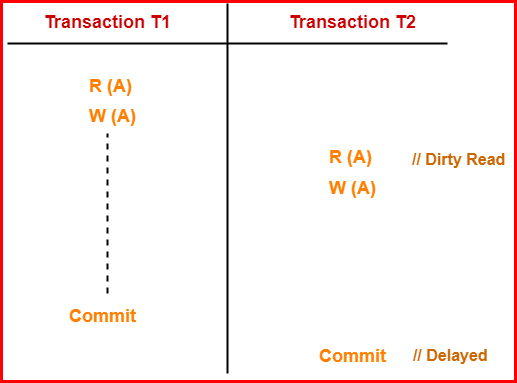
* A transaction performs a dirty read operation from an uncommitted transaction
* And its commit operation is delayed till the uncommitted transaction either commits or roll backs
* Such a schedule is known as a Recoverable Schedule**.**

Here,

* The commit operation of the transaction that performs the dirty read is delayed.
* This ensures that it still has a chance to recover if the uncommitted transaction fails later.

Example

Consider the following schedule



Here,

* T2 performs a dirty read operation.
* The commit operation of T2 is delayed till T1 commits or roll backs.
* T1 commits later.
* T2 is now allowed to commit.
* In case, T1 would have failed, T2 has a chance to recover by rolling back.

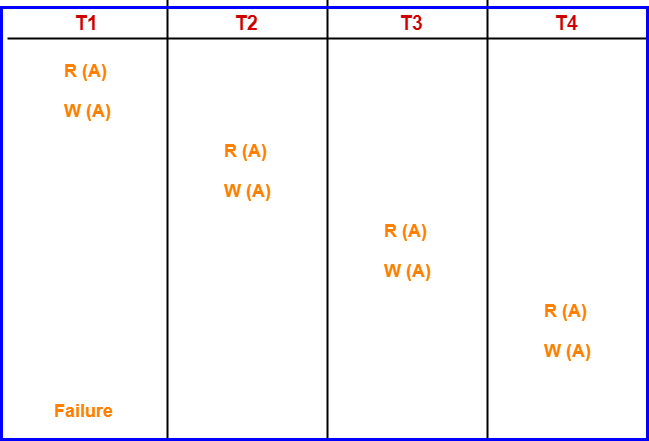
Important points:

* + No dirty read means a recoverable schedule.
  + All conflict serializable schedules are recoverable.
  + All recoverable schedules may or may not be conflict serializable.

Types of Recoverable Schedules

A recoverable schedule may be any one of these kinds

1. Cascading Schedule / cascading rollback
2. Cascade less Schedule
3. **Cascading Schedule / Cascading Rollback**
   * If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is called as a Cascading Schedule or Cascading Rollback.
   * It simply leads to the wastage of CPU time.

Example:

Here,

* Transaction T2 depends on transaction T1.
* Transaction T3 depends on transaction T2.
* Transaction T4 depends on transaction T3.

In this schedule,

* The failure of transaction T1 causes the transaction T2 to rollback.
* The rollback of transaction T2 causes the transaction T3 to rollback.
* The rollback of transaction T3 causes the transaction T4 to rollback.

***Such a rollback is called as a Cascading Rollback.***

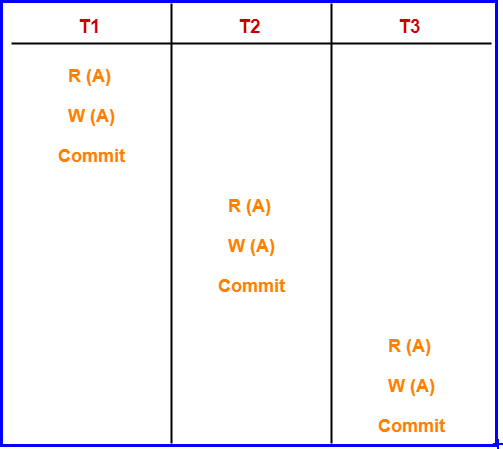
1. **Cascadeless Schedule:**

If in a schedule, a transaction is not allowed to read a data item until the last transaction that has written it is committed or aborted, such a schedule is called as a Cascadeless Schedule**.**

In other words,

* Cascadeless schedule allows only committed read operations.
* Therefore, it avoids cascading roll back and thus saves CPU time.

Example:



Note:

* Cascadeless schedule allows only committed read operations.
* However, it allows uncommitted write operations.

1. **Strict Schedule:**

If in a schedule, a transaction is neither allowed to read nor write a data item until the last transaction that has written it is committed or aborted, such a schedule is called as a Strict Schedule**.**

**CONCURRENCY CONTROL**:

In the multi-user system, we all know that multiple transactions run in parallel, thus trying to access the same data and suppose if one transaction already has the access to the data item and now another transaction tries to modify the data then it leads to error in database. There are issues with the concurrent execution of transactions such as conflicting operation where in simultaneously both transactions might try to access the data item which leads to the problem in database.

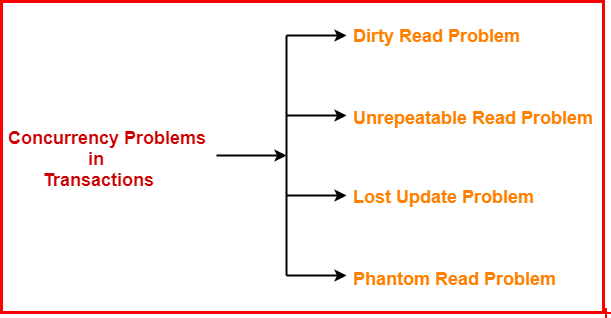
**Concurrency control** can be simply defined as the process of managing the simultaneous execution of transactions in a shared database thus ensuring the serialization of transactions.

Why use Concurrency method?

Reasons for using Concurrency control method is DBMS:

* To apply Isolation through mutual exclusion between conflicting transactions
* To resolve read-write and write-write conflict issues
* To preserve database consistency through constantly preserving execution obstructions
* The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.
* Concurrency control helps to ensure Serializability

The concurrency problems are:



**Concurrency Control Protocols**

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose. Following are the Concurrency Control techniques in DBMS:

* Lock-Based Protocols
* Two Phase Locking Protocol
* Timestamp-Based Protocols
* Validation-Based Protocols

Lock-Based Protocols

Lock Based Protocols in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.

To achieve consistency, isolation is the most important idea. Locking is the simplest idea to achieve isolation i.e first obtains a lock on a data item then performs a desired operation and then unlock it.

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks in DBMS help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

To provide better concurrency along with isolation we use different modes of locks.

**Shared lock:** Transaction can perform read operation; any other transaction can also obtain same lock on same data item at same time. Denoted by lock-S (Q).

**Exclusive mode:** A transaction may acquire exclusive lock on a data item in order to both read/write into it. The lock is excusive in the sense that no other transaction

can acquire any kind of lock (either shared or exclusive) on that same data item. Denoted by lock – X(Q).

The relationship between Shared and Exclusive Lock can be represented by the following table which is known as **Lock Compatibility matrix comp**.

Locks to be granted

Locks already existing

|  |  |  |
| --- | --- | --- |
|  | Shared | Exclusive |
| Shared | TRUE | FALSE |
| Exclusive | FALSE | FALSE |

How Should Lock be used?

In a transaction, a data item which we want to read/write should first be locked before the read/write is done. After the operation is over, the transaction should then unlock the data item so that other transaction can lock that same data item for their respective usage. In the earlier chapter we had seen a transaction to deposit Rs 100/- from account A to account B. The transaction should now be written as the following:

Lock-X (A); (Exclusive Lock, we want to both read A’s value and modify it) Read A

A = A – 100

Write A

Unlock (A) (Unlocking A after the modification is done)

Lock-X (B) (Exclusive Lock, we want to both read B’s value and modify it) Read B

B = B + 100

Write B

Unlock (B) (Unlocking B after the modification is done)

Let us see how these locking mechanisms help us to create error free schedules. You should remember that in the previous chapter we discussed an example of an erroneous schedule:

Transaction T1 Transaction T2

Read(A) A=A-100

Read(A) Temp=A\*0.1 Read(C) C=C + Temp Write(C)

Write(A) Read(B)

B = B + 100

Write (B)

We detected the error based on common sense only that the Context Switching is being performed before the new value has been updated in A. T2 reads the old value of A, and thus deposits a wrong amount in C. Had we used the locking mechanism, this error could never have occurred.

***Let us rewrite the schedule using the locks.***

Transaction T1 Transaction T2 Concurrency control manager

Lock – X(A)

Read A

A = A - 100

Write A Unlock(A)

Lock – S (A) Read A

Temp = A \*0.1

grant – X(A,T1)

grant – S(A,T2)

Unlock (A) grant – X(C,T2) Lock – X (C)

Lock – X(B) Read B

B = B + 100

Write B Unlock (B)

Read (C)

C = C + Temp Write C Unlock (C)

grant – X(B,T1)

And this automatically becomes a very correct schedule.

Problems in lock based protocols:

1. May not sufficient to produce only serializable schedule.
2. May not free from nonrecoverability
3. May not free from deadlock
4. May not free from starvation

***May not sufficient to produce only serializable schedule.***

|  |  |  |
| --- | --- | --- |
| **Transaction T1** |  | **Transaction T2** |
| Lock – X(A) |  |  |
| R(A) |  |  |
| W(A) |  |  |
| Unlock(A) |  |  |
|  |  | Lock – S(B) |
|  |  | R(B) |
|  |  | Unlock (B) |
| Lock – X(B) |  |  |
| R(B) |  |  |
| W(B) |  |  |
| Unlock (B) |  |  |
|  |  | Lock – S(A) |
|  |  | R(A) |
|  |  | Unlock(A) |

Above example is not in conflict serializable (cycle) Since

W1(A), R2(A){ T1 → T2}

R2(B),W1(B){ T2 → T1}

***Deadlock:***

|  |  |
| --- | --- |
| ***T1*** | ***T2*** |
| *L – X(A)* |  |
|  | *L – X(B)* |
| *L – X(B)* |  |
|  | *L – X(A)* |

***Starvation***

|  |  |  |  |
| --- | --- | --- | --- |
| ***T1*** | ***T2*** | ***T3*** | ***T4*** |
| *Lock - S(A)* |  |  |  |
|  | *Lock - X (A* |  |  |
|  |  | *Lock - S(A* |  |
|  |  |  | *Lock - S(A* |

Two Phase Locking Protocol

A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases.

1. **Growing Phase:** New locks on data items may be acquired but none can be released.
2. **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.

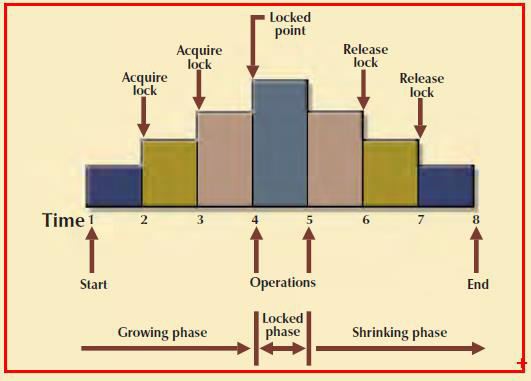
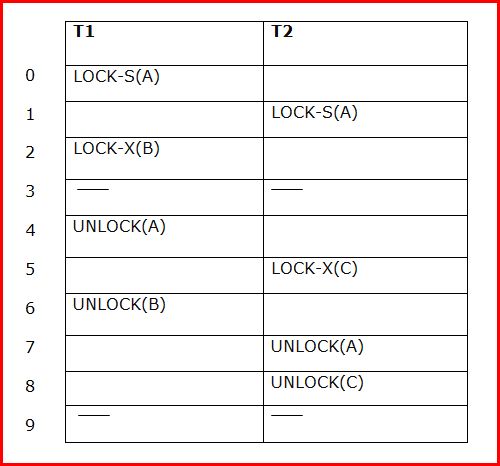


Fig: Depicts Two Phase locking protocol

In this example, the transaction acquires all of the locks it needs until it reaches its locked point. (In this example, the transaction requires two locks.) When the locked point is reached, the data are modified to conform to the transaction’s requirements. Finally, the transaction is completed as it releases all of the locks it acquired in the first phase.

**Note –** If lock conversion is allowed, then upgrading of lock( from S(a) to X(a) ) is allowed in the Growing Phase, and downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Example:



This protocol ensures a serializable schedule

|  |  |
| --- | --- |
| T1 | T2 |
| Lock – S(A) |  |
| Read(A) |  |
| Lock – S(B) |  |
| Read(B) |  |
|  | Lock - S(A) |
|  | Read(A) |
|  | Unlock(A) |
| Unlock (A) |  |
| Unlock(A) |  |

In this example, read locks for both A and B were acquired. Since both transactions did nothing but read, this is easily identifiable as a serializable schedule.

Unserializable schedule example:



Using the 2-phase locking protocol, we get the following:



We see that the locking protocol keeps the schedule serializable.

Timestamp Based protocols

Basic idea of time stamping is to decide the order between the transactions before they enter into the system, so that in case of conflict during execution, we can resolve the conflict using ordering.

Time stamp:

With each transaction Ti, in the system, we associate a unique fixed timestamp, de noted by TS (Ti). This timestamp is assigned by the database system before the transaction Ti, starts execution. If a transaction Ti, has been assigned timestamp TS(Ti), and a new transaction Tj, enters the system, then TS(Ti)) < TS(Tj)).

There are two simple methods for implementing this scheme:

1. Use the value of the system clock as the timestamp; that is, a transaction's time stamp is equal to the value of the clock when the transaction enters the system.
2. Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the Serializability order. Thus, if TS(Ti) <TS(Tj), then the system must ensure that the produced schedule is equiva - lent to a serial schedule in which transaction Ti, appears before transaction Tj.

To implement this scheme, we associate with each data item Q two timestamp values:

* + W-timestamp (WTS) (Q) denotes the largest / last / latest timestamp of any transaction that executed write (Q) successfully.
  + R-timestamp RTS (Q) denotes the largest / last / latest timestamp of any transaction that executed read(Q) successfully.

These timestamps are updated whenever a new read (Q) or write(Q) instruction is executed.

Timestamp - ordering Protocol:

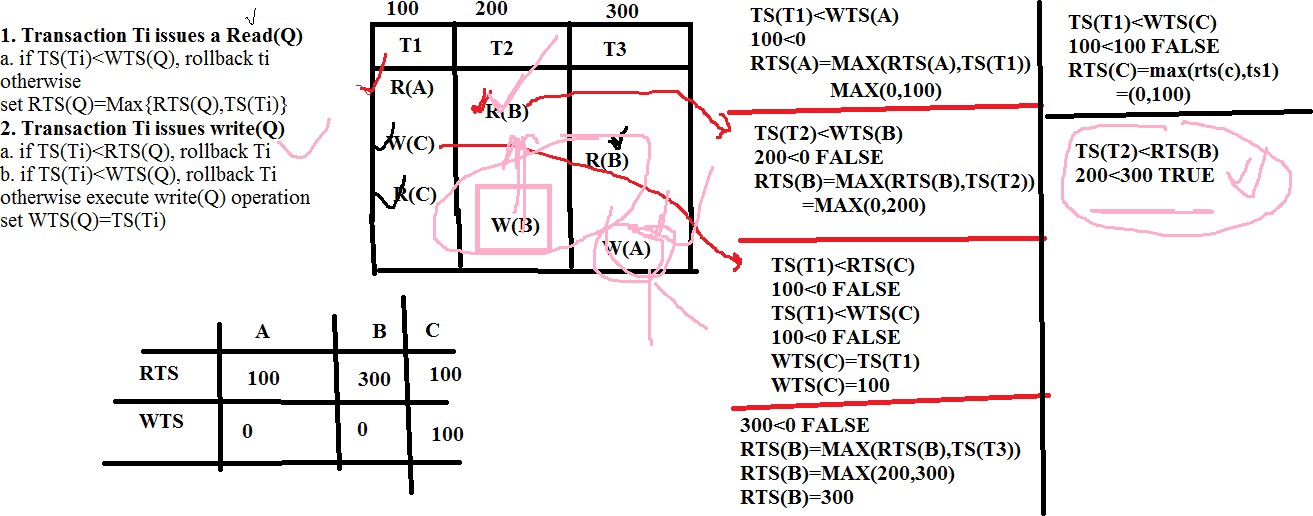
The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction Ti issues read (Q).
   1. If TS(Ti) < W-timestamp(Q), then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.
   2. If TS(Ti) ≥ W-timestamp(Q), then the read operation is executed, and set RTS(Q)=Max(RTS(Q),TS(Ti))
2. Suppose that transaction Ti issues write (Q).
   1. If TS(Ti) < R-timestamp(Q), then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls Ti back. or
   2. If TS(Ti) < W-timestamp(Q), then Ti is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls Ti back.

Otherwise**,** the system executes the write operation and sets W-timestamp(Q) = TS(Ti).

If a transaction Ti is rolled back by the concurrency-control scheme as result of issuance of either a read or writes operation, the system assigns it a new timestamp and restarts it.

Example:



**Validation Based Protocols**

In Validation based Protocol, the local copies of the transaction data are updated rather than the data itself, which results in less interference while execution of the transaction.

Validation based Protocol also known as Optimistic Concurrency Control since transaction executes fully in the hope that all will go well during validation.

Execution of transaction Ti is done in three phases:

* + 1. **Read and execution phase:** During this phase, the system executes transaction Ti. It reads the values of the various data items and stores them in variable local to Ti. It performs all the write operations on temporary local variables without update of the actual database.
    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:

1. Start (Ti): the time when Ti started its execution
2. Validation (Ti): the time when Ti entered its validation phase
3. 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.

Validation Test for Transaction Tj

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.

Example:

**Case 1:**

Ti(TS=9:30) Tj(TS=9:37)

Finish(Ti)

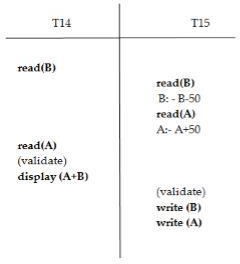
Start(Tj)

Finish (Ti) < Start(Tj), Validation based protocol

Case 2:

|  |  |
| --- | --- |
| **Ti(TS(2:30))** | **Tj(TS(2:37))** |
|  |  |
|  |  |
|  | Start(Tj), 2:37 |
|  |  |
| Finish(Ti),2:39 |  |
|  |  |
|  | Validation(Tj), 2:40 |
|  |  |

**Example:**



It is a validated schedule

**Advantages:**

1. Avoid Cascading roll backs.
2. Avoid deadlock

Disadvantages:

1. Starvation

Failure Classification

To find that where the problem has occurred, we generalize a failure into the following categories:

1. Transaction failure
2. System crash
3. Disk failure
4. Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

* 1. **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.
  2. **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. For example**,** the system aborts an active transaction, in case of deadlock or resource unavailability.

1. System Crash

System failure can occur due to power failure or other hardware or software failure. Example: Operating system error.

In the system crash, non-volatile storage is assumed not to be corrupted.

1. Disk Failure

* It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
* Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

Recovery and atomicity:

Consider banking system

|  |  |
| --- | --- |
| Account X | Account Y |
| Initial amount = 5000 | Initial amount = 10000 |
| Transfer 1000 from X to Y | |
| System Crash occurred after amount deducted from account X but before adding to account Y  Memory content were lost because of system crash | |

We could use one of the two possible recovery procedures.

**Re - execute Ti:** This procedure will result in the value of X becoming 3000 rather than 4000, Thus, the system enters an inconsistent state.

**Do not re - execute Ti:** The current system state has values of 4000 and 10000 for X and Y, respectively. Thus, the system enters an inconsistent state.

In either case, the DB is left in inconsistent state, and thus simple recovery schemes do not work.

Scheme to achieve the recovery from transaction failure is log based recovery.

Log based recovery

The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.

If any operation is performed on the database, then it will be recorded in the log. But the process of storing the logs should be done before the actual transaction is applied in the database.

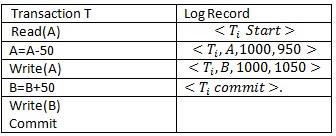
Update log has the following fields-

* 1. **Transaction Identifier**: To get the Transaction that is executing.
  2. **Data item Identifier:** To get the data item of the Transaction that is running.
  3. The old value of the data item (Before the write operation).
  4. The new value of the data item (After the write operation).

Structure of the format of a log record.

1. **<T, Start >.** The Transaction has started.
2. **<T, X, V1, V2>.** The Transaction has performed write on data. V1 is a value that X will have value before writing, and V2 is a Value that X will have after the writing operation.
3. **<T, Commit>**. The Transaction has been committed.
4. **<T, Abort>.** The Transaction has aborted.

Consider the data Item A and B with initial value 1000. (A=B=1000)



In the above table, in the left column, a transaction is written, and in the right column of the table, a log record is written for this Transaction.

Log Based Recovery work in two modes, these modes are as follow-

1. Deferred database modification:

In the Deferred Mode of Log-based recovery method, all modifications to Database are recorded but WRITE Operation is deferred until the Transaction is partially committed. It means In the case of deferred mode, Database is modified after Commit operation of Transaction is performed.

Example:

Consider Transaction T0. It transfers Rs. 50 from account A to account B. Original value of account A and B are Rs. 1000 and Rs. 2000 respectively. This transaction is defined as:

T0 read(A) A:=A - 50

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

write(B)

Consider Second Transaction T1 that withdraws 100 rupees from account C. Original value of C is 700.

This transaction is defined as

T1 read(C) A:=A-100

write(C)

These two transactions are executed serially <T0, T1>

The log containing relevant information on these two transactions is given below:

<T0, start>

<T0, A, 950>

<T0, B, 2050>

<T0, commit>

<T1, start>

<T1, C, 600>

<T1, commit>

The below fig shows the log that results from the complete execution of T0 and T1.

|  |  |
| --- | --- |
| Log | Database |
| < T0, start> |  |
| < T0, A, 950> |  |
| < T0, B, 2050> |  |
| < T0, commit> |  |
|  | A = 950 |
|  | B = 2050 |
| <T1, start> |  |
| <T1, C, 600> |  |
| <T1, commit> |  |
|  | C = 600 |

**Case 1: Crash occurs just after the log record for written (B) operation.**

Log contents after the crash are:

< T0, start>

< T0, A, 950>

< T0, B, 2050>

<T0, commit> is not written, hence no redo operation is possible. The value of account remain unchanged i.e account balance of A is 1000 and 2000.

**Case 2: Crash occur just after the log record for written(C) operation.**

Log contents after the crash are:

< T0, start>

< T0, A, 950>

< T0, B, 2050>

< T0, commit>

<T1, start>

<T1, C, 600>

When the system comes back it finds <T0, start> and <T0, commit>. But there is no

<T1, commit> for <T1, start> Hence, system can execute redo (T0) but not redo (TI). Hence, the value of account C remains unchanged.

**Case 3: Crash occur just after the log record <T1, commit>**

Log contents after the crash are:

<T0, start>

<T0, A, 950>

<T0, B, 2050>

<T0, commit>

<T1, start>

<T1, C, 600>

<T1, commit>

When the system comes back it can execute redo(T0) and redo(T1) operations.

1. Immediate database modification:

In immediate Mode of log-based recovery, database modification is performed while Transaction is in Active State.

It means as soon as Transaction is performed or executes its WRITE Operation, then immediately these changes are saved in Database also**.** In immediate Mode, there is no need to wait for the execution of [the COMMIT Statement](https://docs.microsoft.com/en-us/sql/t-sql/language-elements/commit-transaction-transact-sql?view=sql-server-ver15) to update the Database.

Data modifications written by active transactions are called uncommitted modifications.

Consider the previous example:

The log corresponding to this execution is given below

<T0, start>

<T0, A, 1000, 950>

<T0, B, 2000, 2050>

<T0, commit>

<T1, start>

<T1, C, 700, 600>

<T1, commit>

The order in which output took place to both database system and log as a result of execution of T0 and T1 is

|  |  |
| --- | --- |
| Log | Database |
| < T0, start> |  |
| < T0, A,1000,950> |  |
| < T0, B, 2000, 2050> |  |
|  | A = 950 |
|  | B = 2050 |
| < T0, commit> |  |
| <T1, start> |  |
| <T1, C, 700, 600> |  |
|  | C = 600 |
| <T1, commit> |  |

Using the log, the system can handle failure. Two recovery procedures are

* undo(Ti) restores the value of all data items updated by transaction Ti, to the old values
* redo(Ti) sets the values of all data items updated by transaction Ti to the new values.

Case 1: Failure occur just after the log record for write(B) operation

|  |  |
| --- | --- |
| < T0, start> |  |
| < T0, A,1000,950> |  |
| < T0, B, 2000, 2050> |  |
|  | A = 950 |
|  | B = 2050 |

Log contains < T0, start> but does not contain < T0, commit>. Hence undo T0 is executed: the values of account A and B are restored to 1000 and 2000 resp.

Case 2: Failure occurs just after the log record for write (C) operation.

|  |  |
| --- | --- |
| < T0, start> |  |
| < T0, A,1000,950> |  |
| < T0, B, 2000, 2050> |  |
|  | A = 950 |
|  | B = 2050 |
| < T0, commit> |  |
| <T1, start> |  |
| <T1, C, 700, 600> |  |
|  | C = 600 |

Log contains <T0, start> and <T0, commit>. Hence, the redo(T0) is executed and values of account A and B are restored to the same (or new values) 950 and 2050 respectively. But log doesn't contain <T1, commit> for <T1, Start> Hence, the value of account C to restored to old value by undo(T1) operation. Hence value of C is 700.

Case 3: Crash occur just after the log record <T1, commit>

|  |  |
| --- | --- |
| Log | Database |
| < T0, start> |  |
| < T0, A,1000,950> |  |
| < T0, B, 2000, 2050> |  |
|  | A = 950 |
|  | B = 2050 |
| < T0, commit> |  |
| <T1, start> |  |
| <T1, C, 700, 600> |  |
|  | C = 600 |
| <T1, commit> |  |

Log contains < T0, start>, < T0, commit>and <T1, start>, <T1, commit> Hence, recovery system executes redo (T0) and redo(T1) operation. The value of account

A. B and C are Rs 950, Rs 2050 and Rs 600 respectively

Recovery with Concurrent Transaction

Until now, we considered recovery in an environment where only a single trans- action at a time is executing. We now discuss how we can modify and extend the log-based recovery scheme to deal with multiple concurrent transactions. Regardless of the number of concurrent transactions, the system has a single disk buffer and a single log.

*Recovery with Concurrent Transactions means to recover schedule when multiple transaction executed if any transaction fail.*

Interaction with Concurrency Control

The recovery scheme depends greatly on the concurrency-control scheme that is used. To roll back a failed transaction, we must undo the updates performed by the transaction. Suppose that a transaction *T*0 has to be rolled back, and a data item *Q* that was updated by *T*0 has to be restored to its old value. Using the log-based schemes for recovery, we restore the value by using the undo information in a log record. Suppose now that a second transaction *T*1 has performed yet another update on *Q before T*0 is rolled back. Then, the update performed by *T*1 will be lost if *T*0 is rolled back.

Therefore, we require that, if a transaction *T* has updated a data item *Q*, no other transaction may update the same data item until *T* has committed or been rolled back. We can ensure this requirement easily by using strict two-phase locking— that is, two-phase locking with exclusive locks held until the end of the transaction.

Transaction Rollback

We roll back a failed transaction, *Ti*, by using the log. The system scans the log backward; for every log record of the form *<Ti, Xj , V*1*, V*2*>* found in the log, the system restores the data item *Xj* to its old value *V*1. Scanning of the log terminates when the log record *<Ti,* start*>* is found.

Scanning the log backward is important, since a transaction may have updated a data item more than once. As an illustration, consider the pair of log records

*<Ti, A,* 10*,* 20*>*

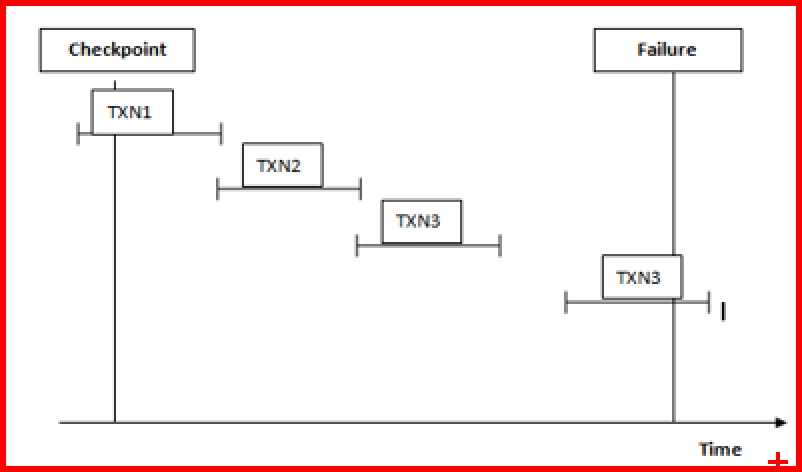
*<Ti, A,* 20*,* 30*>*

The log records represent a modiﬁcation of data item *A* by *Ti*, followed by another modiﬁcation of *A* by *Ti*. Scanning the log backward sets *A* correctly to 10. If the log were scanned in the forward direction, *A* would be set to 20, which is incorrect.

Checkpoint

In database management systems when we execute multiple transactions at the same time log files are created and it utilizes system memory space for storage. The size of the log file may increase continuously and at one point it is difficult to handle. With the help of a checkpoint mechanism, we can remove all log files from the system and store them as secondary storage disks permanently. We can declare a checkpoint before which the DBMS is inconsistent state and all transactions are committed. Sometimes transaction execution may fail and it requires log files for re-execution, but searching log files is a time-consuming process, so we can overcome this problem by using checkpoints.

Let’s understand how a checkpoint works in DBMS with the help of the following diagram.



1. Checkpoint is used in the recovery system to recover failure transactions. The above diagram shows four transactions such as TXN1, TXN2, TXN3, and TXN4. In which that recovery system reads all files from start to end means it reads TXN1 to TXN4.
2. The recovery system maintains two different lists for undo and redo.
3. The transaction goes into the redo list if the recovery system sees all logs file with <TXN N, Commit>. In redo list, all log files as well as all the transactions are removed and differently save their logs files.
4. For example: in all log file transactions TXN2 and TXN3 will have <TXN N, Start> and <TXN N, Commit>. The TXN1 commits in the log file that is the main reason transaction TXN1 is committed after the checkpoint is crossed. So TXN1, TXN2, and TXN3 go into the redo list.
5. In the next step recovery system sees all log files but there is no commit or abort log are found in the undo list so all transactions are undone and all log files are removed from the list.
6. If transaction TXN 4 fails will be put into the undo list because transaction TXN4 is not completed.

\*\* When the Transaction reach to Checkpoint, its updates DB and Remove the Log than again move towards to next Checkpoint and do same (Update DB Remove the Log).

\*\* It is like a bookmark.

Restart recovery:

* When the system recovers from a crash, it constructs two lists.
* The undo-list consists of transactions to be undone, and the redo-list consists of transaction to be redone.
* The system constructs the two lists as follows: Initially, they are both empty. The system scans the log backward, examining each record, until it finds the first <checkpoint> record.

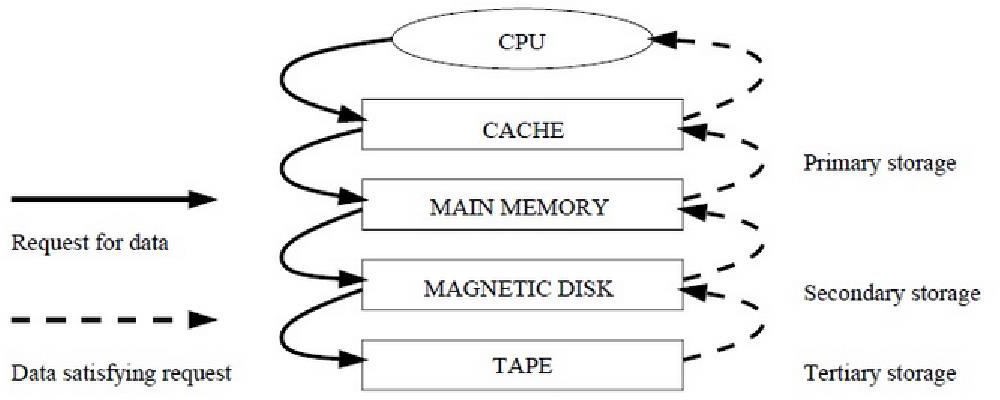
UNIT - V

**Data on External storage**

A DBMS stores the data on external storage because the amount of data is very huge, must persist across program executions and has to be fetched into main memory when DBMS processes the data.

The unit of information for reading data from disk, or writing data to disk, is a page. The size of a page is 4KB or 8KB.

Memory in a computer system is arranged in a hierarchy, as shown in Figure. At the top, we have primary storage, which consists of cache and main memory and provides very fast access to data. Then comes secondary storage, which consists of slower devices, such as magnetic disks (Floppy disk & Hard disk drive). Tertiary storage is the slowest class of storage devices; for example, optical disks (CD) and tapes.



Each record in a file has a unique identifier called a record id, or **rid** for short. A rid has the property that we can identify the disk address of the page containing the record by using the rid.

Buffer Manager:

Data is read into memory for processing, and written to disk for persistent storage, by a layer of software called the *buffer manager.* When the files and access

methods layer (which we often refer to as just the file layer) needs to process a page, it asks the buffer manager to fetch the page, specifying the page's rid. The buffer manager fetches the page from disk if it is not already in memory.

Disk space manager:

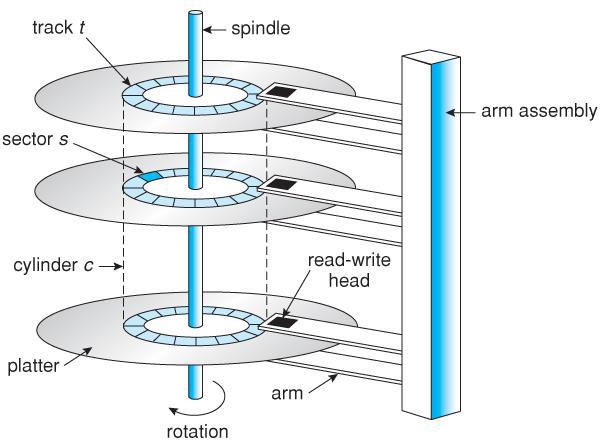
Space on disk is managed by the *disk space manager.* When the files and access methods layer needs additional space to hold new records in a file, it asks the disk space manager to allocate an additional disk page for the file;

Magnetic Disks:

Magnetic disks support direct access (transfers the block of data between the *memory* and *peripheral devices* of the system, without the participation of the processor) to a desired location and are widely used for database applications. A DBMS provides seamless access to data on disk; applications need not worry about whether data is in main memory or disk.







**Platter:** A platter is a circular magnetic plate that is used for storing data in a hard disk. It is often made of aluminum, glass substrate or ceramic. A hard disk drive contains several platters. Each platter has two working surfaces. These surfaces of the platters hold the recorded data.

**Spindle:** A typical HDD design consists of a spindle, which is a motor that holds the platters.

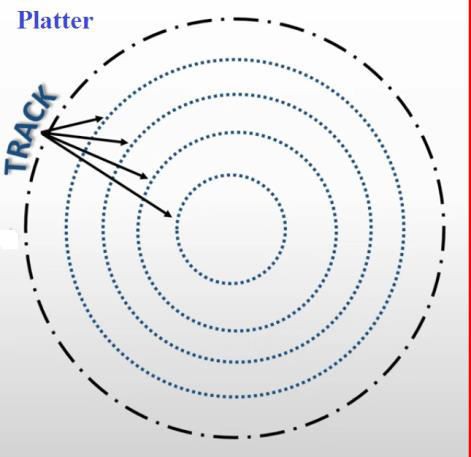
**Tracks:** Each working surface of the platter is divided into number of concentric rings, which are called tracks.

**Cylinder:** The collection of all the tracks that are of the same distance, from the edge of the platter, is called a cylinder.

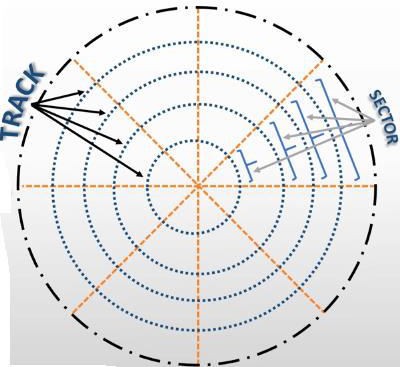
**Read / Write Head:** The data on a hard drive platter is read by read – write heads, of read – write arm. The read – write arm also known as actuator.

**Arm assembly:** Each on a separate read – write arm are controlled by a common arm assembly which moves all heads simultaneously from one cylinder to another.

**Tracks:** Each platter is broken into thousands of tightly packed concentric circles, known as tracks. These tracks resemble the structure of annual rings of a tree. All the information stored on the hard disk is recorded in tracks. Starting from zero at the outer side of the platter, the number of tracks goes on increasing to the inner side. Each track can hold a large amount of data counting to thousands of bytes.



**Sectors:** Each track is further broken down into smaller units called sectors. As sector is the basic unit of data storage on a hard disk, each track has the same number of sectors, which means that the sectors are packed much closer together on tracks near the center of the disk. A single track typically can have thousands of sectors. The data size of a sector is always a power of two, and is almost always either 512 or 4096 bytes.



**Clusters:** Sectors are often grouped together to form clusters. A cluster is the smallest possible unit of storage on a hard disk. If contiguous clusters (clusters that are next to each other on the disk) are not available, the data is written elsewhere on the disk, and the file is considered to be fragmented.



File Organization and Indexing

A database consists of a huge amount of data. The data is grouped within a table in RDBMS, and each table has related records. A user can see that the data is stored in form of tables, but in actual this huge amount of data is stored in physical memory in form of files.

**File:** A file is named collection of related information that is recorded on secondary storage such as magnetic disks, optical disks.

File Organization:

File Organization refers to the logical relationships among various records that constitute the file, particularly with respect to the means of identification and access to any specific record. In simple terms, Storing the files in certain order is called file Organization.

Indexing:

The main goal of designing the database is faster access to any data in the database and quicker insert/delete/update to any data. When a database is very huge, even a smallest transaction will take time to perform the action. In order to reduce the time spent in transactions, Indexes are used. Indexes are similar to book catalogues in library or even like an index in a book.

Indexing is a data structure technique which allows you to quickly retrieve records from a database file. An Index is a small table having only two columns. The first column comprises a copy of the primary or candidate key of a table. Its second column contains a set of [pointers](https://www.guru99.com/c-pointers.html) for holding the address of the disk block where that specific key value stored.

An index

* Takes a search key as input
* Efficiently returns a collection of matching records.

Types of Indexing:

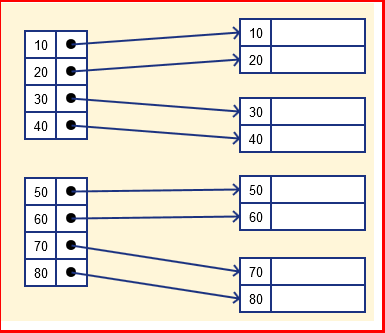
**Primary Index:**

Primary Index is an ordered file which is fixed length size with two fields. The first field is the same a primary key and second, filed is pointed to that specific data block. In the primary Index, there is always one to one relationship between the entries in the index table.

The primary Indexing in DBMS is also further divided into two types.

* Dense Index
* Sparse Index

**Dense Index:** In a dense index, a record is created for every search key valued in the database. This helps you to search faster but needs more space to store index records. In this Indexing, method records contain search key value and points to the real record on the disk.

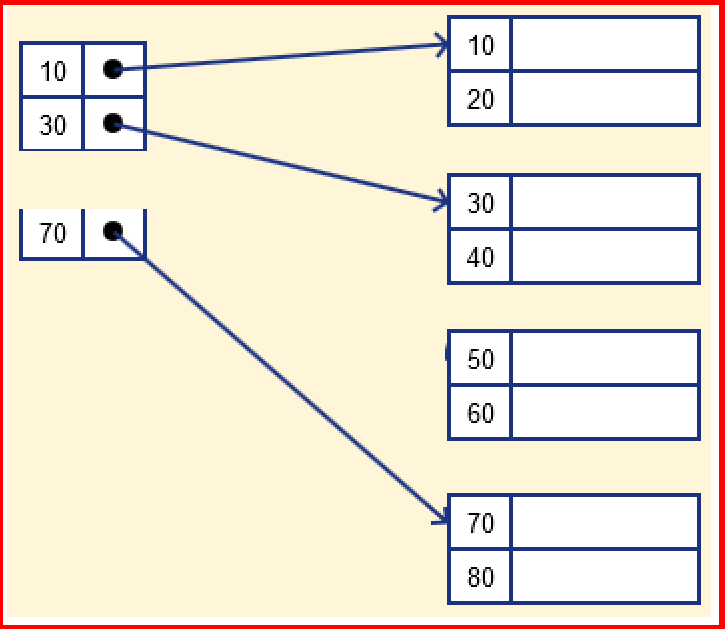


Dense Index

Sparse Index

It is an index record that appears for only some of the values in the file. Sparse Index helps you to resolve the issues of dense Indexing in DBMS. In this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrieved, the block address will be fetched.

However, sparse Index stores index records for only some search-key values. It needs less space, less maintenance overhead for insertion, and deletions but It is slower compared to the dense Index for locating records.



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. Clustered index sorted according to first name (Search key).

For example, students studying in each semester are grouped together. i.e. 1st

Semester students, 2nd semester students, 3rd semester students etc are grouped.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Example:** |  | | | |
|  |  |  | **dno** | **----** |
|  |  |  | **1** |  |
|  |  | **B1** | **1** |  |
| **Key** | **Pointer** |  | **2** |  |
| **1** | **.** |  | **2** |  |
| **2** | **.** |  | **3** |  |
| **3** | **.** | **B2** | **3** |  |
| **4** | **.** |  | **3** |  |
| **5** | **.** |  | **4** |  |
| **6** | **.** | **B3** | **4** |  |
|  |  |  | **4** |  |
|  |  |  | **5** |  |
|  |  | **B4** | **5** |  |
|  |  |  | **6** |  |
|  |  |  | **6** |  |

**Secondary Index:** Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.

Example:

13

.

.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **eid** | **Name** | **Pan no** |  | | |
| 1 | A | 40 |  | **Key** | **Pointer** |
| 2 | B | 51 |  | 23 | **.** |
| 3 | A | 62 |  | 33 | **.** |
| 4 | C | 33 |  | 40 | **.** |
| 5 | A | 71 |  | 51 | **.** |
| 6 | D | 82 |  | 62 | **.** |
| 7 | E | 91 |  | 71 | **.** |
| 8 | F | 23 |  | 82 | **.** |
| 9 | K | 100 |  | 91 | **.** |
| 10 | L | 120 |  | 100 | **.** |
| 11 | M | 150 |  |  |  |
| 12 | C | 136 |  |  |  |

**INDEX DATA STRUCTURES**

Hash Based Indexing

Hashing is the technique of the database management system, which directly finds the specific data location on the disk without using the concept of index structure.

In the database systems, data is stored at the blocks whose data address is produced by the hash function. That location of memory where hash files stored these records is called as data bucket or data block.

Terms used in Hashing:

**Data Bucket:** That location of memory where the hash file stored the records. It is also referred to as a Unit of storage**.**

**Key:** A key in the Database Management system (DBMS) is a field or set of fields that helps the relational database users to uniquely identify the row/records of the database table.

**Hash function:** This mapping function matches all the set of search keys to those addresses where the actual records are located. It is an easy mathematics function.

**Linear Probing:** It is a concept in which the next available block of data is used for inserting the new record instead of overwriting the older data block.

**Quadratic Probing:** It is a method that helps users to determine or find the address of a new data bucket.

**Double Hashing**: Double hashing is a computer programming method used in hash tables to resolve the issues of has a collision.

**Bucket Overflow:** When a record is inserted, the address generated by the hash function is not empty or data already exists in that address .This situation is called bucket overflow.

There are mainly two types of SQL hashing methods:

1. Static Hashing
2. Dynamic Hashing

**Static Hashing:** In the static hashing, the resultant data bucket address will always remain the same.

Therefore, if you generate an address for say Student\_ID = 10 using hashing function mod(3), the resultant bucket address will always be 1. So, you will not see any change in the bucket address.

Therefore, in this static hashing method, the number of data buckets in memory always remains constant.

Static Hash Functions

**Inserting a record:** When a new record requires to be inserted into the table, you can generate an address for the new record using its hash key. When the address is generated, the record is automatically stored in that location.

**Searching**: When you need to retrieve the record, the same hash function should be helpful to retrieve the address of the bucket where data should be stored.

**Delete a record**: Using the hash function, you can first fetch the record which is you wants to delete. Then you can remove the records for that address in memory.

Static hashing is further divided into

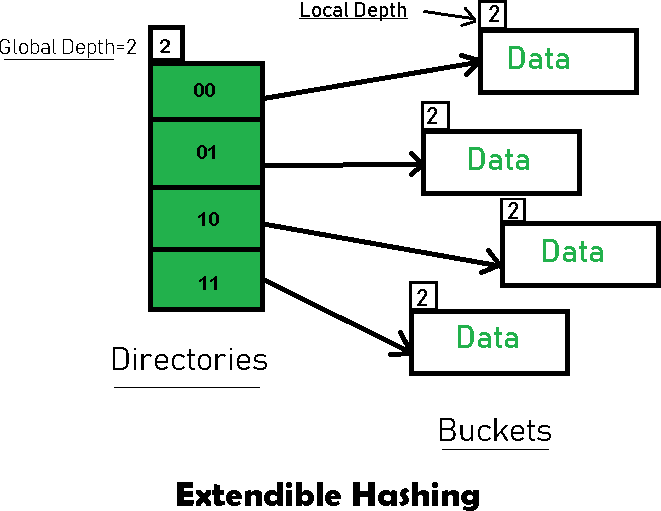
1. Open hashing (Chaining)
2. Close hashing. (Linear probing, quadratic probing, Double hashing)

DYNAMIC HASHING TECHNIQUES EXTENDIBLE HASHING

In Static Hashing the performance degrades with overflow pages. This problem, however, can be overcome by a simple idea: Use a directory of pointers to buckets, and double the size of the number of buckets by doubling just the directory and splitting only the bucket that overflowed which the central concept of Extendible is hashing.

It is a dynamic hashing method wherein directories and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.

Basic structure of Extendible hashing:



**Frequently used terms in Extendible Hashing:**

* + **Directories:** These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place. The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories = 2^Global Depth.
  + **Buckets:** They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointer to it if its local depth is less than the global depth.
  + **Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.
  + **Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth

in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.

* + **Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
  + **Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.

**Example based on Extendible Hashing:** Now, let us consider a prominent example of hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**

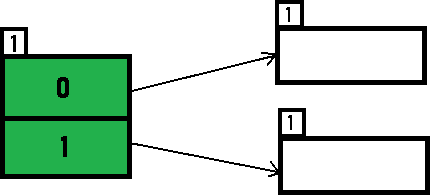
**Bucket Size:** 3 (Assume)

**Hash Function:** Suppose the global depth is X. Then the Hash Function returns X LSBs.

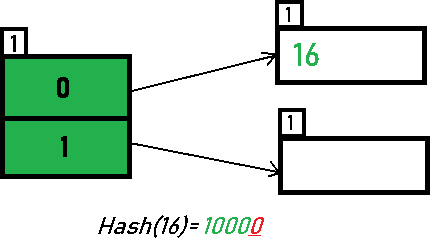
**Solution:** First, calculate the binary forms of each of the given numbers.

|  |  |
| --- | --- |
| 16 | 10000 |
| 4 | 00100 |
| 6 | 00110 |
| 22 | 10110 |
| 24 | 11000 |
| 10 | 01010 |
| 31 | 11111 |
| 7 | 00111 |
| 9 | 01001 |
| 20 | 10100 |
| 26 | 11010 |

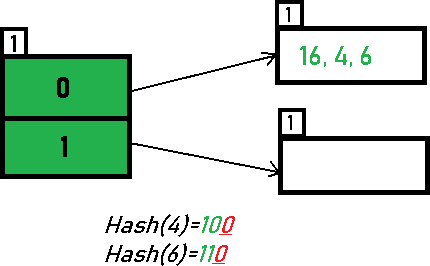
Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



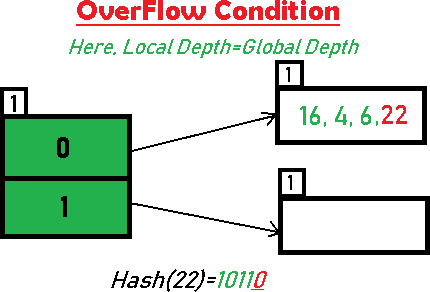
**Inserting 16:** The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.



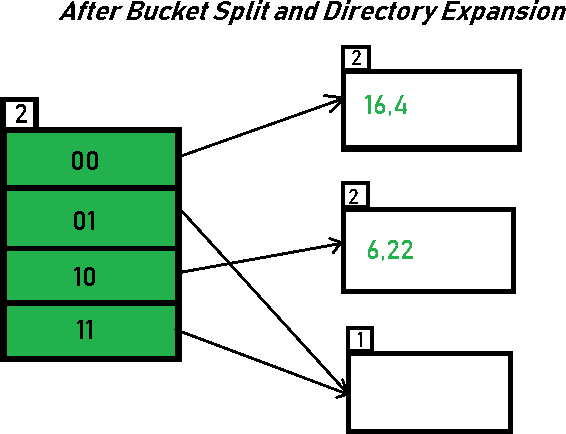
**Inserting 4 and 6:** Both 4(10**0**) and 6(11**0**) have 0 in their LSB. Hence, they are hashed as follows:



**Inserting 22:** The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.

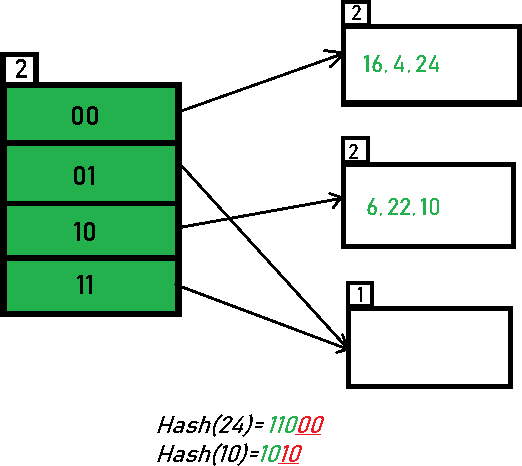


Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now, the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[ 16(100**00**),4(1**00**),6(1**10**),22(101**10**) ]

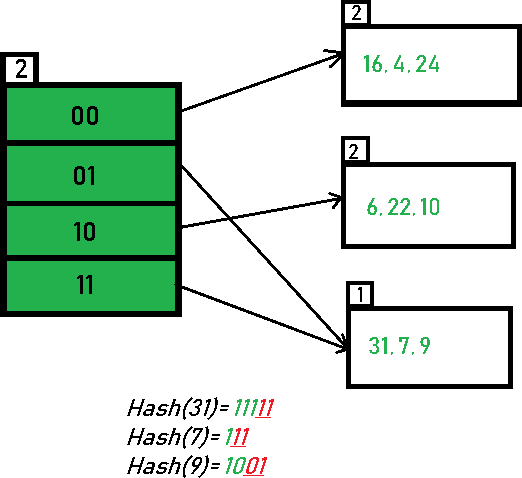


* + - Notice that the bucket which was underflow has remained untouched. But, since the number of directories has doubled, we now have 2 directories 01 and 11 pointing to the same bucket. This is because the local-depth of the bucket has remained 1. And, any bucket having a local depth less than the global depth is pointed-to by more than one directory.

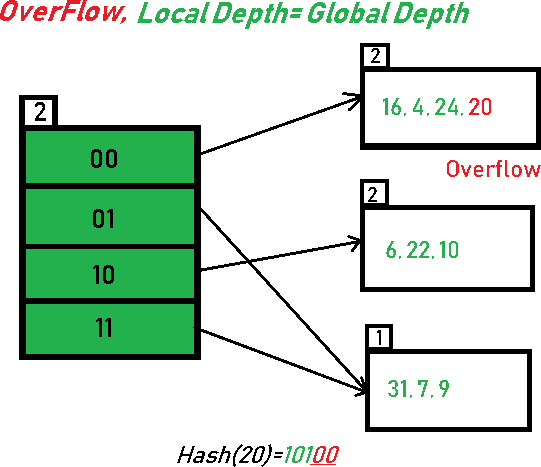
**Inserting 24 and 10:** 24(110**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



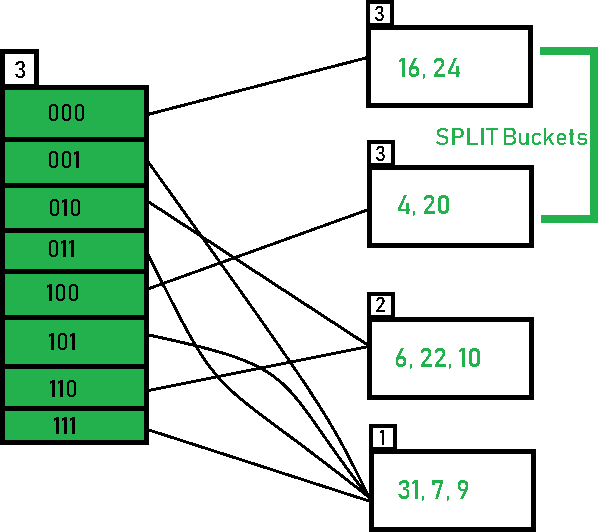
**Inserting 31, 7, 9:** All of these elements [31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



**Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.

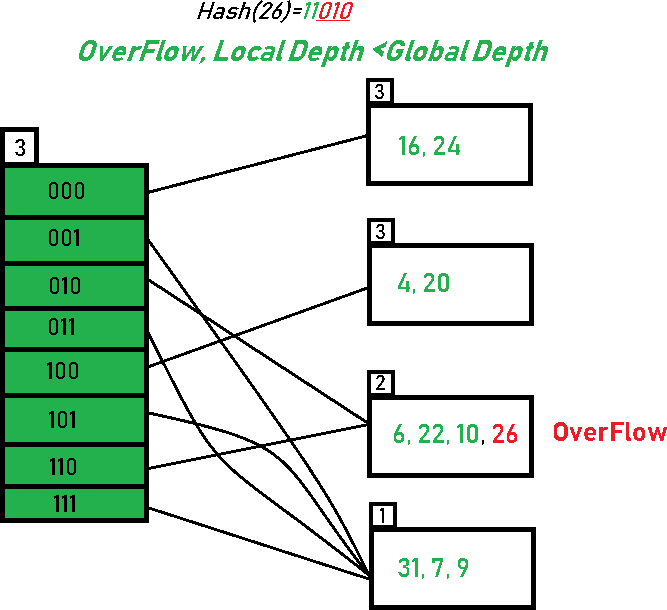


20 is inserted in bucket pointed out by 00. Since the local depth of the bucket = global-depth**,** directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:

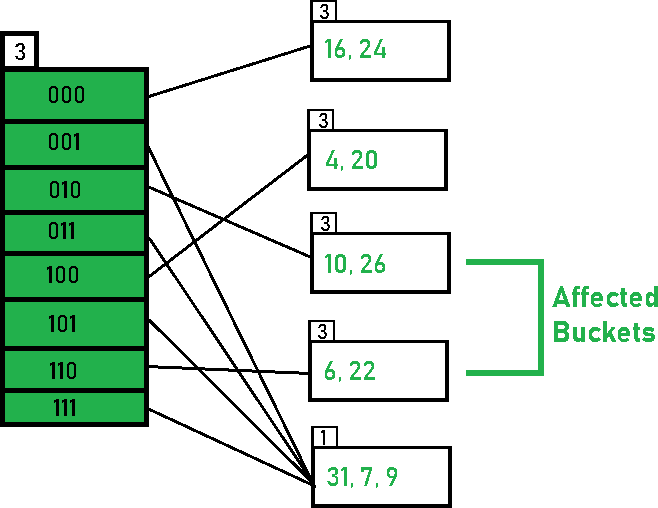


**Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered.

Therefore 26 best fits in the bucket pointed out by directory 010.

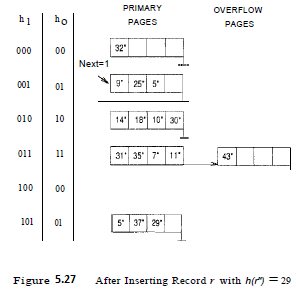
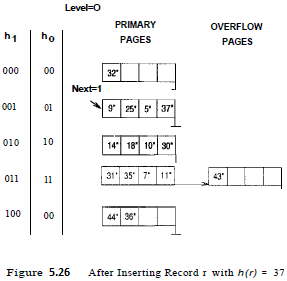
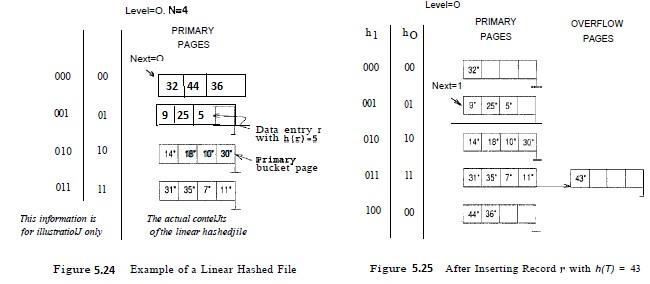


The bucket overflows, and, since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed. Finally, the output of hashing the given list of numbers is obtained.



Hashing of 11 Numbers is Thus Completed.

LINEAR HASHING



245

Linear Hashing is a dynamic hashing technique, it does not require a directory, deals naturally with collisions, and offers a lot of flexibility with respect to the timing of bucket splits. If the data distribution is much skewed, however, overflow chains could cause Linear Hashing performance to be worse than that of Extendible Hashing.

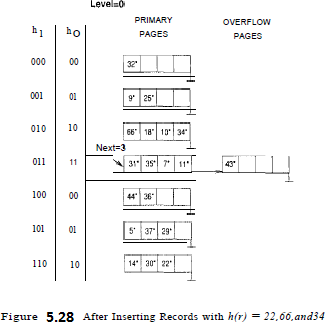
The scheme utilizes a family of hash functions *h0, h1, h2, ...* , with the property that each function's range is *twice* that of its predecessor. That is, if *hi* maps a data entry into one of M buckets, hi+1 maps a data entry into one of M buckets. Such a family is typically obtained by choosing a hash function h and an initial number N of buckets,2 and defining *hi(value) = h(value) mod (2i N)*.

The idea is best understood in terms of rounds of splitting. During round number Level, only hash functions *hLevel* and *hLevel+1* are in use. The buckets in the file at the beginning of the round are split, one by one from the first to the last bucket, thereby doubling the number of buckets. At any given point within a round, therefore, we have buckets that have been split, buckets that are yet to be split, and buckets created by splits in this round.

Consider how we search for a data entry with a given search key value. We apply hash function *h Level* , and if this leads us to one of the unsplit buckets, we simply look there. If it leads us to one of the split buckets, the entry may be there or it may have been moved to the new bucket created earlier in this round by splitting this bucket; to determine which of the two buckets contains the entry, we apply *hLevel+1.*

D

atabase Management Systems G Deepak, Asst.Prof, KITS(S)



Tree Based Indexing

1. ISAM (Indexed sequential access method)
   * Static Index Structure
2. B+ Trees
   * Dynamic Index Structure

**B+ Tree:** B+ tree is dynamic index structure i.e the height of the tree grows and contracts as records are added and deleted.

A B+ tree is also known as balanced tree in which every path from the root of the tree to a leaf is of the same length.

Leaf node of B+ trees are linked, so doing a linear search of all keys will require just one pass through all the leaf nodes.

B+ tree combines features of ISAM and B tress. It contains index pages and data pages. The data pages always appear as leaf nodes in the tree. The root node and intermediate nodes are always index pages. These features are similar to ISAM unlike ISAM, overflow pages are not used in B+ trees.

For order M

* Maximum number of keys per node =M-1
* Minimum Number of keys per node= ceil (M/2)-1
* Maximum number of pointers / children per node=M
* Minimum number of pointers / children per node=ceil (M/2)

Steps for insertion in B+ Tree

1. Every element is inserted into a leaf node. So, go to the appropriate leaf node.
2. Insert the key into the leaf node in increasing order only if there is no overflow. If there is an overflow goes ahead with the following steps mentioned below to deal with overflow while maintaining the B+ Tree properties.

**Properties for insertion B+ Tree Case 1:** Overflow in leaf node

1. Split the leaf node into two nodes.
2. First node contains ceil ((m-1)/2) values.
3. Second node contains the remaining values.
4. Copy the smallest search key value from second node to the parent node.(Right biased)

**Case 2:** Overflow in non-leaf node

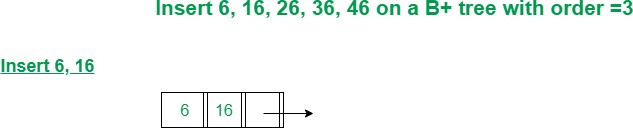
1. Split the non leaf node into two nodes.
2. First node contains ceil (m/2)-1 values.
3. Move the smallest among remaining to the parent.
4. Second node contains the remaining keys.

Example:

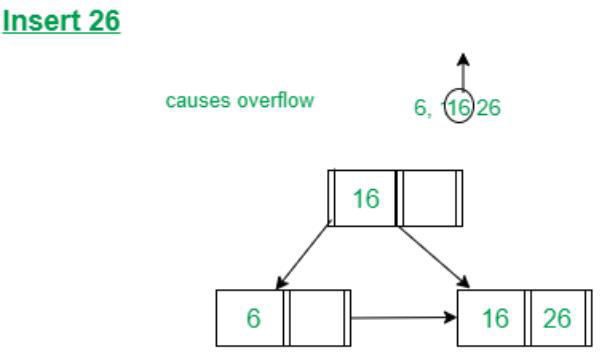
**Problem:** Insert the following key values 6, 16, 26, 36, 46 on a B+ tree with order = 3.

**Solution:**

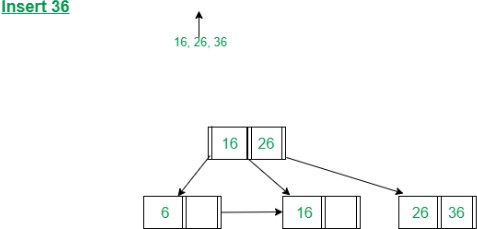
**Step 1:** The order is 3 so at maximum in a node so there can be only 2 search key values. As insertion happens on a leaf node only in a B+ tree so insert search key value **6 and 16** in increasing order in the node. Below is the illustration of the same:



**Step 2:** We cannot insert 26 in the same node as it causes an overflow in the leaf node, We have to split the leaf node according to the rules. First part contains ceil((3-1)/2) values i.e., only 6. The second node contains the remaining values i.e., 16 and 26. Then also copy the smallest search key value from the second node to the parent node i.e., 16 to the parent node. Below is the illustration of the same:



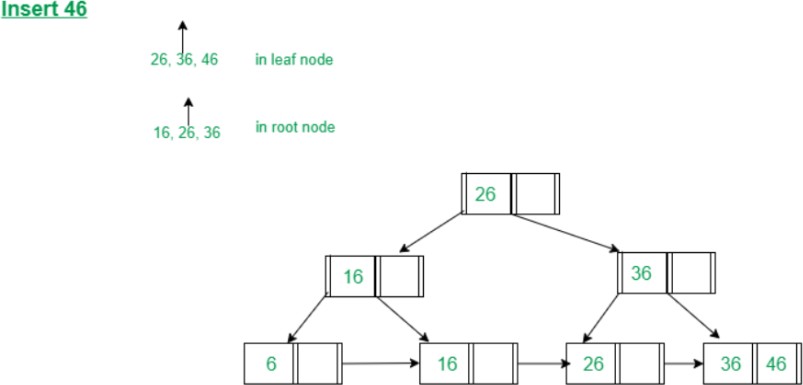
**Step 3:** Now the next value is 36 that is to be inserted after 26 but in that node, it causes an overflow again in that leaf node. Again follow the above steps to split the node. First part contains ceil ((3-1)/2) values i.e., only 16. The second node contains the remaining values i.e., 26 and 36. Then also copy the smallest search key value from the second node to the parent node i.e., 26 to the parent node. Below is the illustration of the same: The illustration is shown in the diagram below.



**Step 4:** Now we have to insert 46 which is to be inserted after 36 but it causes an overflow in the leaf node. So we split the node according to the rules. The first part contains 26 and the second part contains 36 and 46 but now we also have to copy 36 to the parent node but it causes overflow as only two searches key values can be accommodated in a node. Now follow the steps to deal with overflow in the non- leaf node.

First node contains ceil (3/2)-1 values i.e. ’16’.

Move the smallest among remaining to the parent i.e ’26’ will be the new parent node.

The second node contains the remaining keys i.e ’36’ and the rest of the leaf nodes remain the same. Below is the illustration of the same:

Deletion in B+ tree:

Before going through the steps below, one must know these facts about a B+ tree of degree **m**.

1. A node can have a maximum of m children.
2. A node can contain a maximum of m - 1 keys.
3. A node should have a minimum of ⌈m/2⌉ children.
4. A node (except root node) should contain a minimum of ⌈m/2⌉ - 1 keys.

While deleting a key, we have to take care of the keys present in the internal nodes (i.e. indexes) as well because the values are redundant in a B+ tree. Search the key to be deleted then follow the following steps.

Case I

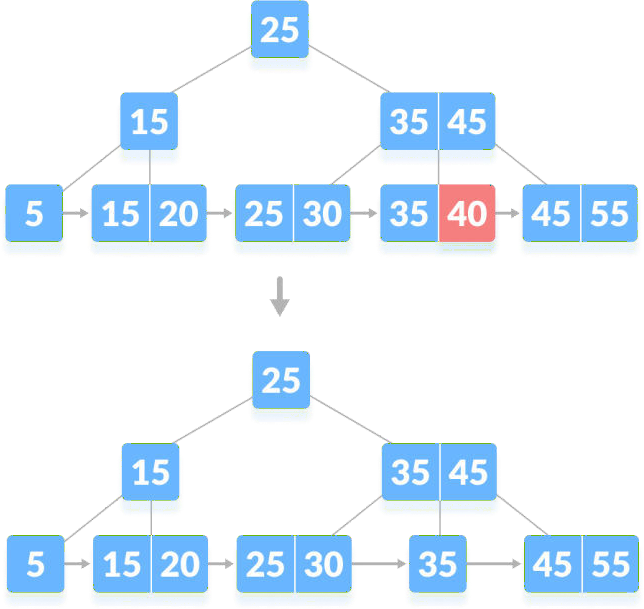
The key to be deleted is present only at the leaf node not in the indexes (or internal nodes). There are two cases for it:

1. There is more than the minimum number of keys in the node. Simply delete the key.

M=3,

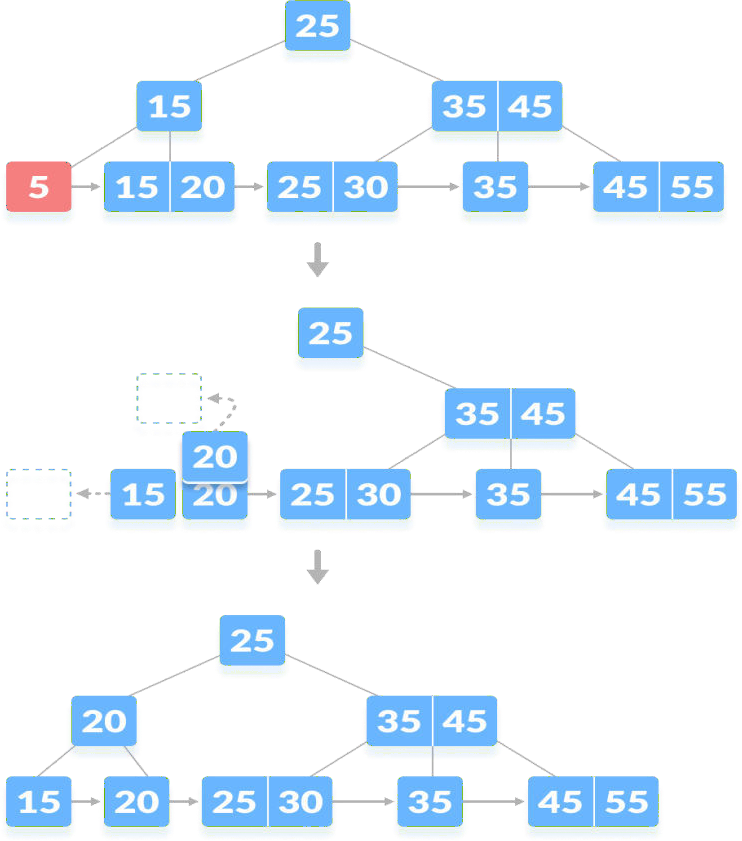
Max. Children=3

Min. Children=ceil (3/2)=2 Max.Keys=m-1=2 Min.keys=ceil (3/2)-1=1



Deleting 40 from B-tree

1. There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling. Add the median key of the sibling node to the parent.

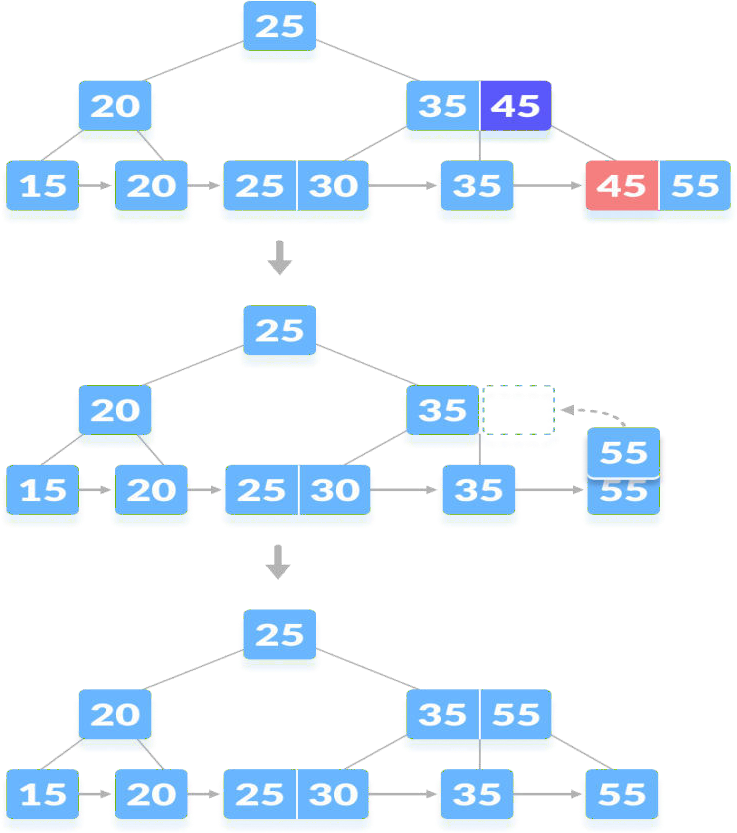


Deleting 5 from B-tree

Case II

The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well. There are the following cases for this situation.

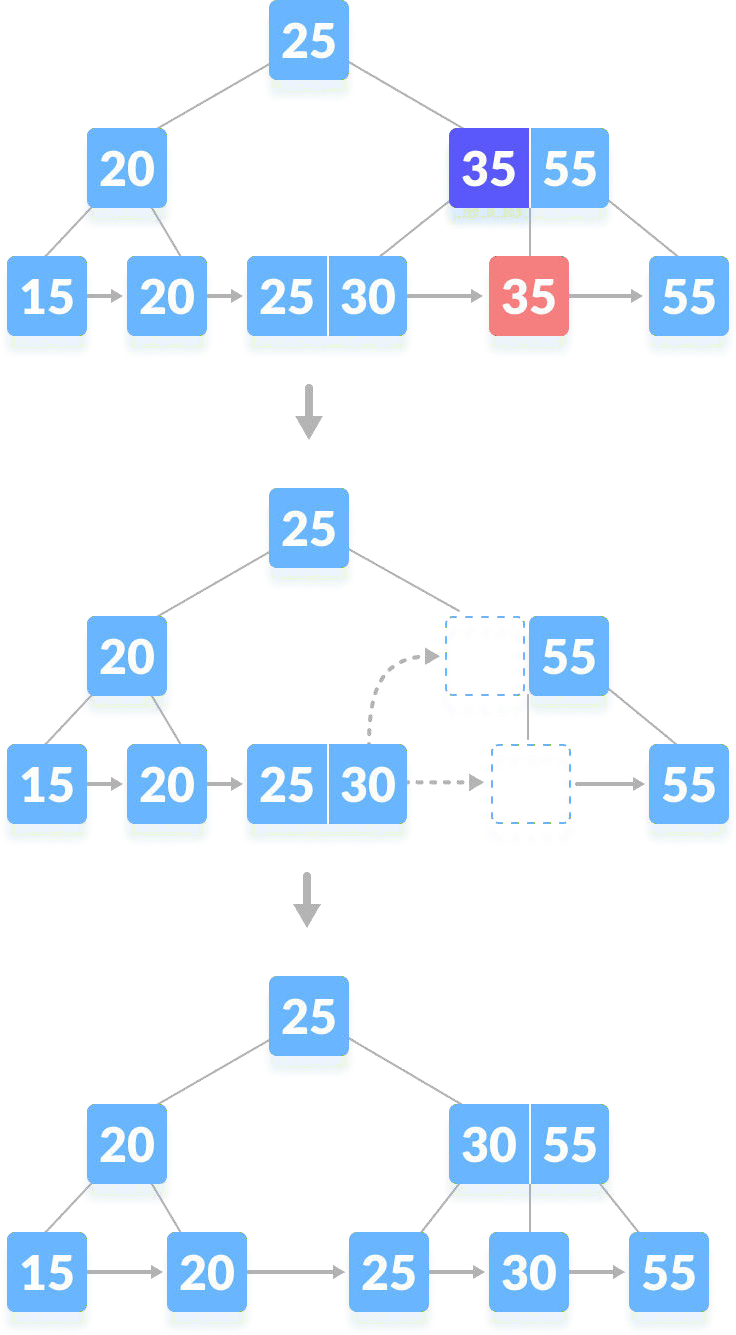
1. If there is more than the minimum number of keys in the node, simply delete the key from the leaf node and delete the key from the internal node as well. Fill the empty space in the internal node with the inorder successor.



Deleting 45

1. If there are an exact minimum number of keys in the node, then delete the key and borrow a key from its immediate sibling (through the parent).

Fill the empty space created in the index (internal node) with the borrowed key.

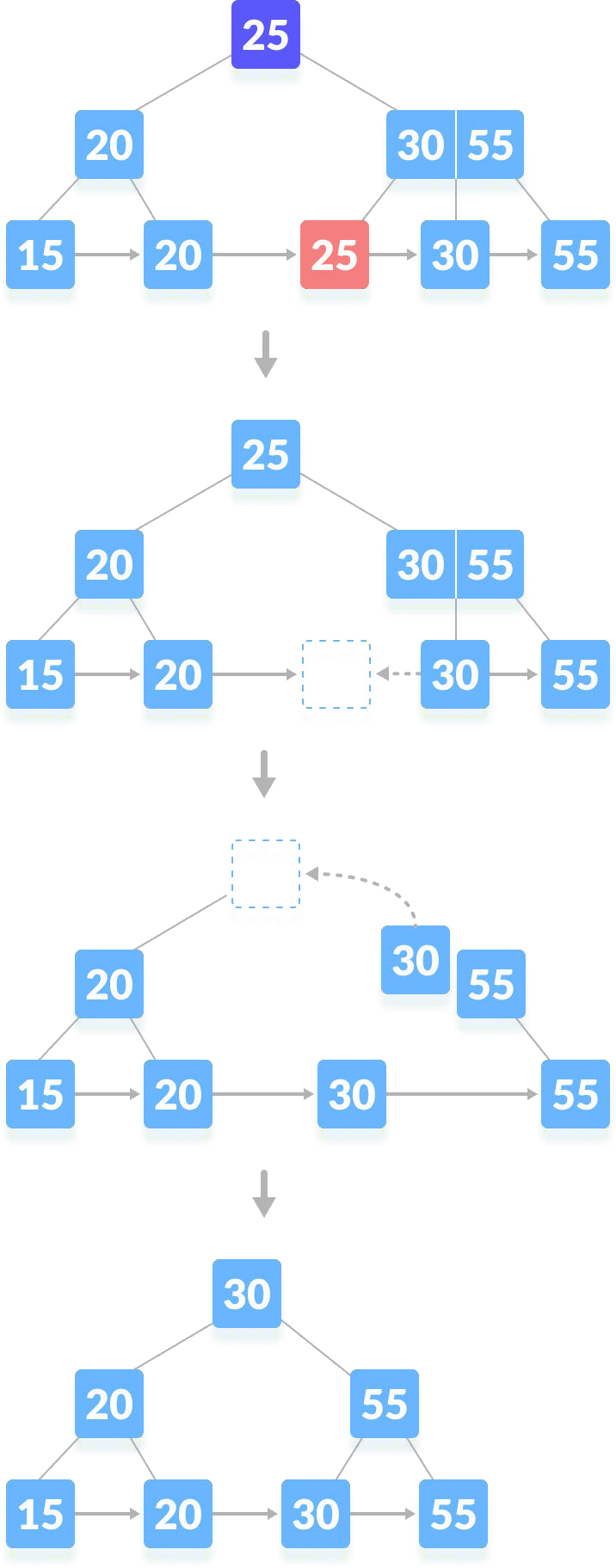


Deleting 35

1. This case is similar to Case II(1) but here, empty space is generated above the immediate parent node.

After deleting the key, merge the empty space with its sibling.

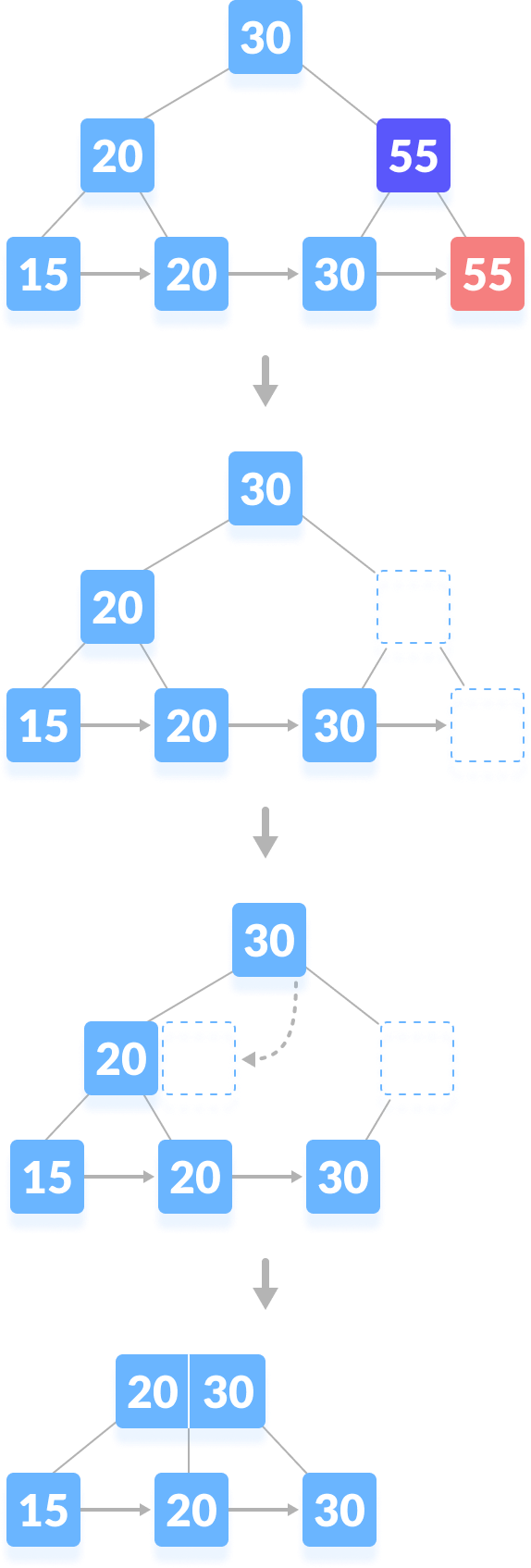
Fill the empty space in the grandparent node with the inorder successor.



Case III

In this case, the height of the tree gets shrinked. It is a little complicated.Deleting

55 from the tree below leads to this condition. It can be understood in the illustrations below.

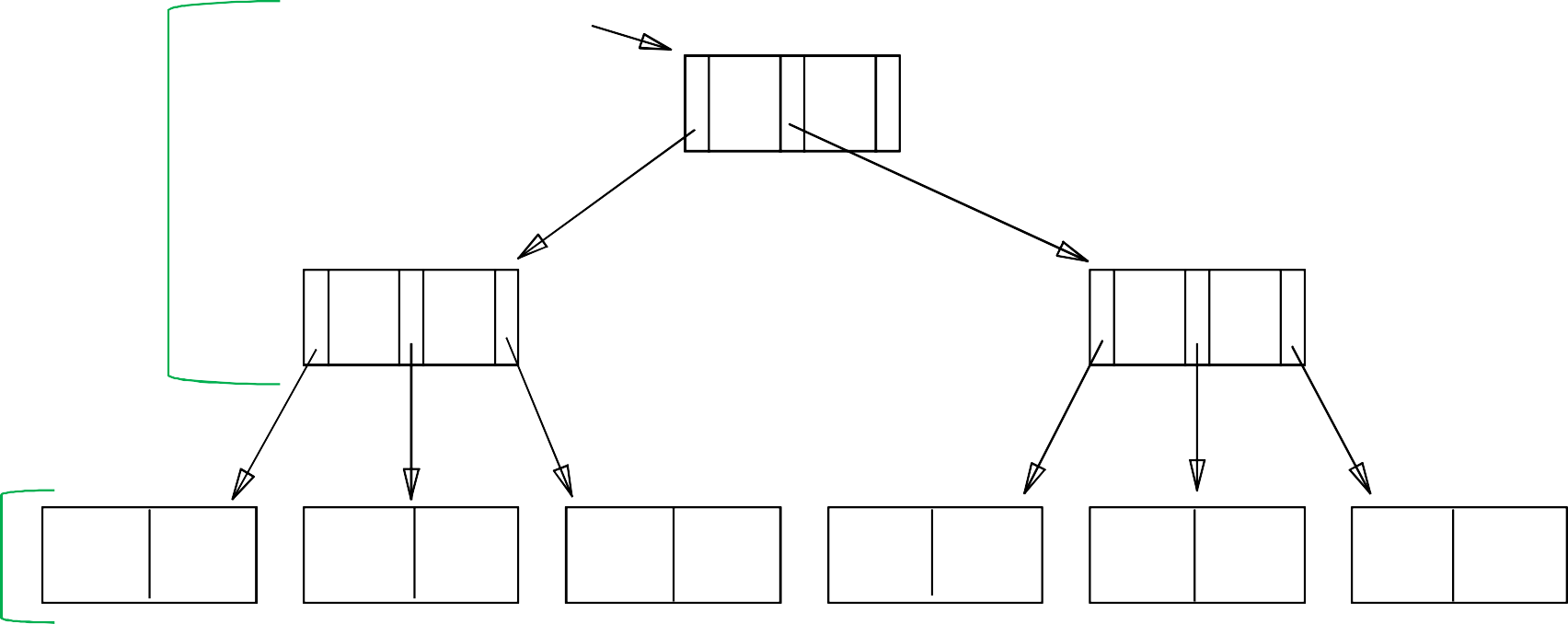


# ISAM Trees

## Indexed Sequential Access Method (ISAM) trees are *static*

**Leaf Pages**

E.g., 2 Entries Per Page



Root

40

**Non-Leaf**

**Pages**

20

33

51

63

10\*

15\*

20\*

27\*

33\*

37\*

40\*

46\*

51\*

55\*

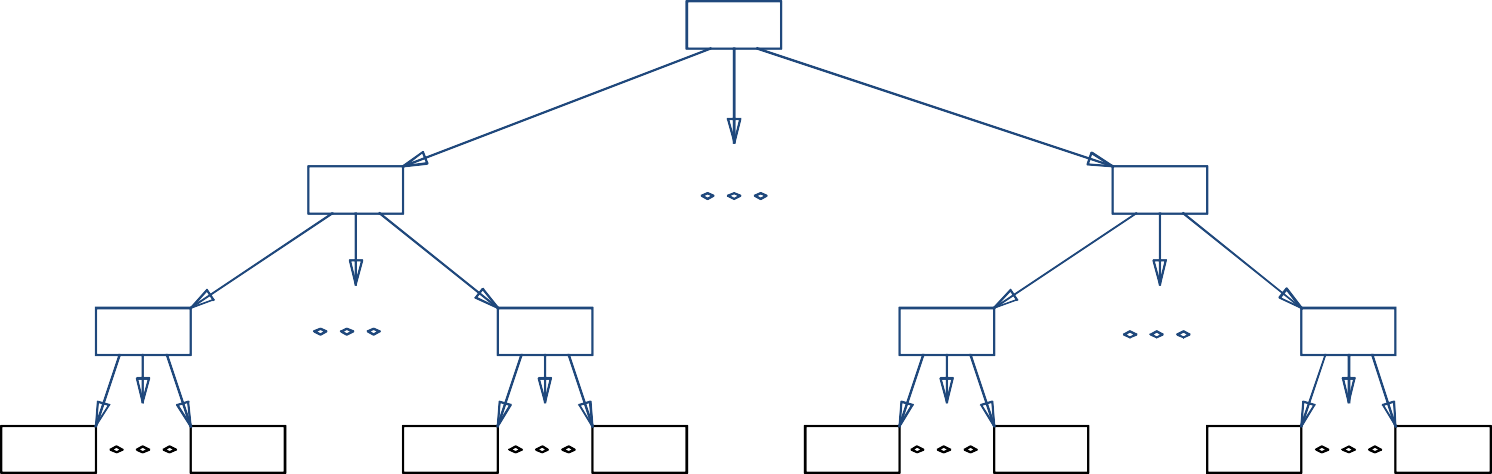
63\*

97\*

# ISAM Trees: Page Overflows

## What if there are a lot of insertions after creating the tree?

Non-leaf Pages



Leaf Pages

Overflow

page Primary pages

# ISAM File Creation

* How to create an ISAM file?
  + All leaf pages are allocated *sequentially* and

*sorted* on the search key value

* + If Alternative (2) or (3) is used, the data records are created and *sorted* before allocating

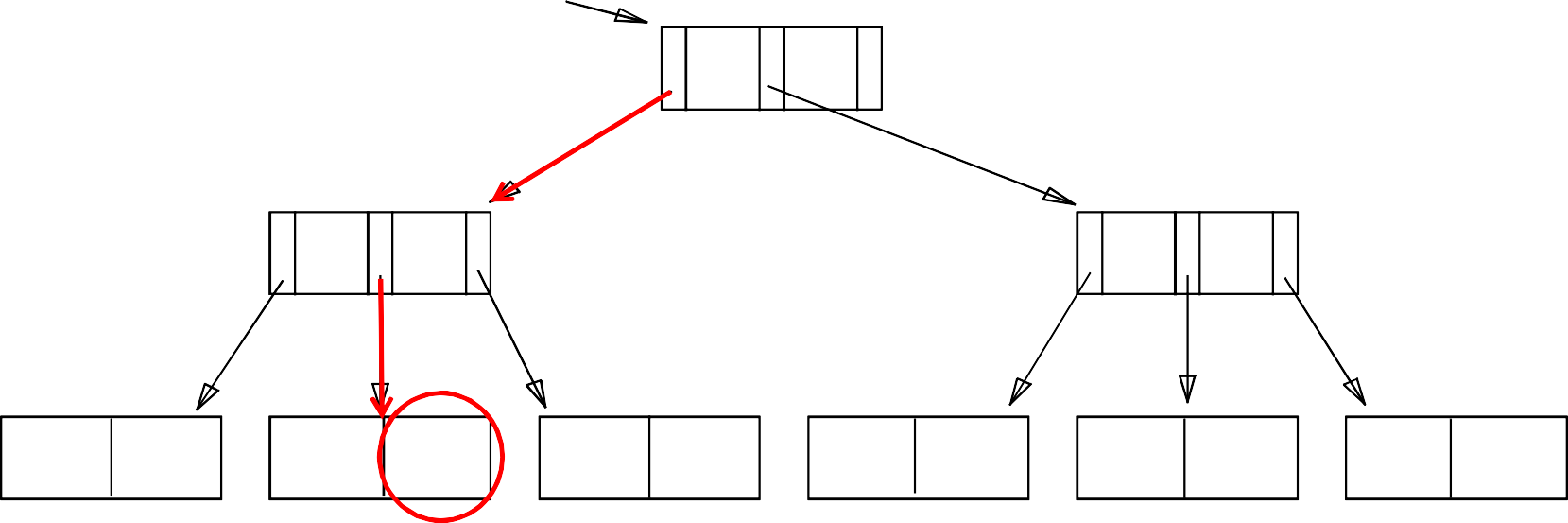
leaf pages

* + The non-leaf pages are subsequently allocated

# ISAM: Searching for Entries

## Search begins at root, and key comparisons direct it to a leaf

* Search for 27\*



**Root**

**40**

**20**

**33**

**51**

**63**

**10\***

**15\***

**20\***

**27\***

**33\***

**37\***

**40\***

**46\***

**51\***

**55\***

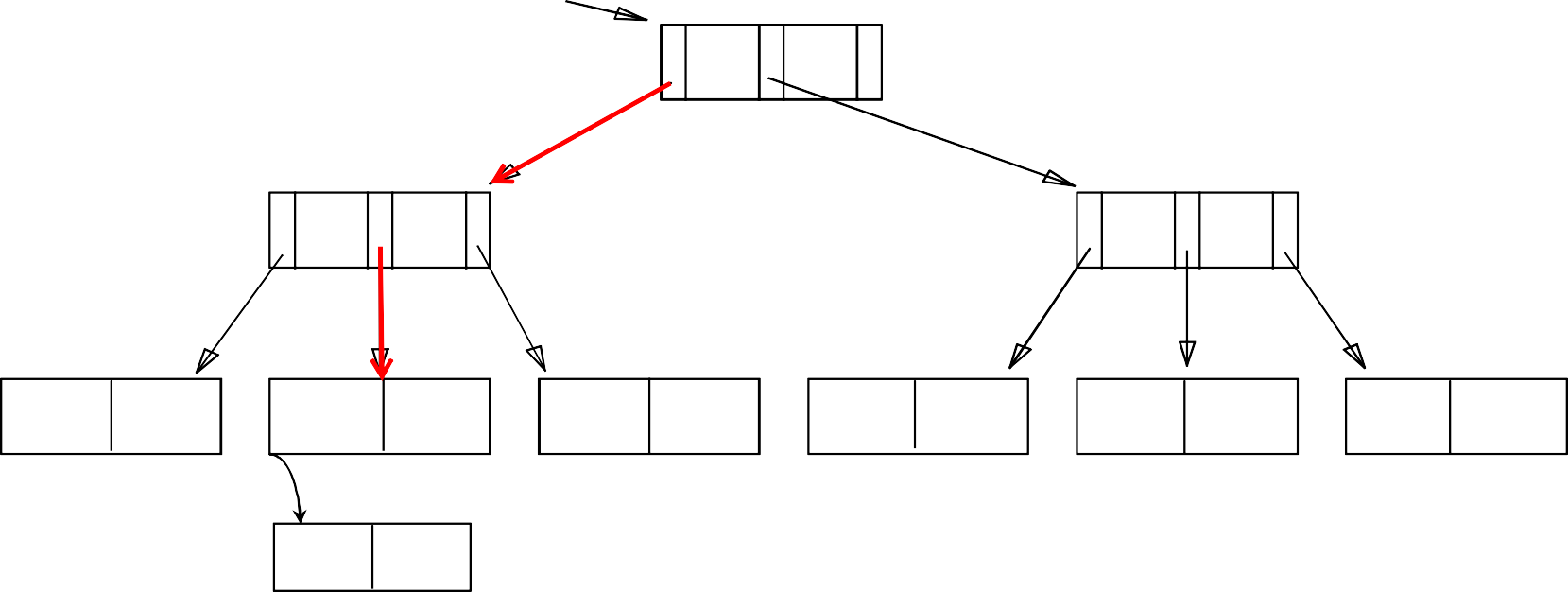
**63\***

**97\***

# ISAM: Inserting Entries

### The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

* Insert 23\*



**Root**

**40**

**20**

**33**

**51**

**63**

**10\***

**15\***

**20\***

**27\***

**33\***

**37\***

**40\***

**46\***

**51\***

**55\***

**63\***

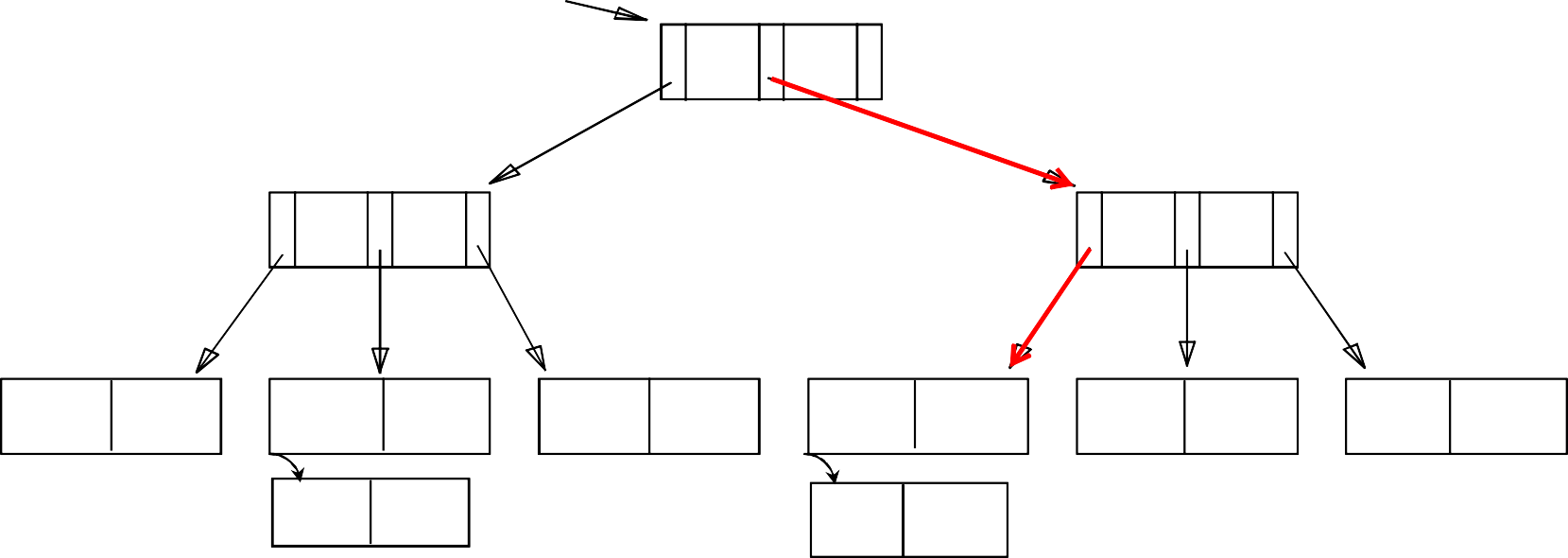
**97\***

**23\***

# ISAM: Inserting Entries

### The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

* Insert 48\*



**Root**

**40**

**20**

**33**

**51**

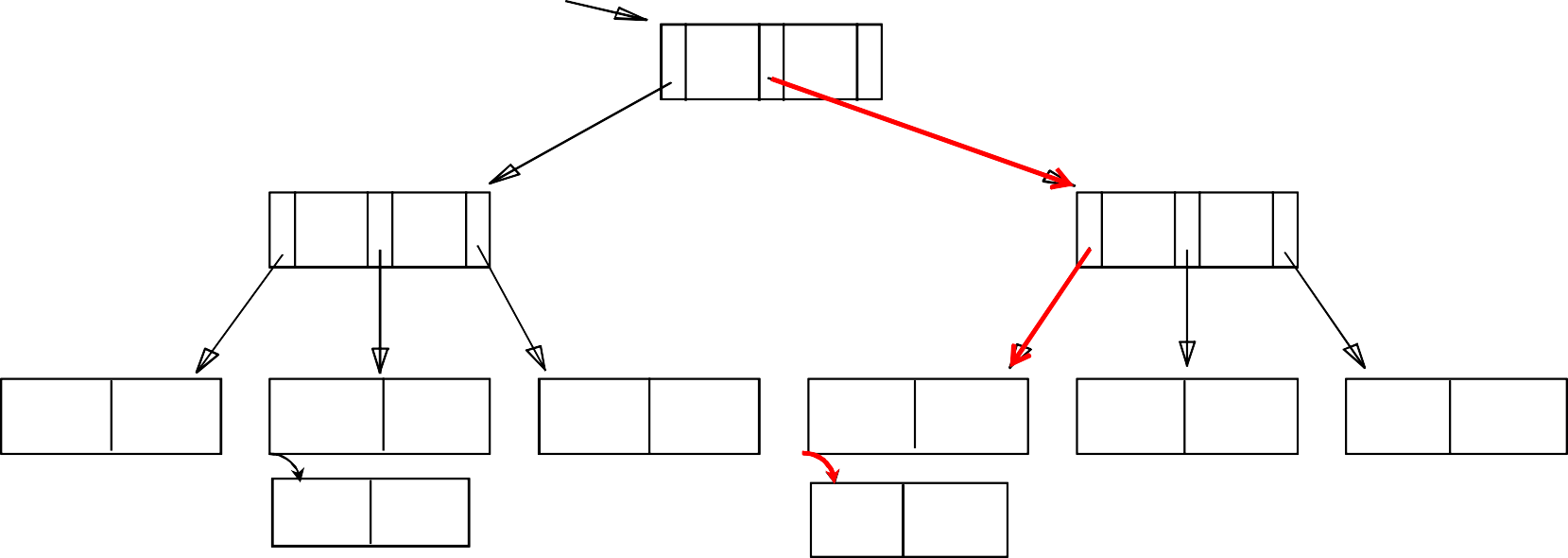
**63**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **10\*** | **15\*** | **20\*** | **27\*** | **33\*** | **37\*** | **40\*** | **46\*** | **51\*** | **55\*** | **63\*** | **97\*** |
|  |  | **23\*** |  |  |  | **48\*** |  |  |  |  |  |

# ISAM: Inserting Entries

### The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

* Insert 41\*



**Root**

**40**

**20**

**33**

**51**

**63**

**10\***

**15\***

**20\***

**27\***

**33\***

**37\***

**40\***

**46\***

**51\***

**55\***

**63\***

**97\***

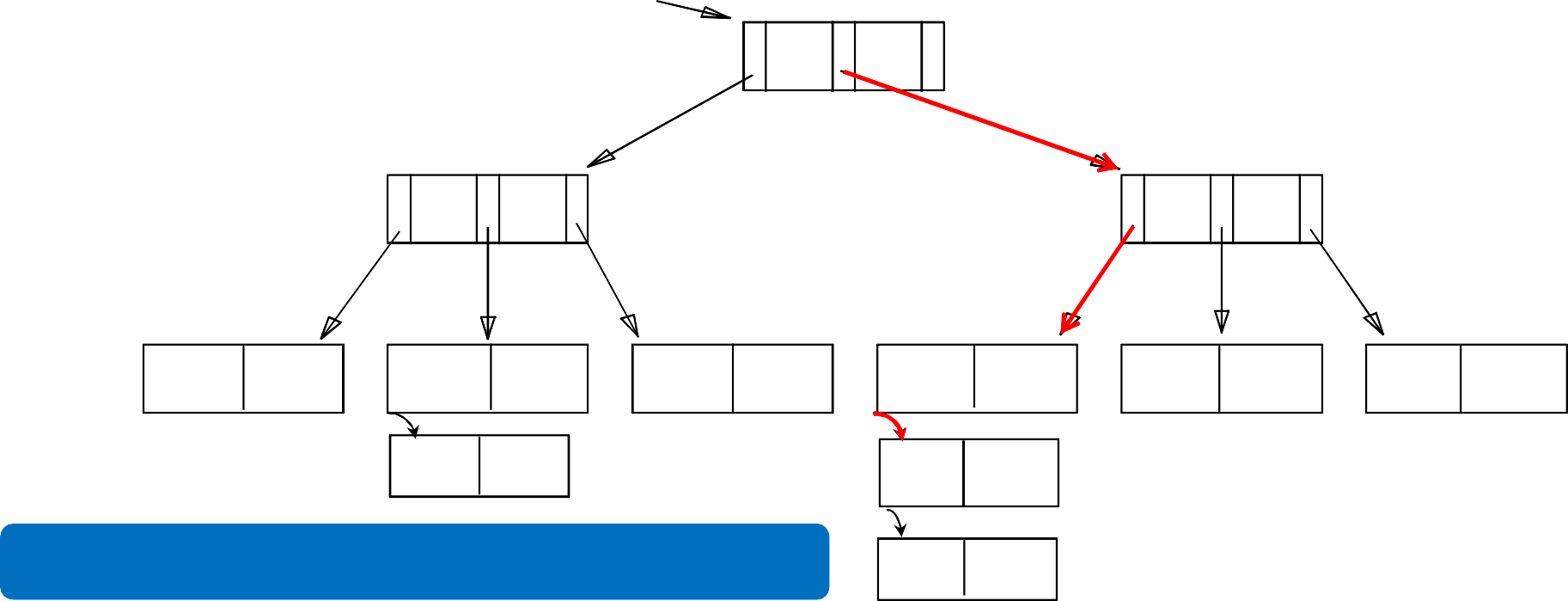
**23\***

**48\*** **41\***

# ISAM: Inserting Entries

### The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

* Insert 42\*



**Root**

**40**

**20**

**33**

**51**

**63**

**10\***

**15\***

**20\***

**27\***

**33\***

**37\***

**40\***

**46\***

**51\***

**55\***

**63\***

**97\***

**23\***

**48\*** **41\***

Chains of overflow pages can easily develop2!64

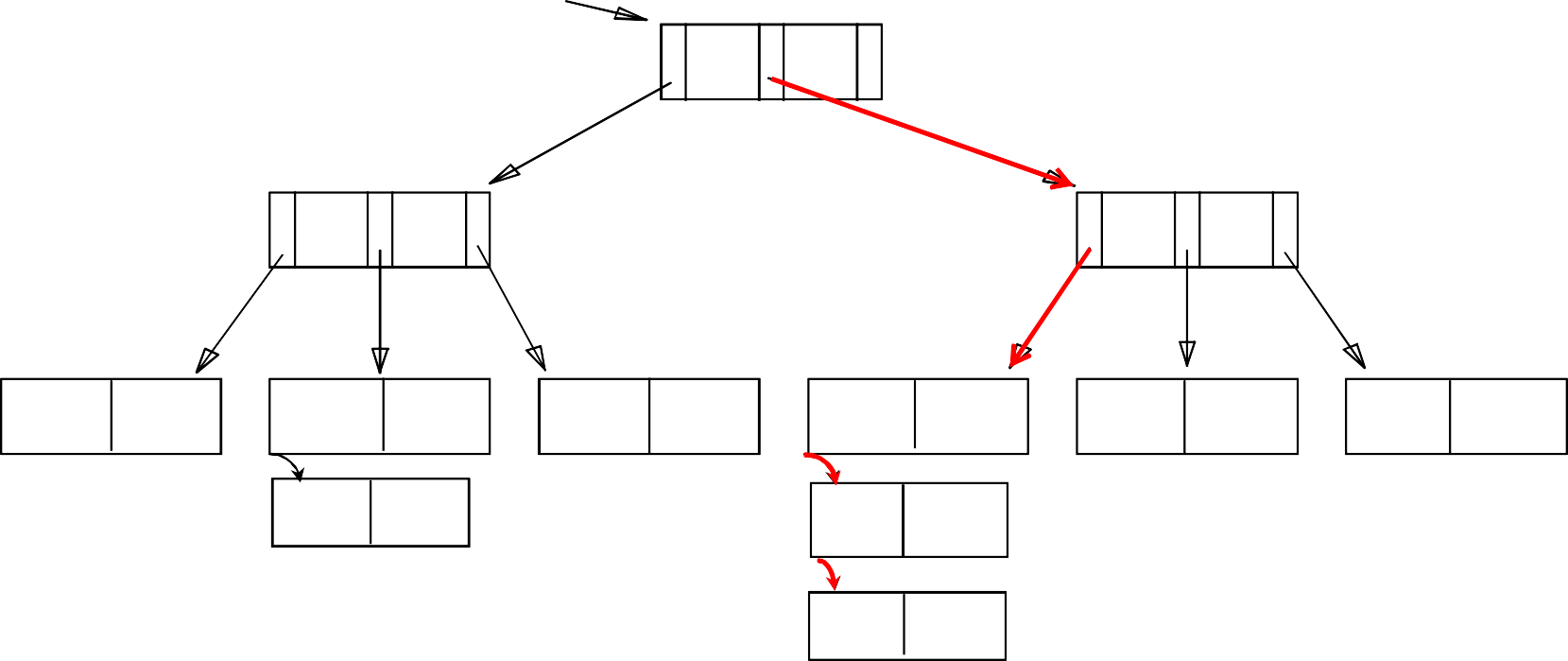
**42\***

# ISAM: Deleting Entries

* The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

### Delete 42\*

Root

**40**

**20** **33** **51** **63**

**10\*** **15\*** **20\*** **27\*** **33\*** **37\*** **40\*** **46\*** **51\*** **55\*** **63\*** **97\***

**23\*** **48\*** **41\***

265

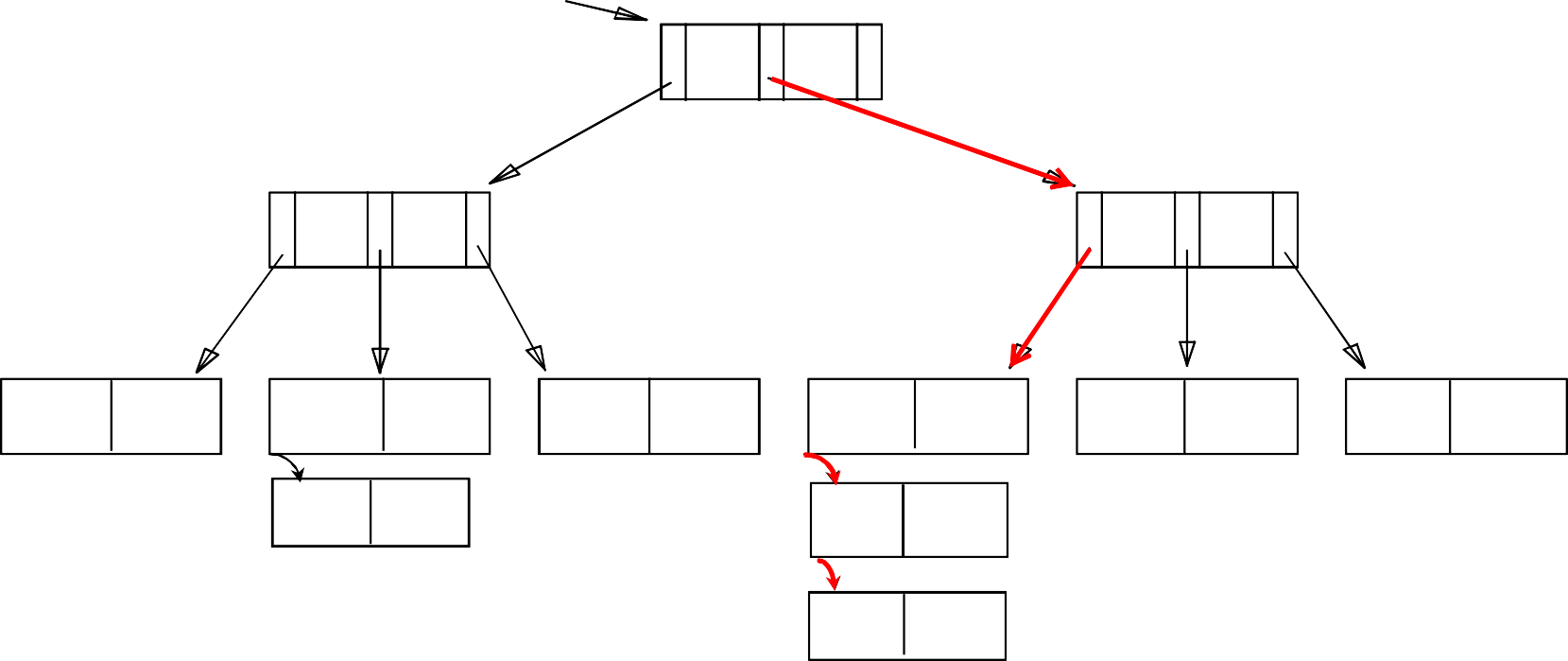
**42\***

# ISAM: Deleting Entries

* The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

### Delete 42\*

Root

**40**

**20** **33** **51** **63**

**10\*** **15\*** **20\*** **27\*** **33\*** **37\*** **40\*** **46\*** **51\*** **55\*** **63\*** **97\***

**23\*** **48\*** **41\***

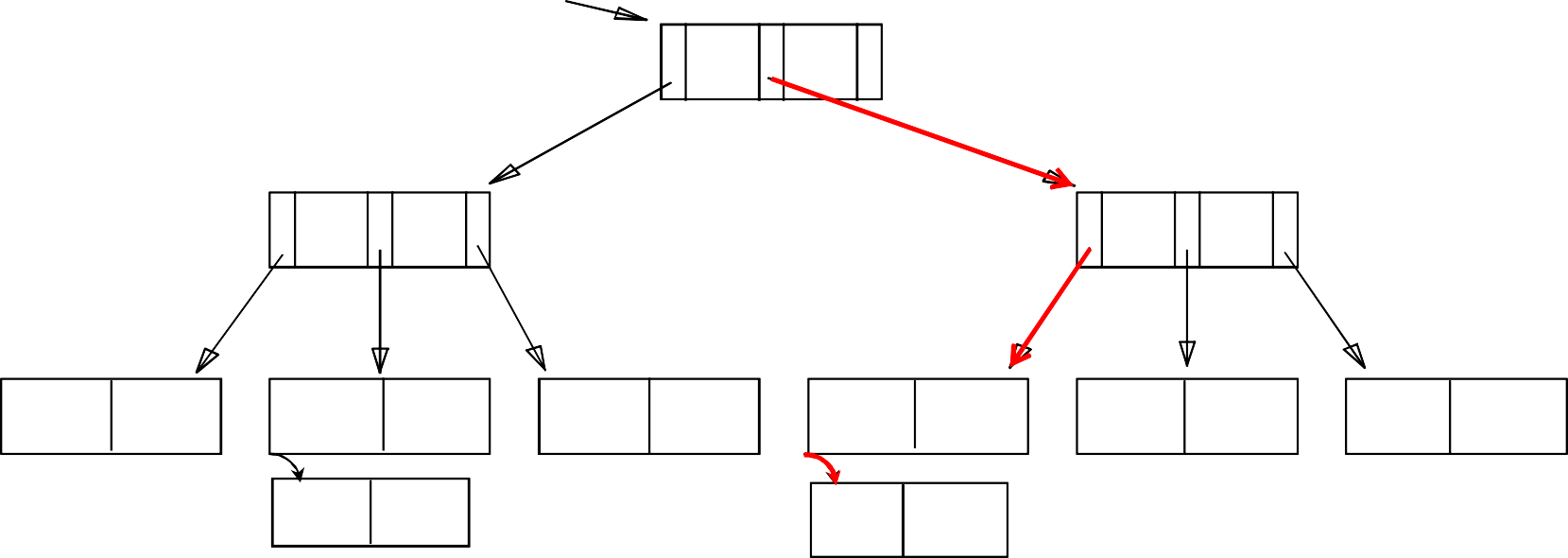
266

# ISAM: Deleting Entries

* The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

### Delete 42\*

Root

**40**

**20** **33** **51** **63**

**10\*** **15\*** **20\*** **27\*** **33\*** **37\*** **40\*** **46\*** **51\*** **55\*** **63\*** **97\***

**23\*** **48\*** **41\***

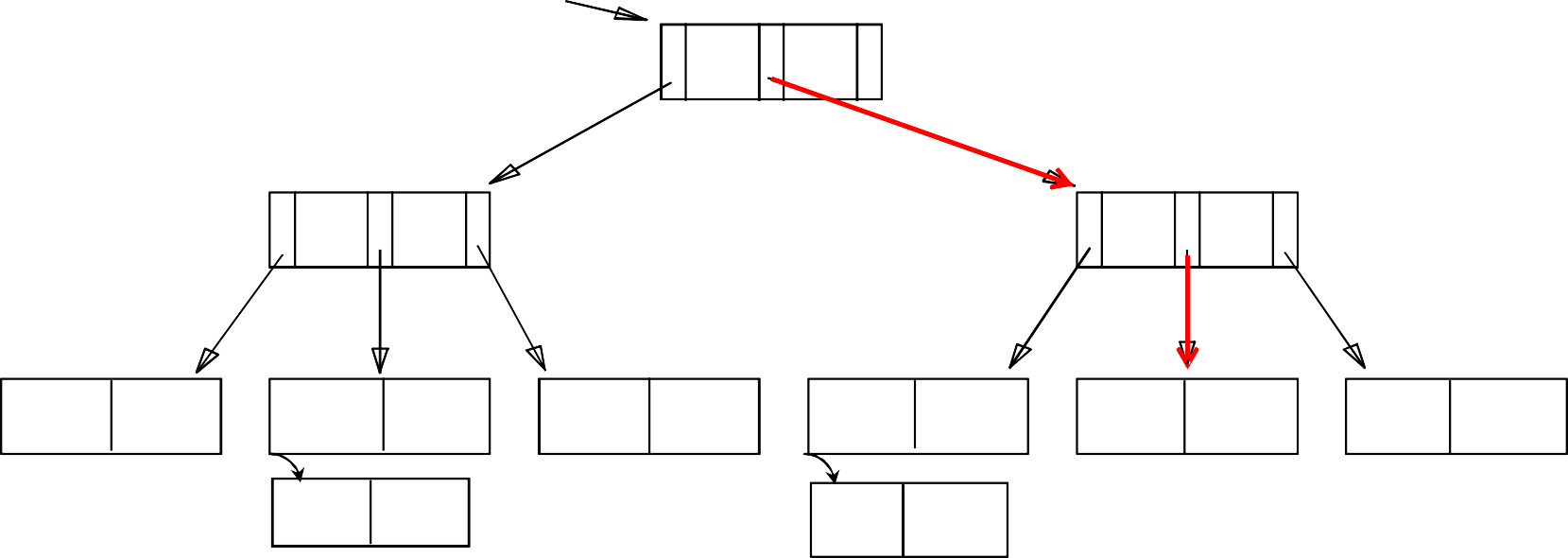
267

# ISAM: Deleting Entries

* The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

### Delete 51\*

Root

**40**

**20** **33** **51** **63**

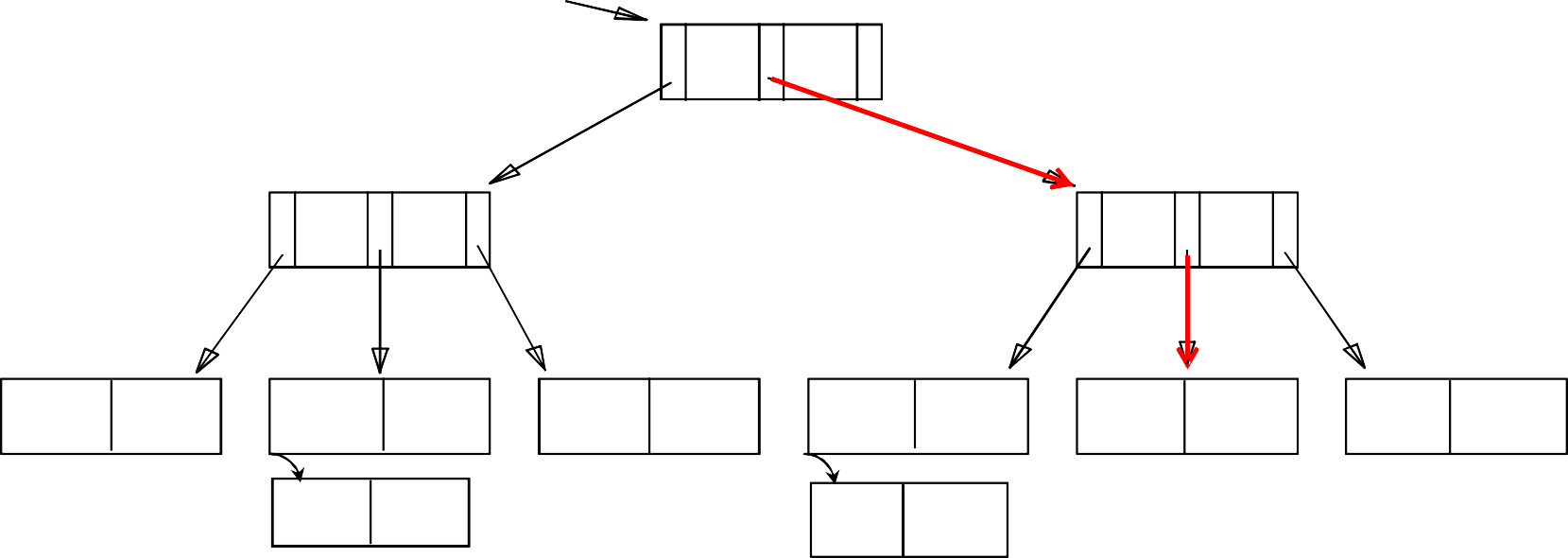
**10\*** **15\*** **20\*** **27\*** **33\*** **37\*** **40\*** **46\*** **51\*** **55\*** **63\*** **97\***

# ISAM: Deleting Entries

* The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

### Delete 55\*

Root

**40**

**20** **33** **51** **63**

**10\*** **15\*** **20\*** **27\*** **33\*** **37\*** **40\*** **46\*** **55\*** **63\*** **97\***

# ISAM: Some Issues

* Once an ISAM file is created, insertions and deletions affect only the contents of leaf pages (i.e., *ISAM is a static structure*!)

#### Since index-level pages are *never* modified, there is no need to

*lock* them during insertions/deletions

* + Critical for concurrency!

#### Long overflow chains can develop easily

* + The tree can be initially set so that ~20% of each page is free

#### If the data distribution and size are relatively static, ISAM might be a good choice to pursue!

270

File Organization

* The File is a collection of records. Using the primary key, we can access the records. The type and frequency of access can be determined by the type of file organization which was used for a given set of records.
* File organization is a logical relationship among various records. This method defines how file records are mapped onto disk blocks.

Types of file organization:

File organization contains various methods. These particular methods have pros and cons on the basis of access or selection. In the file organization, the programmer decides the best-suited file organization method according to his requirement.

Types of file organization are as follows:

1. [Sequential file organization](https://www.javatpoint.com/dbms-sequential-file-organization)
2. [Heap file organization](https://www.javatpoint.com/dbms-heap-file-organization)
3. [Hash file organization](https://www.javatpoint.com/dbms-hash-file-organization)
4. [Cluster file organization](https://www.javatpoint.com/dbms-cluster-file-organization)
5. B+ Tree File Organization

[Sequential file organization](https://www.javatpoint.com/dbms-sequential-file-organization)

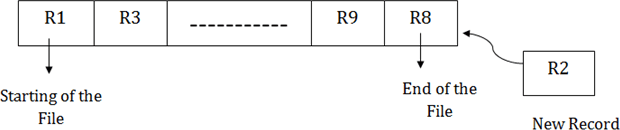
This method is the easiest method for file organization. In this method, files are stored sequentially. This method can be implemented in two ways:

1. Pile File Method:
   * It is a quite simple method. In this method, we store the record in a sequence, i.e., one after another. Here, the record will be inserted in the order in which they are inserted into tables.
   * In case of updating or deleting of any record, the record will be searched in the memory blocks. When it is found, then it will be marked for deleting, and the new record is inserted.



Insertion of the new record:

Suppose we have four records R1, R3 and so on up to R9 and R8 in a sequence. Hence, records are nothing but a row in the table. Suppose we want to insert a new record R2 in the sequence, then it will be placed at the end of the file. Here, records are nothing but a row in any table.

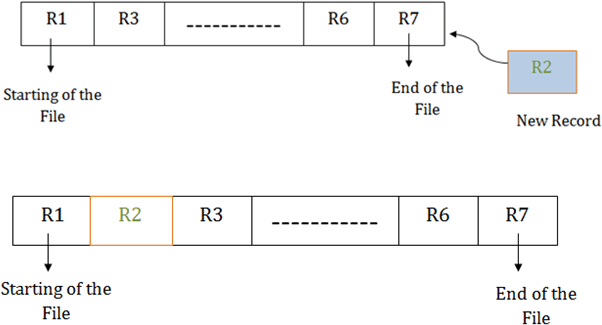


1. Sorted File Method:
   * In this method, the new record is always inserted at the file's end, and then it will sort the sequence in ascending or descending order. Sorting of records is based on any primary key or any other key.
   * In the case of modification of any record, it will update the record and then sort the file, and lastly, the updated record is placed in the right place.



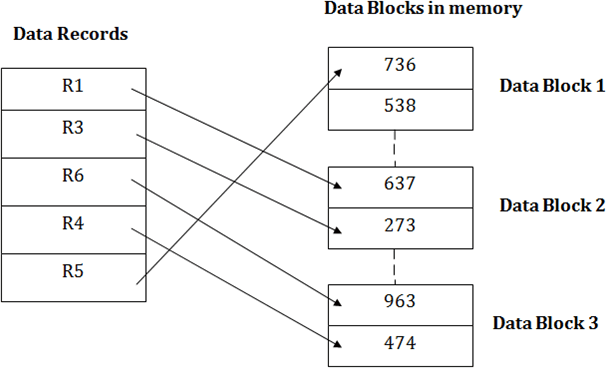
Insertion of the new record:

Suppose there is a preexisting sorted sequence of four records R1, R3 and so on upto R6 and R7. Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file, and then it will sort the sequence.



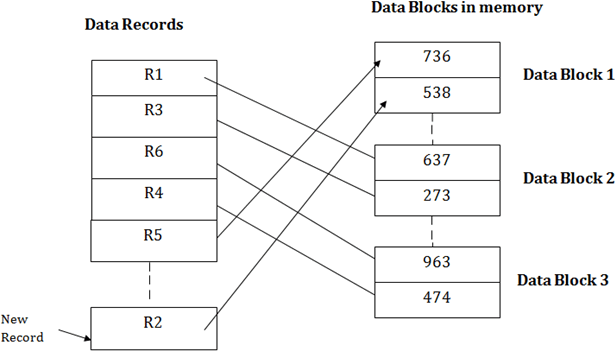
Heap file organization

* + It is the simplest and most basic type of organization. It works with data blocks. In heap file organization, the records are inserted at the file's end. When the records are inserted, it doesn't require the sorting and ordering of records.
  + When the data block is full, the new record is stored in some other block. This new data block need not to be the very next data block, but it can select any data block in the memory to store new records. The heap file is also known as an unordered file.
  + In the file, every record has a unique id, and every page in a file is of the same size. It is the DBMS responsibility to store and manage the new records.



Insertion of a new record

Suppose we have five records R1, R3, R6, R4 and R5 in a heap and suppose we want to insert a new record R2 in a heap. If the data block 3 is full then it will be inserted in any of the database selected by the DBMS, let's say data block 1.



If we want to search, update or delete the data in heap file organization, then we need to traverse the data from staring of the file till we get the requested record.

If the database is very large then searching, updating or deleting of record will be time-consuming because there is no sorting or ordering of records. In the heap file organization, we need to check all the data until we get the requested record.

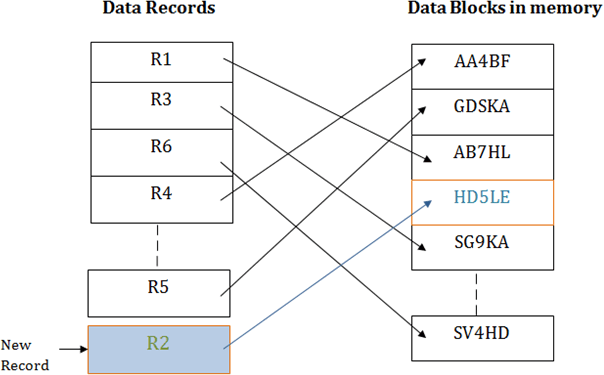
Hash File Organization

Hash File Organization uses the computation of hash function on some fields of the records. The hash function's output determines the location of disk block where the records are to be placed.



When a record has to be received using the hash key columns, then the address is generated, and the whole record is retrieved using that address. In the same way, when a new record has to be inserted, then the address is generated using the hash key and record is directly inserted. The same process is applied in the case of delete and update.

In this method, there is no effort for searching and sorting the entire file. In this method, each record will be stored randomly in the memory.

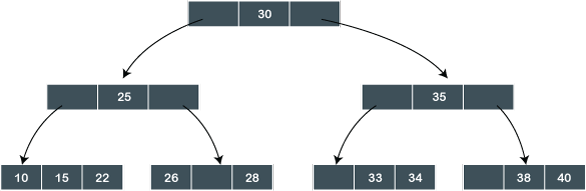


B+ Tree File Organization

B+ Tree is a data structure, which uses a tree-like structure for storing and accessing the records or data from the file.

It is an enhanced method of an ISAM (Indexed Sequential Access Method). This file organization stores that data which is not fit in the system’s main memory.

This file organization uses the concept of key-index. This concept uses the primary key for the sorting of records. An index value of the database record is the record address of the file.

B+ tree is the same as a binary search tree. But, this type of tree can have more than two children. In this type of file organization, all the records or information are stored at the leaf nodes. And the intermediate nodes act as the pointer to the nodes which store the records, i.e., leaf nodes. Intermediate nodes in the tree do not contain any information. Following diagram shows how the values are stored in the B+ Tree File organization:

In this B+ tree, 30 is the only root node of the tree, which is also known as the main node of the B+ tree. There exists an intermediary layer with the nodes, which stores the address of the leaf nodes, not the actual records. Only the leaf nodes contain the records in the order which is sorted.

In the above B+ tree, only one leaf node exists whose values are: 10, 15, 22, 26, 28, 33, 34, 38, 40. As all the leaf nodes of the tree are sorted, so the records can be easily searched.

Clustered File Organization

Cluster is defined as “when two or more related tables or records are stored within the same file”. The related column of two or more database tables in the cluster is called the cluster key. And this cluster key is used to map the two tables together.

This method minimizes the cost of accessing and searching the various records because they are combined and available in a single cluster.

Example:

Suppose we have two tables whose names are Student and Subject**.** Both of the following given tables are related to each other.

Student

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_ID** | **Student\_Name** | **Student\_Age** | **Subject\_ID** |
| 101 | Raju | 20 | C01 |
| 102 | Ravi | 20 | C04 |
| 103 | Rajesh | 21 | C01 |
| 104 | Ranjith | 22 | C02 |
| 105 | Rahul | 21 | C03 |
| 106 | Ramesh | 20 | C04 |
| 107 | Ravinder | 20 | C03 |
| 108 | Rudra | 21 | C04 |

**Subject**

|  |  |
| --- | --- |
| **Subject\_ID** | **Subject\_Name** |
| C01 | Math |
| C02 | Java |
| C03 | C |
| C04 | DBMS |

Therefore, both these tables ‘student’ and ‘subject’ are allowed to combine using a join operation and can be seen as following in the cluster file.

Student + Subject

**Cluster Key**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Subject\_ID** | **Subject\_Name** | **Student\_ID** | **Student\_Name** | **Student\_Age** |
| C01 | Math | 101 | Raju | 20 |
|  |  | 103 | Ravi | 21 |
| C02 | Java | 104 | Rajesh | 22 |
| C03 | C | 105 | Ranjith | 21 |
|  |  | 107 | Rahul | 20 |
| C04 | DBMS | 102 | Ramesh | 20 |
|  |  | 106 | Ravinder | 20 |
|  |  | 108 | Rudra | 21 |

If we have to perform the insert, update and delete operations on the record, then we can perform them directly because the data is sorted on that key with which searching and accessing is done. In the given table (Student + Subject), the cluster key is a Subject\_ID**.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Sequential** | **Heap/Direct** | **Hash** | **ISAM** | **B+ tree** | **Cluster** |
| **Method of storing** | Stored as they come or sorted as they come | Stored at the end of the file. But the address in the memory is random. | Stored at the hash address generated | Address index is appended to the record | Stored in a tree like structure | Frequently joined tables are clubbed into one file based on cluster  key |
| **Types** | Pile file and  sorted file Method |  | Static and  dynamic hashing | Dense, Sparse, multilevel indexing |  | Indexed and Hash |
| **Design** | Simple  Design | Simplest | Medium | Complex | Complex | Simple |
| **Storage Cost** | Cheap (magnetic  tapes) | Cheap | Medium | Costlier | Costlier | Medium |
| **Advantage** | Fast and efficient when there is large volumes of | Best suited for bulk insertion, and small files/tables | Faster Access No Need to Sort  Handles | Searching records is faster.  Suitable for large database. | Searching range of data & partial data are | Best suited for frequently joined |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | data, Report generation, statistical calculations etc |  | multiple transactions Suitable for Online transactions | Any of the columns can be used as key column. Searching range of data & partial data are efficient. | efficient.  No performance degrades when there is insert / delete  / update. Grows and shrinks with data.  Works well in secondary storage devices and hence reducing disk I/O.  Since all datas are at the leaf node, searching is easy.  All data at leaf node are sorted sequential linked list. | tables. Suitable for 1:M  mappings |
| **Disadvantage** | Sorting of data each time for insert/delete/ update takes time and makes system slow | Records are scattered in the memory and they are inefficiently used. Hence increases the memory size.  Proper memory management is needed.  Not suitable | Accidental Deletion or updation of Data  Use of Memory is inefficient Searching range of data, partial data, non-hash key column, searching single hash column when | Extra cost to maintain index.  File reconstruction is needed as insert/update/delete. Does not grow with data. | Not suitable for static tables | Not suitable for large database. Suitable only for the joins on which clustering is done.  Less frequently used joins and 1: 1 Mapping |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | for large tables. | multiple hash keys present or frequently updated column as hash key are  inefficient. |  |  | are inefficient. |

Indexes and Performance Tuning

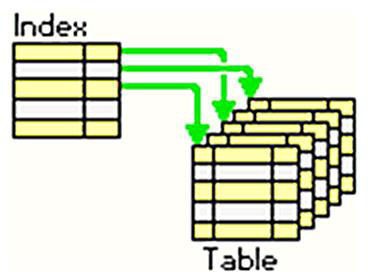
An index is a copy of selected columns of data from a table that can be searched very efficiently. Although indexing does add some overheads in the form of additional writes and storage space to maintain the index data structure, the key focus of implementing index – in various available ways – is to improve the lookup mechanism.

It must improve the performance of data matching by reducing the time taken to match the query value.

Now, let’s understand how index is actually programmed and stored and what causes the speed when an index is created.

Index entries are also "rows", containing the indexed column(s) and some sort of pointing / marking data into the base table data. When an index is used to fetch a row, the index is walked until it finds the row(s) of interest, and the base table is then looked up to fetch the actual row data.

When a data is inserted, a corresponding row is written to the index, and when a row is deleted, its index row is taken out. This keeps the data and searching index always in sync making the lookup very fast and read-time efficient.



[Performance tuning](https://www.quest.com/community/blogs/b/database-management/posts/improve-sql-server-performance-tuning-with-these-3-tips) is the process of making SQL statements run as smoothly as possible by finding and taking the quickest route to answer a query

*Database Management Systems*

Indexing is implemented using specific architecture and approaches. Some architectures worth mentioning are:

create table test(roll int,name char(50),job char(50)); insert into test values(2,'raju','software');

insert into test values(1,'ravi','teacher'); insert into test values(3,'raju','software'); delete from test where roll=2;

select \* from test;

create clustered index roll\_index on test(roll); create nonclustered index job\_index on test(job);

Clustered Index:

A clustered indexed is similar to telephone directory, where data is arranged by first name...

A table can have only one clustered index, However, one clustered index can have multiple columns. Similar to telephone directory is arranged by first name and last name

Non-clustered index:

A nonclustered indexed is similar to the index in textbook, where data is stored at one place and index is stored in another place.

1. The index has pointers to the storage location.
2. Since, Nonclustered index are stored separately, a table can have more than one Nonclustered index
3. In the index itself, data is stored in ascending or descending order of the index

*Database Management Systems*

References:

* + Ramakrishnan - Database Management Systems 3rd Edition
  + Korth Database System Concepts 6th edition
  + Database Systems - Design, Implementation, and Management (9th Edition)
  + Fundamentals-of-Database-Systems-Pearson-2015-Ramez-Elmasri- Shamkant-B.-Navathe
  + An Introduction to Database Systems, 8th Edition, C J Date
  + [www.guru99.com](http://www.guru99.com/)
  + https://[www.programiz.com/](http://www.programiz.com/)
  + [www.javatpoint.com](http://www.javatpoint.com/)
  + [www.geeksforgeeks.org](http://www.geeksforgeeks.org/)
  + [www.tutorialspoint.com](http://www.tutorialspoint.com/)
  + searchsqlserver.techtarget.com
  + [https://beginnersbook.com](https://beginnersbook.com/)
  + [https://www.gatevidyalay.com](https://www.gatevidyalay.com/)
  + [Gate Smashers](https://www.youtube.com/channel/UCJihyK0A38SZ6SdJirEdIOw)
  + [Education 4u](https://www.youtube.com/channel/UCKS34cSMNaXaySe2xgXH-3A)
  + [Jenny's lectures](https://www.youtube.com/channel/UCM-yUTYGmrNvKOCcAl21g3w)
  + [KNOWLEDGE GATE](https://www.youtube.com/channel/UCA6yfpYhy5sWMjRGOT-OAIQ)
  + [Ravindrababu Ravula](https://www.youtube.com/watch?v=1057YmExS-I&list=PLEbnTDJUr_Ic_9b4PcKmlae41cyxEefot)

*Database Management Systems*