MEPL Lucknow

Technical Preparation For IBPS & SBI SO

DBMS

SHOBHIT BANDHU IT_Engineer,MEPL

DBMS Quick Guide

DBMS - Overview

Database is collection of data which is related by some aspect. Data is collection of facts and figures which can be processed to produce information. Name of a student, age, class and her subjects can be counted as data for recording purposes.

Mostly data represents recordable facts. Data aids in producing information which is based on facts. For example, if we have data about marks obtained by all students, we can then conclude about toppers and average marks etc.

A database management system stores data, in such a way which is easier to retrieve, manipulate and helps to produce information.

Characteristics

Traditionally data was organized in file formats. DBMS was all new concepts then and all the research was done to make it to overcome all the deficiencies in traditional style of data management. Modern DBMS has the following characteristics:

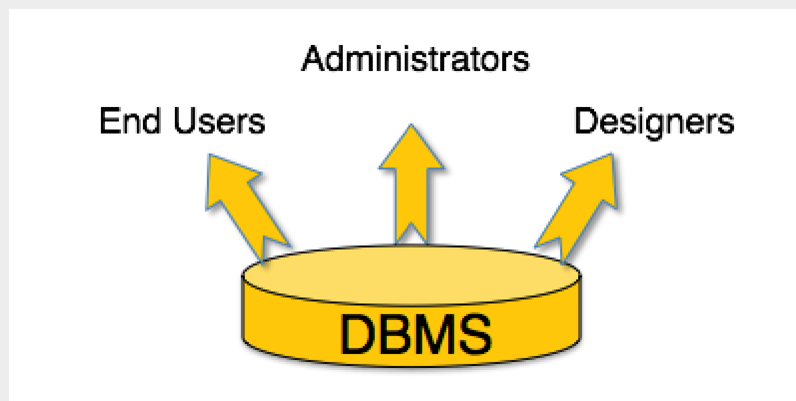
- **Real-world entity**: Modern DBMS are more realistic and uses real world entities to design its architecture. It uses the behavior and attributes too. For example, a school database may use student as entity and their age as their attribute.
- **Relation-based tables**: DBMS allows entities and relations among them to form as tables. This eases the concept of data saving. A user can understand the architecture of database just by looking at table names etc.
- **Isolation of data and application**: A database system is entirely different than its data. Where database is said to active entity, data is said to be passive one on which the database works and organizes. DBMS also stores metadata which is data about data, to ease its own process.
- **Less redundancy**: DBMS follows rules of normalization, which splits a relation when any of its attributes is having redundancy in values. Following normalization, which itself is a mathematically rich and scientific process, make the entire database to contain as less redundancy as possible.
- **Consistency**: DBMS always enjoy the state on consistency where the previous form of data storing applications like file processing does not guarantee this. Consistency is a state where every relation in database remains consistent. There exist methods and techniques, which can detect attempt of leaving database in inconsistent state.
- **Query Language**: DBMS is equipped with query language, which makes it more efficient to retrieve and manipulate data. A user can apply as many and different filtering

options, as he or she wants. Traditionally it was not possible where file-processing system was used.

- **ACID Properties**: DBMS follows the concepts for ACID properties, which stands for Atomicity, Consistency, Isolation and Durability. These concepts are applied on transactions, which manipulate data in database. ACID properties maintains database in healthy state in multi-transactional environment and in case of failure.
- **Multiuser and Concurrent Access**: DBMS support multi-user environment and allows them to access and manipulate data in parallel. Though there are restrictions on transactions when they attempt to handle same data item, but users are always unaware of them.
- **Multiple views**: DBMS offers multiples views for different users. A user who is in sales department will have a different view of database than a person working in production department. This enables user to have a concentrate view of database according to their requirements.
- **Security**: Features like multiple views offers security at some extent where users are unable to access data of other users and departments. DBMS offers methods to impose constraints while entering data into database and retrieving data at later stage. DBMS offers many different levels of security features, which enables multiple users to have different view with different features. For example, a user in sales department cannot see data of purchase department is one thing, additionally how much data of sales department he can see, can also be managed. Because DBMS is not saved on disk as traditional file system it is very hard for a thief to break the code.

Users

DBMS is used by various users for various purposes. Some may involve in retrieving data and some may involve in backing it up. Some of them are described as follows:



[*Image: DBMS Users*]

- **Administrators**: A bunch of users maintain the DBMS and are responsible for administrating the database. They are responsible to look after its usage and by whom it should be used. They create users access and apply limitation to maintain isolation and force security. Administrators also look after DBMS resources like system license, software application and tools required and other hardware related maintenance.

- **Designer**: This is the group of people who actually works on designing part of database. The actual database is started with requirement analysis followed by a good designing process. They people keep a close watch on what data should be kept and in what format. They identify and design the whole set of entities, relations, constraints and views.
- **End Users**: This group contains the persons who actually take advantage of database system. End users can be just viewers who pay attention to the logs or market rates or end users can be as sophisticated as business analysts who takes the most of it.

DBMS - Architecture

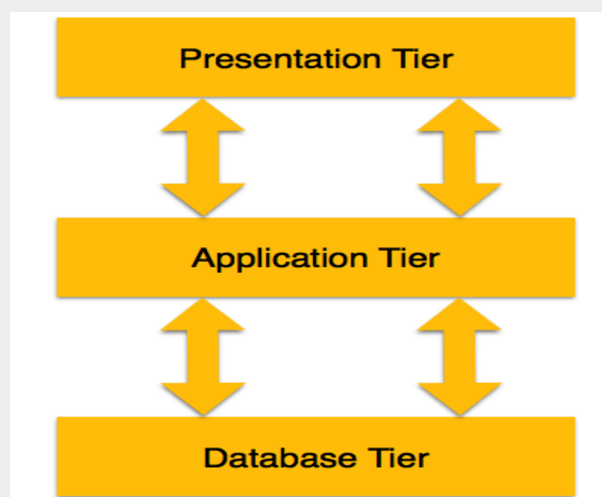
The design of a Database Management System highly depends on its architecture. It can be centralized or decentralized or hierarchical. DBMS architecture can be seen as single tier or multi tier. n-tier architecture divides the whole system into related but independent n modules, which can be independently modified, altered, changed or replaced.

In 1-tier architecture, DBMS is the only entity where user directly sits on DBMS and uses it. Any changes done here will directly be done on DBMS itself. It does not provide handy tools for end users and preferably database designer and programmers use single tier architecture.

If the architecture of DBMS is 2-tier then must have some application, which uses the DBMS. Programmers use 2-tier architecture where they access DBMS by means of application. Here application tier is entirely independent of database in term of operation, design and programming.

3-tier architecture

Most widely used architecture is 3-tier architecture. 3-tier architecture separates it tier from each other on basis of users. It is described as follows:



[*Image: 3-tier DBMS architecture*]

- **Database *Data* Tier:** At this tier, only database resides. Database along with its query processing languages sits in layer-3 of 3-tier architecture. It also contains all relations and their constraints.
- **Application *Middle* Tier:** At this tier the application server and program, which access database, resides. For a user this application tier works as abstracted view of database. Users are unaware of any existence of database beyond application. For database-tier, application tier is the user of it. Database tier is not aware of any other user beyond application tier. This tier works as mediator between the two.
- **User *Presentation* Tier:** An end user sits on this tier. From a users aspect this tier is everything. He/she doesn't know about any existence or form of database beyond this layer. At this layer multiple views of database can be provided by the application. All views are generated by applications, which resides in application tier.

Multiple tier database architecture is highly modifiable as almost all its components are independent and can be changed independently.<

DBMS - Data Models

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

The very first data model could be flat data-models where all the data used to be kept in same plane. Because earlier data models were not so scientific they were prone to introduce lots of duplication and update anomalies.

Entity-Relationship Model

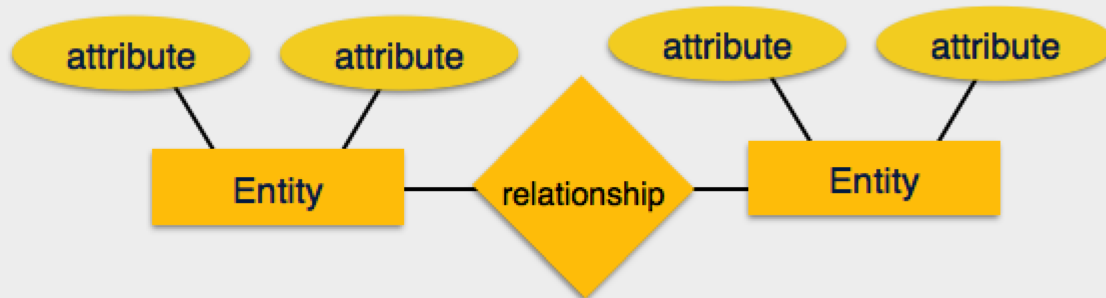
Entity-Relationship model is based on the notion of real world entities and relationship among them. While formulating real-world scenario into database model, ER Model creates entity set, relationship set, general attributes and constraints.

ER Model is best used for the conceptual design of database.

ER Model is based on:

- **Entities** and their *attributes*
- **Relationships** among entities

These concepts are explained below.



[*Image: ER Model*]

- **Entity**

  An entity in ER Model is real world entity, which has some properties called ***attributes***. Every attribute is defined by its set of values, called ***domain***.

  For example, in a school database, a student is considered as an entity. Student has various attributes like name, age and class etc.

- **Relationship**

  The logical association among entities is called ***relationship***. Relationships are mapped with entities in various ways. Mapping cardinalities define the number of association between two entities.

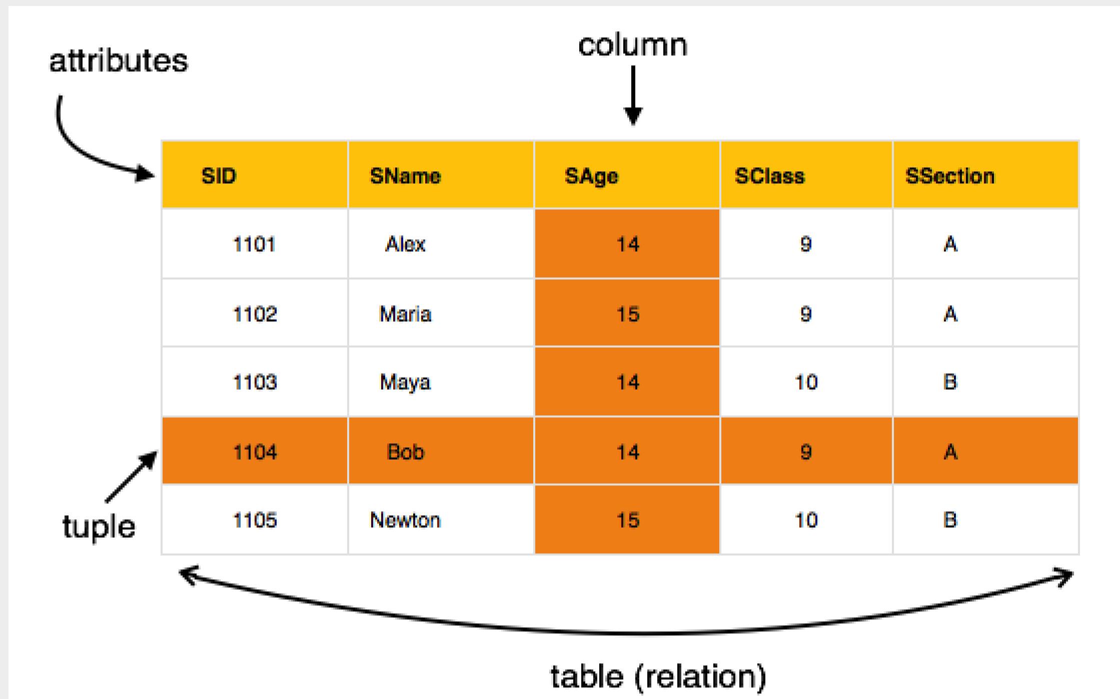
  Mapping cardinalities:

  - one to one
  - one to many
  - many to one
  - many to many

ER-Model is explained here.

Relational Model

The most popular data model in DBMS is Relational Model. It is more scientific model then others. This model is based on first-order predicate logic and defines table as an n-ary relation.



[*Image: Table in relational Model*]

The main highlights of this model are:

- Data is stored in tables called relations.
- Relations can be normalized.
- In normalized relations, values saved are atomic values.
- Each row in relation contains unique value
- Each column in relation contains values from a same domain.

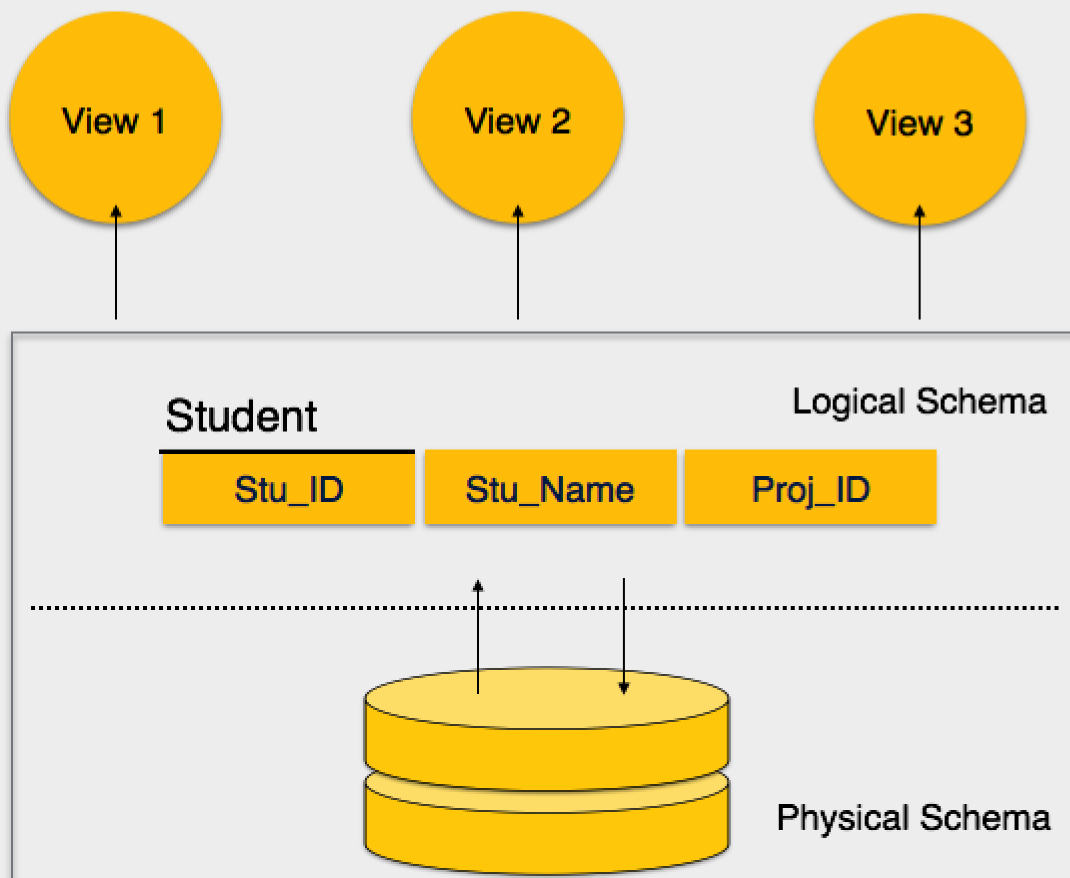
Relational Model is explained here.

DBMS - Data Schemas

Database schema

Database schema skeleton structure of and it represents the logical view of entire database. It tells about how the data is organized and how relation among them is associated. It formulates all database constraints that would be put on data in relations, which resides in database.

A database schema defines its entities and the relationship among them. Database schema is a descriptive detail of the database, which can be depicted by means of schema diagrams. All these activities are done by database designer to help programmers in order to give some ease of understanding all aspect of database.



[*Image: Database Schemas*]

Database schema can be divided broadly in two categories:

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

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

Database Instance

It is important that we distinguish these two terms individually. Database schema is the skeleton of database. It is designed when database doesn't exist at all and very hard to do any changes once the database is operational. Database schema does not contain any data or information.

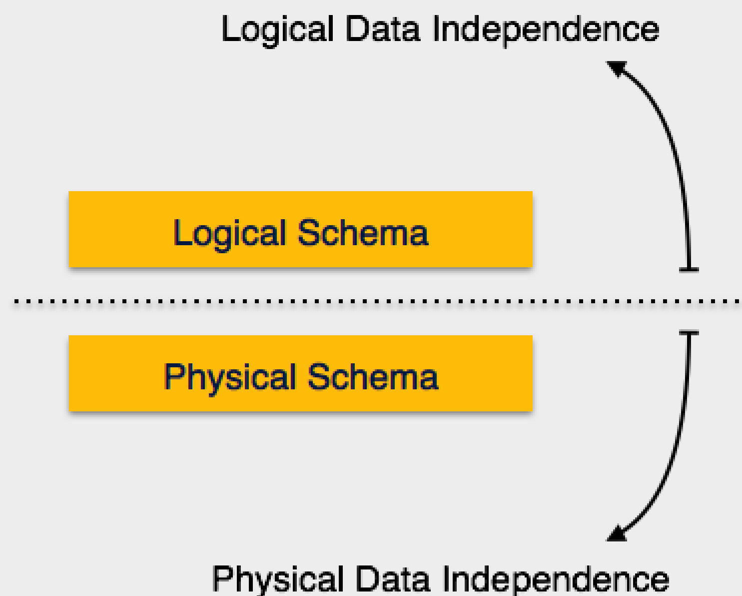
Database instances, is a state of operational database with data at any given time. This is a snapshot of database. Database instances tend to change with time. DBMS ensures that its every instance *state* must be a valid state by keeping up to all validation, constraints and condition that database designers has imposed or it is expected from DBMS itself.

DBMS - Data Independence

If the database system is not multi-layered then it will be very hard to make any changes in the database system. Database systems are designed in multi-layers as we leant earlier.

Data Independence:

There's a lot of data in whole database management system other than user's data. DBMS comprises of three kinds of schemas, which is in turn data about data *Meta−Data*. Meta-data is also stored along with database, which once stored is then hard to modify. But as DBMS expands, it needs to be changed over the time satisfy the requirements of users. But if the whole data were highly dependent it would become tedious and highly complex.



[*Image: Data independence*]

Data about data itself is divided in layered architecture so that when we change data at one layer it does not affect the data layered at different level. This data is independent but mapped on each other.

Logical Data Independence

Logical data is data about database, that is, it stores information about how data is managed inside. For example, a table *relation* stored in the database and all constraints, which are applied on that relation.

Logical data independence is a kind of mechanism, which liberalizes itself from actual data stored on the disk. If we do some changes on table format it should not change the data residing on disk.

Physical Data Independence

All schemas are logical and actual data is stored in bit format on the disk. Physical data independence is the power to change the physical data without impacting the schema or logical data.

For example, in case we want to change or upgrade the storage system itself, that is, using SSD instead of Hard-disks should not have any impact on logical data or schemas.

ER Model Basic Concepts

Entity relationship model defines the conceptual view of database. It works around real world entity and association among them. At view level, ER model is considered well for designing databases.

Entity

A real-world thing either animate or inanimate that can be easily identifiable and distinguishable. For example, in a school database, student, teachers, class and course offered can be considered as entities. All entities have some attributes or properties that give them their identity.

An entity set is a collection of similar types of entities. Entity set may contain entities with attribute sharing similar values. For example, Students set may contain all the student of a school; likewise Teachers set may contain all the teachers of school from all faculties. Entities sets need not to be disjoint.

Attributes

Entities are represented by means of their properties, called attributes. All attributes have values. For example, a student entity may have name, class, age as attributes.

There exist a domain or range of values that can be assigned to attributes. For example, a student's name cannot be a numeric value. It has to be alphabetic. A student's age cannot be negative, etc.

Types of attributes:

- **Simple attribute:**

  Simple attributes are atomic values, which cannot be divided further. For example, student's phone-number is an atomic value of 10 digits.

- **Composite attribute:**

  Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first_name and last_name.

- **Derived attribute:**

  Derived attributes are attributes, which do not exist physical in the database, but there values are derived from other attributes presented in the database. For example, average_salary in a department should be saved in database instead it can be derived. For another example, age can be derived from data_of_birth.

- **Single-valued attribute:**

  Single valued attributes contain on single value. For example: Social_Security_Number.

- **Multi-value attribute:**

  Multi-value attribute may contain more than one values. For example, a person can have more than one phone numbers, email_addresses etc.

These attribute types can come together in a way like:

- simple single-valued attributes
- simple multi-valued attributes
- composite single-valued attributes
- composite multi-valued attributes

Entity-set and Keys

Key is an attribute or collection of attributes that uniquely identifies an entity among entity set.

For example, roll_number of a student makes her/him identifiable among students.

- **Super Key:** Set of attributes *oneormore* that collectively identifies an entity in an entity set.
- **Candidate Key:** Minimal super key is called candidate key that is, supers keys for which no proper subset are a superkey. An entity set may have more than one candidate key.
- **Primary Key:** This is one of the candidate key chosen by the database designer to uniquely identify the entity set.

Relationship

The association among entities is called relationship. For example, employee entity has relation works_at with department. Another example is for student who enrolls in some course. Here, Works_at and Enrolls are called relationship.

Relationship Set:

Relationship of similar type is called relationship set. Like entities, a relationship too can have attributes. These attributes are called descriptive attributes.

Degree of relationship

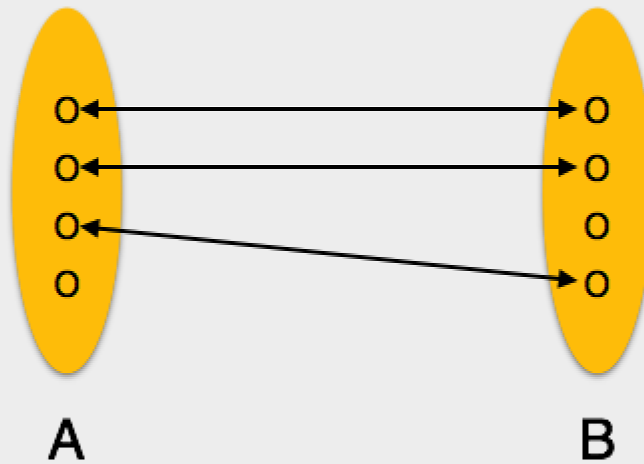
The number of participating entities in an relationship defines the degree of the relationship.

- Binary = degree 2
- Ternary = degree 3
- n-ary = degree

Mapping Cardinalities:

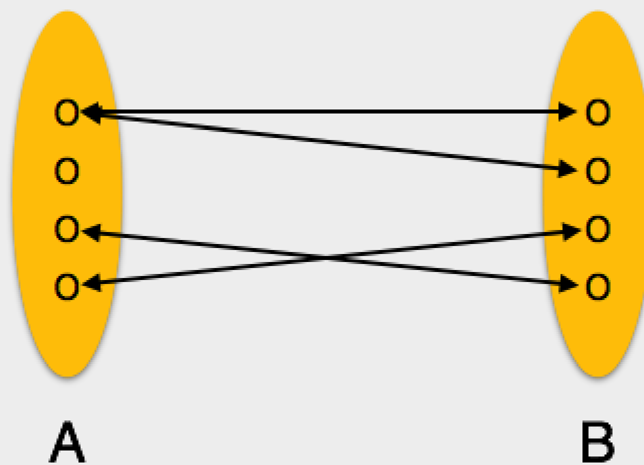
Cardinality defines the number of entities in one entity set which can be associated to the number of entities of other set via relationship set.

- **One-to-one:** one entity from entity set A can be associated with at most one entity of entity set B and vice versa.



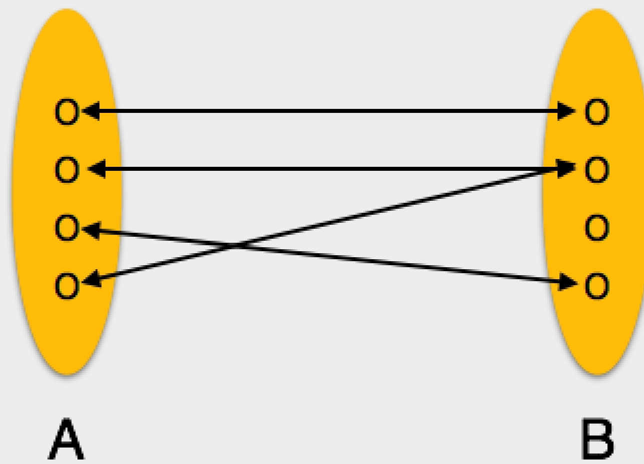
[*Image: One-to-one relation*]

- **One-to-many:** One entity from entity set A can be associated with more than one entities of entity set B but from entity set B one entity can be associated with at most one entity.



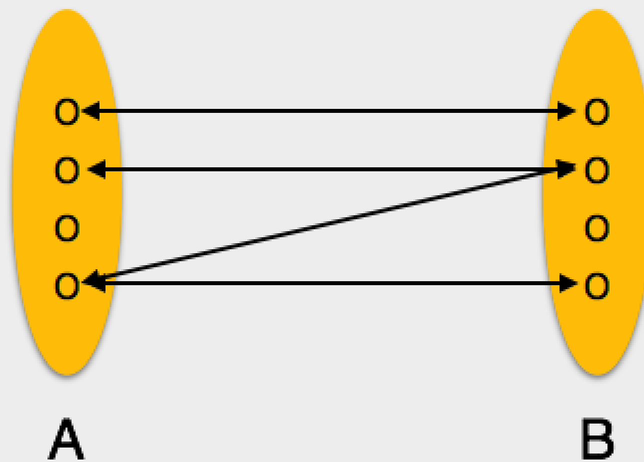
[*Image: One-to-many relation*]

- **Many-to-one:** More than one entities from entity set A can be associated with at most one entity of entity set B but one entity from entity set B can be associated with more than one entity from entity set A.



[*Image: Many-to-one relation*]

- **Many-to-many:** one entity from A can be associated with more than one entity from B and vice versa.



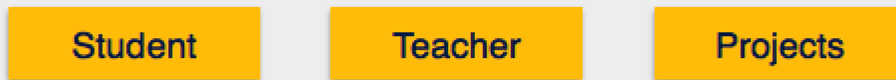
[*Image: Many-to-many relation*]

ER Diagram Representation

Now we shall learn how ER Model is represented by means of ER diagram. Every object like entity, attributes of an entity, relationship set, and attributes of relationship set can be represented by tools of ER diagram.

Entity

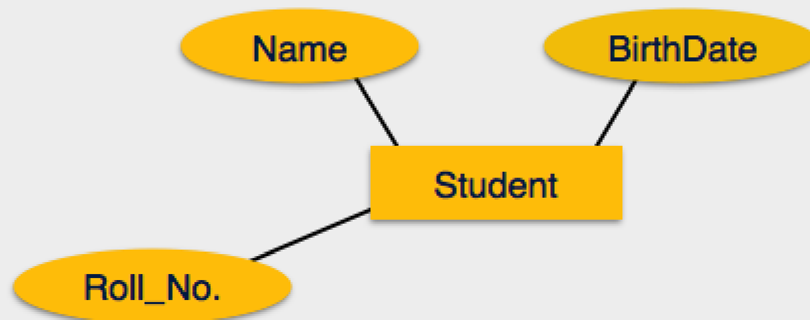
Entities are represented by means of rectangles. Rectangles are named with the entity set they represent.



[*Image: Entities in a school database*]

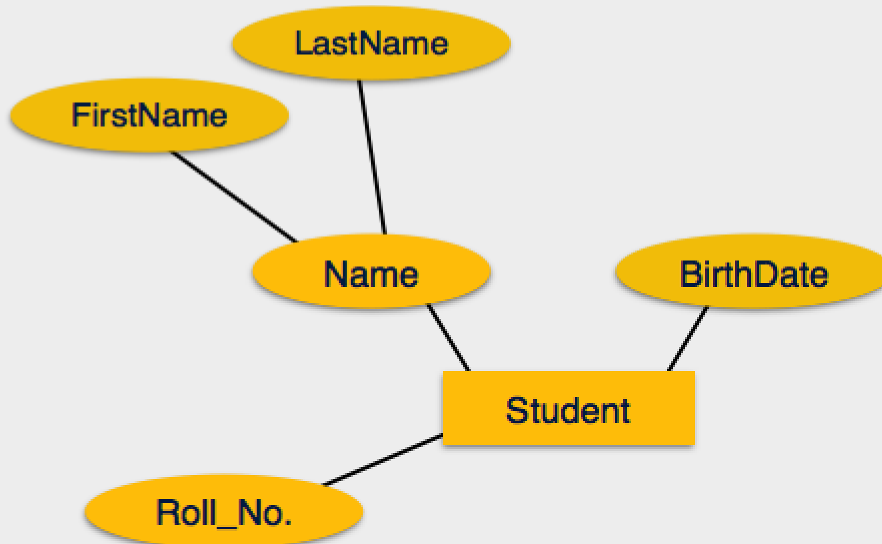
Attributes

Attributes are properties of entities. Attributes are represented by means of eclipses. Every eclipse represents one attribute and is directly connected to its entity *rectangle*.



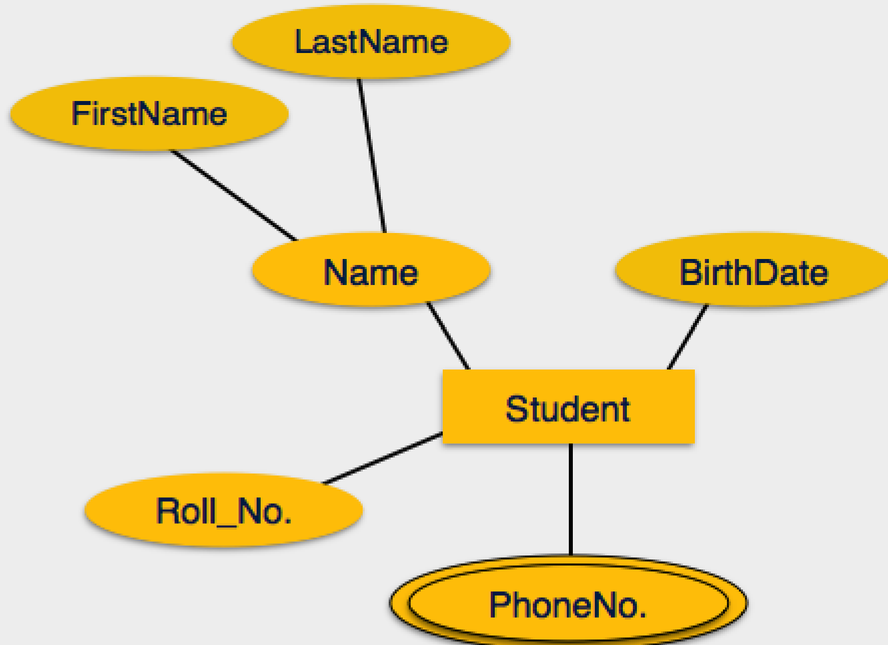
[*Image: Simple Attributes*]

If the attributes are ***composite***, they are further divided in a tree like structure. Every node is then connected to its attribute. That is composite attributes are represented by eclipses that are connected with an eclipse.



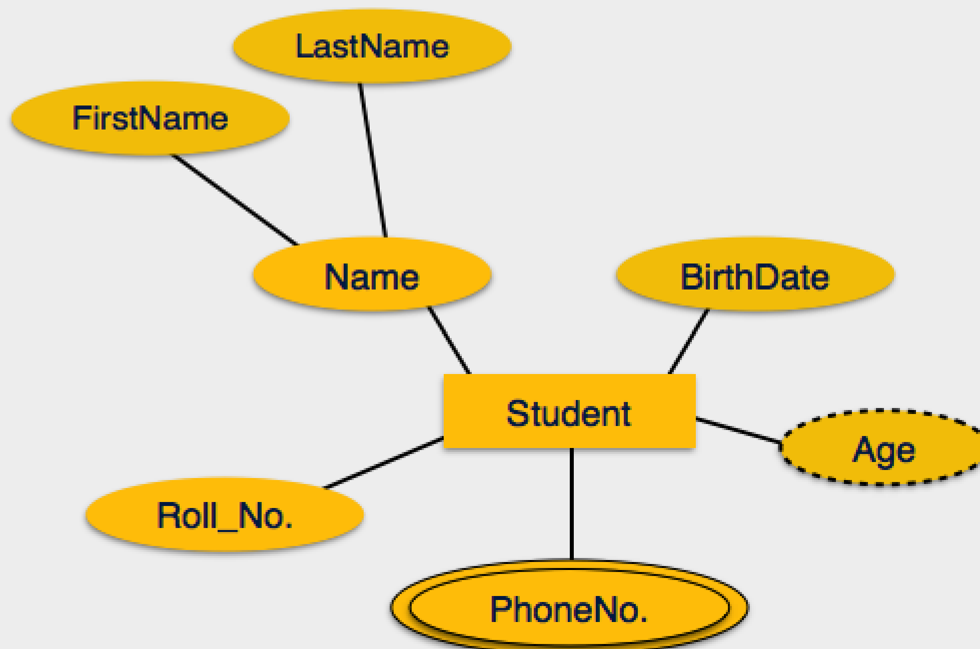
[*Image: Composite Attributes*]

Multivalued attributes are depicted by double eclipse.



[*Image: Multivalued Attributes*]

Derived attributes are depicted by dashed eclipse.



[*Image: Derived Attributes*]

Relationship

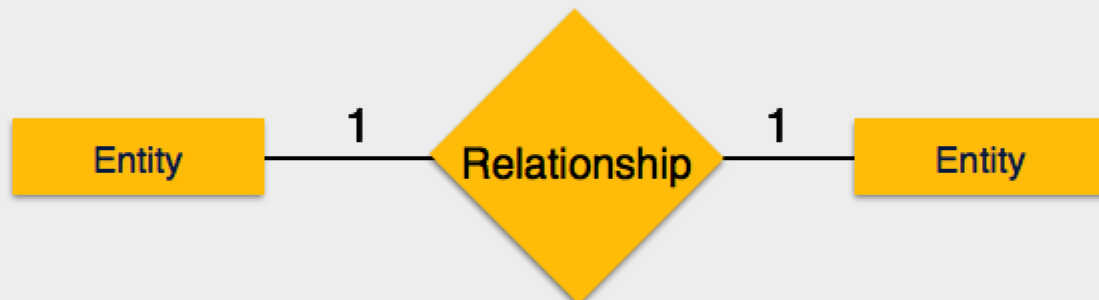
Relationships are represented by diamond shaped box. Name of the relationship is written in the diamond-box. All entities *rectangles*, participating in relationship, are connected to it by a line.

Binary relationship and cardinality

A relationship where two entities are participating, is called a ***binary relationship***. Cardinality is the number of instance of an entity from a relation that can be associated with the relation.

- **One-to-one**

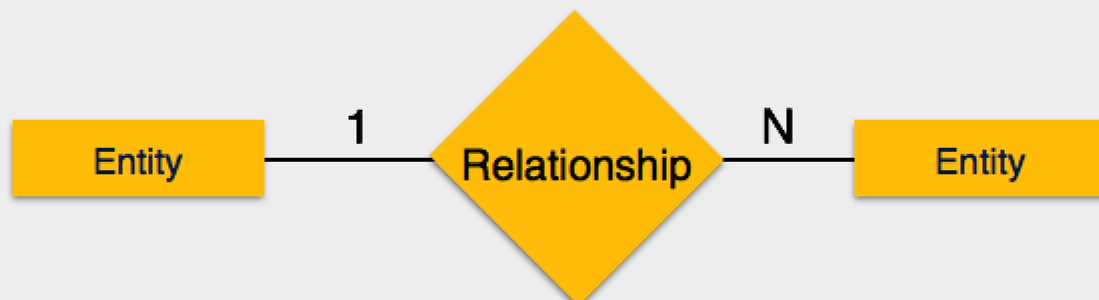
  When only one instance of entity is associated with the relationship, it is marked as '1'. This image below reflects that only 1 instance of each entity should be associated with the relationship. It depicts one-to-one relationship



[*Image: One-to-one*]

- **One-to-many**

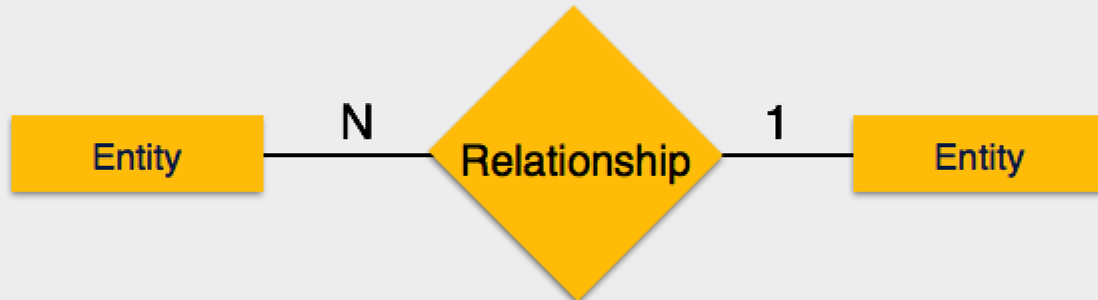
  When more than one instance of entity is associated with the relationship, it is marked as 'N'. This image below reflects that only 1 instance of entity on the left and more than one instance of entity on the right can be associated with the relationship. It depicts one-to-many relationship



[*Image: One-to-many*]

- **Many-to-one**

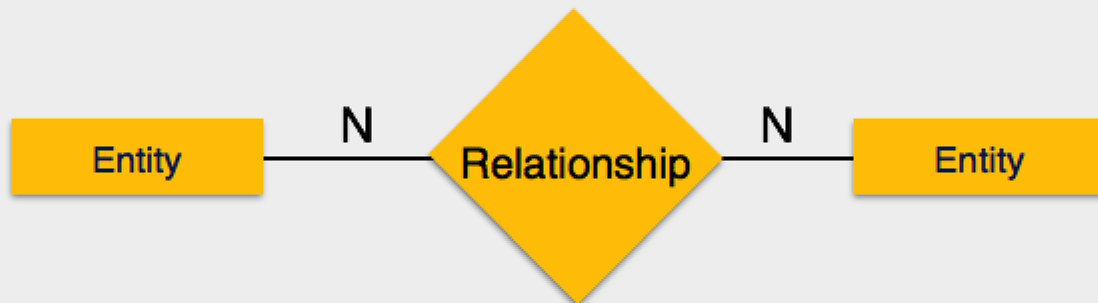
  When more than one instance of entity is associated with the relationship, it is marked as 'N'. This image below reflects that more than one instance of entity on the left and only one instance of entity on the right can be associated with the relationship. It depicts many-to-one relationship



[*Image: Many-to-one*]

- **Many-to-many**

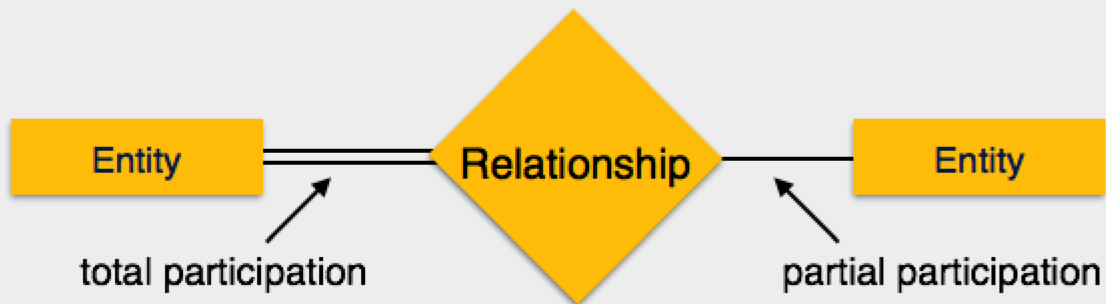
  This image below reflects that more than one instance of entity on the left and more than one instance of entity on the right can be associated with the relationship. It depicts many-to-many relationship



[*Image: Many-to-many*]

Participation Constraints

- **Total Participation:** Each entity in the entity is involved in the relationship. Total participation is represented by double lines.
- **Partial participation:** Not all entities are involved in the relation ship. Partial participation is represented by single line.



[*Image: Participation Constraints*]

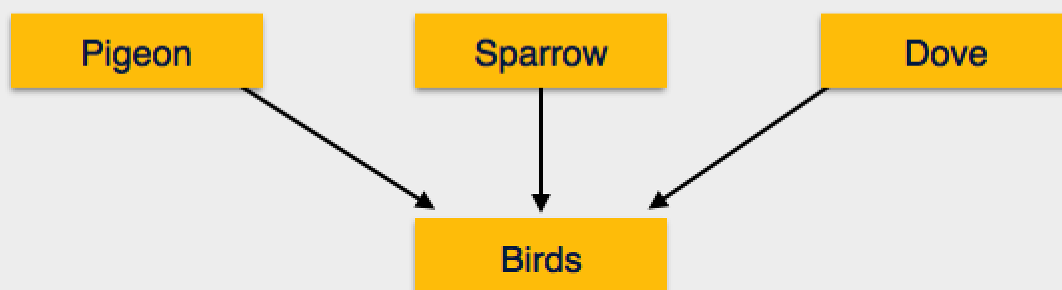
Generalization, Aggregation

ER Model has the power of expressing database entities in conceptual hierarchical manner such that, as the hierarchical goes up it generalize the view of entities and as we go deep in the hierarchy it gives us detail of every entity included.

Going up in this structure is called generalization, where entities are clubbed together to represent a more generalized view. For example, a particular student named, Mira can be generalized along with all the students, the entity shall be student, and further a student is person. The reverse is called specialization where a person is student, and that student is Mira.

Generalization

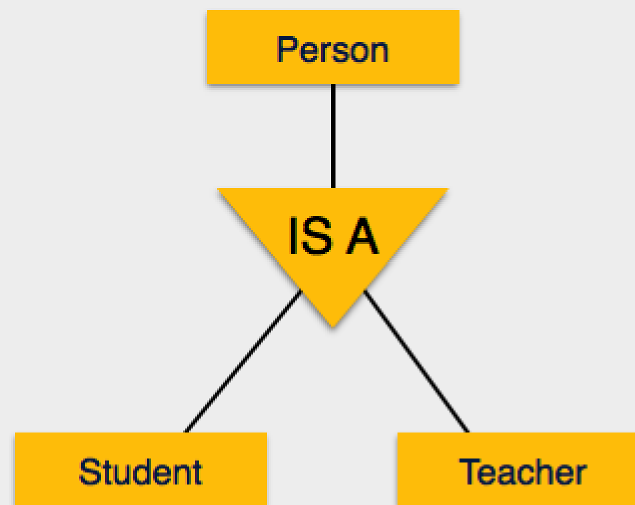
As mentioned above, the process of generalizing entities, where the generalized entities contain the properties of all the generalized entities is called Generalization. In generalization, a number of entities are brought together into one generalized entity based on their similar characteristics. For an example, pigeon, house sparrow, crow and dove all can be generalized as Birds.



[*Image: Generalization*]

Specialization

Specialization is a process, which is opposite to generalization, as mentioned above. In specialization, a group of entities is divided into sub-groups based on their characteristics. Take a group Person for example. A person has name, date of birth, gender etc. These properties are common in all persons, human beings. But in a company, a person can be identified as employee, employer, customer or vendor based on what role do they play in company.



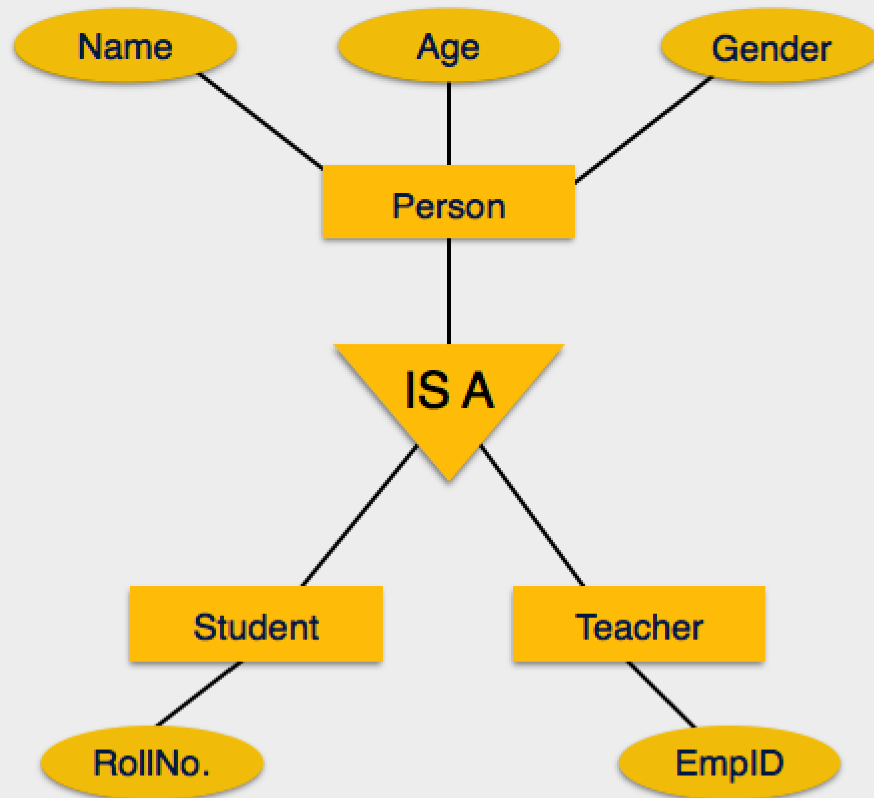
[*Image: Specialization*]

Similarly, in a school database, a person can be specialized as teacher, student or staff; based on what role do they play in school as entities.

Inheritance

We use all above features of ER-Model, in order to create classes of objects in object oriented programming. This makes it easier for the programmer to concentrate on what she is programming. Details of entities are generally hidden from the user, this process known as abstraction.

One of the important features of Generalization and Specialization, is inheritance, that is, the attributes of higher-level entities are inherited by the lower level entities.



[*Image: Inheritance*]

For example, attributes of a person like name, age, and gender can be inherited by lower level entities like student and teacher etc.

DBMS Codd's Rules

Dr Edgar F. Codd did some extensive research in Relational Model of database systems and came up with twelve rules of his own which according to him, a database must obey in order to be a true relational database.

These rules can be applied on a database system that is capable of managing is stored data using only its relational capabilities. This is a foundation rule, which provides a base to imply other rules on it.

Rule 1: Information rule

This rule states that all information *data*, which is stored in the database, must be a value of some table cell. Everything in a database must be stored in table formats. This information can be user data or meta-data.

Rule 2: Guaranteed Access rule

This rule states that every single data element *value* is guaranteed to be accessible logically with combination of table-name, primary-key *row value* and attribute-name *column value*. No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL values

This rule states the NULL values in the database must be given a systematic treatment. As a NULL may have several meanings, i.e. NULL can be interpreted as one the following: data is missing, data is not known, data is not applicable etc.

Rule 4: Active online catalog

This rule states that the structure description of whole database must be stored in an online catalog, i.e. data dictionary, which can be accessed by the authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive data sub-language rule

This rule states that a database must have a support for a language which has linear syntax which is capable of data definition, data manipulation and transaction management operations. Database can be accessed by means of this language only, either directly or by means of some application. If the database can be accessed or manipulated in some way without any help of this language, it is then a violation.

Rule 6: View updating rule

This rule states that all views of database, which can theoretically be updated, must also be updatable by the system.

Rule 7: High-level insert, update and delete rule

This rule states the database must employ support high-level insertion, updation and deletion. This must not be limited to a single row that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical data independence

This rule states that the application should not have any concern about how the data is physically stored. Also, any change in its physical structure must not have any impact on application.

Rule 9: Logical data independence

This rule states that the logical data must be independent of its user's view *application*. Any change in logical data must not imply any change in the application using it. For example, if two tables are merged or one is split into two different tables, there should be no impact the change on user application. This is one of the most difficult rule to apply.

Rule 10: Integrity independence

This rule states that the database must be independent of the application using it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes database independent of the front-end application and its interface.

Rule 11: Distribution independence

This rule states that the end user must not be able to see that the data is distributed over various locations. User must also see that data is located at one site only. This rule has been proven as a foundation of distributed database systems.

Rule 12: Non-subversion rule

This rule states that if a system has an interface that provides access to low level records, this interface then must not be able to subvert the system and bypass security and integrity constraints.

Relational Data Model

Relational data model is the primary data model, which is used widely around the world for data storage and processing. This model is simple and have all the properties and capabilities required to process data with storage efficiency.

Concepts

Tables: In relation data model, relations are saved in the format of Tables. This format stores the relation among entities. A table has rows and columns, where rows represent records and columns represents the attributes.

Tuple: A single row of a table, which contains a single record for that relation, is called a tuple.

Relation instance: A finite set of tuples in the relational database system represents relation instance. Relation instances do not have duplicate tuples.

Relation schema: This describes the relation name *table name*, attributes and their names.

Relation key: Each row has one or more attributes which can identify the row in the relation *table* uniquely, is called the relation key.

Attribute domain: Every attribute has some pre-defined value scope, known as attribute domain.

Constraints

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

- Key Constraints
- Domain constraints
- Referential integrity constraints

Key Constraints:

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

Key constraints forces that:

- in a relation with a key attribute, no two tuples can have identical value for key attributes.
- key attribute can not have NULL values.

Key constrains are also referred to as Entity Constraints.

Domain constraints

Attributes have specific values in real-world scenario. For example, age can only be positive integer. The same constraints has been tried to employ on the attributes of a relation. Every attribute is bound to have a specific range of values. For example, age can not be less than zero and telephone number can not be a outside 0-9.

Referential integrity constraints

This integrity constraints works on the concept of Foreign Key. A key attribute of a relation can be referred in other relation, where it is called ***foreign key***.

Referential integrity constraint states that if a relation refers to an key attribute of a different or same relation, that key element must exists.

Relational Algebra

Relational database systems are expected to be equipped by a query language that can assist its user to query the database instances. This way its user empowers itself and can populate the results as required. There are two kinds of query languages, relational algebra and relational calculus.

Relational algebra

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either unary or binary. They accept relations as their input and yields relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

Fundamental operations of Relational algebra:

- Select
- Project
- Union
- Set different
- Cartesian product
- Rename

These are defined briefly as follows:

Select Operation *σ*

Selects tuples that satisfy the given predicate from a relation.

Notation $\sigma_p r$

Where *p* stands for selection predicate and r stands for relation. *p* is prepositional logic formulae which may use connectors like and, or and not. These terms may use relational operators like: $=, \neq, \geq, <, >, \leq$.

For example:

$\sigma_{subject="database"}$(Books)

Output : Selects tuples from books where subject is 'database'.

$\sigma_{\text{subject="database" and price="450"}}$(Books)

Output : Selects tuples from books where subject is 'database' and 'price' is 450.

$\sigma_{subject="database" \text{ and } price < "450" \text{ or } year > "2010"}$(Books)

Output : Selects tuples from books where subject is 'database' and 'price' is 450 or the publication year is greater than 2010, that is published after 2010.

Project Operation ∏

Projects column*s* that satisfy given predicate.

Notation: $\prod_{A1, A2, An} r$

Where a_1, a_2 , a_n are attribute names of relation r.

Duplicate rows are automatically eliminated, as relation is a set.

for example:

$\prod_{subject, author}$ (Books)

Selects and projects columns named as subject and author from relation Books.

Union Operation ∪

Union operation performs binary union between two given relations and is defined as:

$r \cup s = \{ t \mid t \in r \text{ or } t \in s\}$

Notion: r U s

Where r and s are either database relations or relation result set *temporaryrelation*.

For a union operation to be valid, the following conditions must hold:

- r, s must have same number of attributes.
- Attribute domains must be compatible.

Duplicate tuples are automatically eliminated.

$\prod_{author}$ (Books) ∪ $\prod_{author}$ (Articles)

Output : Projects the name of author who has either written a book or an article or both.

Set Difference −

The result of set difference operation is tuples which present in one relation but are not in the second relation.

Notation: r − s

Finds all tuples that are present in r but not s.

$\prod_{author}$ (Books) − $\prod_{author}$ (Articles)

Output: Results the name of authors who has written books but not articles.

Cartesian Product X

Combines information of two different relations into one.

Notation: r X s

Where r and s are relations and there output will be defined as:

r X s = { q t | q ∈ r and t ∈ s}

$\prod_{author = 'tutorialspoint'}$(Books X Articles)

Output : yields a relation as result which shows all books and articles written by tutorialspoint.

Rename operation *ρ*

Results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. rename operation is denoted with small greek letter rho *ρ*

Notation: $\rho_x E$

Where the result of expression E is saved with name of x.

Additional operations are:

- Set intersection
- Assignment
- Natural join

Relational Calculus

In contrast with Relational Algebra, Relational Calculus is non-procedural query language, that is, it tells what to do but never explains the way, how to do it.

Relational calculus exists in two forms:

Tuple relational calculus *TRC*

Filtering variable ranges over tuples

Notation: { T | Condition }

Returns all tuples T that satisfies condition.

For Example:

{ T.name | Author(T) AND T.article = 'database' }

Output: returns tuples with 'name' from Author who has written article on 'database'.

TRC can be quantified also. We can use Existential ∃and Universal Quantifiers ∀.

For example:

{ R| ∃T ∈ Authors(T.article='database' AND R.name=T.name)}

Output : the query will yield the same result as the previous one.

Domain relational calculus *DRC*

In DRC the filtering variable uses domain of attributes instead of entire tuple values *asdoneinTRC*,*mentionedabove*.

Notation:

$\{ a_1, a_2, a_3, ..., a_n \mid P (a_1, a_2, a_3, ... ,a_n)\}$

where a1, a2 are attributes and P stands for formulae built by inner attributes.

For example:

{< article, page, subject > | ∈ TutorialsPoint ∧ subject = 'database'}

Output: Yields Article, Page and Subject from relation TutorialsPoint where Subject is database.

Just like TRC, DRC also can be written using existential and universal quantifiers. DRC also involves relational operators.

Expression power of Tuple relation calculus and Domain relation calculus is equivalent to Relational Algebra.

ER to Relational Model

ER Model when conceptualized into diagrams gives a good overview of entity-relationship, which is easier to understand. ER diagrams can be mapped to Relational schema that is, it is possible to create relational schema using ER diagram. Though we cannot import all the ER constraints into Relational model but an approximate schema can be generated.

There are more than one processes and algorithms available to convert ER Diagrams into Relational Schema. Some of them are automated and some of them are manual process. We may focus here on the mapping diagram contents to relational basics.

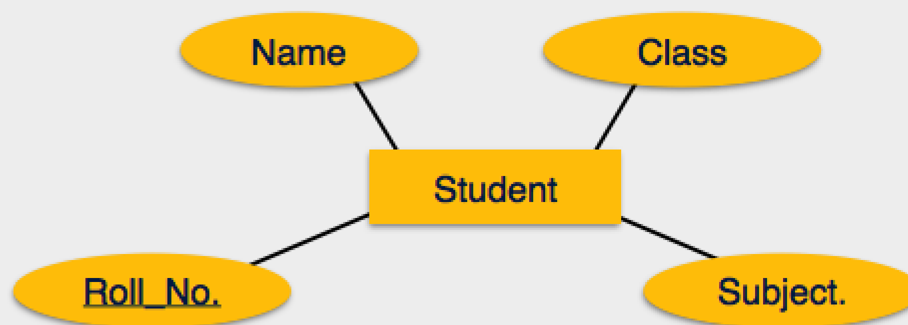
ER Diagrams mainly comprised of:

- Entity and its attributes
- Relationship, which is association among entities.

Mapping Entity

An entity is a real world object with some attributes.

Mapping Process *Algorithm*:



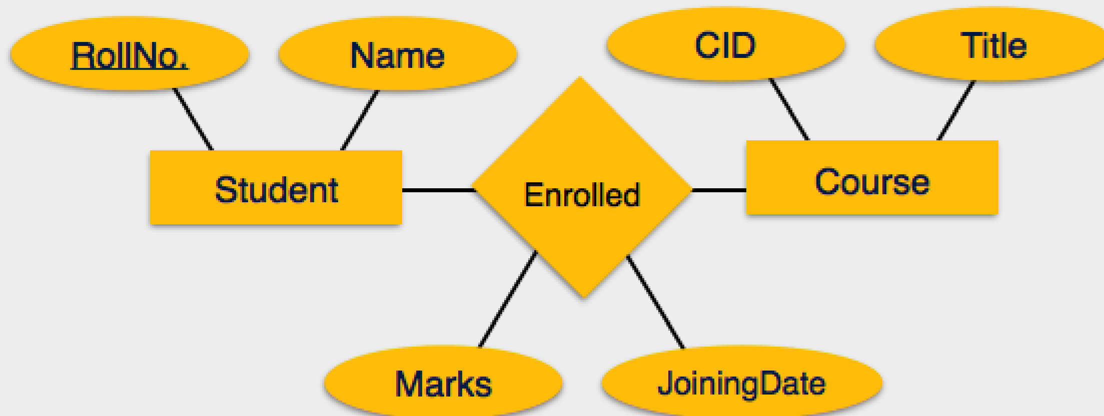
[*Image: Mapping Entity*]

- Create table for each entity
- Entity's attributes should become fields of tables with their respective data types.
- Declare primary key

Mapping relationship

A relationship is association among entities.

Mapping process *Algorithm*:



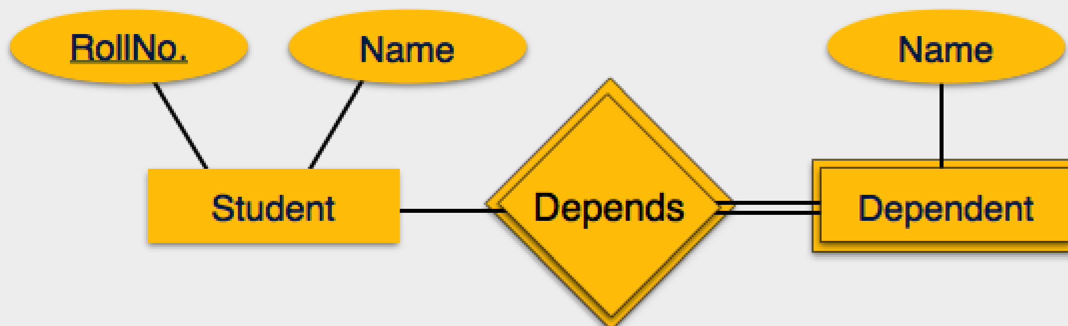
[*Image: Mapping relationship*]

- Create table for a relationship
- Add the primary keys of all participating Entities as fields of table with their respective data types.
- If relationship has any attribute, add each attribute as field of table.
- Declare a primary key composing all the primary keys of participating entities.
- Declare all foreign key constraints.

Mapping Weak Entity Sets

A weak entity sets is one which does not have any primary key associated with it.

Mapping process *Algorithm*:



[*Image: Mapping Weak Entity Sets*]

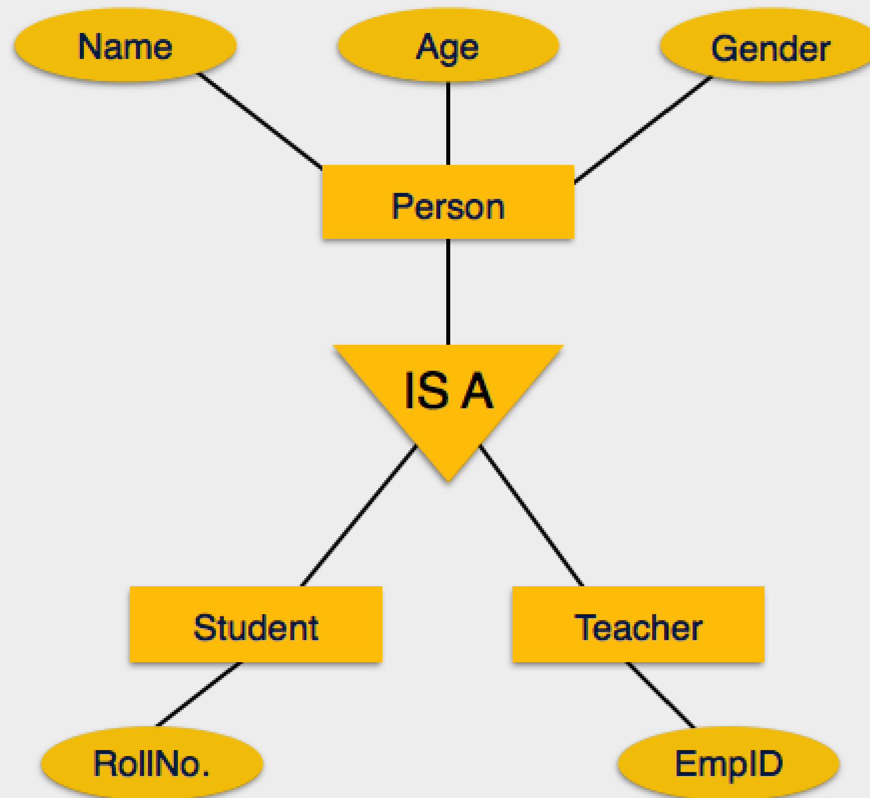
- Create table for weak entity set
- Add all its attributes to table as field

- Add the primary key of identifying entity set
- Declare all foreign key constraints

Mapping hierarchical entities

ER specialization or generalization comes in the form of hierarchical entity sets.

Mapping process *Algorithm*:



[*Image: Mapping hierarchical entities*]

- Create tables for all higher level entities
- Create tables for lower level entities
- Add primary keys of higher level entities in the table of lower level entities
- In lower level tables, add all other attributes of lower entities.
- Declare primary key of higher level table the primary key for lower level table
- Declare foreign key constraints.

SQL Overview

SQL is a programming language for Relational Databases. It is designed over relational algebra and tuple relational calculus. SQL comes as a package with all major distributions of RDBMS.

SQL comprises both data definition and data manipulation languages. Using the data definition properties of SQL, one can design and modify database schema whereas data manipulation properties allows SQL to store and retrieve data from database.

Data definition Language

SQL uses the following set of commands to define database schema:

CREATE

Creates new databases, tables and views from RDBMS

For example:

```
Create database tutorialspoint;
Create table article;
Create view for_students;
```

DROP

Drop commands deletes views, tables and databases from RDBMS

```
Drop object_type object_name;
Drop database tutorialspoint;
Drop table article;
Drop view for_students;
```

ALTER

Modifies database schema.

```
Alter object_type object_name parameters;
```

for example:

```
Alter table article add subject varchar;
```

This command adds an attribute in relation article with name subject of string type.

Data Manipulation Language

SQL is equipped with data manipulation language. DML modifies the database instance by inserting, updating and deleting its data. DML is responsible for all data modification in databases. SQL contains the following set of command in DML section:

- SELECT/FROM/WHERE
- INSERT INTO/VALUES
- UPDATE/SET/WHERE
- DELETE FROM/WHERE

These basic constructs allows database programmers and users to enter data and information into the database and retrieve efficiently using a number of filter options.

SELECT/FROM/WHERE

- **SELECT**

  This is one of the fundamental query command of SQL. It is similar to projection operation of relational algebra. It selects the attributes based on the condition described by WHERE clause.

- **FROM**

  This clause takes a relation name as an argument from which attributes are to be selected/projected. In case more than one relation names are given this clause corresponds to cartesian product.

- **WHERE**

  This clause defines predicate or conditions which must match in order to qualify the attributes to be projected.

  For example:

  ```
  Select author_name
  From book_author
  Where age > 50;
  ```

  This command will project names of author's from book_author relation whose age is greater than 50.

INSERT INTO/VALUES

This command is used for inserting values into rows of table *relation*.

Syntax is

```
INSERT INTO table (column1 [, column2, column3 ... ]) VALUES (value1 [, value2, value3 ... ])
```

Or

```
INSERT INTO table VALUES (value1, [value2, ... ])
```

For Example:

```
INSERT INTO tutorialspoint (Author, Subject) VALUES ("anonymous", "computers");
```

UPDATE/SET/WHERE

This command is used for updating or modifying values of columns of table *relation*.

Syntax is

```
UPDATE table_name SET column_name = value [, column_name = value ...] [WHERE condition]
```

For example:

```
UPDATE tutorialspoint SET Author="webmaster" WHERE Author="anonymous";
```

DELETE/FROM/WHERE

This command is used for removing one or more rows from table *relation*.

Syntax is

```
DELETE FROM table_name [WHERE condition];
```

For example:

```
DELETE FROM tutorialspoints
  WHERE Author="unknown";
```

For in-depth and practical knowledge of SQL, click here.

Database Normalization

Functional Dependency

Functional dependency *FD* is set of constraints between two attributes in a relation. Functional dependency says that if two tuples have same values for attributes A1, A2,..., An then those two tuples must have to have same values for attributes B1, B2, ..., Bn.

Functional dependency is represented by arrow sign →, that is X→Y, where X functionally determines Y. The left hand side attributes determines the values of attributes at right hand side.

Armstrong's Axioms

If F is set of functional dependencies then the closure of F, denoted as F^+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are set of rules, when applied repeatedly generates closure of functional dependencies.

- **Reflexive rule:** If alpha is a set of attributes and beta is_subset_of alpha, then alpha holds beta.
- **Augmentation rule:** if a → b holds and y is attribute set, then ay → by also holds. That is adding attributes in dependencies, does not change the basic dependencies.
- **Transitivity rule:** Same as transitive rule in algebra, if a → b holds and b → c holds then a → c also hold. a → b is called as a functionally determines b.

Trivial Functional Dependency

- **Trivial:** If an FD X → Y holds where Y subset of X, then it is called a trivial FD. Trivial FDs are always hold.
- **Non-trivial:** If an FD X → Y holds where Y is not subset of X, then it is called non-trivial FD.
- **Completely non-trivial:** If an FD X → Y holds where x intersect Y = Φ, is said to be completely non-trivial FD.

Normalization

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

- **Update anomalies:** if data items are scattered and are not linked to each other properly, then there may be instances when we try to update one data item that has copies of it scattered at several places, few instances of it get updated properly while few are left with there old values. This leaves database in an inconsistent state.

- **Deletion anomalies:** we tried to delete a record, but parts of it left undeleted because of unawareness, the data is also saved somewhere else.
- **Insert anomalies:** we tried to insert data in a record that does not exist at all.

Normalization is a method to remove all these anomalies and bring database to consistent state and free from any kinds of anomalies.

First Normal Form:

This is defined in the definition of relations *tables* itself. This rule defines that all the attributes in a relation must have atomic domains. Values in atomic domain are indivisible units.

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

[*Image: Unorganized relation*]

We re-arrange the relation *table* as below, to convert it to First Normal Form

Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

[*Image: Relation in 1NF*]

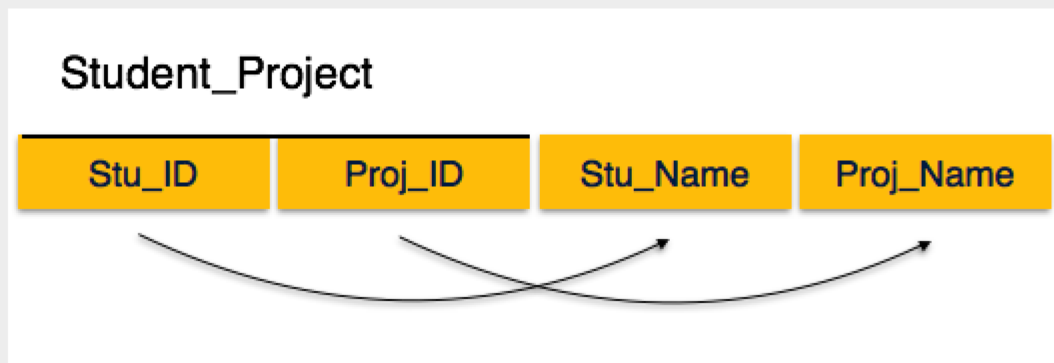
Each attribute must contain only single value from its pre-defined domain.

Second Normal Form:

Before we learn about second normal form, we need to understand the following:

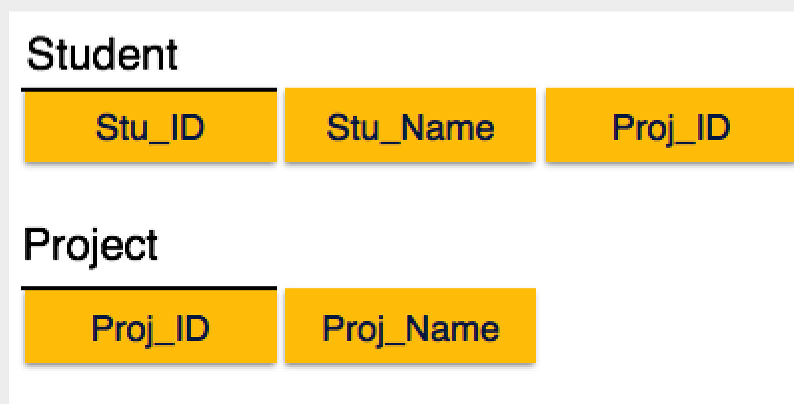
- **Prime attribute:** an attribute, which is part of prime-key, is prime attribute.
- **Non-prime attribute:** an attribute, which is not a part of prime-key, is said to be a non-prime attribute.

Second normal form says, that every non-prime attribute should be fully functionally dependent on prime key attribute. That is, if X → A holds, then there should not be any proper subset Y of X, for that Y → A also holds.



[*Image: Relation not in 2NF*]

We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID. According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attribute individually. But we find that Stu_Name can be identified by Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called partial dependency, which is not allowed in Second Normal Form.



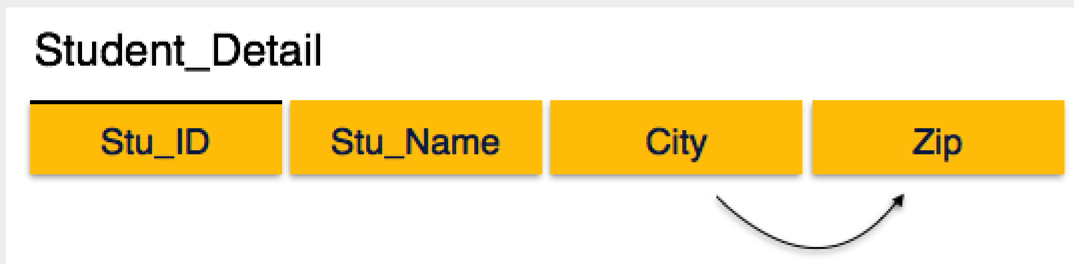
[*Image: Relation in 2NF*]

We broke the relation in two as depicted in the above picture. So there exists no partial dependency.

Third Normal Form:

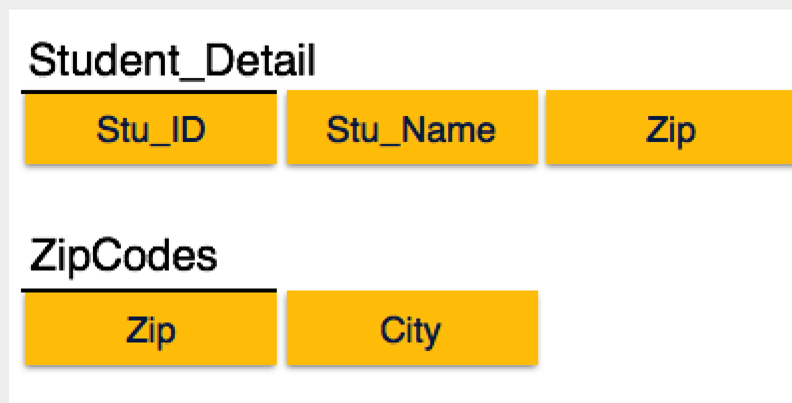
For a relation to be in Third Normal Form, it must be in Second Normal form and the following must satisfy:

- No non-prime attribute is transitively dependent on prime key attribute
- For any non-trivial functional dependency, X → A, then either
- X is a superkey or,
- A is prime attribute.



[*Image: Relation not in 3NF*]

We find that in above depicted Student_detail relation, Stu_ID is key and only prime key attribute. We find that City can be identified by Stu_ID as well as Zip itself. Neither Zip is a superkey nor City is a prime attribute. Additionally, Stu_ID → Zip → City, so there exists ***transitive dependency***.



[*Image: Relation in 3NF*]

We broke the relation as above depicted two relations to bring it into 3NF.

Boyce-Codd Normal Form:

BCNF is an extension of Third Normal Form in strict way. BCNF states that

- For any non-trivial functional dependency, X → A, then X must be a super-key.

In the above depicted picture, Stu_ID is super-key in Student_Detail relation and Zip is super-key in ZipCodes relation. So,

Stu_ID → Stu_Name, Zip

And

Zip → City

Confirms, that both relations are in BCNF.

Database Joins

We understand the benefits of Cartesian product of two relation, which gives us all the possible tuples that are paired together. But Cartesian product might not be feasible for huge relations where number of tuples are in thousands and the attributes of both relations are considerable large.

Join is combination of Cartesian product followed by selection process. Join operation pairs two tuples from different relations if and only if the given join condition is satisfied.

Following section should describe briefly about join types:

Theta *θ* join

θ in Theta join is the join condition. Theta joins combines tuples from different relations provided they satisfy the theta condition.

Notation:

$R1 \bowtie_{\theta} R2$

R1 and R2 are relations with their attributes $A1, A2, .., An$ and $B1, B2, .., Bn$ such that no attribute matches that is R1 ∩ R2 = Φ Here θ is condition in form of set of conditions C.

Theta join can use all kinds of comparison operators.

Student

SID	Name	Std
101	Alex	10
102	Maria	11

[*Table: Student Relation*]

Subjects

Class	Subject
10	Math
10	English
11	Music
11	Sports

[*Table: Subjects Relation*]

Student_Detail =

STUDENT $\bowtie_{Student.Std = Subject.Class}$ SUBJECT

Student_detail

SID	Name	Std	Class	Subject
101	Alex	10	10	Math
101	Alex	10	10	English
102	Maria	11	11	Music
102	Maria	11	11	Sports

[*Table: Output of theta join*]

Equi-Join

When Theta join uses only **equality** comparison operator it is said to be Equi-Join. The above example conrresponds to equi-join

Natural Join (⋈)

Natural join does not use any comparison operator. It does not concatenate the way Cartesian product does. Instead, Natural Join can only be performed if the there is at least one common attribute exists between relation. Those attributes must have same name and domain.

Natural join acts on those matching attributes where the values of attributes in both relation is same.

Courses

CID	Course	Dept
CS01	Database	CS
ME01	Mechanics	ME
EE01	Electronics	EE

[*Table: Relation Courses*]

HoD

Dept	Head
CS	Alex

ME	Maya
EE	Mira

[*Table: Relation HoD*]

Courses ⋈ HoD

Dept	CID	Course	Head
CS	CS01	Database	Alex
ME	ME01	Mechanics	Maya
EE	EE01	Electronics	Mira

[*Table: Relation Courses ⋈ HoD*]

Outer Joins

All joins mentioned above, that is Theta Join, Equi Join and Natural Join are called inner-joins. An inner-join process includes only tuples with matching attributes, rest are discarded in resulting relation. There exists methods by which all tuples of any relation are included in the resulting relation.

There are three kinds of outer joins:

Left outer join (R ⟕S)

All tuples of Left relation, R, are included in the resulting relation and if there exists tuples in R without any matching tuple in S then the S-attributes of resulting relation are made NULL.

Left

A	B
100	Database
101	Mechanics
102	Electronics

[*Table: Left Relation*]

Right

A	B
100	Alex
102	Maya
104	Mira

[*Table: Right Relation*]

Courses ⟕HoD

A	B	C	D
100	Database	100	Alex
101	Mechanics	---	---
102	Electronics	102	Maya

[*Table: Left outer join output*]

Right outer join: (R ⋈S)

All tuples of the Right relation, S, are included in the resulting relation and if there exists tuples in S without any matching tuple in R then the R-attributes of resulting relation are made NULL.

Courses ⋈HoD

A	B	C	D
100	Database	100	Alex
102	Electronics	102	Maya
---	---	104	Mira

[*Table: Right outer join output*]

Full outer join: (R ⋈S)

All tuples of both participating relations are included in the resulting relation and if there no matching tuples for both relations, their respective unmatched attributes are made NULL.

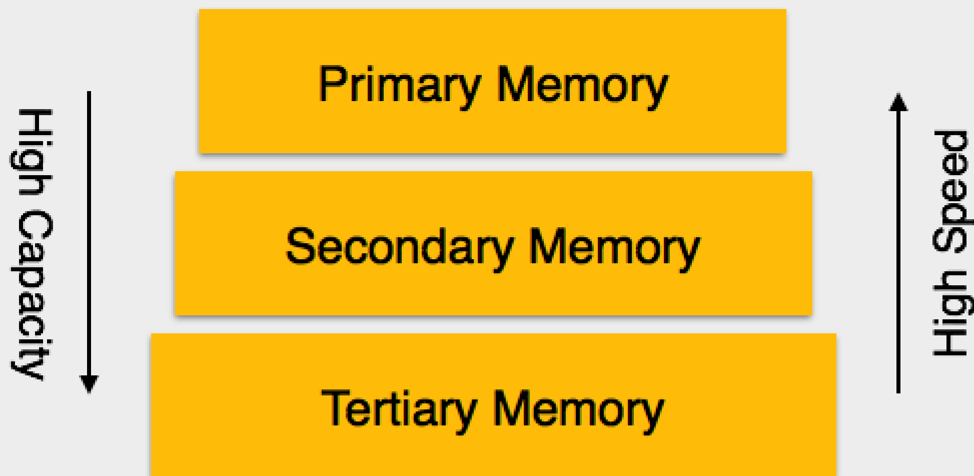
Courses ⋈HoD

A	B	C	D
100	Database	100	Alex
101	Mechanics	---	---
102	Electronics	102	Maya
---	---	104	Mira

[*Table: Full outer join output*]

DBMS - Storage System

Databases are stored in file formats, which contain records. At physical level, actual data is stored in electromagnetic format on some device capable of storing it for a longer amount of time. These storage devices can be broadly categorized in three types:



[*Image: Memory Types*]

- **Primary Storage:** The memory storage, which is directly accessible by the CPU, comes under this category. CPU's internal memory *registers*, fast memory *cache* and main memory *RAM* are directly accessible to CPU as they all are placed on the motherboard or CPU chipset. This storage is typically very small, ultra fast and volatile. This storage needs continuous power supply in order to maintain its state, i.e. in case of power failure all data are lost.
- **Secondary Storage:** The need to store data for longer amount of time and to retain it even after the power supply is interrupted gave birth to secondary data storage. All memory devices, which are not part of CPU chipset or motherboard comes under this category. Broadly, magnetic disks, all optical disks *DVD*,*CDetc*., flash drives and magnetic tapes are not directly accessible by the CPU. Hard disk drives, which contain the operating system and generally not removed from the computers are, considered secondary storage and all other are called tertiary storage.
- **Tertiary Storage:** Third level in memory hierarchy is called tertiary storage. This is used to store huge amount of data. Because this storage is external to the computer system, it is the slowest in speed. These storage devices are mostly used to backup the entire system. Optical disk and magnetic tapes are widely used storage devices as tertiary storage.

Memory Hierarchy

A computer system has well-defined hierarchy of memory. CPU has inbuilt registers, which saves data being operated on. Computer system has main memory, which is also directly accessible by CPU. Because the access time of main memory and CPU speed varies a lot, to minimize the loss cache memory is introduced. Cache memory contains most recently used data and data which may be referred by CPU in near future.

The memory with fastest access is the costliest one and is the very reason of hierarchy of memory system. Larger storage offers slow speed but can store huge amount of data compared to CPU registers or Cache memory and these are less expensive.

Magnetic Disks

Hard disk drives are the most common secondary storage devices in present day computer systems. These are called magnetic disks because it uses the concept of magnetization to store information. Hard disks consist of metal disks coated with magnetizable material. These disks are placed vertically a spindle. A read/write head moves in between the disks and is used to magnetize or de-magnetize the spot under it. Magnetized spot can be recognized as 0 *zero* or 1 *one*.

Hard disks are formatted in a well-defined order to stored data efficiently. A hard disk plate has many concentric circles on it, called tracks. Every track is further divided into sectors. A sector on a hard disk typically stores 512 bytes of data.

RAID

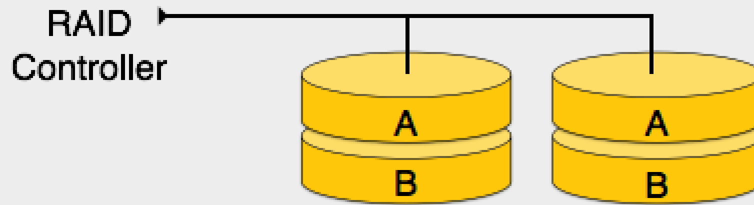
Exponential growth in technology evolved the concept of larger secondary storage medium. To mitigate the requirement RAID is introduced. RAID stands for Redundant Array of Independent Disks, which is a technology to connect multiple secondary storage devices and make use of them as a single storage media.

RAID consists an array of disk in which multiple disks are connected together to achieve different goals. RAID levels define the use of disk arrays.

- **RAID 0:** In this level a striped array of disks is implemented. The data is broken down into blocks and all blocks are distributed among all disks. Each disk receives a block of data to write/read in parallel. This enhances the speed and performance of storage device. There is no parity and backup in Level 0.

[*Image: RAID 0*]

- **RAID 1:** This level uses mirroring techniques. When data is sent to RAID controller it sends a copy of data to all disks in array. RAID level 1 is also called mirroring and provides 100% redundancy in case of failure.

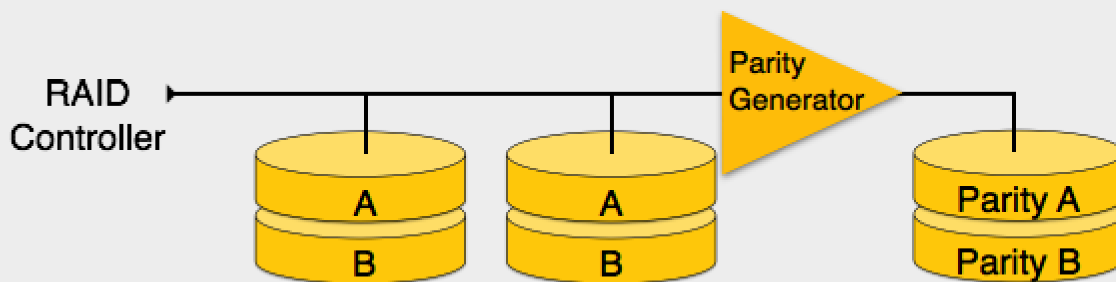


[*Image: RAID 1*]

- **RAID 2:** This level records the Error Correction Code using Hamming distance for its data striped on different disks. Like level 0, each data bit in a word is recorded on a separate disk and ECC codes of the data words are stored on different set disks. Because of its complex structure and high cost, RAID 2 is not commercially available.

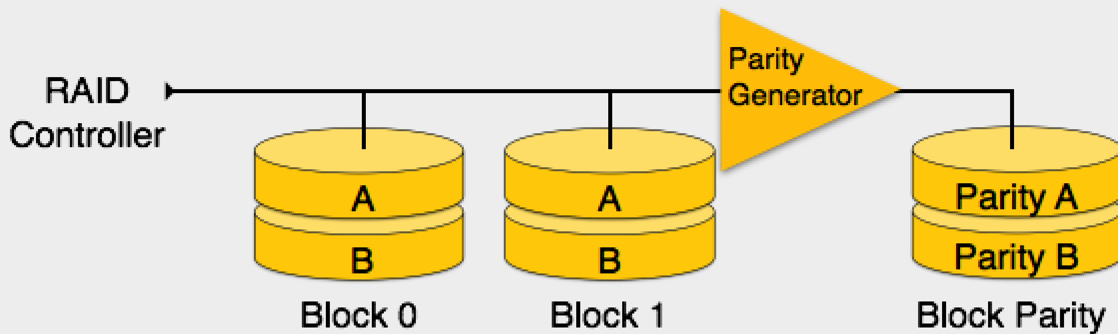
[*Image: RAID 2*]

- **RAID 3:** This level also stripes the data onto multiple disks in array. The parity bit generated for data word is stored on a different disk. This technique makes it to overcome single disk failure and a single disk failure does not impact the throughput.



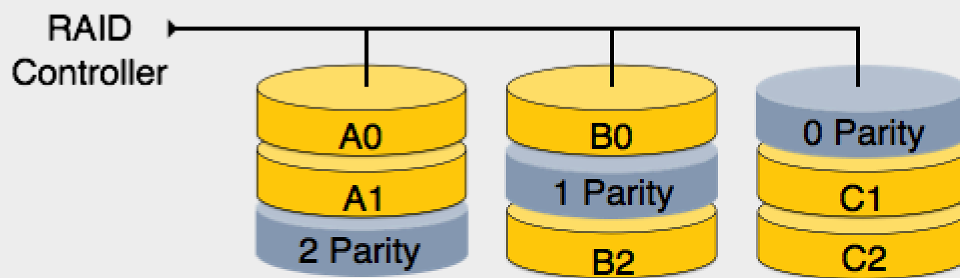
[*Image: RAID 3*]

- **RAID 4:** In this level an entire block of data is written onto data disks and then the parity is generated and stored on a different disk. The prime difference between level 3 and 4 is, level 3 uses byte level striping whereas level 4 uses block level striping. Both level 3 and 4 requires at least 3 disks to implement RAID.



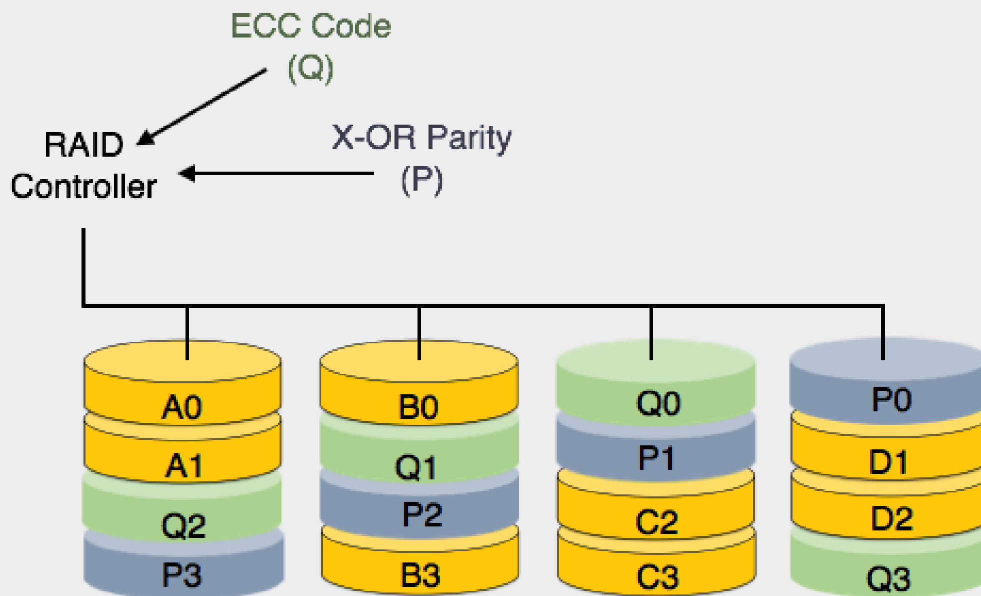
[*Image: RAID 4*]

- **RAID 5:** This level also writes whole data blocks on to different disks but the parity generated for data block stripe is not stored on a different dedicated disk, but is distributed among all the data disks.



[*Image: RAID 5*]

- **RAID 6:** This level is an extension of level 5. In this level two independent parities are generated and stored in distributed fashion among disks. Two parities provide additional fault tolerance. This level requires at least 4 disk drives to be implemented.



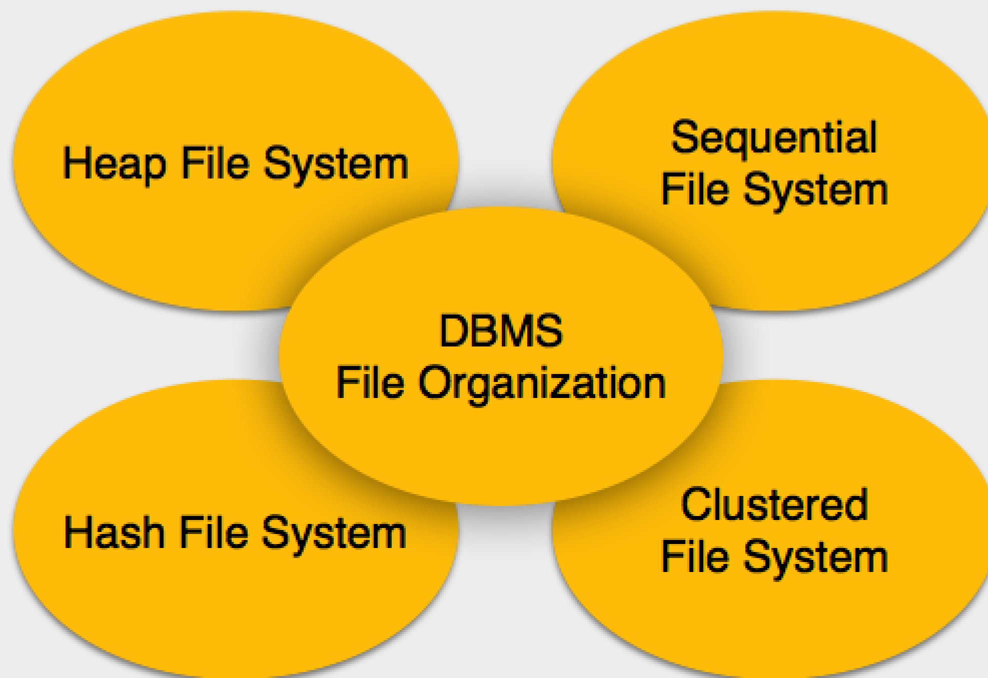
[*Image: RAID 6*]

DBMS - File Structure

Relative data and information is stored collectively in file formats. A file is sequence of records stored in binary format. A disk drive is formatted into several blocks, which are capable for storing records. File records are mapped onto those disk blocks.

File Organization

The method of mapping file records to disk blocks defines file organization, i.e. how the file records are organized. The following are the types of file organization



[*Image: File Organization*]

- **Heap File Organization**: When a file is created using Heap File Organization mechanism, the Operating Systems allocates memory area to that file without any further accounting details. File records can be placed anywhere in that memory area. It is the responsibility of software to manage the records. Heap File does not support any ordering, sequencing or indexing on its own.
- **Sequential File Organization**: Every file record contains a data field *attribute* to uniquely identify that record. In sequential file organization mechanism, records are placed in the file in the some sequential order based on the unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form.
- **Hash File Organization**: This mechanism uses a Hash function computation on some field of the records. As we know, that file is a collection of records, which has to be mapped on some block of the disk space allocated to it. This mapping is defined that the

hash computation. The output of hash determines the location of disk block where the records may exist.

- **Clustered File Organization**: Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in a same disk block, that is, the ordering of records is not based on primary key or search key. This organization helps to retrieve data easily based on particular join condition. Other than particular join condition, on which data is stored, all queries become more expensive.

File Operations

Operations on database files can be classified into two categories broadly.

- **Update Operations**
- **Retrieval Operations**

Update operations change the data values by insertion, deletion or update. Retrieval operations on the other hand do not alter the data but retrieve them after optional conditional filtering. In both types of operations, selection plays significant role. Other than creation and deletion of a file, there could be several operations, which can be done on files.

- **Open**: A file can be opened in one of two modes, read mode or write mode. In read mode, operating system does not allow anyone to alter data it is solely for reading purpose. Files opened in read mode can be shared among several entities. The other mode is write mode, in which, data modification is allowed. Files opened in write mode can be read also but cannot be shared.
- **Locate**: Every file has a file pointer, which tells the current position where the data is to be read or written. This pointer can be adjusted accordingly. Using find *seek* operation it can be moved forward or backward.
- **Read**: By default, when files are opened in read mode the file pointer points to the beginning of file. There are options where the user can tell the operating system to where the file pointer to be located at the time of file opening. The very next data to the file pointer is read.
- **Write**: User can select to open files in write mode, which enables them to edit the content of file. It can be deletion, insertion or modification. The file pointer can be located at the time of opening or can be dynamically changed if the operating system allowed doing so.
- **Close**: This also is most important operation from operating system point of view. When a request to close a file is generated, the operating system removes all the locks *if in shared mode* and saves the content of data *if altered* to the secondary storage media and release all the buffers and file handlers associated with the file.

The organization of data content inside the file plays a major role here. Seeking or locating the file pointer to the desired record inside file behaves differently if the file has records arranged sequentially or clustered, and so on.

DBMS - Indexing

We know that information in the DBMS files is stored in form of records. Every record is equipped with some key field, which helps it to be recognized uniquely.

Indexing is a data structure technique to efficiently retrieve records from database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to the one we see in books.

Indexing is defined based on its indexing attributes. Indexing can be one of the following types:

- **Primary Index:** If index is built on ordering 'key-field' of file it is called Primary Index. Generally it is the primary key of the relation.
- **Secondary Index:** If index is built on non-ordering field of file it is called Secondary Index.
- **Clustering Index:** If index is built on ordering non-key field of file it is called Clustering Index.

Ordering field is the field on which the records of file are ordered. It can be different from primary or candidate key of a file.

Ordered Indexing is of two types:

- Dense Index
- Sparse Index

Dense Index

In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index record contains search key value and a pointer to the actual record on the disk.

[*Image: Dense Index*]

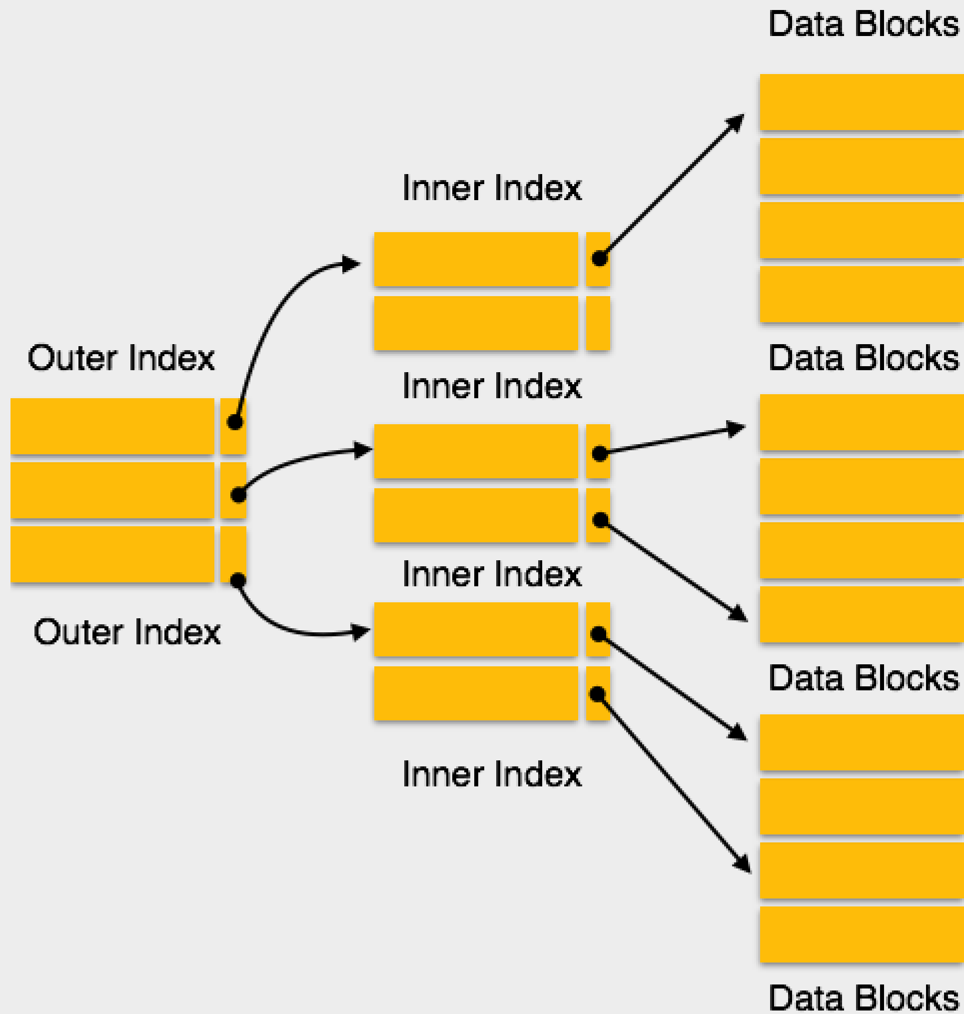
Sparse Index

In sparse index, index records are not created for every search key. An index record here contains search key and actual pointer to the data on the disk. To search a record we first proceed by index record and reach at the actual location of the data. If the data we are looking for is not where we directly reach by following index, the system starts sequential search until the desired data is found.

[*Image: Sparse Index*]

Multilevel Index

Index records are comprised of search-key value and data pointers. This index itself is stored on the disk along with the actual database files. As the size of database grows so does the size of indices. There is an immense need to keep the index records in the main memory so that the search can speed up. If single level index is used then a large size index cannot be kept in memory as whole and this leads to multiple disk accesses.



[*Image: Multi-level Index*]

Multi-level Index helps breaking down the index into several smaller indices in order to make the outer most level so small that it can be saved in single disk block which can easily be accommodated anywhere in the main memory.

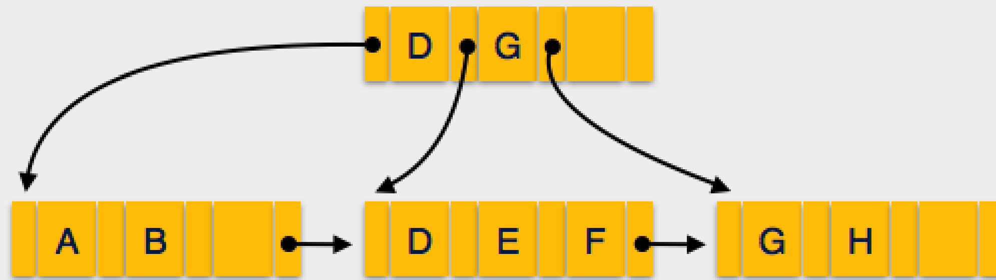
B^+ Tree

B<sup+< sup=""> tree is multi-level index format, which is balanced binary search trees. As mentioned earlier single level index records becomes large as the database size grows, which also degrades performance.</sup+<>

All leaf nodes of B^+ tree denote actual data pointers. B^+ tree ensures that all leaf nodes remain at the same height, thus balanced. Additionally, all leaf nodes are linked using link list, which makes B^+ tree to support random access as well as sequential access.

Structure of B^+ tree

Every leaf node is at equal distance from the root node. A B^+ tree is of order n where n is fixed for every B^+ tree.



[*Image: B^+ tree*]

Internal nodes:

- Internal *non−leaf* nodes contains at least ⌈n/2⌉ pointers, except the root node.
- At most, internal nodes contain n pointers.

Leaf nodes:

- Leaf nodes contain at least ⌈n/2⌉ record pointers and ⌈n/2⌉ key values
- At most, leaf nodes contain n record pointers and n key values
- Every leaf node contains one block pointer P to point to next leaf node and forms a linked list.

B^+ tree insertion

- B^+ tree are filled from bottom. And each node is inserted at leaf node.
- **If leaf node overflows:**
  - Split node into two parts
  - Partition at i = $\lfloor m+1_{/2} \rfloor$
  - First i entries are stored in one node
  - Rest of the entries $i+1 onwards$ are moved to a new node
  - i^{th} key is duplicated in the parent of the leaf
- **If non-leaf node overflows:**
  - Split node into two parts
  - Partition the node at i = $\lceil m+1_{/2} \rceil$
  - Entries upto i are kept in one node
  - Rest of the entries are moved to a new node

B^+ tree deletion

- B^+ tree entries are deleted leaf nodes.
- The target entry is searched and deleted.

    - If it is in internal node, delete and replace with the entry from the left position.
- After deletion underflow is tested
    - If underflow occurs
        - Distribute entries from nodes left to it.
    - If distribution from left is not possible
        - Distribute from nodes right to it
    - If distribution from left and right is not possible
        - Merge the node with left and right to it.

DBMS - Hashing

For a huge database structure it is not sometime feasible to search index through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate direct location of data record on the disk without using index structure.

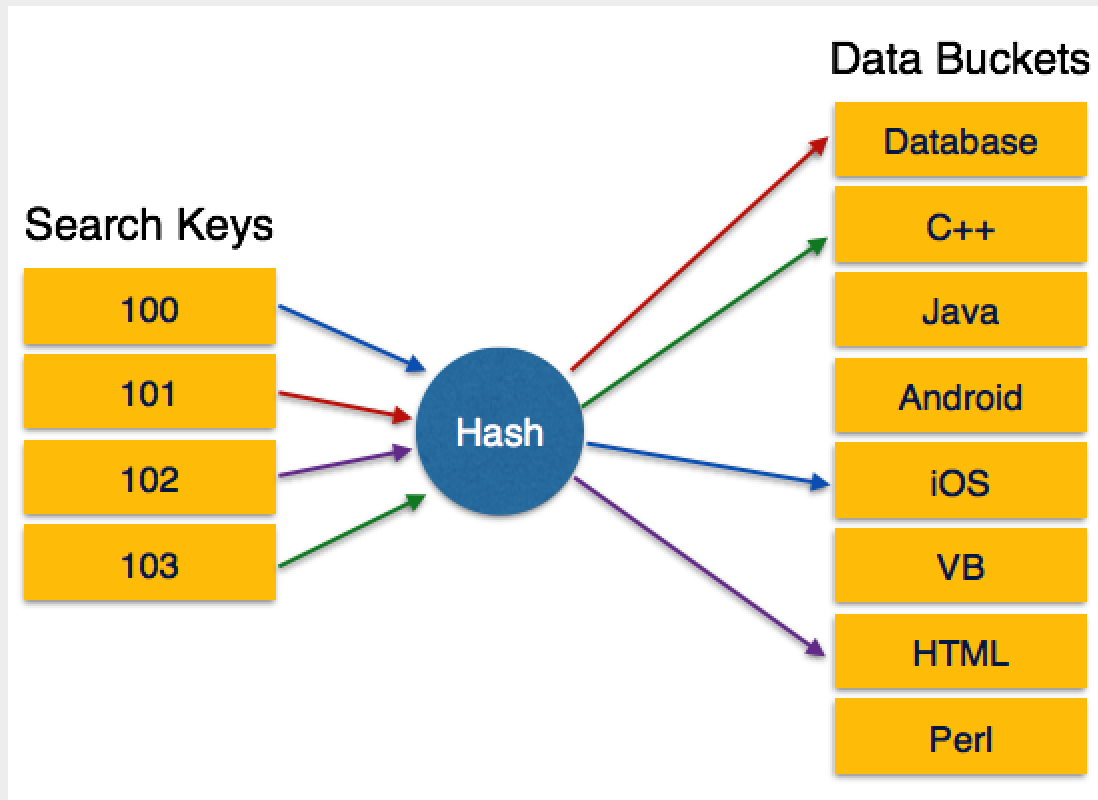
It uses a function, called hash function and generates address when called with search key as parameters. Hash function computes the location of desired data on the disk.

Hash Organization

- **Bucket:** Hash file stores data in bucket format. Bucket is considered a unit of storage. Bucket typically stores one complete disk block, which in turn can store one or more records.
- **Hash Function:** A hash function h, is a mapping function that maps all set of search-keys K to the address where actual records are placed. It is a function from search keys to bucket addresses.

Static Hashing

In static hashing, when a search-key value is provided the hash function always computes the same address. For example, if mod-4 hash function is used then it shall generate only 5 values. The output address shall always be same for that function. The numbers of buckets provided remain same at all times.



[*Image: Static Hashing*]

Operation:

- **Insertion:** When a record is required to be entered using static hash, the hash function h, computes the bucket address for search key K, where the record will be stored.

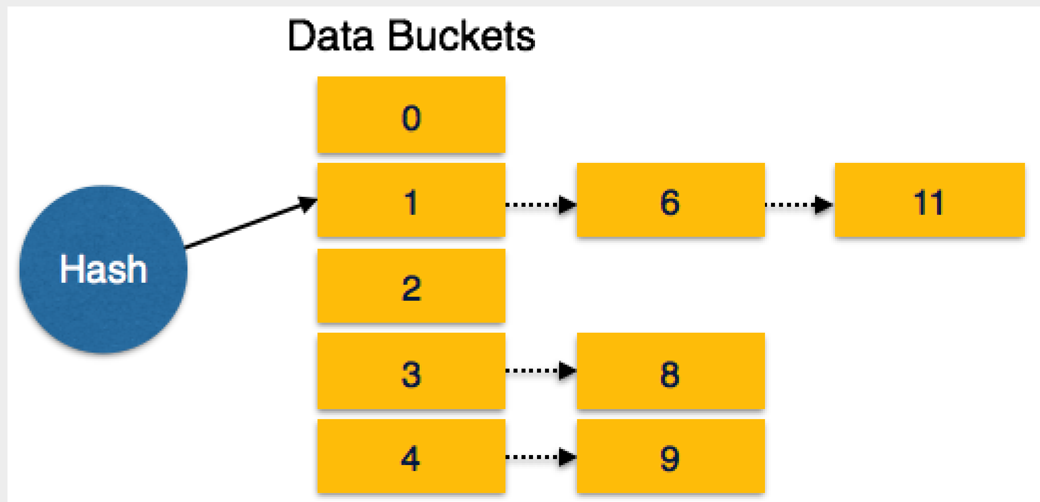
  Bucket address = h*K*

- **Search:** When a record needs to be retrieved the same hash function can be used to retrieve the address of bucket where the data is stored.
- **Delete:** This is simply search followed by deletion operation.

Bucket Overflow:

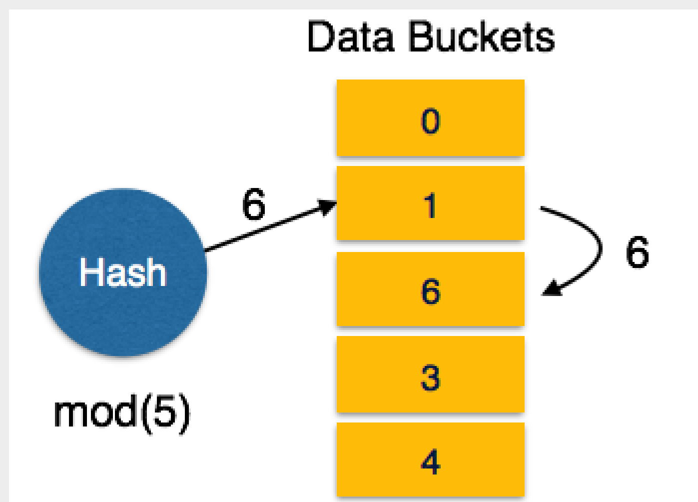
The condition of bucket-overflow is known as collision. This is a fatal state for any static hash function. In this case overflow chaining can be used.

- **Overflow Chaining:** When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called Closed Hashing.



[*Image: Overflow chaining*]

- **Linear Probing:** When hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called Open Hashing.



[*Image: Linear Probing*]

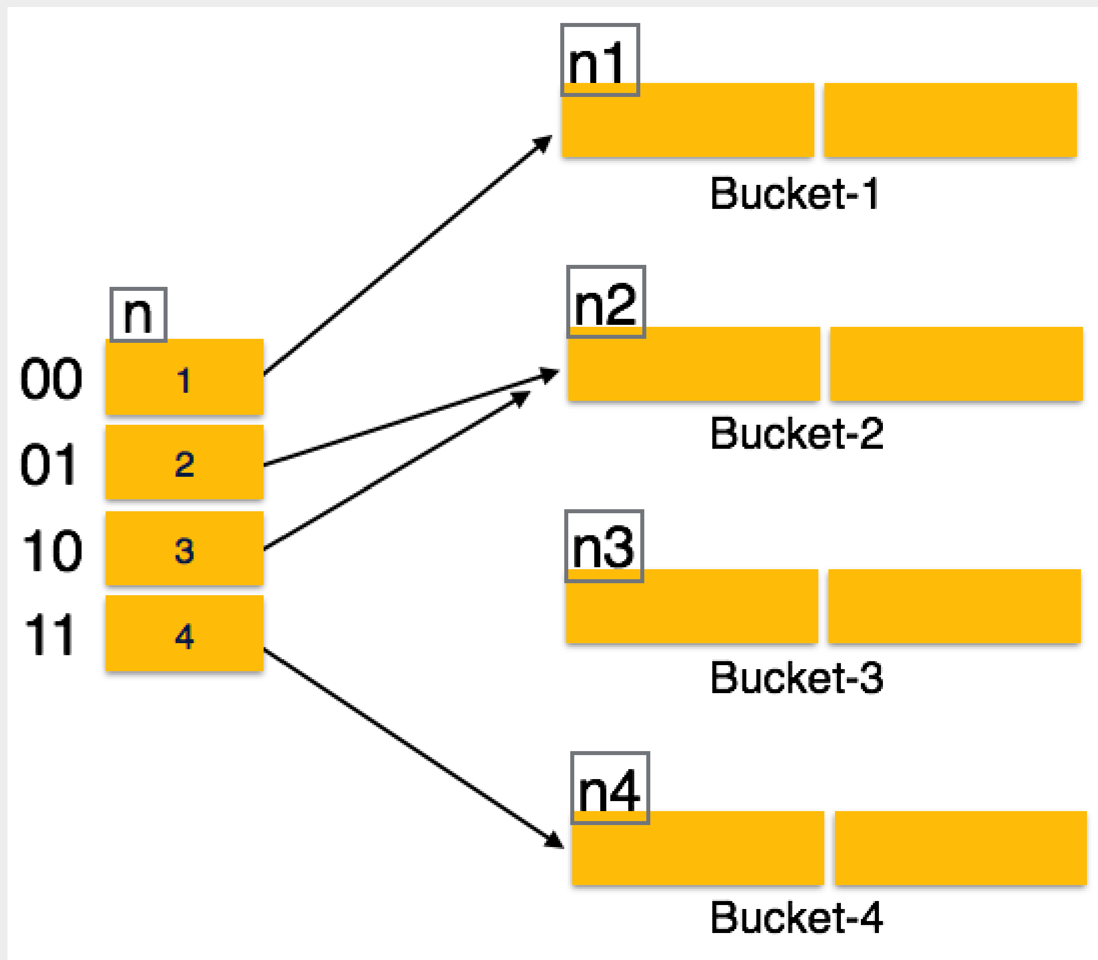
For a hash function to work efficiently and effectively the following must match:

- Distribution of records should be uniform
- Distribution should be random instead of any ordering

Dynamic Hashing

Problem with static hashing is that it does not expand or shrink dynamically as the size of database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as extended hashing.

Hash function, in dynamic hashing, is made to produce large number of values and only a few are used initially.



[*Image: Dynamic Hashing*]

Organization

The prefix of entire hash value is taken as hash index. Only a portion of hash value is used for computing bucket addresses. Every hash index has a depth value, which tells it how many bits are used for computing hash function. These bits are capable to address 2n buckets. When all these bits are consumed, that is, all buckets are full, then the depth value is increased linearly and twice the buckets are allocated.

Operation

- **Querying:** Look at the depth value of hash index and use those bits to compute the bucket address.
- **Update:** Perform a query as above and update data.
- **Deletion:** Perform a query to locate desired data and delete data.
- **Insertion:** compute the address of bucket

- If the bucket is already full
    - Add more buckets
    - Add additional bit to hash value
    - Re-compute the hash function
- Else
    - Add data to the bucket
- If all buckets are full, perform the remedies of static hashing.

Hashing is not favorable when the data is organized in some ordering and queries require range of data. When data is discrete and random, hash performs the best.

Hashing algorithm and implementation have high complexity than indexing. All hash operations are done in constant time.

DBMS - Transaction

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

An example of transaction can be bank accounts of two users, say A & B. When a bank employee transfers amount of Rs. 500 from A's account to B's account, a number of tasks are executed behind the screen. This very simple and small transaction includes several steps: decrease A's bank account from 500

```
Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)
```

In simple words, the transaction involves many tasks, such as opening the account of A, reading the old balance, decreasing the 500 from it, saving new balance to account of A and finally closing it. To add amount 500 in B's account same sort of tasks need to be done:

```
Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)
```

A simple transaction of moving an amount of 500 from A to B involves many low level tasks.

ACID Properties

A transaction may contain several low level tasks and further a transaction is very small unit of any program. A transaction in a database system must maintain some properties in order to ensure the accuracy of its completeness and data integrity. These properties are refer to as ACID properties and are mentioned below:

- **Atomicity:** Though a transaction involves several low level operations but this property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in database where the transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.
- **Consistency:** This property states that after the transaction is finished, its database must remain in a consistent state. There must not be any possibility that some data is incorrectly affected by the execution of transaction. If the database was in a consistent state before the execution of the transaction, it must remain in consistent state after the execution of the transaction.
- **Durability:** This property states that in any case all updates made on the database will persist even if the system fails and restarts. If a transaction writes or updates some data in database and commits that data will always be there in the database. If the transaction commits but data is not written on the disk and the system fails, that data will be updated once the system comes up.
- **Isolation:** In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Serializability

When more than one transaction is executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transactions are interleaved with some other transaction.

- **Schedule:** A chronological execution sequence of transaction is called schedule. A schedule can have many transactions in it, each comprising of number of instructions/tasks.
- **Serial Schedule:** A schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle then next transaction is executed. Transactions are ordered one after other. This type of schedule is called serial schedule as transactions are executed in a serial manner.

In a multi-transaction environment, serial schedules are considered as benchmark. The execution sequence of instruction in a transaction cannot be changed but two transactions can have their instruction executed in random fashion. This execution does no harm if two transactions are mutually independent and working on different segment of data but in case these two

transactions are working on same data, results may vary. This ever-varying result may cause the database in an inconsistent state.

To resolve the problem, we allow parallel execution of transaction schedule if transactions in it are either serializable or have some equivalence relation between or among transactions.

Equivalence schedules: Schedules can equivalence of the following types:

- **Result Equivalence:**

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

- **View Equivalence:**

  Two schedules are view equivalence if transactions in both schedules perform similar actions in similar manner.

  For example:

  - If T reads initial data in S1 then T also reads initial data in S2
  - If T reads value written by J in S1 then T also reads value written by J in S2
  - If T performs final write on data value in S1 then T also performs final write on data value in S2
- **Conflict Equivalence:**

  Two operations are said to be conflicting if they have the following properties:

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

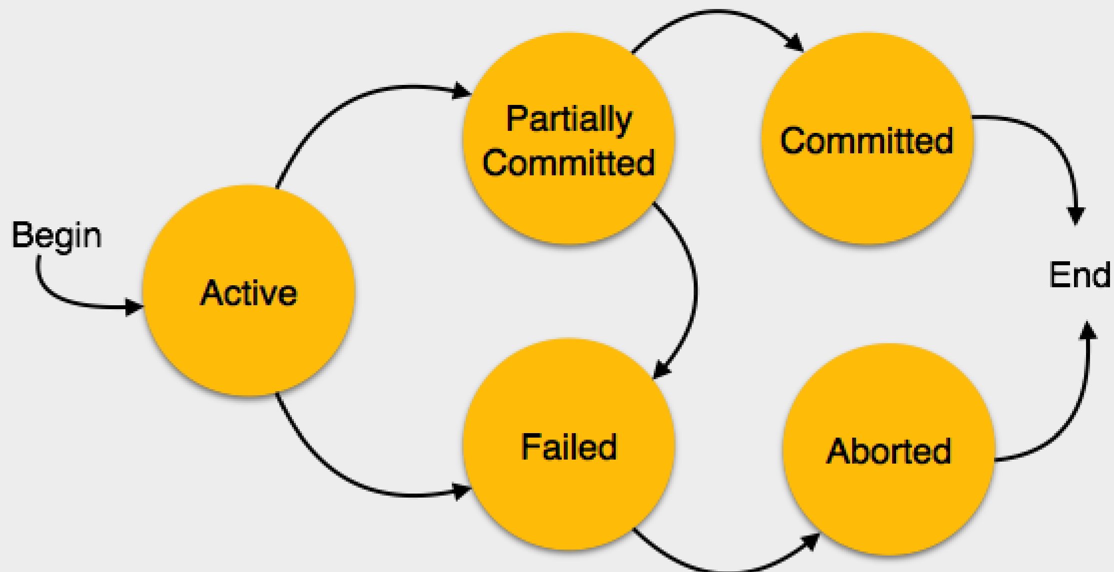
Two schedules have more than one transactions with conflicting operations are said to be conflict equivalent if and only if:

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

View equivalent schedules are view serializable and conflict equivalent schedules are conflict serializable. All conflict serializable schedules are view serializable too.

States of Transactions:

A transaction in a database can be in one of the following state:



[*Image: Transaction States*]

- **Active:** In this state the transaction is being executed. This is the initial state of every transaction.
- **Partially Committed:** When a transaction executes its final operation, it is said to be in this state. After execution of all operations, the database system performs some checks e.g. the consistency state of database after applying output of transaction onto the database.
- **Failed:** If any checks made by database recovery system fail, the transaction is said to be in failed state, from where it can no longer proceed further.
- **Aborted:** If any of checks fails and transaction reached in failed state, the recovery manager rolls back all its write operation on the database to make database in the state where it was prior to start of execution of transaction. Transactions in this state are called aborted. Database recovery module can select one of the two operations after a transaction aborts:
    - Re-start the transaction
    - Kill the transaction
- **Committed:** If transaction executes all its operations successfully it is said to be committed. All its effects are now permanently made on database system.

DBMS - Concurrency Control

In a multiprogramming environment where more than one transactions can be concurrently executed, there exists a need of protocols to control the concurrency of transaction to ensure atomicity and isolation properties of transactions.

Concurrency control protocols, which ensure serializability of transactions, are most desirable. Concurrency control protocols can be broadly divided into two categories:

- Lock based protocols
- Time stamp based protocols

Lock based protocols

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

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

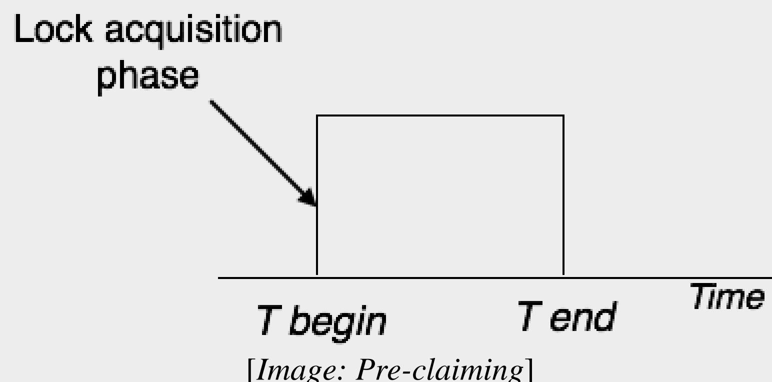
There are four types lock protocols available:

- **Simplistic**

  Simplistic lock based protocols allow transaction to obtain lock on every object before 'write' operation is performed. As soon as 'write' has been done, transactions may unlock the data item.

- **Pre-claiming**

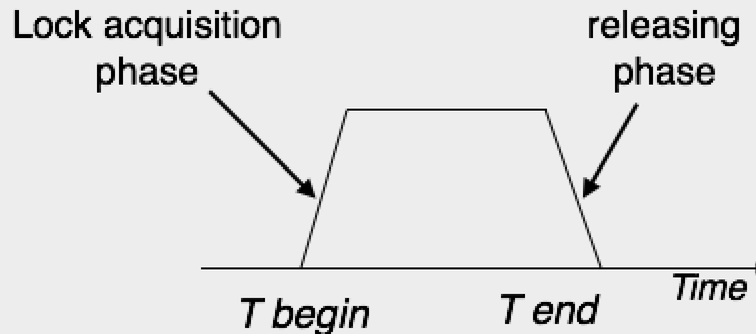
  In this protocol, a transactions evaluations its operations and creates a list of data items on which it needs locks. Before starting the execution, transaction requests the system for all locks it needs beforehand. If all the locks are granted, the transaction executes and releases all the locks when all its operations are over. Else if all the locks are not granted, the transaction rolls back and waits until all locks are granted.



[*Image: Pre-claiming*]

- **Two Phase Locking - 2PL**

This locking protocol is divides transaction execution phase into three parts. In the first part, when transaction starts executing, transaction seeks grant for locks it needs as it executes. Second part is where the transaction acquires all locks and no other lock is required. Transaction keeps executing its operation. As soon as the transaction releases its first lock, the third phase starts. In this phase a transaction cannot demand for any lock but only releases the acquired locks.



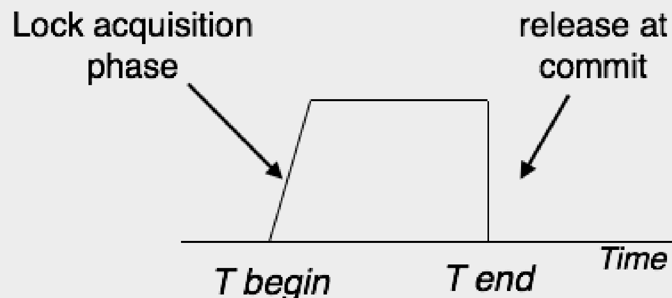
[*Image: Two Phase Locking*]

Two phase locking has two phases, one is growing; where all locks are being acquired by transaction and second one is shrinking, where locks held by the transaction are being released.

To claim an exclusive *write* lock, a transaction must first acquire a shared *read* lock and then upgrade it to exclusive lock.

- **Strict Two Phase Locking**

The first phase of Strict-2PL is same as 2PL. After acquiring all locks in the first phase, transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release lock as soon as it is no more required, but it holds all locks until commit state arrive. Strict-2PL releases all locks at once at commit point.



[*Image: Strict Two Phase Locking*]

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

Time stamp based protocols

The most commonly used concurrency protocol is time-stamp based protocol. This protocol uses either system time or logical counter to be used as a time-stamp.

Lock based protocols manage the order between conflicting pairs among transaction at the time of execution whereas time-stamp based protocols start working as soon as transaction is created.

Every transaction has a time-stamp associated with it and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transaction, which come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and priority may be given to the older one.

In addition, every data item is given the latest read and write-timestamp. This lets the system know, when last read was and write operation made on the data item.

Time-stamp ordering protocol

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

- Time-stamp of Transaction Ti is denoted as TS (T_i).
- Read time-stamp of data-item X is denoted by R-timestamp*X*.
- Write time-stamp of data-item X is denoted by W-timestamp*X*.

Timestamp ordering protocol works as follows:

- **If a transaction Ti issues read*X* operation:**
  - If TS*Ti* < W-timestamp*X*
    - Operation rejected.
  - If TS*Ti* >= W-timestamp*X*
    - Operation executed.
  - All data-item Timestamps updated.

- **If a transaction Ti issues write*X* operation:**
  - If TS*Ti* < R-timestamp*X*
    - Operation rejected.
  - If TS*Ti* < W-timestamp*X*
    - Operation rejected and Ti rolled back.
  - Otherwise, operation executed.

Thomas' Write rule:

This rule states that in case of:

- If TS*Ti* < W-timestamp*X*
-
- Operation rejected and Ti rolled back. Timestamp ordering rules can be modified to make the schedule view serializable. Instead of making Ti rolled back, the 'write' operation itself is ignored.

DBMS - Deadlock

In a multi-process system, deadlock is a situation, which arises in shared resource environment where a process indefinitely waits for a resource, which is held by some other process, which in turn waiting for a resource held by some other process.

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

Deadlock is not a good phenomenon for a healthy system. To keep system deadlock free few methods can be used. In case the system is stuck because of deadlock, either the transactions involved in deadlock are rolled back and restarted.

Deadlock Prevention

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations which transactions are about to execute. DBMS inspects operations and analyze if they can create a deadlock situation. If it finds that a deadlock situation might occur then that transaction is never allowed to be executed.

There are deadlock prevention schemes, which uses time-stamp ordering mechanism of transactions in order to pre-decide a deadlock situation.

Wait-Die Scheme:

In this scheme, if a transaction request to lock a resource *dataitem*, which is already held with conflicting lock by some other transaction, one of the two possibilities may occur:

- If $TS(T_i) < TS(T_j)$, that is T_i, which is requesting a conflicting lock, is older than T_j, T_i is allowed to wait until the data-item is available.

- If $TS(T_i) > TS(t_j)$, that is T_i is younger than T_j, T_i dies. T_i is restarted later with random delay but with same timestamp.

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

Wound-Wait Scheme:

In this scheme, if a transaction request to lock a resource *dataitem*, which is already held with conflicting lock by some other transaction, one of the two possibilities may occur:

- If $TS(T_i) < TS(T_j)$, that is T_i, which is requesting a conflicting lock, is older than T_j, T_i forces T_j to be rolled back, that is T_i wounds T_j. T_j is restarted later with random delay but with same timestamp.
- If $TS(T_i) > TS(T_j)$, that is T_i is younger than T_j, T_i is forced to wait until the resource is available.

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

In both cases, transaction, which enters late in the system, is aborted.

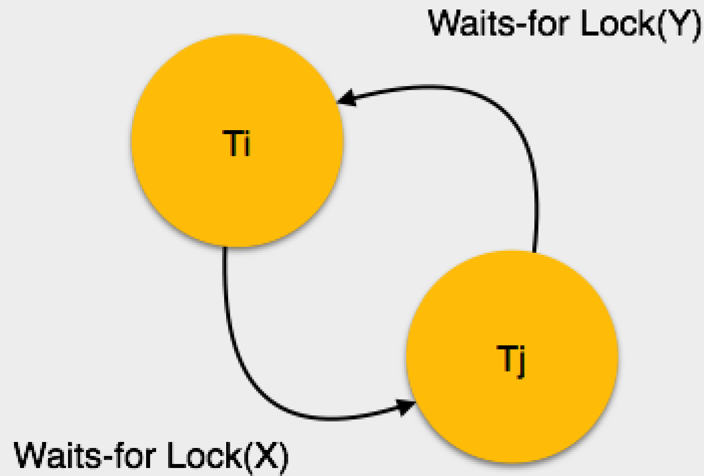
Deadlock Avoidance

Aborting a transaction is not always a practical approach. Instead deadlock avoidance mechanisms can be used to detect any deadlock situation in advance. Methods like "wait-for graph" are available but for the system where transactions are light in weight and have hold on fewer instances of resource. In a bulky system deadlock prevention techniques may work well.

Wait-for Graph

This is a simple method available to track if any deadlock situation may arise. For each transaction entering in the system, a node is created. When transaction T_i requests for a lock on item, say X, which is held by some other transaction T_j, a directed edge is created from T_i to T_j. If T_j releases item X, the edge between them is dropped and T_i locks the data item.

The system maintains this wait-for graph for every transaction waiting for some data items held by others. System keeps checking if there's any cycle in the graph.



[*Image: Wait-for Graph*]

Two approaches can be used, first not to allow any request for an item, which is already locked by some other transaction. This is not always feasible and may cause starvation, where a transaction indefinitely waits for data item and can never acquire it. Second option is to roll back one of the transactions.

It is not feasible to always roll back the younger transaction, as it may be important than the older one. With help of some relative algorithm a transaction is chosen, which is to be aborted, this transaction is called victim and the process is known as ***victim selection.***

DBMS - Data Backup

Failure with loss of Non-Volatile storage

What would happen if the non-volatile storage like RAM abruptly crashes? All transaction, which are being executed are kept in main memory. All active logs, disk buffers and related data is stored in non-volatile storage.

When storage like RAM fails, it takes away all the logs and active copy of database. It makes recovery almost impossible as everything to help recover is also lost. Following techniques may be adopted in case of loss of non-volatile storage.

- A mechanism like checkpoint can be adopted which makes the entire content of database be saved periodically.
- State of active database in non-volatile memory can be dumped onto stable storage periodically, which may also contain logs and active transactions and buffer blocks.
- <dump> can be marked on log file whenever the database contents are dumped from non-volatile memory to a stable one.

Recovery:

- When the system recovers from failure, it can restore the latest dump.
- It can maintain redo-list and undo-list as in checkpoints.
- It can recover the system by consulting undo-redo lists to restore the state of all transaction up to last checkpoint.

Database backup & recovery from catastrophic failure

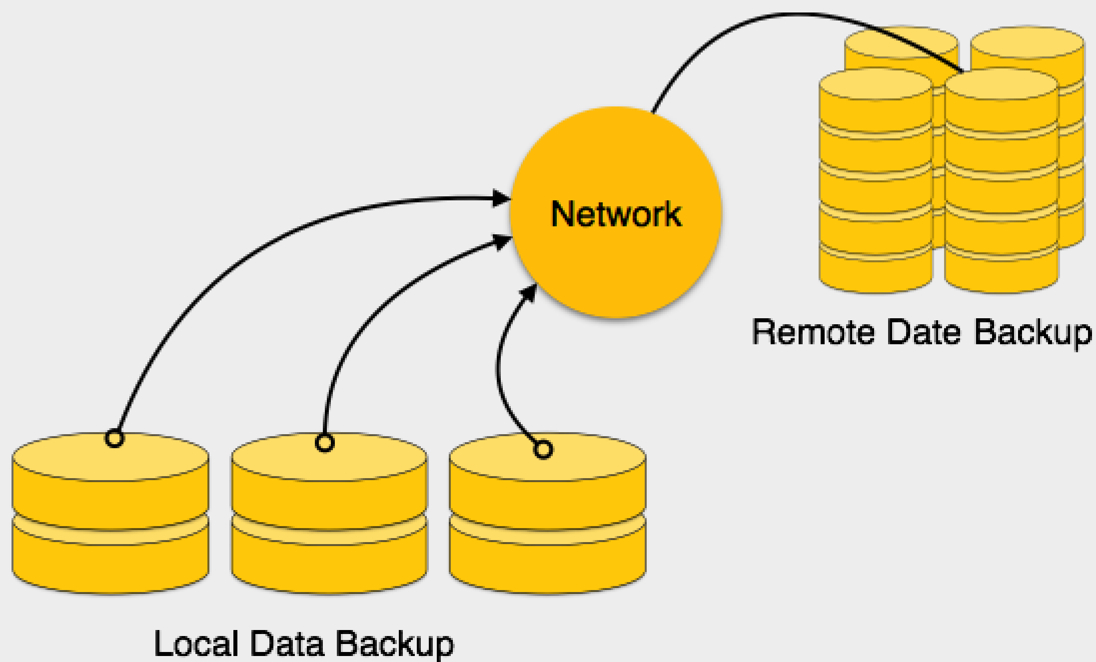
So far we have not discovered any other planet in our solar system, which may have life on it, and our own earth is not that safe. In case of catastrophic failure like alien attack, the database administrator may still be forced to recover the database.

Remote backup, described next, is one of the solutions to save life. Alternatively, whole database backups can be taken on magnetic tapes and stored at a safer place. This backup can later be restored on a freshly installed database and bring it to the state at least at the point of backup.

Grown up databases are too large to be frequently backed-up. Instead, we are aware of techniques where we can restore a database by just looking at logs. So backup of logs at frequent rate is more feasible than the entire database. Database can be backed-up once a week and logs, being very small can be backed-up every day or as frequent as every hour.

Remote Backup

Remote backup provides a sense of security and safety in case the primary location where the database is located gets destroyed. Remote backup can be offline or real-time and online. In case it is offline it is maintained manually.



[*Image: Remote Data Backup*]

Online backup systems are more real-time and lifesavers for database administrators and investors. An online backup system is a mechanism where every bit of real-time data is backed-up simultaneously at two distant place. One of them is directly connected to system and other one is kept at remote place as backup.

As soon as the primary database storage fails, the backup system sense the failure and switch the user system to the remote storage. Sometimes this is so instant the users even can't realize a failure.

DBMS - Data Recovery

Crash Recovery

Though we are living in highly technologically advanced era where hundreds of satellite monitor the earth and at every second billions of people are connected through information technology, failure is expected but not every time acceptable.

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

Failure Classification

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

Transaction failure

When a transaction is failed to execute or it reaches a point after which it cannot be completed successfully it has to abort. This is called transaction failure. Where only few transaction or process are hurt.

Reason for transaction failure could be:

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

System crash

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

Examples may include operating system errors.

Disk failure:

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

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

Storage Structure

We have already described storage system here. In brief, the storage structure can be divided in various categories:

- **Volatile storage:** As name suggests, this storage does not survive system crashes and mostly placed very closed to CPU by embedding them onto the chipset itself for examples: main memory, cache memory. They are fast but can store a small amount of information.

- **Nonvolatile storage:** These memories are made to survive system crashes. They are huge in data storage capacity but slower in accessibility. Examples may include hard disks, magnetic tapes, flash memory, non-volatile *battery backed up* RAM.

Recovery and Atomicity

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

When DBMS recovers from a crash it should maintain the following:

- It should check the states of all transactions, which were being executed.
- A transaction may be in the middle of some operation; DBMS must ensure the atomicity of transaction in this case.
- It should check whether the transaction can be completed now or needs to be rolled back.
- No transactions would be allowed to left DBMS in inconsistent state.

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

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

Log-Based Recovery

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

Log based recovery works as follows:

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

  $<T_n, Start>$

- When the transaction modifies an item X, it write logs as follows:

  $<T_n, X, V_1, V_2>$

  It reads Tn has changed the value of X, from V1 to V2.

- When transaction finishes, it logs:

  <T_n, commit>

Database can be modified using two approaches:

1. **Deferred database modification:** All logs are written on to the stable storage and database is updated when transaction commits.
2. **Immediate database modification:** Each log follows an actual database modification. That is, database is modified immediately after every operation.

Recovery with concurrent transactions

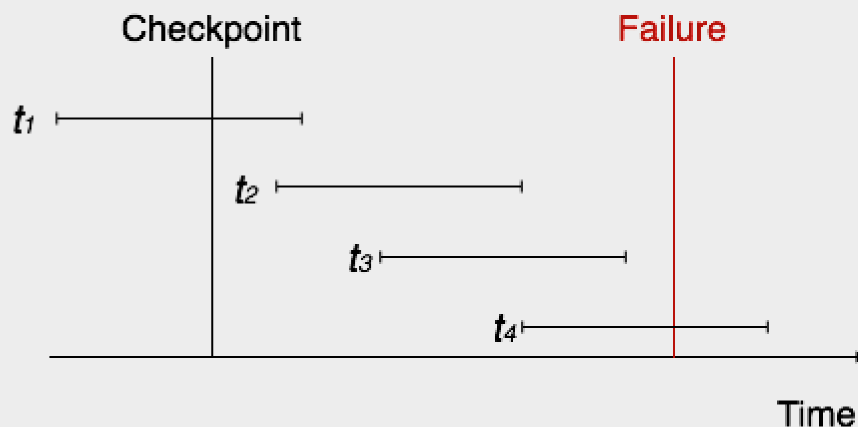
When more than one transactions are being executed in parallel, the logs are interleaved. At the time of recovery it would become hard for recovery system to backtrack all logs, and then start recovering. To ease this situation most modern DBMS use the concept of 'checkpoints'.

Checkpoint

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

Recovery

When system with concurrent transaction crashes and recovers, it does behave in the following manner:



[*Image: Recovery with concurrent transactions*]

- The recovery system reads the logs backwards from the end to the last Checkpoint.
- It maintains two lists, undo-list and redo-list.

- If the recovery system sees a log with <T_n, Start> and <T_n, Commit> or just <T_n, Commit>, it puts the transaction in redo-list.
- If the recovery system sees a log with <T_n, Start> but no commit or abort log found, it puts the transaction in undo-list.

All transactions in undo-list are then undone and their logs are removed. All transaction in redo-list, their previous logs are removed and then redone again and log saved.