A database management system (DBMS) is a collection of interrelated data and a set of programs to access those data.

The collection of data, usually referred to as the database.

The information it stores is usually about an enterprise or company.

The primary goal of a DBMS is to provide a way to store & retrieve db information that is both convenient and efficient.

DB systems are designed to manage large bodies of information.

Managing the db involves many things like defining structures to store the information, providing mechanisms to manipulate the infor. Apart from these data must be protected from unauthorized access & system crashes. When data is shared by two or more users, the db should be consistent enough to be used further.

Database System Applications:

- **Banking** – Here the information is about customers and their access. It also involves the transactions made by the users.

- **Airlines** – These are the first to use databases in a geographically distributed manner. They maintain information about world-wide terminals and their schedules, their booking & reservations.

- **Universities** – These contain information about colleges & their offered courses & student information.

- **Credit card transactions** – These maintain data about the purchases made by the card holders & send monthly statements.

- **Telecommunications** – Keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, & storing infor about communication networks.

- **Finance** – stores information about sales, purchases of financial information such as stocks & bonds.

- **Sales** – Maintains information about customer, sales, & purchase.

- Manufacturing:- Information about the needs of goods, no of products to be manufacturing product store, warehouse etc.

- Human resources: Maintains infor about employees, salaries, payroll taxes and benefits and for generation of pay checks.

Database Systems versus File Systems:

To explain the difference between the two, we consider the example of banking. To maintain infor we choose the operating system files. Now, the users retrieve information with the facilities provided. To facilitate different users, some application programs are written. They are:
- to debit & credit the account
- create a new account
- Balance enquiry
- To generate monthly statements.

These can facilitate the users to retrieve information from the database. Now, if more facilities are to be provided, then the num of application programs increase. If the no. of programs increase the no. of files to be maintained also increases, which becomes tedious for a computer to maintain.

Such file processing system has many disadvantages:

1) Data Redundancy & inconsistency:

A customer information can be maintained in different files (i-e) he can have a saving account and a loan account. Now, information about customer (phone no, address) is maintained in saving account file and loan account file. This is duplicate data (i-e) the same information is maintained in two files. This leads to Data Redundancy.

When we change the address of the customer in savings account only, then it leads to Data inconsistency.

2) Difficulty in accessing data:

Suppose, a list of all the customers who have an account balance of 1,00,000/- is required. Actually, an application program to retrieve such information is not devised, So, the officer may get the list of all the customers with different account balances. Then, either he has to pick up the names manually or he can ask the programmers to write a program that can retrieve such information.

If any request for another list is there, then again another program has to be devised for the new request.

3) <u>Data Isolation:</u>

Because data is scattered among different files & all the files are in different formats, to retrieve the appropriate information new application programs must be written, which is difficult.

4) <u>Integrity Problems:</u>

The data stored in the db must satisfy some consistency constraints. Eg: the minimum balance of a savings account must be 300/-. This constraint is enforced in the code of application programs.

If the constraints are to be changed, then making changes in the code is difficult. If the constraints involve the data items in diff files, then it would be very difficult.

5) <u>Atomicity Problems:</u> The operations or transactions made by the customers should be atomic i.e operation must be done in its entirety. It shouldn't be done to the half or incomplete.

eg: Transfer money 200/- from A/C-A to A/C-B. During the sum of money transfer application program if the system crashes (i.e) money is debited from A/c - A but not credited to A/c - B. This leads to database inconsistency.

~~The funds transfer must be atomic~~.

6) <u>Concurrent access anamolies:</u>

Suppose, an account A/c : A has a balance of 1000/- . If two persons withdraw an amount of 100/- and 200/- concurrently, the balance that is reflected to the account may not be correct, (i.e) both the persons may take 1000/- as balance and reflect the current balance as 900/- & 800/-, which is not correct. The actual balance after withdrawls should be 700/-. So, supervising such concurrent actions is not possible with file systems.

7) <u>Security Problems:</u>

Every user of the db system should not be able to access all the data.

Ex: In a banking sys, payroll personnel need to see only that part of the db that has information about the various bank employees. They do not need access to information about customer accounts.

All these difficulties lead to the evolution of db systems.

View of Data:

A db system is a collection of interrelated files and a set of programs that allow users to access and modify these files.

A major purpose of a db system is to provide users with an abstract view of the data.

(a) → Data Abstraction

Any db system must receive data efficiently. After receiving the data, it must be stored efficiently. To store in an appropriate manner, complex data structures are used, which is not understood by many of the users. For this reason, the details of data storage is hidden from the users. To do this, developers use several levels of abstraction.

(a) Physical level: The lowest level of abstraction describes how the data are actually stored.

(b) Logical level: This is the next higher level of abstraction that describes what data is stored in the db. It also defines what relationships exist among the data. DB Administrator uses this level to decide what data to store in the db.

(c) View Level: The highest level of abstraction that describes only part of the db. A user may need only a part of the db. So, he is given only that view. A system can provide many views for the ~~db~~ same db.

Eg: If we create a record called customer, it has four fields like identity, name, address. Like this, we can create many records with their corresponding attributes or fields.

employee - emp_name, salary

Account - A/c no, balance

Now the relationship between the records can be like,

At physical level various records with their fields are stored in terms of words (or) bytes. This is actually hidden from the user.

At logical level the relationship between the records is defined. The datatypes of various fields of a records are also described. Mostly programmers work at this level of abstraction.

A view level set of application programs are provided to the user. Not only supplying different views of the db, but security to some of the views is given.

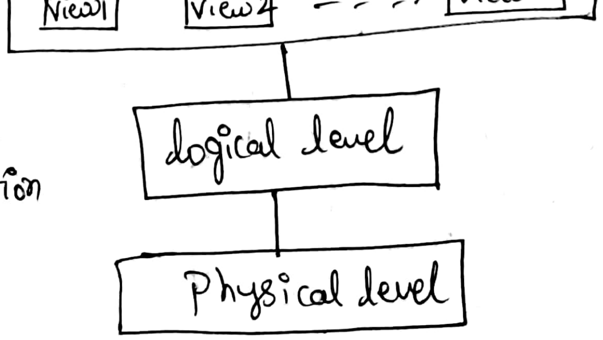


fig: 3 levels of data abstraction

(b) Instances and Schemas:

whenever, data is inserted and deleted, the db ~~d~~ changes. The collection of information stored in the db at a particular moment of time is called as instance of the db.

The overall design of the db is called as the schema
↳ These are not changed frequently.

With respect to a prog language, a schema refers to the declarations of variables.

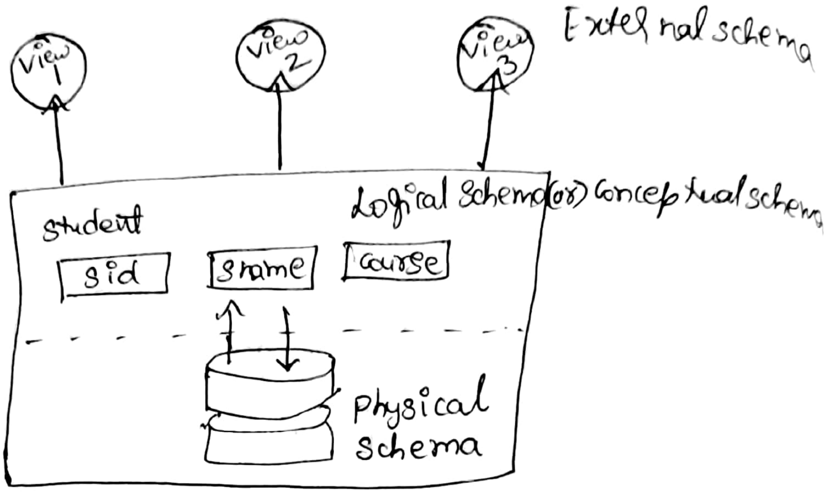
when the variable is given a value at any point of time, it is referred to as the instance of the variable.

A db schema can be divided broadly into ~~two~~ three categories:

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

— External Schema — (view schema)

Design of db at view level is called View schema or External schema. This generally describes end user interaction with db systems.



Eg: Schema —

sid	sname	DOB

template for a table.

Instances —

sid	sname	DOB
101	XYZ	1-3-1990
102	Abc	2-6-1990

Data-filled tables

Data Independence:

A db sys normally contains a lot of data in addition to users data. For ex, it stores data about data, known as metadata, to locate & retrieve data easily, It is rather difficult to modify or update a set of metadata once it is stored in the db. But as a DBMS expands, it needs to change over time to satisfy the requirements of the users. If the entire data is dependent, it would become a tedious & highly complex job.

Scanned by CamScanner

Logical Data Independence

Logical schema

---

physical schema

physical data Independence

Metadata itself follows a layered architecture, so that when we change data at one layer, it does not affect the data at another level. This data is independent but mapped to each other.

→ Logical data Independence :-

The ability to change the logical schema without changing the external schema or application programs is called as Logical data Independence.

ex: The addition or removal of new entities, attributes, or relationships to the conceptual schema should be possible without having to change external schemas or having to rewrite existing application programs.

→ Physical Data Independence - Ability to change the physical schema without changing the logical schema is called as Physical Data Independence.

Changes in physical schema may include:

- Using new storage devices
- " diff data structures
- switching from one access method to another
- Using diff file organisations
- etc

Data Models:

Data Models define how the logical structure of a db is modeled. Data Models are fundamental entities to introduce abstraction in a DBMS.

Data models define how data is connected to each other & how they are processed & stored inside the system.

Two data models are mostly used:

1. Entity - Relationship model
2. Relational model

1. Entity-Relationship Model:- (An entity can be a real world object)

ER model consists of basic objects called Entities. It also consists of relationships among those objects.

ex: In a school db - student, teachers, classes are considered as entities.

(Property of an entity) All these entities have some attributes or properties that give them their identity.

All attributes have values.

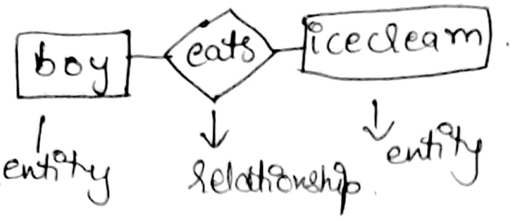
ex: a student entity may have name, class & age as attributes.

The association among entities is called a relationship.

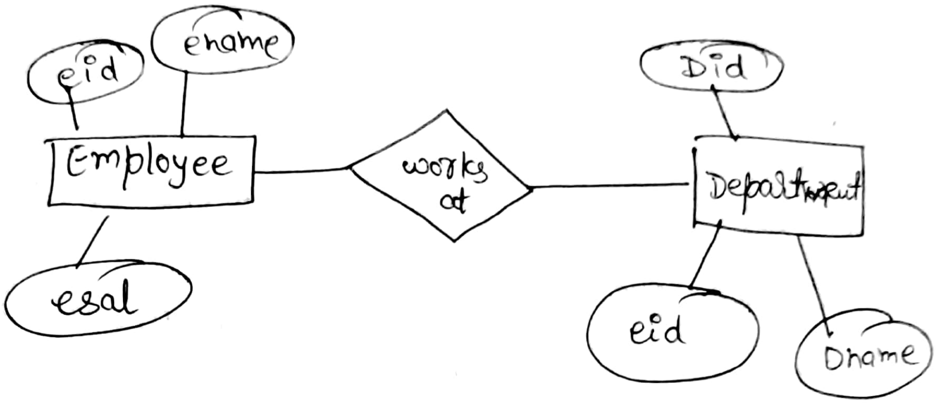
ex: a student enrolls in a course.

enrolls is a relationship.

ex:



ex:



Relationships are mapped with entities in various ways. Mapping cardinalities define the num of association between two entities.

Mapping cardinalities —

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

② Relational Model:

The relational model uses a collection of tables to represent both data and the relationships among those data.

Each table has multiple columns, and each column has a unique name.

Describing the data in terms of data model is called as the schema.

In a relational model, the schema for a relation (this is defined as set of records) specifies its name & the name of each field or attribute & the type of each field.

ex: student (sid : string, sname : string)

Course (cid : number, cname : string)

This is the schema given for college db.

Each relation is represented using a table with a set of records.

Because of this only relational model is called a record based model

attributes (or) field names (or) cols

ex:

sid	sname
101	Abc

rows (or) tuples

Student relation

Course relation

cid	cname
5	cp

- The Relational model is most widely used.
- But this is at a lower level of abstraction than the ER model.
- Many dbs are often designed in ER model & then translated to relational model.

Other Data Models:

- **Object Oriented Data Model:**

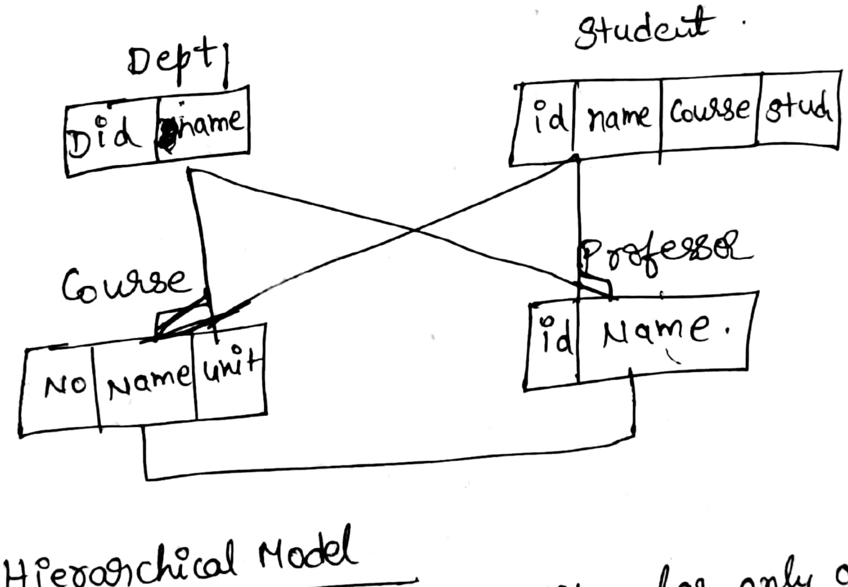
It is an extention to the ER Model with encapsulation, methods (functions) and object identity.

- **Object Relational Data Model:**

This combines the features of object-oriented and relational models

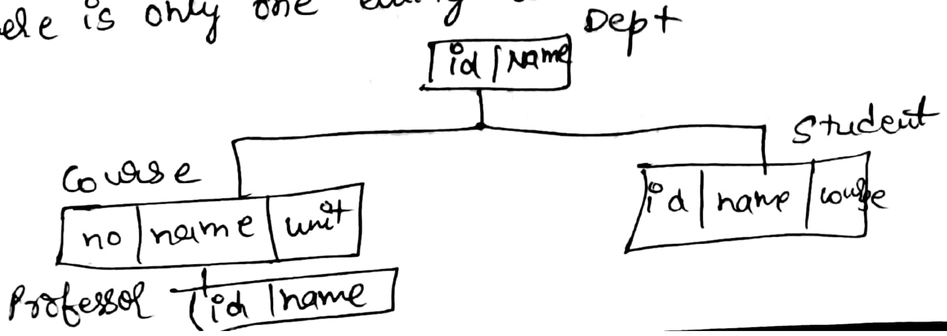
- **Network data Model:**

In the n/w model, entities are organised in a graph, in which some entities can be accessed through several path.



- **Hierarchical Model**

In this model each entity has only one parent but can have several children. At the top of hierarchy there is only one entity which is called Root.



Note: The n/w model & heirarchial model didn't hide the underlying implementation much. So, they are used infrequently.

Among all these models Relational Model is widely used model for databases.

Database Users and Administrators:

Many people are associated with the creation & use of the db. Usually, they are categorized as ~~that~~ database users & Database Administrators (DBA).

→ Database Users & User Interfaces:

Depending upon the way they interact, the users are divided into 4 types. Each type of user has his own interface to interact with the system.

(a) Naive Users:

Naive users are unsophisticated users who interact with the system by invoking any one of the application progs which are already written.

The interface used by them is a forms interface, where the user fill in appropriate fields of the form.

Eg: Transfering amount from one acct to another.

A user who wants to transfer amount from A/c-A to A/c-B, he invokes the application program called transfer. This, when executed shows a form with fields, that the user has to fill in. The form has fields for the amount to be transferred. Then it has fields for source & target accounts.

(b)

w

(H

d

p

4

(c)

w

c

(b) Application Programmers:

They are computer professionals who write application progs to interact with the system. They choose many tools to develop a program.

RAD - Rapid Application Development tools are mostly used to develop forms and reports without writing a program.

There are some special types of programming langs that can combine the control structures (ex for, while, if loops) with the manipulation stmts. Such languages are called as fourth-generation languages.

(c) Sophisticated Users:

They interact with the system without writing programs. They retrieve data using queries through db query lang. The queries are submitted to a query processor → transforms DML stmts to instructions that storage manager understands.

↓ analyst.

An analyst uses OLAP - Online Analytical Processing tools, which enable him to view the data in different ways. Another tool is data mining tool, which help the analyst to find certain kinds of data.

(d) Specialized Users:

They write specialized db applications that is not part of traditional data-processing frame work.

Eg: computer aided design systems, Knowledge base & expert systems & environment-modeling systems (graphics, audio-data etc).

→ Database Administrator:

The function of DBMS is to have central control of data. The person who has control of the entire system is Database Administrator.

The functions of DBA are:

(a) → Schema definition:

DBA creates the original db schema by using some stmts in the data definition language.

(b) → Storage Structure and Access method definition

(c) → Schema & Physical Organization Modification.

The changes to the schema & physical organization depending upon the requirements & to improve performance.

(d) Granting Access to the data & Authorization: 1-12

DBA is responsible for ensuring that unauthorized access is not allowed. So, for this diff. types of authorizations are granted. By doing this, DBA can regulate which parts of the db, a user can access.

This authorization information is kept in a special system structure that the db system consults whenever some user tries to access the system.

(e) Routine Maintenance:

- Periodically backup the db. To prevent loss of data in case of disasters such as flooding.
- Ensure enough free space is available for normal operations & upgrade disk space as required.
- Monitor the jobs running & ensure that performance is not degraded.

(f) Database tuning:

This is changing the organization of data according to the requirements of users.

Transaction Management:

(collection of operations that form a single logical unit of work)

Multiple operations on the db form a single logical unit of work. All the operations have to be performed in its entirety.

Eg: Money transfer among accounts.

To do this, money has to be debited from source Account A and credited to the target account. It is essential that either both credit & debit occur, or neither occur. This [redacted] all or none requirement of operations is called as atomicity.

During execution of operations, it is necessary to preserve consistency of db. After execution, the db must conform durability, in case of system failure (i.e after successful execution of a funds transfer the new values of accounts A & B must persist, despite the possibility of sys failure.

∴ A transaction is a collection of operations that perform single logical function in a db application. It is a unit of Atomicity & Consistency.

Structure of DBMS (DB System Architecture)

A db system is divided into modules that deal with each of the responsibilities of the overall system. They are Storage Manager and Query Processor.

Storage Manager is a very important component of db. Databases usually require large amount of memory to be stored. But the main memory of a computer is not that large enough. So, data is stored in secondary devices. They are moved to main memory when the necessity arises. But frequent movement of data from the secondary devices or to the devices is not advisable. So, we need to structure the data in such a way that movement of data is minimized. To do this, a storage manager is required.

Query Processor helps to simplify the access to data. For this, high level views of data is given: (1.14)

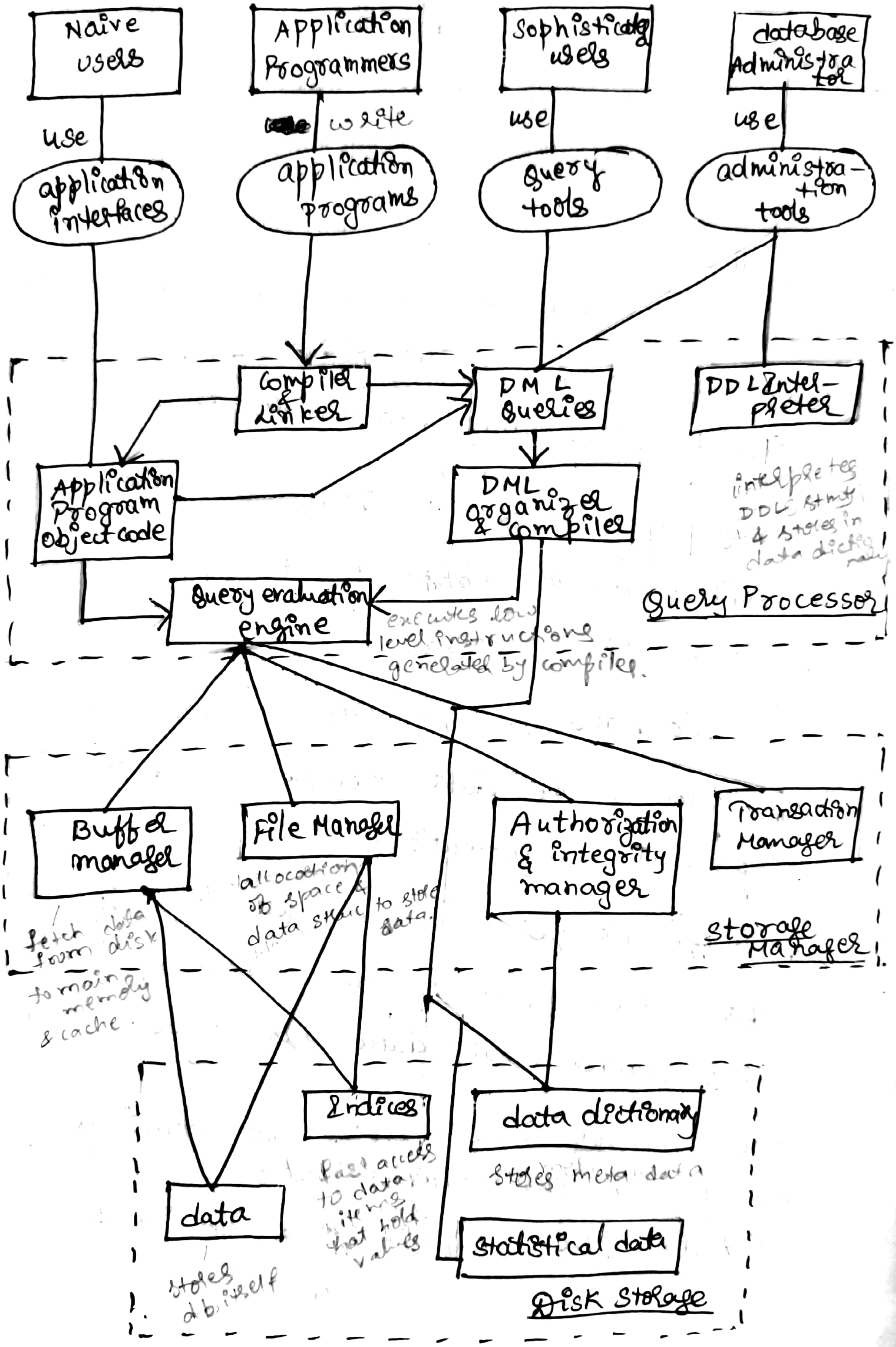


fig: DB system structure

Storage Manager:

It is an interface between the application programs & queries, and the low-level data stored in the db. It is responsible for interaction with the file systems.

The storage Manager translates the high level DML stmts to the low-level file-system commands. So he is responsible for storing, retrieving & updating data in the db.

The components of storage Manager are:

a) Authorization & integrity Manager.

b) Transaction Manager

c) File Manager: This take care of allocation of space & the data structures used to represent information.

d) Buffer Manager: This is used to fetch data from disk to main memory. It also decides what data has to be cached.

Storage Manager uses several data structures for the physical data implementation

1) Data files: This stores db itself

2) Data Dictionary: This stores metadata i.e about the structure of the db (1.15)

3) Indices: Provides fast access to data items that hold values.

Query Processor:

The components of query processor are:

(a) DDL interpreter:

This interprets the DDL stmts & records them in the data dictionary.

(b) DML compiler:

This translates the DML stmts into an evaluation plan

↳ it consists of low level instructions that query evaluate engine understands.

The compiler also does query optimization:

↳ It selects the lowest cost evaluation plan from many alternatives generated by the compiler.

(c) Query evaluation Engine:

This executes the low-level instructions generated by the DML compiler.

Application Architecture:

Mostly database applications are partitioned into two or three parts. They are:

a) Two-tier Architecture:

Here, the application is partitioned into components of a client which invokes database system function using queries. User resides on client side & the db resides on server side. User communicates with the server using n/w.

user
↓
application } client
↓ network
server { database system

ODBC or JDBC are used for interaction between client & server.

b) Three-tier Architecture:

Here, the client does not contain any direct database calls. It is just like a front end. So, the user communicates with an application server, which in turn communicates with the database system.

Note: Application that run in a worldwideweb.

user
↓
application client } client
↓
server { application server
↓
database system

→ **Entity:**

An Entity is an object in the real world that is distinguishable from other objects.

ex: In school database students, teachers, classes, courses offered etc are entities.

→ **Attributes:**

Entities are represented by means of their properties called attributes.

ex: a student entity may have name, class & age as attributes.

(domain of the attribute is the set of permitted values)

There exist a domain or range of values that can be assigned to attributes.

ex: student name should not be numeric value.
student age cannot be negative, etc.

→ **Types of Attributes:**

- **Single valued Attributes:** An attribute that has a single value.

  ex: age of a person.
  DOB of a person.

Multivalued Attribute:

Multivalued attribute can have multiple values.

ex: - colors of a car
- a person may have multiple phone numbers.

1.19

- **Composite Attribute:** (compound Attribute)
  It can be divided into two or more other attribute.
  ex: Name can be divided into Firstname, middle name, last name
  Address can be divided into Flat no, street, city, district, state.

- **Simple Attribute / Atomic Attribute:**
  An attribute which cannot be divided into smaller subparts.
  ex: age

- **Stored Attribute:** An attribute, which cannot be derived from other attribute.
  ex: Date of birth.

- **Derived Attribute:** Attribute derived from other stored attribute.
  ex: age from DoB minus current date.

→ **Entity Set** is a collection of one or more entities. (An entity set is a set of entities of the same type that share the same properties).
ex: student — entity type → collection of entities having common attributes

id	Name	age
e1	Ram	18
e2	John	19
⋮	⋮	⋮
en	shyam	20

(row e2 → entity)

entity set → e1, e2, e3

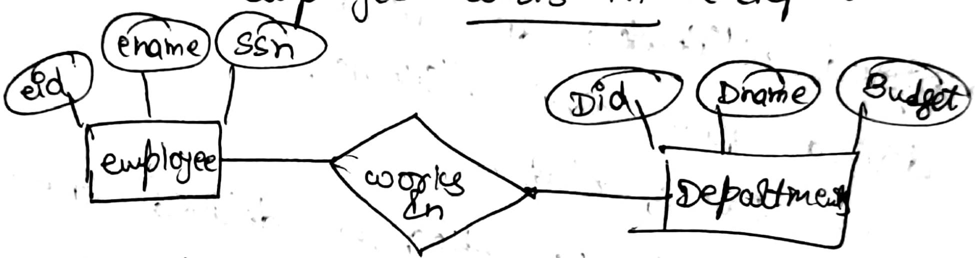
→ **key:** Key is an attribute or collection of attributes that uniquely identifies an entity among entity set.

→ **Relationship:**

It is an association among two or more entities.

eg: In an employee db employee is an entity & departments is an entity.

The relationship between these two entities is an employee works in a department.



→ **Relationship set:**

The collection of similar relationships is referred to as relationship set.

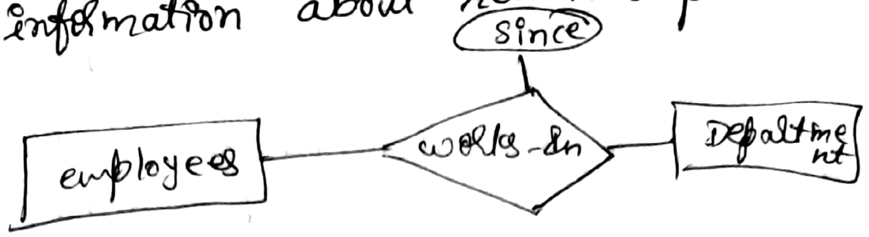
ex: – In the above fig relationship set Works In, in which each relationship indicates a department in which an employee works.

– Several relationship sets might involve the same entity sets.

ex: we could also have a Manages relationship set involving Employees & Departments.

- A relationship ~~set~~ can also have descriptive attributes. These descriptive attributes record the information about relationship.

- eg:



Here since is the attribute that maintains information about the experience in the dept of an employee.

Actually, the association between the entities is referred to an participation i.e entity sets $E_1, E_2, \ldots E_n$ participate in relationship set R.

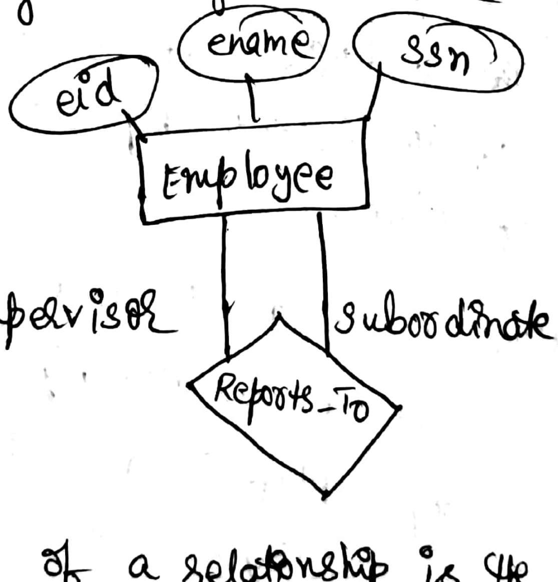
- A relationship instance is an association between the entities. It is nothing but a snapshot of the relationship set at some instance of time..

- Any entity is said to have some kind of participation in a relationship set. The fun that an entity plays in a relationship is called as the entity's role.

- If the entity sets participating in the relationship are distinct, then the roles need not be specified, but if the same entity sets have participation more than once in the relationship then their roles have to be specified.

- Such relationship set also called as recursive relationship set. The role names must be specified for the entity sets depending upon the participation. These names describe about the the participation function that the entity does in the relationship.

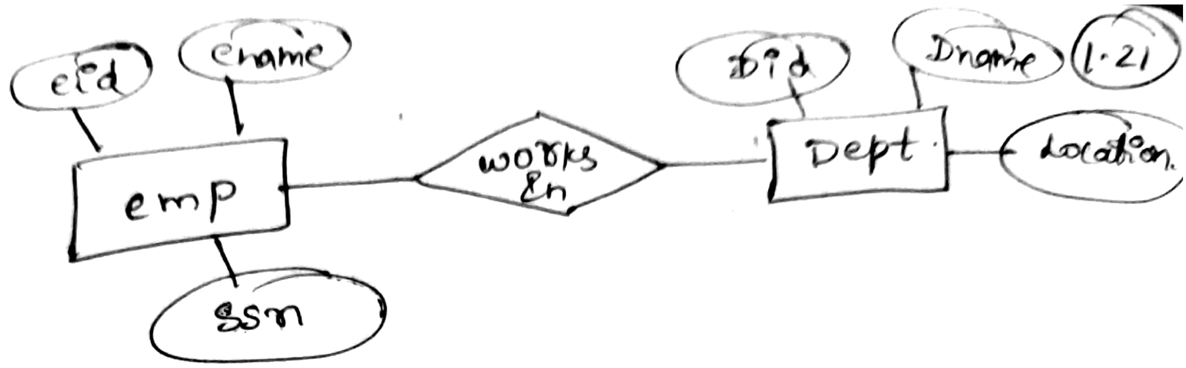
Eg: In employee db, an employee has to report his work to another employee. He may be a collegue or a higher authority. So, we develop a relationship 'Reports_To' that define whom an employee reports. The details of the person to whom the employee is reporting depends upon the role he is participating. It is given with role indicators.



The degree of a relationship is the num of entity types that participate in the relationship.

A binary relationship is one that involves two entity sets.

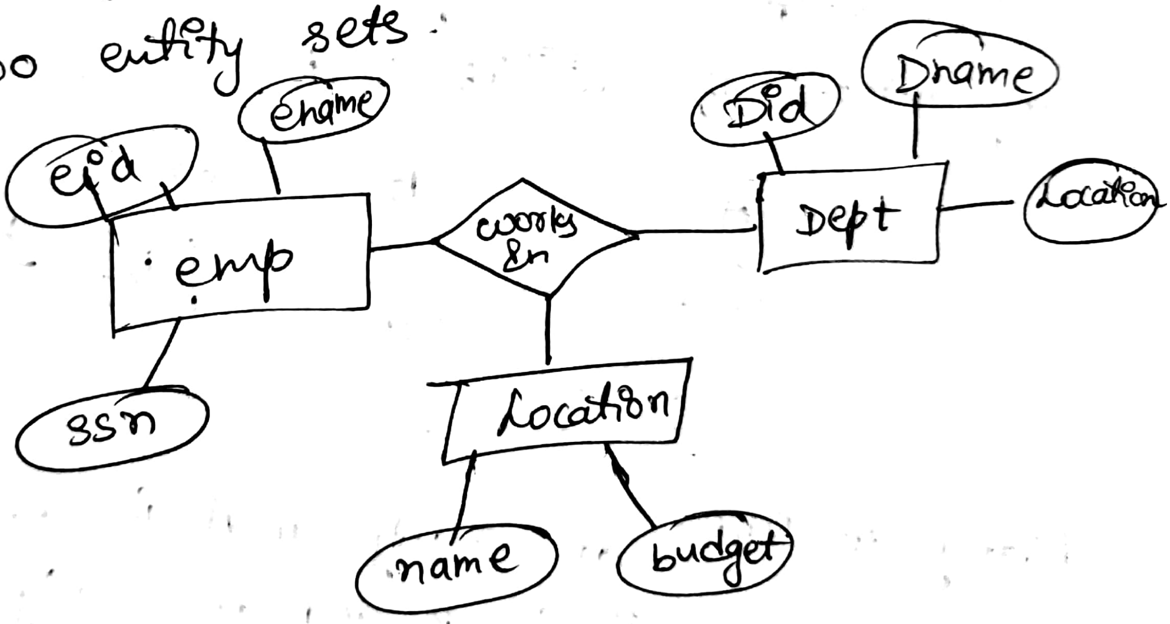
eg:



Degree - 2 .

A **ternary** relationship set involves more than two entity sets.

eg:



Degree - 3.

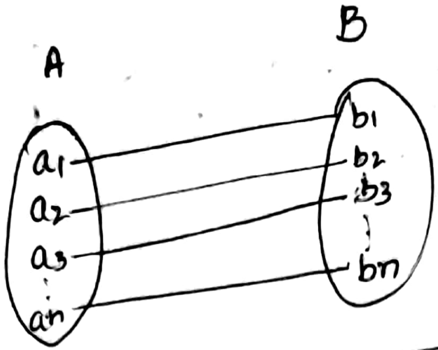
Mapping Cardinalities:

These are also called as cardinality ratios, this express the number of entities to which another entity can be associated to through a relationship set. They are useful in describing binary relationship set.

For a binary relationship set, the mapping cardinalities may be one of the following:

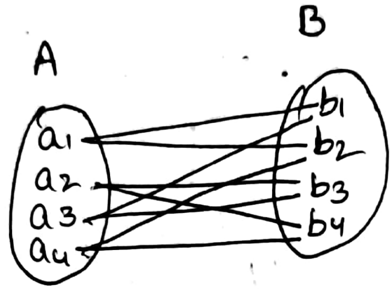
a) One to one: One entity from entity set A can be associated with at most one entity of entity set B and vice versa.

15.1

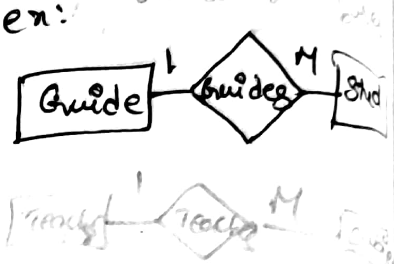


ex:

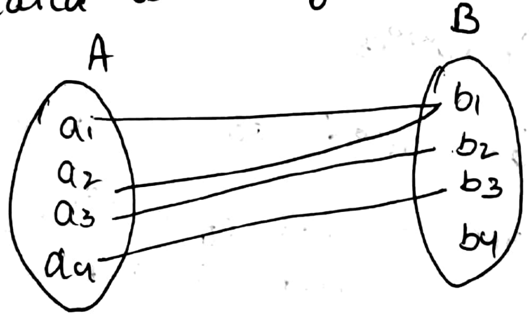
One to many: An entity in A is associated with any number of entities in B.



ex:

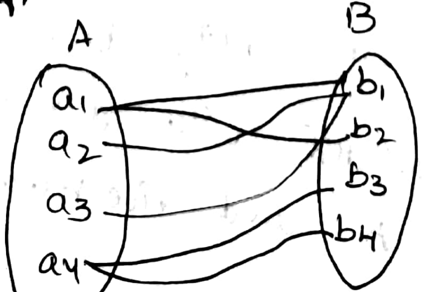


Many to one: An entity in A is associated with atmost one entity in B and an entity in B can be associated with any number of entities in A.



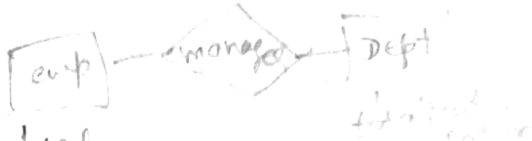
ex:

Many to Many: An entity in A is associated with any number of entities in B and an entity in B is associated with any num of entities in A.



ex:

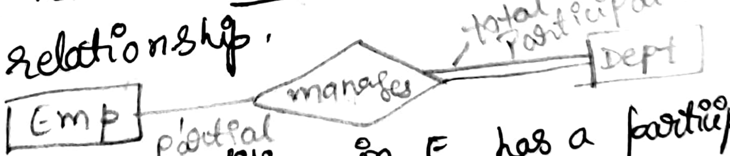
- The constraint that the above cardinality ratios has followed for the association between the entities is called as key constraint. These are the constraints for binary relationship set.
- The same convention of cardinality ratios can be extended to ternary relationship set.



Participation Constraints:

The participation of an entity set in E in a relationship set R is said to be total if every entity in E participates in atleast one relationship in R.

Eg: In employee db, 'employee', 'departments' are the entities. We build a relationship called 'Manager' between the two entities. Now, to see the participation, every dept must have a manager. So, the entity departments has a total participation through manages relationship.



If only some entities in E has a participation in relationship set R, then it is called as partial participation.

Eg: All the employees are not managers to the depts. So, the entity 'employees' has partial participation in relationship 'manager'.

Keys:

A key is an attribute or set of attributes in a relation that identifies a tuple in a relation. They are also used to create relationship between different tables.

Types of Keys:

-(i) Super Key- A super key is an attribute or combination of attributes in a relation that identifies a tuple uniquely within the relation.

Ex: Book

Bid	BName	Author
B_1	XYZ	A_1
B_2	ABC	A_1
B_3	XYZ	A_2
B_4	PQR	A_3
B_5	RSP	A_1
B_6	ABC	A_3

Superkeys → {Bid},
{BName, Author}
{BName, Bid, Author}
-- etc.

In superkey there may be a Redundant attribute
↓
attribute which is having redundant data

(ii) Candidate key: A candidate key is a super key without redundancy, and which is not reducible.

(OR) A candidate key is a super key which cannot have any cols removed from it without losing the unique identification property

Eg: from book Database

candidate keys - bid.

- {BName, Bid, Author}.
  X X

not candidate keys because they are redundant attributes.

So only bid is candidate key.

(iii) Primary key: A Primary key is a candidate key that is selected by the database designer.

From the above book example bid is primary key.

Note: - A relation can have only one primary key.
- Each value in Primary key attribute must be unique
- Primary key cannot contain null values.

(iv) Composite key: A primary key that consists of two or more attributes is known as composite key.

(v) Alternative key: The candidate key which are not selected for primary key.

v) Foreign key:

A foreign key is an attribute or set of attributes in a relation whose values match a primary key in another relation.

A relation may contain more than one foreign keys.

ex: Employee table

EId	EName	Job	Did

EId → primary key; Did → Foreign key.

Dept table

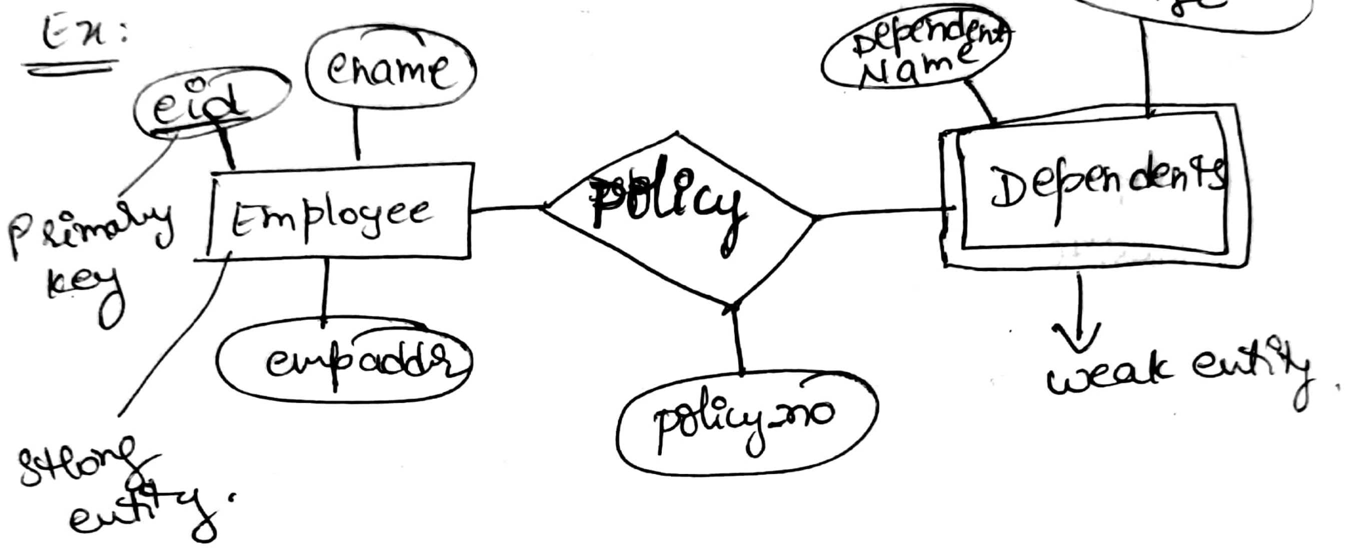
Did	Dname	location

Did → primary key

Weak Entities:

The entity which doesnot have sufficient attributes to form a primary key is called as Weak entity set.

Ex:



In this example Eid identifies unique data in employee entity.

But the entity 'Dependents', cannot identify unique data with its attributes Dependent Name and age.

Such type of entities are called Weak entity. (i.e. Dependents in the above ex).

An entity set that has a primary key is called as strong entity set.

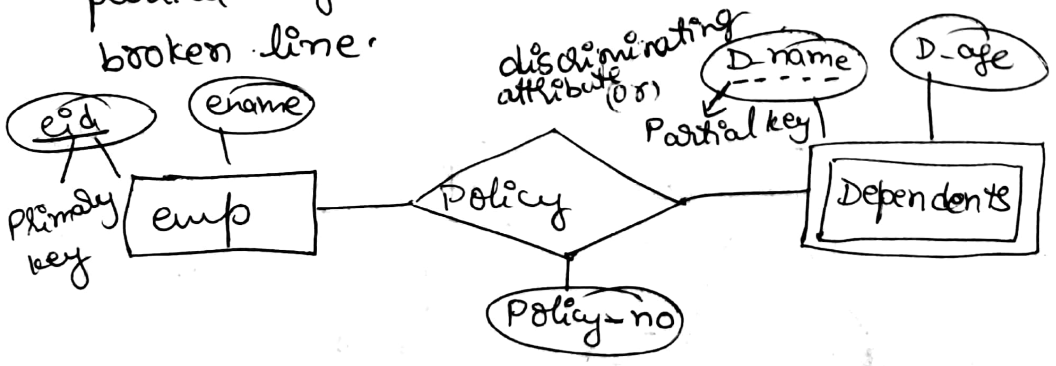
ex: Employee → strong entity from above figure.

Rules for confirming weak entity:

- The owner & weak entity sets must participate in one to many relationship set.
- It must have a total participation in identifying relationship set.

The set of attributes of the weak entity set used in conjunction with the primary key of identifying owner is called as the partial key.

To indicate this in the E-R diag, the partial-key names are underlined with a broken line.



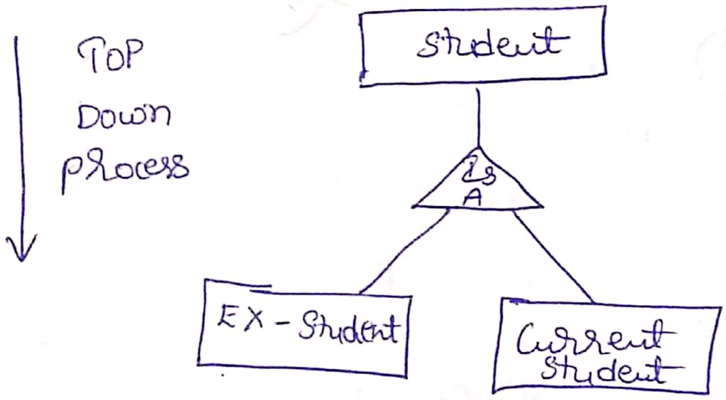
Class Hierarchies:

Class Hierarchy is a method of classifying the entities into subclasses. It can be viewed as

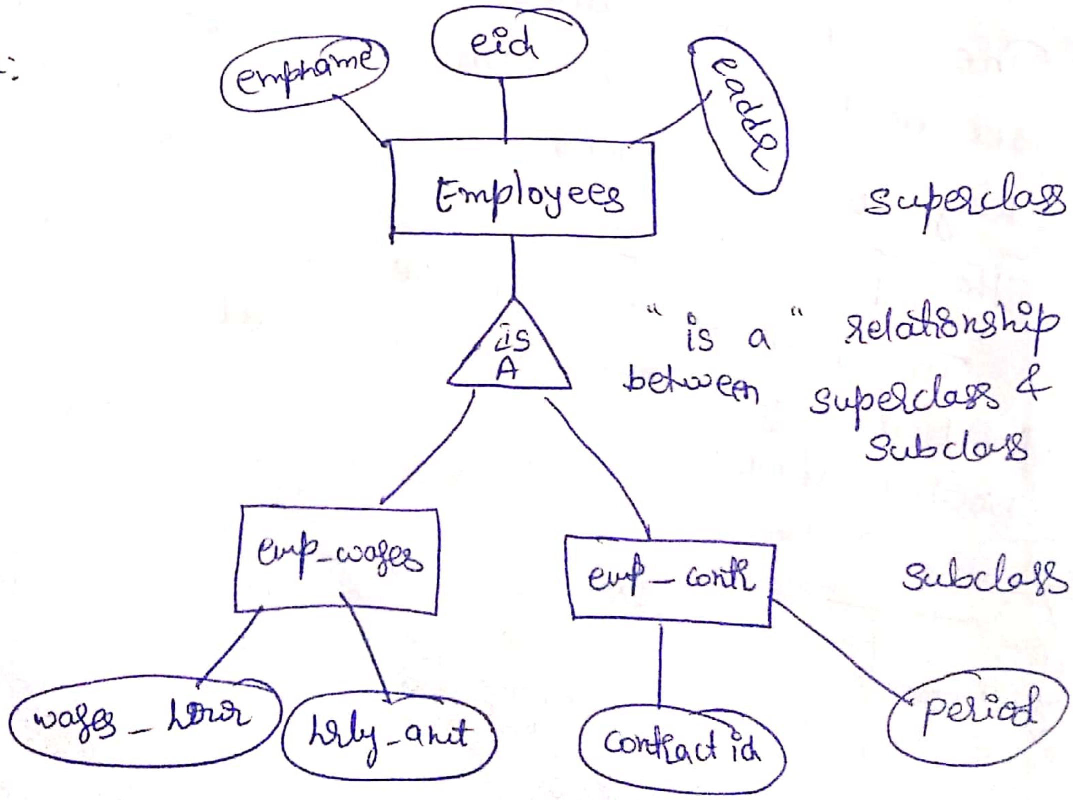
(a) Specialization:

It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, some higher level entities may not have lower-level entity sets at all.

Ex:

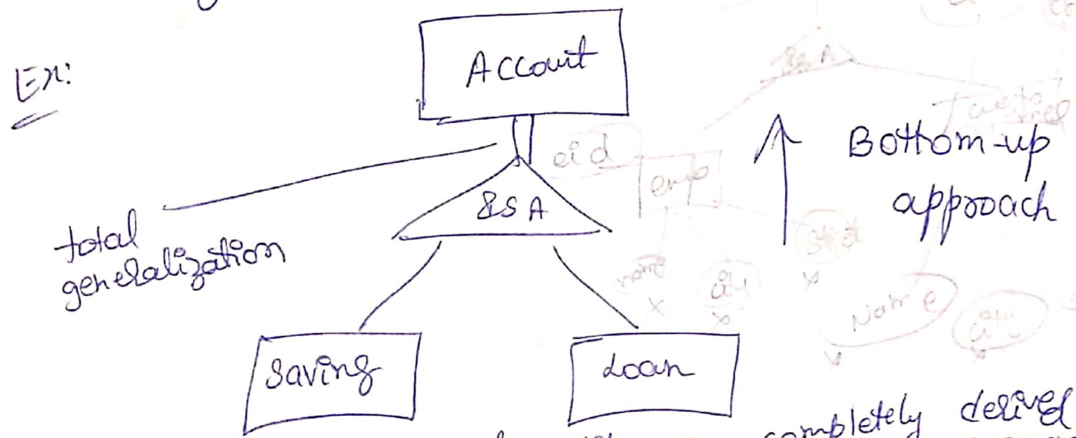


Ex:



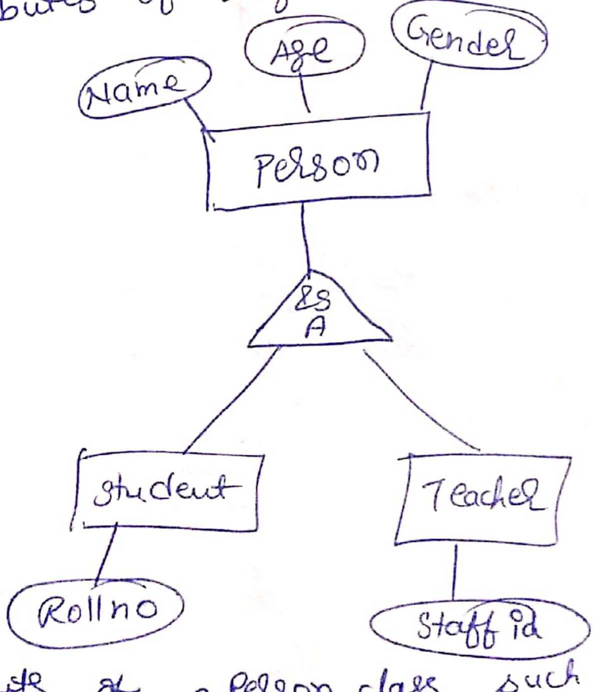
(b) Generalization: Generalization is a bottom up approach in which two lower level entities combine to form a higher level entity. In generalization, the higher level entity can also combine with other lower level entity to make further higher level entity.

Ex:



Note: If the higher level entity is completely derived from lower level entity sets, then it is called total generalization

(c) Inheritance: Inheritance is an important feature of Generalization & Specialization. It allows lower-level entities to inherit the attributes of higher-level entities.



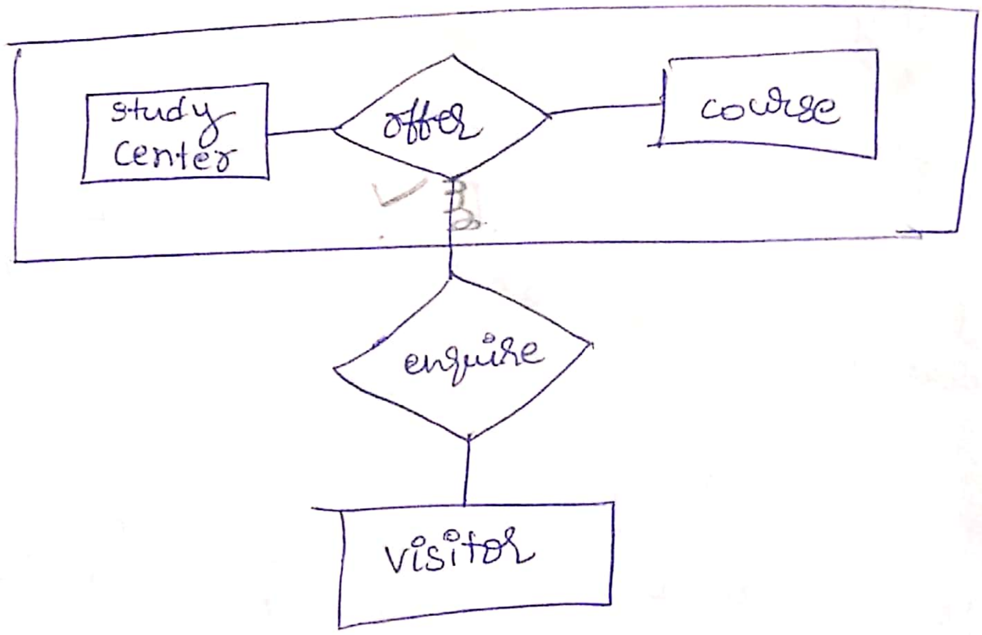
The attributes of a Person class such as name, age, & gender can be inherited by lower-level entities such as Student and Teacher.

(d) Aggregation: (It's a abstraction)

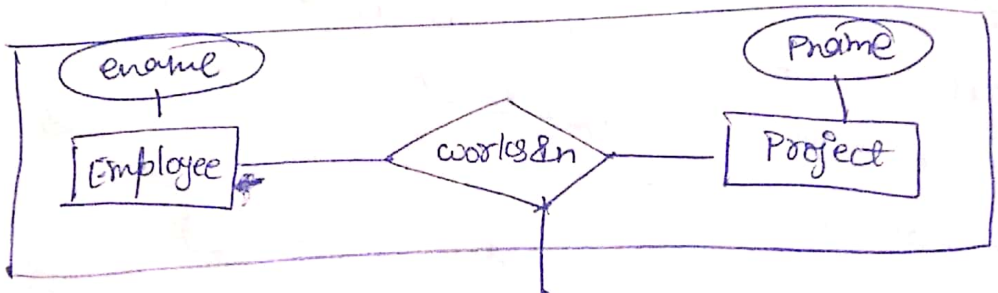
It is used when we need to express a relationship among relationships.

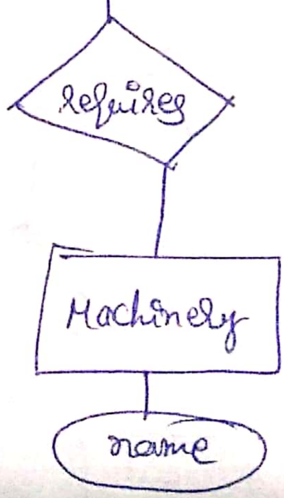
It is the feature of ER model that allows a relationship set participate in another relationship set. It is also called as "Has a" relationship.

Ex: (1)



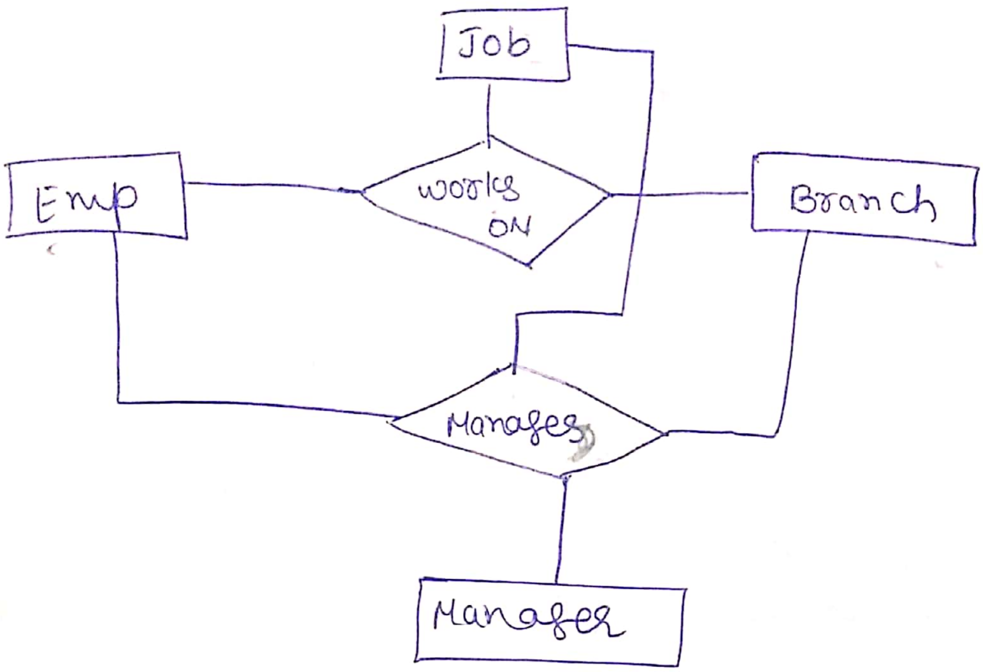
Ex: (2)



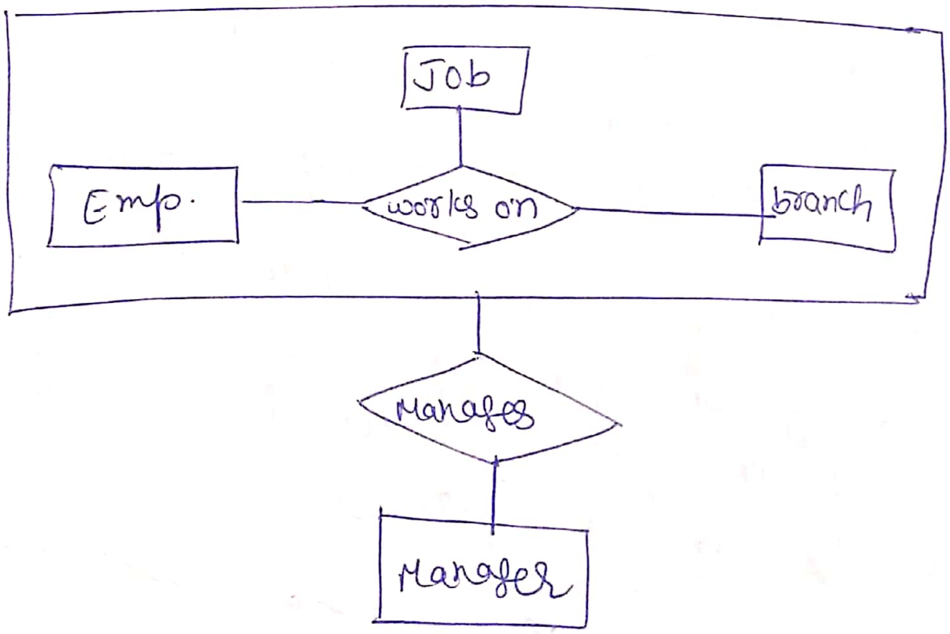


Employee work for projects.
An emp working for a particular project uses various machinery.

Ex: 3



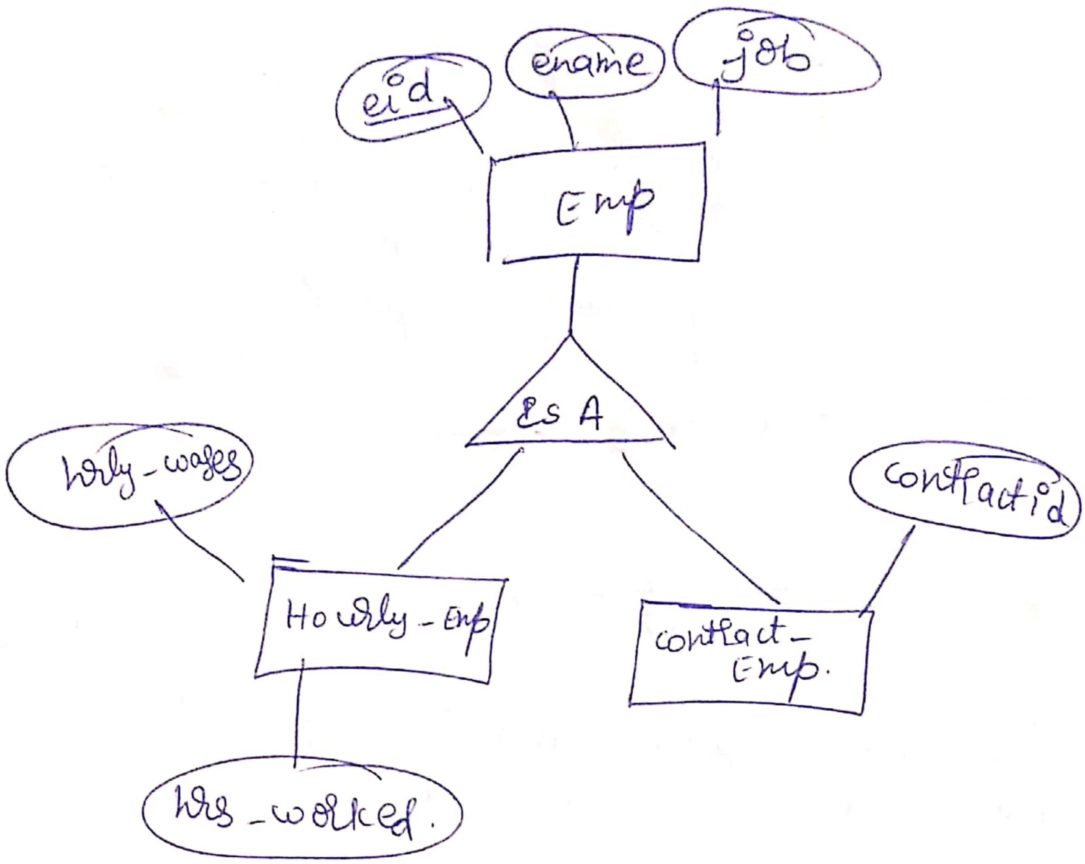
This is quite complicated in representation.
(we have redundant relationships).



There are two kinds of constraints regarding class Hierarchies:

→ Overlap Constraints: These constraints determine whether the two classes can have the same entity in common.

Eg: Employee ER Model.



In this scenario the hrly employee cannot be a contract employee. So the entity sets are constrained to no overlap constraints.

→ Covering Constraints: within an ISA Hierarchy, a covering constraint determines where the entities in the subclass collectively include all entities in the superclass.

Eg: From the above ER model
Does every hourly emp & contract emps are emps of this organization. - Yes, whether the contract / hrly employees belongs to the master emp entity. This is covering constraints.

Conceptual Design with ER Model:

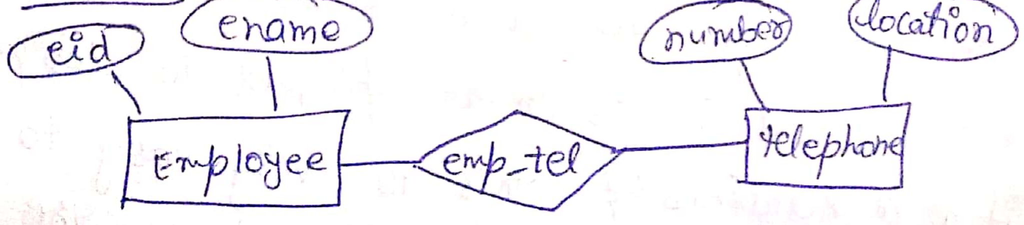
To develop an ER diagram, the following points must be considered.

a) Should a concept be an entity or an attribute
b) entity or an relationship
c) binary or ternary relationship
d) Use Aggregation.

a) Entity vs attribute:
While identifying the attributes of an entity set, it is sometimes not clear whether a property should be modeled as attribute or as an entity.

Ex: Consider the employee entity set with attributes emp_name & telephone_num, eid. This is only assumption in the begining. Now we will see, which one suits as an entity and which suits as an attribute.

Actually, in the above assumptions, "tel_num" is an attribute. But it can be taken as an entity with attributes 'number' and 'location'.



Like this, we can redefine the assumptions. But we have to see that the redefinition of the entities & their attributes will store the complete information.

With the above two assumptions, we can derive the following

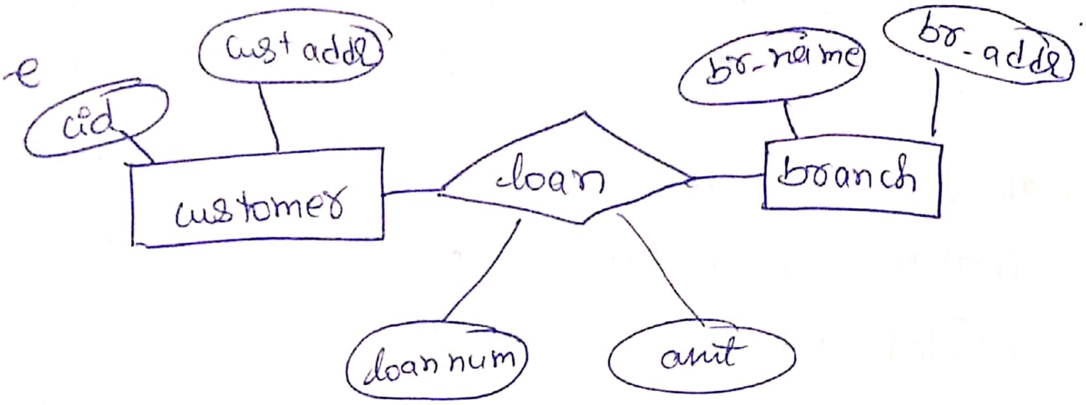
- With telephone as attribute, we insist that an employee should have a phone number.
- With telephone as an entity, we can allow 0 or more num of phone numbers for an employee.

But, this can even be redefined by using telephone as a multivalued attribute. If we do this, only number information is available but location infor is not available. So, taking telephone as an entity will be better model. So that we can store extra information regarding telephone.

(b)→ Entity sets vs Relationship sets:

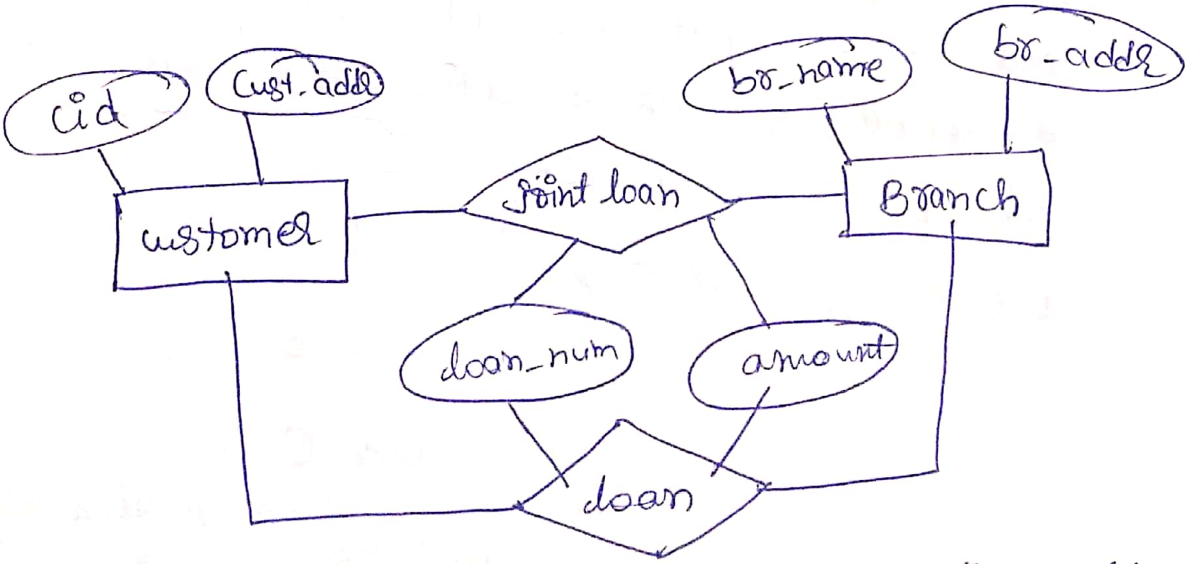
Which object has to be used as an entity & which one as a relationship is very different. In, banking db, loan is designated as an entity. But it is more proper to designate it as a relationship. This is very easy to represent the loan accounts between the customer & branches.

i.e



Now, this design will help us to keep the information about the customers who has taken loans & amounts from different branches. But this may fail, if more than one customer has taken a loan. The above design works for one-to-one mapping.

To allow joint loans, we define another relationship that describes about joint loans. We name it as "joint loan".



The new relationship, "joint_loan" should also maintain the same information as loan (i.e) loan-num and amount.

The disadvantage with this approach is the data is replicated. Updates to the attributes in one relationship may lead to database inconsistencies. All these can be avoided by using Normal Forms.

The best way to designat an object as a relationship is the relationship should describe an action that occurs between the entities.

(C) Binary vs n-ary Relationship sets:

Usually, relationships in dbs are binary. Sometimes, relations may be ternary also i.e >2. Then, the ternary relationships can be divided in several binary relationships.

Let R be a relationship between entities A, B and C. To split this into binary relationships we replace R with an entity set E and redefine three binary relationship sets,

i.e R_A, relating E and A

R_B, " E and B

R_C, " E and C

If there are any attributes for R, then those are assigned to E.

i.e (e_i, a_i) in R_A

(e_i, b_i) in R_B

(e_i, c_i) in R_C.

The same convention can be carried out for n-ary relationships. This may not be proper always. We need to take care of some points.

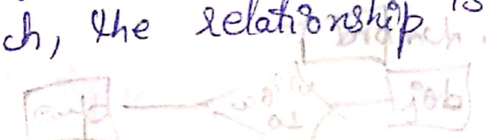
If there is an identifying attribute for the created entity E, then assigning the attributes of R to E can increase the complexity.

The cardinality ratios between the relationships & entities cannot be translated to give a meaning-ful picture.

Eg: If a relationship is many-to-one from A, B, to C, it means that A and B are associated with atmost one entity in C. This may not be clearly shown with the relationships R_A, R_B & R_C.

An n-ary relationship shows clearly how the entities participate in a single relationship.

Eg: The relationship between employee, branch and job is works_as that gives the details of an employee with a designation & branch in which employee works. This ternary relationship if split into binary relationships as R_A (i.e) between employee & job & R_B (i.e) between employee & branch, the relationship is not complete.

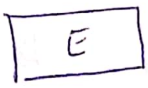


Because, R_A defines which employee is working under what designation and R_B defines in which branch an employee is working. But the relationship between the working of an employee in a particular branch, is missing.

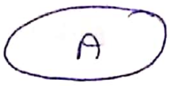
So, all ternary or n-ary relationships cannot be split into several binary relationships. Wherever there is possibility & the relationship remains intact, then the ternary relationship can be split into binary relationships.

Entity Relationship Diagram:

E-R diagram is used to express the overall logical structure of the db in a graphical form. The components of a ER-diagram are:



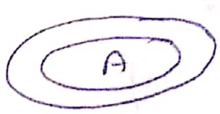
→ represent entity sets



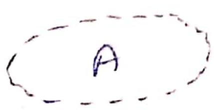
→ represent attributes

→ represent relationship

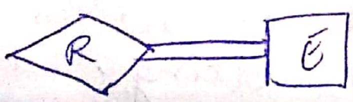
—— (lines) → link entity sets to the attribute & entity sets to relationship sets.



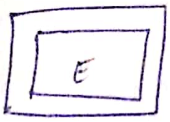
→ represent multi-valued attributes



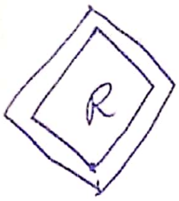
→ represent derived attributes



→ double lines represent total participation of an entity set in a relationship.



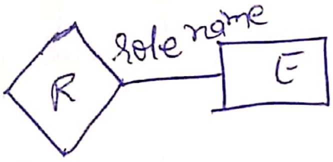
→ represent weak entity sets



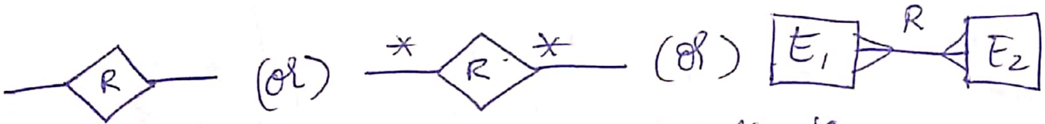
→ represent the identifying relationship set for weak entity sets.

→ represent primary key

→ represent discriminating attribute of weak entity set.



role indicator



Many-to-Many Relationship.

Many-to-One Relationship.

One-to-Many Relationship.

One-to-One Relationship

: ISA (i.e) is a specialization or Generalization

– Total generalization

→ Aggregation vs Ternary Relationships:

The choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a relationship set of a relationship on. entity set.

ex:

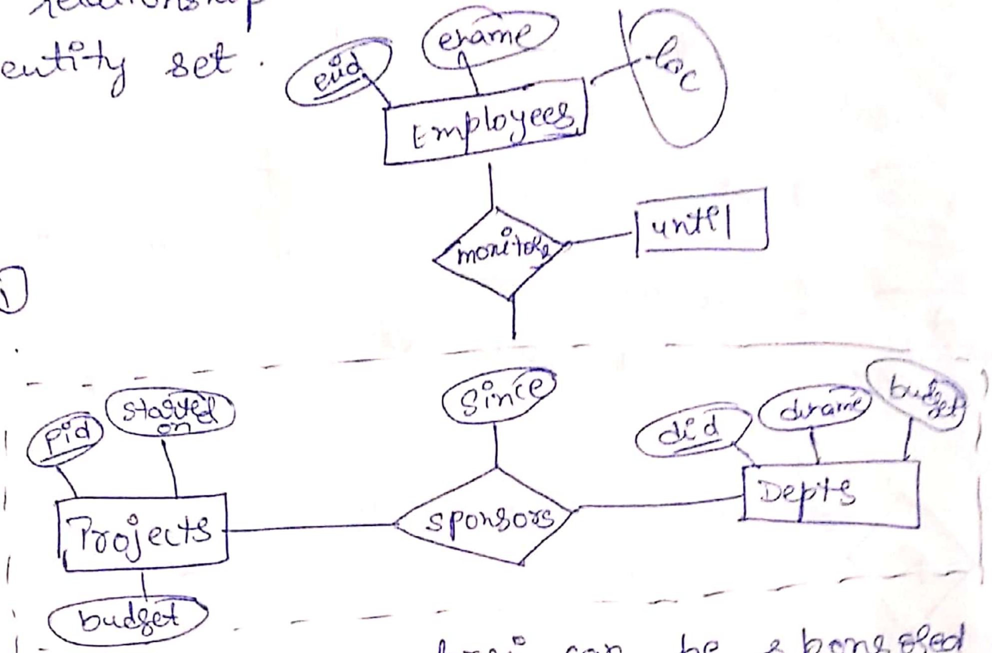


fig: ①

According to this a proj can be sponsored by any num of depts, a dept can sponsor one or more projects, & each sponsorship is monitored by one or more employees. If we don't need to record the until attribute of Monitors then we might reasonably use a ternary relationship as shown in below fig ②.

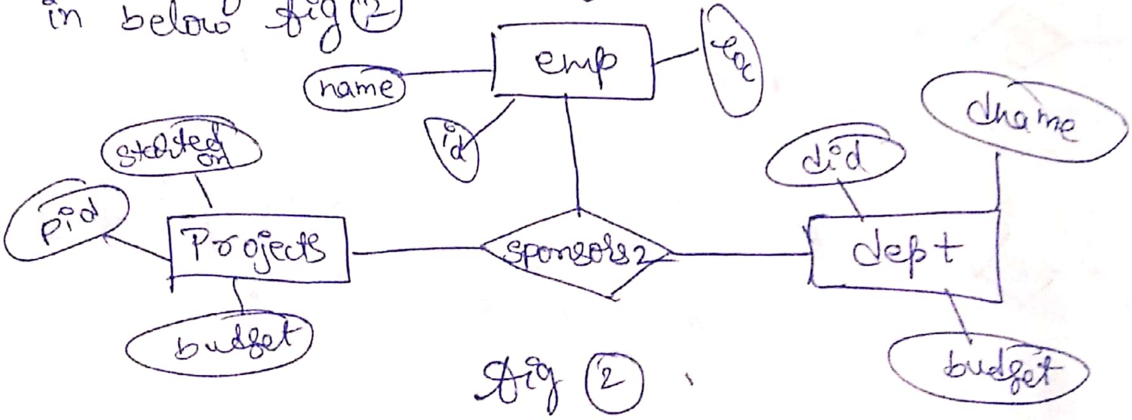


fig ②.

If the constraint is each sponsorship be monitored by at most one employee, we cannot express this constraint in terms of sponsors2 relationship set. We can express the constraint by drawing an arrow from the aggregated relationship sponsors to the relationship monitors in fig ①.

UNIT-II

Relational Model:

Relational model stores data in the form of tables. This concept proposed by Dr.E.F.Codd a researcher of IBM in the year 1960s. The relational model consists of 3 major components:

→ (i) The set of relations & set of domains that defines the way data can be represented.

(ii) Integrity rules that define the procedure to protect the data (data integrity).

(iii) The operations that can be performed on data (data manipulation).

A relational model db is defined as a db that allows you to group its data items into one or more independent tables that can be related to one another by using fields common to each related table.

Tables – In relational data model, relations are saved in the format of Tables. This format stores the relation among entities. A table has rows & cols, where rows represent records & cols represent the attributes

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

Relation instance – A finite set of tuples in the relational db sys represents relation instance. Relation instance do not have duplicate tuples.

Relation schema - A relation schema describes the relation name (table name), attributes & their names.

eg: stud Relation

Attributes →

Sid	Sname	Login	Age
101	John	John@gmail.com	19
102	Paul	Pa@gmail.com	20
103	Ram	Ram@Yahoo.co.in	19

tuples (rows 101–103)

Cardinality of a relation: The num of tuples in a relation determines its cardinality. In this case, ~~the~~ (i e above table) cardinality is 3.

Degree of a relation: Each coloum in the tuple is called an attribute. The num of attributes in a relation determine its degree. From the above table degree is 4.

Domains: A domain definition specifies the kind of data represented by the attribute.

Domain Constraints:

Attributes have specific values in real-world scenario. For example, age can only be a positive integer. Every attribute is bound to have a specific range of values. For example age cannot be less than zero. and telephone num cannot contain a digit outside 0-9.

Creating & Modifying Relations Using SQL:

SQL, stands for Structure Query language. This is used for creating & modifying relations in a db. SQL uses the word 'table' to denote a relation. SQL has two subsets in the language. where each one is used for a specific purpose.

SQL
- DDL
  - create
  - alter
  - drop
  - truncate
  - comment
  - Rename
- DML
  - insert
  - select
  - update
  - delete.

eg: create table stud(sid number(5), sname varchar2(10), login varchar2(20), Age number(2));

with this table will be created.

To insert values the command is

insert into stud values (101, 'xyz', 'xy@gmail.com', 19);

To DELETE a tuple command is

DELETE from stud where sid=101;

To modify a col value command is

UPDATE stud set sname='John' where sid=101;

Integrity ~~const~~ Constraints over Relations:

An integrity Constraint is a condition specified on the db schema which restricts the data that has to be stored in the db. If the db instance satisfies all the integrity constraints imposed, then the instance is called as a legal instance.

DBMS enforces such integrity constraints inorder to store only legal instances in the db. If there are any violations, then DBMS sees that changes are not made to the db. There are many types of integrity constraints. They are:

1) **Key Constraints:**

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

A set of fields that uniquely identifies a tuple according to a key constraint is called a <u>candidate key</u>

ex: <u>sid</u> is a candidate key in student relation

According to the candidate key:

→ Two distinct tuples in a legal instance cannot have identical values in all the fields of a key.

→ No subset of the set of fields in a key is a unique identifier for a tuple.

Ex: In students relation, if sname is 1:33 chosen as a candidate key it may not identify the tuple uniquely because, two students may have the same name.

A relation can have any number of candidate keys. ex, {studid}, {sname, age}, {login, age}.

We have observe that the values of these fields in the keys in two tuples should not be identical.

Out of all the candidate keys, a key called primary key should be chosen by a database designer.

eg: In employee db, eid is a candidate key.
If {emp-name, address} is used as a candidate key, then it cannot be primary key because, the 'address' attribute may change.

Specifying key constraints in SQL:

Ex: create table student (sid varchar2(10), login char(20),
sname char(30), UNIQUE (name, age),
age number(3),
CONSTRAINT studskey PRIMARY KEY (sid)).

This def says that sid is the primary key and the combination of name & age is also a key.

We can name a constraint by preceding it with CONSTRAINT constraint_name.

If the constraint is violated, the constraint name is returned & can be used to identify the error.

Foreign key constraints:

Sometimes the information stored in a relation is linked to the information stored in another relation. If one of the relations is modified, the other must be checked, & perhaps modified, to keep the data consistent. An integrity constraint must be specified that involves both the relations. Such a constraint is called as Foreign key Constraint.

Ex:

Enrolled Relation.

cid	cname	grade	sid
101	History	C	5318
102	Topology	A	5832
103	Economics	B	5624

Student Relation

sid	sname	login	age
5000	Dave	D@cs	19
5318	Smith	S@cs	20
5368	John	J@cs	21
5832	Paul	P@cs	19
5624	Jones	Jones@cs	20

Any value that appears in sid field of enrolled relation should also appear in the sid field of student relation. The sid field of enrolled relation is called a foreign key & refers to student relation.

If we try to insert a tuple <104, Art, 'A', 5622> into enrolled relation, the integrity constraint is violated because there is no tuple in student relation with 5622.

Similarly, if we delete the tuple <5624, 'Jones', 'Jones@ss', 20> from stud relation, we violate the foreign key constraint because the tuple in enrolled relation contains sid value 5624.

Specifying Foreign key constraint in SQL:

Ex: create table enrolled (sid varchar(10), cid number(3), cname varchar2(10), primary key (sid), Foreign key (sid) references student (sid));

General constraints:

Apart from domain, primary and foreign key constraints, we can impose constraints that are general to a table.

Eg: If we want to impose a constraint like, the age of a student should be atleast 15. This is diff from other constraints. Such a constraint is called General constraint. If any insertion violates this constraint, the operation is not performed.

Relational databases consider these constraints as either table constraints or assertions.

- table constraints ↓ these are associated with a single table & checked when table is modified.
- assertions ↓ these are associated with multiple tables, & checked whenever any one of the multiple tables are changed.

Enforcing integrity constraints:

There are different types of key constraints domain, primary, candidate keys. If any operations like INSERT, DELETE, UPDATE are performed, then the ICs are checked for violations. If any IC is violated, then the operation is not performed.

Eg: 1: student db with sid as Primary key

operation:

insert into stud values (5000, 'ABC', 'a@gmail.com', 20);

This insertion will be violated because stud-id - 5000 already exists in the relation. This violation is due to primary key.

Eg: 2: Students db with domain constraints.

operation:

insert into stud values (NULL, 'ABC', 'A@cs', 20);

This operation is violated because the primary key cannot have null value. This relation is due to the domain constraints.

The complexity increases when the Foreign key constraints are involved.

SQL sometimes try to rectify the violation instead of simply rejecting the operations.

There are many cases where the referential integrity is involved. We shall see the steps

Eg:1. Consider the referential integrity between students relation & enroll-students relation. In this sid is the Foreign key & sid is primary key of students relation.

If any insertion is done to the enroll_stud relation whose sid is not value in stud- relation then a violation occurs
eg: insert into enroll values (5001, 010, 'Art', C);

Here 5001 is not a value of stud relation sid. so the foreign key is violated.

Eg:2 Deletions to the stud tuples may lead to violation. i-e
delete from stud where sname = 'Smith';

This refers to a stud-id which may have a value of sid in enroll_stud table. so deletions must be performed in both the tables or disallow the deletion.

Eg:3 Any Updations that change the sid values also violate the referential integrity.
i-e UPDATE student set sid = 5235 where sid = 5318;

In this case also, the updation must be either violated or the operation has to be performed in both the tables by modifying the value of sid in stud relation & sid in enroll relation to 5235.

All the above circumstances give us options to be specified:

Eg: We can quote an option that when a row of students relation is deleted, the rows of the corresponding sid values must be deleted. But updations to the students relation should not be updated with the referred sid rows of enroll_stud relation.

```
i-e create table enroll (sid char(20), cid char(..),
        cname varchar2(10), grade char(5),
    primary key (sid),
    Foreign key (sid) references stud (sid)
    ON DELETE CASCADE ON UPDATE
    NO ACTION);
```

The options are specified as part of the Foreign key definition. The default action is "NO ACTION" (i-e) operations on the table must be rejected. The other actions is "CASCADE" that defines the same operation has to be performed in the referencing & referenced relations.

Transactions and Constraints:

Any program that runs against the database is called as transaction. (it may consist of queries & stmts that are used to access the database.

If any stmt or query violates any integrity constraint, then what action must be taken (i-e) whether all other integrity constraints must be checked before the transaction completes or not. (1.36)

By default, Oracle checks for the constraints at the end of SQL stmt. If there is any violation, the stmt is rejected. But this may lead to problems.

Eg: We have two relations students and courses.

FK references courses (cid))

students(studid, sname, login, honors, age)

courses(cid, cname, credits, grader) → FK references stud(sid)

- These two are participating in referential integrity where students foreign key is "honors" and courses foreign key is "grader" and each relation is refering to other.
- When the tuples are inserted in the relations for the first time, the constraints are violated because grader and honors are related to each other. To insert a value of grader, there must be a value of honors. Both the tuples cannot be inserted simultaneously.

The only way to insert the tuples are to defer the constraints.

SQL allows two modes for a constraint:
(a) Deferred mode (b) Immediate mode

A constraint in deferred mode will be checked at commit time (i.e) when the insertion is made. But in immediate mode, the constraint is checked before the insertion.

CONSTRAINTS

Constraints are the rules enforced on the data columns of a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Following are some of the most commonly used constraints available in SQL.

- NOT NULL Constraint – Ensures that a column cannot have NULL value.
- DEFAULT Constraint – Provides a default value for a column when none is specified.
- UNIQUE Constraint – Ensures that all values in a column are different.
- PRIMARY Key – Uniquely identifies each row/record in a database table.
- FOREIGN Key – Uniquely identifies a row/record in any of the given database table.
- CHECK Constraint – The CHECK constraint ensures that all the values in a column satisfies certain conditions.

DEFAULT Constraint –

Ex:1

SQL>create table emp13(eid int primary key,ename varchar2(10),addr varchar2(10) **default 'hyd'**);

Table created;

SQL>insert into emp13(eid,ename) values(3,'aaa');

1 row created.

```
SQL> select * from emp13;

       EID ENAME      ADDR
---------- ---------- ----------
         3 aaa        hyd
```

Ex:2

```
SQL> create table emp14(eid int primary key,ename varchar2(10),hdate varchar2(10) default sysdate);
Table created.
SQL> insert into emp14(eid,ename) values(3,'aaa');
1 row created.
SQL> select * from emp14;
       EID ENAME      HDATE
---------- ---------- ----------
         3 aaa        19-JAN-20
```

CHECK Constraint :

```
create table voter(adhar int,name varchar2(10),age number(3),check (age>=18));

Table created.
```

insert into voter values(12345,'aaaa',18);

1 row created.

SQL> insert into voter values(12345,'aaaa',17);
insert into voter values(12345,'aaaa',17)
*
ERROR at line 1:
ORA-02290: check constraint (SREEDEVI.SYS_C004033) violated
SQL> create table stud1(rno number(10),name char(10),age number(3),constraint roll primary key(rno));
Table created.
SQL> alter table stud1 drop constraint roll;
Table altered.

NOT NULL Constraint
SQL> create table bus(bid int, bname varchar2(10) NOT NULL);

UNIQUE Constraint

create table bus1(bid int unique, bname varchar2(10) NOT NULL);

Table created.

SQL>alter table bus1 add constraint empname unique(bname);

SQL> insert into bus1 values(122,'viz');

1 row created.

SQL> insert into bus1 values(122,'viz');
insert into bus1 values(122,'viz')
*
ERROR at line 1:
ORA-00001: unique constraint (SREEDEVI.SYS_C004036) violated

SQL> insert into bus1 values(null,'viz');

1 row created.

INITIALLY DEFERRED DEFERRABLE -(Deferring Constraint Checking)

Sometimes it is necessary to defer the checking of certain constraints, most commonly in the "chicken-and-egg" problem. Suppose we want to say:

```
CREATE TABLE chicken (cID INT PRIMARY KEY,
                  eID INT REFERENCES egg(eID));

CREATE TABLE egg(eID INT PRIMARY KEY,
              cID INT REFERENCES chicken(cID));
```

But if we simply type the above statements into Oracle, we'll get an error. The reason is that the CREATE TABLE statement for chicken refers to table egg, which hasn't been created yet! Creating egg won't help either, because egg refers to chicken.

To work around this problem, we need SQL schema modification commands. First, create chicken and egg without foreign key declarations:

```
CREATE TABLE chicken(cID INT PRIMARY KEY,
                 eID INT);

CREATE TABLE egg(eID INT PRIMARY KEY,
              cID INT);
```

Then, we add foreign key constraints:

```
ALTER TABLE chicken ADD CONSTRAINT chickenREFegg
   FOREIGN KEY (eID) REFERENCES egg(eID)
   INITIALLY DEFERRED DEFERRABLE;

ALTER TABLE egg ADD CONSTRAINT eggREFchicken
   FOREIGN KEY (cID) REFERENCES chicken(cID)
   INITIALLY DEFERRED DEFERRABLE;
```

INITIALLY DEFERRED DEFERRABLE tells Oracle to do deferred constraint checking. For example, to insert (1, 2) into chicken and (2, 1) into egg, we use:

```
INSERT INTO chicken VALUES(1, 2);
INSERT INTO egg VALUES(2, 1);
COMMIT;
```

Because we've declared the foreign key constraints as "deferred", they are only checked at the commit point. (Without deferred constraint checking, we cannot insert anything into chicken and egg, because the first INSERT would always be a constraint violation.)

Finally, to get rid of the tables, we have to drop the constraints first, because Oracle won't allow us to drop a table that's referenced by another table.

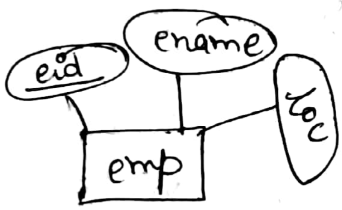
```
ALTER TABLE egg DROP CONSTRAINT eggREFchicken;
ALTER TABLE chicken DROP CONSTRAINT chickenREFegg;
DROP TABLE egg;
DROP TABLE chicken;
```

Logical Database Design : ER to Relational

ER model is used for representing high level database design. This logical structure of db design is translated to Relational db schema. The translation involves entity sets to tables & relationship set to tables & regarding constraints too.

1) Entity sets to tables:

This translation is very simple. The entity set name becomes the relation name and the attributes are column names.



⇒

emid	ename	loc

The domain & key constraints are also included in the SQL stmt.

Create table emp(eid varchar2(10), ename char(10), loc char(20), Primary key (eid)).

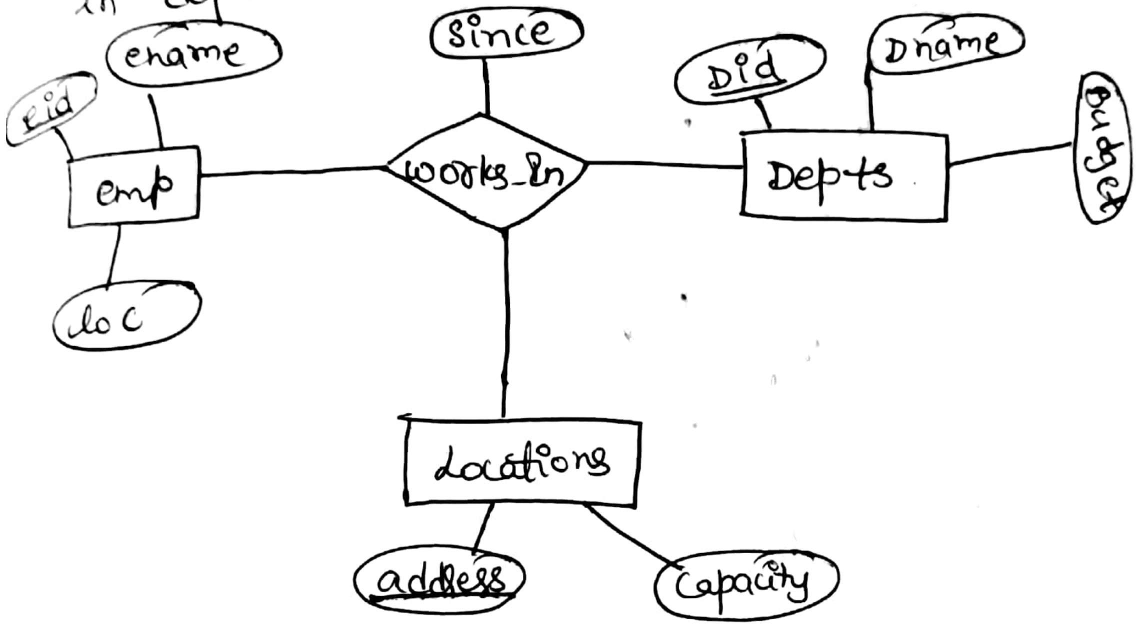
2) Relationship sets to tables (without constraints):

A relationship can also be translated to tables. To represent a relationship, we need to represent each entity in the entity sets participating in the association.

The descriptive attributes also must be given some values. For this we describe the attributes of the relationship as :

- The primary key attributes of each participating entity set, as foreign key fields. (1.37)
- The descriptive attributes of the relationship set.

Eg: Consider "works in" relationship among the entity sets "employee", "Departments", "locations". This relationship describes about the employees working in departments in a location.



Here, the attributes that describe "works_in" relationship are the primary keys of the participating entity sets & its descriptive attributes (i.e) eid, did, address, and since.

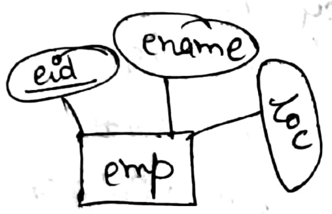
```
create table Works_in (eid varchar2(10) Primary key,
did varchar2(10), address varchar2(20), since date,
Foreign key (eid) References emp(eid),
Foreign key (did) References dept(did),
Foreign key (address) References locations
(address));
```

Logical Database Design : ER to Relational

ER model is used for representing high level database design. This logical structure of db design is translated to Relational db schema. The translation involves entity sets to tables & relationship set to tables & regarding constraints too.

1) Entity sets to tables:

This translation is very simple. The entity set name becomes the relation name and the attributes are column names.



⟹

eid	ename	loc

The domain & key constraints are also included in the SQL stmt.

Create table emp(eid varchar2(10), ename char(10), loc char(20), Primary key (eid));

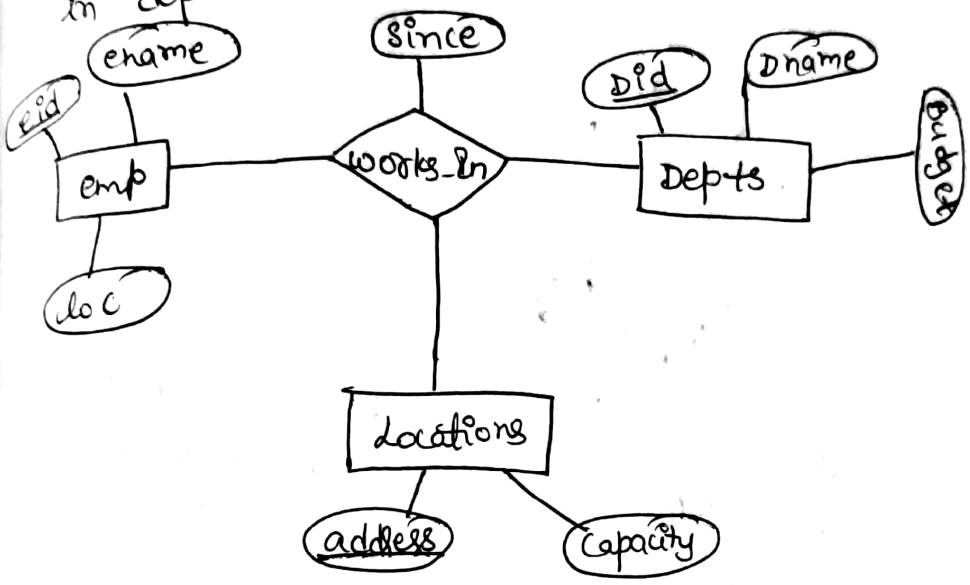
2) Relationship sets to tables (without constraints):

A relationship can also be translated to tables. To represent a relationship, we need to represent each entity in the entity sets participating in the association.

The descriptive attributes also must be given some values. For this we describe the attributes of the relationship as :

- The primary key attributes of each participating entity set, as foreign key fields. (1.37)
- The descriptive attributes of the relationship set.

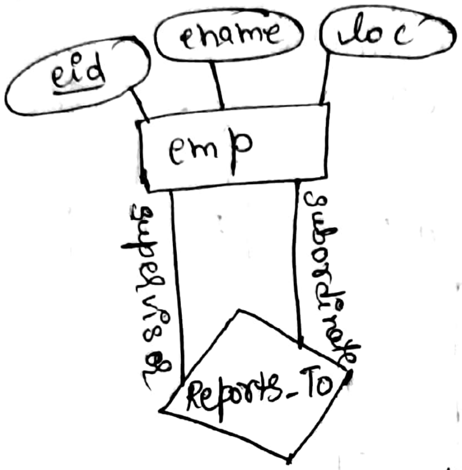
Eg: Consider "Works In" relationship among the entity sets "employee", "Departments", "locations". This relationship describes about the employees working in departments in a location.



Here, the attributes that describe "Works_In" relationship are the primary keys of the participating entity sets & its descriptive attributes (i.e) eid, did, address, and since.

```
create table Works_In (eid varchar2(10) Primary key,
did varchar2(10), address varchar2(20), since date,
Foreign key (eid) References emp (eid),
Foreign key (did) References dept (did),
Foreign key (address) References ~~dept~~ locations
(address));
```

Eg 2: Consider the relationship "Reports_to", that has association with the entity set "employees". In this, employees entity set participates more than once in the relationship. So, role indicators are used to identify the participation.



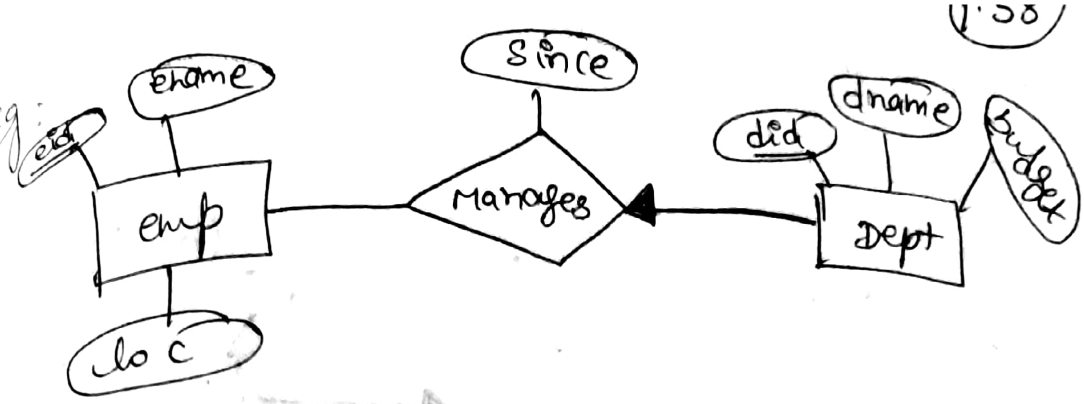
Here role indicators are used for creating meaningful column names in the table with name "Reports_to".

```
Create table Reports_to (supervisor_eid varchar2(10),
    subordinate_eid varchar2(10),
    primary key (supervisor_eid, subordinate_eid), Foreign key (supervisor_eid)
    References emp(eid),
    Foreign key (subordinate_eid) references
    emp(eid));
```

3) Translating Relationship sets with key constraints:

If there are n entity sets participating in the relationship, only m entity sets are linked via ER diagram. So, the key used by those m entity sets is the key for the relation. Then, we have m candidate keys and out of m candidate keys, one is chosen as the primary key.

eg:



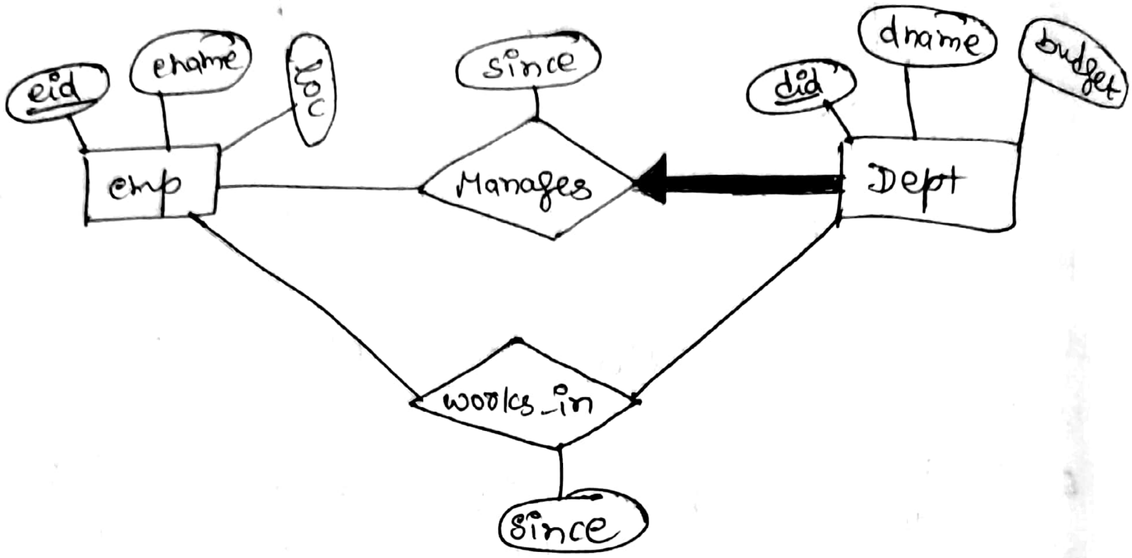
Here the key is "each department will have atmost one manager ie no two tuples must have the same did. A dept can have 0 or 1 manager only. This key is given with the attribute did, because the reference we take to check the constraint is only did.
So, did is the primary key for "Manages" relationship. This is given as:

Create table Manages (eid varchar(10), did varchar(10), since date, primary key (did), Foreign key (eid) References emp (eid), Foreign key (did) references dept (did));

This is one approach. Another approach is we translate depts & Manages to ~~Depts Manager~~ "Dept_Managers"

ie Create table Dept_Managers (did varchar(10), dname varchar(20), budget real, eid varchar(10), since date, primary key(did), Foreign key(eid) references emp(eid));

4) Translating Relationship sets with participation constraints:



To translate the participation constraint we write the query as

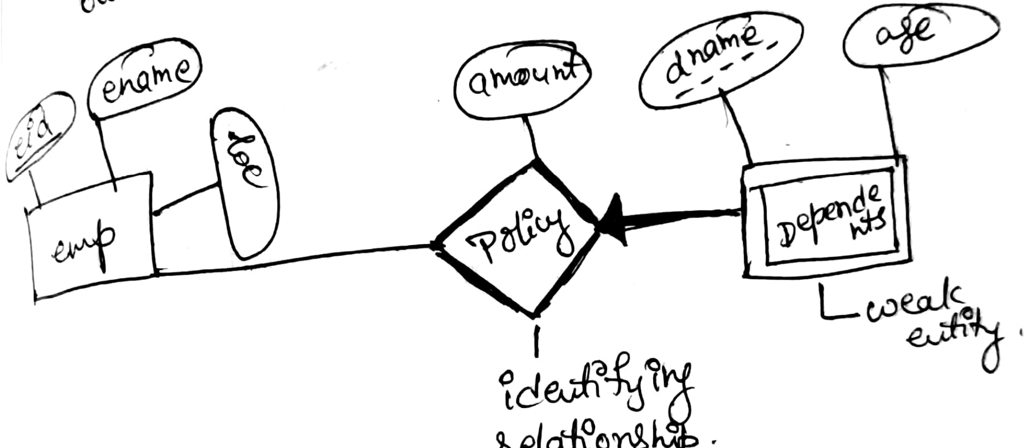
```
create table Dept_Manager(did varchar(10),
        dname char(15), budget number(10),
        eid varchar(10) NOT NULL, since date,
        primary key (did), Foreign key (eid)
        references emp (eid));
```

In ~~the~~ ER diagram we mention the participation constraint that every department must have a manager (total participation) ~~by~~ and no value ~~of~~ eid in "dept_manager" must be null

In the SQL query stmt ON DELETE NO ACTION also can be specified.

5) Translating Weak Entity sets:

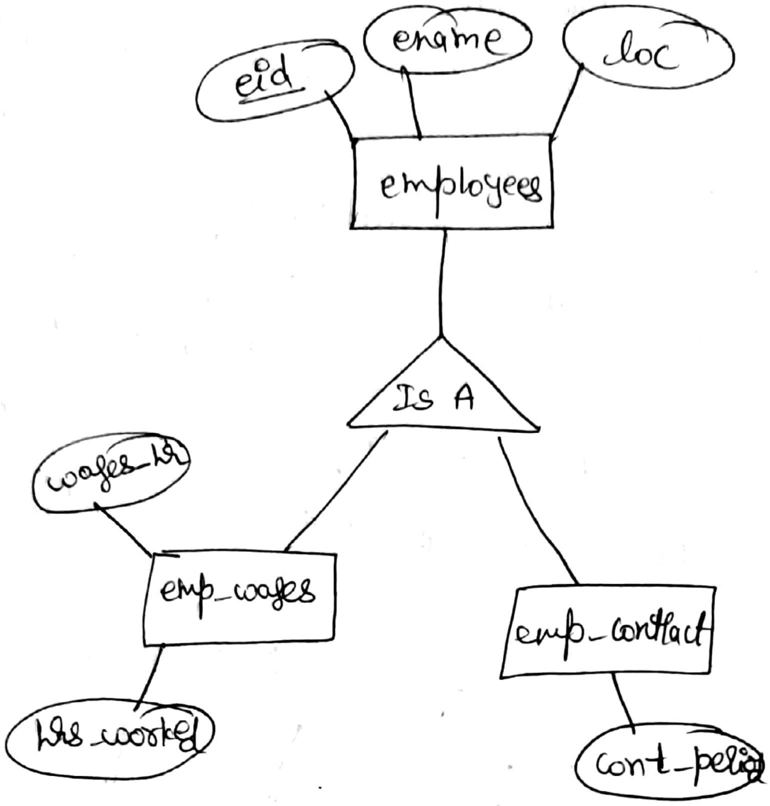
Consider the example of employees and dependents entity sets via the relationship Policy. Here, "dependents" is the weak entity set. It has one-to-many relationship & total participation with the relationship "Policy". because there are no enough attributes to identify the entities of dependents, we use the primary key eid as the identifying owner.




```
create table policy-dependents (dname char(15),
    age number(3), amount number(10),
    eid varchar(10), Primary key (dname, eid),
    Foreign key (eid) references emp (eid)
    ON DELETE CASCADE);
```

Here, whenever an employee details is deleted from employees, then the corresponding tuple in Policy-dependents must also be deleted.

6) Translating class Hierarchies:



create one relation for each entity set.
i e a table for employees, emp_wages, emp_contract.

i-e

-> create table emp(eid varchar(10), ename varchar2(10), loc varchar2(20), primary key (eid));

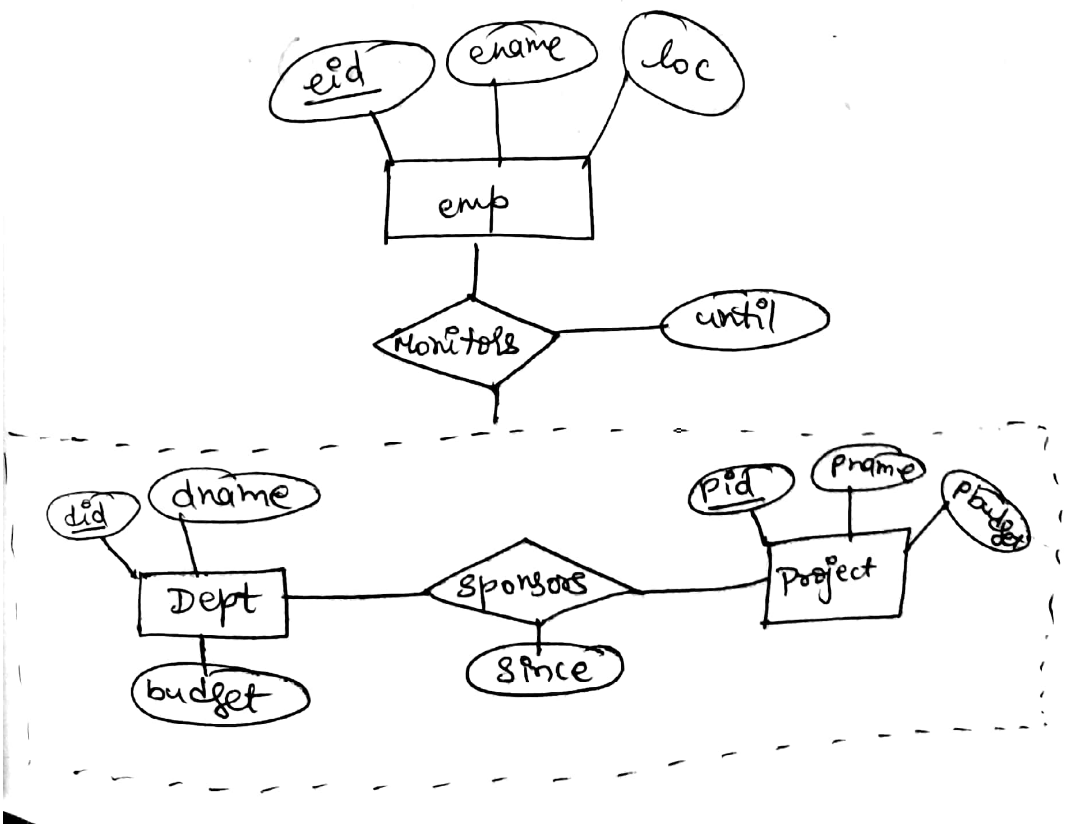
-> create table emp_wages(eid varchar(10) primary key, wages_hr real, hrs_worked integer, foreign key (eid) references emp (eid) ON DELETE CASCADE);

-> create table emp_contract(eid varchar(10) primary key, contperiod integer, foreign key (eid) references emp (eid) ON DELETE CASCADE);

If we want to retrieve ~~any~~ only about 1.40
employees but not the information specific about the subclasses, it can be done using employees. But when we have to retrieve the attributes "ename", "loc" of any of the subclasses, we need to combine the tables of superclass & subclasses.

If we have created only two tables i.e emp-wages and emp-contract, this approach may fail in some cases because there may be employees that are neither emp-wages nor emp_contract. Also, the information of the attributes ename, location are duplicated.

7) Translating ER diagrams with Aggregation:



In the above ER diag, dept sponsor projects. This sponsorship must be monitored. For this, we define another relationship "monitors" that describes about the employee who is monitoring a sponsorship. Because this cannot be given individually to either Projects or Departments we used **Aggregation**, that treats "sponsors" relationship between "projects" and "departments" as a high level entity set. With this, there is the new relationship between "employees" and "sponsors" i.e "monitors".

→ create table monitors (eid varchar(10), did varchar(10), pid varchar(10), until date, primary key (eid, ~~primary key~~ did, ~~primary key~~ pid), Foreign key (eid) references emp (eid), Foreign key (did) references sponsors (did), Foreign key (pid) references sponsors (pid));

Examples in this unit are taken using the following schema:

sailors (sid : integer, sname : string, rating : integer, age : real);

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

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

Relational Algebra:

- The relational algebra is a procedural query language.
- Relational Algebra is a frame work for query optimization & Query implementation
- It consists of a set of operations that take one or two relations as i/p and produce a new relation as their result.
- The fundamental operations are select, project, union, set difference, cartesian product and rename.
- other operations - set intersection, natural join, division, & assignment.

(6.2)

Select Operation (σ)

$\sigma_{\theta}(r)$ — θ: condition, r: relation

- Select operation selects tuples that satisfy a given predicate.
- Use Greek letter sigma (σ) to denote selection

Eg: Find all sailors with rating > 8

~~$\sigma_{rating > 8}(s1)$~~

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

- We allow comparisons using $=, \neq, <, \leq, >, \geq$ in the selection predicate.
- We can combine several predicates by using and ($\wedge$), or ($\vee$), not ($\neg$).

Eg: Find the sailors whose rating is > 8 and age is > 35.

$$\sigma_{rating > 8 \wedge age > 35}(sailors)$$

Project Operation (Π)

- It is denoted by Greek letter pi (Π)
- Project used to extract columns from a relation.

Eg: ① List all sailor ids, and sailor names from sailor relation

$$\Pi_{sid, sname}(sailors)$$

Eg: ② Find all sailor names with a rating > 7.

$$\Pi_{sname, rating}(\sigma_{rating > 7}(sailors))$$

Set Operations:

Union (∪)

Intersection (∩)

Set-difference (−)

Cross product (X)

- Union (∪): R ∪ S returns a relation instance containing all tuples that occur in either relation instance R of relation instance S (or both). R and S must be union compatible. ($R \cup S = \{t \mid t \in R \vee t \in S\}$

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

- They have same num of fields &
- Corresponding fields, taken in order from left to right, have the same domains.

Eg: Instance S1 of Sailors

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

Instance S2 of Sailors.

sid	sname	rating	age.
28	Yuppy	9	35.0
31	Lubber	8	55.5
44	Guppy	5	35.0
58	Rusty	10	35.0

Union of S_1 and S_2

i.e $S_1 \cup S_2$

↓

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

- Intersection (∩): $R \cap S$ returns a relation instance containing all tuples that occur in both R and S. $R \cap S = \{ t \mid t \in R \wedge t \in S \}$

Eg: $S1 \cap S2$

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

- set-difference (−): $R - S$ returns a relation instance containing all tuples that occur in R but not in S. $R - S = \{ t \mid t \in R \wedge t \notin S \}$

Eg: $S1 - S2$

sid	sname	rating	age
22	Dustin	7	45.0

- Cross-product (X): R X S returns a relation instance whose schema contains all the fields of R followed by all fields of S. The cross product operation is sometimes called Cartesian product.

Eg: instant R1 of Reserves

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

S1 X R1

↓

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

Renaming: (ρ) (rho)

For example if we take cross product example S1 X R1, in this we have two fields with same name i.e (sid). This fields can be renamed by oldname → newname or position → newname.

Eg: $\rho(C(1 \rightarrow sid1, 5 \rightarrow sid2), S1 \times R1)$

1 sid1	2 sname	3 rating	4 age	5 sid2	6 bid	7 day
22	Dustin	7	45.0	22	101	10/10/96

Joins

- Combining information from two or more relations.
- Although joins can be defined as cross-product followed by selections and projections, joins arise much more frequently in practice than plain cross products.
- Further the cross-product result is much larger than join result.

→ Conditional joins:

The join operation accepts a join condition c and a pair of relation instances as arguments, and returns a relation instance.

The operation is defined as follows:

$$R \bowtie_c S = \sigma_c (R \times S)$$

$\bowtie$ defined as cross product followed by a selection. condition c can refer to attributes of both R and S.

Eg: $S1 \bowtie_{S1.sid < R1.sid} R1$

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

Equi join:

- Equi join is when the join condition consists solely of equalities between two fields in R and S.
- There is some redundancy in retaining both attributes in the result.
- The schema of the result of an equijoin contains the fields of R followed by the fields of S that donot appear in the join conditions.

Eg: $S1 \bowtie_{R.sid = S.sid} R1$

sid	sname	rating	age	sid	bid	day
22	Dustin	7	45.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

Natural Join:

In this join we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields.

The equijoin expression $S1 \bowtie_{R.sid = S.sid} R1$ is actually a natural join and can simply be denoted as $S1 \bowtie R1$, since the only common field is <u>sid</u>.

If the two relations have no attributes in common $S1 \bowtie R1$ is simply the cross product.

Division

Consider an example to understand division.

Consider two relation instances A & B in which A has two fields x & y and B has one field y with the same domain as in A.

→ We define division operation A/B as the set of all x values such that for every y value in B, there is a tuple <x, y> in A.

Eg.

A

sno	Pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

B1

Pno
P2

B2

Pno
P2
P4

B3

Pno
P1
P2
P4

A/B1 →

sno
S1
S2
S3
S4

A/B2 →

sno
S1
S4

A/B3 →

sno
S1

Examples

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

$$\pi_{sname}((\sigma_{bid=103}\ Reserves) \bowtie Sailors)$$

(2) Find the names of sailors who have reserved red boat.

$$\pi_{sname}((\sigma_{color='red'}\ Boat) \bowtie Reserves \bowtie Sailors)$$

(3) Find the colors of boats reserved by dubber

$$\pi_{color}((\sigma_{sname='dubber'}\ sailor) \bowtie Reserves \bowtie Boats)$$

(4) Find the names of sailors who have reserved atleast one boat.

$$\pi_{sname}(sailor \bowtie Reserves)$$

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

$$\rho(Tempboats, (\sigma_{color='red'}\ Boats) \cup (\sigma_{color='green'}\ Boats))$$

$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

(or)

$\rho(Tempboats, (\sigma_{color='red' \lor color='green'} Boats))$

$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$

⑥ Find the names of sailors who have reserved red and green boats.

$\rho(Tempredboats, \pi_{sid}((\sigma_{color='red'} boats) \bowtie reserves))$

$\rho(Tempgreenboats, \pi_{sid}((\sigma_{color='green'} boats) \bowtie reserves))$

$\pi_{sname}((Tempredboats \cap Tempgreenboats) \bowtie Sailors)$

or ? $\rho(Templed, \pi_{sname}((\sigma_{color='red'} boats) \bowtie Reserves \bowtie Sailors)$

$\rho(Tempgreen, \pi_{sname}((\sigma_{color='green'} boats) \bowtie Reserves \bowtie Sailors)$

$\pi_{sname}(Templed \cap Tempgreen)$

This is incorrect because the sailor name is used to identify sailors. In the result we will get Horatio, but no individual Horatio has reserved a red & a green boats.

⑦ Find the names of sailors who have reserved atleast two boats.

$\rho(Reservations, \pi_{sid, sname, bid}(Sailors \bowtie Reserves))$

$\rho(Reservationpairs(1 \rightarrow sid1, 2 \rightarrow sname1, 3 \rightarrow bid1, 4 \rightarrow sid2, 5 \rightarrow sname2, 6 \rightarrow bid2), Reservations \times Reservations)$

$\pi_{sname}(\sigma_{sid1=sid2 \land bid1 \neq bid2}) Reservationpairs$

First we compute tuples of the form (sid, sname, bid), where sailor sid has made a reservation for boat bid; this set of tuples is the temporary relation Reservations. Next we find all pairs of Reservations tuples where the same sailor has made both reservations & the boats involved are distinct boats.

Finally we project the names of such sailors to obtain the answer containing the names Dustin, Horatio, Lubber.

(8) Find the sids of sailors with age > 20 who have not reserved a red boat.

$\Pi_{sid}(\sigma_{age > 20}\ Sailors) - \Pi_{sid}((\sigma_{color = 'red'}\ Boat) \bowtie Reserves \bowtie Sailors)$

(9) Find the names of sailors who have Reserved all boats

$\rho(Tempsids, (\Pi_{sid, bid}(Reserves) / (\Pi_{bid}\ Boats)))$

$\Pi_{sname}(Tempsids \bowtie Sailors)$

(10) Find the names of sailors who have reserved all boats called Interlake.

$\rho(Tempsids, (\Pi_{sid, bid}\ Reserves) / (\Pi_{bid}(\sigma_{bname = 'Interlake'}\ Boats)))$

$\Pi_{sname}(Tempsids \bowtie Sailors)$

Relational Calculus:

- It is a non-procedural or Declarative language.
- It allows us to describe set of answers without being explicit about how they should be computed.
- Relational Calculus
  - Tuple Relational Calculus (TRC) — take on tuples as values
  - Domain Relational Calculus (DRC) — The variables range over field values. (DRC has influence on Query By Example QBE).

TRC has more influence on SQL.

TRC. (Tuple Relational Calculus)

SQL -> tuple Relational Calculus

TRC query has the form

$$\{T \mid p(T)\}$$

where T is a tuple variable and P(T) denotes a formula that describes T.

ex: Find all sailors with a rating above 7.

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

The tuple variable S is instantiated successively with each tuple, & the test S.rating > 7 is applied.

General form of the condition in TRC queries

Atomic expressions:

RE Rel — r(t) — true if t is a tuple in the relation instance r
S ∈ Sailors

- $t_1.A_i$ <comp op> $t_2.A_j$ (R.a, S.b) comp Op is one of $\{<, \leq, >, \geq, =, \neq\}$
  ex: S.sid = R.sid
- $t.A_i$ <comp Op> C (R.a, constant) C is a constant of appropriate type.
  ex: S.rating = 10

Composite expression

1.) Any atomic expression

2.) $F_1 \wedge F_2$, $F_1 \vee F_2$, $\neg F_1$, $F_1 \Rightarrow F_2$ where F_1 & F_2 are expressions (F_2 is true if F_1 is true)

3.) $(\forall t)(F)$, $(\exists t)(F)$ where F is an expression and t is a tuple variable

The quantifiers $\exists$ (for each) and $\forall$ (for all) are said to bind the variable.

$\neg P$, $P \wedge Q$, $P \vee Q$, $P \Rightarrow Q$.

$\forall R(P(R))$

$\exists R(P(R))$

Examples:

① Find the names & ages of sailors with a rating > 7.

$\{P \mid \exists S \in \text{Sailors}\ (S.\text{rating} > 7 \wedge P.\text{sname} = S.\text{sname} \wedge P.\text{age} = S.\text{age})\}$

② Find the sailor name, boat id, and reservation date for each reservation.

$\{P \mid \exists R \in \text{Reserves}\ \exists S \in \text{Sailors}\ (R.\text{sid} = S.\text{sid} \wedge P.\text{day} = R.\text{day} \wedge P.\text{sname} = S.\text{sname} \wedge P.\text{bid} = R.\text{bid})\}$

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

$\{ P \mid \exists S \in sailors\ \exists R \in Reserves\ \exists B \in Boats$
$(B.color = 'red' \wedge R.sid = S.sid \wedge$
$R.bid = B.bid \wedge P.sname = S.sname) \}$

(4) Find the names of sailors who have reserved all boats.

$\{ P \mid \exists S \in Sailors\ \forall B \in Boats$
$(\exists R \in Reserves (S.sid = R.sid \wedge$
$R.bid = B.Bid \wedge P.sname = S.sname)) \}$

(5) Find the names of sailors who have reserved all red boats.

$\{ P \mid \exists S \in Sailors\ \forall B \in Boats (B.color = 'red'$
$\Rightarrow (\exists R \in Reserves (S.sid = R.sid \wedge$
$R.bid = B.bid \wedge P.sname = S.sname)) \}$

(Note): $P \Rightarrow q$ (q is true whenever p is true)
it is logically equivalent to $\neg P \vee q$.

(6) Find the names of sailors who have reserved atleast two boats.

$\{ P \mid \exists S \in Sailors\ \exists R_1 \in Reserves\ \exists R_2 \in Reserves$
$(S.sid = R_1.sid \wedge R_1.sid = R_2.sid \wedge$
$R1.bid \neq R_2.bid \wedge P.sname = S.sname) \}$

Domain Relational Calculus:

- A domain variable is a variable that ranges over the values in the domain of some attribute.
- QBE (Query by Example) — Domain Relational Calculus
- A DRC query has the form

condition over domain variables

$$\{ \langle x_1, x_2, \ldots, x_n \rangle \mid P(\langle x_1, x_2, \ldots, x_n \rangle) \}$$

(forming an n degree relation) (query)

domain variables.

where x_i – is a domain variable or a constant.

$P(\langle x_1, x_2, \ldots, x_n \rangle)$ denotes a DRC formula

The result of this query is set of all tuples $\langle x_1, x_2, \ldots, x_n \rangle$ for which the formula evaluates to true.

An atomic formula in DRC is one of the following:

- $\langle x_1, x_2 \ldots, x_n \rangle \in Rel$ where Rel is a relation with n attributes.

- X op Y
- X op constant, or constant op X

op denote an operator in the set $\{<, >, =, \leq, \geq, \neq\}$

X and Y be domain variables.

A formula is defined recursively as one of the following:

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$ where p and q are themselves a formula.
- $\exists X (p(X))$, where X is a domain variable. $p(X)$ denotes a formula
- $\forall X (p(X))$, where X is a domain variable.

Examples:

① Find all sailors with a rating above 7.

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

② Find the names of sailors who have reserved boat 103.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in sailors \wedge \exists I_r, B_r, D(\langle I_r, B_r, D\rangle \in Reserves \wedge I_r = I \wedge B_r = 103))\}$$

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

$$\{\langle N\rangle \mid \exists I\,(\langle I, N, T, A\rangle \in Sailors \wedge \exists I_r, B_r\,(\langle I_r, B_r, D\rangle \in Reserves\} \wedge \exists B_i, B_c\,(\langle B_i, B_N, B_c\rangle \in Boats \wedge I = I_r \wedge B_r = B_i \wedge B_c = \text{'red'})))\}$$

(4) Find the names of sailors who have reserved all boats.

$$\{\langle N\rangle \mid \exists I\,(\langle I, N, T, A\rangle \in sailors \wedge \forall B_i, B_n, B_c\,(\langle B_i, B_n, B_c\rangle \in Boats\,(\exists I_r, B_r\,\langle I_r, B_r, D\rangle \in Reserves\,(I = I_r \wedge B_r = B_i))))\}.$$

(5) Find the sailors who have reserved all red boats.

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in sailors \wedge \forall \langle B_i, B_n, B_c\rangle \in Boats\,(c = \text{'red'} \Rightarrow \exists I_r, B_r\,\langle I_r, B_r, D\rangle \in Reserves\,(I = I_r \wedge B_r = B))\}$$

UNIT-III

The SQL **SELECT** statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

Syntax

The basic syntax of the SELECT statement is as follows −

```
SELECT column1, column2, columnN FROM table_name;
```

Here, column1, column2... are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax.

```
SELECT * FROM table_name;
```

The SQL **WHERE** clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table. You should use the WHERE clause to filter the records and fetching only the necessary records.

The WHERE clause is not only used in the SELECT statement, but it is also used in the UPDATE, DELETE statement, etc.,

Syntax

The basic syntax of the SELECT statement with the WHERE clause is as shown below.

```
SELECT column1, column2, columnN
FROM table_name
WHERE [condition]
```

You can specify a condition using the comparison or logical operators like >, <, =, **LIKE, NOT**, etc.

Ex:

1. *List the information from sailors table*

SQL> select * from sailors;

2. Displays the sailor names and ratings of each sailor

   SQL>select s.sname , s.rating from sailors s;

3. *find all the sailors with rating above 7?*

SQL> select s.sname from sailors s where s.rating>7;

set operation queries:(Union, Intersect, MINUS or Except)

1. Union

- The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
- In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.
- The union operation eliminates the duplicate rows from its resultset.

Syntax

1. SELECT column_name FROM table1
2. UNION
3. SELECT column_name FROM table2;

2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

Syntax

1. SELECT column_name FROM table1
2. UNION ALL

3. SELECT column_name FROM table2;

3. Intersect

- It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
- In the Intersect operation, the number of datatype and columns must be the same.
- It has no duplicates and it arranges the data in ascending order by default.

Syntax

1. SELECT column_name FROM table1
2. INTERSECT

3. SELECT column_name FROM table2;

4. Minus or EXCEPT

- It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
- It has no duplicates and data arranged in ascending order by default.

Syntax

1. SELECT column_name FROM table1
2. MINUS

3. SELECT column_name FROM table2;

1.find the sids of sailors who have reserved red or a green boat?

- **SQL> select s.sid from sailors s ,reserves r,boats b where s.sid=r.sid and r.bid=b.bid and b.color='red' union select s1.sid from sailors s1,reserves r1,boats b1 where s1.sid=r1.sid and r1.bid=b1.bid and b1.color='green';**

SID

22

31

64

74

- *Same query using* **UNION ALL**

select s.sid from sailors s ,reserves r,boats b where s.sid=r.sid and r.bid=b.bid and b.color='red' ***union all*** *select s1.sid from sailors s1,reserves r1,boats b1 where s1.sid=r1.sid and r1.bid=b1.bid and b1.color='green'*

SID

22

22

31

31

64

22

31

74

2.find the sids of sailors who have reserved both red and green boat?

- **SQL> select s.sid from sailors s ,reserves r,boats b where s.sid=r.sid and r.bid=b.bid and b.color='red' intersect select s1.sid from sailors s1,reserves r1,boats b1 where s1.sid=r1.sid and r1.bid=b1.bid and b1.color='green';**

```
     SID
----------
      22
      31
```

3.find the sids of sailors who have reserved a red boat but not a green boat?

- **SQL> select s.sid from sailors s ,reserves r,boats b where s.sid=r.sid and r.bid=b.bid and b.color='red' minus select s1.sid from sailors s1,reserves r1,boats b1 where s1.sid=r1.sid and r1.bid=b1.bid and b1.color='green';**

```
     SID
----------
      64
```

4.find the sids of sailors who have rating equal to 10 or who have reserved boat no 104?

- **SQL> select s.sid from sailors s where s.rating=10 union select r1.sid from sailors s1,reserves r1 where s1.sid=r1.sid and r1.bid=104;**

```
     SID
----------
      22
      31
      58
      71
```

<u>set compartions operations and nested queries:</u>

Set Comparison Operators

There are different types of set comparison operators like **EXISTS, IN , NOT IN and UNIQUE**. SQL also supports **op ANY** and **op ALL**, where `op` means arithmetic comparison operators such as `<`, `<=`, `=`, `<>`, `>=`, `>`. SOME are also one of the set comparison operators but it is similar to ANY.

➔ The SQL ANY Operator

The `ANY` operator:

- returns a boolean value as a result
- returns TRUE if ANY of the subquery values meet the condition

`ANY` means that the condition will be true if the operation is true for any of the values in the range.

In short, ANY (or SOME) allows you to specify the comparison you want in each predicate, such as X<ANY (A1, A2, A3) is translated to X < A1 OR X < A2 OR X < A3.

Sal<ANY(2000,3000,4000)   sal<2000 or sal<3000 or sal<4000

➔ The SQL ALL Operator

The `ALL` operator:

- returns a boolean value as a result
- returns TRUE if ALL of the subquery values meet the condition
- is used with `SELECT`, `WHERE` and `HAVING` statements

`ALL` means that the condition will be true only if the operation is true for all values in the range.

For illustrative purposes, X != ALL (A1, A2, A3) is translated to X != A1 AND X != A2 AND X != A3.

Sal!=ALL(2000,3000,4000) sal!=2000 and sal!=3000 and sal!=4000

➔ The **EXISTS** condition in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not. The result of EXISTS is a boolean value True or False. It can be used in a SELECT, UPDATE, INSERT or DELETE statement.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
```

```
(SELECT column_name(s)
 FROM table_name
 WHERE condition);
```

Nested Query:

In nested queries, a query is written inside a query. The result of inner query is used in execution of outer query. Various operators like **IN, NOT IN, ANY, ALL etc** are used in writing independent nested queries.

1.find the sids of sialors whose boat no is 103?

- **SQL> select s.sid from sailors s where s.sid in(select r.sid from reserves r where r.bid=103);**

```
  SID
----------
      22
      31
      74
```

2.find the names of sialors who have reserved boat no 101?

- **SQL> select s.sname from sailors s where s.sid in(select r.sid from reserves r where r.bid=101);**

```
SNAME
----------
dustin
horatio
```

3.find the names of sailors who have not reserved red colour boat?

- **SQL> select s.sname from sailors s where s.sid in(select r.bid from reserves r where r.bid not in(select b.bid from boats b where b.color='red'));**

no rows selected

4.find sialors names and ids whose rating is better than some sailors called horatio?

- **SQL> select s.sname,s.sid from sailors s where s.rating>any(select s1.rating from sailors s1 where s1.sname ='horatio');**

```
SNAME          SID
---------- ----------
rusty          58
zorka          71
horatio        74
lubber         31
andy           32
```

5.find the sailors with highest rating?

- **SQL> select s.* from sailors s where s.rating >=all(select s1.rating from sailors s1);**

```
     SID SNAME        RATING      AGE
```

```
---------- ---------- ---------- ----------
      58 rusty            10         35
      71 zorka            10         16
```

corelated nested queries:

In co-related nested queries, the output of inner query depends on the row which is being currently executed in outer query.

Syntax:

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

1.find sids from sailors who reserved boat no 103?

- **SQL> select s.sid from sailors s where exists(select * from reserves r where r.sid=s.sid and r.bid=103);**

```
     SID
----------
      22
      31
      74
```

Ex (1) Find the names of sailors who have reserved boat no 103.

select S.sname from sailors S where EXISTS
(select * from Reserves R where R.bid = 103
AND R.sid = S.sid);

- The EXISTS operator allows us to test whether a set is nonempty.
- For each sailor row, we test whether the set of 'R' rows such that R.bid = 103 and S.sid = R.sid is nonempty. If so, sailor 'S' has reserved boat 103, & we retrieve sailor name.

- The occurrence of 'S' in subquery is called a correlation & sub queries are correlated queries (i.e S.sid).

steps how the query is executed

sid

22 (22 103 day) T

29 (no row) - F

31 (31, 103, day) T

32 (no row) - F

58 (no row) - F

64 (no row) - F

71 (no row) - F

74 (74, 103, day) - T

85 (no row) - F

95 (no row) - F

O/P

Dustin

Lubber

Horatio.

2.find the names of sialors who have reserved all boats ?

- **SQL> select s.sid,s.sname from sailors s where not exists((select b.bid from boats b)minus(select r.bid from reserves r where r.sid=s.sid));**

```
      SID SNAME
---------- ----------
       22 dustin
```

22 $\begin{pmatrix}101\\102\\103\\104\end{pmatrix} - \begin{pmatrix}101\\102\\103\\104\end{pmatrix} \Rightarrow$ 22 not exists (empty / no row) - T

Dustin

29 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix} - \begin{pmatrix}\text{no}\\\text{bids}\end{pmatrix} \Rightarrow$ 29 not exists $\begin{bmatrix}101\\102\\103\\104\end{bmatrix}$ nonempty - F

31 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix} - \begin{pmatrix}102\\103\\104\end{pmatrix} \Rightarrow$ 31 not exists (101) - F

32 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix} - \begin{pmatrix}\text{no}\\\text{bids}\end{pmatrix} \Rightarrow$ 32 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix}$ - F

58 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix} - \begin{pmatrix}\text{no}\\\text{bids}\end{pmatrix} \Rightarrow$ 32 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix}$ - F

64 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix} - \begin{pmatrix}\text{no}\\\text{bids}\end{pmatrix} \Rightarrow$ 64 not exists $\begin{pmatrix}103\\104\end{pmatrix}$ - F

71 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix} - \begin{pmatrix}\text{no}\\\text{bids}\end{pmatrix} \Rightarrow$ 71 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix}$ - F

74 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix} - (103) \Rightarrow$ 74 not exists $\begin{pmatrix}101\\102\\104\end{pmatrix}$ - F

85 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix} - \begin{pmatrix}\text{no}\\\text{bids}\end{pmatrix} \Rightarrow$ 85 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix}$ - F

95 not exists $\begin{pmatrix}101\\102\\103\\104\end{pmatrix} - \begin{pmatrix}\text{no}\\\text{bids}\end{pmatrix} \Rightarrow$ 95 not exist $\begin{pmatrix}101\\102\\103\\104\end{pmatrix}$ - F

Aggregation Operators

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

1.find the average of age of all the sailors?

- **SQL> select avg(s.age) from sailors s;**

```
AVG(S.AGE)
----------
    36.7
```

2.find the average of age of sailors with rating 10?

- **SQL> select avg(s.age) from sailors s where s.rating=10;**

```
AVG(S.AGE)
----------
    25.5
```

3.find the name and age of the oldest sailor?

- **SQL> select s.sname ,s.age from sailors s where s.age=(select max(s1.age) from sailors s1);**

```
SNAME          AGE
---------- ----------
bob           63.5
```

4.count te no of sialors in sialors relation?

- SQL> select count(*) from sailors s;

```
 COUNT(*)
----------
      10
```

5.count the no of distinct sailor names?

- **SQL> select count(distinct(s.sname)) from sailors s;**

```
COUNT(DISTINCT(S.SNAME))
------------------------
                   9
```

6.find the names of sailor who are older than oldest sailors with rating of 10?

- **SQL> select s.sname from sailors s where s.age>(select max(s1.age) from sailors s1 where s1.rating=10);**

```
SNAME
----------
dustin
lubber
bob
```

7.find the max age from sailors relation?

- **SQL> select min(s.age) from sailors s;**

```
MIN(S.AGE)
----------
        16
```

Group by and Having clause:

GROUP BY :

The GROUP BY Clause is utilized in SQL with the SELECT statement to organize similar data into groups. It combines the multiple records in single or more columns using some functions. Generally, these functions are aggregate functions such as min(),max(),avg(), count(), and sum() to combine into single or multiple columns

Syntax:

```
SELECT column1, function_name(column2)

FROM table_name

WHERE condition

GROUP BY column1, column2

ORDER BY column1, column2;
```

function_name: Name of the function used for example, SUM() , AVG().
table_name: Name of the table.
condition: Condition used.

1.find the age of the youngest sailor for each rating level?

- **SQL> select min(s.age),s.rating from sailors s group by s.rating;**

```
MIN(S.AGE)    RATING
---------- ----------
        33          1
      25.5          8
        35          7
      23.5          3
        16         10
        35          9
6 rows selected.
```

Having Clause:

We can use HAVING clause to place conditions to decide which group will be the part of final result-set. Also we can not use the aggregate functions like SUM(), COUNT() etc. with WHERE clause. So we have to use HAVING clause if we want to use any of these functions in the conditions.

Syntax:

```
SELECT column1, function_name(column2)

FROM table_name

WHERE condition

GROUP BY column1, column2

HAVING condition

ORDER BY column1, column2;
```

function_name: Name of the function used for example, SUM() , AVG().
table_name: Name of the table.
condition: Condition used.

1. *find the age of the youngest sailor who is eligible to vote for each rating level with atleast 2 sailors?*

- **SQL> select min(s.age),s.rating from sailors s where s.age>=18 group by s.rating having count(s.rating)>=2;**

MIN(S.AGE)	RATING
25.5	8
35	7
23.5	3

NULL Value

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

Note: A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

Ex:
```
SQL> CREATE TABLE CUSTOMERS(
   ID   INT              NOT NULL,
```

```
    NAME VARCHAR (20)      NOT NULL,
    AGE  INT               NOT NULL,
    ADDRESS  CHAR (25) ,
    SALARY   DECIMAL (18, 2),
    PRIMARY KEY (ID)
);
```

Here, **NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns where we did not use NOT NULL, which means these columns could be NULL.

How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the `IS NULL` and `IS NOT NULL` operators instead.

IS NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

IS NOT NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

Assertions

When a constraint involves 2 (or) more tables, the table constraint mechanism is sometimes hard and results may not come as expected. To cover such situation SQL supports the creation of assertions which are constraints not associated with only one table. And an assertion statement should ensure a certain condition will always exist in the database. DBMS always checks the assertion whenever modifications are done in the corresponding table.

Syntax –

```
CREATE ASSERTION  [ assertion_name ]

CHECK ( [ condition ] );
```

Example –Number of boats plus the number of sailors should be less than 100.

```
CREATE TABLE sailors (sid int,sname varchar(20), rating int,primary
key(sid),

CHECK(rating >= 1 AND rating <=10)

CHECK((select count(s.sid) from sailors s) + (select
count(b.bid)from boats b)<100) );
```

JOINS:

- SQL Join is used for combining col from two or more tables by using values common to both tables.
- A table can also join to itself known as self join.
- The following are the types of joins:

→ CROSS JOIN
→ INNER JOIN or EQUI Join
→ Natural join
→ Outer join
  ↳ Left outer join
  ↳ Right outer join
  ↳ Full outer join

Ex: class table

ID	Name
1	John
2	Paul
4	Alex

class_info table

ID	Address
1	Delhi
2	Mumbai
3	Chennai

cross join

↳ combines each row from the first table with each row of the second table.

select * from class_info cross join class ;

... join or equi join

... this is a simple join in which the ... it is based on matched data as per the equality condition specified in the query.

```
select * from class inner join class_info ~~where~~ on
    class.id = class_info.id;
```

o/p:

ID	Name	ID	Address
1	John	1	Delhi
2	Paul	2	Mumbai
~~4~~	~~Alex~~		

Natural Join

↳ Natural join is a type of inner join which is based on col having same name & same datatype present in both the tables to be joined.

```
select * from class Natural join class_info;
```

o/p

ID	Name	Address
1	John	Delhi
2	Paul	Mumbai

Outer Join.
→ The joined table retains each row - even if no other matching row exists'
→ it is based on both matched & unmatched data.
→ it is further divided into:
Left outer join
Right outer join
Full outer join

Left outer join

→ The left outer join returns a result table with the matched data of two tables then remaining rows of the left table & null for the right table's col.

select * from class left outer join class_info on (class.id = class_info.id);

o/p.

id	Name	id	Address.
1	John	1	Delhi
2	Paul	2	Mumbai
4	Alex	null	null

Right outer join

select * from class right outer join class_info on (class.id = class_info.id);

o/p:

2.18

id	Name	id	Address
1	John	1	Delhi
2	Paul	2	Mumbai
null	null	3	chennai

Full outer join

↳ it returns a result table with the matched data of two tables then remaining rows of both left table & then the right table

select * from class full outer join class_info on (class.id = class_info.id);

o/p.

id	Name	id	Address
1	John	1	Delhi
2	Paul	2	Mumbai
4	Alex	null	null
null	null	3	Chennai

Complex Integrity constraints in SQL:

↳ constraints over a single table:

We can specify complex constraints over a single table using table constraints, which have the form CHECK conditional-expression.

Ex:- ① rating must be between 1 to 10.

create table sailors (sid integer,
sname char(10), rating integer, age
real, Primary key (sid),
CHECK(rating >=1 AND rating <= 10));

Ex: ② enforce the constraint that Interlake boats cannot be reserved.

create table reserves (sid integer, bid integer,
day date, Foreign key (sid) references
sailors (sid), Foreign key (bid) references
boats (bid), CONSTRAINT noInterlakeRes
CHECK ('Interlake' <> (select b.bname
from boats b where
b.bid = reserves.bid)));

Domain Constraints:

A user can define a new domain using the CREATE DOMAIN statement, which makes use of CHECK constraints.

Ex: ① CREATE DOMAIN ratingval INTEGER
DEFAULT 0 CHECK (VALUE >=1 AND
VALUE <= 10);

INTEGER is the base type for the domain ratingval, and every ratingval value must be of this type. Values in ratingval are further restricted by using a CHECK constraint. 2.19

VALUE - to refer to a value in the domain.

we can use in schema declaration as

ex: ① create table sailors (sid integer primary key, sname char(10), rating ratingval, age number (5, 2));

ex: ② create domain id_val int constraint id_test check (value >100);

create table stud (sid id_val primary key, sname varchar2(10), sage number(2));

Assertions:

↳ Integrity constraints over several tables.

↳ An assertion is a named constraint that may relate to the content of individual rows of a table, to the entire contents of a table, or to a state required to exist among a num of tables.

→ An assertion is satisfied if and only if the specified <search conditions> is not false.

- Assertions are similar to check constraints, but unlike ~~to~~ check constraints they are not defined on table or col level but are defined on schema level.

Syntax:

create ASSERTION <assertion_name> CHECK <search condition>

Ex: number of boats plus the number of sailors should be less than 100.

CREATE ASSERTION sailor_boats CHECK
((select count (s.sid) from sailors s)
+ (select count (b.bid) from boats b)
< 100);

TRIGGERS AND ACTIVE DATA BASES:

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are in fact written to be executed in response to any of the following events:

- A database manipulation (DML) stmt. (DELETE, INSERT, or UPDATE).
- A database definition (DDL) stmt (CREATE, ALTER, or DROP).

- Triggers could be defined on the table, view, schema or database with which the event is associated.

- A database that has a set of associated triggers is called an active database.

- A trigger description contains 3 parts:
→ Event: A change to the db that activates the trigger (An insert, delete or update stmt could activate a trigger

2-20

→ condition: A query or test that is run when the trigger is activated.

→ Action: A procedure that is executed when the trigger is activated & its condition is true.

Syntax:

CREATE or REPLACE TRIGGER Trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT or | UPDATE OR | DELETE }
[OF col_name] ON Table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
Declaration_stmts
BEGIN
Executable stmts
EXCEPTION
Exception_handling_stmts
END;

Types of Triggers

- Row trigger: a trigger is fired once for each row affected by the insert (or) update (or) delete.
- statement trigger: ↳ once it is fired independent of num of rows

where

- create [OR] Replace TRIGGER trigger_name: creates or replaces an existing trigger with trigger_name.
- {BEFORE | AFTER | INSTEAD OF}: - specifies when the trigger would be executed. INSTEAD OF is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE}: specifies the DML operation.
- {OF col_name} specifies the col_name that would be updated.
- {ON table_name}: specifies the name of the table associated with the trigger.
- {REFERENCING OLD AS o NEW AS n}
  This allow you to refer new & old values for various DML stmts like INSERT, UPDATE and DELETE.
- {FOR EACH ROW}: specifies a row level trigger, i.e the trigger would be executed for each row being affected.
- WHEN (condition): provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers.

Example: ①

To write a trigger that inserts or updates values of ename & job as uppercase strings even if we give lowercase strings.

Sol

```
create or replace trigger uppername before
INSERT OR UPDATE on emp
for each row

begin
:new.ename:=upper(:new.ename);
:new.job:=upper(:new.job);
end;
```

when the above code is executed it produces the following result:

Trigger created.

sql> insert into emp values (101, 'smith', 'clerk', 7499, '12 mar 1990', 2000, null, 40);

it will convert ename & job to uppercase letters. (i.e SMITH, CLERK).

sql> update emp set job='manager' where ename ='SMITH';

Ex: (2) Write a col level trigger that doesn't allow a salary to be updated if the employee commission is null.

sol

```
create or replace trigger empupdatesal before
update of sal on emp
for each row

begin
if :old.comm is null then
raise_application_error (-201000, 'commis
      null, sal can't be updated');

end if;
end;
```

Trigger created.

```
sql > update emp set sal= 3000 where
empno= 7902;
```

Error at line 1:
ORA -201000 : comm is null, sal can't be updated.

error during execution of trigger 'SRS. empupdatesal',

Drop Trigger:

Drop Trigger trigger_name;

Assertions vs. Triggers:

Assertions	Triggers.
1) Assertions do not modify the data, they only check certain conditions.	① Triggers are more powerful because they can check conditions & also modify the data.
2) They are not linked to specific tables in the db & not linked to specific events	② They are linked to specific tables & specific events
3) All assertions can be implemented as triggers	③ Not all triggers can be implemented as assertions
4) Ex:	Ex:

→ Statement level triggers vs row level

Statement level Triggers	Row level triggers.
1) trigger will be fired only once irrespective of the no of records getting affected.	will fire to the no of records getting affected.
2) By default, stmt level	"for each row."
3) Can't use the co-relation identifiers (:old & :new)	we can use
4) Only once always it will be fired even though there are no rows get affected.	No fire if no row affects.
5) Ex:	Ex:

```
create or replace trigger
    "empt" before
Update of gender on
            emp

begin
dbms_output.put_line('stmt level
            trigger executed');

end;
```

2

Introduction to Schema Refinement:

Refinement approach is based on decompositions. Although decomposition can eliminate redundancy, it can lead to problems of its own and should be used with caution.

→ **Problems caused by Redundancy:**

storing data redundantly in more than one database can lead to several problems:

- Redundant storage - information is stored repeatedly
- update anamolies - if one copy of repeated data is updated, an inconsistency is created unless all copies are similarly updated.
- Insertion anamolies - It may not be possible to store some information unless some other information is stored as well.
- Deletion anamolies - It may not be possible to delete some information without losing some information as well.

Ex: Hourly - Emps schema

Hourly - Emps (eid , ename, rating, hrly - wages, hrs - worked),

here eid is primary key.

hrly - wages attribute is determined by rating attribute.

i.e for a given rating value there is only one hrly_wages value. This integrity constraint is an example of Functional dependency.

Instance of Hrly-emps relation.

eid	ename	rating	hrly_wages	hrs_worked
101	John	8	10	40
102	smith	8	10	30
118	Alex	5	7	30
123	Paul	5	7	32
134	dubber	8	10	40

The value that appears in the rating col of two tuples, the integrity constraint tells that the same value must appear in hrly_wages column as well.

Example of Redundant storage :-

The rating value 8 corresponds to the hrly-wage 10, and this association is repeated 3 times.

Example for Update anamoly:

The hrly_wages in the 1st tuple could be updated without making similar change in the second tuple.

Example of insertion anamoly:

we cannot insert a tuple for an employee unless we know the hrly-wage for the employee's rating value.

Example of deletion anamoly:

If we delete all tuples with a given rating value we lose the association between that rating value & its hrly-wage value.

We can decompose Hrly_Emps relation into

Hrly_Emps1 (eid, ename, rating, hrs_worked)

eid	ename	rating	hrs_worked
101	John	8	40
102	Smith	8	30
118	Alex	5	30
123	Paul	5	32
134	Lubbel	8	40

wages (rating, Hrly_wages)

rating	Hrly wages
8	10
5	7

Functional Dependency:

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

If 'R' is a relation with attributes X and Y, a FD between the attributes is represented as X → Y, which specifies Y is functionally dependent on X. Here X is a determinant and Y is a dependent attribute.

Each value of X is associated precisely with one Y value.

The left-hand side attributes determine the values of attributes on the right hand side.

Ex:

N Name	R Roll NO	C CGPA
A	R_1	7.6
B	R_2	5.5
C	R_3	9.2
A	R_4	9.1
B	R_5	8.7

Now if we have a question as "what is CGPA of A?" we could not answer because there are two students with name A.

Now if we have a question as "what is CGPA of R,"
then it is clear that 7.6, as the rollno will be unique.

Now we can say that

R (rollno) $\longrightarrow C$ (CGPA)
R (rollno) $\longrightarrow N$ (name)
} Functional Dependency.

Ex: ① Account relation :

A Acct no	B Branch	Bl Balance
A101	Downtown	500
A 102	Perryridge	400
A 201	Brigton	900
A 215	Hianus	700
A 217	Brigton	750
A 222	Redwood	700
A 305	Round hill	350

A (Acctno) $\rightarrow Bl$ (Balance) — True.

B (Branch) $\longrightarrow A$ — False

$A \rightarrow B$ — True

$B \rightarrow Bl$ — False

$Bl \rightarrow A$ — False

Ex: ④

A	B	C	D	E
a1	b1	c1	d1	e1
a2	b1	c2	d2	e1
a3	b2	c1	d1	e1
a4	b2	c2	d2	e1
a5	b3	c3	d1	e1

$A \rightarrow B$, $A \rightarrow E$, $B \rightarrow E$
$C \rightarrow E$, $AB \rightarrow E$,
$BC \rightarrow ADE$
(All are True)

Ex: ③

A	B	C	D	E
a	2	3	4	5
2	a	3	4	5
a	2	3	6	5
a	2	3	6	6

$A \rightarrow BC$ — T
$DE \rightarrow C$ — T
$C \rightarrow DE$ — F
$BC \rightarrow A$ — T.

Ex: ②

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_2	c_2	d_3
a_3	b_3	c_2	d_4

$A \rightarrow C$ — T
$C \rightarrow D$ — F
$AB \rightarrow D$ — F
$C \rightarrow A$ — F
$A \rightarrow B$ — F

{ ∵ we can have $a_2, a_3 \rightarrow c_2$ ✓
but we can't have $a_2 \rightarrow c_3, c...$ ✗

Scanned with CamScanner

(3.3)

Axioms or rules (Inference Rules) - It can apply to a set of FD to derive other FDs.

Armstrong axioms: (a person who proposed first).

Axiom Name	Axiom	Example
(1) Reflexivity rule	if $Y \subseteq X$ then $X \to Y$	eid → ename. $X = \{a,b,c,d,e\}$ $Y = \{a,b,c\}$.
(2) Augmentation i.e adding attributes in dependencies does not change the basic dependencies	if $X \to Y$ holds and Z is a set of attribute then $XZ \to YZ$	eid → ename phone eid, phone → ename, phone. R(A,B,C,D) if A → B then AC → BC
(3) Transitivity	if $X \to Y$ holds & $Y \to Z$ holds then $X \to Z$	eid → zip zip → city then eid → city
(4) Union/Additive	if $X \to Y$ & $X \to Z$ then $X \to YZ$	eid → ename eid → zip then eid → ename, zip
(5) Pseudo Transitivity	if $X \to Y$ and $YZ \to W$ ~~and~~ then $XZ \to W$	addr → proj proj, day → amt then addr, day → amt.
(6) Decomposition / Productive rule	if $X \to YZ$ then $X \to Y$ $X \to Z$	eid → ename, zip. then eid → ename eid → zip.

Fully and Partial Functional Dependency:

→ **Fully FD :** It is defined as attribute Y is Fully FD on an attribute X, if it is FD on X and not FD on proper subset of X.

if X → Y

i.e if X determines Y

i.e Y is fully functional dependent on X

i.e Y cannot be determined by any of the proper subset of X.

Ex: R(ABCD)
ABC → D

if D can't be determined by any of the propersubsets you think.

Proper subsets not present are { BC → D, C → D, A → D }.

So as the above propersubsets are not present D is fully FD on ABC.

→ **Partial FD.**

Ex: AC → P
A → D } Proper subsets.
D → P

AC → P.

P is not fully FD on AC since if we find A+ (A closure)
ADP.

without C also we can determine P.

So we can write $AC \rightarrow P$ as $A \rightarrow P$ by this we can tell that P is partially functional dependent on A.

Trivial Functional Dependency:

trivial means always true.

- If a FD $X \rightarrow Y$ holds, where Y is a subset of X ($Y \subseteq X$) then it is trivial FD.
- A trivial FD is the one where RHS is a subset of LHS.

Ex: if R(A, B) then

$A \rightarrow A$
$AB \rightarrow A$
$AB \rightarrow BA$
$AB \rightarrow AB$

} Trivial FDs.

Ex: eid → eid
eid, ename → eid
eid, ename → ename.

Ex: Take a Relation R

A	B
1	1
2	1
1	3
4	2

$A \rightarrow B$ NOT FD
$B \rightarrow A$ NOT FD.

$AB \rightarrow A$ — FD..

A	B	A
1	1	1
2	1	2
1	3	1
4	2	4

if $AB \rightarrow C$

A	B	C
1	1	5
2	1	4
1	1	6

This is not FD since

A	B	→ C	
1	1	5	✗
1	1	6	✗

Non trivial FD - If an FD $X \to Y$ holds, where Y is not a subset of X ($Y \not\subseteq X$) then it is called a non-trivial FD.

(or)

if $X \to Y$ is FD and $X \cap Y = \phi$ then it is non trivial FD.

Closure of Attributes

- Closure of attributes (X^+) is a set of attributes which can be determined using X.
- It is useful to find out a candidate key.

Ex: (1) R(A, B, C, D)

$A \to B$

$B \to D$

$C \to B$

what is closure of A^+.

$A^+ = A\ B\ D.$

$\therefore A^+$ is not a good candidate key, since it doesnot have all attributes.

Ex: (2). R(A B C D E) $F = \{A \to D, D \to B, B \to C, E \to B\}$.

what is closure of A^+, BD^+, AE^+ ?

$A^+ = \{ADBC\}$

$BD^+ = \{BDC\}$

$AE^+ = \{AEDBC\}$

As AE^+ is set of all attributes, AE^+ is a good candidate key.

Scanned with CamScanner

Ex: 3 R(A B C D E F G)

A → B

BC → DE

AEG → G   what is AC^+ ?

$AC^+ = ACBDE$.

Ex: (4) R(A B C D E)

A → BC

CD → E

B → D

E → A

what is B^+

list of candidate keys

{ A^+, CD^+, E^+ } candidate keys.

$B^+ = BD$

Ex: (5) The following FDs are given { AB → CD, AF → D, DE → F, C → G, F → E, G → A }

which of the following option is False ?

[a] $\{CF\}^+ = \{ACDEFG\}$ ✓

[b] $\{BG\}^+ = \{ABCDG\}$ ✓

[c] $\{AF\}^+ = \{ACDEFG\}$ ✗

∴ answer is option [c]

Ex: (6) R (ABCDEF)

AB → C, BC → AD, D → E, CF → B

$AB^+ = ?$

$AB^+ = ABCDE$

Find out a good candidate key -?

$CF^+ = CFBADE$.

(candidate key).

Closure of Functional Dependency:

Closure of set of FDs F is set of all FDs that include F as well as all FDs that can be inferred from F.

Note: if R has n attributes there are 2^{n+1} Possible FDs.

Ex: R has 2 attributes then

$$2^{2+1} = 2^3 = 8 \text{ FDs}.$$

FD closure is denoted as F^+

The Armstrong axioms can be applied repeatedly to compute all FDs implied by a set of 'F' of FDs.

Ex. R (A B C G H I)

$F = \{A \to B,\ A \to C,\ CG \to H,\ CG \to I,\ B \to H\}$

Find out F^+.

Sol

$F^+ =$

$A \to H$ from transitivity rule — if $A \to B$ & $B \to H$ then $A \to H$.

$CG \to HI$ from union
$CG \to H$
$CG \to I$

$AG \to H$. from pseudo transitivity rule.
if $A \to C$ and $CG \to H$ then
$AG \to H$

$AG \to I$ from pseudo transitivity
if $A \to C$ & $CG \to I$ then
$AG \to I$.

$\therefore F^+$ (Closure of FDs) $= \{A \to B,\ A \to C,\ CG \to H,\ CG \to I,\ B \to H,\ A \to H,\ CG \to HI,\ AG \to H,\ AG \to I\}$

Examples on Closure of Attributes.

Ex: (7) R = {E, F, G, H, I, J, K, L, M, N} find out a candidate key for R.

EF → G
F → IJ
EH → KL
K → M
L → N.

(a) E, F ✓(b) EFH (c) EFHKL (d) E.

Sol:

$EF^+ = EFGIJ$.
$EFH^+ = EFHGIJKLMN$.
$EFHKL^+ = EFHKLIJMNG$.
$E^+ = E$

Candidate keys are EFH^+ & $EFHKL^+$.
but EFH^+ is minimal it is a good candidate key.

Ex: (8) R(A, B, C, D, E) FDs are A → B, A → C, CD → E, B → D, E → A
which of the following FDs is not implied by the below set.

(a) CD → AC ✓(b) BD → CD (c) BC → CD (d) AC → BC

Sol:

$CD^+ = CDEAB$ $AC^+ = ACBDE$
$BD^+ = BC$
$BC^+ = BCDEA$

{∵ BD → CD can't hold true this FD is not implied in FD set?}

(9) $R(A, B, C, D, E)$.

$A \to BC$, $C \to B$, $D \to E$, $E \to D$.

Calculate closure of attributes.

Sol:

$A^+ = ABC$

$B^+ = B$

$C^+ = C, B$

$D^+ = D, E$

$E^+ = E, D$.

In this case, a single attribute is unable to determine all the attributes on its own.

Hence we need to combine two or more attributes to determine the candidate keys.

$\{A, D\}^+ = ABCDE$

$\{AE\}^+ = AEBCD$.

$\therefore$ AD & AE are two candidate keys.

(1) $R(A, B, C, D, E)$

F: $A \to BC$

$CD \to E$

$B \to D$

$E \to A$

Find closure of FD.

Sol. $A \to B$, $A \to C$ {∵ decomposition}

$CD \to A$ {Transitivity}

(2) $A(ABCDEF)$ $\{A \to B, B \to CD, A \to C, CE \to F, B \to D\}$.

Sol

$A \to BC$ — union.

$B \to D$ — decom

$A \to D$ — Trans

$AE \to F$ — pseudo

$B \to C$ — decomp.

---

10) $R(A, B, C, D, E, F, G, H)$ $F = \{A \to BC, CH \to G, B \to CFH, E \to A\}$

Calculate closure of attributes & find out candidate keys.

Here take the attributes which are not present in RHS and find closure of it.

i.e DE^+

$DEABCFHG$

So candidate key is DE^+.

Scanned with CamScanner

Canonical Cover / Minimal cover of set of FDs.

Canonical cover (F_c) of F is a minimal set of FDs equivalent to F such that it follows 3 rules:

1. Singleton on RHS.
2. Removal of extraneous attribute on LHS
3. Removal of redundant FDs.

Ex: R (A, B, C) F = {A → B, B → C, A → C}

find canonical cover of F.

from transitivity rule

A → B

B → C

we can write A → C.

So there is no need of A → C.

∴ Canonical cover of F is

F_c = {A → B, B → C}.

Ex: (2). Find out minimal cover of FDs

F = {A → B
AB → C
D → AC
D → E}

G = {A → BC
D → AB}

(a) F covers G (b) G covers F (c) F & G are equal

(d) None.

find out which one is true.

1st we will take F.

Step 1: in F we have to make all FDs singleton.

F = A → B
AB → C
D → A
D → C
D → E

Step 2: Now find for extraneous attributes on LHS.

F = A → B
AB → C
D → A
D → C
D → E

Check for FDs which have two or more attributes on LHS

find $\frac{A^+}{ABC}$ $\frac{B^+}{B}$

as from A we can findout B.

A ~~B~~ → C
is extraneous attribute.

∴ A → B
A → C
D → A
D → C
D → E

∴ in LHS & RHS we don't have extraneous attribute FD.

Step 3: Removal of Redundant FD.

A → B
A → C
D → A
D → C
D → E

if A → B is not existing you think
Now check we can find B from remaining FDs.

find $\frac{A^+}{AC}$ B is not found so A → B is necessary

Now find for next one i.e $A \to C$.

$\frac{A^+}{AB}$ we can't find C so $A \to C$ is necessary.

$D \to A$ remove

$\frac{D^+}{DCE}$ so $D \to A$ is necessary

$D \to C$ remove

$\frac{D^+}{DAEBC}$ here we can remove $D \to C$ as 'C' can be obtained from $A \to C$.

$D \to E$ remove

$\frac{D^+}{DACB}$ so $D \to E$ is necessary.

So final FDs are

$A \to B$
$A \to C$
$D \to A$
$D \to E$

Now same way check for G $\{A \to BC, D \to AB\}$.

Convert to singleton

$A \to B$
$A \to C$
$D \to A$
$D \to B$

as here we don't have any extraneous attribute check for Redundant FDs.

here $D \to B$ is redundant. So if we remove it we will have
$A \to B$, $A \to C$, $D \to A$.

∴ Answer for the question is option (a).
i.e F covers G.

Decomposition

Ex: Find out minimal cover of R(ABCDEFGHK)

F = ABH → CK
A → D
C → E
BGH → F
F → AD
E → F
BH → E

Step 1:

ABH → C
ABH → K
A → D
C → E
BGH → F
F → A
E → F
BH → E
F → D

Step 2:

BH → C
BH → K
A → D
C → E
BGH → F
F → A
E → F
BH → E
F → D.

Step 3:

$\frac{BH^+}{BHKEFAD}$ $\frac{BH^+}{BHCEFAD}$ $\frac{A^+}{A}$ $\frac{C^+}{C}$

$\frac{BGH^+}{BGHCKEFAD}$ $\frac{F^+}{FD}$ $\frac{E^+}{E}$ $\frac{BH^+}{BHCKE}$ $\frac{F^+}{FAD}$

Minimal cover of FD is.

BH → C
BH → K
A → D
C → E
~~BGH → F~~
F → A
E → F

Ex: R (ABCDEFG)

$A \rightarrow B$

$ABCD \rightarrow E$

$EF \rightarrow G$

$EF \rightarrow H$

$ACDF \rightarrow EG$.

find out minimal cover ~~of~~ ?

Sol

Step 1: $A \rightarrow B$

$ABCD \rightarrow E$

$EF \rightarrow G$

$EF \rightarrow H$

$ACDF \rightarrow E$

$ACDF \rightarrow G$.

Step 2:

$A \rightarrow B$.

A~~B~~CD → E   { ∵ A → B }

extraneous

$ACD \rightarrow E$

~~E~~F → G

~~E~~F → H

$ACDF \rightarrow E$

$ACDF \rightarrow G$.

$A \rightarrow B$

$ACD \rightarrow E$

$F \rightarrow G$, $F \rightarrow H$

$ACDF \rightarrow E$

$ACDF \rightarrow G$

Step 3: we can remove $ACDF \rightarrow E$, $ACDF \rightarrow G$

since E and G are determined by

$ACD \rightarrow E$

$F \rightarrow G$.

∴ minimal cover for F is

$A \rightarrow B$,

$ACD \rightarrow E$

$F \rightarrow G$

$F \rightarrow H$.

Ex: R(ABCDE)

ABCD → E
E → D
A → B
AC → D.

step 1:

ABCD → E
E → D
A → B
AC → D.

step 2: as B D determined from

E → D
A → B.

we can remove

A~~B~~C~~D~~ → E.

step 3.

AC → E ✓
E → D ✓
A → B ✓
AC → ~~B~~D. ✗.

$AC^+ = ACBD.$

$E^+ = E.$

$A^+ = A$

$AC^+ = ACEDB.$

minimal cover of FDs.

AC → E
E → D
A → B.

Decomposition:

Dividing the relations into sub relations.

- A decomposition of a relation schema R consists of replacing the relation schema by two or more relation schemas that each contain a subset of the attributes of R and together include all attributes in R.
- If on combining the sub relations, if we get back the original relation then it is loss less join decomposition.

Ex: Hrly_Emp(eid, ename, rating, hr_wages, hr_work);

eid	ename	rating	hr_wages	hr_work
101	John	8	10	40
102	Paul	8	10	30
203	Alex	5	7	30
301	John	5	7	20
208	Bob	8	10	40

we can decompose this relation into

hrly_empl

eid	ename	rating	hr_work
101	John	8	40
102	Paul	8	30
203	Alex	5	30
301	John	5	20
208	Bob	8	40

wages

rating	hr_wages
8	10
5	7

Now to get back original relation we can join these two relations with natural join.

(i.e Hrly_empl ⋈ wages)

here we can get back the original relation i.e Hrly_Emp so this is a lossless join decomposition.

The relation is decomposed to smaller relations to eliminate anamolies caused by redundancy.

Problems related to decomposition:

We should be careful while decomposing a relational schema otherwise it may create more problems.

Two important questions may be asked repeatedly:

① Do you need to decompose a relation?

② What problems does the given decomposition cause?

- To help with 1st question, several normal forms have been proposed for relations. If the relation schema is in one of the normal forms then certain kinds of problems cannot arise.

- with respective to second question, two properties of decomposition are of particular interest.

(a) Loss less join Decomposition

(b) Dependency preserving Decomposition.

(a) Loss less join Decomposition:

In this we should check whether a decomposition allows us to recover the original relation from the decompos smaller relations.

$$\Pi_x(r) \bowtie \Pi_y(r) = r$$

(OR)

3.9 a

To check for lossless join decomposition using FD set, following conditions must hold:

step (1) Union of attributes of R1 and R2 must be equal to attribute of R. Each attribute of R must be either in R1 or in R2.

$$Attr(R1) \cup Attr(R2) = Attr(R)$$

step (2) Intersection of Attributes of R1 and R2 must not be NULL

$$Attr(R1) \cap Attr(R2) \neq \phi$$

step (3) Common attribute must be a key for atleast one relation (R1 or R2).

$$Attr(R1) \cap Attr(R2) \rightarrow Attr(R1) \quad (or)$$

$$Attr(R1) \cap Attr(R2) \rightarrow Attr(R2)$$

Ex: R(A, B, C, D)

F = A → BC.

Relation R is decomposed into R1(ABC) R2(AD)

is it a lossless join decomposition or not.

Sol

step1: $Attr(R1) \cup Attr(R2) = Attr(R)$.

$(ABC) \cup (AD) = (ABCD)$

2: $Attr(R1) \cap Attr(R2) \neq \phi$

$(ABC) \cap (AD) \neq \phi$ (A)

3: $Attr(R1) \cap Attr(R2) = Attr(R1)$ or $Attr(R2)$

$(ABC) \cap (AD) = A \rightarrow$ key of R1(ABC)

because $A \rightarrow BC$ is given.

Ex:

R.

S	P	D
S_1	P_1	D_1
S_2	P_2	D_2
S_3	P_1	D_3

Decomposed to

R_1

S	P
S_1	P_1
S_2	P_2
S_3	P_1

R_2

S	D
S_1	D_1
S_2	D_2
S_3	D_3

natural join
$R_1 \bowtie R_2$

S	P	D
S_1	P_1	D_1
S_2	P_2	D_2
S_3	P_1	D_3

This is loss less join decomposition, because we are able to get original relation R.

For example if we decompose same relation R into

R_1

S	P
S_1	P_1
S_2	P_2
S_3	P_1

R_2

P	D
P_1	D_1
P_2	D_2
P_1	D_3

$R_1 \bowtie R_2$

S	P	D
S_1	P_1	D_1
S_2	P_2	D_2
S_3	P_1	D_3
S_1	P_1	D_3
S_3	P_1	D_1

} extra tuple are obtained

After performing natural join, original table is not regained. This is lossy decomposition.

Note: Let 'R' be a relation & 'F' be a set of FDs that hold over R. The decomposition of 'R' into relations with attribute sets R_1 and R_2 is lossless if F^+ contains either the FD $R_1 \cap R_2 \rightarrow R_1$ (or) $R_1 \cap R_2 \rightarrow R_2$.

(b) Dependency Preserving Decomposition

- If each FD specified $(X \rightarrow Y)$ in F either appears directly in one of the relations in the decomposition or could be inferred from the dependencies that appear in some R_i then it is Dependency Preserving Decomposition.

- It is not necessary that all the dependencies from the relation are appeared in some relation R_i. It is sufficient that the union of the dependencies on all the relations R_i be equivalent to the dependencies on R.

Ex: R(A B C D)

FDs
1. $A \rightarrow B$
2. $B \rightarrow C$
3. $C \rightarrow D$

R_1 (A B C) R_2 (C D)

find out R_1 & R_2 are dependency preserving or not.

R_1 (A B C) — FD1 (A, B), FD2 (B, C)

R_2 (C D) — FD3 (C, D)

as all FDs are existing it is dependency preserving decomposition.

Ex: R (A B C D)

FD1 A → B

FD2 B → C

FD3 C → D

R_1 (A C D) — FD3 R_2 (B D) — no FD.

This is not Dependency preserving decomposition.

Ex: R (A B C D E G)

FD1 AB → C

2 AC → B

3 AD → E

4 B → D

5 BC → A

6 E → G

R_1 (A B) R_2 (BC) R_3 (A B D E) R_4 (E G)

FD1 (AB → C) is not present in any of the decomposed relations.

FD2 also not present

FD5 also not present.

FD3, 4, 6 are present

As FD1, FD2, FD5 are not present it is not dependency preserving decomposition.

Ex: ① Consider a schema R(A, B, C, D) & FD = A → B, C → D

Then the decomposition of R into R1(AB) & R2(CD) is.

(a) dependency preserving & lossless join
(b) lossless join but not dependency preserving
(c) dependency preserving but not lossless join
(d) not dependency preserving and not lossless join.

Sol.

(AB) ∪ (CD) = ABCD True.

(AB) ∩ (CD) = ∅ violates

So it is not lossless join decomposition.

R1 (AB) — FD1

R2 (CD) — FD2.

it is dependency preserving decomposition.

Answer is option c.

Ex: ② R(A, B, C, D) be a relational schema with the following FDs: A → B, B → C, C → D & D → B.

decomposition of R into (A, B) (B, C) (B, D).

(a) lossless & dependency preserving
(b) lossless but not dependency preserving
(c) does not give a lossless join, but dependency preserving
(d) does not give a lossless, ~~does~~ not dependency preserving.

option is (A)

B^+ = BCD

C^+ = CDB

D^+ = DBC

So B → C (or)
C → B

D → B or
B → D }

C → D

(3)

(5.11 a).

Ex: R(A, B, C, D).

FD = {A → B, B → C, C → D, D → B}.

The decomposition of R into (A, B) (B, C) (B, D)

Check whether this decomposition is dependency preserving or not.

$R_1(A, B)$

A → B.

$R_2(B, C)$

B → C

C → B

$R_3(B, D)$.

~~B → D~~.

D → B.

B → D.

C → B } C → D.
B → D

Hence given decomposition is dependency preserving.

Normalization :

Normalization of data can be considered as a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of minimizing redundancy and minimizing the insertion, deletion and update anomolies.

Given a relation schema, we need to decide whether it is good design or we need to decompose it into smaller relations. Such a decision must be guided by an understanding of what problems, if any arise from the correct schema. To provide such guidance, several normal forms have been proposed.

The normal forms based on FDs are first normal form, second normal form, third normal form and Boyce codd normal form (BCNF).

These forms have increasingly restrictive requirements : Every relation in BCNF is also in 3NF, every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF.

First Normal Form :

A relation is in 1st NF if

- values of each attribute is atomic
- No composite values
- All entries in any col must be of same kind
- Each col name must have a unique name
- No too rows are identical.

Ex: Student Relation.

Sid	Name	Courses
101	John	DBMS, CN CD, SE
102	Amit	CO, OS
103	Arpit	CD, OS, DBMS, CN

Convert above relation to 1st normal form.

1st NF →

Sid	Name	Courses
101	John	DBMS
101	John	CN
101	John	CD
101	John	SE
102	Amit	CO
102	Amit	OS
103	Arpit	CD
103	Arpit	OS
103	Arpit	DBMS
103	Arpit	CN

Note: Using 1st NF data redundancy increases, as there will be many cols with same data in multiple rows but each row as a whole will be unique.

Second Normal Form (2nd NF):

A relation is said to be in 2nd NF if it is in
1st NF and
All non-prime attributes are fully functional dependent on any primary key attribute of relation R.

(or)

A table is in 2nd NF it should be in 1st NF and there is no partial dependency $X \rightarrow a$ is partial dependency
if X is proper subset of same candidate key.
'a' is non prime or non key attribute.

Ex: Student Relation.

sid	sname	Prof-id	Prof-Name	Grade
101	Amit	201	John	5
102	Smit	202	Paul	4
103	Miller	203	Miller	6

- This table is in 1st NF since all attributes are single valued.
- But it is not in 2NF since if student 101 is deleted, then leaves the college then the tuple is also lost. professor information is since this attribute is Fully FD on primary key sid.
- To solve this, we must decompose the table into
  stud-details (sid, sname)
  prof-details (pid, prof-name)
  grade-details (sid, pid, grade)

As the grade relation is necessary to combine two tables we are decomposing it into new tables.
Now if student tuple is deleted no info is lost.

Ex: (2)

teacher_id	subject	teacher_age
111	Maths, physics	38
222	Biology	38
333	physics Chemistry	40

convert to 1^{st} NF

teacher_id	subject	teacher_age
111	Maths	38
111	physis	38
222	biology	38
333	physics	40
333	Chemistry	40

candidate keys (teacher_id, subject)
Non prime attribute (teacher_age)
it is in 1^{st} NF but not in 2^{nd} NF because non-prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate keys.
To make this table to 2^{nd} NF decompose to

R_1

teacher_id	teacher_age
111	38
222	38
333	40

R_2

teacher_id	subject
111	Maths
111	Physics
222	biology
333	Physics
333	chemistry.

Now the tables are in 2^{nd} NF.

Ex: (3) R(A,B,C,D)

A → B
C → D.

Sol:

$A^+ = AB$

$C^+ = CD$

$AC^+ = ABCD$

AC is candidate key.

Prime attribute	Non-prime attribute.
A	B
C	D.

The relation R is not in 2nd NF. ~~since~~

(~~Prime attribute should be belonging to atleast one candidate key~~).

Third Normal Form: (3rd NF).

A database is in 3rd NF if it satisfies the following conditions:

- The relation is in 2nd NF.
- There is no transitive FDs.

i-e if A → B holds &
B → C holds then
A → C also holds.

This is Transitive FD.
This should not be present.

The advantage of removing transitive FD is:
- Amount of data duplication is reduced
- Data integrity achieved.

Ex: Book-details.

Bookid	Category-id	Category-Type	Price
11	1	Gardening	25.99
21	2	sports	14.99
31	1	Gardening	10.00
41	3	Travel	12.99
51	2	sports	17.99

In the table
Bookid → category-id
category-id → category-type
∴ Bookid → category-type. → this is transitive FD.

So this table is not in 3rd NF.

To bring the table to 3rd NF we decompose the table to

Table - book

Bookid	Categoryid	Price
1	1	25.99
2	2	14.99
3	1	10.00
4	3	12.99
5	2	17.99

Table - category.

category-id	Category-Type.
1	Gardening
2	Sports
3	Travel

Now all non-key attributes are fully functional dependent only on the primary key.
i.e In Table-Book — Categoryid & Price are only dependent on Bookid.

In Table category — Category Type is dependent on category-id.

Ex: (2) Student Details:

Student (sid, sname, dob, street, city, state, pin code),

In this table sid is primary key, but street, city and state depends upon pincode. The dependency between pin code and other fields is Transitive dependency.

- Hence to apply 3rd N F, we need to move the street, city and state to new table with pin code as primary key.

Stud 1 (sid, sname, dob, pincode)

. Stud_address (pincode , street, city, state)

Now it is in 3rd NF.

Ex: R (A B C D E)

AB → C

C → D

B → E.

Transitive FD is existing, so the relation is not in 3NF.

Find out closure of

$(AB)^+ = \{ABCDE\}$.

$C^+ = \{CD\}$

$B^+ = \{BE\}$

Candidate key is AB.

Now decompose the relation R.

R (ABCDE)

R_1 (A B C D)

AB → C

C → D

Still transitive FD is existing, so again decompose Relation R_1

R_2 (B E)

R_{11} (A B C)

R_{12} (C D)

$R_{11} \cup R_{12} \cup R_2 = R$ not lossless join. but dependency preserving decompos

Note: 3rd NF is always dependency preserving.

3.

Boyce-Codd Normal Form (BCNF)

A relation schema R is considered to be in BCNF if

- it is in 3rd NF and
- if X → Y is non trivial FD then X must be a super key.

(OR)

A relation schema R is in BCNF with respect to a set F of FDs if, for all FDs in F+ of the form X → Y, where X ⊆ R and Y ⊆ R, at least one of the following holds:

- X → Y is a trivial FD (i.e Y ⊆ X)
- X is a superkey for schema R.

Ex: customer_schema

customername	cust_street	city
Adams	spring	pittsfield
Brooks	senator	Brooklyn
Curry	North	Rye
Glenn	sandhill	woodside
Green	walnut	stamford
Johnson	Alma	PaloAlto
Jones	Park	pittsfield
Turner	Putnum	stamford
smith	North	Rye

The FD holds is

custname → cust_street, city.

∴ it is in BCNF.

(here custname is superkey)

Ex: loan-info schema.

Lno → amt, br-name

br-name	cust-name	lno	amt
Roundhill	Adams	L-11	900
Downtown	Jackson	L-14	1500
Perryridge	Hayes	L-15	1300
Perryridge	Adams	L-15	1300
Downtown	Jones	L-17	1000
Redwood	smith	L-23	2000
Hianus	curry	L-93	500
Downtown	williams	L-17	1000

The loan-info schema is not in BCNF

The loan-info schema).

lno is not a superkey for loan-info-schema, since we have a pair of tuples representing a single loan made to two people. For ex:

(Downtown, Jones, L-17, 1000)

(Downtown, Williams, L-17, 1000)

here loan-num is not a superkey, & the FD

lno → amt, br-name is nontrivial

Therefore loan-info schema doesnot satisfy the definition of BCNF.

Decompose the schema to

loan-schema (lno, br-name, amt)

Borrower (cust_name, lno)

This decomposition is a loss-less join decomposition.

The FD lno → amt, brname applies to loan schema.

lno - superkey in loan relation.

Thus both schemas are in BCNF

Ex: Suppose we have a relation schema R(A, B, C) with FDs A → B
B → C

From this set we can derive another FD
A → C (Transitivity rule)
If we used the dependency A → B to decompose R we would end up with two relations R_1(A B) R_2 (A C)
FD B → C is not preserved.
Instead if we used the FD B → C to decompose R we would end up with two relations.
R_1 (A B) R_2 (B C) which are in BCNF
& decomposition is also dependency preserving.

BCNF decomposition Algorithm

1. Suppose R is not in BCNF
Let X ⊂ R
A be a single attribute in R and
X → A be an FD that causes a violation of BCNF
Decompose R into R − A and XA

2. If either R − A or XA is not in BCNF, decompose them further by a recursive application of this algo.

Ex: R(A B C D E F G H)

F = $ABH \to C$
$A \to DE$
$BGH \to F$
$F \to AH$
$BH \to G$

sol

step1: Find FD that violates BCNF

$(ABH)^+$ = ABHDEGFC

$(BGH)^+$ = BGHFADEC

$(F)^+$ = FAHDE

$(BH)^+$ = BHGFADEC

A^+ = ADE

$(BH)^+$ is a key

$A \to DE$ violates BCNF since 'A' is not a superkey

step2: split R into

R1 (A D E)   {$A \to DE$}

R2 (A B C F G H)

{$ABH \to C$, $BGH \to F$, $BH \to G$, $F \to AH$}

In R_2 relation, it violates BCNF {∵ we have transitivity}

i.e $BGH \to F$
$F \to AH$
not in 3NF.

$ABH \to C$
$BGH \to F$
since BH is a key of R_2 they are in BCNF.

So decompose R_2 (ABCFGH)

R_{21} (BCFG) R_{22} (FAH)

Both R_{21} and R_{22} are lossless join and they are in BCNF

But R_{21} and R_{22} are not dependency preserving :

(As FDs $ABH \rightarrow C$, $BGH \rightarrow F$, $BH \rightarrow G$ are not in R_{21} or R_{22} and can't be derived from $R_1 \cup R_{21} \cup R_{22}$

Note : BCNF always lossless but not dependency preserving always.

Ex: $R(A, B, C, D)$

FDs – $A \rightarrow BCD$
$BC \rightarrow AD$
$D \rightarrow B$.

It is in 3rd NF.

Keys are;

$A^+ = ABCD$.
$BC^+ = BCAD$.
$D^+ = DB$

∴ Keys are A^*, BC

$A \rightarrow BCD$ – BCNF.
$BC \rightarrow AD$ – BCNF.

$D \rightarrow B \rightarrow$ not in BCNF since D is not a key.

Hence break a relation R into R_1 & R_2

~~R(A, B, C, D)~~
~~R_1 (A B C)~~ ≠ ~~R_2 (D, B)~~

R(ABCD)
R_1 (AD C) R_2 (DB) ✓

Ex:

R (A B C D)

AB → CD.

C → B.

Find out CK.

$AB^+ = ABCD$.

$C^+ = CB$.

So CK is AB.

AB → CD ✓ is in BCNF.

C → B ✗ not in BCNF

∵ C is not a super key.

So break relation into

R (ABCD)

R1 (A C D)

R2 (C B).

C → B.

is a superkey.

∴ it is in BCNF.

This is not dependency preserving but loss less join decomposition.

what is the highest NF satisfied for the below relation.

R(ABCD)

AB → C
C → D.

Sol:

$AB^+ = ABCD$.

$C^+ = CD$.

So AB is a superkey.

AB → C ✓ in BCNF.

C → D × not in BCNF.

As transitive FD existing it is not in 3NF also.

Now check whether it is in 2nd NF.

C → D (As 'C' is not a proper subset of AB
C → D doesnot have partial dependency).

So C → D is in 2nd NF.

∴ The relation is in 2nd NF.

Scanned with CamScanner

Multivalued Dependencies:

Although BCNF removes anamolies due to FD another type of dependency called MVD can also cause data redundancy.

MVD is a consequence of 1^{st} NF which this allows an attribute in a tuple to have a set of values.

Whenever two independent 1:N relationships between A:B and A:C are mixed in the same relation a multivalued dependency may arise.

In a Relation 'R' one attribute value is multidependent on other attributes then the MVD exists.

Definition of MVD: (MVD is determined by $\rightarrow\rightarrow$)

Let R be a relation schema and let X Y be subsets of the attributes of R.

The MVD $X \rightarrow\rightarrow Y$ holds over R the following must be true for every legal instance r of R.

i.e

$$t_1(X) = t_2(X) = t_3(X) = t_4(X)$$

$$t_1(Y) = t_3(Y) \text{ and } t_2(Y) = t_4(Y)$$

$$t_1(Z) = t_4(Z) \text{ and } t_2(Z) = t_3(Z)$$

Ex:

X	Y	Z	
a	b_1	c_1	t_1
a	b_2	c_2	t_2
a	b_1	c_2	t_3
a	b_2	c_1	t_4

- Now with the definition we will illustrate above example.

$t_1(X) = t_2(X) = t_3(X) = t_4(X)$ — 1st rule satisfied.
$a \quad a \quad a \quad a$

$t_1(Y) = t_3(Y)$ & $t_2(Y) = t_4(Y)$ — 2nd rule satisfied.
$b_1 = b_1$ $\quad b_2 = b_2$

$t_1(Z) = t_4(Z)$ and $t_2(Z) = t_3(Z)$ — 3rd rule satisfied.
$c_1 = c_1$ $\quad c_2 = c_2$

$\therefore X \rightarrow\rightarrow Y$. is true.

Ex: (2). R (A B C).

A	B	C		
a_1	b_2	c_3	t_1	t_2
a_1	b_3	c_2	t_2	t_1
a_1	b_2	c_2	t_3	t_3
a_3	b_2	c_1		
a_3	b_2	c_3		

Find out whether multivalued dependency exists for

(a) $A \rightarrow\rightarrow C$ (b) $C \rightarrow\rightarrow B$

a) $t_1(A) = t_2(A) = t_3(A)$ ✓
$a_1 \quad a_1 \quad a_1$

$t_1(C) = t_3(C)$ ✗
$c_3 \quad c_2$

$t_2(B) = t_3(B)$
$b_2 = b_2$

Now exchange t_1 t_2 tuples & check

$t_1(C) = t_3(C)$. satisfied.
$c_2 \quad c_2$

but we have only two tuples with a_3. (minimum 3 tuples should be there.

Ex: (3)

~~Did~~	~~Dname~~	Course	Teacher
~~H~~	~~cse~~		

Ex: (3).

ename	pname	dep-name
smith	X	John
smith	Y	Anna
smith	X	Anna
smith	Y	John

here in the above relation the 3 rules are satisfied.

∴ MVDs are

ename →→ Pname

ename →→ dep-name.

Ex: (4).

Dept	Job	part
d1	J1	P1
d1	J1	P2
d1	J2	P1
d1	J2	P2
d2	J3	P2
d2	J3	P4
d2	J4	P2
d2	J4	P4
d2	J5	P2
d2	J5	P4
d3	J2	P5
d3	J2	P6

The MVDs are

(a) dept →→ Job

(b) dept →→ part.

(c) both (a) & (b)

(d) none of these.

Ans - (d).

Ex: (5)

B	C	A	D	
b	c_1	a_1	d_1	t_1
b	c_2	a_2	d_2	t_2
b	c_1	a_2	d_2	t_3

B →→ C exists or not.

True.

Ex: ④

Course	teacher	book.
C	Alex	Yashwanth Kanetkar
C	Alex	Balaguruswamy
C	John	Fourozan
C	John	Balaguruswamy.
C++	Alex	Yashwanth Kanetkar
C++	Alex	Hanasgesh
C++	Alex	~~Hanasgesh~~ Balaguruswamy.

Find out MVD exists or not in the above relation.

---

note: ① MVD $X \rightarrow\rightarrow Y$ is trivial if (a) $Y \subseteq X$ and
(b) $X \cup Y = R$.

Nontrivial if Y is not a subset of X, and X and Y are not, together all the attributes.

Remember that every FD $X \rightarrow Y$ is also a MVD $X \rightarrow\rightarrow Y$

② FDs some times referred as equality generating dependencies

MVDs some times referred as tuple generating dependencies.

Fourth Normal Form (4 NF):

A relation R is said to be in 4^{th} NF if for every MVD $X \rightarrow\rightarrow Y$ that holds over R one of the following is true.

- $X \rightarrow\rightarrow Y$ is a trivial MVD holds. (if $Y \subseteq X \subseteq R$ or $XY = R$). $(R_1 \cup R_2 = R)$
- X is a super key.

(or) A table is in 4NF, if it is in BCNF & it doesn't contain MVD.

Ex: (1)

ename	Pname	dname (dependentname)
smith	X	John
smith	Y	Anna
smith	X	Anna
smith	Y	John

MVDs existing are

ename $\rightarrow\rightarrow$ Pname

ename $\rightarrow\rightarrow$ dname

According to def either we should have trivial MVD or ename is superkey.

But both conditions are not satisfied.

so to convert to 4^{th} NF decompose the relation to

R_1

ename	Pname
smith	X
smith	Y

R_2

ename	dname
smith	John
smith	Anna

Now it is in 4NF.

($X \cup Y = R$ is satisfied)

Ex: (2)

C Course	T teacher	B book
Physics	Green	Mechanics
physics	Green	optics
physics	Brown	Mechanics
physics	Brown	optics
Math	Green	Mechanics
Math	Green	Vectors
Math	Green	~~Meba~~ Geometry

The relation is not in 4^{th} NF since

$C \rightarrow\rightarrow T$ is non trivial MVD.

and 'C' is not a key.

Decompose the relation (C T B) into

R_1 (C T) and R_2(CB)

Now the relations R_1 & R_2 are in 4NF.

Join Dependencies:

(3.22)

A join dependency is a further generalization of MVDs.

A join dependency JD ⋈ {R_1, ..., R_n} is said to hold over a relation R if R_1, ..., R_n is a loss-less-join decomposition of R.

Ex:

R.

supplier	Parts	Project
S1	P1	r1
S1	P2	r2
S2	P1	r1
S2	P1	r2

R_1

supplier	Parts
S1	P1
S1	P2
S2	P1

R_2

supplier	Project
S1	r1
S1	r2
S2	r1
S2	r2

R_3.

Parts	Project
P_1	r_1
P_1	r_2
P_2	r_2

R_1 ⋈ R_3.

supplier	Parts	Project	
S1	P_1	r_1	✓
S1	P_1	r_2	extra tuple.
S1	P_2	r_2	✓
S2	P_1	r_1	✓
S2	P_1	r_2	✓

The extra tuple which are obtained (i.e not present in original relation) is called <u>additive join dependency</u>.

Scanned with CamScanner

After joining if we get the original relation, it is non-additive join dependency.

5th Normal Form: also called as Projection join Normal Form

A relation R is said to be in 5th NF if for every JD ⋈ (R₁, ..., Rn) that holds over R, one of the following is true:

- $R_i = R$ for some i (or) (trivial join depen)
- The JD is implied by the set of those FDs over R in which left side is a key for R.

- 5th NF is satisfied when all relations are broken into as many relations as possible. (In order to avoid redundancy).

- Once it is in the 5th NF it cannot be broken into smaller relations without changing the facts.

Ex:

R.

Buyer	Vendor	Item
Smith	X	Pen
Mary	X	Pen
Smith	Y	Pencil
Mary	Y	Pencil
Smith	Y	eraser

R_1

Buyer	Vendor
Smith	X
Mary	X
Smith	Y
Mary	Y

R_2

Vendor	Item
X	Pen
Y	Pencil
Y	eraser

R_3

Buyer	Item
Smith	Pen
Mary	Pen
Smith	Pencil
Mary	Pencil
Smith	eraser

$R_1 \bowtie R_2$

Buyer	Vendor	Item
Smith	X	Pen
Mary	X	Pen
Smith	Y	Pencil
Smith	Y	eraser
Mary	Y	Pencil
Mary	Y	eraser

additive join dependency.

$R_1 \bowtie R_2$ is additive join dependency.

hence it is not in 5^{th} NF.

The original relation R is in ~~5^{th} NF~~, no need to decompose it.

Note: if a relation is in 3^{rd} NF and each of its keys consists of a single attribute, it is also in 5^{th} NF.

5^{th} NF is mainly used from theoritical point of view and not ~~for~~ practical DB design.

Ex: 2

Agent	Company	Product
A1	PQR	Nut
A1	PQR	Bolt
A1	XYZ	Nut
A1	XYZ	Bolt
A2	PQR	Nut

R1

Agent	Company
A1	PQR
A1	XYZ
A2	PQR

R2

Agent	Product
A1	Nut
A1	Bolt
A2	Nut

R3

Company	Product
PQR	Nut
PQR	Bolt
XYZ	Nut
XYZ	Bolt

$R_1 \bowtie R_3$. $\quad R_{13} \bowtie R_2$

Ex:

R

Pname	Skill	Job
Aman	DBA	J1
Mohan	Tester	J2
Rohan	Programmer	J3
Sohan	Analyst	J1

R_1

Pname	Skill
Aman	DBA
Mohan	Tester
Rohan	Programmer
Sohan	Analyst

R_2

Pname	Job
Aman	J1
Mohan	J2
Rohan	J3
Sohan	J1

R_3

Skill	Job
DBA	J1
Tester	J2
Programmer	J3
Analyst	J1

$R_1 \bowtie R_2 \bowtie R_3$

$R_1 \bowtie R_2$

Pname	Skill	Job
Aman	DBA	J1
Mohan	Tester	J2
Rohan	Programmer	J3
Sohan	Analyst	J1

$R_{12} \bowtie R_3$

Pname	Skill	Job
Aman	DBA	J1
Mohan	Tester	J2
Rohan	Programmer	J3
Sohan	Analyst	J1

∴ The relation R is not in 5NF, after decomposing into R_1 R_2 R_3 it is in 5NF.

$R_1 \bowtie R_2 \bowtie R_3 = R$.

Transaction:

- A transaction is a collection of operations that form a single logical unit of work.

- A transaction is an exeution of an user program and seen by DBMS as a series of actions.
- A transaction is written between begin transaction & end transaction.

Transaction operations: read (X), write(X)

- Read (X): It performs reading of data items from the db, to logical buffer (or) program workarea.
- Write (x): Performs writing of data items to the db from local buffer.

Ex: Write a transaction to transfer an amt of Rs. 50/- from A to B.

A —50→ B

Read (A)

A: = A - 50

write (A)

Read (B);

B: = B+50

write (B)

(To read db obj it is first brought into disk from main memory & copied in prog work area in db. This is done by read operation)

(To write a db obj to memory, copy of the obj is 1st modified and then written to disk. This is done by write operation)

Properties of Transaction:

There are 4 properties of transactions to maintain concurrent access of data and recovery from system failures in DBMS. These properties are called ACID properties.

A – Atomicity
C – Consistency
I – Isolation
D – Durability

Atomicity: Either all operations of the transactions are properly selected in db or none are.

Transactions are incomplete due to 3 reasons:

(i) If a transaction is aborted or terminated for internal reasons (ex: If it is automatically restarted, then it gives incomplete transaction)

(ii) System may crash due to system failure

(iii) Read a value which is not in disk.

Transaction Management Component takes care of Atomicity.

Scanned with CamScanner

Consistency:

This property states that after the transaction is finished, its db 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 db was in a consistent state before the execution of transaction, it must remain in consistent state after the execution of the transaction.

No separate module takes care of consistency. Only sys progs takes care of consistency.

(If Atomicity, Isolation, Durability holds good consistency also holds good).

Isolation:

In a db sys 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 & executed as if it is the only transaction in the sys.

Ex: User A withdraw \$100 and user B withdraw \$250 from user Z acct, one of the users is required to wait until the other user transaction is completed, avoiding inconsistent data. If B is required to wait, then B must wait until A's transaction is completed, & Z's acct balance changes to \$900. Now B can withdraw \$250 from this \$900 balance.

(Even if multiple transactions are executing together none has to be affected by other, then it is logical Isolation)

Concurrency control Module will take care of logical Isolation.

Durability:

This property states that in any case all updates made on the db will persist even if the sys fails and restarts.

If a transaction writes or updates some data in db & commits, that data will always be there in db.

If the transaction commits but data is not written on the disk and the sys fails, that data will be updated once the sys comes up.

States of Transactions:

A transaction in a db can be in one of the following states:

- Active state: It is the initial state of transaction. The transaction will be in this state while it starts execution.

- Partially committed state: This state occurs after the final state of the transaction has been executed. (After execution of all operations, the db sys performs some checks. eg: the consistency state of db after applying o/p of trans on to db.

- Failed state: If a normal execution can no longer proceed it is said to be failed state.

- Aborted: If any checks fails & transaction reached in failed state, the recovery manager rollsback all its operation on the db to make db in the state where it was prior to start of execution of transaction. Transaction in this state are called aborted.

Scanned with CamScanner

Committed state:

If the transaction executes all its operations successfully it is said to be committed. All its effects are now permanently made on db system.

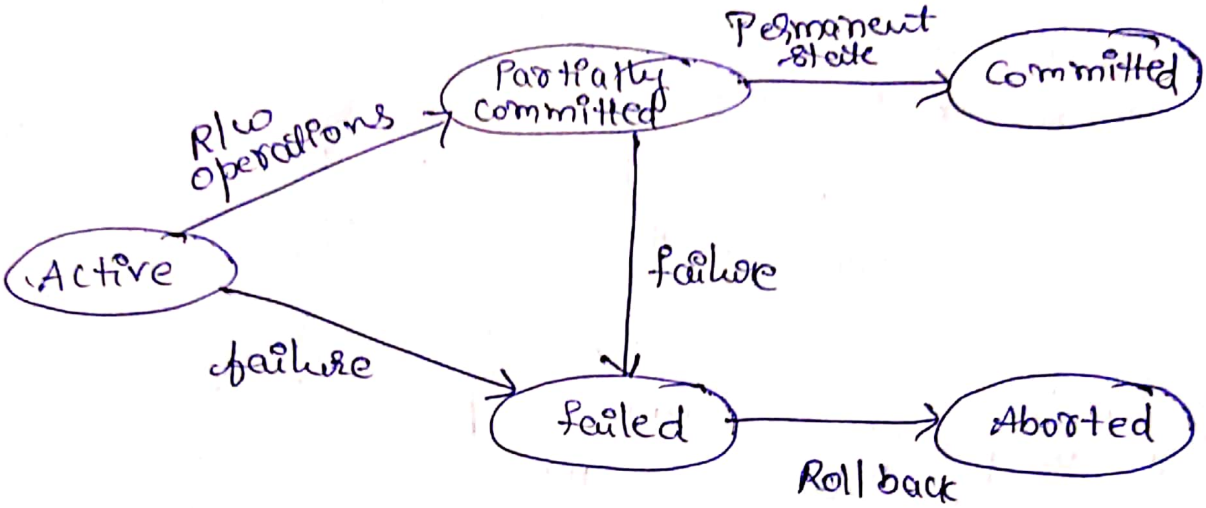


fig: transaction state diagram.

A transaction enters the failed state after the sys determines that the transaction can no longer proceed with its normal execution. Such a transaction must be rolled back. Then it enters the aborted state.

At this point the sys has two options:

(a) Restart the transaction:

If the transaction was aborted due to some H/w failure (or) S/w errors we can restart the transaction. A restarted transaction is considered as new transaction.

(b) Kill transaction:

If some logical errors occurs & if they can be corrected only by rewriting the transaction then it is called kill the transaction.

Implementation of Atomicity & Durability:

The recovery-mgmt component of a db sys implements the support for atomicity & durability.

- The shadow copy scheme which is based on making copies of the db called shadow copies, assumes that only one transaction is active at a time.
- A pointer called db-pointer is maintained on disk, it points to the current copy of the db.
- All updates are made on a shadow copy of the db and db-pointer is made to point to the updated shadow copy only after the transaction reaches commit and all updated pages have been flushed to disk.
- In case transaction fails, old consistent copy pointed to by db pointer can be used, & the shadow can be deleted.

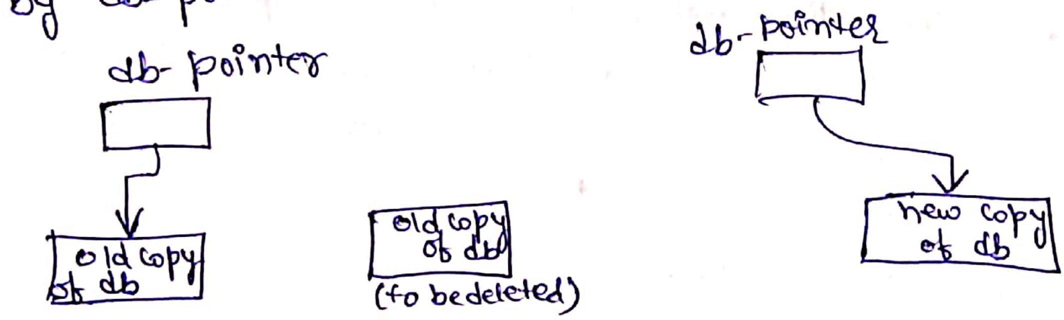


fig (a): Before update   fig (b): After update.

Thus the atomicity & durability properties of transactions are ensured by the shadow copy implementation of the Recovery-mgmt component.

Ex: text editors.

But this implementation is extremely inefficient for large dbs. Since executing a single transaction requires copying the entire db.

Storage structure:

Storage media can be distinguished by their relative speed, capacity & resilience to failure, & classified as volatile storage or nonvolatile storage. or stable storage.

- **Volatile storage:**
  Information in volatile storage doesnot survive when system crashes.
  Ex: Main memory & Cache memory.
  - Access to volatile storage is extremely fast.

- **Non Volatile Storage:**
  Information in non volatile storage survives when system crashes.
  Ex: Secondary storage devices such as magnetic disk, online storage, magnetic tapes, tertiary storage etc.
  Non volatile storage is slower than volatile storage.

- **Stable storage:**
  Information residing in stable storage is never lost.
  To implement stable storage, we replicate the information in several non volatile storage media (usually disk) with independent failure modes.
  Updates must be done with care to ensure that a failure during an update to stable storage doesnot cause a loss of information.

(Note:)
For a transaction to be durable, its changes need to be written to stable storage.

Transaction Isolation:

(Concurrent executions)

Multiple transactions which are running at same time is called concurrent execution of transactions.

Allowing multiple transactions cause several good things and bad things.

There are two good reasons for allowing concurrency:

→ **Improved throughput & Resource Utilization:**

- A transaction involve I/O activity, CPU activity etc.
- The I/O activity can ~~d~~ be done in parallel with processing at the CPU.
- The parallelism of the CPU & the I/O sys can therefore be exploited to run multiple transactions in parallel.
- While a Read or Write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction.
- All of this increases throughput of the sys. (i.e num of transactions executed in a given amt of time).
- The processor & disk utilization also increase. (i.e the processor & disk spend less time idle).

→ **Reduced waiting time:**

There may be some ~~tim~~ transactions running on a sys, some short & some long.

- If trans ~~ts~~ are running serially and a short trans has to wait till long trans complete, this will lead to delays in running ~~concurrently~~, transactions.

- If the trans are running concurrently, they share CPU cycles & disk access among them.

- Concurrent execution reduces the avg response time.

→ **Schedule:**

A schedule is a set of transactions to be executed. They represent the chronological order in which instructions are executed in the system.

There are two types of schedules:

(i) **Complete Schedule:**

A schedule that contains either a abort (or) commit statement is called complete schedule.

(ii) **Serial Schedule:**

If transactions are executed from start to finish one by one without any interchange (or) interleave then we call the schedule as a serial schedule.

Ex: Banking system.

Let T_1 & T_2 be two transactions that transfers funds from one acct to another.

- T_1 transfers $50 from Acct A to Acct B

It is defined as

T_1 : Read (A)
A := A − 50
write (A)
Read (B)
B := B + 50
write (B)

- T_2 transfers 10% of the balance from Acct A to Acct B.

T_2 : Read (A)
Temp := A * 0.1
A := A − temp
write (A)
Read (A)
B := B + temp
write (B)

Suppose current values are Acct A = $1000, Acct B = $2000 and two transactions are executed in order T_1 followed by T_2.

T_1	T_2
Read(A)	
A:= A - 50	
write(A)	
Read(B)	
B:= B+50	
write(B)	
	Read(A)
	Temp:= A*0.1
	A:= A - temp
	write(A)
	Read(B)
	B:= B+Temp
	write(B)

fig: Schedule 1 - T_1 followed by T_2.

The final values of Acct A and Acct B after execution are \$855 and \$2145.
Thus total amt A+B is preserved after the execution of both transactions.

Now T_2 followed by T_1

T_1	T_2
	Read(A)
	temp:= A* 0.1
	A:= A - temp
	write(A)
	Read(B)
	B:= B+temp
	write(B)
Read(A)	
A:= A - 50	
write(A)	
Read(B)	
B:= B+50	
write(B)	

fig: Schedule-2 T_2 followed by T_2 (Serial schedule)

∴ These schedules are serial: Each serial schedule consists of a sequence of instructions from various transactions.

Thus for a set of n transactions, there exist n! diff valid serial schedules.

If two transactions are running concurrently, the operating sys may execute one transaction for a little while, then perform a context switch, execute the 2nd trans for some time, & then switch back to 1st trans for some time and so on. (CPU time is shared among all transactions).

In general, it is not possible to predict how many instructions of a trans will be executed before the CPU switches to another transaction.

T_1	T_2
Read(A) A := A − 50 Write(A)	
	Read(A) temp := A * 0.1 A := A − temp write(A)
Read(B) B := B + 50 write(B)	
	Read(B) B := B + temp write(B)

fig: Schedule 3 – Concurrent schedule equivalent to serial schedule-1

- After this execution takes place, we arrive at some state as schedule 1. The sum(A+B) is preserved.
- Not all concurrent executions results in a correct state.

T_1	T_2
Read(A)	
A:= A-50	
	Read(A)
	temp:= A*0.1
	A:= A-temp
	write(A)
	Read(B)
write(A)	
Read(B)	
B:= B+50	
write(B)	
	B:= B+temp
	write(B)

fig: schedule-4 a concurrent schedule.

After the execution of this schedule, we arrive at a state where the final vals are Acct A & B are $950 & $2100.

This final state is <u>inconsistent</u> state. as sum(A+B) is not preserved by execution.

<u>Note</u>: We can ensure consistency of the db under concurrent execution by making sure that any schedule that executed has same effect as a schedule that could have occurred without any concurrent execution.

Serializability: is a concurrency scheme where the concurrent transactions is equivalent to one that executes the transactions serially.

The db sys must control concurrent execution of transactions, to ensure that the db state remains consistent.

Since transactions are programs, it is computationally difficult to determine exactly what operations a transaction performs & how operations of various transactions interact. For this reason we will consider only two operations:

Read and write.

The different forms of schedule equivalence

- Conflict serializability
- View serializability

Conflict serializability

- Let us consider a schedule S in which there are consecutive instructions I_i, I_j of Transactions T_i and T_j respectively $(i \neq j)$.
- If I_i and I_j refer to different data items, then we can swap I_i & I_j without affecting the results of any instruction in the schedule.
- If I_i and I_j refer to same data, then the order of two steps may matter.
- There are 4 cases that we need to consider (or Anamolies due to Interleaved fashions)

i) (Read, Read) (I_i = Read (Q), I_j = Read (Q))

order of I_i and I_j does not matter, since same value is read by T_i & T_j

ii) (Read, Write) (I_i = Read (Q), I_j = Write(Q))

order of I_i and I_j matters.

- If I_i comes before I_j then T_i does not read the value of Q that is written by T_j in inst I_j
- If I_j comes before I_i then T_i reads the value of Q that is written by T_j.

iii) (Write, Read) (I_i = Write (Q), I_j = Read (Q))

The order of I_i & I_j matters.

iv) (Write, Write) (I_i = Write (Q) I_j = Write (Q))

The order of instructions does not effect either T_i or T_j.

(Note :- I_i and I_j conflict if they are operations by different transactions on the same data item, and atleast one of these instructions is a <u>write</u> operation.

Now we will see examples :

→ (Write Read) (W.R Conflict): Reading uncommit data.

WR conflicts is ~~sau~~ that a transaction T_2 could read a ~~data~~ database object A that has been modified by another transaction T_1 which has not yet committed. Such a read is called a dirty read

Ex: ① T_1 transfers Rs. 100 from A to B.
& T_2 increments both A & B by 6%.

suppose actions are interleaved like

(i) T_1 deducts Rs. 100/- from Acct A

(ii) T_2 reads current vals of accts A & B & ~~an~~ adds 6% interest to each.

(iii) T_1 adds Rs. 100 to acct B.

T_1	T_2
R(A)	
A:= A-100	
W(A)	
	R(A)
	A:= A+0.06
	W(A)
	R(B)
	B:= B+0.06
	W(B)
	Commit
R(B)	
B:= B+100	
W(B)	
Commit	

Preserves db consistency.

If ~~for~~ ex value of A written by T_1 read by T_2 before T_1 has completed all changes then db may be in inconsistent.

Ex: 2   A=5

	T1	T2
	R(A)	
	W(A)	
		R(A)
		commit
(failure)	Abort	
	A=5	A=6

dirty read problem exists.

Ex: 3

T1	T2
R(A)	
	R(A)
	W(A)
R(A)	
	commit
W(A)	
Commit	

dirty read problem exists

→ **Read Write Conflicts (RW):-**   Unrepeatable Reads

T_2 could change the value of an obj A that has been read by a transaction T_1 while T_1 is still in progress.

Ex:① T_1 & T_2 reads some value of A say 5.
T_1 has increased A as 6 but before incrementing T_2 occurs & decremented A to 4 which is to be 5. which is incorrect.

T1	T2
R(A)	
A=A+1	
	R(A)
	A=A-1

This situation can never arise in serial execution of T_1 & T_2.

Ex: ②   X=10

	T1	T2
10	R(X)	
		R(X) 10
15	W(X)	
		R(X) 15

→ **WW (Write Write Conflicts):-** overwriting uncommitted data (Lost update problem)

T_2 could overwrite the value of an obj A, which has already been modified by a Trans T_1 while T_1 is still in progress.

Ex: ① Suppose A & B are two employees, & their salaries must be kept equal.

T_1	T_2
A := 2000	
B := 2000	
	A := 1000
	B := 1000

T_1 followed T_2
∴ both sals are 1000

T_2 followed by T_1
∴ sals are 2000

} such a write is called a blind write

Now consider interleaving actions of T_1 & T_2.

T_1	T_2
	A := 1000
B := 2000	
	B := 1000
	Commit
A := 2000	
Commit	

The result is not identical A = 2000, B = 1000 & inconsistent state.

Ex: ②

T_1	T_2	T_3
	$R_2(X)$	
$R(X)$		
		$w_3(X)$
	$w_2(X)$	

$w_3(X)$ is a blind write as there is no read before write.

Swapping:

Let I_i and I_j be consecutive instructions of a schedule S. If I_i and I_j refer to diff data items, then we can swap I_i and I_j to produce new schedule S'. We expect S to be equivalent to S'.

ex:

	T_1	T_2
1	R(A)	
2	W(A)	
3		R(A)
4		W(A)
5	R(B)	
6	W(B)	
7		R(B)
8		W(B)

fig: schedule (S_5) before swapping.

S_5 (1, 2, 3, 4, 5, 6, 7, 8)
swap (4, 5)

	T_1	T_2
1	R(A)	
2	W(A)	
3		R(A)
5	R(B)	
4		W(A)
6	W(B)	
7		R(B)
8		W(B)

fig: schedule s' after swapping.

swap (1, 2, 3, 5, 4, 6, 7, 8)
swap (3, 5)

	T_1	T_2
1	R(A)	
2	W(A)	
5	R(B)	
3		R(A)
4		W(A)
6	W(B)	
7		R(B)
8		W(B)

S_5(1, 2, 5, 3, 4, 6, 7, 8)
swap (4, 6)

	T_1	T_2
1	R(A)	
2	W(A)	
5	R(B)	
3		R(A)
6	W(B)	
4		W(A)
7		R(B)
8		W(B)

S_5(1, 2, 5, 3, 6, 4, 7, 8)
swap (3, 6).

	T_1	T_2
1	R(A)	
2	W(A)	
5	R(B)	
6	W(B)	
3		R(A)
4		W(A)
7		R(B)
8		W(B)

∴ $S_5 = S_1$

We cannot swap if I_i & I_j have same data item.

I_i	I_j
R(A)	W(A)

Conflicting instructions.

(i) belong to diff transaction

(ii) belong to same data

ex: A = 10

I_i (T_1)	I_j (T_2)
	R(A)
15 W(A)	
15 W(A)	
	R(A) 15

(iii) at least one of them is write operation.

I_i (T_1)	I_j (T_2)
	W(A) 10
15 W(A)	
15 W(A)	
	W(A) – 10.

all 3 conditions happens then there is a conflicting existing.

Testing for Serializability:

In order to know that a particular transaction can be serialized, we can draw a precedence graph.

Precedence graph:

This is a graph of nodes & vertices, where the nodes are the transaction names & the vertices are attribute collisions.

The schedule is said to be serialized if and only if there are no cycles in the resulting diagram.

- how to draw graph:

(i) Draw a node for each transaction in the schedule.

(ii) Where Transaction T_1 writes to an attribute which transaction T_2 has read from, draw a line pointing from T_2 to T_1 $W_1(A) \to R_2(A)$

(iii) Where transaction T_1 writes to an attribute which transaction T_2 has written to, draw a line pointing from T_2 to T_1. $W_1(A) - W_2(A)$

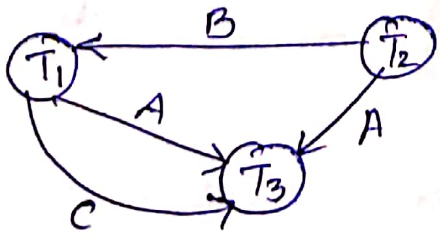
(iv) Where transaction T_1 reads from an attribute which transaction T_2 has written to, draw a line pointing from T_2 to T_1. $R_1(A) \to W_2(A)$

If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph.

ex: ①

T_1	T_2	T_3
R(A)		
R(B)		
	R(A)	
	R(B)	
		W(A)
R(C)		
W(B)		
		W(C)

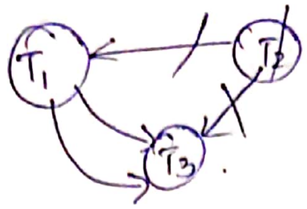
$R_1(A) \to W_2(A)$
$W_1(A) \to R_2(A)$
$W_1(A) \to W_2(A)$



No cycle so it is conflict serializable schedule. Now find serializability order by topological sorting.

- Find indegree of each node.
$T_1 - 1$, $T_2 - 0$, $T_3 - 3$

So schedule T_2

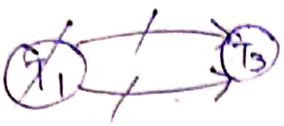


consider only T_1 & T_3.

Indegree $T_1 = 0$

$T_3 = 2$.

So schedule is $T_2 \rightarrow T_1$



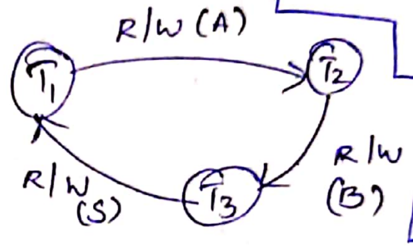
Indegree of T_3 is 0.

∴ final schedule is

$T_2 \rightarrow T_1 \rightarrow T_3$.

Ex: 2

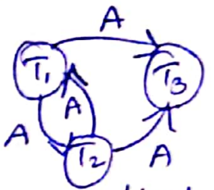
T_1	T_2	T_3
R(A)		
	W(A)	
	R(B)	
		W(B)
		R(S)
W(S)		



As there is a cycle this is not conflict serializable.

Ex: 4

T_1	T_2	T_3
R(A)		
	R(B)	
	W(A)	
W(A)		
		W(A)



Not conflict serializable

Ex: 3

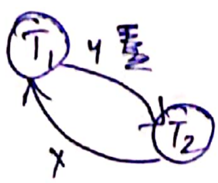
S1

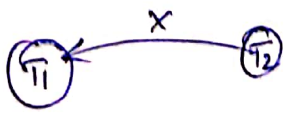
T_1	T_2
R(X)	
R(Y)	
	R(X)
	R(Y)
	W(Y)
W(X)	

S2

T_1	T_2
R(X)	
	R(X)
	W(Y)
W(Y)	
R(Y)	
W(X)	

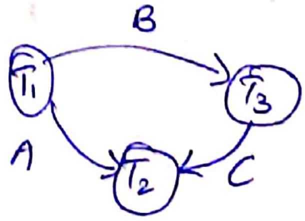
S1





Ex: 5

T_1	T_2	T_3
R(A)		
R(B)		
	R(A)	
	R(C)	
W(B)		
		R(B)
		R(C)
		W(B)
	W(A)	
	W(C)	



$T_1 = 0$ $T_2 = 2$

$T_3 = 1$

order is T_1, T_3, T_2

Conflict equivalent:

If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S & S' are conflict equivalent.

ex: Schedule 1 is conflict equivalent to schedule 3.

T_1	T_2
R(A)	
A = A - 50	
W(A)	
R(B)	
B = B + 50	
W(B)	
	R(A)
	t = A * 0.1
	A = A - t
	W(A)
	R(B)
	B = B + t
	W(B)

Schedule (1)

T_1	T_2
R(A)	
A = A - 50	
W(A)	
	R(A)
	t = A * 0.1
	A = A - t
	W(A)
R(B)	
B = B + 50	
W(B)	
	R(B)
	B = B + t
	W(B)

Schedule (3)

Conflict Serializable:

A schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

The above example schedule(3) is conflict serializable, since it is conflict equivalent to the serial schedule(1).

Ex: (2).

S_1: T_1	S_1: T_2	S_2: T_1	S_2: T_2	S_3: T_1	S_3: T_2
R(A)		W(A)		R(A)	
	W(A)	W(B)		R(B)	
R(B)			R(A)		W(A)
	W(B)		R(B)		W(B)
R - W conflict.		W - R		R - W	

∴ S_1 & S_2 are not conflict equivalent.

∴ S_1 & S_3 are conflict equivalent.

Ex: Consider schedule 7 of below fig:

T_3	T_4
R(Q)	
	W(Q)
W(Q)	

it consists of only the significant operations (i.e read & write) of transactions T_3 & T_4.
This schedule is not conflict serializable, since it is not equivalent to either the serial schedule (T_3, T_4) or the serial schedule (T_4, T_3).

→ **View Serializability:**

→ **View Equivalent:**

The schedules S & S' are said to be view equivalent if three conditions are met:

(i) For each data item Q
if T_i reads initial val of Q in schedule S then T_i in schedule S' also read initial value of Q.

(ii) (W-R conflict)
if T_i executes Read(Q) in schedule S, & if that val was produced by a write(Q) operation executed by Transaction T_j then
read(Q) oper in T_i must in S' also read the val of Q that was produced by the same write(Q) operation of Trans T_j.

(iii) The trans that performs the final write(Q) oper in S must perform the final write(Q) in S'.

A schedule S is view serializable if it is view equivalent to a serial schedule.

Ex: S_1 (Concurrent Schedule)

T_1	T_2	T_3	
		R(X)	initial read
	R(X)		
		W(X)	W-R conflict
R(X)			
W(X)			final write in T_1

⟹

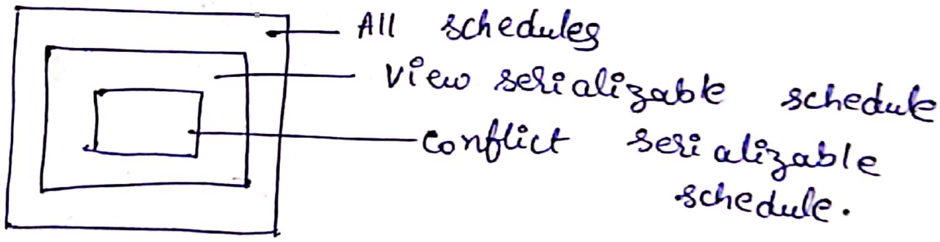
S_2 (serial schedule):

T_2	T_3	T_1	
R(X)			initial read
	R(X)		
	W(X)		W-R conflict
		R(X)	
		W(X)	final write in T_1

(i) initial read – ✓ (satisfied)
(ii) W-R conflict – ✓
(iii) Final write – ✓

all 3 conditions are satisfied so it is view serializable.

Note: Any conflict serializable schedule is view serializable. But any view serializable schedule doesnot necessarily be conflict serializable schedule.



Ex: Concurrent · Serial schedule.

S_1

T_1	T_2
R(A)	
A = A - 10	
	R(A)
	T = 2*A
	W(A)
	R(B)
W(A)	
R(B)	
B = B+10	
W(B)	

S_2

T_1	T_2
R(A)	
A = A-10	
W(A)	
R(B)	
B = B+10	
W(B)	
	R(A)

S_3

T_1	T_2
R(A)	
⋮	
W(A)	
⋮	
	R(A)
	⋮

(i) initial read – X not satisfied in S_2 & S_3. so not view serializable schedule.

Scanned with CamScanner

Ex: View Serializable.

Non-Serial S1

T_1	T_2
R(X)	
W(X)	
	R(X)
	W(X)
R(Y)	
W(Y)	
	R(Y)
	W(Y)

Serial S2

T_1	T_2
R(X)	
W(X)	
R(Y)	
W(Y)	
	R(X)
	W(X)
	R(Y)
	W(Y)

Let us check 3 conditions of View Serializability:

→ Initial Read: In schedule S1, T_1 first first Reads(X) In S_2 also T_1 first Reads (X)
Lets check Y. In S1, T_1 first reads (Y), In S_2 also T_1 first reads Y.
Both data items X & Y initial read is satisfied in S1 & S2

→ Update Read (WR): In S1 T_2 reads the value of X after it is written by T_1. In S2 T_2 reads the X after it is written by T_1.
In S_1 T_2 reads the value of Y written by T_1
In S_2 T_2 reads the value of Y " " ",
∴ This condition is also satisfied.

→ Final write : In S_1, the final write(X) is done by T_2.
In S_2 also T_2 performs final write on X.

In S_1 final write operation on Y is done by transaction T_2.
In S_2 final write on Y is done by T_2.

∴ final write is satisfied.

Since all 3 conditions are satisfied it is (S_1 & S_2) view equivalent schedule.

As it is view equivalent it is view serializable schedule.

Transaction Isolation & Atomicity:

If a transaction T_i fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction.
In a sys that allows concurrent execution, the atomicity property requires that any transaction T_j that is dependent on T_i is also aborted. To achieve this, we need to place restrictions on the type of schedules permitted in the sys.

→ **Recoverable Schedule:**

A recoverable schedule is one where, for each pair of Transaction T_i and T_j such that T_j reads data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j.

ex: ①

T_8	T_9
R(A)	
W(A)	
	R(A)
	Commit
R(B)	

T_9 is dependent on T_8
Non-recoverable schedule.
If T_9 commits before T_8.

Suppose that the system allows T_9 to commit immediately after execution of read(A) instruction. Thus T_9 commit before T_8 does.
Now suppose that T_8 fails before it commits. Since T_9 has read the value of data item A written by T_8 we must abort T_9 to ensure transaction Atomicity.

However, T_9 has already committed & cannot be aborted.
Thus we have a situation where it is impossible to recover correctly from the failure of T_8.

Ex: ②

T_1	T_2
1000 R(A)	
900 A=A-100	
900 W(A)	
	R(A) 900
	A=A+500 1400
	W(A) 1400
	~~commits~~
failure point	
Commit,!	Commit!

T_1 reads & writes A & that value is read & written by T_2. But later on T_1 fails. So we have to rollback T_1. Since T_2 read the value written by T_1 it should also be rollbacked. As it is not committed we can rollback T_2 as well. So it is recoverable schedule with cascading schedule.

→ Cascadeless schedule:

T_{10}	T_{11}	T_{12}
R(A)		
R(B)		
W(A)		
	R(A)	
	W(A)	
		R(A)
Abort		

- Transaction T_{10} writes a value of A that is read by Transaction T_{11}.
- T_{11} writes a value of A that is read by T_{12}.
- Suppose at this point T_{10} fails, T_{10} must be rolled back, since T_{11} is dependent on T_{10}, T_{11} must be rolled back, T_{12} is dependent on T_{11}, T_{12} must be rolled back.
- This phenomenon, in which a single transaction failure leads to a series of transaction rollback is called cascading rollback.
- Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work.
- It is desirable to restrict the schedules to those where cascading rollbacks cannot occur, such schedules are called cascadeless schedules.
- Formaly, A cascadeless schedule is one where for each pair of transaction T_i & T_j such that T_j reads data item, previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

(Note:) Every cascadeless schedule is also recoverable schedule.

Cascadeless Schedule-

T_{10}	T_{11}
R(A)	
W(A)	
	R(B)
Commit	
	R(A)

Transaction Isolation Levels:

The isolation levels specified by the SQL standard are as follows:

- Serializable usually ensures serializable execution.
- Repeatable read allows only committed data to be read & further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it.
- Read committed allows only committed data to be read, but doesnot require repeatable reads. For instance, between two reads of a data item by the transaction, another transaction may have updated the data item and committed.
- Read uncommitted allow uncommitted data to be read. It is the lowest isolation level allowed by SQL.

All the isolation levels above additionally disallow dirty writes, i-e they disallow writes to a data item that has already been written by another transaction that has not yet committed or Aborted.

Implementation of Isolation levels:

(i) Locking (ii) Timestamps.

Lock Based Protocols:

- A lock is a mechanism to control concurrent access to a data item.
- Data items can be locked in two modes:
  - Shared (S) mode – Data item can only be read. S-lock is requested using lock-S instruction.
  - Exclusive (X) mode – Data item can be both read as well as written. X-lock is requested using lock-X instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after requested is granted.

T_i

T_j \ T_i	S	X
S	true	false
X	false	false

fig: lock compatibility matrix.

Shared mode is compatible with shared mode but not with exclusive mode.

At any time several shared-mode locks can be held simultaneously on a particular data item.

A subsequent exclusive-mode lock req has to wait until the currently held shared-mode locks are released.

4.16

To access a data item, Transaction must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released.
Thus T_i is made to <u>wait</u> until all incompatible locks held by other transactions have been released.

Ex:

T_1 (A = 100, B = 200)

lock - X(B)
R(B)
B = B - 50
W(B)
unlock (B)
lock - X(A)
R(A)
A = A + 50
W(A)
unlock (A)

T_2

lock - S(A)
R(A)
unlock (A)
lock - S (B)
R(B)
unlock (B)
display (A + B)

schedule <T_1, T_2> Consistent
<T_2, T_1> Consistent

T_1	T_2	concurrency-control manager
lock-X(B)		grant-X(B, T_1)
200 R(B)		
150 B = B - 50		
150 W(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	R(A) 100	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	R(B) 150	
	unlock(B)	
	display(A+B) 100+150 = 250	
lock-X(A)		
		grant-X(A, T_1)
50 R(A)		
50 A = A + 50		
50 W(A)		
unlock(A)		

fig: schedule 1.

The concurrent transactions are in inconsistent state. The reason is Transaction T_1 unlocked data item B too early, as a result of which T_2 saw an inconsistent state.

serial schedule <T3, T4> consistent.

Now we will see concurrent schedule.

T_3	T_4
lock-X(B)	
R(B)	
B=B-50	
W(B)	
	lock-S(A)
	R(A);
	lock-S(B)
lock-X(A)	

fig: schedule 2

From the above fig since T_3 is holding an X-mode lock on B and T_4 is holding a S-mode lock on B, T_4 is waiting for T_3 to unlock B.

similarly T_4 is holding a S-mode lock on A & T_3 is requesting an X-mode lock on A, T_3 is waiting for T_4 to unlock A.

Thus, we have arrived at a state where neither of these transactions can never proceed with its normal execution. This situation is called deadlock. When a deadlock occurs, the sys must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to other transa, which can continue with its execution.

- If we donot use locking, or if we unlock data items as soon as possible after reading or writing them, we may get inconsistent states.
- If we donot unlock a data item before requesting a lock on another data item, deadlock may occur.

The inconsistent states may lead to real-world problems that cannot be handled by the db sys. So each transaction in sys follow a set of rules called locking protocols indicating when a transaction may lock & unlock each of data items.

Granting of Locks:

If a transaction request a lock on data item in a particular mode & no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.

eg:

starvation

T_1	T_2	T_3	T_4
	lock-S(Q)		
lock-X(Q)	⋮	lock-S(Q)	lock-S(Q)
	released		

(4') (19)

Suppose a transaction T_2 has a shared-mode lock on a data item, & another Trans T_1 requests an exclusive-mode lock on the data item. Clearly, T_1 has to wait for T_2 to release the shared-mode lock. Meanwhile, a trans T_3 may request a shared-mode lock on the same data item. The lock req is compatible with the lock granted to T_2, so T_3 may be granted the shared-mode lock. At this point T_2 may release the lock, but still T_1 has to wait for T_3 to finish. But again, there may be a new transaction T_4 that requests a shared-mode lock on the same data item, & is granted the lock before T_3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, & each trans releases the lock a short while after it is granted, but T_1 never gets the X-mode lock on the data item.

✗ The transaction T_1 may never make progress, & is said to be <u>starved</u>.

We can avoid <u>starvation</u> of transactions by granting locks in following manner:

When a Trans T_i requests a lock on a data item Q in a particular mode M, the <u>concurrency-control-mgr</u> grants the lock provided that

(i) There is no other transaction <u>holding a lock on</u> Q in a mode that <u>conflicts</u> with M.

(ii) There is no other trans that is <u>waiting</u> for a lock on Q, & that made its <u>lock request before</u> <u>T_i</u>

Thus, a lock req will never get blocked by a lock req that is made later.

Two-Phase locking Protocol:

The protocol that ensures serializability is the two-phase locking protocol. This protocol requires that each transaction issue lock and unlock requests in two phases:

(i) **Growing phase**: A transaction may obtain locks, but may not release any lock.

(ii) **Shrinking phase**: A trans may release locks, but may not obtain new locks.

Initially, a transaction is in the growing phase. The trans acquires locks as needed. Once the trans releases a lock, it enters the shrinking phase; & it can issue no more lock requests.

ex: ① T_3 :
```
lock-X(B)
R(B)
B = B-50
W(B)
lock-X(A)
R(A)
A = A+50
W(A)
unlock(B)
unlock(A)
```

T_4 :
```
lock-S(A);
R(A)
lock-S(B)
R(B)
display(A+B)
unlock(A)
unlock(B)
```

T_3 & T_4 are two phase.

ex: ② T_1 :
```
lock-X(B)
R(B)
B = B-50
W(B)
unlock(B)
lock-X(A)
R(A)
A = A+50
W(A)
unlock(A)
```

T_2 :
```
lock-S(A)
R(A)
unlock(A)
lock-S(B)
R(B)
unlock(B)
display(A+B)
```

T_1 & T_2 are not two phase.

The two phase locking protocol ensures conflict serializability.

The point in the schedule where the transaction has obtained its final lock is called the lock point of the transaction.

Now transaction can be ordered according to their lock points.

Two phase locking does not ensure freedom from deadlock.

Ex: T_3 & T_4 are Two phase.

T_1	T_2
lock - S(A)	
	lock - S(A)
lock - X(B) (lockpoint)	
unlock - (A)	
	lock - X(C) (lock point)
unlock(B)	
	unlock (A)
	unlock (C)

ex:

T_1	T_2
lock - X(B)	
R(B)	
B = B - 50	
W(B)	
	lock - S(A)
	R(A)
	lock - S(B)
	⋮
lock - X(A)	

This schedule is Two - phase but it is in deadlock state.

Scanned with CamScanner

cascading rollback may occur under two-phase locking.

Ex:

T_5	T_6	T_7
lock-X(A)		
R(A)		
lock-S(B)		
R(B)		
W(A)		
unlock(A)		
	lock-X(A)	
	R(A)	
	W(A)	
	unlock(A)	
		lock-S(A)
		R(A)

each transaction observes the two-phase locking protocol but the failure of T_5, after the R(A) step of T_7 leads to cascading rollback of T_6 & T_7.

Cascading rollbacks can be avoided by making modification of 2-phase locking called strict 2-phase locking protocol.

This protocol requires not only that locking be two-phase, but also that all Exclusive-mode locks taken by a transaction be held until that transaction commits.

Ex: Strict - 2 Phase Locking Protocol

T_1	T_2
lock - S(A)	
R(A)	
	lock - S(A)
lock - X(B)	
unlock(A)	
R(B)	
W(B)	
	R(A)
	unlock(A)
Commit	
unlock(B)	
	lock - S(B)
	R(B)
	unlock(B)
	Commit

It guarantees cascadeless recoverability.

Rigorous two phase locking protocol requires that all locks be held until the transaction commits, In this transaction can be serialized in the order in which they commit.

Ex:

T_1	T_2
lock - S(A)	
R(A)	
	lock - S(A)
lock - X(B)	
	R(A)
R(B)	
W(B)	
commit	
unlock(B)	
	lock - S(B)
	R(B)
unlock(A)	
	commit
	unlock(A) unlock(B)

lock conversions:

- It is a mechanism for upgrading a shared lock to an exclusive lock (upgrade). Upgrading can take place in growing phase.

- mechanism for downgrading a X-lock to a shared lock (downgrade). Down grading can take place in shrinking phase.

Ex: T_8 : $R(A_1)$
$R(A_2)$
⋮
$R(A_n)$
$w(A_1)$
⋮

T_9 : $R(A_1)$
$R(A_2)$
display $(A_1 + A_2)$

T_8 and T_9 run concurrently under 2-phase locking protocol.

T_8	T_9
lock-S(A_1)	
	lock-S(A_1)
lock-S(A_2)	
	lock-S(A_2)
lock-S(A_3)	
lock-S(A_4)	
	unlock (A_1)
	unlock (A_2)
⋮	
lock-S(A_n)	
upgrade (A_1)	

By locking A, in shared mode we can archieve concurrency. Thus, we can get no conflict. By completing all the locking we have to convert A_1 into X-lock. Thus we use upgrade, which converts shared to exclusive and then we can write. This is used in growing phase.

Downgrade is used in shrinking phase.

Implementation of locking: (mechanism to manage the locking requests made by transactions)

- A lock-manager can be implemented as a process that receives messages from transactions & sends messages in reply.
- The lock-mgr process replies to lock-request msgs with lock-grant msgs or roll back (abort) in case of deadlocks.
- Unlock msgs requires only an acknowledgement in response.
- The lock mgr uses this data structure: For each data item ie currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived.

It uses a hash table, indexed on the name of data item, to find the linked list for a data item. This table is called lock table

Each record of the linked list for a data item ~~this~~ notes which transaction made the request, & what lock mode is requested. The record also notes if the request has currently been granted.

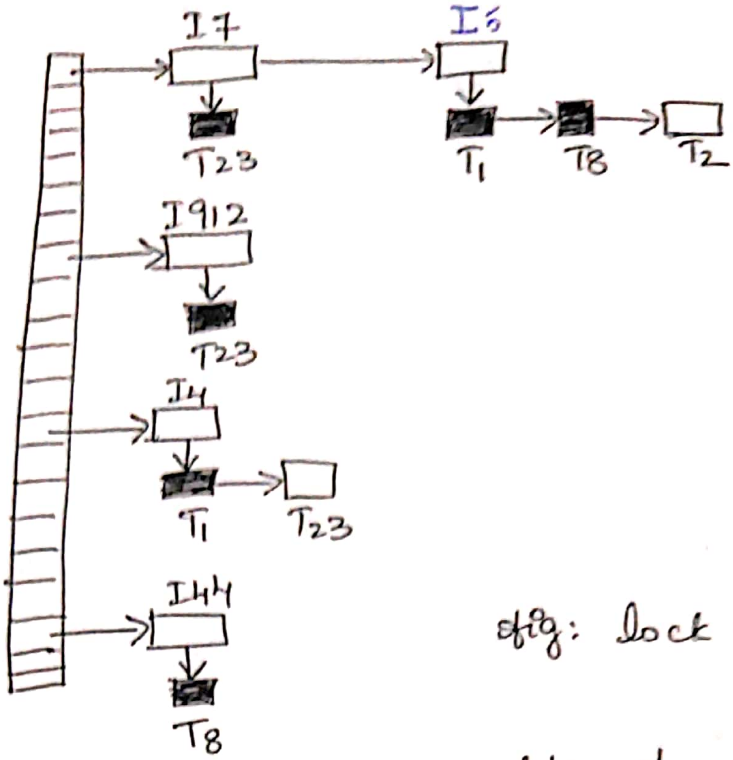


fig: lock table

The above fig shows an example of a lock table. The table contains locks for five different data items, I4, I7, I23, I44 & I912.

The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items.

Granted locks are filled-in (black) rectangles, while waiting requests are the empty rectangles.

for example, T23 has been granted locks on I912 & I7 and is waiting for a lock on I4.

New requests are added at end of queue & granted if it is compatible with earlier locks

If transaction aborts, all waiting or granted requests of the transaction are deleted;

Graph Based Protocol.

Simplest graph based protocol is Tree locking protocol

Tree locking protocol is used to employ exclusive lock & when the db is in the form of a tree of data items. Tree locking protocol is serializable.

→ Tree based protocol.

- Only exclusive locks are allowed.
- The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
- Data items can be unlocked at any time.

This protocol ensures Conflict serializability & Deadlock Free schedule.

Here we need not wait for unlocking a Data item as we did in 2-PL Protocol, thus increasing the concurrency.

~~Ex~~

The prerequisite of this protocol is that we know the order to access a Database Items.

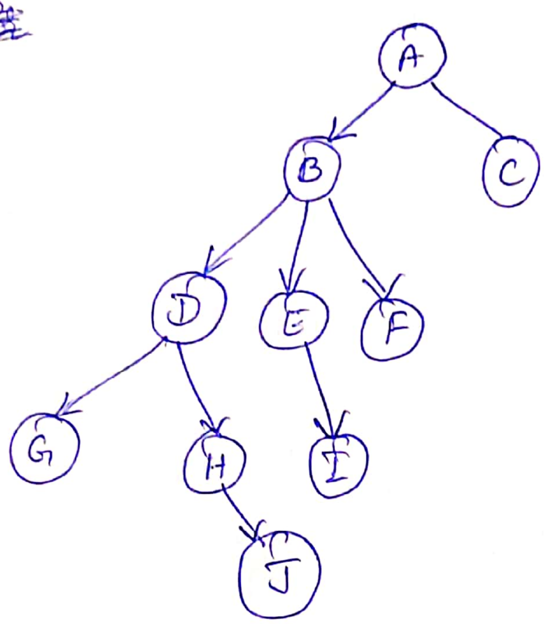
For this we implement a Partial Ordering on a set of Database items (D) $\{d_1, d_2, d_3, \ldots, d_n\}$

The protocol following the implementation of Partial ordering is stated as —

→ if $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.

- Implies that the set D may now be viewed as a directed acyclic graph (DAG) called a database graph.

Ex:



Data base graph

Based on the above Database graph we will see a example.

T_1	T_2	T_3	T_4
Lock-X(B)			
	Lock-X(D) Lock-X(H) unlock-X(D)		
Lock-X(E) Lock-X(D) unlock-X(B) unlock-X(E)			
		Lock-X(B) Lock-X(E)	
	unlock-X(H)		
Lock-X(G) unlock-X(D)			
			Lock-x(D) Lock-x(H) unlock-D unlock-H
		unlock(B) " (E)	
unlock(G)			

Ex: The following 4 transaction follow the ~~the~~ tree protocol on previous db graph.

T_1	T_2	T_3	T_4
lock-X(B)	lock-X(D)	lock-X(B)	lock-X(H)
lock-X(E)	lock-X(H)	lock-X(E)	lock-X(J)
lock-X(D)	unlock-X(D)	unlock-X(E)	unlock-X(H)
unlock-X(B)	unlock-X(H)	unlock-X(B)	unlock-X(J)
unlock-X(E)			
lock-X(G)			
unlock-X(D)			
unlock-X(G)			

Advantages:

- Shorter waiting time and increases in concurrency.
- Protocol is deadlock free, no rollback are required.

Disadvantages:

- Protocol doesnot guarantee recoverability or cascade freedom
- In tree locking protocol a transaction that needs to access data item A & J in the db graph, must lock not only A & J but also data items B, D, H. This additional locking results in increased locking overhead & the possibility of additional waiting time & potential decrease in concurrency.

Time Stamp Ordering Protocol:

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first.
  To determine the timestamp of the transaction, this protocol uses system time or logical counter.

Ex: Let's assume there are two transactions T_1 & T_2. Suppose the transaction T_1 has entered the system at 007 sec & transaction T_2 has entered at 009 sec. T_1 has higher priority, so it executes first as it is entered the sys first.

- For each data item we maintain two Timestamps:

  W-TS(X) : largest timestamp of any transaction that executed write(X) successfully.

  R-TS(X) : is the largest timestamp of any transaction that executed read(X) successfully.

These two timestamps are updated each time a successful read/write operation is performed on the data item X.

T issues a read(x) operation:

```
if (W-TS(X) > TS(T))
    Abort T & Roll back;

else
{
  Read (X)
  R-TS(X)= TS(T);

}
```

T issues a write(x) operation:

```
if (R-TS(X) > TS(Ti)) OR (W-TS(X)>TS(Ti)) then
    Abort-T & Rollback;

else
{
 write(X)
 W-TS(X) = TS(Ti);

}
```

Ex:

T_1	T_2
R(x)	
	W(x)
R(x)	

initially RTS = 0 , WTS = 0

TS of T_1 = 1

T_2 = 2

1) T_1 issues Read operation

WTS(X) > TS(T)

0 > 1 False.

else

Read(X)

R_TS(X) = 1. (i.e TS(T_1))

2) Now Write(X)

R-TS(X) > TS(T_i) or W-TS(X) > TS(T_i)

1 > 2 0 > 2

False

else Write(X)

W-TS(X) = 2

3) Read(X)

2 > 1

True

Abort & roll back.

- Time stamp protocol ensures freedom from deadlock as no transaction ever waits.
- But these schedules may not be cascade-free & may not even be recoverable.

Thoma's write Rule:

This rule states that in case of

if $TS(T_i) < W\text{-}TS(x)$

Operation rejected & T_i rolled back.

Time stamp ordering rules can be modified to make the schedule view serializable.
Instead of making T_i rolled back, the 'write' operation itself is ignored.

```
If (RTS(x) > TS(Ti))
    Abort & Rollback.
If (write-TS(x) > TS(Ti))
    Reject write but commit
    the transaction.
else
  {
    w(x)
    W_TS(x) = TS(Ti);
  }.
```

Ex:

T_1	T_2
w(x)	
	w(x)
w(x)	

This is not necessary.

T_1	T_2
w(x)	
	commit
w(x)	

4.26

Validation Based Protocols also called optimistic concurrency control

It imposes less overhead

Also based on Timestamp Protocol. It has three phases:

1. **Read Phase:** During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variable local to T_i. It performs all the write operations on temporary local variables without update of the actual database.
2. **Validation Phase:** Transaction T_i performs a validation test to determine whether it can copy to database the temporary local variables that hold the result of write operations without causing a violation of serializability.
3. **Write Phase:** If Transaction T_i succeeds in validation, then the system applies the actual updates to the database, otherwise the system rolls back T_i.

To perform the validation test, we need to know when the various phases of transaction T_i took place. We shall therefore associate three different timestamps with transaction T_i.

1. **Start (T_i):** the time when T_i, started its execution.
2. **Validation (T_i):** the time when T_i finished its read phase and started its validation phase.
3. **Finish (T_i):** the time when T_i finished its write phase.

The Validation Test for T_j requires that, for all transaction T_i with $TS(T_i) < TS(T_j)$ one of the following condition must hold

1. Finish (T_i) < Start (T_j): Since T_i completes its execution before T_j started, the serializability order is indeed maintained.
2. Start(T_j)<Finish(T_i) <validation(T_j): The validation phase of T_j should occur after T_i finishes.

Few Points

1. Resolves the cascade rollbacks
2. Suffers from the starvation.
3. Optimistic concurrency control.

Ex: Schedule Produced by Validation:

T_{14}	T_{15}
R(B)	
	R(B)
	B = B − 50
	R(A)
	A = A + 50
R(A)	
(validate)	
display (A+B)	
	(validate)
	W(B)
	W(A)

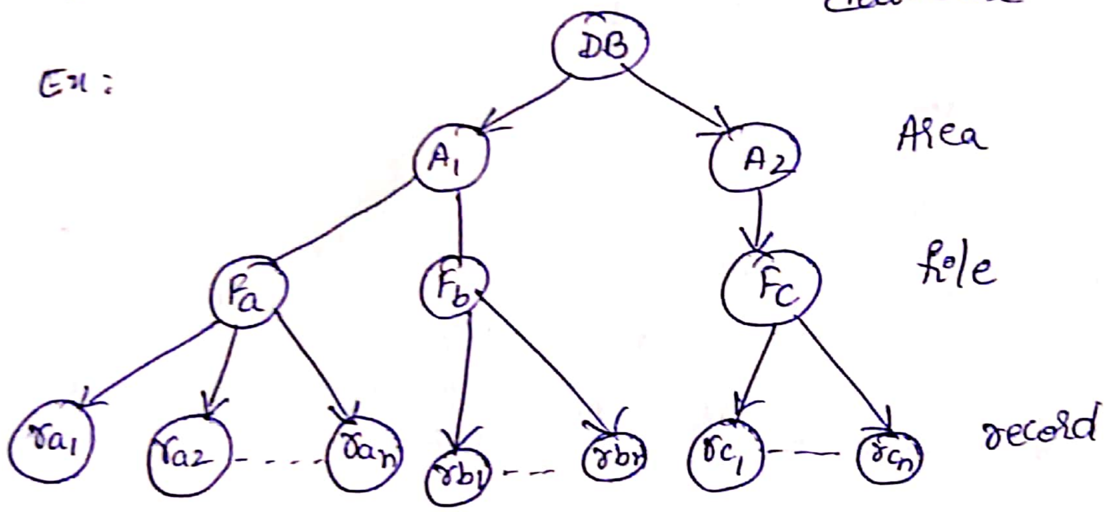
Suppose that $TS(T_{14}) < TS(T_{15})$ then the validation phase suceeds in the above schedule. Note that the writes to the actual variates are performed only after the validation phase of T_{15}. Thus T_{14} reads the old values of B & A & this schedule is serializable.

Multiple Granularity:

Granularity means diff level of data.

- Multiple granularity locks allow us to lock at different granularities (db, tables, pages, tuples).
- This is useful because we can choose diff granularities for diff transactions.

Ex:



Consider the above tree which consists of 4 levels of nodes.

- Highest level represents entire db.
- below it area
- area in turn has file as child (No file & more than in one area).
- file consists of records. (no record can be present in more than one file).

When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in same lock mode.

Consider two cases

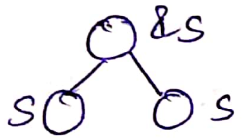
Case-1: If we want to lock ra_2, we have to check from root to node, is there any anscester locked already.

Case-2: for example ra_1 is locked.
Now if we want to lock DB we can't as ra_1 is already locked.

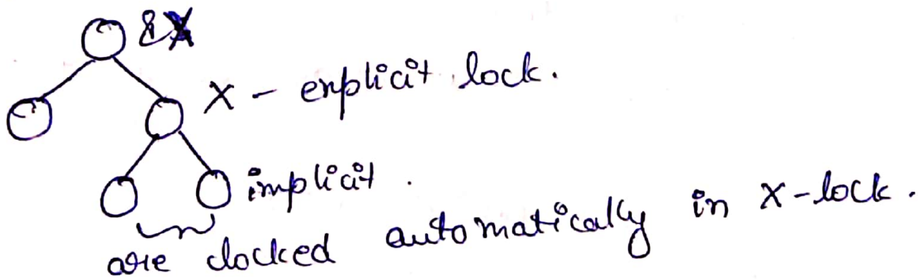
To address the above problems we introduce Intention lock modes:

In addition to S & X lock modes, there are three additional lock modes with multiple granularity:

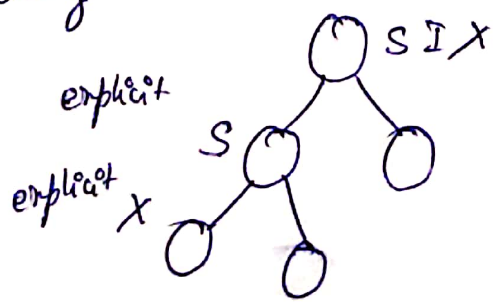
- Intension - shared (IS): indicates explicit locking at a lower level of the tree but only with shared locks



- Intention - exclusive (IX): indicates explicit locking at a lower level with exclusive or shared locks.



- shared & Intention - exclusive (SIX): the subtree rooted by that node is locked explicitly in shared mode & explicit locking is being done at a lower level with exclusive-mode locks.



Compatibility Matrix:

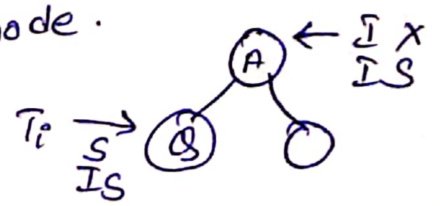
Ti \ Tj	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

Parent locked in	child can be locked in.
IS	IS, S
IX	IS, S, IX, X, SIX
S	{S, IS} not necessary
SIX	X, IX, {SIX}
X	none

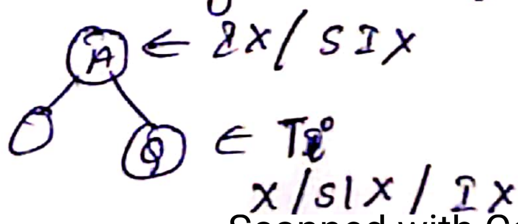
rules:

Transaction Ti can lock a node Q using following rules:

- lock compatibility matrix must be observed.
- Root of the tree must be locked first, & may be locked in any mode.
- A node Q can be locked by Ti in S or IS mode only if the parent of Q is currently locked by Ti in either IX or IS mode.

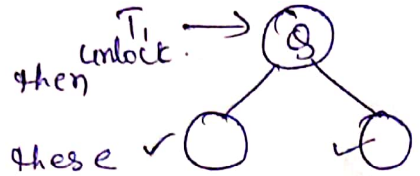


- A node Q can be locked by Ti in X, SIX, or IX mode only if the parent of Q is currently locked by Ti in either IX or SIX mode.



— T_i can lock a node only if it has not previously unlocked any ~~mode~~ node so far.

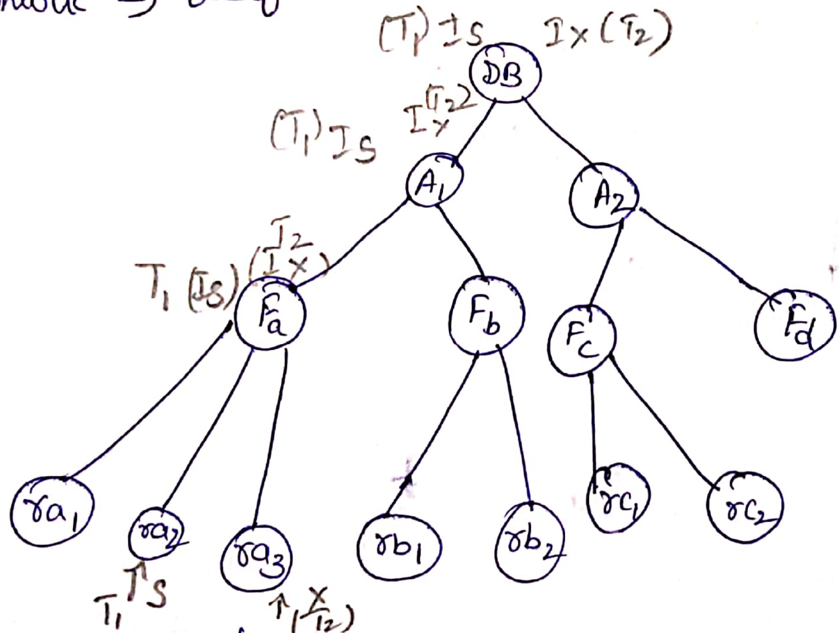
— T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.



these two nodes must be unlocked by T_1.
then only Q be unlocked.

Lock → root-to-leaf order
unlock → leaf-to-root order.

Ex:



→ (a) T_1 reads ra_2
to apply share lock to ra_2 traverse from root & apply IS to DB, A, Fa.

→ (a) T_2 modifies ra_2 (writing the data).
This will not be obtained as ra_2 is locked in shared mode.

T_2 modifies ra_3.
ra_3 must obtain x lock at upper level.

→ T_3 reads all record on Fa shared mode will not possible to make to compatible with SX. as it is not compatible with SX.
As shared lock can't be granted, T_3 doesnot execute.

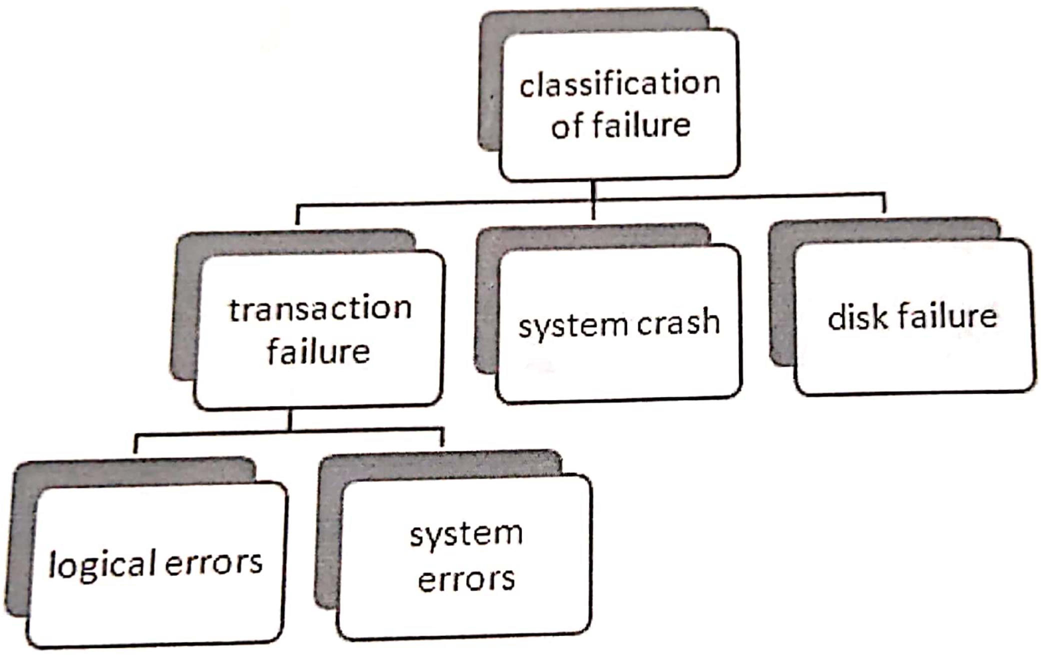
→ T_4 reads entire DB i.e DB in shared mode not possible as it is locked by T_2 in IX mode.

Recovery System

Database systems, like any other computer system, are subject to failures but the data stored in it must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transactions are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database.

Failure Classification:

We generalize a failure into various categories, as follows –



1. Transaction failure

A transaction has to abort when it fails to execute or when it reaches a point from where it can't go any further.

Reasons for a transaction failure could be –

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

2. System Crash

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

3. Disk Failure

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

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

Storage Structure

We have already described the storage system. In brief, the storage structure can be divided into two categories –

- **Volatile storage** – As the name suggests, a volatile storage cannot survive system crashes. Volatile storage devices are placed very close to the CPU; normally they are embedded onto the chipset itself. For example, main memory and cache memory are examples of volatile storage. They are fast but can store only a small amount of information.
- **Non-volatile 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, and non-volatile (battery backed up) RAM.

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*B*) transfers the physical block *B* to main memory.

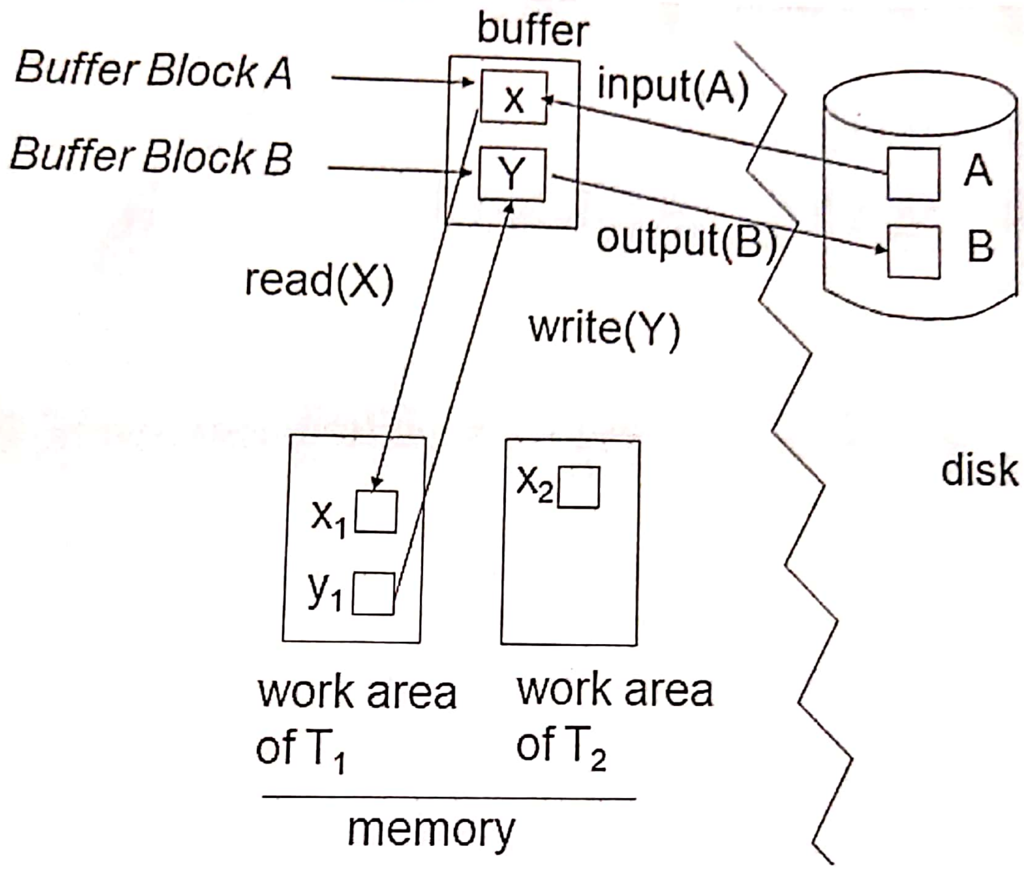
- **output**(B) transfers the buffer block B to the disk, and replaces the appropriate physical block there.

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.

  T_i's local copy of a data item X is called x_i.

- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.
- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  - **read**(X) assigns the value of data item X to the local variable x_i.
  - **write**(X) assigns the value of local variable x_i to data item {X} in the buffer block.
  - both these commands may necessitate the issue of an **input**(B_X) instruction before the assignment, if the block B_X in which X resides is not already in memory.
- Transactions
  - Perform **read**(X) while accessing X for the first time;
  - All subsequent accesses are to the local copy.
  - After last access, transaction executes **write**(X).
- **output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.

Example of Data Access



Recovery and Atomicity

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

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

- It should check the states of all the transactions, which were being executed.

- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

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

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

Log-based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction.

- When transaction T_i starts, it **registers itself** by writing a <T_i **start**>log record
- *Before* T_i executes **write**(X), a log record

  <T_i, X, V_1, V_2> is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.

  Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.

- When ***T_i* finishes it last statement, the log record**

  <T_i , **commi**t> is written.

- **< Ti ,abort> Transaction Ti has aborted.**

We assume for now that log records are written directly to stable storage (that is, they are not buffered)

Two approaches using logs

- **Deferred Modification Technique:** All logs are written on to the stable storage and the **database is updated when a transaction commits.**

Assume that transactions execute serially

- **Transaction starts by writing <T_i *start*> record to log.**
- **A write(X) operation results in a log record <T_i, X, V> being written, where V is the new value for X**

Note: old value is not needed for this scheme

- **The write is not performed on X at this time, but is deferred(postponed).**
- **When T_i partially commits, <T_i commit> is written to the log**
- **Finally, the log records are read and used to actually execute the previously deferred writes.**
- **During recovery after a crash, a transaction needs to be redone if and only if both <T_i start> and<T_i commit> are there in the log.**
- **Redoing a transaction T_i (redoT_i) sets the value of all data items updated by the transaction to the new values.**
- **Crashes can occur while the transaction**
    - **is executing the original updates, or**
    - **while recovery action is being taken**

example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: read (A)

$A := A - 50$

Write (A)

read (B)

$B := B + 50$

write (B)

T_1 : read (C)

$C := C - 100$

write (C)

- Below we show the log as it appears at three instances of time.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>
(a)	(b)	(c)

- If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

(b) redo(T_0) must be performed since <T_0 commit> is present

(c) redo(T_0) must be performed followed by redo(T_1) since

<T_0 commit> and <T_i commit> are present

- **Immediate Modification Technique:** the database is modified immediately after every operation.

  <T_i start>

  <T_i, X, V1,V2>

  <T_i commit>

➢ Update log record must be written *before* database item is written

- We assume that the log record is output directly to stable storage
- Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage

➢ Output of updated blocks can take place at any time before or after transaction commit

➢ Order in which blocks are output can be different from the order in which they are written.

Log	Write	Output
<T_0 start>		
<T_0, A, 1000, 950>		
T_0, B, 2000, 2050		
	A = 950	
	B = 2050	
<T_0 commit>		
<T_1 start>		
<T_1, C, 700, 600>		
	C = 600	
		B_B, B_C
<T_1 commit>		
		B_A

Note: B_X denotes block containing X.

Recovery procedure has two operations instead of one:

- **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
- **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i

When recovering after failure:

- transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
- Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.

Undo operations are performed first, then redo operations.

Below we show the log as it appears at three instances of time.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Check Points

When a system crash occurs we must consult the log to determine those transactions that need to be redone & those that need to be undone. We need to search entire log to determine this information. There are two difficulties with this approach:

- Searching entire log is time-consuming
- We might unnecessarily redo transactions which have already o/p their updates to the database.

To reduce these problems we introduce checkpoints.

A checkpoint is performed as follows:

- o/p all log records residing in main memory to stable storage.
- o/p all modified buffer blocks to the disk.
- Write a log record <checkpoint> onto stable storage.

During recovery we have to consider only the most recent transaction T_i that started before the checkpoint, & the transactions that started after T_i.

(i) Scan backwards from end of log to find the most recent <checkpoint> record.

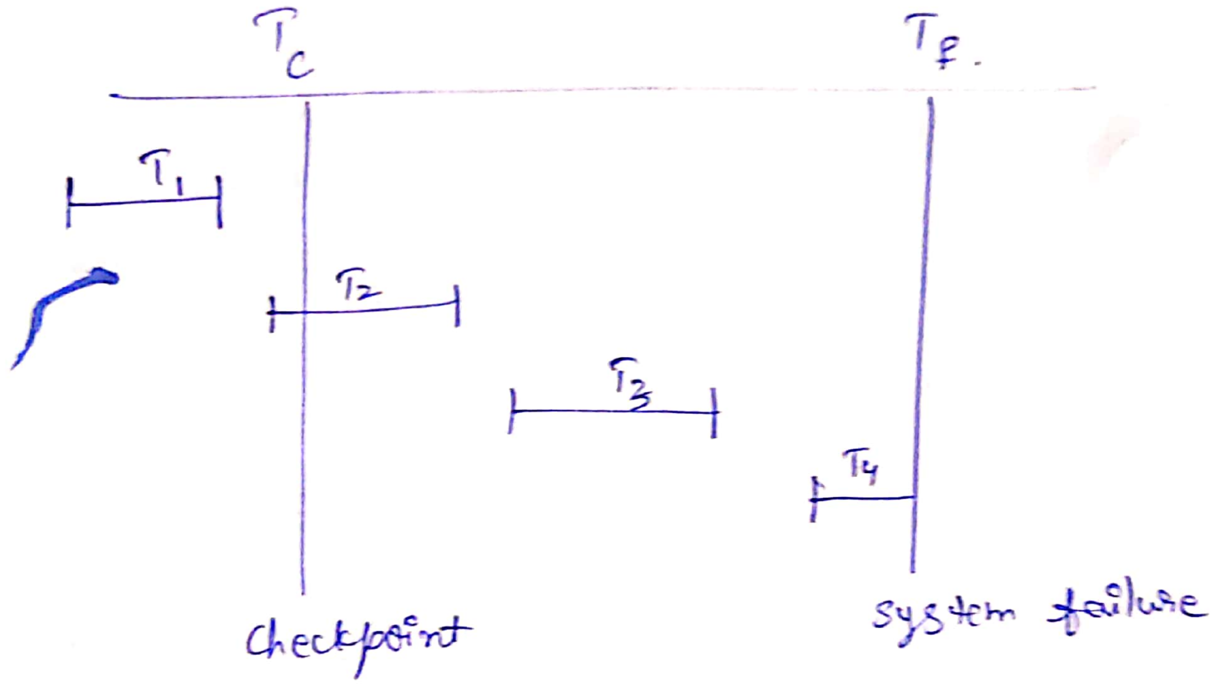
(ii) Continue scanning backwards till a record <T_i start> is found.

(iii) Need only consider the part of log following above start record. Earlier part of log can be ignored during recovery, & can be erased whenever necessary.

(iv) For all transactions (starting from T_i or later) with no <T_i commit> execute undo(T_i).

(v) Scanning forward in the log, for all transactions starting from T_i or later with a <T_i commit> execute redo(T_i).

Ex:



- T_1 can be ignored (updates already o/p to disk due to checkpoint.
- T_2 & T_3 redone
- T_4 undone.

Ex: if we have T_1 to T_{100} &

<T_{95}, starts>
:
<checkpoint 5>
<T_{96}, starts>
:
<T_{97} starts>
:
<T_{100} starts>
:

as recent checkpoint is checkpoint 5, recent trans is T_{95}, so at the time of recovery $T_{95}, T_{96}, \ldots T_{100}$ need to be considered.

4.37 ②

Recovery with Concurrent Transactions:

We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.

- All trans share a single disk buffer & a single log.
- A buffer block can have data items updated by one or more transactions.

We assume concurrency control using strict-two phase locking:
- i.e the updates of uncommitted transactions should not be visible to other transactions.

otherwise how to perform undo if T_1 updates A, then T_2 updates A & commits, & finally T_1 has to abort?

Checkpoint log record is of the form:
- <checkpoint L>
  where L is the list of transactions active at the time of checkpoint.
- We assume no updates are in progress, while the checkpoint is carried out.

When sys recovers from a crash, it first does the following:
- (i) initialize undo-list & redo-list to empty.
- (ii) Scan the log backwords from the end, stopping when the first <checkpoint L> record is found.

For each record found during the backward scan:

- if the record is <Ti Commit> add Ti to <redo-list>

- if the record is <Ti start> then if Ti is not in redo-list add Ti to undo list.

- For every Ti in L, if Ti is not in redo-list, add Ti to undo-list.

Once the lists are created, the recovery proceeds as follows:

- The sys again scans the log from most recent record in backward, & performs undo for each log record belongs to Ti in undo list. Log records of redo trans list is ignored.
  The scan stops when <Ti start> records have been found for every trans Ti in undo list.

- The sys locates the most recent <checkpoint L> record on the log.

- The sys scans log in forward from most recent <checkpoint L> record & performs a redo for each log record that belongs to Ti that is an redo list. It ignores log records of trans on the undolist in this phase.

undo - reverse
redo - forward
only operations after <checkpoint L> has to be undo or [illegible]

Ex:

< T1 Start >

< write, T_1, B, 2, 3 >

< start T_2 >

< Commit, T_1 >

< write T_2, C, 5, 7 >

< checkpoint, { T_2 } >

< start, T_3 >

< write, T_3, A, 1, 9 >

< Commit, T_3 >

< Start, T_4 >

< write, T_4 C, 7, 2 >

imagine that there is a crash.

undo list ↑ backward.

{ T_4, T_2 } — active transaction, as it is not committed.

redo list → ↓ forward.

{ T_3 }

As T_1 has committed before checkpoint list it does not appear in undo or redo list.

order : T_4 (undo), T_2 (undo), T_3 (redo).

(undolist first then redolist)

order of operations:

(old)

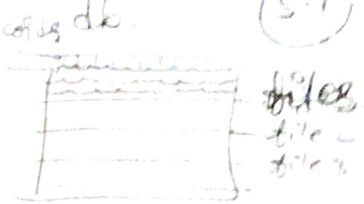
T_4 : C : 7.

T_2 : X (nothing appears so no operation is done).

T_3 : A : 9. redo is done.

we have to redo or undo after the transaction after checkpoint.

Data on External Storage:



- DBMS stores vast quantities of data.
- Data is stored on external storage devices & fetched into main memory as needed for processing.
- Page is unit of information read from or written to disk. (in DBMS a page may have size 8KB or more).
- Data on external storage devices:
  - Disks – Can retrieve random page at fixed cost.

    (But reading several consecutive pages is much cheaper than reading random order).
  - Tapes – Can only read pages in sequence (Cheaper than disks).
- Record id (rid) – used to identify the disk address of the page containing the record by using the rid.
- Buffer manager: Layer of s/w used to read data into memory for processing, & written to disk for persistent storage.
- File Access layer: Takes the help of: buffer manager layer & fetches the page, specifying the page's rid.
- Disk space mgr: Manages the space on the disk.

  (provides commands to allocate / deallocate, read / write a page to external storage).

File organizations & Indexing:

File organization:

- Method of arranging a file of records on external storage.
- record id (rid) is used to physically locate the record.
- The file layer keeps tracks of pages allocated to each file, & as records are inserted into deleted from the file, it also tracks available space within pages allocated to the file.

File structures:

(a) - Heap file - Places the records on the disk in random order.
Retrieval of records (particular record) is specified by its rid.

(b) Index: (Indexing in db sys is similar to the one we see in books).

An index is a datastructure that organizes data records on disk to optimize certain kinds of retrieval operations.

An index allows us to efficiently retrieve all records that satisfy search conditions on the search key fields of the index.

Updates are faster.

ex: emp records
we can store records in a file organized as an index on employee age.

Data entry – refers to the records stored in an index file.

- A data entry with search key value k, denoted as **k***, contains enough info to locate data records with search key value k.

In a data entry k*, alternatives include that we can store:

- alternative 1: Full data record with key value k,
  - 2: <k, rid of data record with search key value k> or
  - 3: <k, list of ~~records~~ rids of data records with search key k>

~~clustered indexes~~

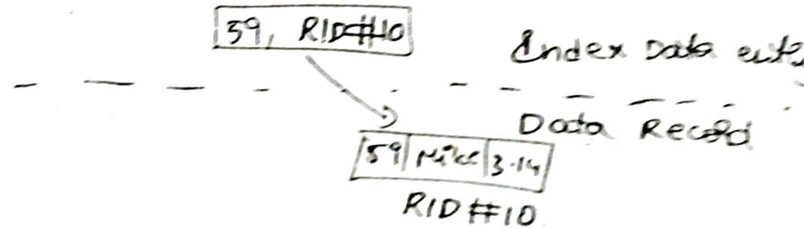
ex:

- Alternative 1: <k>

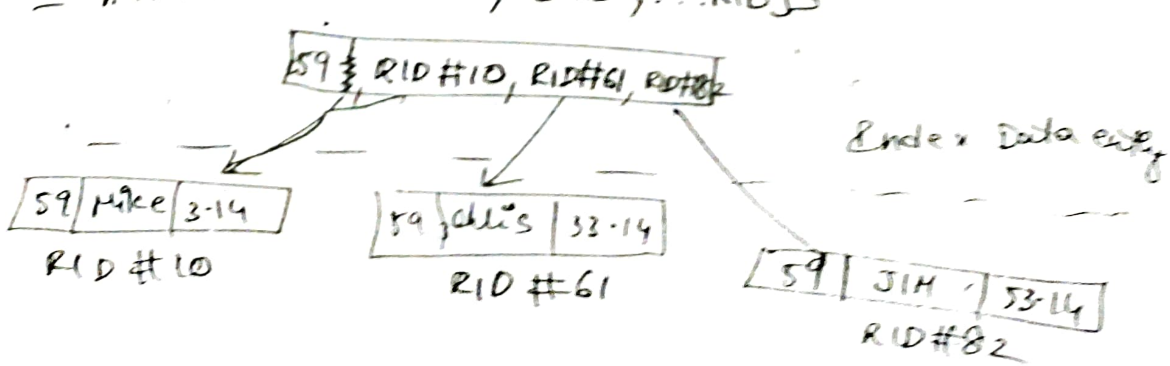
  | 59, Mike, 3.14 |

  Index data entry

- Alternative 2: <k, RID>



- Alternative 3: <k, {RID, ... RID}>



Clustered Indexes. (Primary key + ordered)

A clustered index determines the order in which the rows of a table are stored on disk. If a table has clustered index, then the rows of that table will be stored on disk in the same exact order as the clustered index.

ex: Table — owners

owner name	owner age

cars

car type	owner name

Let's assume that a given owner can have multiple cars - so a single owner-name can appear multiple times in the cars table. Now let's say that we create a clustered index on the owner-name col in the cars table.

Then a given owner-name ~~with~~ would have all his/her car entries stored.

(clustering or grouping of similar values)

Records are used frequently will be placed together

clustered index is the type of indexing that established a physical sorting order of rows.

Suppose we have a table stud which contains rollno as a PK then clustered index is self created on that primary key will sort the stud table as per rollno.

ex: Dictionary

in dic sorting order is alphabetical order so no separate index page

Non-Clustered Index:

A non-clustered index is a special type of index in which the logical order of the index does not match the physical stored order of the rows on disk.

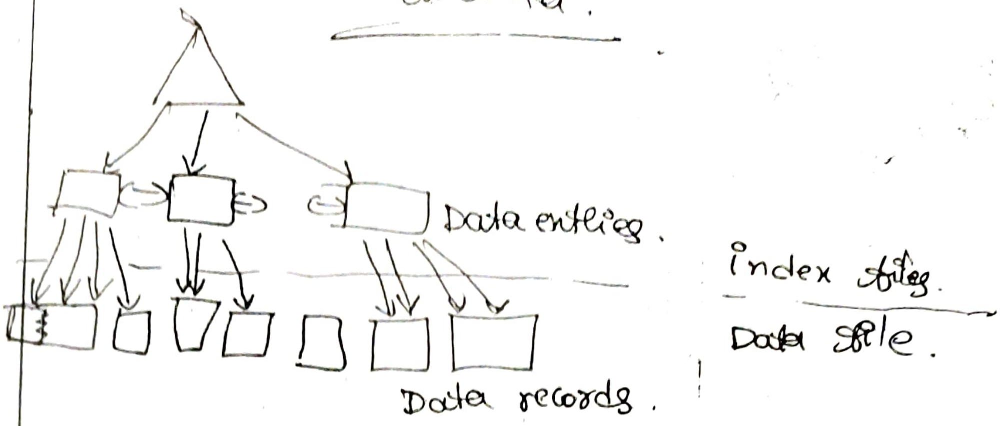
The leaf node of a non-clustered index does not consist of data pages. Instead, the leaf nodes contain index rows.

(Non clustered indexes are like simple indexes)

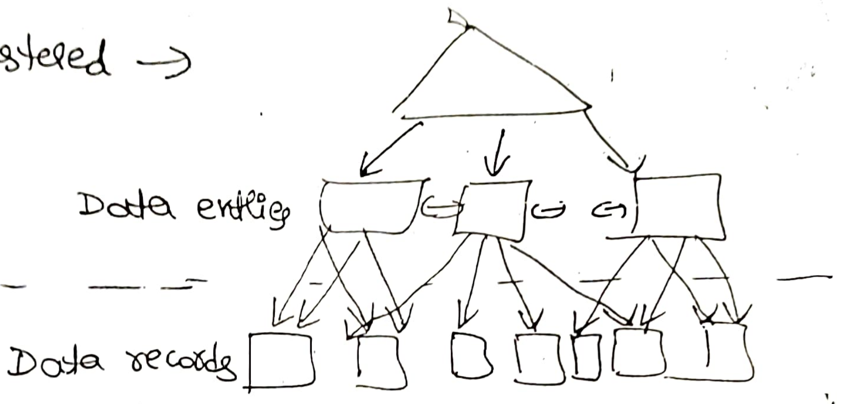
The leaf pages doesnot contain any actual data, but instead contain pointers to the actual data. These pointers would point to the clustered index data page where the actual data exists.

ex: Text book, the index page is created separately at the begining of that book.

clustered.



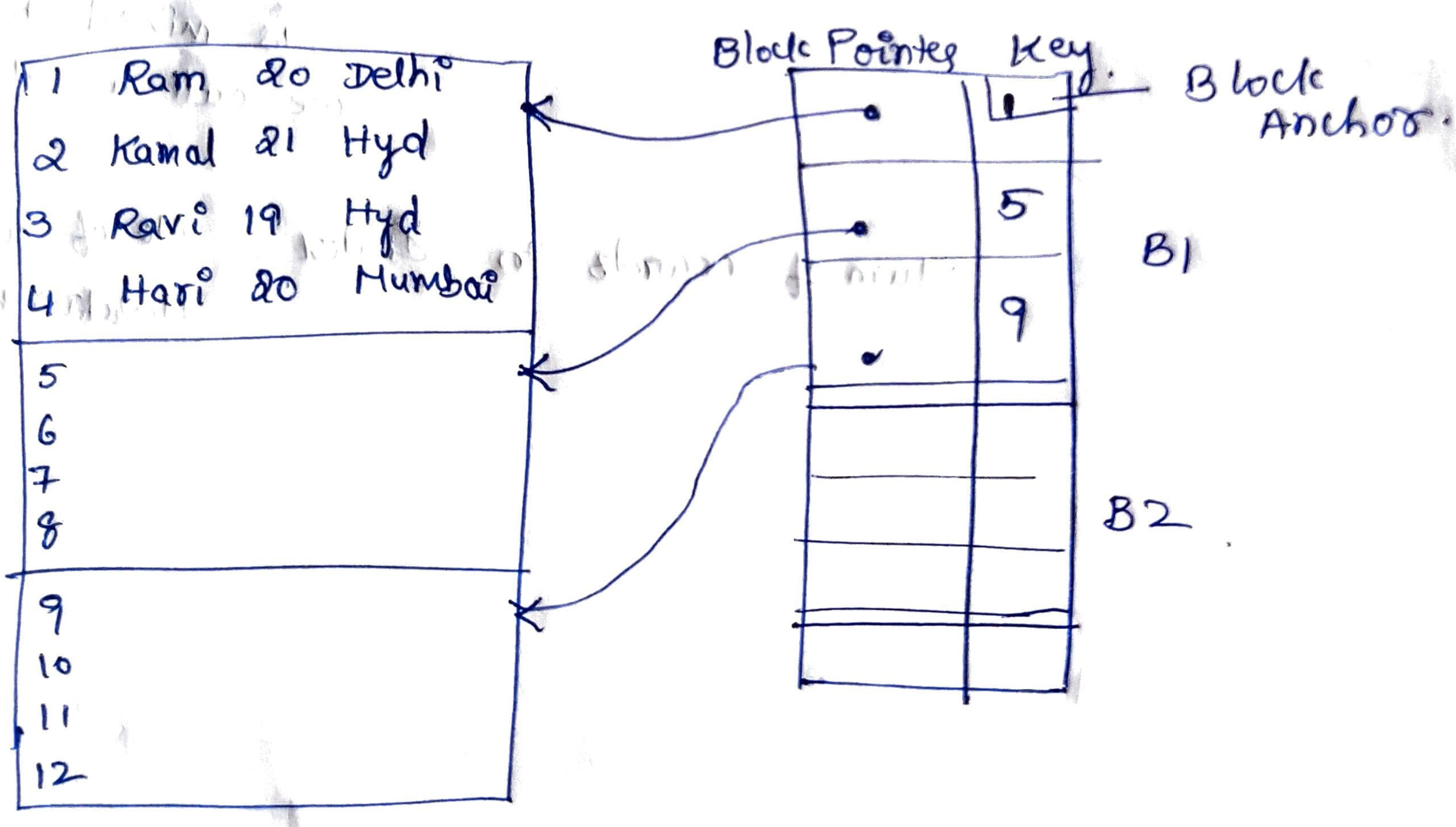
unclustered →



Primary Index. (ordered, key).

Primary index is defined on an ordered data file. The data file is ordered on a key field. The key field is generally the primary key of the relation.

Ex: Stud table. Index.



(Sparse)

No of entries in Index Table = No of blocks in HD.

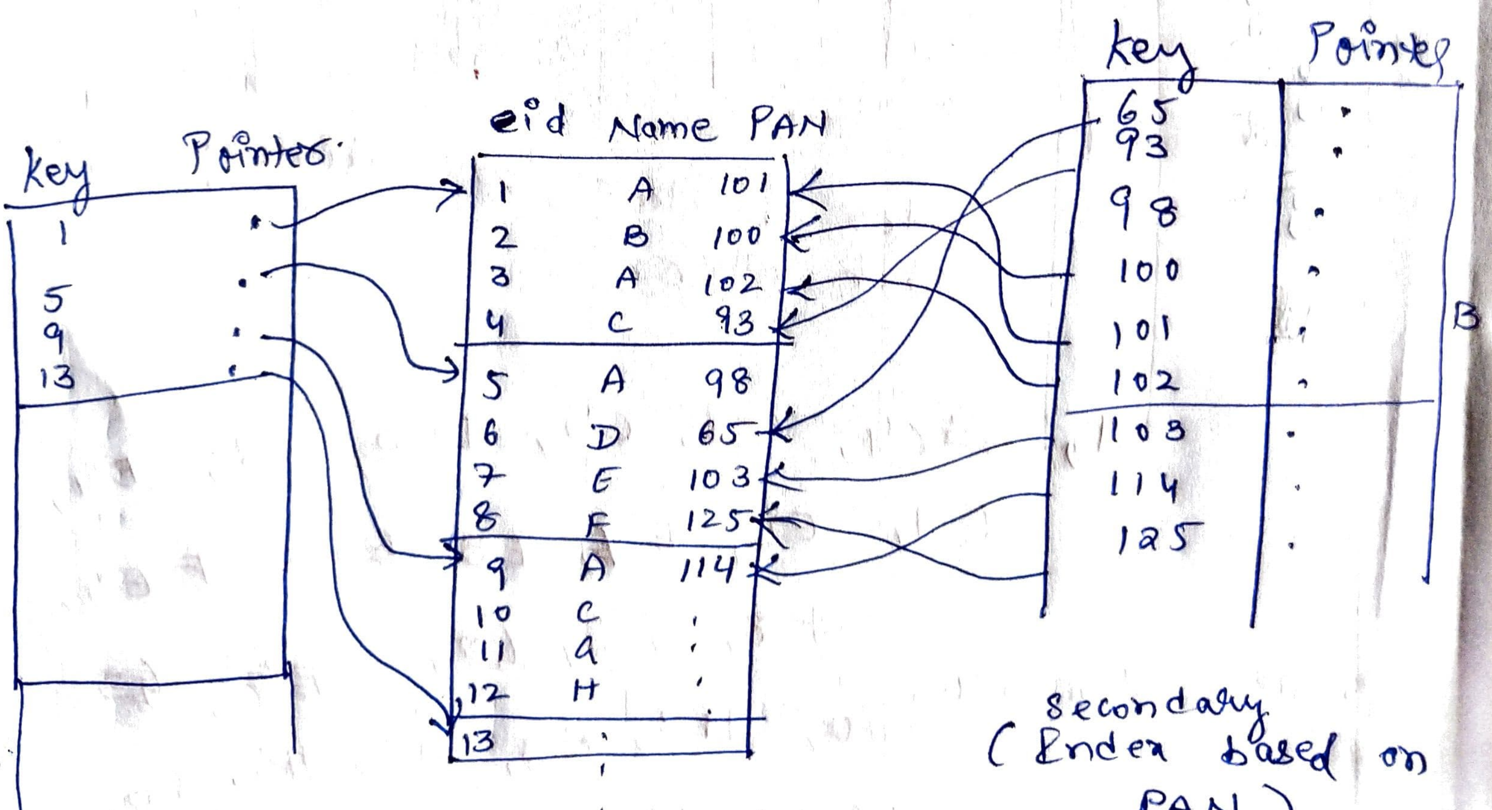
Secondary Index.

→ Index (Unique value) is created for each record in a data file which is a candidate key

→ Secondary Index is a type of dense Index & also called as non clustering index.

→ Secondary mapping size will be small as the two level DB indexing is used.

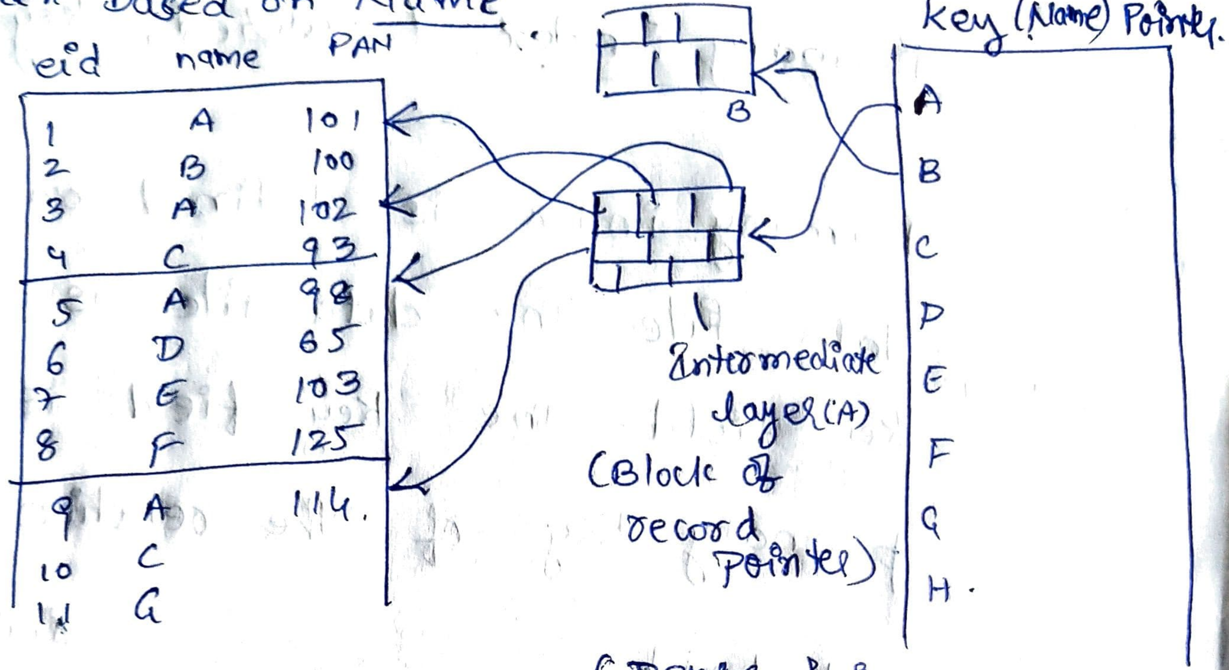
Note: unordered Index. (Key, Non key)



[Primary Index]

Secondary (Index based on PAN)

Secondary Index based on Name.



(Dense & Sparse is mixed but we call as Dense).

Num of records in Index = No of Records in HardDisk.

Ordered Indices:

There are two types of ordered indices:

(a) Dense Index (b) Sparse Index.

(a) Dense Index:

- An index record appears for every search key value in file.
- This record contains search key value and a pointer to the actual record.
- This makes searching faster but requires more space to store index records itself.

Ex: (1)

Search key value (block anchor)

Search key	Block pointer
Brighton	→
Downtown	→
Mianus	→
Perriridge	→
Redwood	→
Round hill	→

Brighton	217	Green	750
Downtown	101	Johnson	500
Downtown	110	Peterson	600
Mianus	215	Smith	700
Perriridge	102	Hayes	400
Perriridge	201	Williams	900
Perriridge	218	Lyle	700
Redwood	222	Lindsay	700
Round hill	305	Turner	350

Ex: (2)

Search key value

100	→
101	→
102	→
103	→
104	→
105	→

Stud id	sname	age
100	Joseph	20
101	Allen	20
102	Chris	21
103	Patty	22
104	James	19
105	Antony	22

(b) Sparse Index:

- An index entry appears for only some tuples.
- Sparse index can be used only if the relation is stored in sorted order of the search key.
- To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
- We start at that record pointed to by the index record, & proceed along the pointers in the file until we find the desired record.

Ex: (1)

(SK)

Search key	Block pointer
Brighton	→ Brighton
Mianus	→ Mianus
Redwood	→ Redwood

(ST)

Brighton	217	Green	750	Block 1
Downtown	101	Johnson	500	
Downtown	110	Peterson	600	
Mianus	215	Smith	700	Block 2
Perryridge	102	Hayes	400	
Perryridge	201	Williams	900	
Redwood	222	Lindsay	700	Block 3
Roundhill	305	Turner	350	

Ex: (2)

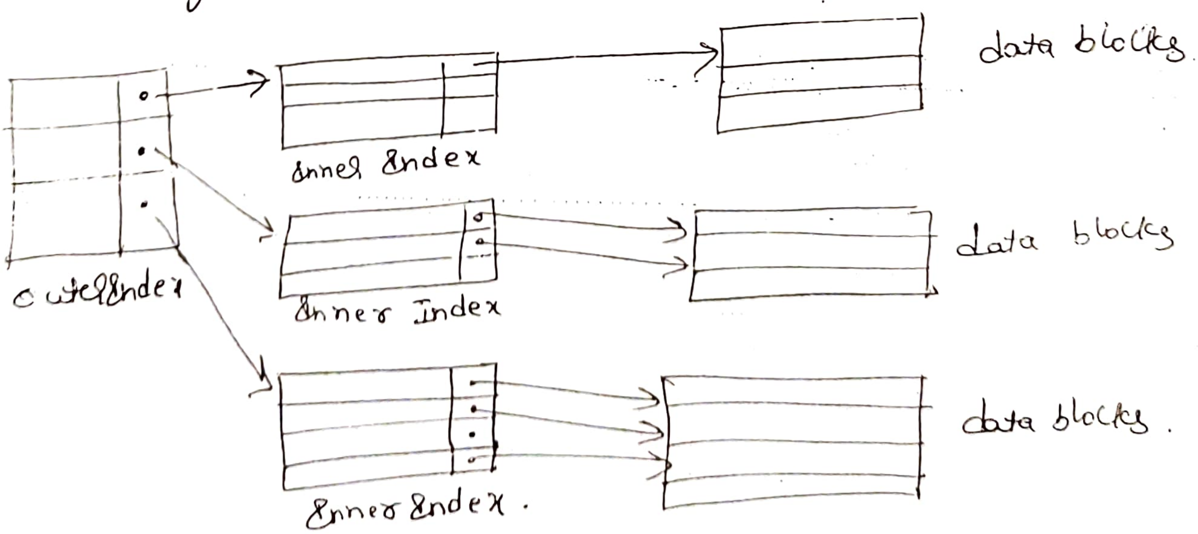
100	→ 100
103	→ 103
106	→ 106

100	Joseph	20
101	Allen	20
102	Chris	21
103	Allen	22
104	Jack	21
105	James	22
106	Paul	23
107	Antony	20
108	Amit	21

if we want to search with ID 102, then the address for the ID less than or equal to 102 is searched - which returns the address of ID 100. This address location is then fetched linearly till we get the records for 102. Hence it makes the searching faster & also reduces the storage space for indexes.

Multi-level Indexes:

- Indexes with two or more levels are called multilevel indexes
- Multilevel 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 main memory.

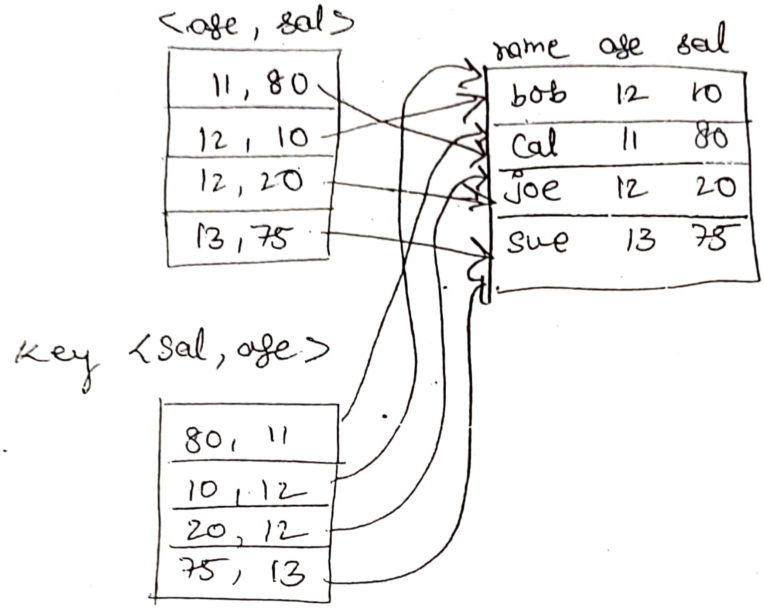


Ex: dictionary

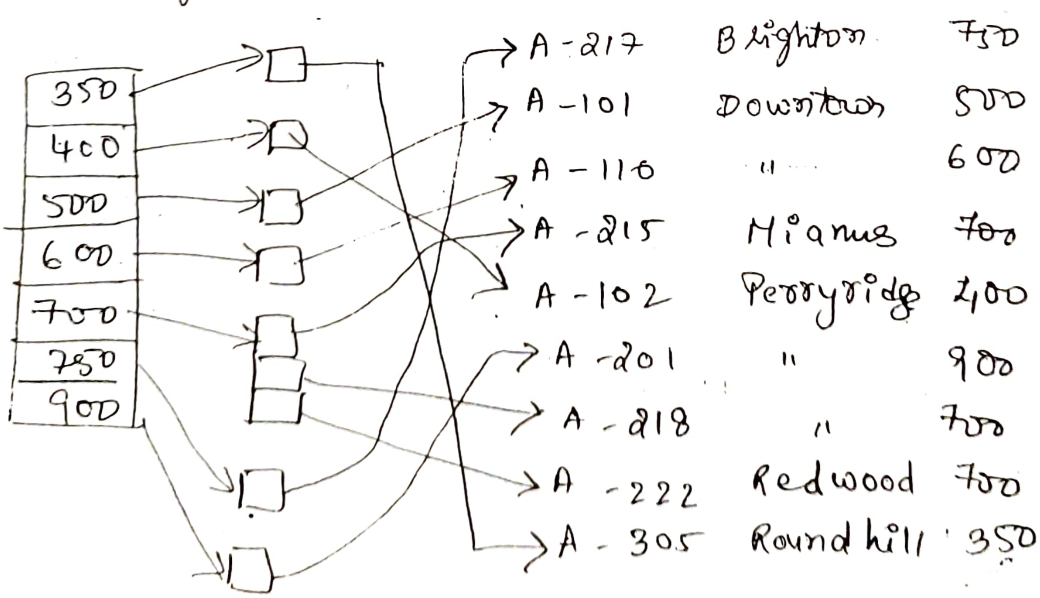
Indexes using Composite Search Keys.

The search key for an index can contain several fields, such keys are called composite search keys.

Ex: Key <age, sal>.



Secondary Indices Example:

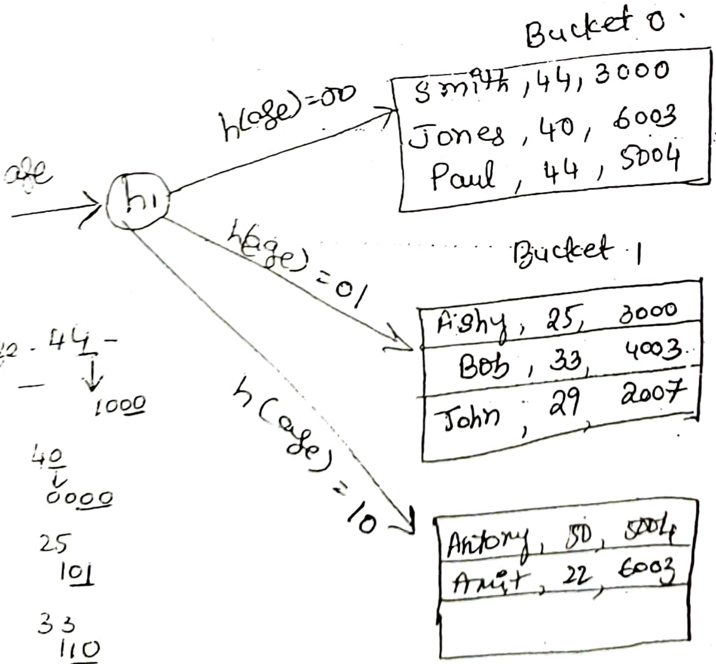


Hash-Based Indexing

Hashing is an effective technique to calculate direct location of data record on the disk without using index structure.

- Bucket: Bucket is a unit of storage. A bucket stores one complete disk block, which in turn can store one or more records. Hash file stores data in bucket format.

- 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 address.



age 44 → 1000

40 → 0000

25 → 101

33 → 110

29 → 1001

50 → 00

22 → 10

There are 2 types of hashing:

- Static
- Dynamic
  - extendable
  - linear

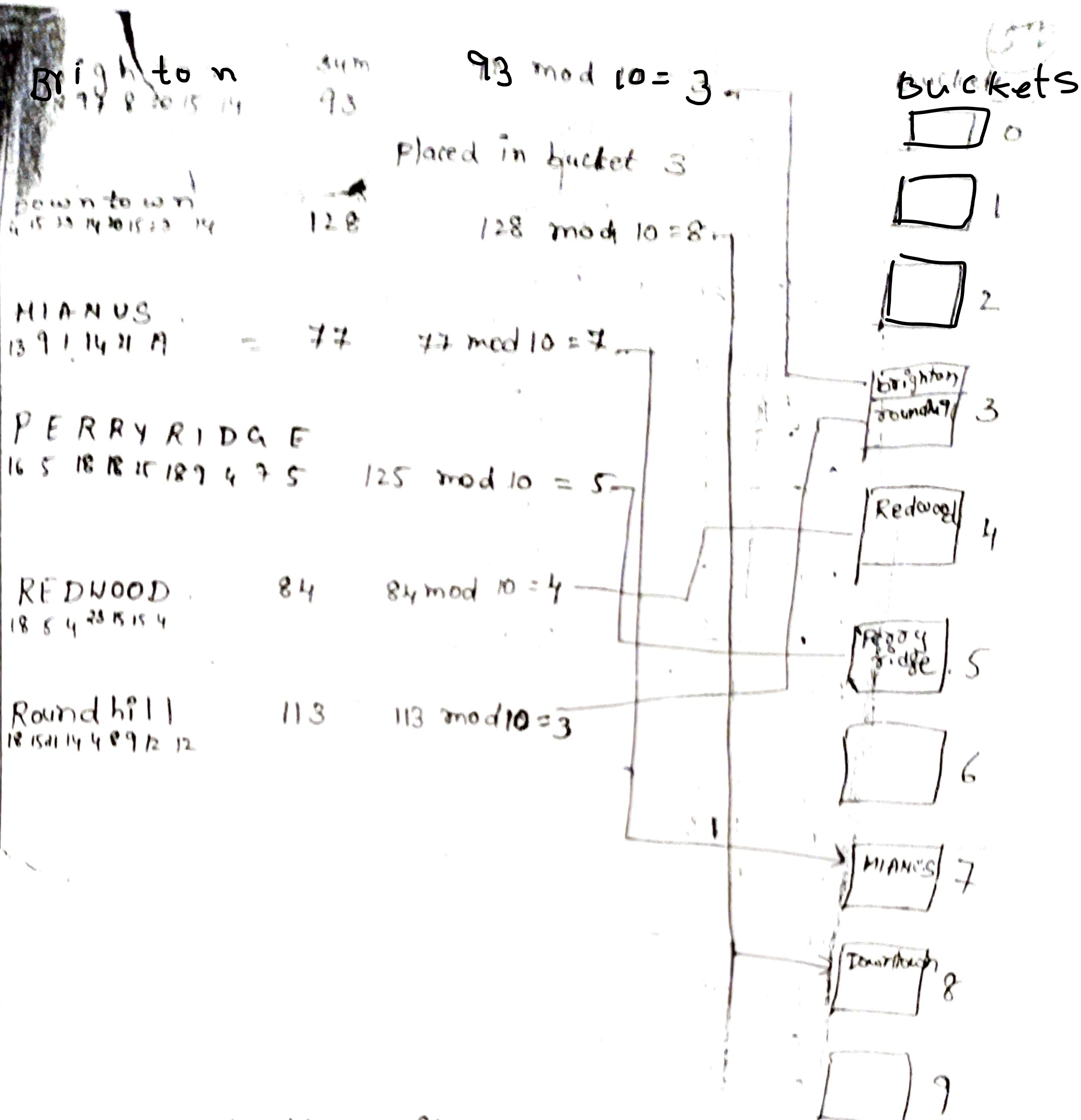
(1) __Static hashing__: Num of buckets are __fixed.__
each bucket holds num of records.
In this each bucket holds num of records then if we want to give more num of records then overflow buckets are used.

- We have to choose a hash fun that assigns search key values to the buckets in such a way that the distribution has the following qualities.
- distribution is uniform
  " " random.

eg: Let the search key value be branch name, the total num of buckets are 10. Apply this on the accounts relation.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26

	Account no	Branchname	Balance.
2000/100	A217	Brighton	750
2000101	A101	downtown	500
2000102	A110	downtown	600
200.103	A215	Mianus	700
2000104	A102	Perryridge	400
	A201	"	900
	A218	"	700
	A222	Redwood	700
	A305	Round hill	350



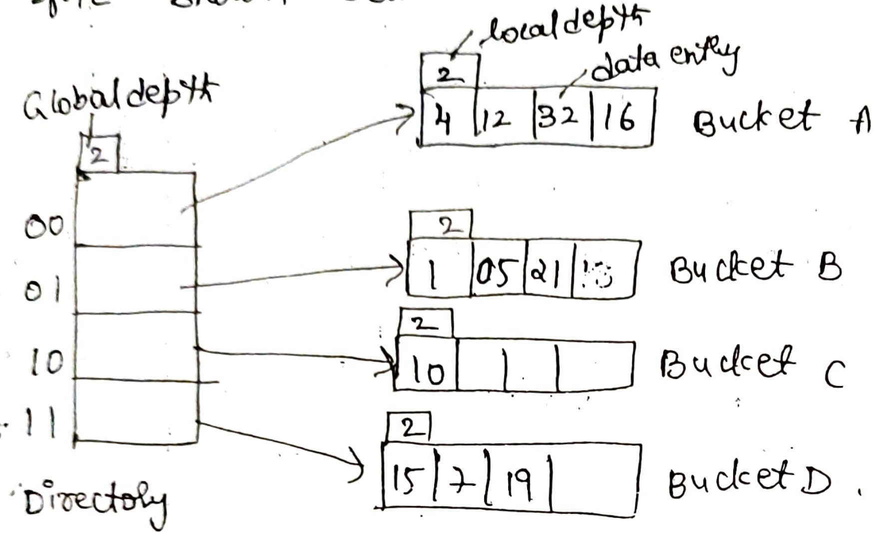
The main problem with static hashing is that the num of buckets is fixed. If a file shrinks greatly, a lot of space is wasted; more important, if a file grows a lot, long overflow chains develop, resulting in poor performance.

Deletion – If we want to delete a record, using the hash function we will first fetch the record which is supposed to be deleted. Then we will remove the records for that address in memory.

Dynamic hashing: → It is a dynamic hashing method wherein directories & buckets are used to hash data

Extendible Hashing:

To understand the idea, consider the sample file shown below



Directories: store addresses of buckets in pointers. An id is assigned to each directory which may change each time when dir expansion takes place.

Global depth: associated with directories. They denote the num of bits which are used by the hash func to categorize the keys.

Local depth: It is associated with buckets. Local depth is always less than or equal to global depth.

Bucket splitting: When num of elements in a bucket exceeds a particular size, then the bucket is split into two parts.

Directory expansion: Takes place when a bucket overflows.

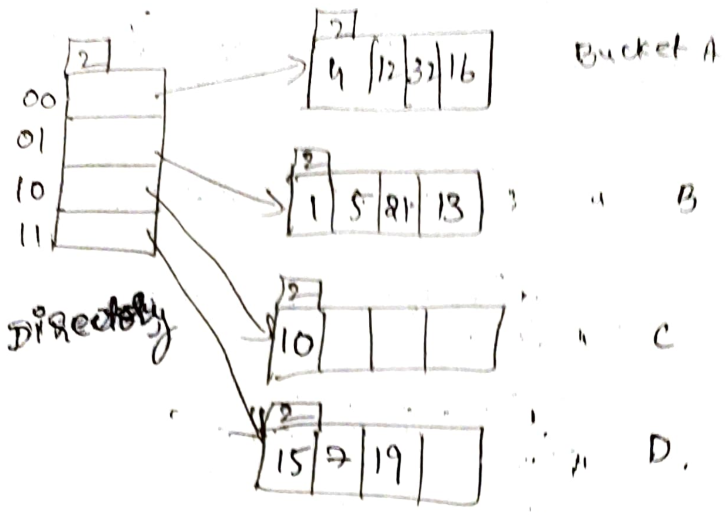
4 - 0100
12 - 1100
32 - 100000
16 - 10000
} Bucket A.

1 - 0001
5 - 0101
21 - 10101
} Bucket B

10 - 1010 — Bucket C.

15 - 01111
7 - 00111
19 - 10010
} Bucket D.

Now if we want to insert 13 the binary value is 1101. So we have to store in Bucket B.



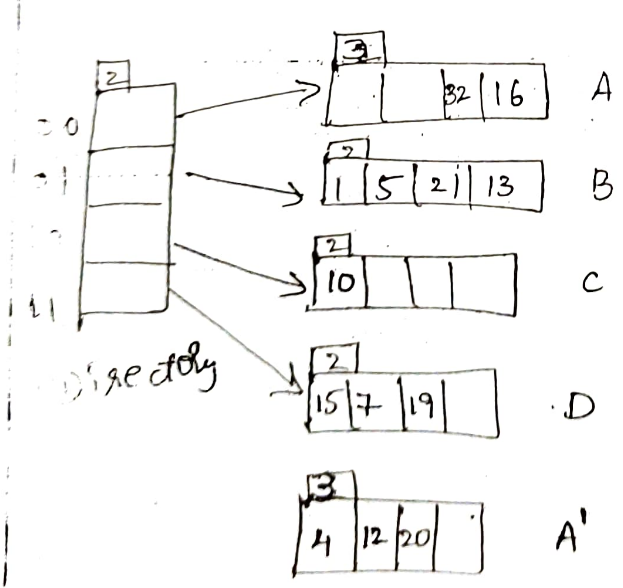
Now we want to insert 20

binary value is 10100

we have to store in Bucket A but it is already full.

We must first split the bucket by allocating a new bucket & redistributing the contents across the bucket & its split image.

To redistribute entries across the old bucket & its split image, we consider last three bits.



A' (split image of bucket A)

4 — 0100

12 — 1100

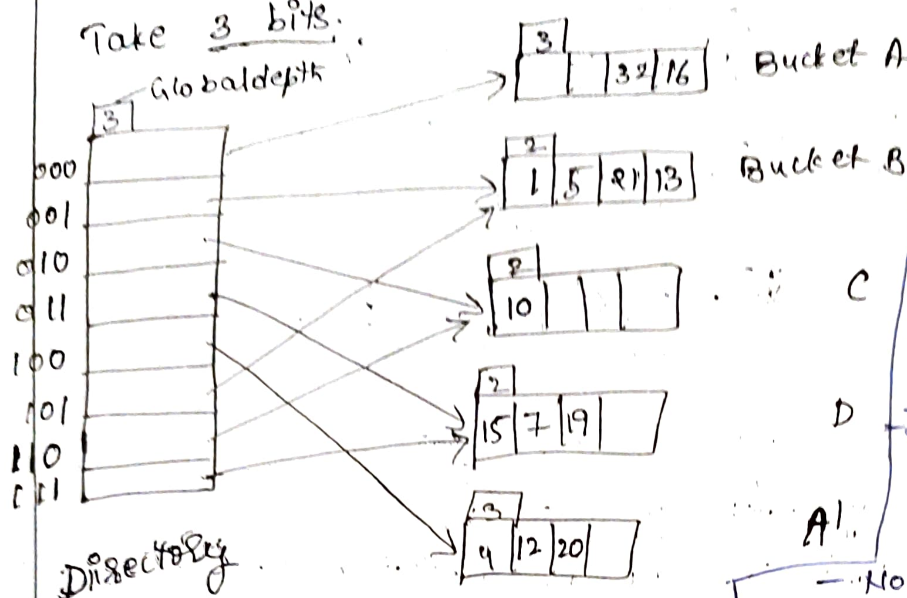
20 — 10100

32 — 100000

16 — 10000

Now the solution is to double the directory.

Take 3 bits.



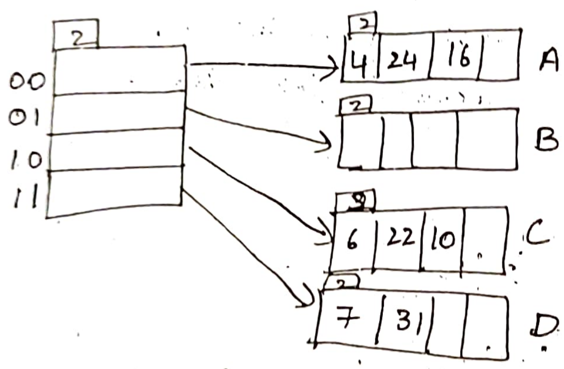
Advantages.

- Data retrieval is less expensive (in terms of computing)
- No problem of data loss since the storage capacity increases dynamically.
- With dynamic changes in hashing fun, associated old values are rehashed w.r.t the new hash fun.

Disadv:

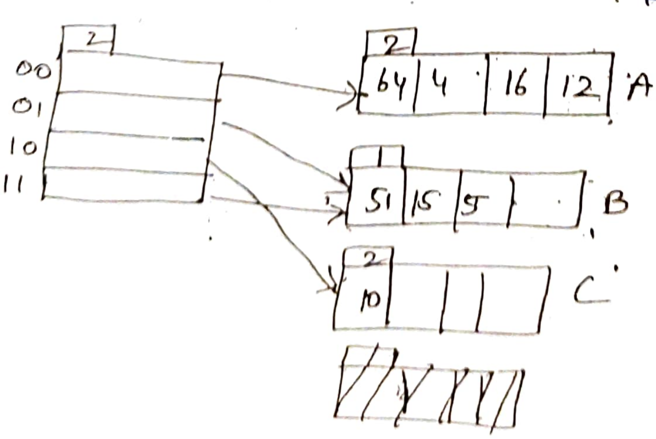
1. Size of every bucket is fixed
2. The dir size may increase significantly if several records are hashed on the same dir while keeping the record distribution non-uniform
3. This method is complicated to code.

Ex: (2) Perform extendable hashing for 4, 6, 22, 24, 16, 10, 7, 31



Insert 9, 20, 26, 28, 14.

Ex: (3) Perform extendable hashing for



insert
18
10
7
21

Linear Hashing:

The scheme utilizes a family of hash funcs $h_0, h_1, h_2, \ldots$ with the property that each funcs range is twice that of its predecessor.

If h_i maps a data entry into one of M buckets, h_{i+1} maps a data entry into one of 2M buckets.

This is obtained by choosing a hash fun h and defining an initial number N of buckets and defining

$h_i(val) = h(val) \bmod (2^i N)$.

We will describe in more detail.

- A counter Level is used to indicate the current round num & is initialized to 0.
- The bucket to split is denoted by Next & is initially bucket 0 (first bucket).
- We denote the num of buckets in the file at the begining of round Level by N_{Level}.

$N_{Level} = N * 2^{Level}$

- We can split whenever new overflow page is added.
- Whenever a split is triggered the Next bucket is split, & hash fun $h_{Level+1}$ redistribute entries between this bucket & its split image.

After splitting a bucket, the value of Next is incremented by 1.

Level = 0 N = 4

h_1	h_0	Primary pages	overflow pages
000	00	Next = 0 → [32 | 44 | 36 |] Bucket 0	
001	01	[9 | 25 | 5 |] Bucket 1	
010	10	[14 | 18 | 10 | 30] Bucket 2	
011	11	[31 | 35 | 7 | 11] Bucket 3	

This info is for illustration only

The actual contents of the linear hash table.

Insert data entry [43]

43 % 4 = 3 (or) 43 – 1010<u>11</u>

store in Bucket 3.

As bucket 3 is full store it in overflow pages.

Level = 0.

h_1	h_0	Primary Pages	overflow pages
000	00	Next = 0 [32 | 44 | 36 |] B0	
001	01	[9 | 25 | 5 |] B1	
010	10	[14 | 18 | 10 | 30] B2	
011	11	[31 | 35 | 7 | 11] B3 →	[43 | | |]

Now increment Next = 1.

(5.9a)

check with last 3 bits of
perform contents of bucket 0 with mod 8.
the we will have

Level = 0.

h_1	h_0	Primary pages		overflow pages.
000	00	32	B0	
001	01.	Next=1 9 | 25 | 5	B1	
010	10	14 | 18 | 10 | 30	B2	
011	11.	31 | 35 | 7 | 11	B3	43
100	00	44 | 36	B4	

(or)
32 % 8 = 0
32 – 100000 – 0
44 % 8 = 4
36 % 8 = 4.

Now insert. 37 ⇒ (37 % 4 = 1. (or) 37 – 100101).

Level = 0

h_1	h_0	Pri pages		overflow pages.
000	00	32	0	
001	01	Next=1 9 | 25 | 5 | 37	1	
010	10	14 | 18 | 10 | 30	2	
011	11	31 | 35 | 7 | 11	3	43
100	00	44 | 36	4	

Now insert 29. (29 % 4 = 1 (or) 29 = 11101)

Bucket 1 is full.

Increment Next = 2

split is triggered & don't need overflow bucket.

Level = 0.

h_1	h_0	Pri pages		overflow pages
000	00	[32 | | |]	0	
001	01	[9 | 25 | |]	1	
010	10	Next=2 [14 | 18 | 10 | 30]	2	
011	11	[31 | 35 | 7 | 11]	3	[43 | | |]
100	00	[44 | 36 | |]	4	
101	01	[5 | 37 | 29 |]	5	

9 % 8 = 1 (or) 9 - 1001 - 1

25 % 8 = 1 (or) 1001 - 1

5 % 8 = 5 (or) 101 - 5

37 % 8 = 5 100101 - 5

29 % 8 = 5 11101 - 5

Now insert 22, 66, 34.

22 % 4 = 2 (or) 22 - 10110 - 2

66 % 4 = 2 (or) 66 - 1000010 - 2

34 % 4 = 2 (or) 34 - 100010 - 2

As bucket is full increment Next = 3

& take last 3 bits or do

22 % 8 = 6 (or) 10110 - 6.

66 % 8 = 2 (or) 1000010 - 2

34 % 8 = 2 (or) 100010 - 2

in Bucket 2.

14 % 8 = 6. 30 % 8 = 6

18 % 8 = 2

10 % 8 = 2

Level = 0

h_1	h_0	Primary pages		overflow pages
000	00	32	0	
001	01	9, 25	1	
010	10	66, 18, 10, 34	2	
011	11	Next=3. 31, 35, 7, 11	3	43
100	00	44, 36	4	
101	01	5, 37, 29	5	
110	10	14, 30, 22	6	

Now insert 50

$50 \% 8 = 2$

$50 - 110010$

This has to be inserted in Bucket 2, but as bucket 2 is full we will increment level to 1

Level = 1

h_1	h_0	Primary pages		overflow
000	00	Next=0 32	0	
001	01	9, 25	1	
010	10	66, 18, 10, 34	2	50
011	11	43, 35, 11	3	
100	00	44, 36	4	
101	01	5, 37, 29	5	
110	10	14, 30, 22	6	
111	11	31, 7	7	

31 % 8 = 7
35 % 8 = 3
7 % 8 = 7
11 % 8 = 3
43 % 8 = 3

- Deletion is inverse of insertion.
- If the last bucket in the file is empty it can be removed & Next can be decremented.
- If Next is 0 & the last bucket becomes empty, Next is made to point to bucket (M/2)-1, where M is the current num of buckets, level is decremented, & the empty bucket is removed.

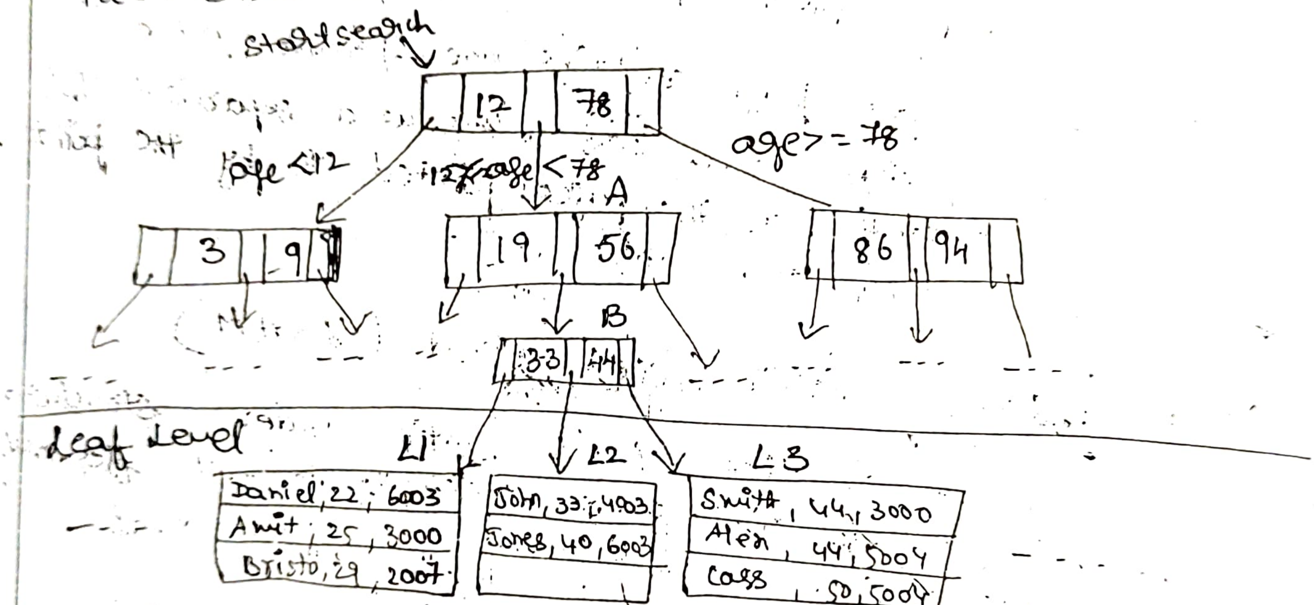
Differences between Extendable & Linear hashing.

	Extendable hashing	Linear hashing
1	This is dynamic hashing technique in which the concept of directory is used.	This is dynamic hashing technique in which the primary data pages are arranged in buckets.
2	It doesnot use the structure for overflow pages.	It uses explicit arrangement for representing the overflow page.
3.	The bucket overflow is handled by doubling the directory	The bucket overflow is handled by splitting the bucket in round robin manner.
4.	Space utilization is more	is less
5.	It may lead to reduced num of splits & higher bucket occupancy	It may cause more num of bucket splits.
6	Same hash fun is used in this by doubling the directory	The bucket split is done with the help of moving the split ahead using h_i & h_{i+1} functions in the linear hashing.

Tree-Based Indexing:

- organizing the records using a tree like data structure.
- data entries are arranged in sorted order by search key value.

below fig shows the emp records, organized in a tree-structured index with search key age.



- lowest level of the tree, called the leaf level contains the data entries.
- Top most node called the root

In the above ex. suppose we want to find all data entries with age > 24 and age < 50.

Each edge from the root node to a child node has a label that explains what the corresponding subtree contains.

In our example we look for data entries with search key value > 24 & get directed to the middle child, node A. Again examining the contents of this node, we are directed to node B. Examining the contents of node B, we are directed to leaf node L1, which contains data entries we are looking for.

The height of a balanced tree is
length of a path from root to leaf.
In the above example it is 3

The format of a page in the second index file is

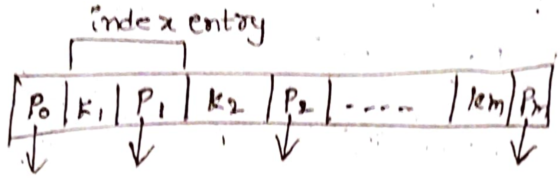
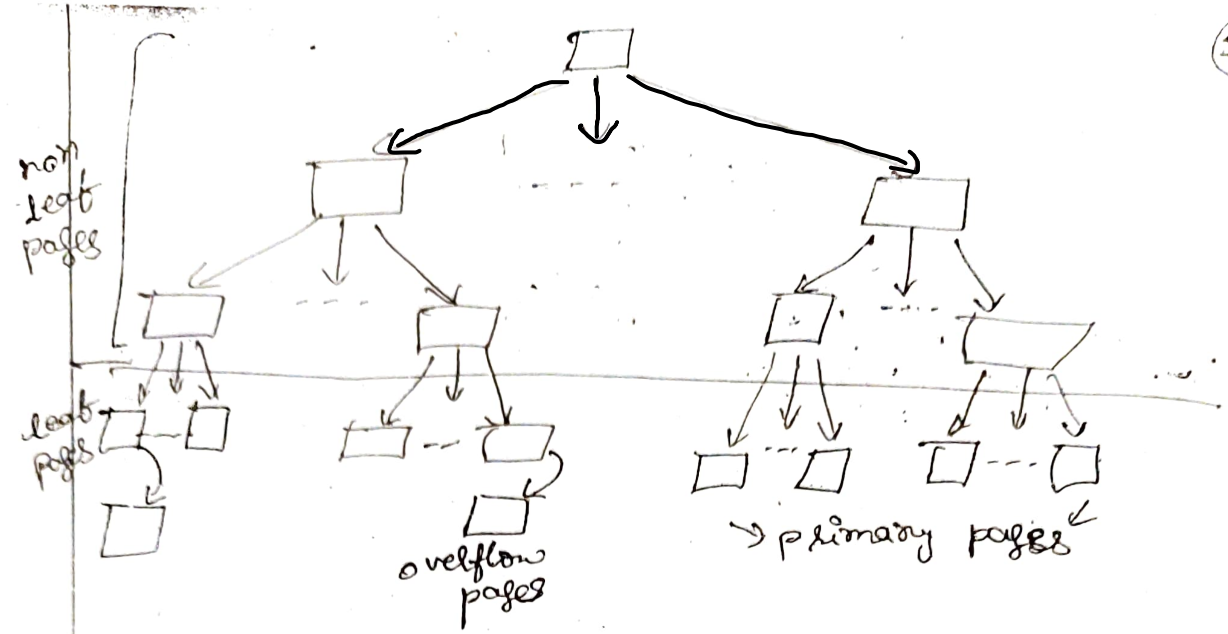


fig: format of an index page.

each index page contains one pointer more than num of keys - each key serves as a separator for the contents of the pages pointed to by the pointers to its left and right.

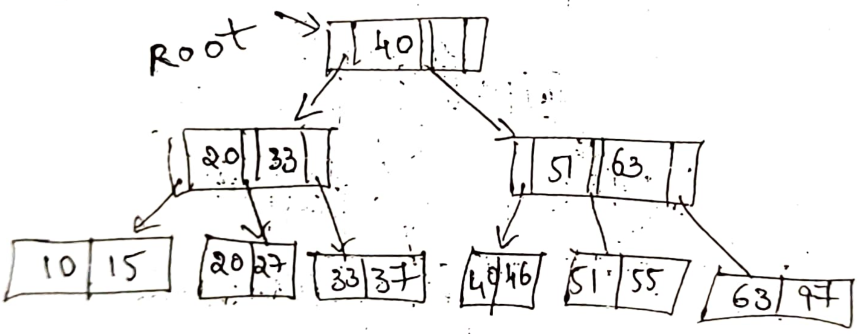
* (i) Indexed Sequential Access Method (ISAM)

An ISAM tree is a static index structure which is effective when the file is not frequently updated but is unsuitable for files that grows & shrinks a lot.
Here the leaf page contains actual data entries whereas the nonleaf node contains index entries which are used to direct the search for a desired data entry which is stored in some leaf node.
An ISAM tree consists of 3 levels.
→ non-leaf pages
→ leaf pages
→ overflow pages.



since the size is static, we can insert the data into the ~~ovef~~ overflow page & original data is placed in leaf page in sorted order.

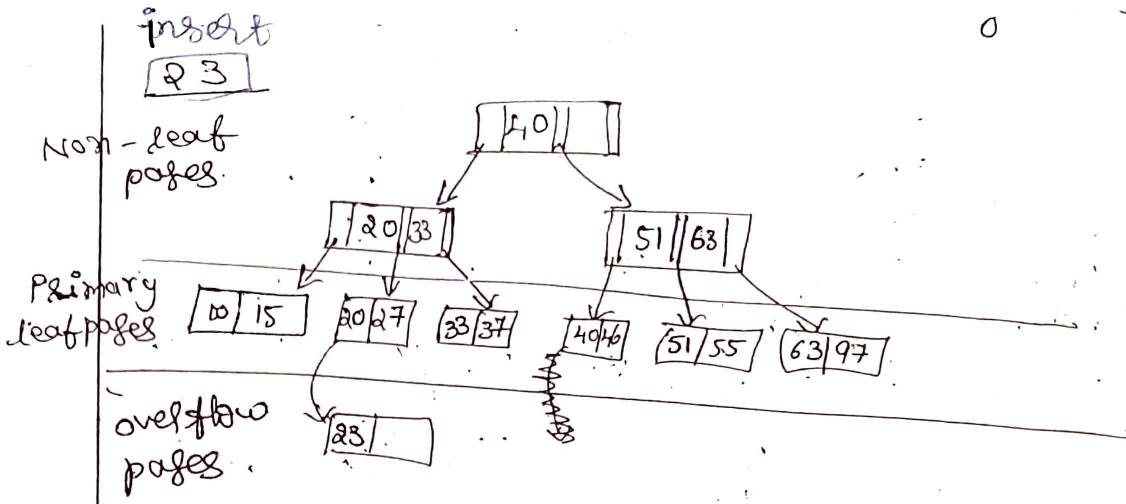
ex: Consider the tree shown below.



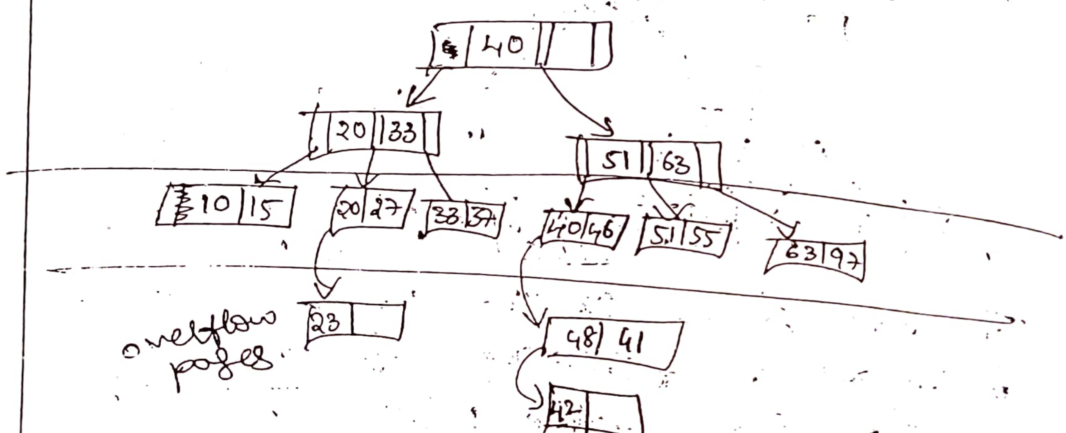
To locate record with the key value 27, we start at the root & follow the left pointer, since 27 < 40 we then follow middle pointer, since 20 <= 27 < 33.

- The primary leaf pages are assumed to be allocated sequentially, because the num of such pages is known when the tree is created & does not change subsequently under inserts & deletes & so no 'next leaf page' pointers are needed.

insert 23



The entry 23 belongs to the second data page, which already contains 20 & 27 & has no more space.
We deal this by adding an overflow page & putting 23 in overflow page.
For ex we want to insert 48, 41, 42



The deletion of an entry k* is handled by simply removing the entry. If this entry is on an overflow page & the overflow page becomes empty, the page can be removed.
If the entry is on primary page & deletion makes the primary page empty

The simplest approach is to simply leave the empty primary page as it is. (5-13)

The num of primary pages is fixed at file creation time.

The main disadvantage of ISAM is that performance degrades as file grows. To overcome this, we use B^+ tree index structure.

* B^+ trees (A Dynamic index structure). It is a balanced binary search tree. It follows multi-level index format.

- The B^+ tree search structure, is widely used balanced tree, in which the internal nodes direct the search & the leaf nodes contain the data entries.
- Data stored only in leaves.
- Internal nodes only contains keys & pointers.
- All leaves are at the same level (the lowest one)
- B^+ trees have an order n.
- An internal node can have up to $n-1$ keys and n pointers.
- Built from the bottom up.
- All nodes must have between $n/2$ & n keys.
- For a B^+ tree of order n & height h, it can hold upto n^h keys.

Format of a Node.

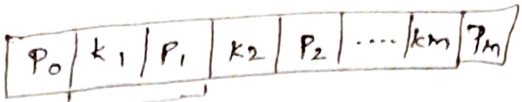


fig: Format of an Index page.

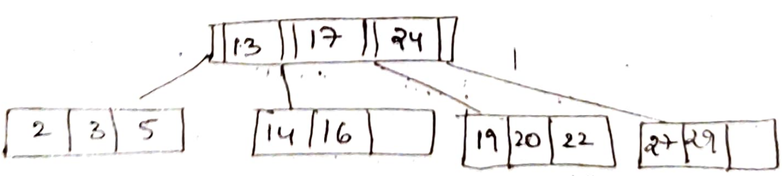
- The format of an index page is as shown above.
- Non-leaf nodes with m index entries contain m+1 pointers to children.
- Pointer P_i points to a subtree in which all key values K are such that $K_i \leq K < K_{i+1}$.

P_0 points to a tree in which all key values are less than K_1;
P_m points to a tree in which all key values are greater than or equal to K_m.
leaf node entries are denoted by $k*$

Search.

- Starts at the root, works down to the leaf level.
- The comparison of the search value & the current "separation value", goes left or right.

ex:



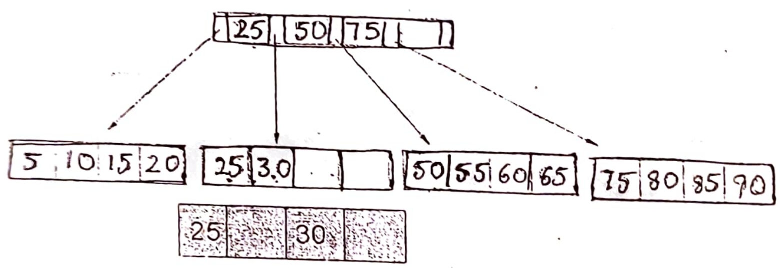
To search for entry 5, we follow the left-most child pointer, since 5 < 13.
To search for 14 we follow the second pointer since 13 < 14.

Insertion:

- A search is first performed, using the value to be added.
- After the search is completed, the location for the new value is known
- If the tree is empty, add to the root.
- Once the root is full, split the data into 2 leaves, using the root to hold keys & pointers.
- If adding an element will overload a leaf, take the median & split it.

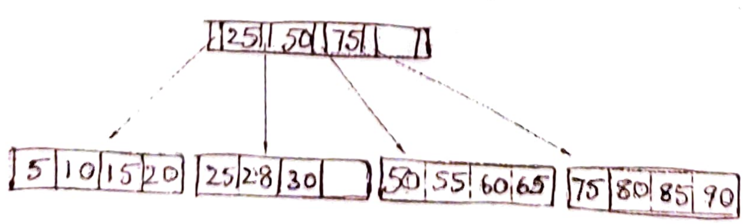
Insertion

- Since insert a value into a B+ tree may cause the tree unbalance, so rearrange the tree if needed.
- Example #1: insert 28 into the below tree.



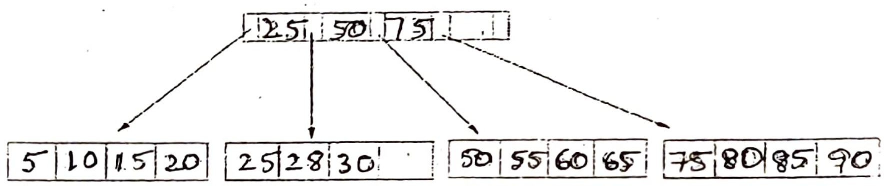
Insertion

- Result:



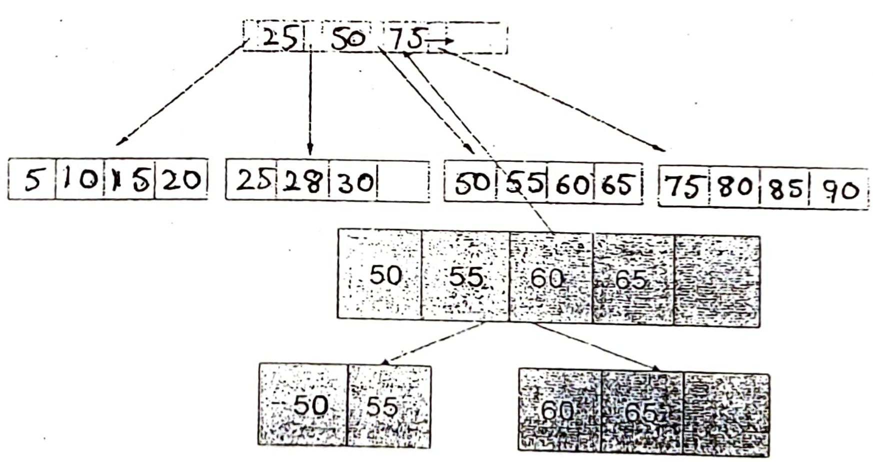
Insertion

- Example #2: insert 70 into below tree



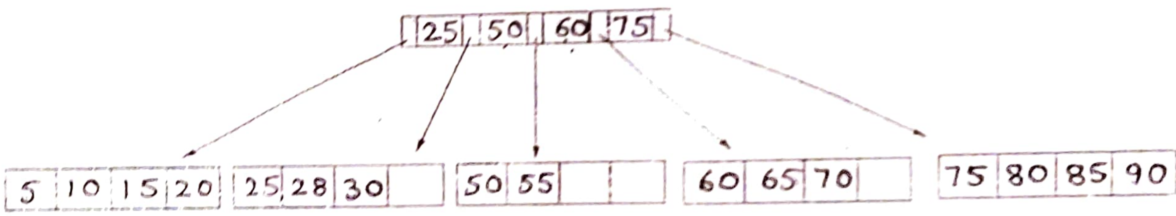
Insertion

- Process: split the tree



Insertion

- Result: chose the middle key 60, and place it in the index page between 50 and 75.



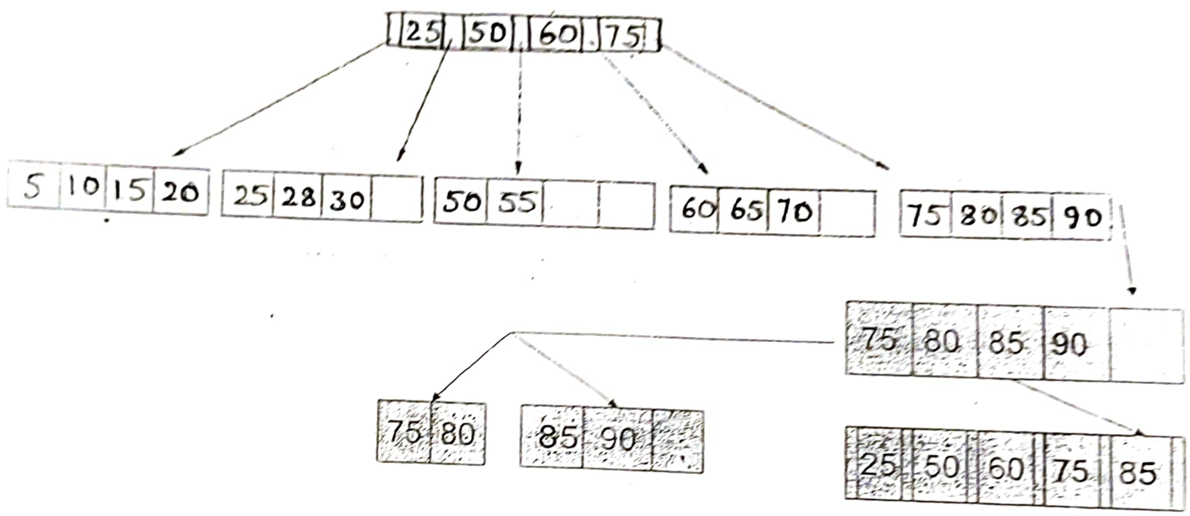
Insertion

The insert algorithm for B+ Tree

Data Page Full	Index Page Full	Action
NO	NO	Place the record in sorted position in the appropriate leaf page
YES	NO	Split the leaf page Place Middle Key in the index page in sorted order. Left leaf page contains records with keys below the middle key Right leaf page contains records with keys equal to or greater than the middle key.
YES	YES	Split the leaf page. Records with keys < middle key go to the left leaf page. Records with keys >= middle key go to the right leaf page Split the index page. Keys < middle key go to the left index page. Keys > middle key go to the right index page. The middle key goes to the next (higher level) index. IF the next level index page is full, continue splitting the index pages.

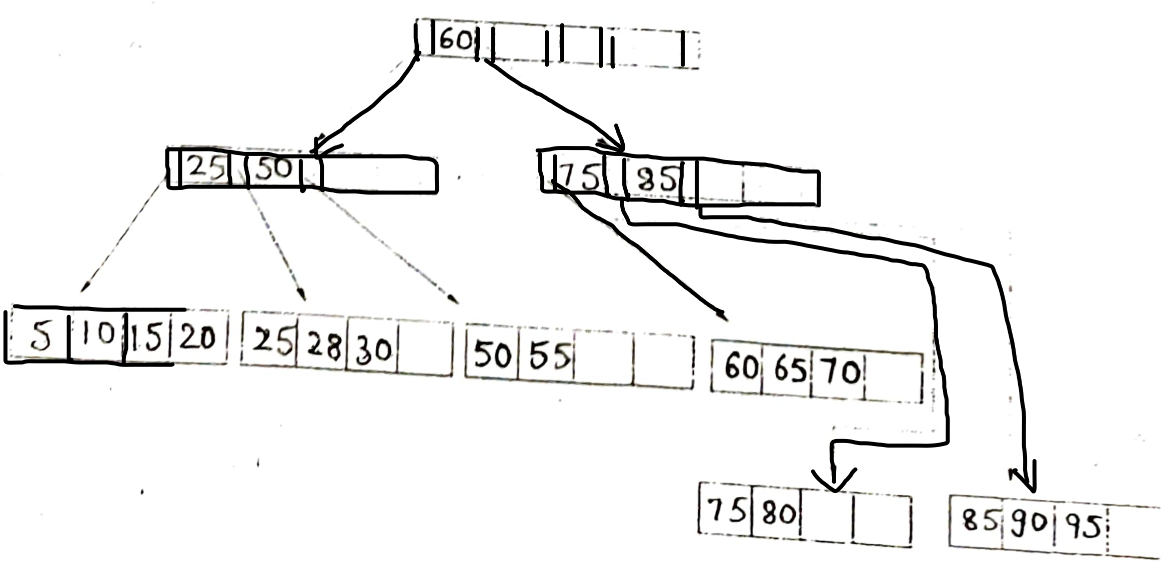
Insertion

- Exercise: add a key value 95 to the below tree.



Insertion

- Result: again put the middle key 60 to the index page and rearrange the tree.



5-19

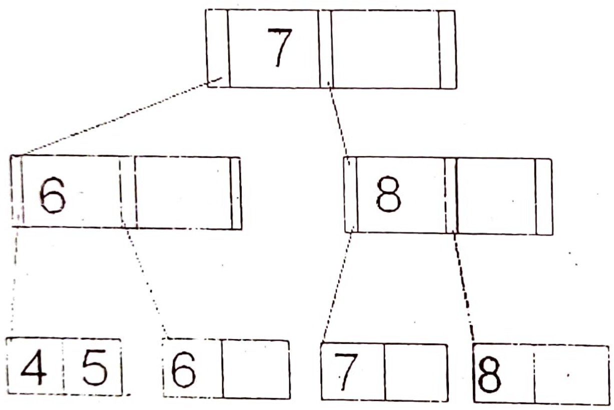
Deleting from B+ Trees

- Deletion, like insertion, begins with a search.

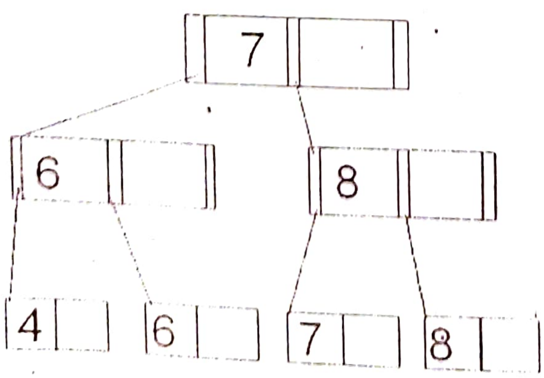
- When the item to be deleted is located and removed, the tree must be checked to make sure no rules are violated.

- The rule to focus on is to ensure that each node has at least ceil(n/2) pointers.

- example:

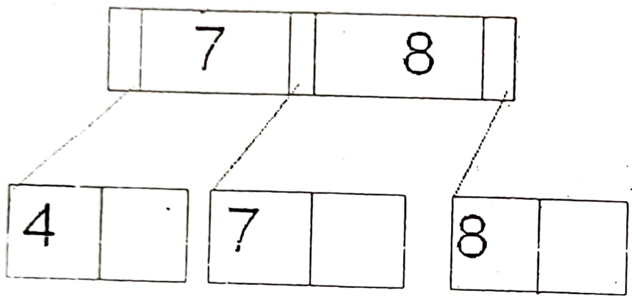


- Suppose we want to delete 5.
- This would not require any rebalancing since the leaf that 5 was in still has 1 element in it.



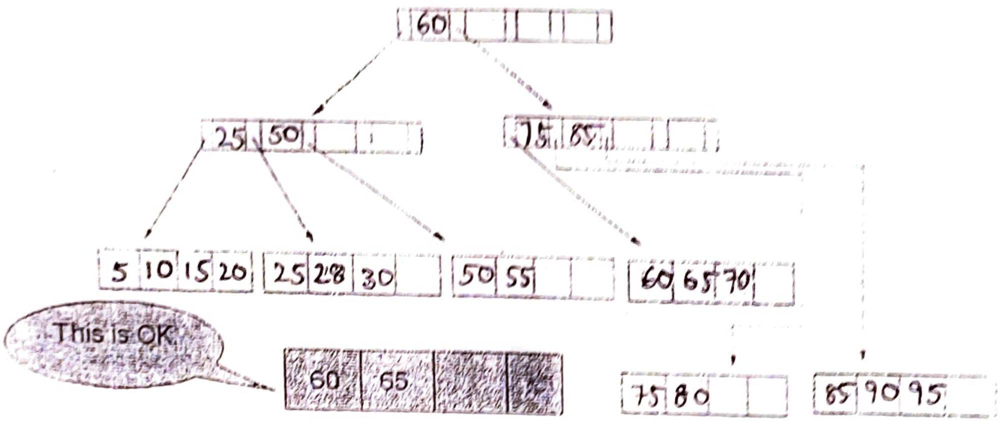
- Suppose we want to remove 6.

- This would require rebalancing, since removing the element 6 would require removal of the entire leaf (since 6 is the only element in that leaf).
- Once we remove the leaf node, the parent of that leaf node no longer follows the rule.
- It has only 1 child, which is less than the 2 required (ceil(3/2) = 2).
- Then, the tree must be compacted in order to enforce this rule.
- The end product would look something like this:



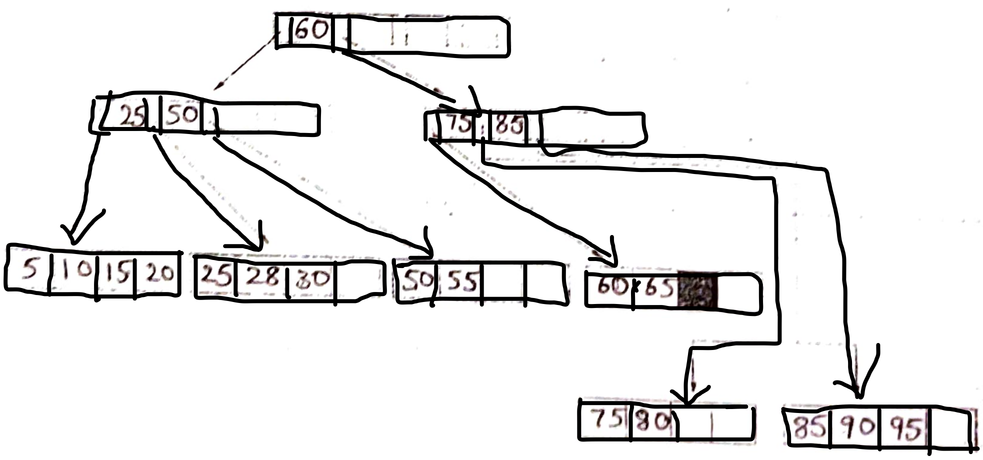
Deletion

- Same as insertion, the tree has to be rebuild if the deletion result violate the rule of B+ tree.
- Example #1: delete 70 from the tree



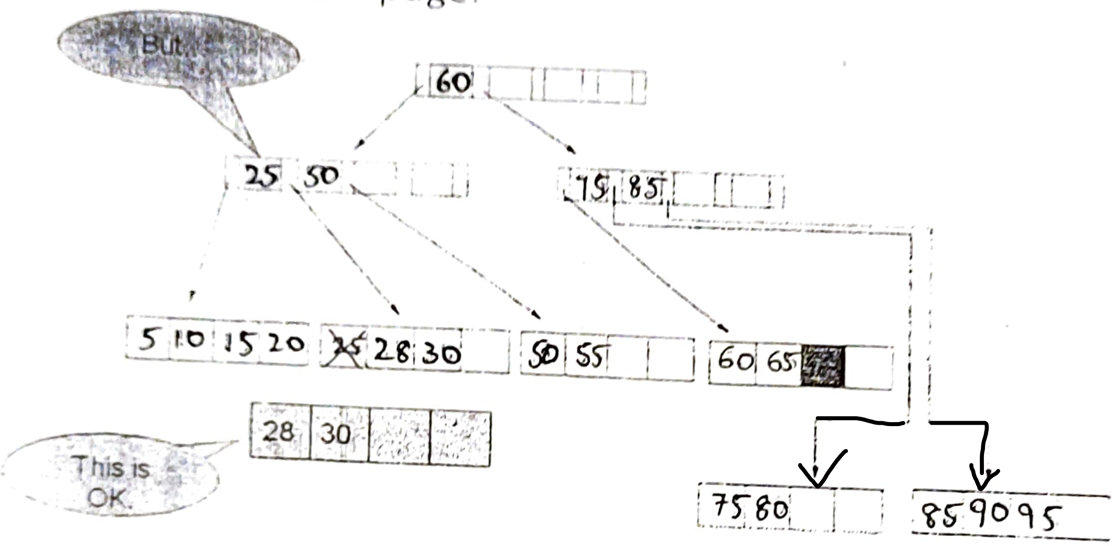
Deletion

Result



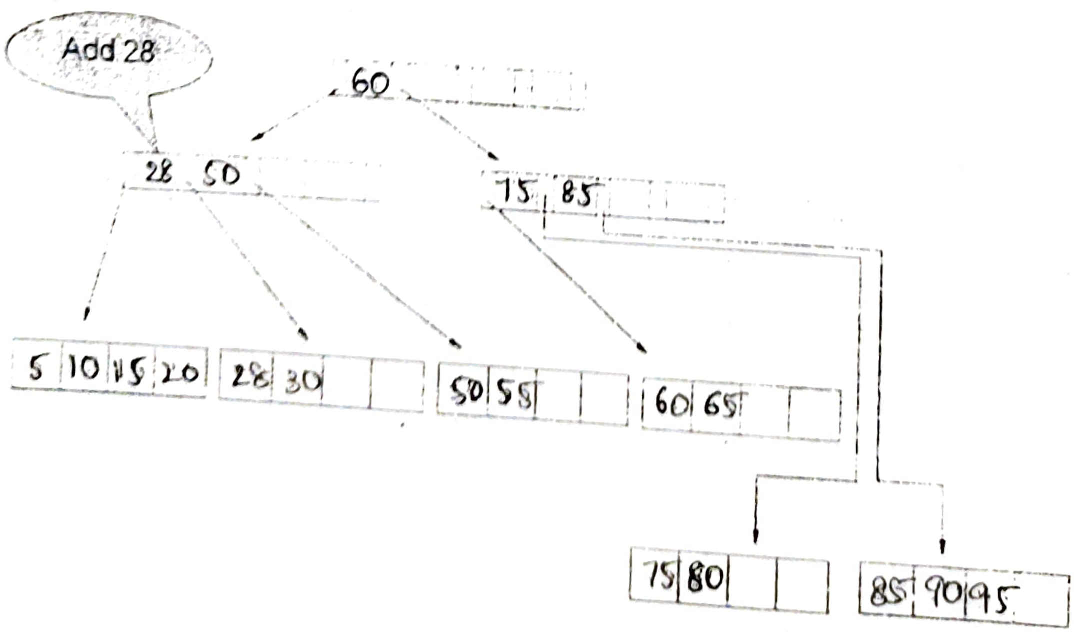
Deletion

Example #2: delete 25 from below tree, but 25 appears in the index page.



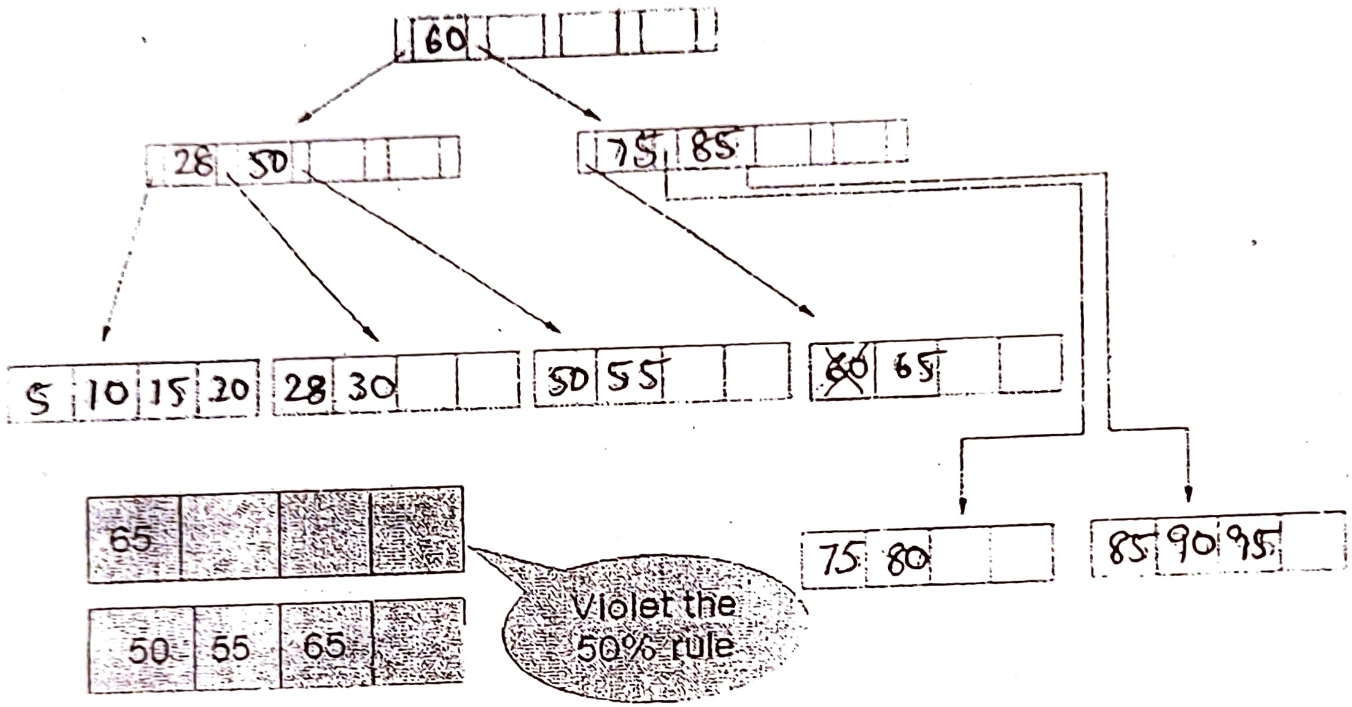
Deletion

- Result: replace 28 in the index page.



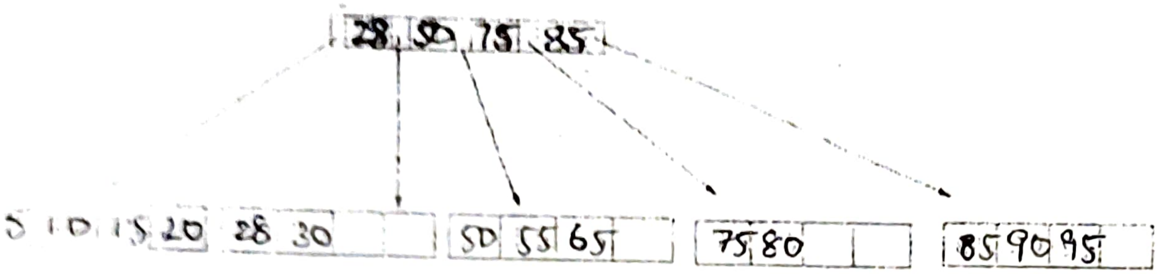
Deletion

- Example #3: delete 60 from the below tree



Deletion

ult. delete 60 from the index page and combine the rest of index pages.



Deletion

e algorithm for B+ trees

ge Below Fill Factor	Index Page Below Fill Factor	Action
	NO	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page use the next key to replace it.
	NO	Combine the leaf page and its sibling. Change the index page to reflect the change
	YES	Combine the leaf page and its sibling Adjust the index page to reflect the change. Combine the index page with its sibling Continue combining index pages until you reach a page with the correct fill factor or you reach the root page