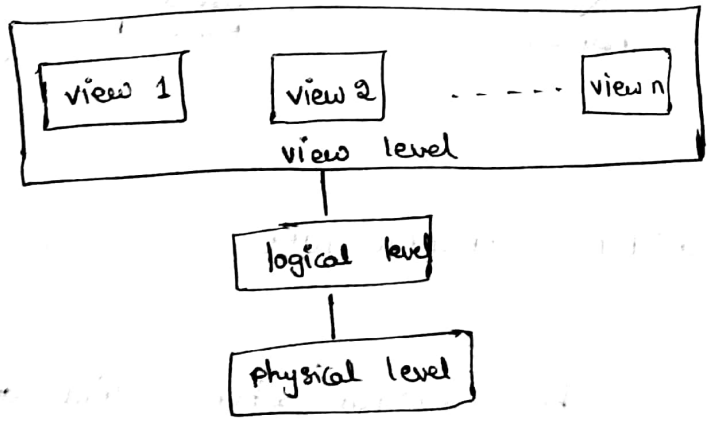
- A database is collection of data, typically describing the activities of one or more related data
- DBMS (Database Management System) is software designed to assist in maintaining and utilizing large collections of data.
- Data Redundancy – Multiple file formats, duplication of info

*. Applications of DBMS :

1. Banking & Financial services
2. Transport, Tourism
3. Online Booking systems
4. Inventory & e-commerce, etc...

*. View of Data :

An architecture for database system




Physical : how the data are actually stored.

Logical : what relationship exists.

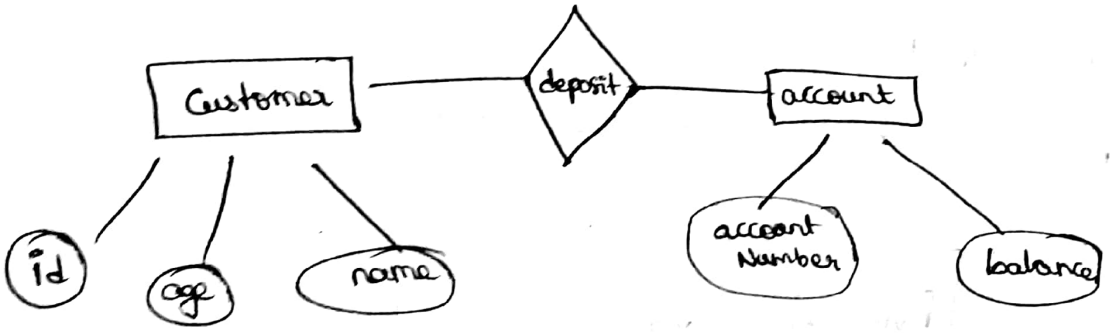
view : only view part of entire database.

* · Schema – the logical structure of database

* · Instance – the actual content of database at a particular point in time.

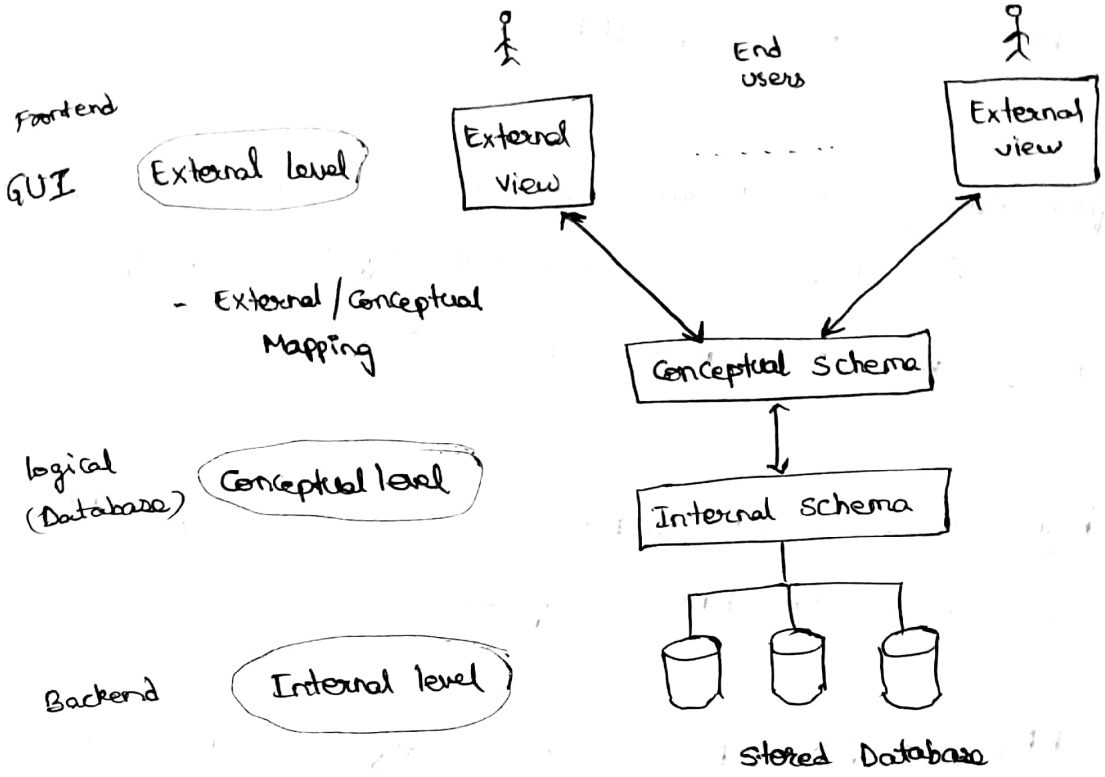
## * · RELATIONAL DATABASE :

- A relational dbase is collection of related data.
- Relation – A table is called relation
- In a table → rows + Columns
  ||  ||  ||
  Relation → Tuples + Attributes
- ER Model – Entity relationship Model
  L, logical schema of a database (pictorial / chart)




## * · THREE - SCHEMA ARCHITECTURE : (applicable to all DB)

- Internal level – has "internal schema that describes the physical storage, structure of the database
- Conceptual level – describe the every detailed structure of DB (how, what type, many, which entities)
  (entities, data-types, relationships, constraints)




- External level - describes the part of the database that a particular user group (a user view)

*. Data Independance :

The capacity to change the schema at one level of database system without changing the schema at the next higher level.

- Logical data independance - Capacity to change conceptual schema with change in external schema
- Physical data independance - Capacity to change the internal schema without having change in Conceptual schema.

- Physical data independance - (file environments) location, hardware, compression, merging, are hidden from user
- logical data independance is harder to achieve because it allows structural & constraint changes.

## * . Component modules of DBMS :

- Buffer management module to schedule disk read/write, because this has a considerable effect on performance. Reducing disk read/write improves performance.

- A higher-level stored data manager module of DBMS controls access to DBMS information, stored on disk.

- The DDL compiler process schema defines, specified in DDL and stored descriptions of schemas (meta-data) in the DBMS catalog.

- Interactive query interface is to provide the interface between the user and DBMS for basic information

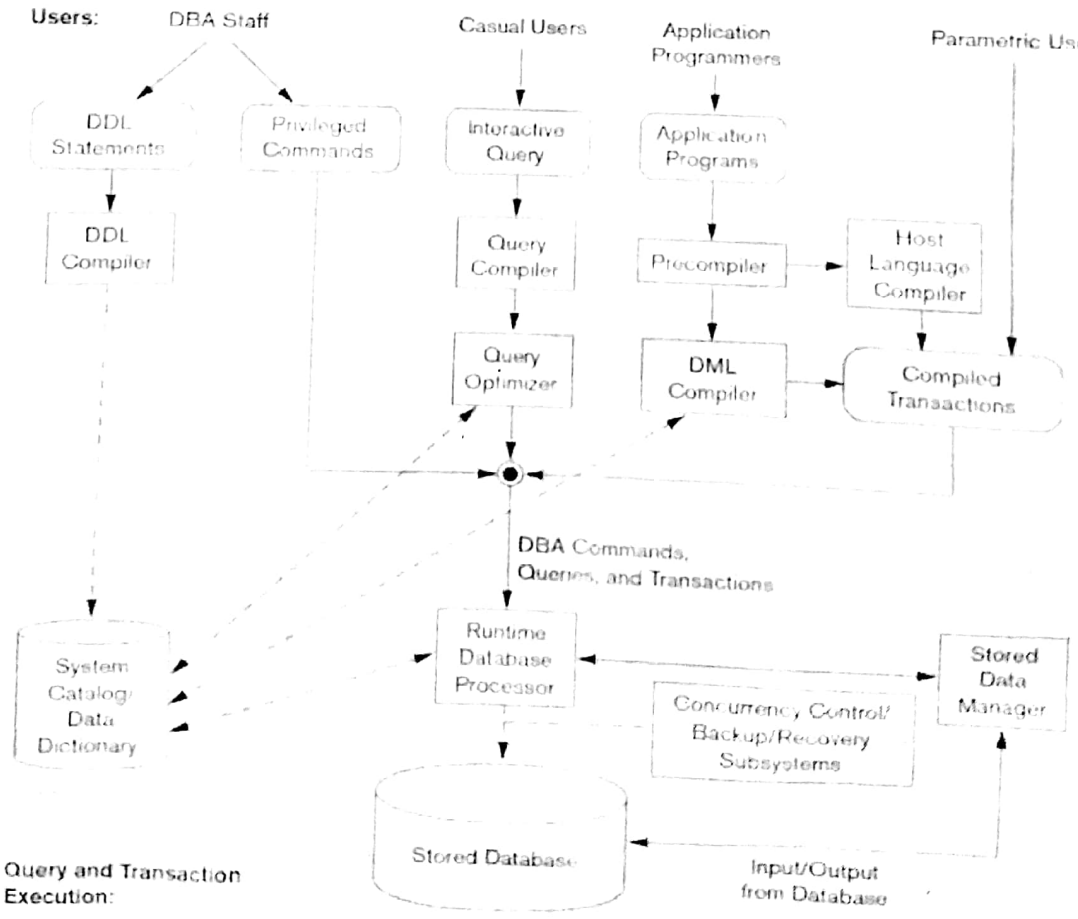
- Query compiler used to parse and validate the correctness of the query selector/syntax, file names.

- Query optimizer is concerned with the rearrangement and possible reordering of operations, elimination of redundancies and use of correct algorithms and indexes during execution.

- The precompiler extracts DML commands from an application program written in a host programming language

- DML compiler is to compile DML commands into object code for database access.

- The runtime database processor executes :
  i. the privileged commands,
  ii - the executable query plans &
  iii. the canned transactions with run time parameters.
- It works with the system catalog and may update it with statistics.
- It also works with stored data manager, which in turn uses basic operating systems services for carrying out low level input / output.
- Concurrency control and Backup & recovery are integrated into the working of the runtime database processor for purposes of transaction management.



## *. Attributes :

1. Simple Attribute : single component + independent existence

2. Compesite Attribute : multiple Components + each is independent

   → Name – First Name, last Name

   → Address – Door No, street No, Colony, Area, ..

3. Single valued Attribute : A single valued attributed is one that holds single value for single entity

   → Room No, Customer id

4. Multi valued Attribute : multiple value + single entity

   → Hobby ? : reading, playing

5. Derived Attribute : derived from related attribute / set of attributes

   → age : from DOB.

## *. RELATIONSHIP & RELATIONSHIP SETS :

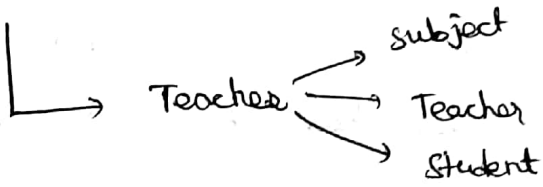
- A relationship expres an association among several entities.
- A relationship set is a set of relationship of same type.
- Entity Role : The function that an entity plays in a relationship is called entity's role.
- Descriptive attributes : Attribute of relationship.

## *. Types of Relationship :

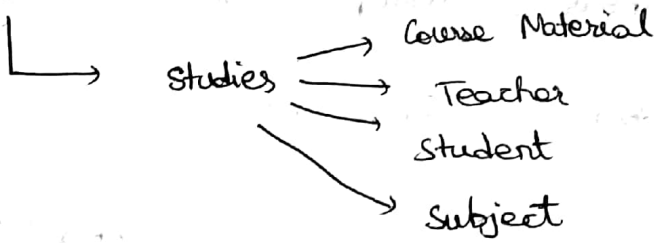
1. Unary Relationship : association within a single entity


2. Binary : 2 entities

3. Ternary relationship : association within 3 entities




4. Quaternary relationship : association with 4 entities

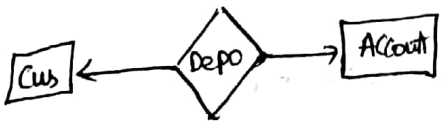



IMP
## * CONSTRAINTS :

An E-R enterprise schema define certain constraints to which the content of database system must confirm & accept.

### I: Mapping Cardinality Types :

① one to one




② one to Many


→ one

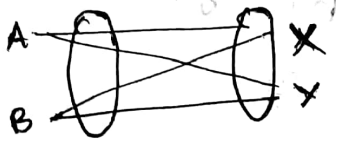
— many

③ Many to one


④ Many to Many

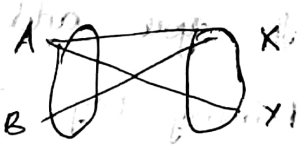

II. Participation Constraints :

1. Total Participation :




every connection

2. Partial Participation :




partial / incomplete

*. KEYS : A Key is attribute or set of Attribute which uniquely identifies a tuple within the relation
row within a table

*. Types of Keys :

① Primary Key :

It is the columns you choose to maintain uniqueness in a table. Here in Employee Table you can choose either EMP_ID or EMP_SSN columns, EMP_ID is preferable choice as EMP_SSN is secure value.

② Candidate Key :

It is individual columns in a table that qualifies for uniqueness of all the rows. Here in Employee Table EMP_ID & EMP_SSN are candidate keys.

③ Alternate Key :

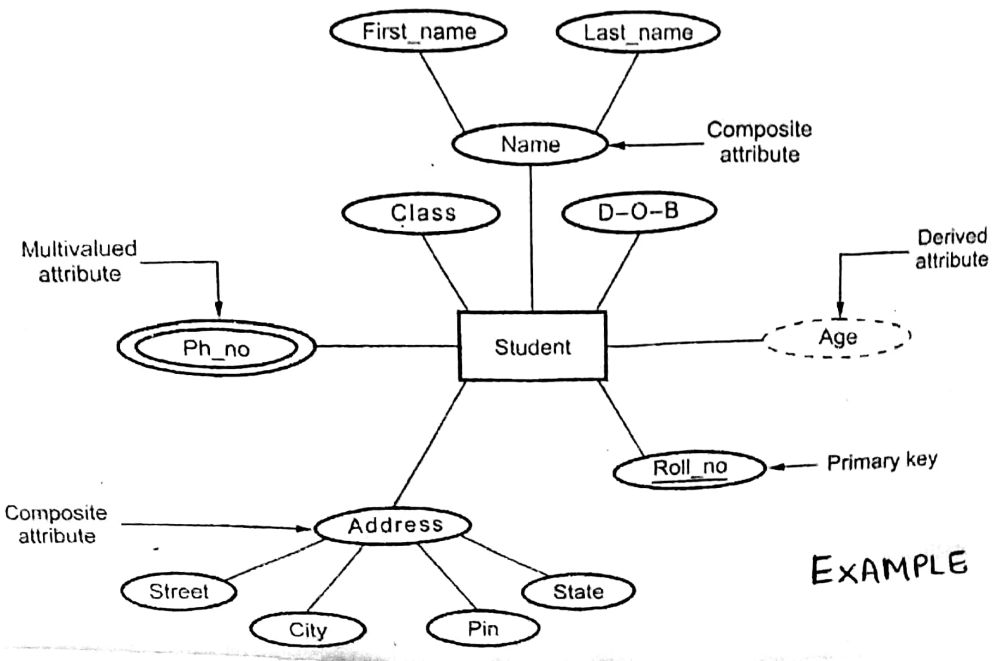
Candidate column other the Primary Key column, like if EMP_ID is Primary Key then EMP_SSN would be Alternate Key.

④ Super Key :

If you add any other column / attribute to a Primary Key then it becomes a super Key. Here, EMP_ID + EMP_NAME is a super key.

⑤ Composite Key :

If a table do have a single key columns that qualifies for a Candidate Key, then you have to select 2 or more columns to make a row unique. Like if there is no EMP_ID or EMP_SSN columns, then you can make EMP_NAME + EMP_DOB as composite primary key But still there may be narrow chance of duplicate row.



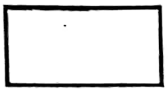
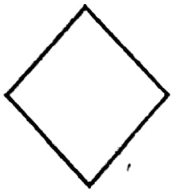
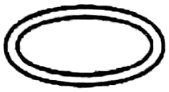
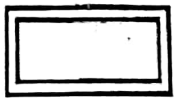
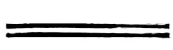

EXAMPLE

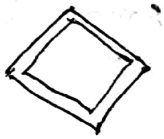
# *. E-R DIAGRAM :

- **Dependency** : If the existence of entity X depends on the existence of entity Y, then X is said to be existence dependent on Y.

→ **Strong entity** : entity set on which weak entity depends.

→ **weak entity** : existence dependency on some other entity.

| Component name | Symbol | Description |
|---|---|---|
| 1) Rectangles |  | Represents entity sets |
| 2) Ellipses |  | Represents attributes |
| 3) Diamonds |  | Represents relationship sets |
| 4) Lines |  | Links attributes to entity sets & entity sets to relationship sets |
| 5) Double ellipses |  | Represents multivalued attributes |
| 6) Dashed ellipses |  | Represents derived attributes |
| 7) Double rectangles |  | Represents weak entity sets |
| 8) Double lines |  | Represents total participation of an entity in a relationship set |

9) Double Diamonds  Represents weak relationship sets

## *. RELATIONAL DATABASE CONCEPTS :

- Table - set of rows that have same attributes.
- Relations that store data are called "Base Relations" and in implementation are called "Tables".
- A Tuple / record / row holds all the Information about one item or object.
- A Field / Column holds one piece of information about an item or object. (Attribute)
- A Domain describes the set of possible values for a given attribute.
- Foreign Key is a Key used to link two tables together.
- A stored Procedure is a high end tool that adds programming capability into database.

imp
- A stored procedure is executable group of queries that is associated with and generally stored in database.
- An index is one way of providing quicker access to a data

## *. RELATIONAL OPERATIONS :

1. Union - $A \cup B$
2. Intersection - $A \cap B$

} same columns

3. Difference - $A \cap \overline{B}$
4. Cartesian Product $A \times B$

5. Selection – Combination – all rows from relation

6. Projection – Permutation – all rows from relation without duplicate.

7. Join – Connected by common attributes.

8. Relation division – opposite of Cartesian product operator.

9. Normalization – Splitting of tables to avoid duplicity.

## *. CHARACTERISTICS OF RELATIONAL MODEL :

- It eliminates all parent and child relationships and represented the data as a table.
- Relational model table consists of rows & columns.
- Each table is an individual entity and there is no physical relationship between tables.
- All relational databases supports query language like SQL
- Relational model of data is based on set theory and the user interface with the relational models is non-procedural.
- Cardinality – No. of tuples (rows) in a relation (table).
- Degree – No. of attributes (column) in a relation (table).

## *. DATA INTEGRITY :

Integrity constraints provide a means of ensuring that changes made to the database by authorized for ensuring data consistency.

The types of integrity constraints are :

→ Domain constraints

→ Referential integrity

→ NULLs

→ Entity integrity

→ Enterpise constraints

## ① Domain Constraints :

- Domain Constraints specifies the set of values that can be associated with an attribute.
- These are the most elementary form of the integrity constraint.
- They are easily tested by the system whenever a new data item is entered into database.
- Constraints also prohibits the use of null values for particular fields.

Example : varchar(10) ✓ , Number(5,3) ✓

## ② Referential integrity :

Referential Integrity can be defined as value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.

```
CREATE TABLE deposit (
        branch_name varchar2(20),
        acc_no number(20) not null,
        cust_name varchar2(20),
        balance number(5),
        PRIMARY KEY (acc_no),
        FOREIGN KEY (branch_name) REFERENCES branch,
③       FOREIGN KEY (cust_name) REFERENCES customer);
```

**Primary Table**

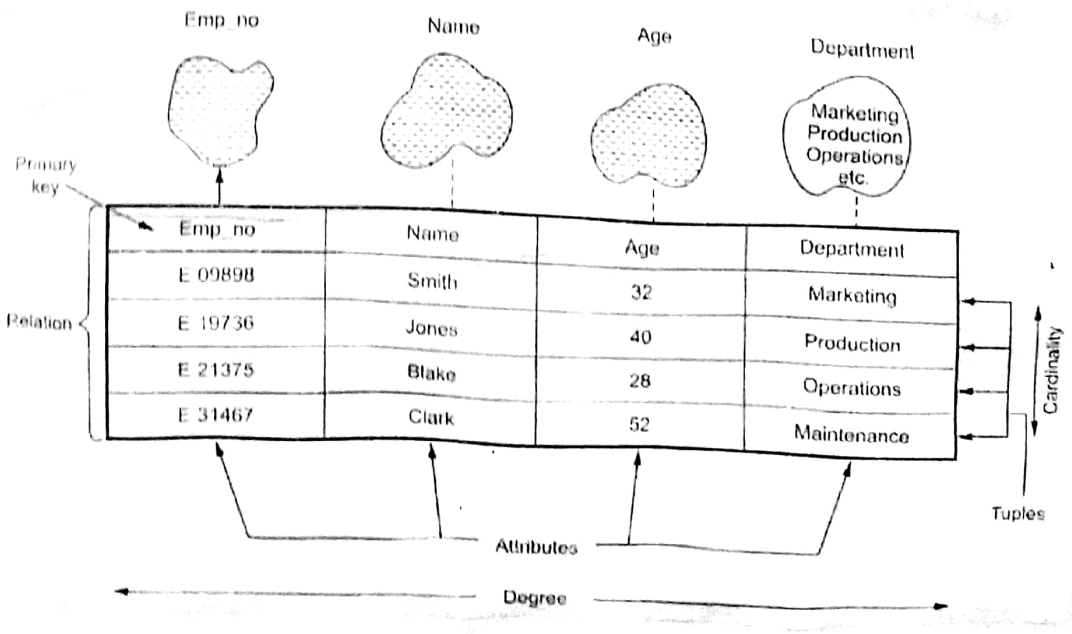
| CompanyId | CompanyName |
|---|---|
| 1 | Apple |
| 2 | Samsung |

**Related Table**

| CompanyId | ProductId | ProductName |
|---|---|---|
| 1 | 1 | iPhone |
| 15 | 2 | Mustang |

Orphaned Record ✗

✱. RELATIONAL STRUCTURE :



## ③ NULL CONSTRAINTS :

- Nulls represent a value for an attribute that is currently unknown or is not applicable for this tuple.

NULLs are a way to deal with incomplete or exceptional data.

## ④ ENTITY CONSTRAINTS :

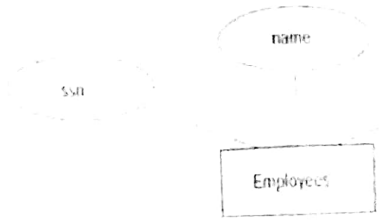
- In a base relation, no attributes of primary key can be null.

## ⑤ ENTERPRISE CONSTRAINT :

- The additional rules or constraints specified by the database administrators are known as enterprise constraints.

Ex : No. of students in class should be 65.

*. Relational model describes design of high-level database with in-depth logical design with its constraints.




| ssn | name | lot |
|---|---|---|
| 123-22-3666 | Attishoo | 48 |
| 231-31-5368 | Smiley | 22 |
| 131-24-3650 | Smethurst | 35 |

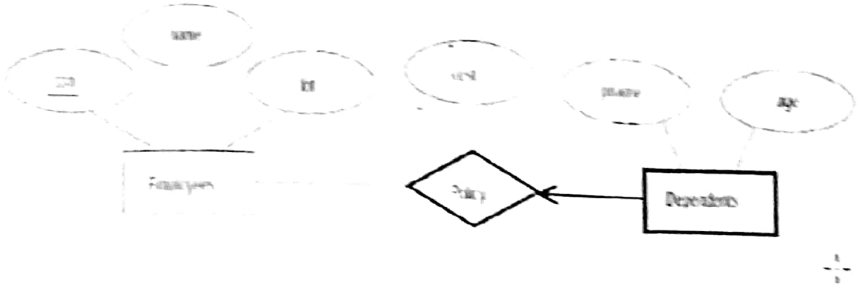
```
CREATE TABLE Employees ( ssn CHAR(11),
                         name CHAR(30),
                         lot NUMBER(3),
                         PRIMARY KEY (ssn) )
```

## Translating Relationship Sets with Participation Constraints

## Translating Relationship Sets with Participation Constraints

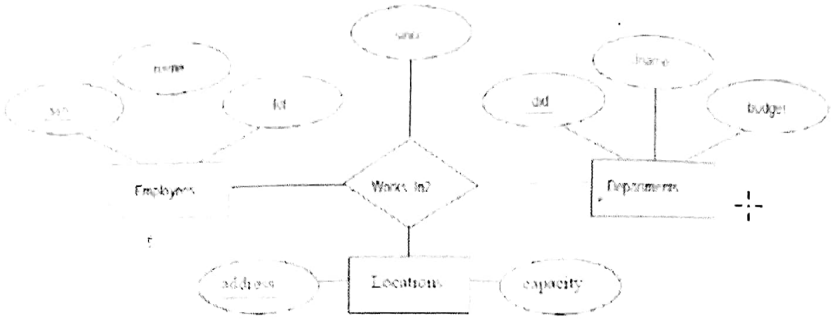
```
CREATE TABLE Dept Mgr ( did NUMBER,
                        dname VARCHAR2(20),
                        budget NUMBER(5,2),
                        ssn VARCHAR2(11) NOT NULL,
                        since DATE,
                        PRIMARY KEY (did),
                        FOREIGN KEY (ssn) REFERENCES Employees
                              ON DELETE NO ACTION )
```

## Translating Weak Entity Sets



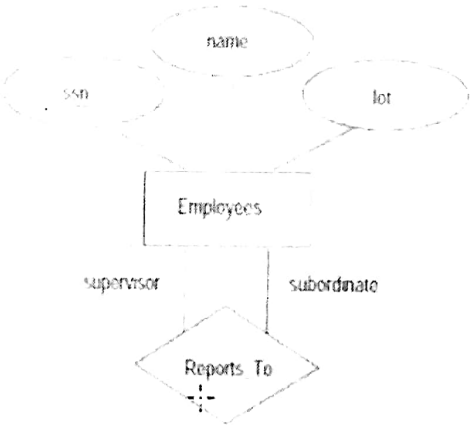
```
CREATE TABLE Dep Policy ( pname VARCHAR2(20), age NUMBER, cost NUMBER(5,2),
                          ssn CHAR(11),
                          PRIMARY KEY (pname, ssn),
                          FOREIGN KEY (ssn) REFERENCES Employees
                          ON DELETE CASCADE )
```

# Relationship Sets (without Constraints) to Tables



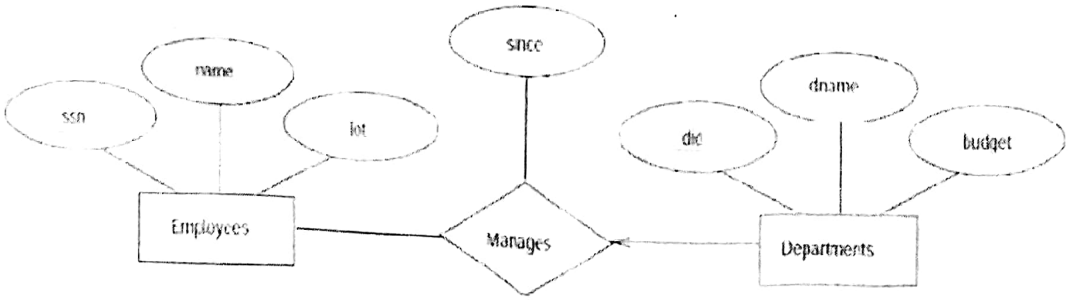

CREATE TABLE Works_In2 ( ssn CHAR(11), did NUMBER, address CHAR(20), since DATE,
PRIMARY KEY (ssn, did, address),
FOREIGN KEY (ssn) REFERENCES Employees,
FOREIGN KEY (address) REFERENCES Locations,
FOREIGN KEY (did) REFERENCES Departments )

# Relationship Sets (without Constraints) to Tables




CREATE TABLE Reports_To (
Supervisor_ssn CHAR(11),
Subordinate_ssn CHAR(11),
PRIMARY KEY (supervisor_ssn, subordinate ssn),
FOREIGN KEY (supervisor_ssn) REFERENCES Employees(ssn),
FOREIGN KEY (subordinate ssn) REFERENCES Employees(ssn) )

# Translating Relationship Sets with Key Constraints




CREATE TABLE Manages ( ssn CHAR(11), did INTEGER, since DATE,
PRIMARY KEY (did),
FOREIGN KEY (ssn) REFERENCES Employees,
FOREIGN KEY (did) REFERENCES Departments )

# MODULE - 3 :

## *. RELATIONAL QUERY LANGUAGES :

- A query language is a language in which a user requests information from the database. It can be categorized as either procedural or non-procedural.
- Procedural Language : The user instructs the system to perform a sequence of operations on the database to compute the desired result. Ex : Relational Algebra.
- Non-Procedural Language : The user describes the desired information without giving a specific procedure for obtaining the information. Ex : Tuple relational calculus.

## *. RELATIONAL ALGEBRA :

- An algebra, in general, consists of operators and atomic operands.
- For instance, in the algebra of arithmetic :
  - → The atomic operands are variables like x & constant 15.
  - → The operators are : +, −, *, /
  - → For instance : arithmetic expression (x + y) * z ...
- Relational Algebra is another example of an algebra. Its atomic operands are :
  1. Variables that stand for relations.
  2. Constants, which are finite relations. ($\cup, \cap, -$)

## *. RELATIONAL ALGEBRA OPERATIONS :

1. Set operations - union, Intersection, difference
2. Removes parts - selection - eliminate rows
   projection - eliminate columns
3. Combine parts - cartesian product : joins
4. Relation schema change - renaming

## *. Conditions for set operations :

- When we apply these operations to relations, we need to put some conditions on R & S.

→ R and S must have schemas with identical sets of attributes, and the types (domains) for each attribute must be same in R & S.

→ Before we compute the set-theoretic union, intersection or difference of sets of tuples, the columns of R and S must be ordered so that the order of attributes is the same for both relations.

## * R U S Example

| name | address | gender | birthdate |
|---|---|---|---|
| Carrie Fisher | 123 Maple St., Hollywood | F | 9/9/99 |
| Mark Hamill | 456 Oak Rd., Brentwood | M | 8/8/88 |

Relation $R$ ↗

| name | address | gender | birthdate |
|---|---|---|---|
| Carrie Fisher | 123 Maple St., Hollywood | F | 9/9/99 |
| Harrison Ford | 789 Palm Dr., Beverly Hills | M | 7/7/77 |

Relation $S$ ↗

| name | address | gender | birthdate |
|---|---|---|---|
| Carrie Fisher | 123 Maple St., Hollywood | F | 9/9/99 |
| Mark Hamill | 456 Oak Rd., Brentwood | M | 8/8/88 |
| Harrison Ford | 789 Palm Dr., Beverly Hills | M | 7/7/77 |

Relation RUS ↗

## * R ∩ S Example

| name | address | gender | birthdate |
|---|---|---|---|
| Carrie Fisher | 123 Maple St., Hollywood | F | 9/9/99 |
| Mark Hamill | 456 Oak Rd., Brentwood | M | 8/8/88 |

Relation $R$ ↗

| name | address | gender | birthdate |
|---|---|---|---|
| Carrie Fisher | 123 Maple St., Hollywood | F | 9/9/99 |
| Harrison Ford | 789 Palm Dr., Beverly Hills | M | 7/7/77 |

Relation $S$ ↗

| name | address | gender | birthdate |
|---|---|---|---|
| Carrie Fisher | 123 Maple St., Hollywood | F | 9/9/99 |

## * R − S Example

| name | address | gender | birthdate |
|---|---|---|---|
| Carrie Fisher | 123 Maple St., Hollywood | F | 9/9/99 |
| Mark Hamill | 456 Oak Rd., Brentwood | M | 8/8/88 |

Relation $R$ ↗

| name | address | gender | birthdate |
|---|---|---|---|
| Carrie Fisher | 123 Maple St., Hollywood | F | 9/9/99 |
| Harrison Ford | 789 Palm Dr., Beverly Hills | M | 7/7/77 |

Relation $S$ ↗

| name | address | gender | birthdate |
|---|---|---|---|
| Mark Hamill | 456 Oak Rd., Brentwood | M | 8/8/88 |

Relation (R-S)

## ✱. OPERATIONS THAT REMOVE PARTS OF :

### The relation Movies

| *title* | *year* | *length* | *genre* | *studioName* | *producerC#* |
|---|---|---|---|---|---|
| Star Wars | 1977 | 124 | sciFi | Fox | 12345 |
| Galaxy Quest | 1999 | 104 | comedy | DreamWorks | 67890 |
| Wayne's World | 1992 | 95 | comedy | Paramount | 99999 |

✱. Projection Example :

$\pi_{(title,year,length)}(\boldsymbol{Movies})$

The resulting relation is

| *title* | *year* | *length* |
|---|---|---|
| Star Wars | 1977 | 124 |
| Galaxy Quest | 1999 | 104 |
| Wayne's World | 1992 | 95 |

$\pi_{(genre)}(\boldsymbol{Movies})$

The resulting relation is

| *genre* |
|---|
| sciFi |
| comedy |

✱. Selection Example :

$\sigma_{(length \geq 100)}(\boldsymbol{Movies})$ - The resulting relation is

| *title* | *year* | *length* | *genre* | *studioName* | *producerC#* |
|---|---|---|---|---|---|
| Star Wars | 1977 | 124 | sciFi | Fox | 12345 |
| Galaxy Quest | 1999 | 104 | comedy | DreamWorks | 67890 |

$\sigma_{(length \geq 100\ AND\ studioName = "Fox")}(\boldsymbol{Movies})$ - The resulting relation is

| *title* | *year* | *length* | *genre* | *studioName* | *producerC#* |
|---|---|---|---|---|---|
| Star Wars | 1977 | 124 | sciFi | Fox | 12345 |

✱. CARTESIAN PRODUCT Example :

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

(a) Relation $R$

| *B* | *C* | *D* |
|---|---|---|
| 2 | 5 | 6 |
| 4 | 7 | 8 |
| 9 | 10 | 11 |

(b) Relation $S$

| *A* | *R.B* | *S.B* | *C* | *D* |
|---|---|---|---|---|
| 1 | 2 | 2 | 5 | 6 |
| 1 | 2 | 4 | 7 | 8 |
| 1 | 2 | 9 | 10 | 11 |
| 3 | 4 | 2 | 5 | 6 |
| 3 | 4 | 4 | 7 | 8 |
| 3 | 4 | 9 | 10 | 11 |

(c) Result $R \times S$

## *. WHAT IS JOIN ?

- Join is a binary operation which allows you to combine join product and selection in one single statement.
- The JOIN operation, denoted by $\bowtie$, is used to combine related tuples from two relations into single "longer tuples".
- The goal of creating a join condition is that it helps you to combine the data from multiple join tables.
- SQL joins allows you to retrieve data from two or more DBMS table.
- The tables in DBMS are associated using the primary key and foreign keys.

Ex: $DEPT\_MGR \leftarrow DEPARTMENT \bowtie_{Mgrssn = ssn} EMPLOYEE$

$RESULT \leftarrow \pi_{(DNAME, LNAME, FNAME)} (DEPT\_MGR)$.

## *. INNER JOIN :

- An Inner Join is the widely used join operation and can be considered as a default JOIN Type. The inner JOIN is used to return rows from both tables which satisfy the given condition.
- A inner join or equi join is a comparator - based join which uses equality comparisions in the join-predicate.

- Inner Join further divided into three sub types :

1. THETA JOIN :

→ Theta Join allows you to merge two tables based on the condition represented by theta. Theta joins work for all comparision operators. ($<$, $>$, $\leq$, $\geq$)

→ The general case of JOIN operation is called Theta Join. It is denoted by symbol $\theta$.

→ Syntax : $A \bowtie_\theta B$.

2. EQUI JOIN :

→ When Theta join uses only equality comparision operator it is said to be equijoin.

→ Syntax : $A \bowtie B$. (relational operator, $=$, $!=$)

3. NATURAL JOIN :

- This is same as Cartesian product of two relations.
- One common ~~tuple~~ column must exist in both tables.
- Natural join doesnot utilize any comparision operators.
- It performs selection forming equality on those attributes which appear in both relations and eliminates the duplicate attributes.

Consider two relations

Car $\dagger$ Bike

| CName | CPrice |
|---|---|
| Swift | 20000 |
| City | 30000 |
| Verna | 50000 |

| BName | BPrice |
|---|---|
| Apache | 10000 |
| Shine | 40000 |
| Xtreme | 60000 |

Car $\bowtie_{(Car.CPrice = Bike.BPrice)}$ Bike

| CName | CPrice | BName | BPrice |
|---|---|---|---|
| Swift | 20000 | Apache | 10000 |
| City | 30000 | Apache | 10000 |
| Verna | 50000 | Apache | 10000 |
| Verna | 50000 | Shine | 40000 |

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

| Student | | |
|---|---|---|
| SID | Name | Std |
| 101 | Alex | 10 |
| 102 | Maria | 11 |

| Subjects | |
|---|---|
| Class | Subject |
| 10 | Math |
| 10 | English |
| 11 | Music |
| 11 | Sports |

Student_detail

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

C

| Num | Square |
|---|---|
| 2 | 4 |
| 3 | 9 |

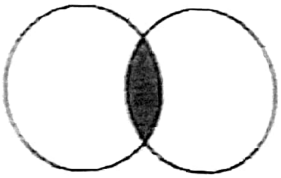
D

| Num | Cube |
|---|---|
| 2 | 8 |
| 3 | 18 |

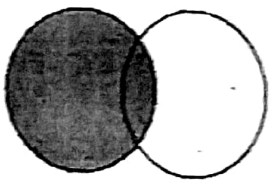
C $\bowtie$ D $\dagger$

| Num | Square | Cube |
|---|---|---|
| 2 | 4 | 8 |
| 3 | 9 | 18 |

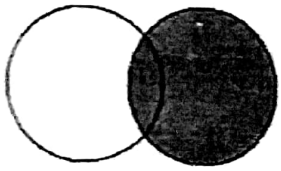
JOINS AND SET OPERATIONS IN RELATIONAL DATABASES



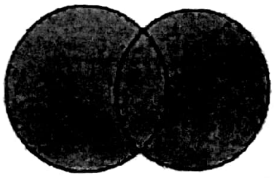
Inner join (result similar to Intersect)



Left outer join



Right outer join



Full outer join

## *. OUTER JOIN :

- An outer join doesn't require each record in the two join tables to have a matching record.
- In this type of join, the table retains each record even if no other matching record exists.
- There three types of outer joins.

### 1. Left outer join : (A ⟕ B)

→ The LEFT JOIN returns all the rows from the table on the left even if no matching rows have been found in the table on the right.

→ Where no matching record found in the table on the right, NULL is retained, returned.

### 2. Right outer join : (A ⟖ B)

→ RIGHT outer join is the opposite of left outer join.

→ The right outer join returns all the columns from the table on the right even if no matching rows have been found in the table on the left.

→ Where no matches have been founded in the table on the left, NULL is returned.

### 3. FULL outer join :

Normally merging all the tuples from both the tables.

## Left Outer Join (A ⟕ B)

| Course | |
|---|---|
| A | B |
| 100 | Database |
| 101 | Mechanics |
| 102 | Electronics |

| Chair | |
|---|---|
| A | B |
| 100 | Alex |
| 102 | Maya |
| 104 | Mira |

Course ⟕ Chair

| Courses ⟕ Chair | | | |
|---|---|---|---|
| A | B | C | D |
| 100 | Database | 100 | Alex |
| 101 | Mechanics | --- | --- |
| 102 | Electronics | 102 | Maya |

## Right Outer Join (A ⟖ B)

| Course | |
|---|---|
| A | B |
| 100 | Database |
| 101 | Mechanics |
| 102 | Electronics |

| Chair | |
|---|---|
| A | B |
| 100 | Alex |
| 102 | Maya |
| 104 | Mira |

Course ⟖ Chair

| Courses ⟖ Chair | | | |
|---|---|---|---|
| A | B | C | D |
| 100 | Database | 100 | Alex |
| 102 | Electronics | 102 | Maya |
| --- | --- | 104 | Mira |

## Full Outer Join (A ⟗ B)

| Courses ⟗ Chair | | | |
|---|---|---|---|
| A | B | C | D |
| 100 | Database | 100 | Alex |
| 101 | Mechanics | --- | --- |
| 102 | Electronics | 102 | Maya |
| --- | --- | 104 | Mira |

# MODULE - 3 : SQL

## *. What is SQL ?

- SQL is Structured Query language, which is a Computer language for storing, manipulating and retrieving data stored in relational database.
- SQL is the standard language for Relational Database System. All the Relational Database Management System (RDMS) like

Ex :- MySQL, MS Access, Oracle, Sybase, Informix, Postgres, SQL Server.

use SQL as their standard database language.

## *. Applications of SQL ?

- Allows Users to access data in the relational database management systems.
- Allow User to describe the data.
- Allow User to define the data in database & manipulate that data.
- Allows to embed within other languages using SQL

modules, libraries & pre-compilers.

- Allows users to create and drop databases & tables.
- Allows users to create views, stored procedure, functions in a database.
- Allows users to set permissions on tables, procedures & views.

## * SQL COMMANDS :

### ① DDL - Data Definition language :

→ Many DB Administrators will use this type of language.
→ This will create physical schema of the logical schema
→ CREATE, ALTER, DROP, TRUNCATE, RENAME
→ It modies Columns (attributes of Relation) of Table.

### ② DML - Data Manipulation language :

→ This is used by any user of DBMS.
→ It will perform operations only on Rows. (Tuples of Relation).
→ SELECT, INSERT, UPDATE, DELETE

### ③ DCL - Data Control language :

→ It is used by all DB Admins.
→ Creating new users, authentication, control permissions.
→ GRANT, REVOKE.

### ④ TCL - Transaction Control language :

→ To ensure Consistency of the DB, TCL plays major role.
→ To maintain valid Information.
→ COMMIT, ROLLBACK, SAVEPOINT.

## *- SQL CONSTRAINTS :

- NOT NULL Constraint - Ensures that a column cannot have a NULL value.

Ex:
```
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)
);
```

```
ALTER TABLE customers MODIFY salary DECIMAL(18,2)
NOT NULL;
```

- interactive insert Query :

```
> INSERT INTO customers VALUES (&id, '&name', &age,
    '&address', &salary);
```

```
Enter value for id : 125
Enter value for name: name1
Enter value for age: 22
Enter value for address :
Enter value for salary : 3000
old 1 : INSERT INTO CUSTOMERS VALUES (&id, '&name', &age, ...);
new 1 : INSERT INTO customers VALUES(125, 'name1', 22, '', 3000);
> 1 row created.
```

- DEFAULT Constraint - Provides a default value for a column when none is specified.

```
> CREATE TABLE customers (
    id INT NOT NULL,
    name VARCHAR(20) NOT NULL,
    age INT NOT NULL,
    address CHAR(25),
    salary DECIMAL (18,2)
    DEFAULT 5000.00 ,
    PRIMARY KEY (id)
  );
```

```
> ALTER TABLE CUSTOMERS MODIFY
  Salary DECIMAL (18,2) DEFAULT 5000.00 ;
```

```
> ALTER TABLE customers MODIFY age DEFAULT NULL;
```

- UNIQUE Constraint - Ensures that all the values in a column are different.

```
> CREATE TABLE Customers (
    id INT NOT NULL,
    name VARCHAR(20) NOT NULL,
    age INT NOT NULL UNIQUE,
    address CHAR(25),
    Salary DECIMAL (18,2)
    DEFAULT 5000.00,
    PRIMARY KEY (id)
  );
```

> ALTER TABLE customers MODIFY age INT UNIQUE;

> ALTER TABLE customers ADD UNIQUE (age);

> ALTER TABLE customers ADD CONSTRAINT myUniqueConstraint
UNIQUE (age, salary);

> ALTER TABLE customers DROP CONSTRAINT myUniqueConstraint

- PRIMARY Key - Uniquely identifies each row | record in a database table.

Ex: > 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)
);

> ALTER TABLE customers ADD PRIMARY KEY (id);

> ALTER TABLE customers DROP PRIMARY KEY;

- FOREIGN KEY - Uniquely identifies a row | record in any another database Table

> ALTER TABLE orders ADD CONSTRAINT myconst
FOREIGN KEY (id) REFERENCES customers;

```
> 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)
  );

> CREATE TABLE orders (
    ord_id INT NOT NULL,
    ord_date DATE,
    id INT REFERENCES customers,
    amount DECIMAL(18,2),
    PRIMARY KEY(ord_id)
  );

> ALTER TABLE orders DROP CONSTRAINT myCons;
```

- CHECK Constraint - The CHECK Constraint ensures that all values in a column satisfy certain conditions.

```
> CREATE TABLE customers (
    id INT NOT NULL,
    name VARCHAR(20) NOT NULL,
    age INT NOT NULL CHECK (age >= 18),
    address CHAR(25),
    salary DECIMAL(18,2) DEFAULT 5000.00,
    PRIMARY KEY(id)
);
```

> ALTER TABLE customers MODIFY age INT
CHECK (age >= 18);

> ALTER TABLE customers ADD CONSTRAINT myUniqueConstraint
CHECK (age >= 18);

> ALTER TABLE customers DROP CONSTRAINT myUniqueConstraint;

• INDEX – Used to create and retrieve data FROM the database very quickly.

> 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)
);

> CREATE INDEX idx_age ON customers(age);

> DROP INDEX idx_age;

*. SELECT STATEMENT :

• Purpose of SELECT COMMAND :

- SELECT : is a list of one or more columns.
- * : Selects all columns
- DISTINCT : supresses duplicates

- COLUMN EXPRESSION : Expression selects named column
- ALIAS : gives the selected columns of different headings
- FROM TABLE : specifies the table containing the column.

Ex:

```
SELECT * FROM employee ;
SELECT eid , ename . FROM employee ;
SELECT eid, ename AS employee FROM employee ;
SELECT ename || dept_id AS name FROM employee ;
SELECT distinct dept_id FROM employee ;
SELECT ename , salary , Salary+3000 AS updated_salary FROM employee;
SELECT ename , salary , 12 * (salary + 100) AS updated FROM employee;
SELECT * FROM employee where salary ≤ 5000 ;
```

*. DELETE STATEMENT :

- Deletes one or more records.
- Delete cannot delete column from table. It deletes only rows.
- To delete a column from table, ALTER table must be used.

Syntax :

```
DELETE FROM vasaprasad [ where condition ] ;
```

- Delete all rows of table customer_details :

```
DELETE FROM customer_details ;
```

- Deleting some rows of table customer_details;

```
DELETE FROM customer_details WHERE cust_id = 102 ;
```

* Difference Between TRUNCATE and DELETE :

| TRUNCATE | DELETE |
|---|---|
| • Truncate is a DDL statement. | • Delete is a DML Query. |
| • Truncate deletes all rows. | • Delete is selectively used to delete some record. |
| • Truncate releases the memory occupied by the records of table. | • Delete statement deletes the specific rows & its content only. |
| • Data removed using TRUNCATE cannot be recovered. | • Data removed using DELETE can be recovered (ROLLBACK) |

* UPDATE STATEMENT :

• Purpose of Update command is Update modifies the values of one or more columns in selected rows of table

Syntax : UPDATE <table_name> SET <field> = <value> WHERE <--->;

• Targeted table to be updated is named.
• 'WHERE' clause selects the rows of table to be modified.
• 'SET' clause specifies which column are to be updated and calculates the new values for them.

Ex :

• Changing all rows :

```
UPDATE employee SET hra= NULL;
```

• Changing value for more than one column :

```
UPDATE employee SET salary = 9000, hra = 900 WHERE eid=102;
```

- changing some rows :

```
UPDATE employee SET hra = 1000 WHERE salary > 3000;
```

## *. DATA CONTROL LANGUAGE – DCL :

- The Data Control language provides users with privilege commands.
- Grant privileges can be granted to others using SQL command GRANT.
- To withdraws the privileges that has been granted to user, we use the REVOKE command.

(important in BANK Transactions)

Syntax :

① COMMIT :

```
COMMIT employee ;
COMMIT ;
```

② SAVEPOINT :

```
SAVEPOINT savepoint_id ;
```

③ ROLLBACK :

```
ROLLBACK employee ;
ROLLBACK ;
ROLLBACK TO SAVEPOINT savepoint_id ;
```

④ GRANT PRIVILEGE :

```
GRANT PRIVILEGES ON <object-name> TO <user_name> ;
```

⑤ REVOKE PRIVILEGE :

```
REVOKE PRIVILEGES ON <object_name> FROM <username> ;
```

## GRANT Privileges :

- The Grant statement is used to grant security privileges on database objects to specific users.
- Grant statement is used by the owner of the table to give other user access to data.

Example : (IMP) DCL :

> UPDATE employee SET eid = '17s23' WHERE name = 'reddy';

> SAVEPOINT S1 ;

> DELETE FROM employee WHERE name = 'reddy' ;

> SAVEPOINT S2 ;

> ROLLBACK to SAVEPOINT S1 ;

> ROLLBACK ;

> GRANT select, update ON employee TO 17bcd2212 ;

> REVOKE select, update ON employee FROM 17bcd 2212 ;

## * SQL CLAUSES :

① WHERE Clause :

- Can be applied with any DML Query.
- It is used to filter the results and apply conditions in a select, insert, update or delete statement.

Syntax : SELECT col_name, col_name FROM table_name WHERE cond ;

Ex : SELECT * FROM employee WHERE ename = 'reddy' ;

SELECT * FROM employee WHERE (ename = 'reddy' AND eid = '12b23'
OR (salary >= 1000);

PRACTICE:

Display all the student name starting with 'S' in student table?

> SELECT sname FROM ~~prod~~ student WHERE sname LIKE 'S%';

> SELECT * FROM student WHERE sname LIKE 'S%';

Display ~~qty~~ list of items from prod table where Quantity is more than equal to 3 & less than equal to 5.

> SELECT * FROM prod WHERE qty >= 3 AND qty <= 5;

> SELECT * FROM prod WHERE qty BETWEEN 3 AND 5;

② SQL ORDER BY CLAUSE:

- The ORDER BY clause is used in a SELECT statement to sort results either in ascending or descending order.
- Oracle sorts query results in ascending order by default.
- Syntax:

  SELECT column-list FROM table_name [WHERE condition]
  [ORDER BY column1, [column2, ... columnN]] [DESC];

- Example:

  If we want to sort the employee table by salary of employee

> SELECT ename, salary FROM employee ORDER BY salary;

> SELECT ename, salary FROM employee ORDER BY name, salary DESC;

default : Ascending order

DESC : descending order

- The columns specified in ORDER BY clause should be one of the columns selected in the SELECT column list.

- We can represent the columns in ORDER BY clause by specifying the position of a column in the SELECT list, instead of writing column name:

  > SELECT name, salary FROM employee ORDER BY 1, 2;

## *. GROUP BY & HAVING CLAUSE:

- Before we are going to Group by & Having clause, we need to learn about Aggregate Function in SQL.

- Aggregate function: functions that take a collection of values as input and return a single value.

- Some of the commonly used aggregate functions are: SUM, COUNT, AVG, MIN, MAX.

## *. COUNT() FUNCTION:

- The count function is used to count rows & values of a column that doesn't contain a NULL value.

- The COUNT function returns a numeric value.

- Syntax:

```
SELECT COUNT [(*) | (DISTINCT | ALL)] (COLUMN_NAME)
FROM Table_name;
```

Ex:

> SELECT COUNT(eid) FROM employee ;

> SELECT COUNT(*) FROM employee ;

> SELECT COUNT (DISTINCT (did)) FROM employee ;

> SELECT COUNT (ALL salary) FROM employee ;

> SELECT COUNT (*) FROM employee ;

Note : '*' Counts all values including NULL value but 'ALL' Counts only NON NULL values in Table.

*. SUM() Function :

- The SUM Function is used to return a total on the values of a column for a group of rows.
- The SUM function can also be used in conjunction with DISTINCT.
- When SUM is used with DISTINCT, only the distinct rows are total, but total will not accurate in this case, because rows of data are omitted.
- Syntax : SUM ([DISTINCT] COLUMN_NAME)

Ex:

> SELECT SUM (salary) FROM employee ; → Totals the Salaries (All)

> SELECT SUM (DISTINCT Salary) FROM employee

→ Totals the distinct rows Salaries (duplicate sala excluded)

*. AVG() Function :

- AVG Function is used to find averages for a group of rows.
- Syntax : AVG ([DISTINCT] COLUMN_NAME)

Ex :

> SELECT AVG (salary) FROM employee ;

> SELECT AVG (DISTINCT salary) employee ;

> SELECT AVG (exp), AVG(salary) FROM employee ;

*. MIN() FUNCtion :

- The MIN Function returns the minimum value of a column for a group of rows.
- NULL values are ignored when using the MIN function.
- Syntax : SELECT MIN ([DISTINCT] COLUMN_NAME) FROM table_Name ;
- Ex :

> SELECT MIN(salary) FROM employee ;

> SELECT MIN (DISTINCT salary) FROM employee ;

*. Similary is MAX() Function :

> SELECT MAX (salary) FROM employee ;

> SELECT MAX (DISTINCT salary) FROM employee ;

RETURN all rows with 'MAX' value in SQL :

> SELECT * FROM employee WHERE salary =
(SELECT MAX(Salary) FROM employee);

## *. GROUP BY ( ) CLAUSE :

- Grouping data is a process of combining column with duplicate values in a logical order.
- Grouping data is accomplished through the use of the GROUP_BY clause of a SELECT statement (query).
- The GROUP BY clause is used in collaboration with the SELECT statement to arrange identical data into groups.
- The GROUP BY clause follows the WHERE clause in a select statement & precede the ORDER BY clause.
- The position of GROUP BY clause in a query is :
  - SELECT
  - FROM
  - WHERE
  - GROUP BY
  - ORDER BY

Syntax :

> SELECT column1, column2 FROM table1, table2 WHERE CONDITION
GROUP BY column1, column2 ORDER BY column1, column2;

Ex :

> SELECT did, SUM(salary) FROM employee GROUP BY did;

Employees

| DepartmentId | Salary | |
|---|---|---|
| 10 | 4400 | 4400 |
| 20 | 13000 | 19000 |
| 20 | 6000 | |
| 30 | 11000 | 24900 |
| 30 | 3100 | |
| 30 | 2900 | |
| 30 | 2800 | |
| 30 | 2600 | |
| 30 | 2500 | |
| 40 | 6500 | 6500 |
| 50 | 8000 | 156400 |
| 50 | 8200 | |
| 50 | 7900 | |
| 50 | 6500 | |
| 50 | 5800 | |
| ⋮ | ⋮ | |

More than 15 Rows

| DID | SUM(Sala |
|---|---|
| 10 | 4400 |
| 20 | 19000 |
| 30 | 24900 |
| 40 | 6500 |
| 50 | 156400 |

> SELECT did "DEPARTMENT CODE", COUNT(*) 'NO. OF EMPLOYEE" SUM(salary) "TOTAL SALARY" FROM Employee GROUP BY did ;

| DEPARTMENT CODE | NO. OF EMPLOYEE | TOTAL SALARY |
|---|---|---|
| 100 | 6 | 51600 |
| 30 | 6 | 24900 |
| - | 1 | 7000 |
| 90 | 3 | 58000 |
| 20 | 2 | 19000 |
| 70 | 1 | 10000 |
| 110 | 2 | 20300 |
| 50 | 45 | 156400 |
| 80 | 34 | 304500 |
| 40 | 1 | 6500 |
| 60 | 5 | 28800 |
| 10 | 1 | 4400 |

- SQL GROUP BY with WHERE clause :

> SELECT did, SUM(salary) FROM employee WHERE eid=10 GROUP BY did ;

> SELECT did, eid, SUM(salary) FROM employee GROUP BY did, eid ;

## *. HAVING ( ) CLAUSE :

- SQL HAVING clause specifies a search condition for a group or an aggregate.
- HAVING is usually used in a GROUP BY clause, but even if we are not using GROUP BY clause, we can use HAVING to a function like a WHERE clause.
- We must use HAVING with SQL SELECT ;
- Syntax :

SELECT <column_list> FROM <table_name> WHERE <condition> GROUP BY <columns> [HAVING] <condition> ;

- How a HAVING clause works ?

- The SELECT clause specifies the columns.
- The from clause supplies a set of potential rows for the result.
- The WHERE clause gives a filter for these potential rows.
- The GROUP BY clause divide the rows in a table into smaller groups.
- The having clause gives a filter for these group rows.

Ex :

→ SELECT did, SUM (salary) FROM employee GROUP BY did HAVING did >1 ;

| DID | SUM(SALARY) |
|---|---|
| 1 | 7500 |
| 2 | 8800 |
| 3 | 1200 |

⟹

| DID | SUM(SALARY) |
|---|---|
| 2 | 8800 |
| 3 | 1200 |

~~→ SELECT~~

* - Difference between WHERE & HAVING CLAUSE :

- The WHERE clause specifies the criteria which individual records must meet to be selected by a query. It can be used without the GROUP BY clause.
- The HAVING clause cannot be used without the GROUP BY clause.
- The WHERE clause selects rows before grouping. The HAVING clause selects rows after grouping.
- The WHERE clause cannot contain aggregate functions.
  The HAVING clause can contain aggregate functions.

## *. SQL SUB QUERY :

- A subquery is a SELECT statement that is embedded in a clause of another SELECT statement.

Syntax :

```
SELECT  < column ... >
FROM    < table >                    (Outer Query (or) Main Query)
WHERE   expression operator (
        SELECT  < column ... >       (Inner Query (or) Sub Query)
        FROM    < table >
        WHERE   < condition > );
```

- A sub query is a query within another query. The outer query is called as Main Query and inner query is called sub query.
- The sub query is often referred to as a nested SELECT, sub-SELECT, or inner SELECT statement.
- The subquery generally executes first and its output is used to complete the query condition for the main or outer query.

## *. Types of Subqueries :

- Single Row subquery – Returns one row of results that consists of one column to the outer query.

- Multiple Row Subquery - Returns more than one row of results to the outer query.
- Multiple - Column Subquery - Returns more than one value Column of results to the outer query.
- Correlated Subquery - Reference a column in outer Query. Executes the subQuery once for every row in the outer query.
- Uncorrelated Subquery - Executes subquery first and passes the value to the outer query.

1. SINGLE - ROW SUBQUERY :

- A single row subquery is one that returns one row from the inner SELECT statement. This type of subquery uses single - row operation.
- The operators that can be used with single row subqueries are =, >, >=, <, <=, <> → not equal to, etc..

Ex :

```
> SELECT ename FROM employee WHERE eid =
    (SELECT eid FROM projects WHERE pid = 801);

> SELECT dname, COUNT(PID) FROM projects GROUP BY name;

> INSERT INTO employee1 SELECT * FROM employee;

> INSERT INTO employee1 SELECT * FROM employee WHERE
    did = 3;
```

> INSERT into Employee1 SELECT * FROM employee WHERE eid = ANY (SELECT eid FROM projects WHERE pid = 803);

> UPDATE employee1 SET salary = (SELECT AVG(salary) FROM employee1);

> UPDATE employee1 SET salary = (SELECT AVG(salary) FROM employee1), hra = (SELECT AVG(salary)/10 FROM employee1);

> DELETE FROM employee1 WHERE salary = (SELECT AVG(salary) FROM employee1);

2. MULTIPLE - ROW Subqueries :

- Returns more than one rows.
- Use multiple row comparision operator:
  - IN : Equal to ANY number in the result
  - ALL : Compare value to every value returned by subQuery.
  - ANY : Compare value to each value returned by sub query

Ex :

> SELECT * FROM employee WHERE salary IN (SELECT MIN(SALARY) FROM employee GROUP BY did);

> SELECT * FROM employee WHERE salary > ALL (SELECT AVG(SALARY) FROM employee GROUP BY did);

> SELECT * FROM employee WHERE salary > ANY (
SELECT AVG (SALARY) FROM employee GROUP BY did);

> SELECT dname FROM department WHERE did IN
(SELECT did FROM employee WHERE eid IN
(SELECT eid FROM projects WHERE pid = 803));

---

## *. SQL JOINS :

- An SQL Join clause combines rows from two or more tables. It creates a set of rows in a temporary table.
- A JOIN works on two or more tables if they have atleast one common field and having a relationship between them.
- JOIN keeps the base tables (structured & data) unchanged.

Syntax :

```
SELECT col1, col2, col3 ... FROM table_name1, table_name2
    WHERE table_name1.col = table_name2.col;
```

- There are 2 types of SQL JOINS :
  1. EQUI JOIN / JOIN
  2. NON EQUI JOIN / THETA JOIN

1. EQUI JOIN – The SQL EQUI JOIN is a simple SQL join uses the equal sign (=) as the comparision operator for the condition. ⟶ SQL outer JOIN
   ⟶ SQL INNER JOIN

## 2. SQL NON EQUI JOIN :

The SQL NON EQUI JOIN is a join uses comparision operator other than the equal sign like >, <, >=, <= with the condition.

Ex :

CUSTOMERS

| ID | NAME | AGE | ADDRESS | SALARY |
|---|---|---|---|---|
| 1 | Ramesh | 32 | Ahmedabad | 2000 |
| 2 | Khilan | 25 | Delhi | 1500 |
| 3 | Kaushik | 23 | Kota | 2000 |
| 4 | Chaitali | 25 | Mumbai | 6500 |
| 5 | Hardik | 27 | Bhopal | 8500 |
| 6 | Komal | 22 | MP | 4500 |
| 7 | Muffy | 24 | Indore | 10000 |

ORDERS

| OID | DATE | | CUSTOMER_ID | AMOUNT |
|---|---|---|---|---|
| 102 | 2009-10-08 | 00:00:00 | 3 | 3000 |
| 100 | 2009-10-08 | 00:00:00 | 3 | 1500 |
| 101 | 2009-11-20 | 00:00:00 | 2 | 1560 |
| 103 | 2008-05-20 | 00:00:00 | 4 | 2060 |

```
> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS
  INNER JOIN ORDERS ON CUSTOMERS.ID = ORDERS
  .CUSTOMER_ID;
```

| ID | Name | Amount | Date |
|---|---|---|---|
| 3 | Kaushik | 3000 | 2009-10-08 00:00:00 |
| 3 | Kaushik | 1500 | 2009-10-08 00:00:00 |
| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
| 4 | chaitali | 2060 | 2008-05-20 00:00:00 |

> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS
LEFT JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUST_ID;

output :

| ID | Name | Amount | DATE |
|---|---|---|---|
| 1 | Ramesh | NULL | NULL |
| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
| 3 | Kaushik | 3000 | 2009-10-08 00:00:00 |
| 3 | Kaushik | 1500 | 2009-10-08 00:00:00 |
| 4 | chaitajali | 2060 | 2008-05-20 00:00:00 |
| 5 | Hardik | NULL | NULL |
| 6 | Komal | NULL | NULL |
| 7 | Muffy | NULL | NULL |

> SELECT ID, NAME, AMOUNT, DATE FROM Customers RIGHT
JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;

output :

| ID | Name | Amount | Date |
|---|---|---|---|
| 3 | Kaushik | 3000 | 2009-10-08 00:00:00 |
| 3 | Kaushik | 1500 | 2009-10-08 00:00:00 |
| 2 | Khilan | 1560 | 2009-11-20 00:00:00 |
| 4 | Chaitali | 2060 | 2008-05-20 00:00:00 |

> SELECT ID, NAME, AMOUNT FROM CUSTOMERS FULL JOIN ORDERS ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID

| ID | Name | Amount |
|---|---|---|
| 1 | Ramesh | NULL |
| 2 | Khilan | 1560 |
| 3 | Kaushik | 3000 |
| 3 | Kaushik | 1500 |
| 4 | Chaitanjali | 2060 |
| 5 | Hardik | NULL |
| 6 | Komal | NULL |
| 7 | Muffy | NULL |
| 3 | Kaushik | 3000 |
| 3 | Kaushik | 1500 |
| 2 | Khilan | 1560 |
| 4 | Chaitanjali | 2060 |

> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS, ORDER

output : Cartesian product of both Tables

total 4 columns & (7×4) 28 rows.

## *. SQL : STRING BASED RETRIEVALS :

Example Table :

| EID | EName | AGE | DID | EXP | Salary | HRA |
|---|---|---|---|---|---|---|
| 106 | John | 23 | 3 | 2 | 2000 | 100 |
| 103 | Reddy | 30 | 3 | 7 | 1200 | 120 |
| 101 | Naidu | 28 | 2 | 5 | 1800 | 180 |
| 102 | Mark | 32 | 1 | 4 | 1100 | 70 |
| 104 | David | 25 | 2 | 1 | 1000 | 520 |
| 105 | Reddy | 28 | 2 | 5 | 5000 | 300 |
| 107 | Text Text | 30 | 4 | 5 | 8000 | 800 |

Questions (Practice) :

① Find employee names starting from 'J' ?

> SELECT ename FROM employee WHERE ename LIKE 'J%';

② Find employee names ending with 'y' ?

> SELECT ename FROM employee WHERE ename LIKE '%y' ;

③ Find employee names having 'a' in any position ?

> SELECT ename FROM employee WHERE ename LIKE '%a%' ;

④ Find employee names having 'i' only in 3rd position?

> SELECT ename FROM employee WHERE ename LIKE '__i%' ;

⑤ Find employee names having exactly 4 characters ?

> SELECT ename FROM employee WHERE LENGTH (ename) = 4;

⑥ Find employee names starting with 'M' & ending with 'K'

> SELECT ename FROM employee WHERE ename LIKE 'M%K';

⑦ Find employee names having '%' symbol ?

> SELECT ename FROM employee WHERE ename LIKE '%\%%' ESCAPE '\';

⑧ Find employee names start with 'J' or 'M' ?

> SELECT ename FROM employee WHERE ename LIKE 'J%' OR ename LIKE 'M%';

⑨ Display all employee names in uppercase in descending order

> SELECT UPPER (ename) FROM employee ORDER BY ename DESC;

⑩ Similar like UPPER (ename) we have LOWER (ename);

## *. FUNCTIONAL DEPENDENCY :

- The functional dependency is a relationship that exists between two attributes (columns).
- It typically exists between the primary key and non-key attributes within a table. $X \longrightarrow Y$
- A column Y of a relational table is said to be functionally dependent upon column X when values of column Y can uniquely identified by values of column X.
- The left side of FD is known as determinant, the right side of the production is dependent.

  Ex: Emp_Id → Emp_Name.
- If the information stored in a table can uniquely determine another information in the same table, then it is called Functional dependency.
- Consider it as an association between two attributes of same relation.

ex: < Employee >

| EmpID | EmpName | EmpAge |
|---|---|---|
| E01 | Amit | 28 |
| E02 | Rohit | 31 |

In the beside table Ename is functionally dependent on EmpID because EmpName can take only one value for given value of EmpID

EmpID → EmpName

Ex :

| A | B | C |
|---|---|---|
| 1 | 3 | 5 |
| 2 | 4 | 5 |
| 1 | 5 | 6 |
| 2 | 3 | 6 |

If $a \cdot \gamma_x = a \cdot \gamma_y$

then $b \cdot \gamma_x = b \cdot \gamma_y$

then $a \longrightarrow b$

(Functional dependency)

$A \longrightarrow A$

$B \longrightarrow B$

$C \longrightarrow C$

$AB \longrightarrow C, AB$

$AC \longrightarrow AC, B$

$BC \longrightarrow BC, A$

$ABC \longrightarrow ABC$

Ex :

| A | B | C |
|---|---|---|
| 1 | 3 | 5 |
| 2 | 4 | 6 |
| 3 | 3 | 7 |
| 4 | 4 | 8 |

If $a \cdot \gamma_x = a \cdot \gamma_y$

and $b \cdot \gamma_x = b \cdot \gamma_y$

then $a \longrightarrow b$

(Functional dependency)

$A \longrightarrow B, C, A$

$B \longrightarrow B$

$C \longrightarrow A, B, C$

$AB \longrightarrow AB, C$

$AC \longrightarrow AC, B$

$BC \longrightarrow BC, A$

} Partial Dependency

Ex :

| A | B | C |
|---|---|---|
| 1 | 3 | 5 |
| 2 | 4 | 6 |
| 1 | 3 | 7 |
| 2 | 4 | 8 |

$A \longrightarrow B, A$

$B \longrightarrow A, B$

$C \longrightarrow A, B, C$

} Fully Functional Dependency

$AB \longrightarrow AB$ } No Dependency

$AC \longrightarrow B, AC$

$BC \longrightarrow A, BC$

} Partial Dependency

Ex :

| A | B | C | D |
|---|---|---|---|
| 1 | 7 | 5 | 5 |
| 2 | 8 | 6 | 5 |
| 1 | 9 | 7 | 5 |
| 2 | 7 | 8 | 5 |

A → D, A

B → D, B

C → A, B, D, C

D → D

AB → C, AB, D

BC → A, BC, D

AC → B, D, AC

AD → AD

CD → A, B, CD

BD → BD,

ABC → D, ABC

ABD → C, ABD

BCD → A, BCD

ACD → B, ACD

ABCD → ABCD

$\{2^n - 1\}$
Formula

Note : If you have same value in all rows, it will be determined by any column. (It can be written right side) → ?

Note : If you don't have same value in all rows, (all unique) then it is determinant to any column. (It can be written left hand side) ? →

Ex :

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

A → B, D .

B → A, D

C → D

D → X

AB → D

BC → A, D

AC → B, D

AD → B

CD → X

BD → A

ABC → D

ABD → X

BCD → A

ACD → B

ABCD → X

Here, C is determinant only cannot be determined by any others. Like C is only present on left side but not no where on right side.

Ex :

| A | B | C | D |
|---|---|---|---|
| 1 | 6 | 5 | 5 |
| 2 | 7 | 6 | 5 |
| 1 | 8 | 7 | 5 |
| 2 | 9 | 8 | 5 |

A → D

B → A, C, D

C → A, B, D

D → X

AB → C, D

BC → A, D

AC → B, D

AD → X

CD → A, B

BD → A, C

ABC → D

ABD → C

BCD → A

ACD → B

ABCD → X

## *. FULLY FUNCTIONAL DEPENDENCY :

- An attribute is fully functional dependent on another attribute, if it is Functionally Dependent on that attribute and not on any of its proper subset.
- For example, an attribute Q is fully functional dependent on another attribute P, if it is Functionally Dependent on P and not on any of proper subset of P.

< Employee Project >

| EmpID | ProjectID | Days |
|---|---|---|
| E099 | 001 | 320 |
| E056 | 002 | 190 |

Duplicates may there

$\{EMPID, ProjectID\} \longrightarrow \{Days\}$

- Same Employee may work on multiple Projects

IMP

Ex:

| A | B | C |
|---|---|---|
| 1 | 3 | 3 |
| 1 | 4 | 7 |
| 2 | 3 | 8 |
| 2 | 4 | 3 |

$A \rightarrow X$

$B \rightarrow X$

$C \rightarrow A, B$

$AB \rightarrow C \longrightarrow$ Fully Functional

$AC \rightarrow B$, $BC \rightarrow A$ } Partial Dependency

- To determine C values A alone & B alone is not sufficient, so, AB (combined) needed to determine C. Such type $(AB \rightarrow C)$ of dependency is "Fully Functional Dependency."
- Whereas to determine A values, C alone is enough & BC alone is enough, so C (or) BC either of them are sufficient $\{C \longrightarrow A, BC \longrightarrow A\}$. Such type of dependency is "Partial Dependency."

## *. PARTIAL DEPENDENCY :

- Partial Dependency occurs when a non prime attribute is functionally dependent on part of a Candidate Key.

- Ex:

| Student ID | projectNo | StudentName | ProjectName |
|---|---|---|---|
| S01 | 199 | Katie | Geo location |
| S02 | 120 | Ollie | Cluster web |

Here,

- The prime key attributes are StudentID & ProjectNo.
- As stated, the non-prime attributes i.e. StudentName & ProjectName should be functionally dependent on a part of a Candidate key, to be partial Dependent.
- The studentName can be determined by studentID that makes the relation Partial Dependent.
- The projectName can be determined by ProjectID, which that the relation Partial Dependent.

## *. TRIVIAL FUNCTIONAL DEPENDENCY :

- $A \rightarrow B$ has trivial functional dependency if B is subset of A.
- The following dependencies are also trivial like:

  $A \rightarrow A$ , $B \rightarrow B$ ,

Ex: {EmployeeID, EmployeeName} → EmployeeID is a trivial functional dependency as EmployeeID is a subset of {Employee_ID, EmployeeName}.

## *- NON-TRIVIAL FUNCTIONAL DEPENDENCY:

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

| | |
|---|---|
| AB → A (or) AB → B | Trivial |
| A → B, B → C | Non Trivial |

## *- INFERENCE RULE (IR):

- The inference Rule is a type of assertion. It can apply to a set of FD (functional dependency) to derive other FD.
- Using the inference rule, we can derive additional functional dependencies from initial set.
- The Functional dependency has 6 types of Inference Rule:

1) Reflexive Rule
2) Augmentation Rule
3) Transitive Rule
4) Union Rule
5) Decomposition Rule
6) Pseudo Transitive Rule

## Rule 1. Reflexive Rule (IR1)

- In the reflexive rule, if Y is subset of X, then X determines Y.
- If $X \supseteq Y \rightarrow X \rightarrow Y$
- Example :

  $X = \{a, b, c, d, e\}$

  $Y = \{a, b, c\}$

## Rule 2. Augmentation Rule (IR2)

- The Augmentation is called as partial Dependency. In Augmentation, if X determines Y, then XZ determines YZ for any Z.
- If $X \rightarrow Y$ then $XZ \rightarrow YZ$
- Example :

  For R(A B C D), if $A \rightarrow B$ then $AC \rightarrow BC$

## Rule 3. Transitive Rule (IR3)

- In transitive rule, if X determines Y and Y determines Z then X must also determine Z.
- If $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

Rule (4). Union Rule (IR4)

- Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.
- If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$

Rule (5). Decomposition Rule (IR5)

- Decomposition Rule (IR5) is also known as project rule.
- It is the reverse of Union rule.
- This rule says, if X determines Y and Z, then X determines Y and X determines Z seperately.
- If $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$

Rule (6). Pseudo Transitive Rule (IR6)

- In Pseudo Transitive rule, if X determines Y and YZ determines W, then XZ determines W.
- If $X \rightarrow Y$ and $YZ \rightarrow W$ then $XZ \rightarrow W$

*. DECOMPOSITION :

- A Functional decomposition is the process of breaking down the functions of an organization into progressively greater (finer & finer) levels of detail.

- Decomposition helps in eliminating some of the problems of bad design such as redundancy, inconsistencies & anomalies.

- $X \rightarrow YZ$ $\Rightarrow$ $X \rightarrow Y$ and $X \rightarrow Z$

- There are two types of decomposition:

## ① LOSSY DECOMPOSITION:

- The decomposition of relation R into R1 & R2 is lossy when the join of R1 and R2 is lossy when the join of R1 & R2 doesnot yield the same relation as in R. $\{R \neq R'\}$

Ex:

R

| RollNo | Sname | Dept |
|---|---|---|
| 111 | pavimal | Computer |
| 222 | pavimal | Electrical |

$=$

R1

| Roll No | Sname |
|---|---|
| 111 | Pavimal |
| 222 | Pavimal |

$+$

R2

| Sname | Dept |
|---|---|
| Pavimal | Computer |
| Pavimal | Electrical |

Merge / JOIN

Not same

R'

| RollNo | Sname | Dept | |
|---|---|---|---|
| 111 | Pavimal | Computer | |
| 111 | Pavimal | Electrical | extra |
| 222 | Pavimal | Computer | extra |
| 222 | Pavimal | Electrical | |

## ② LOSSLESS JOIN DECOMPOSITION:

- "The decomposition of relation R into R1 and R2 is lossless when the join of R1 & R2 yield same relation as in R" $\{R = R'\}$

Eg:

R

| Roll no | Sname | Dept |
|---|---|---|
| 111 | Parimal | Computer |
| 222 | Parimal | Electrical |

=

R1

| Roll No | Sname |
|---|---|
| 111 | Parimal |
| 222 | Parimal |

+

both are same

+

| RollNo | Dept |
|---|---|
| 111 | Computer |
| 222 | Electrical |

R2

Merge / JOIN →

| RollNo | Sname | Dept |
|---|---|---|
| 111 | Parimal | Computer |
| 222 | Parimal | Electrical |

R'

---

*. PRACTICE ON KEYS: FAT **IMP

Consider a relation R (A B C)

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

i. Find Functional dependency:

Step 1:

A → B, C  Relation R

B → ✗

C → ✗

AB → C
AC → B
BC → A
ABC → ABC

} Relation R

Step 2: Check all Functional Dependies whether it returns Relation R or not!

A → BC = R

AB → C = R

BC → A = R

AC → B = R
ABC → ABC = R

} Returns Relation R

- If it returns R ⇒ it is "Super Key"

★ Super key : A , AB , AC , ABC , BC

★ Candidate Key : A minal Super key

ABC - Super key

AB✓ - AB is minimal & Super key - AB is minimal Super key

A✓ - A is minimal & Super key - A is minimal Super key

So, A Cannot be futher minimized. Hence A is Candidate Key 1 ∴ **A - CK 1**

BC - Super key → Hence BC is already minimal super key

B✗ - B - not a minal superkey ✗

C✗ - C - not a minal super key ✗

} So, No minimization possible

∴ **BC is Candidate Key 2**

| Proper subset | | Candidate Key | |
|---|---|---|---|
| A | → ● | A | ✓ |
| AB | → A , B | B | ✗ |
| AC | → A , C | C | ✗ |
| ABC | → A , AB , AC , BC , B , C | BC | ✓ |
| BC | → B , C | | ✗ |

∴ A — Candidate key 1

BC — Candidate key 2

{ A, BC }

★ Primary Key : A Candidate key without Null value & not a secure value

A – Candidate key, not Null, secure value (if) ✗

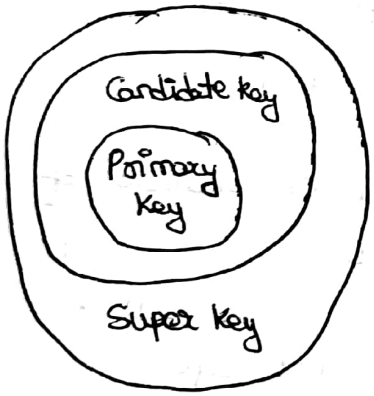
BC – Candidate key, not Null, not secure value ✓ (PK)

---

+ Primary Key → NOT NULL, NOT SECURE VALUES
→ CANDIDATE KEY
→ MINIMUM COLUMNS COMBO
→ VERY VERY UNIQUE

+ Candidate Key → SUPER KEY
→ MINIMAL SUPER KEY
→ MAY HAVE SECURE VALUES BUT UNIQUE

+ Super Key → RETURNS WHOLE RELATION
→ MAY BE NULL
→ HAS FUNCTIONAL DEPENDCY




RELATION R

+ Alternative key → another PRIMARY KEY
(or) Alternate key → A Candidate key not a Primary key
(or) Secondary key → MUST NOT BE NULL & UNIQUE TO BE

---

FAT { important Questions }

Ex :

R(

| A | B | C | D |
|---|---|---|---|
| 1 | 1 | 5 | 1 |
| 2 | 1 | 7 | 1 |
| 3 | 1 | 7 | 1 |
| 4 | 2 | 7 | 1 |
| 5 | 2 | 5 | 1 |
| 6 | 2 | 5 | 7 |

Super key :

- A is unique row ⟹ all combinations of A are super keys

A, AB, AC, AD, ABC, ACD, ABD, ABCD

Also, BCD is not super key ⟹ any combinations of BCD — B, C, D, BC, CD, BD ✗ are not super keys

A - ✓
AB - ✓
AC - ✓
AD - ✓
ABC - ✓
ACD - ✓
ABD - ✓
B - ✗
C - ✗
D - ✗
BC - ✗
CD - ✗
BD - ✗
BCD - ✗
ABCD - ✓

} All combinations of Functional Dependencies

Super Keys are :

that returns relation R

A, AB, AC, AD, ABC, ACD, ABD, ABCD

(8 super keys)

Candidate Key : Minimal Super Key

A, AB, AC, AD, ABC, ACD, ABD, ABCD → Here all are combinations of A. Therefore minal + Superkey of all these will be only A

A – Candidate key

Primary key – A Candidate key being not null & unique

since we have only 1 Candidate key ⇒ It is only our Primary key also

A – primary key

∴ Super key = 8 = {A, AB, AC, AD, ABC, ACD, ABD, ABCD}

Candidate key = 1 = {A}

Primary key = 1 = {A}

Alternate key = 0 = no key

---

Example :

Consider R(A, B, C, D, E)

| A | B | C | D | E |
|---|---|---|---|---|
| x | 2 | 3 | 4 | 5 |
| 2 | x | 3 | 4 | 5 |
| x | 2 | 3 | 6 | 5 |
| x | 2 | 3 | 6 | 6 |

## CLOSURE SET : [FAT]

Find closure set of following relation :

① R (A, B, C, D, E, F, G)

FD1 : A → B

FD2 : BC → DE

FD3 : AEG → G

Q : (AC)* = ?

sol :

(AC)* — ABC by FD1

ABC — ABCDE by FD2

ABCDE — X no more

(AC)* — A B C D E

columns can be found

∴ closure set = {A, B, C, D, E}

② R (A, B, C, D, E)

FD1 : A → BC

FD2 : CD → E

FD3 : B → D

FD4 : E → A

Q : (B)* = ?

sol :

(B)* — BD by FD3

BD — X no more

closure set = {B, D}

③ R (A, B, C, D, E, F)

FD1 : AB → C

FD2 : BC → AD

FD3 : D → E

FD4 : CF → B

Q : (AB)* = ?

sol :

(AB)* — ABC by FD1

ABC — ABCD by FD2

ABCD — ABCDE by FD3

closure set = {A, B, C, D, E}

④ $R(A, B, C, D, E, F, G, H)$

FD1 : $A \rightarrow BC$

FD2 : $CD \rightarrow E$

FD3 : $E \rightarrow C$

FD4 : $D \rightarrow AEH$

FD5 : $ABH \rightarrow BD$

FD6 : $DH \rightarrow BC$

FD7 : $BCD \rightarrow H$ → Is this Valid ? { YES }

Sol : $(BCD)^*$ — BCDE by FD2

BCDE — ABCDEH by FD4

So, we obtained H by BCD $(BCD \rightarrow H)$

So, the Functional Dependancy given is Valid (YES)

BCD determines H.

(FAT)

IMP

* Find super key, Primary key, Candidate key of the given Relation R (A, B, C).

| A | B | C |
|---|---|---|
| 1 | P | X |
| 2 | Q | Y |
| 3 | Q | X |
| 4 | S | Y |

A - B, C = R

B - X

C - X

AB - C = R

BC - A = R

AC - B = R

ABC - ✓ = R

- **Super Key** : A Combination that returns relation R

All combinations of A : {A, AB, AC, ABC} and BC returns relation R.

∴ A, AB, AC, ABC ⟶ minimal super key = A

BC ⟶ Candidate key 2

- Superkey : {A, AB, AC, BC, ABC}
- Candidate key : {A, BC}
- Primary key : {A, BC} either of them = {A} ; {BC} if than
- Alternate key (or) secondary key : = {BC} ; {A} if than

---

**Ex** : Find Candidate key for given Functional Dependencies :

Consider R (A, B, C, D)

FD1 : A ⟶ BCD

FD2 : AB ⟶ CD

FD3 : ABC ⟶ D

FD4 : BD ⟶ AC

FD5 : C ⟶ AD

✓ A ⟶ ABCD = R by FD1

AB ⟶ CD by FD2

CD ⟶ AD by FD5

⇒ ABCD = R by

✓ AB ⟶ ABCD = R

ABC ⟶ D by FD3

✓ ABC ⟶ ABCD = R

BD ⟶ AC by FD4

✓ BD ⟶ ABCD = R

Super key - which returns relation R

| | proper subset | Candidate key |
|---|---|---|
| A | × | yes - A |
| AB | A, B | × |
| ABC | AB, BC, AC, A, B, C | × |
| BD | B, D | yes - BD |

{ not minimal

Candidate key = { A, BD }

Primary key = {A} or {BD}

Secondary key = {BD} or {A}

Super key = { A, AB, ABC, BD }

---

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

FD1: A → BCD

FD2: AB → CD

FD3: ABC → D

FD4: BD → AC

FD5: C → AD

Super keys :

A → ABCD by FD1 ≠ R

AB → ABCD by FD2 ≠ R

ABC → ABCD by FD3 ≠ R

BD → ABCD by FD4 ≠ R

C → ACD by FD5

ACD → ABCD by FD1 ≠ R

- Super key = ∅ null set
- Candidate key = no super key so No Candidate key
- Primary key = No Candidate key so No primary key

---

By using Functional Dependencies we can conclude that . Here if we use Functional Dependencies, we obtain above conclusions which is incorrect because a database cannot be designed without primary, super, candidate keys

So, we apply the concept of closure set here.

⇒ Let us assume our key $(AEFGH)^*$

$AEFGH - ABCDEFGH = R$ ( AEFGH is super key)

⇒ Let us assume our key $(BEFGH)^*$

$BEFGH - X \neq R$

For $(CEFGH)^*$

$C \rightarrow AD$ by FD5

$CEFGH \rightarrow CEFGHAD - ACDEFGH$ by FD5

$A \rightarrow ABCD$ by FD1

$CEFGH \rightarrow ABCDEFGH = R$ (∵ CEFGH is super key)

For $(DEFGH)^*$

$DEFGH - X \neq R$

∴ super key = { AEFGH , CEFGH }

## *. USING CONCEPT OF EDGE DIAGRAM :

Consider relation R (A B C D E F G H I J)

Given Functional dependencies are :

$A \rightarrow DE$

$B \rightarrow F$

$AB \rightarrow C$

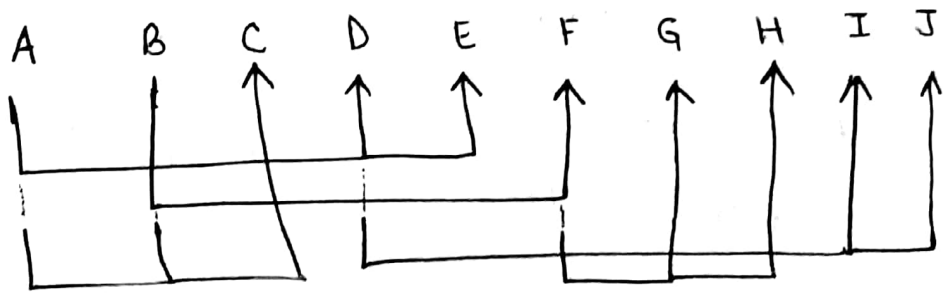
$D \rightarrow IJ$

$F \rightarrow GH$

Find the superkey Candidate Key primary key & alternate key for given Relation R.

Sol: Since we obtain different (so many) Combination for R, it is difficult to find exact super key. Hence to reduce our work, we can apply Edge Diagram Concept.

EDGE DIAGRAM :




which don't have a pointing arrow that is closure set here except {AB} remaining all have pointed arrow.

$A^* \neq R$ ✗

$B^* \neq R$ ✗

$AB^* = R \rightarrow$ Super Key → candidate key

$A^* = R$

$B^* = R$ } Candidate Key

$AB^* = R \rightarrow$ Super Key

$A^* \neq R$ ✗

$B^* = R \rightarrow$ Candidate Key

$AB^* = R \rightarrow$ Super Key

$A^* = R \rightarrow$ candidate key

$B^* \neq R$ ✗

$AB^* = R \rightarrow$ super key

closures : $(AB)^*$ $(A)^*$ $(B)^*$

A, B → prime attributes, mandatory for Candidate Key

$(AB)^*$ → ABDE by FD1

ABDE → ABDEF by FD2

ABDEF → ABCDEF by FD3

ABCDEF → ABCDEFIJ by FD4

ABCDEFIJ → ABCDEFGHIJ by FD5 = R (super key)

[ ∴ AB & AB's combinations are super key

All AB combinations → super key ]

Since other than AB, AB combinations, remaining are not at all possible because (they have pointing arrows) in Edge Diagram.

Since AB combinations are further minimised to AB

So, minimal super key = {AB}

∴ Candidate key = {AB}

∴ Primary key = {AB}

# NORMALIZATION :

- Normalization is the process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations.

- Normalization is the process of organizing the data in the database.

- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.

- dB → data redundancy → we have to go for normalization
- why need to avoid ?

1. Insert Anomaly — inserting unwanted
   — missing mandatory

2. Update Anomaly — update based on non key columns
   — updating non matching records.

3. Delete Anomaly : It deletes, affects other columns

↓

| | Sid | dept |
|---|---|---|
| × | 1. | ~~ece~~ |
| | 2. | cse |
| | 3. | cse |
| | 4. | mech |
| | | ~~ece~~ |

- Normalization divides the larger table into the smaller table and links them using relationship.
- The normal form is used to reduce redundancy from the database table.

## *. FIRST NORMAL FORM (1NF) :

For a table to be in First Normal Form, it should follow the following 4 rules :

- It should have only single (atomic) valued attributes.
- Values stored in a column should be of same domain.
- All the columns in table should have unique names.
- And the order in which data is stored, does not matter.

Ex:

Before Normalization

| roll_No | name | subject |
|---|---|---|
| 101 | AKon | OS, CN |
| 103 | CKon | Java |
| 102 | BKon | C, C++ |

After 1NF

| roll_no | name | subject |
|---|---|---|
| 101 | AKon | OS |
| 101 | AKon | CN |
| 103 | CKon | Java |
| 102 | BKon | C |
| 102 | BKon | C++ |

## *. SECOND NORMAL FORM (2NF):

- In the 2NF, relational must be in 1NF.
- In second normal form, all non-key attributes are fully functional dependent on the primary key.

| Teacher_ID | Subject | Teacher_Age |
|---|---|---|
| 25 | chemistry | 30 |
| 25 | Biology | 30 |
| 47 | English | 35 |
| 83 | Math | 38 |
| 83 | Computer | 38 |

| Teacher_ID | Teacher_Age |
|---|---|
| 25 | 30 |
| 47 | 35 |
| 83 | 38 |

| Teacher_ID | Subject |
|---|---|
| 25 | chemistry |
| 25 | Biology |
| 47 | English |
| 83 | Math |
| 83 | Computer |

## *. THIRD NORMAL FORM (3NF):

- A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
- 3NF is used to reduce the data duplication. It is also used to achieve the data Integrity.
- If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in 3NF if it holds atleast one of the following conditions for every non-trivial function dependency. $X \longrightarrow Y$

- X is a super key.
- Y is a prime attribute i.e. each element of Y is part of some Candidate key.

Ex: Resultant table of 2NF,

| emp_id | emp_name | emp_zip | emp_state | emp_city | dist |
|---|---|---|---|---|---|
| 1001 | John | 282005 | UP | Agra | Dayal |
| 1002 | Ajeet | 222008 | TN | chennai | Mcity |
| 1006 | Lora | 282007 | TN | chennai | Urbapak |
| 1101 | Lilly | 292008 | UK | Pauri | Bhagwan |
| 1201 | Steve | 222999 | MP | Gwalior | Ratan |

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

emp_id → emp_zip

emp_zip → emp_state, emp_city, emp_dist } Transitive dependency exists

solve by,

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

| emp_zip | emp_state | emp_city | dist |
|---|---|---|---|
| 282005 | UP | Agra | Dayal |
| 222008 | TN | chennai | Mcity |
| 282007 | TN | chennai | Urbapak |
| 292008 | UK | Pauri | Bhagwan |
| 222999 | MP | Gwalior | Ratan |

## *. BOYCE CODD NORMAL FORM (BCNF) :

- BCNF is the advance version of 3NF. It is stricter than 3NF.
- A table is in BCNF if every functional dependency $X \rightarrow Y$, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

Ex :

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

$A \rightarrow BCD$
$BC \rightarrow AD$
$D \rightarrow B$

} Keys = {A, BC}

Above relationship is already in 3NF.

In Functional dependency, $A \rightarrow BCD$, A is super key. In second relation $BC \rightarrow AD$, BC is also key. But in $D \rightarrow B$, D is not a key.

Here, we break our relationship into R1 & R2 :

R(A, B, C, D)

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

## *. MULTI VALUED DEPENDENCY :

- Multivalued Dependency occurs when two attributes in a table are independent of each other but, both depend on third attribute.
- A multivalued dependency consists of atleast two attributes that are dependent on a third attribute that's why it always require atleast three attributes.

Ex:

| Bike_Model | Year | Color |
|---|---|---|
| M2011 | 2008 | White |
| M2001 | 2008 | Black |
| M3001 | 2013 | White |
| M4006 | 2017 | White |
| M4006 | 2017 | Black |

Here columns COLOR and year are dependent on Bike_Model and independent of each other.

## *. FOURTH NORMAL FORM (4NF):

- A relation will be in 4NF if it is in Boyce Codd normal form and has no multiple valued dependency.
- For a dependency $A \longrightarrow B$, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

Ex:

| STU ID | Course | Hobby |
|---|---|---|
| 21 | Computer | Dancing |
| 21 | Math | singing |
| 34 | Chemistry | Dancing |
| 74 | Biology | Cricket |

STUDENT relation ←

In the STUDENT relation, a student with STU_ID, 21 contains two courses Computer and Math and two hobbies, Dancing & Singing. So there is Multi valued dependency on STUID, which leads to unnecessary repetition of data.

Resultant 4NF :

| STUID | Course |
|---|---|
| 21 | Computer |
| 21 | Math |
| 34 | Chemistry |
| 74 | Biology |
| 59 | Physics |

| STUID | Hobby |
|---|---|
| 21 | Dancing |
| 21 | Singing |
| 34 | Dancing |
| 74 | Cricket |
| 59 | Hockey |

## *- FIFTH NORMAL FORM (5NF) :

- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
- 5NF is satisfied when all the tables are broken into as many tables as possible inorder to avoid redundancy.
- 5NF is also known as Project Join Normal Form (PJ/NF).

Ex :

| Subject | Lecturer | Semester |
|---|---|---|
| Computer | Anshika | Sem 1 |
| Computer | John | Sem 1 |
| Math | John | Sem 1 |
| Math | Akash | Sem 2 |
| Chemistry | Pearson | Sem 1 |

In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields require a valid data

Suppose we add a new semester as Sem 3 but don't know about the subject and who will be taking the subject so we leave lecturer & subject as NULL. But all the three columns acts as primary key, so we cannot leave other two columns blank.

So, to make the above table, into 5NF, we can decompose it into three relations P1, P2 & P3.

Resultant 5NF :

| Subject | lecturer |
|---|---|
| computer | Anshika |
| computer | John |
| Math | John |
| Math | Akash |
| chemistry | praveen |

| Subject | Semester |
|---|---|
| Computer | Sem 1 |
| Computer | Sem 1 |
| Math | Sem 1 |
| Math | Sem 2 |
| Chemistry | Sem 1 |

| lecturer | Semester |
|---|---|
| Anshika | Sem 1 |
| John | Sem 1 |
| John | Sem 1 |
| Akash | Sem 2 |
| Praveen | Sem 1 |

## 1NF

No multivalues in columns

Only single valued

## 2NF

1NF +

No partial Dependancy

Proper subset of candidate key should not determine non prime attribute.

Only Full Dependancy

$(AB) \longrightarrow C$

$B \rightarrow C$ ×

$A \rightarrow C$

★ proper subset of CK → Non unique, Non prime attribute

is there then it is not in 2NF

## 3NF

2NF +

No Transitive Dependancy

No Non prime → Non prime

$X \rightarrow Y$

$Y \rightarrow Z$ ×

LHS should be ★ Candidate key (or)

RHS should be prime attribute

## BCNF

3NF +

All Functional dependencies, LHS must be Super key or Compose Candidate Key.

$X \rightarrow Y$ ✓

super key must

★ LHS of all FD's should be Candidate key.

## 4NF

BCNF +

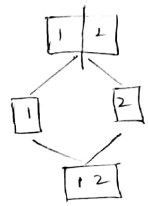
No multivalued Dependancy

$X \twoheadrightarrow Y$

## 5NF

4NF +

All must be in lossless decompositi...




common attribute must be only Candidate Key.

Example:

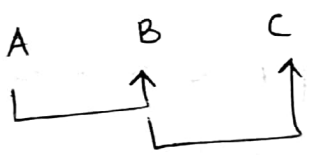
Consider Relation R(A B C D)

Normalize the table upto 3NF | BCNF | 4NF

$A \rightarrow B$

$B \rightarrow C$

Step1: Find Super key, Candidate key, Prime Attributes




A, D → are mandatory in Candidate key

$(AD)^* \rightarrow B$ by FD1

$B \rightarrow ABCD$ by FD2

$= R$

∴ AD is Super key ✓

Since $(A)^* \neq R \rightarrow BC$

$(D)^* \neq R \rightarrow \{\}$ } ×

Also, AD is Candidate key

Prime Attributes = A, D

Proper subset of AD = {A, D}

Step2: check 2NF

Given $A \rightarrow B$ : If is proper subset determines non prime violates 2NF.

$B \rightarrow C$ ✓

Proper subset of Candidate key {AD} → {A, D} proper subset, non prime attribute {B} should not determine

$A \rightarrow B$ × violates

Step3: Normalize R into 2NF

R(A B C D)

1) Dependency preserving
2) It must be lossless decomposition } ensure this during decomposition.

$(A)^* \longrightarrow ABC$

① Assume we divided as R1(ABC) & R2(D)

- If $R1 \cap R2$ = prime Attribute ⟶ Lossless Decomposition
- If FDS of R1 ∪ FDS of R2 = FDS of R then Dependancy preserving.

R1 (ABC) ✗ $R1 \cap R2 = \{\}$ ≠ prime Attribute

Lossless Decomposition Failed

R2 (D)

FD of R1 = $A \rightarrow B$, $B \rightarrow C$

FD of R2 = { }

✗ FD of R1 ∪ FD of R2 ≠ FD of R. So, Dependancy preserving not there.

② Assume we decomposed as R1(ABC) & R2(AD)

{ $A \rightarrow B$, $B \rightarrow C$ } : FD1 for R1 | FD2 : { } for R2

FD1 of R1 ∪ FD2 of R2 = FD of R. ✓

$R1 \cap R2 = ABC \cap AD = A$ = prime Attribute ✓

∴

R(A B C D)

R1 (A B C) R2 (A D)

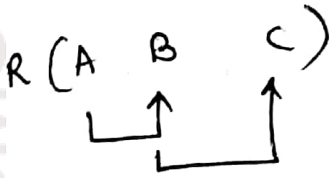
This is in 2nd Normal Form.

Step 4 : check 3NF

R1 (ABC)

A → B
B → C  } violates 3NF ⟹ Transitive dependency exists

checking 2NF




(A)* → ABC

A is candidate key

It is in 2NF

Converting & decomposing R1 (A B C) into

R11 (A B)  R12 (BC)

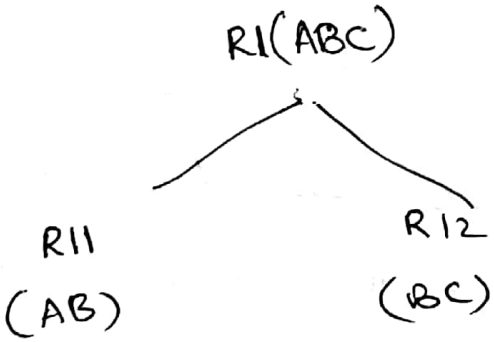
A → B  B → C

R11 ∪ R12 = R ✓

R11 ∩ R12 = {B}

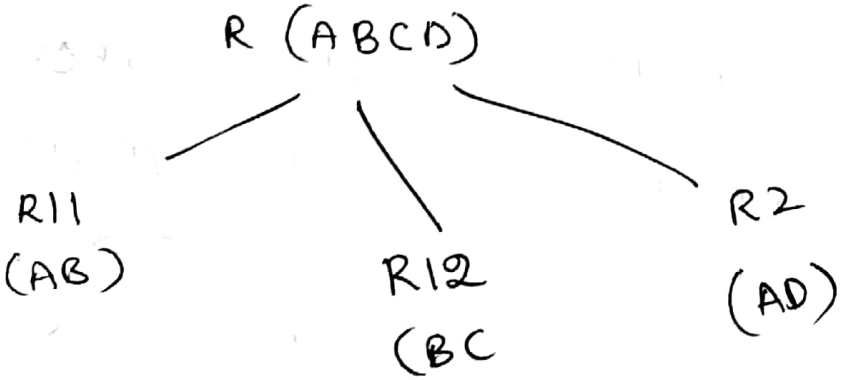
↓ prime attribute of R12

{B → C}




3NF obeyed ⟹
2NF obeyed ⟹
1NF obeyed

So,



Ex: R(A B C D E F), check the highest Normal form!

FD: {AB → C, C → DE, E → F, F → A}

Step1: Find all Candidate keys in Relation

(AB)* ⟶ ABC by FD1

ABC → ABCDE by FD2

ABCDE → ABCDEF by FD3 = R ⇒ Super key

AB is super key

A − {A}
B − {B}
} ≠ R ⇒ AB − {ABCDEF} = R

minimal super key so, AB is Candidate key

Now to get other Candidate keys, A (or) B is present on RHS of any FD. If it is present then replace

AB
FB by FD4 (A is on RHS)
EB by FD3 (F is on RHS)
CB

(FB)* ⟶ FBA by FD1

FBA ⟶ ABCF by FD1

ABCF ⟶ ABCDEF by FD2

= R

∴ FB is super key & minimal also

So, FB is ~~sup~~ CK

Candidate key = {AB, FB, EB, CB}

Step2: Write all Prime Attributes

{A, B, C, E, F}

Step3: Write all Non prime Attributes : {D}

| F.D | AB → C | C → DE | E → F | F → A | |
|---|---|---|---|---|---|
| BCNF | ✓ | ✗ | ✗ | ✗ | |
| 3NF | ✓ | ✗ | ✓ | ✓ | |
| 2NF | ✓ | ✗ | ✓ | ✓ | |
| 1NF | ✓ | ✓ | ✓ | ✓ | (by default) |

Ex: In the same above, reduce / Normalize.

By default its' in 1NF, so, we should convert to 2NF. They are not in 2NF.

| AB → C | C → D | C → E | E → F | F → A |
|---|---|---|---|---|
| Fully Functional Dependent ✓ | Partial Dependancy ✗ | FD ✓ | FD ✓ | FD ✓ |

↳ This Functional depency is not making entire relation to attain 2NF. So, we need to remove this & create it as another relation/table

R(A B C D E F)

R1 (A B C E F)

{AB → C, C → E, E → F, F → A}

Candidate Keys = {AB, FB, EB, CB}

R2 (C D)

{C → D}

Candidate Keys = {C}

Now R1 (ABCEF) & R2 (CD) are in 2NF.

★ LHS must be a Candidate key (OR) RHS of Prime Attribute ⟹ 3NF Satisfied

| $AB \to C$ | $C \to E$ | $E \to F$ | $F \to A$ |
|---|---|---|---|
| LHS : ✓ | LHS: X | LHS: X | LHS : X |
| RHS : ✓ | RHS: ✓ | RHS: ✓ | RHS: ✓ |

R1 (ABCEF)

Candidate key = {AB, FB, EB, CB}

Prime Attribute = {A, B, C, E, F}

$C \to D$ LHS : ✓ RHS : X

Candidate key = {C} Prime Attribute = {C}

R2 (CD)

So, R1 (ABCEF) & R2 (CD) are in 3NF.

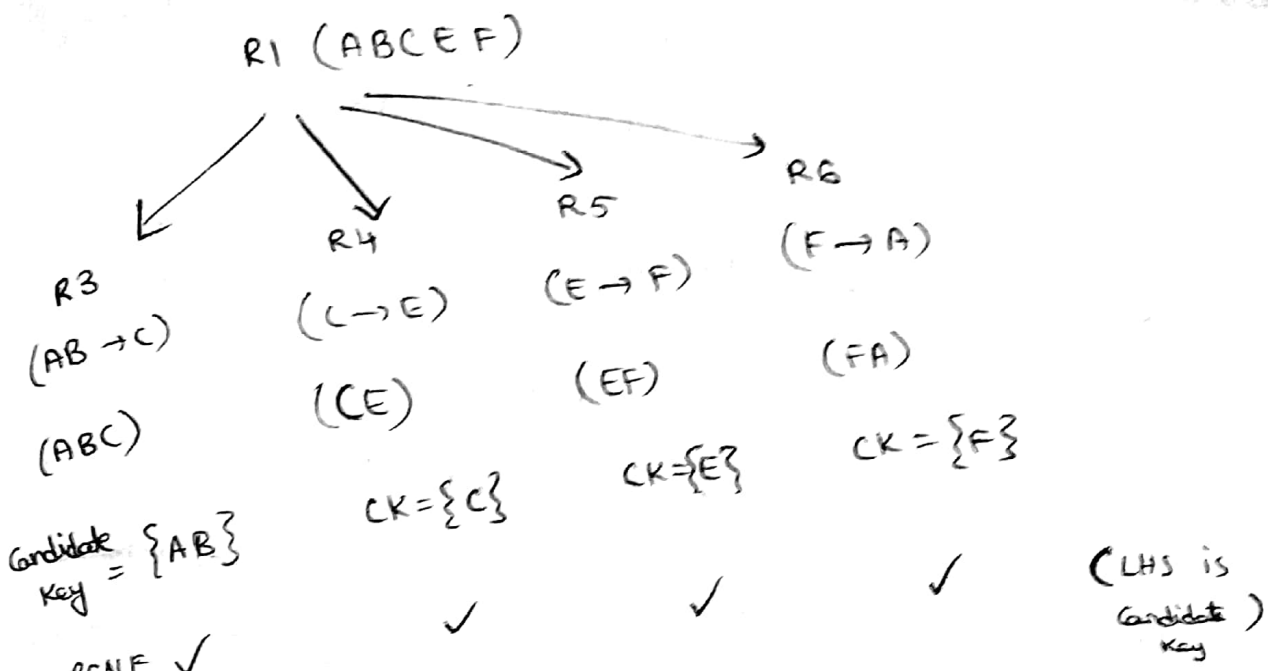
★ LHS must be a Candidate key ⟹ BCNF Satisfied

| $AB \to C$ | $C \to E$ | $E \to F$ | $F \to A$ |
|---|---|---|---|
| LHS: ✓ | LHS: X | LHS: X | LHS: X |
| [redacted] | [redacted] | [redacted] | [redacted] |

R1 (ABCEF)

Not in BCNF, So we need to decompose the Relation R1 (ABCEF).

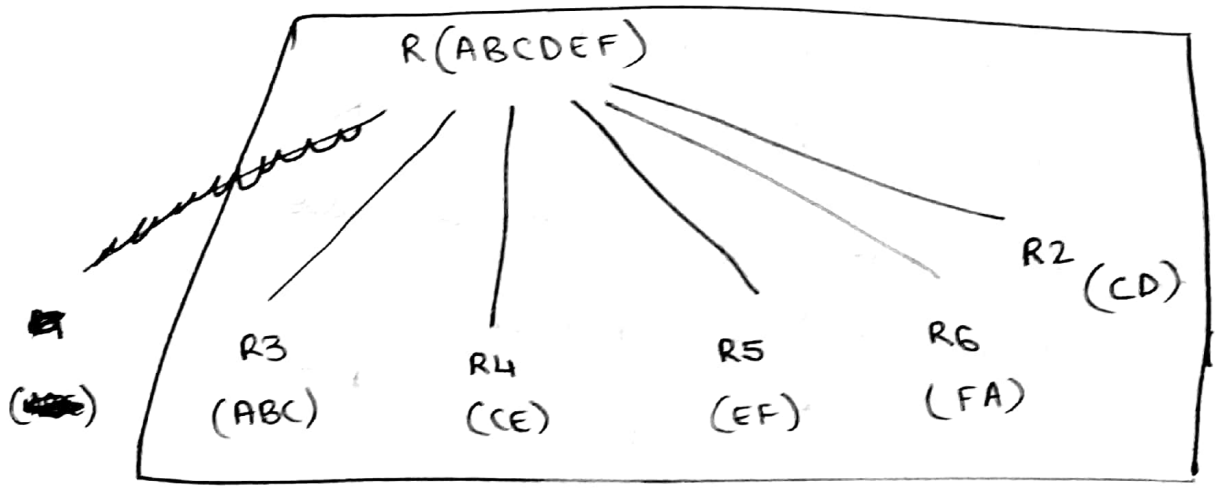
here $C \to E$, $E \to F$, $F \to A$ are not making entire Relation to attain BCNF - So, they have to removed & to be divided into different tables.




C → D  LHS: ✓  Candidate Key = {C}  ] R2 (CD)

So, R2 (CD) is in BCNF ✓

So, R3 (ABC), R4 (CE), R5 (EF), R6 (FA), R2 (CD) are in BCNF.




GATE

Q) Relation R has eight attributes (ABCDEFGH). Given:

CH → G  B → CFH  F → EG

A → BC  E → A  How many Candidate Keys are there in R.

a) 3  b) 4  c) 5  d) 6

**Ans:** A attribute that determines all the attributes in given relation / Minimal super key.

**Trick:** Write all attributes of RHS in Functional dependencies.

G BC CFH A EG ~ A B C E F G H

Missing is D: ⇒ D is must to be present in any Candidate key of relation R as ~~it is determine~~ it is not determined by any of the other attributes.

So, All combinations of D ⇒ Super key of R.

$(D)^* = D \neq R$

So, combinations of D ⇒

$(AD)^* \rightarrow$ ADBC by FD2

ABCD → ABCDFH by FD3

ABCDFH → ABCDEFGH by FD5 = R

∴ AD is super key & minimal also

since $(A)^* = \{ABCEFGH\}$, $(D)^* = \{D\}$ $\Big\} \neq R$

So, Minimal Super key, Candidate Key = {AD}

A D

E D

F D

BD

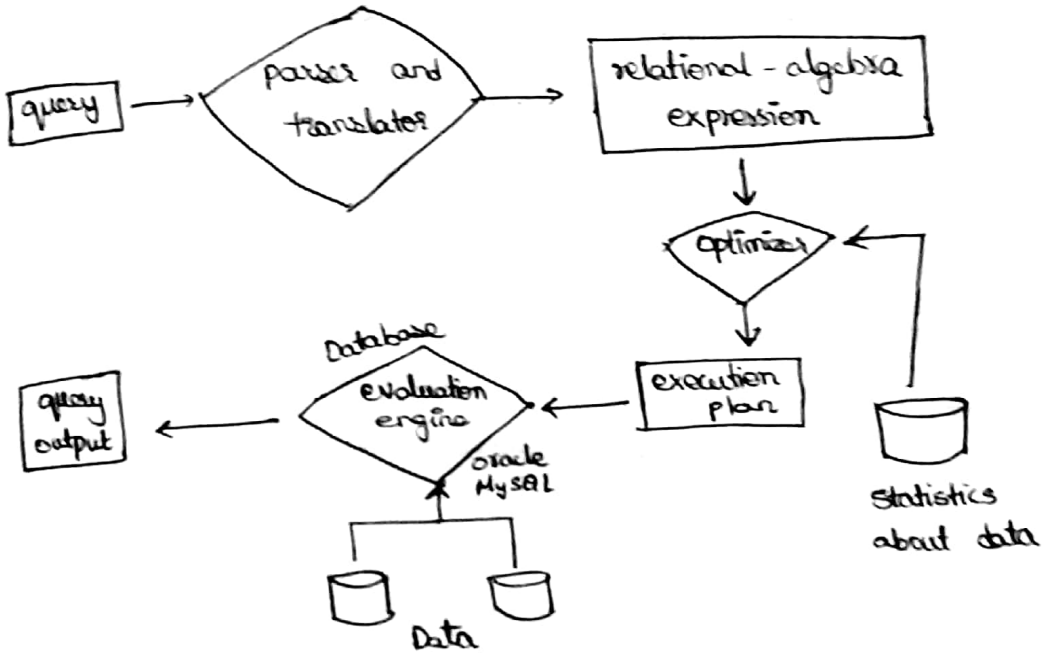
∴ Candidate Keys = {AD, ED, FD, BD}

# MODULE 4 : QUERY PROCESSING

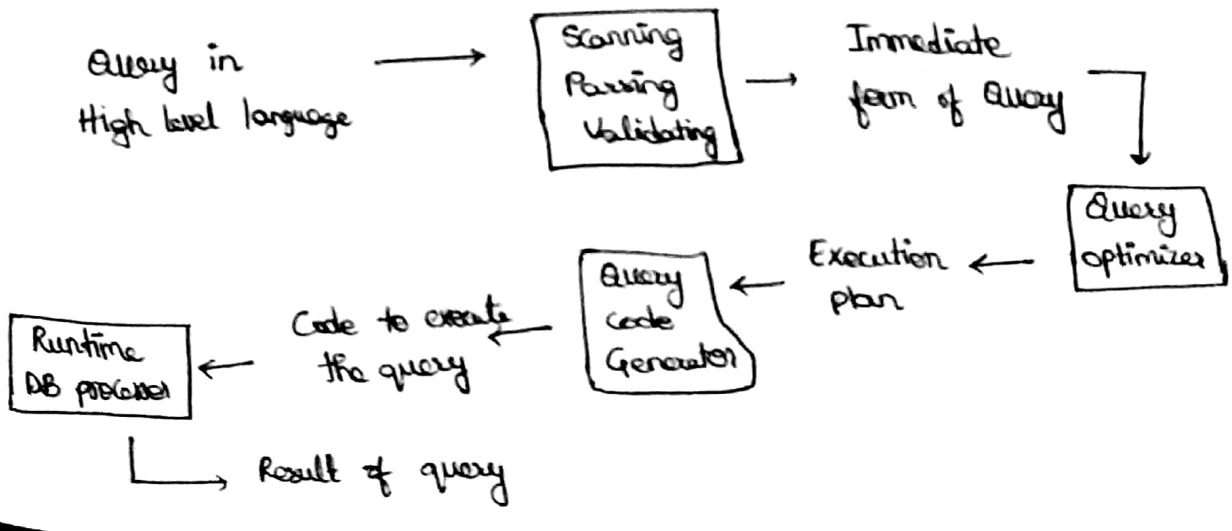
* Introduction :

- First Query optimization - The process of choosing a suitable execution strategy for processing a query.

- Two internal representation of query :
  1. Query Tree
  2. Query Graph




- Query processing is a set of all activities starting from query placement to displaying the results of query.



* There are two main techniques of query optimization.

- The first technique is based on Heuristic Rules for ordering the operations in a query execution strategy that works well in most cases but is not guaranteed to work well in every case.
- The rules typically reorder the operations in a query tree.
- The second technique involves cost estimation of different execution strategies and choosing the execution plan that minimizes estimated cost.

*. QUERY BLOCK :

- The basic unit that can be translated into the algebraic operators and optimization.

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

```
SELECT    LNAME, FNAME
FROM      EMPLOYEE
WHERE     SALARY > C
```

```
SELECT    MAX (SALARY)
FROM      EMPLOYEE
WHERE     DNO = 5
```

$\pi_{LNAME, FNAME}(\sigma_{SALARY > C}(EMPLOYEE))$

$\mathcal{F}_{MAX\,SALARY}(\sigma_{DNO=5}(EMPLOYEE))$

## *. Algorithms for External Sorting :

- **External Sorting :**

→ Refers to sorting algorithms that are suitable for large files of records stored on disk that don't fit entirely in main memory, such as most database files.

- **Sort - Merge Strategy :**

→ Starts by sorting small subfiles (runs) of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn.

→ Sorting phase : $n_R = \left\lceil \left( \frac{b}{n_B} \right) \right\rceil$

- **Merging phase :**

$$d_M = \text{Min}\ (n_B - 1,\ n_R)$$

$$n_P = \left\lceil (\log_{d_M} (n_R)) \right\rceil$$

where,

$n_R$ - no. of initial runs

$b$ - no. of file blocks

$n_B$ - available buffer space

$d_M$ - degree of merging

$n_P$ - no. of passes.

## *. JOIN operations :

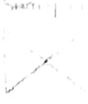
① Semi-join is generally used for unnesting EXISTS, IN, and ANY subqueries

EMPLOYEE (Ssn, Bdate, Address, Sex, Salary, Dno)
DEPARTMENT (Dnumber, Dname, Dmgrssn, Zipcode)

```
Q(SJ): SELECT COUNT(*)
FROM   DEPARTMENT D
WHERE  D.Dnumber IN (SELECT E.Dno
                     FROM   EMPLOYEE E
                     WHERE  E.Salary > 200000)
```

```
(SELECT E.Dno
 FROM   EMPLOYEE E
 WHERE  E.Salary > 200000)
```

```
SELECT COUNT(*)
FROM   EMPLOYEE E, DEPARTMENT D
WHERE  D.Dnumber = E.Dno and E.Salary > 200000;
```



Anti-join is generally used for witnessing NOT IN subqueries

Q4B: SELECT COUNT(*)
FROM EMPLOYEE
WHERE EMPLOYEE.Dno NOT IN (SELECT DEPARTMENT.Dnumber
FROM DEPARTMENT
WHERE Zipcode = '30332');

↓

SELECT COUNT(*)
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.Dno = DEPARTMENT.Dnumber AND Zipcode = '30332';

Ex:

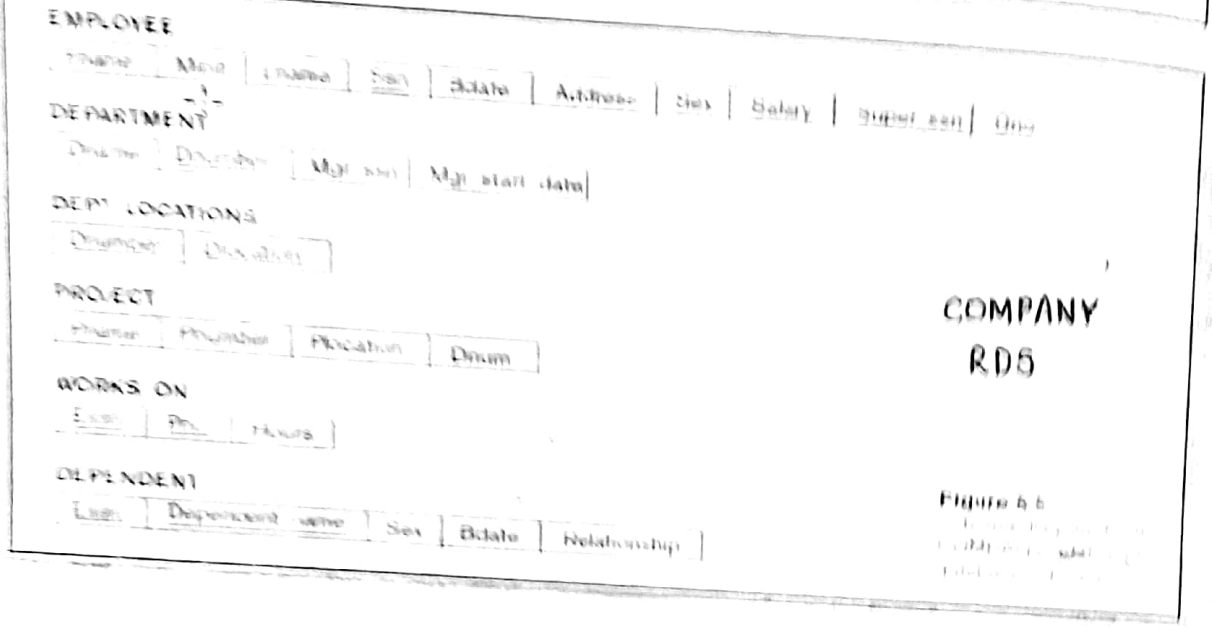



Figure 6.5

- There are many algorithms for executing a SELECT operation, which is basically a search operation to locate the records in a disk file that satisfy a certain condition.

Ex: OP1 : $\sigma_{SSN = '123456789'}$ (EMPLOYEE)

OP2 : $\sigma_{DNUMBER > 5}$ (DEPARTMENT)

OP3 : $\sigma_{DNO = 5}$ (EMPLOYEE)

OP4 : $\sigma_{DNO = 5 \text{ AND } SALARY > 3000 \text{ AND } SEX = 'F'}$ (EMPLOYEE)

OP5 : $\sigma_{ESSN = '123456789' \text{ AND } PNO = 10}$ (WORKS_ON)

## ALGORITHMS : (to implement SELECT operation)

S1: Linear Search (Brute Force) :

- Retrieve every record in the file, and test whether its attribute values satisfy the selection condition.

S2: Binary Search :

- If the selection condition involves an equality comparision on a key attribute on which the file is ordered, binary search (which is more efficient than linear search) can be used. (OP1)

S3: Using a primary index or hash key to retrieve a single record :

- If the selection condition involves an equality comparision on a key attribute with a primary index (or a hash key), use the primary index (or the hash key) to retrieve the record.

S4: Using a primary index to retrieve multiple records :

- If the comparision condition is $>$, $\geq$, $<$ or $\leq$ on a key field with a primary index, use the index to find the record satisfying the corresponding equality condition, then retrieve all subsequent records in the (ordered) file.

Ex: Dnumber $> 5$ in OP2 - use the index to find the record satisfying the corresponding equality condition (Dnumber $= 5$); then retrieve all subsequent records in the (ordered) file. For Dnumber $< 5$, retrieve all preceding records.

S9 : Conjuctive selection using a composite index :

- If two or more attributes are involved in equality conditions in the conjuctive select condition and a composite index (or hash structure) exists on the combined field.

Ex: If an index has been created on the composite key (ESSN, PNO) of the WORKS_ON file for OP5 - we can use the index directly.

S10 : Conjuctive selection by intersection of record pointers

- This method is possible if secondary indexes are available on all (or some of) the fields involved in equality comparision conditions in the conjuctive condition and if the indexes include record pointers (rather than block pointers).

*. SEARCH METHODS for DISJUNCTIVE SELECTION :

- Compared to a conjuctive selection condition, a disjunctive condition (where simple conditions are connected by the OR logical connective rather than by AND) is much harder to process & optimize.

- If any of one conditions doesn't have an access path we are compelled to use the brute force, linear search approach.

## *. IMPLEMENTING THE JOIN OPERATION :

- There are many possible ways to implement a two-way join, which is a join on two files, Joins involving more than two files are called multiway joins.
- The no. of possible ways to execute multiways joins grows rapidly because of the combinatorial explosion of possible join orderings.
- Join (EQUI JOIN, NATURAL JOIN)
  - + Two-way join : a join on two files.

    Ex : $R \bowtie_{A=B} S$
  - + Multi-way joins : joins involving more than two files.

    Ex : $R \bowtie_{A=B} S \bowtie_{C=D} T$

## *. METHODS for implementing joins :

J1 : Nested-loop join (brute force).

J2 : Single-loop join (using an access structure to retrieve the matching records).

J3 : Sort-merge join.

J4 : Hash-join.

## *. ALGORITHM for PROJECT operations :

$\Pi_{<\text{attribute list}>}(R)$ .

1. If < attribute list > has a key of relation R, extract all tuples from R with only the values of the attributes in < attribute list > .

2. If < attribute list > doesn't include a key of relation R, duplicated tuples must be removed from the results.

- Methods to remove duplicate tuples :

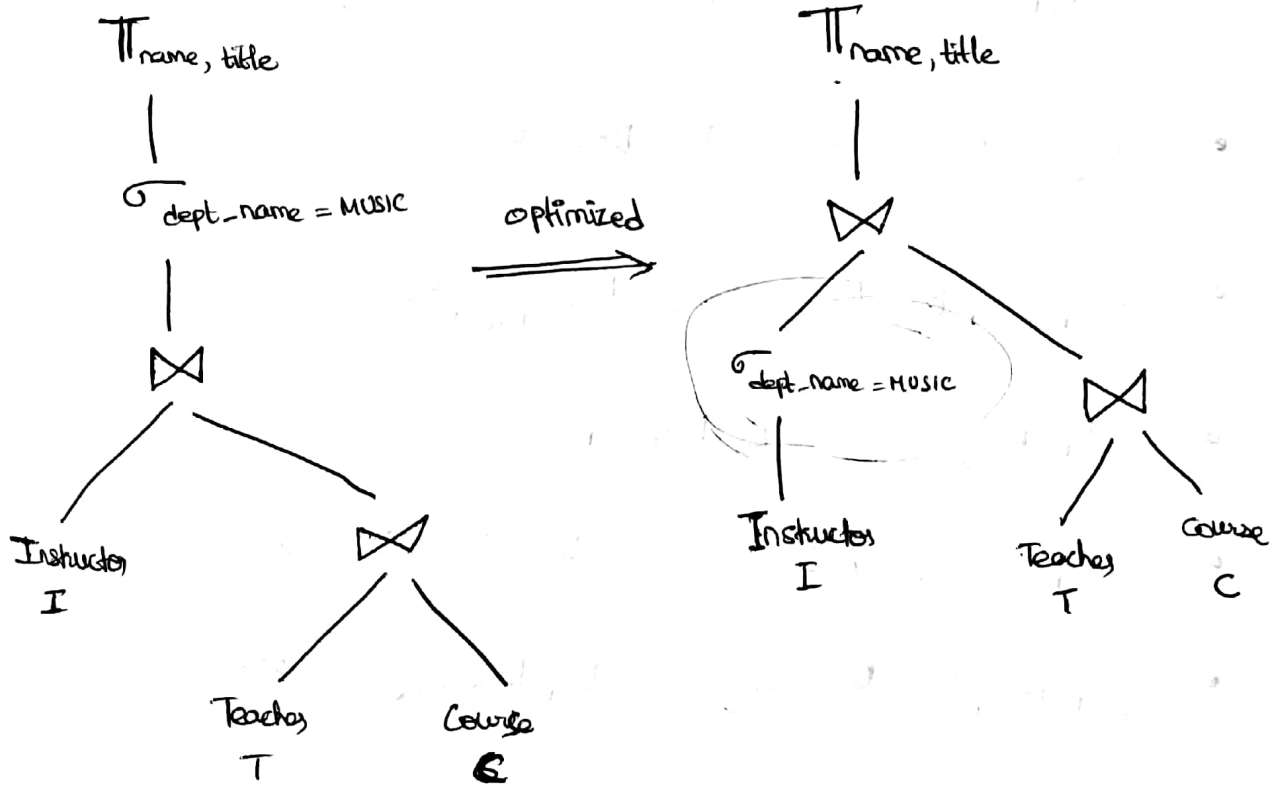
  1. Sorting  2. Hashing

---

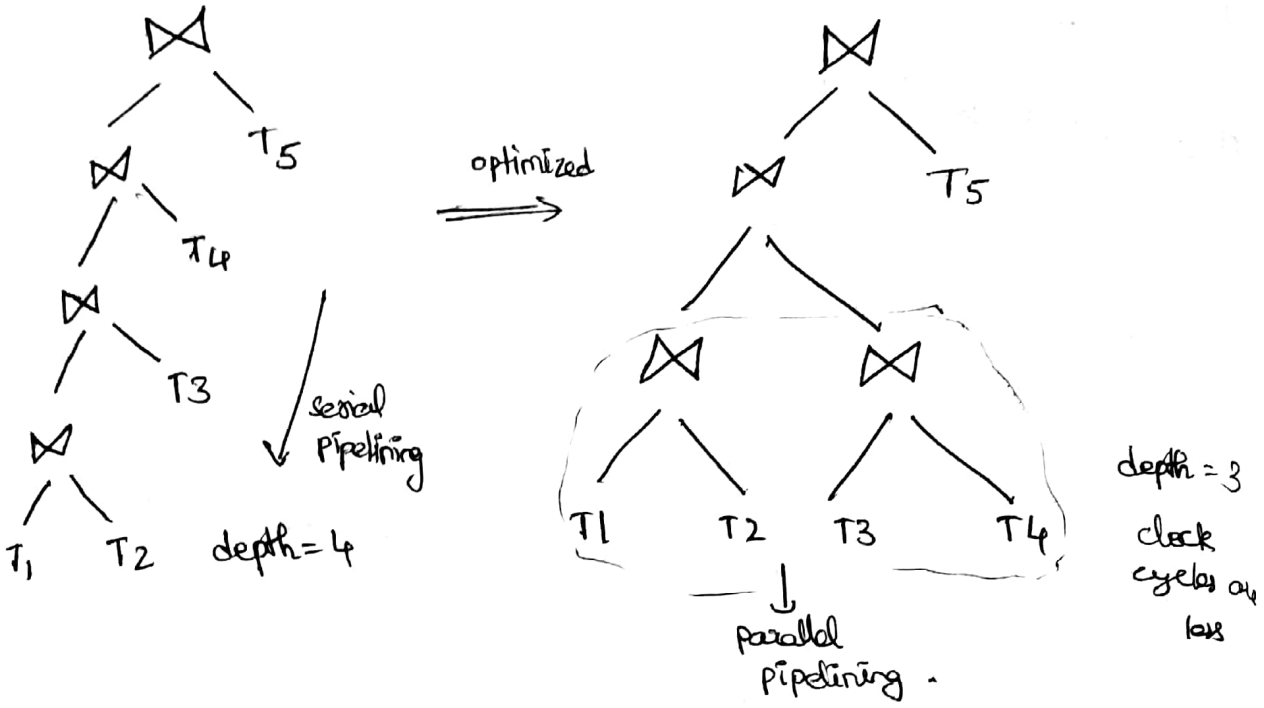
## *. QUERY OPTIMIZATION :

$$\Pi_{name, title}\left(\sigma_{dept\_name = MUSIC}\left(I \bowtie (T \bowtie C)\right)\right)$$

(1)



(2)




✱- Equivallance Rules :

- $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
- $\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
- $\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$
- $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$
- $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
- $E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$
- $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
- $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_2} (E_2 \bowtie_{\theta_2} E_3)$

- $\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\pi_{L_1}(E_1)) \bowtie_\theta (\pi_{L_2}(E_2))$
- $\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\pi_{L_1 \cup L_2}(\pi_{L_1 \cup L_3}(E_1))) \bowtie_\theta (\pi_{L_2 \cup L_4}(E_2))$

## * HEURISTIC OPTIMIZATION :

- Heuristic optimization transforms the Query-Tree by using a set of rules that typically (but not in all cases) improve execution performance :

i. Perform Selection early.

ii. Perform Projection early.

iii. Perform most restrictive selection & join operations.

iv. Combine heuristic with partial cost-based optimization.

- Apply select ($\sigma$) operation before Project ($\pi$).
- Apply Project ($\pi$) before join ($\bowtie$, $\times$).

Example : Consider the DB table :

EMPLOYEE ( Fname, Mint, Lname, SSN, Bdate, Address, Sex, Salary)

WORKS_ON ( ESSN, PNo, Hours)

PROJECT ( Pname, Pnumber, Plocation)

SQL :

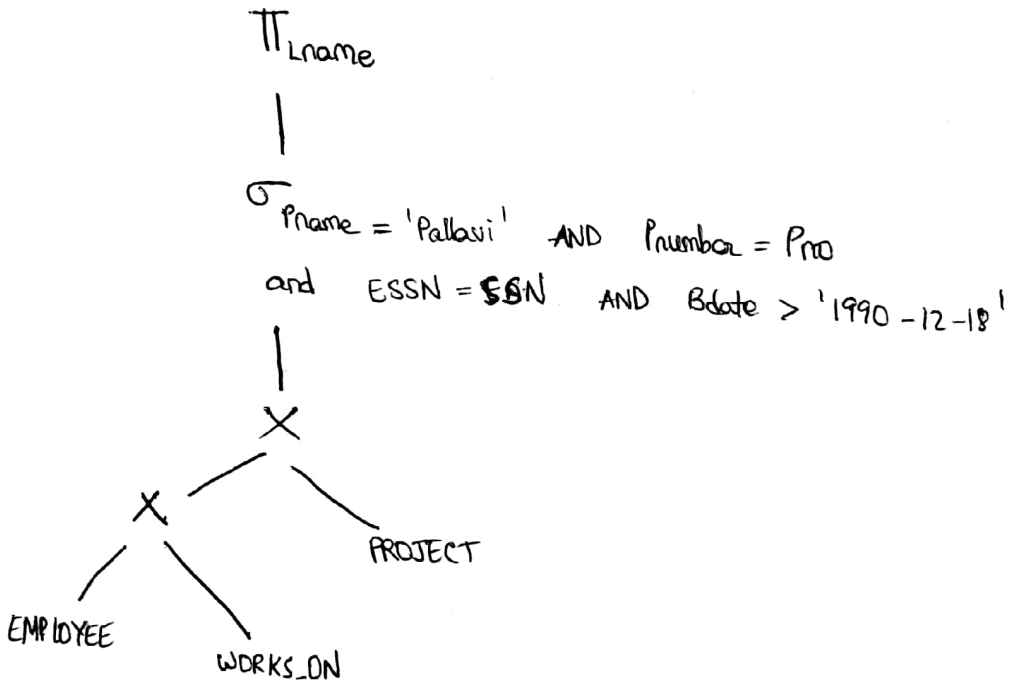
```
SELECT Lname FROM Employee, Works_ON, Project
WHERE Pname = 'Pallavi' & Pnumber = Pno AND
ESSN = SSN and Bdate > '1990-12-18';
```

Relational Algebra :

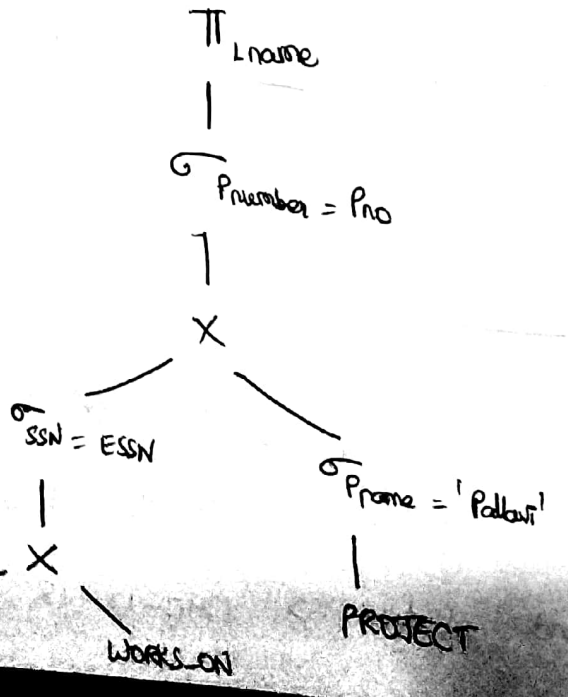
$$\Pi_{Lname}\left(\sigma_{Pname='Pallavi'} \text{ AND } (EMPLOYEE \times WORKS\_ON \times PROJECT)\right.$$

$$ESSN = SSN \text{ AND } Pnumber = Pno \text{ AND } Bdate > '1990-12-18'\Big)$$

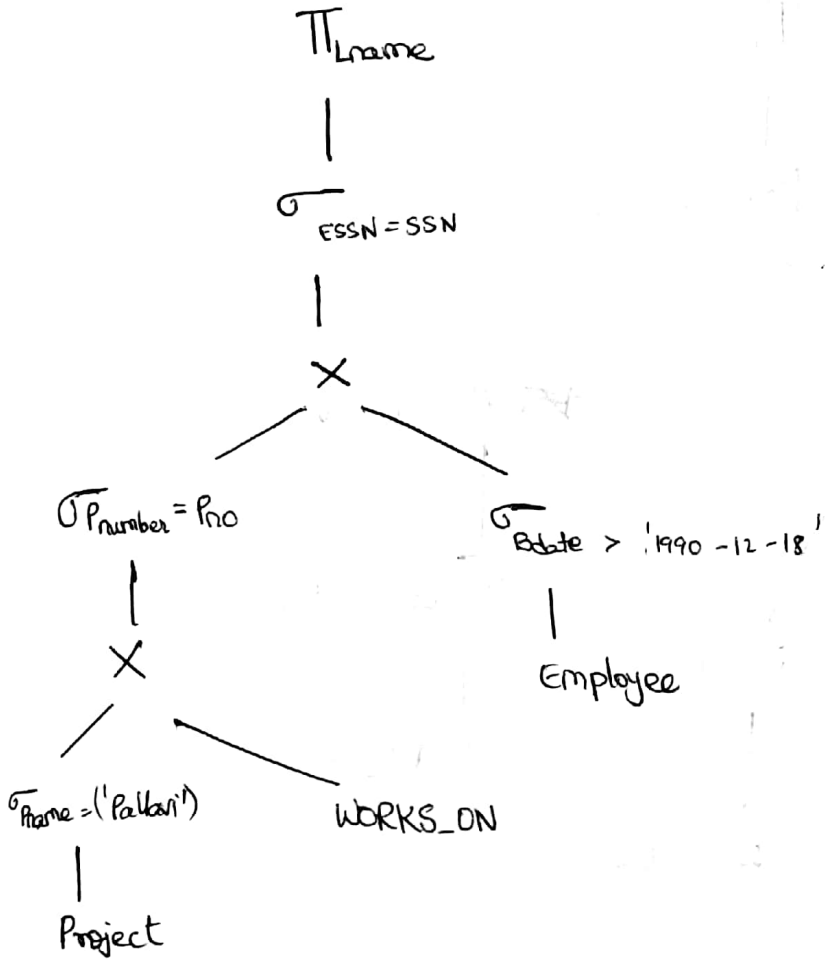
i. Build the initial Query tree,



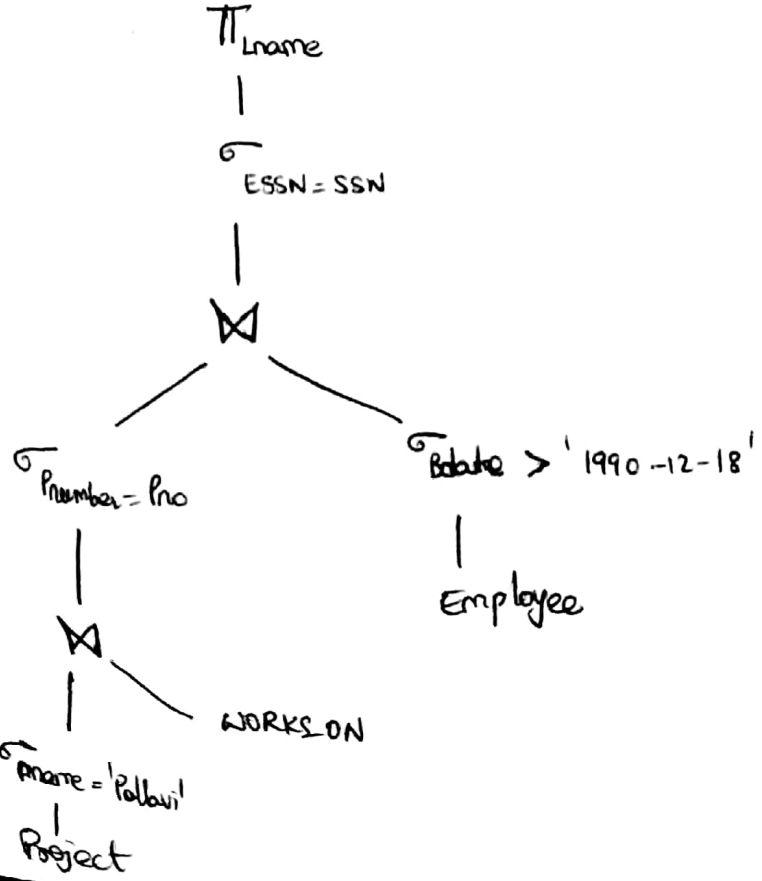

ii. Moving SELECT (σ) operation down the Query tree



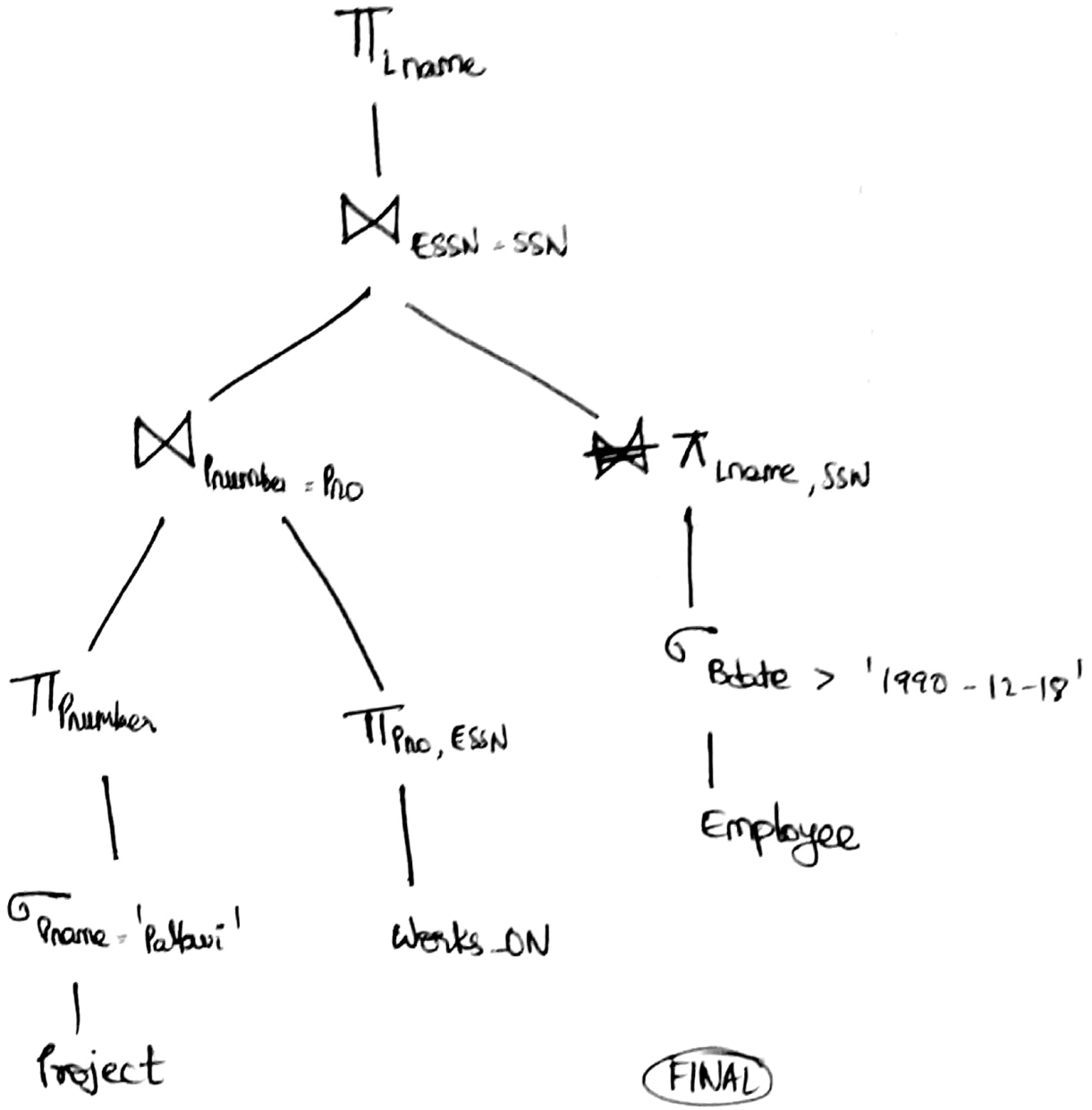
iii. Applying more restrictive selection operations first




iv. optimized query : ( Replace Cartesian Product with Cross Join )



v. Moving Project ($\pi$) operation down Query Tree:




FINAL

# Module 5 : TRANSACTION PROCESSING

* TRANSACTION : It is a set of operations used to perform a logical unit of work.

- A transaction generally represent change in database.
- R(A) ; Transfer of data read from A to B to access
- W(A) : change of data from A.

Ex: Suppose A = 1000 B = 2000. The performed transaction operations are :

1. R(A) = 1000
2. A = A - 500
3. W(A) = 500
4. R(B) = 2000
5. B = B + 500
6. W(B) = 500

Changes are made in RAM only not in the original DB

original DB
A = 1000
B = 2000

RAM
A = 500
B = 2500

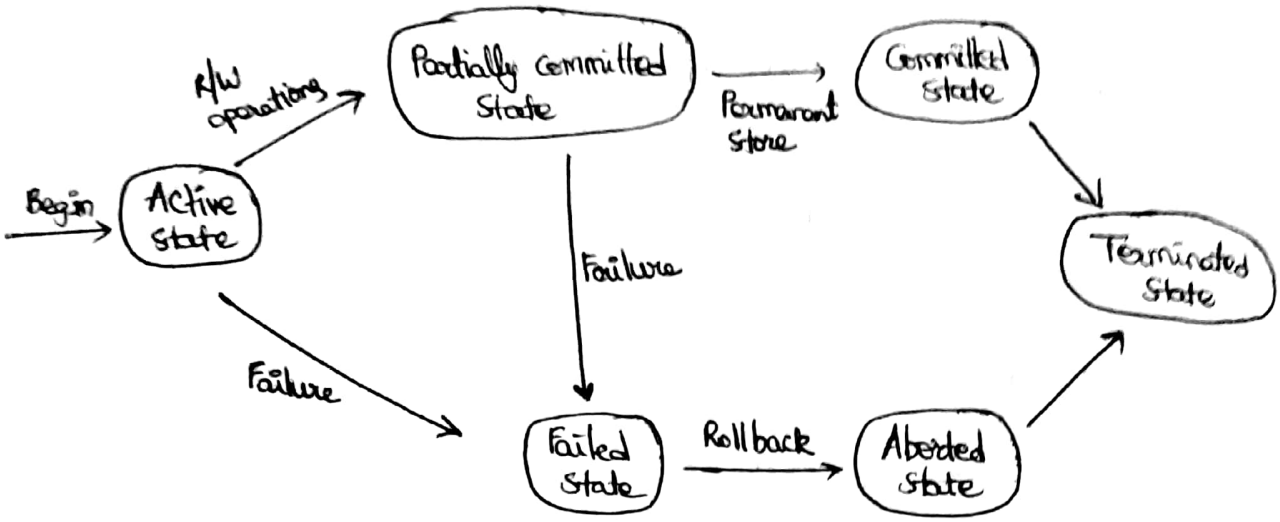
——— Transfer of Rs. 500/- from A to B ———

Now to change the current in database, then we need to type `commit ;` ⇒ Now in DB also its updated.

* ACID Properties of Transaction :

A - ~~Accuracy~~ Atomicity
C - Consistency
I - Isolation
D - Durability

*. Lifecycle of Transaction is :




1. Atomicity : either transaction occurs completely or it doesnot occur at all. (All or nothing rule).

2. Consistency : ensures that database remains consistent before and after the transaction.

3. Isolation : multiple transactions can occur without causing any inconsistency, (occurs simultaneously).

4. Durability : All the changes made by a transaction after its successful execution are written successfully to disk.

*. SCHEDULE :

- It is the chronological execution sequence of multiple transactions. There are two types of schedules

Serial

Parallel

1. Serial : one after the other & only after completing current one, the next one starts. (ATM)

2. Parallel : multiple transactions at same time.
Throughput - No of transactions / time (online Payment Banks)

## *. READ-WRITE CONFLICTS :

| Same Data | |
|---|---|
| R(A) | R(A) — no Conflict |
| R(A) | W(A) |
| W(A) | W(A) } Problem exists |
| W(A) | R(A) |

2 users

Suppose A = 2

| User 1 | User 2 |
|---|---|
| T1 | T2 |
| 2 seats left R(A) | |
| | R(A) 2 seats left |
| | A = A-2 |
| | W(A) |
| | Commit 0 seats left |
| 0 seats left R(A) | |
| W(A) | |
| Commit | |

## *. RECOVERABILITY :

- Irrecoverable schedule : ability to not getting recovered due to failure or improper abort.

Ex: Consider A = 10 & following schedule:

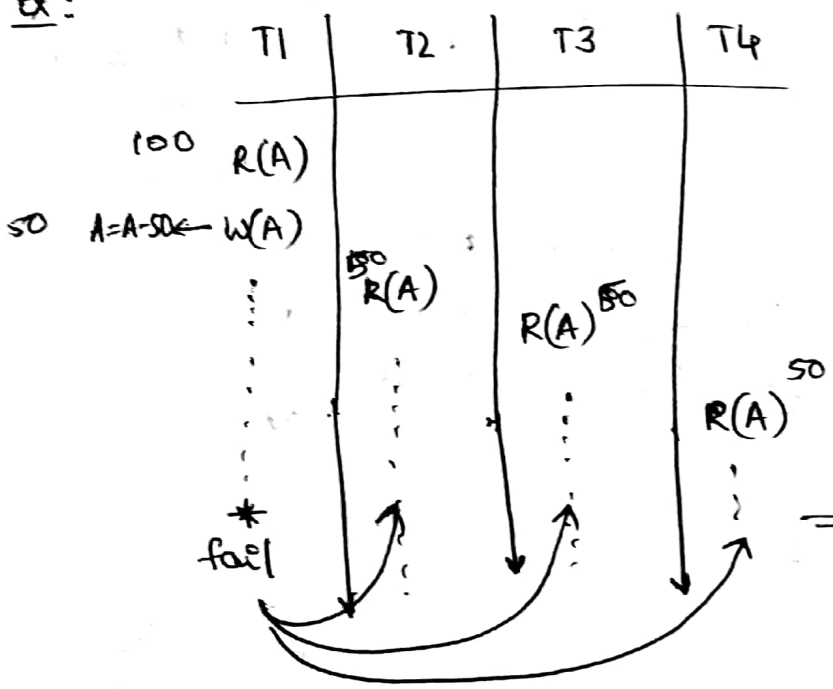
| | T1 | T2 | |
|---|---|---|---|
| 10 | R(A) | | |
| 5 | A = A-5 | | |
| 5 | W(A) | | |
| | : | R(A) | 5 |
| | : | A = A-2 | 3 |
| 3 fail | * R(B) | W(A) | 3 |
| | | commit | |

If it fails, the whole transactions will be undo / rollback. So, due to fail of T1, it will undo entire transactions operation & rollbacks to 1st point. $\Rightarrow$ $\boxed{A = 10}$ is updated

But T2 has performed something & it is not getting that $\Rightarrow$ the schedule is not recoverable.

## *. Cascading Schedule (vs) Cascadeless Schedule :

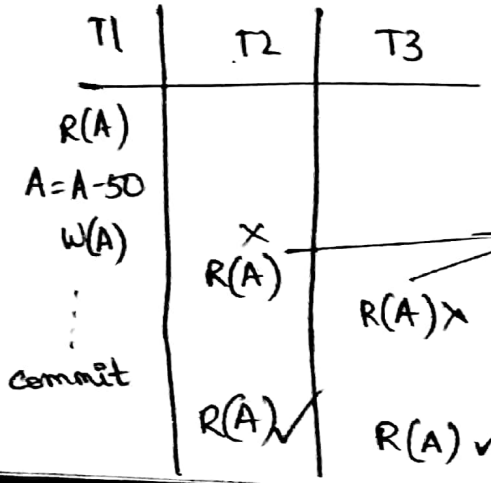
Ex:




Suppose $\boxed{A = 100}$

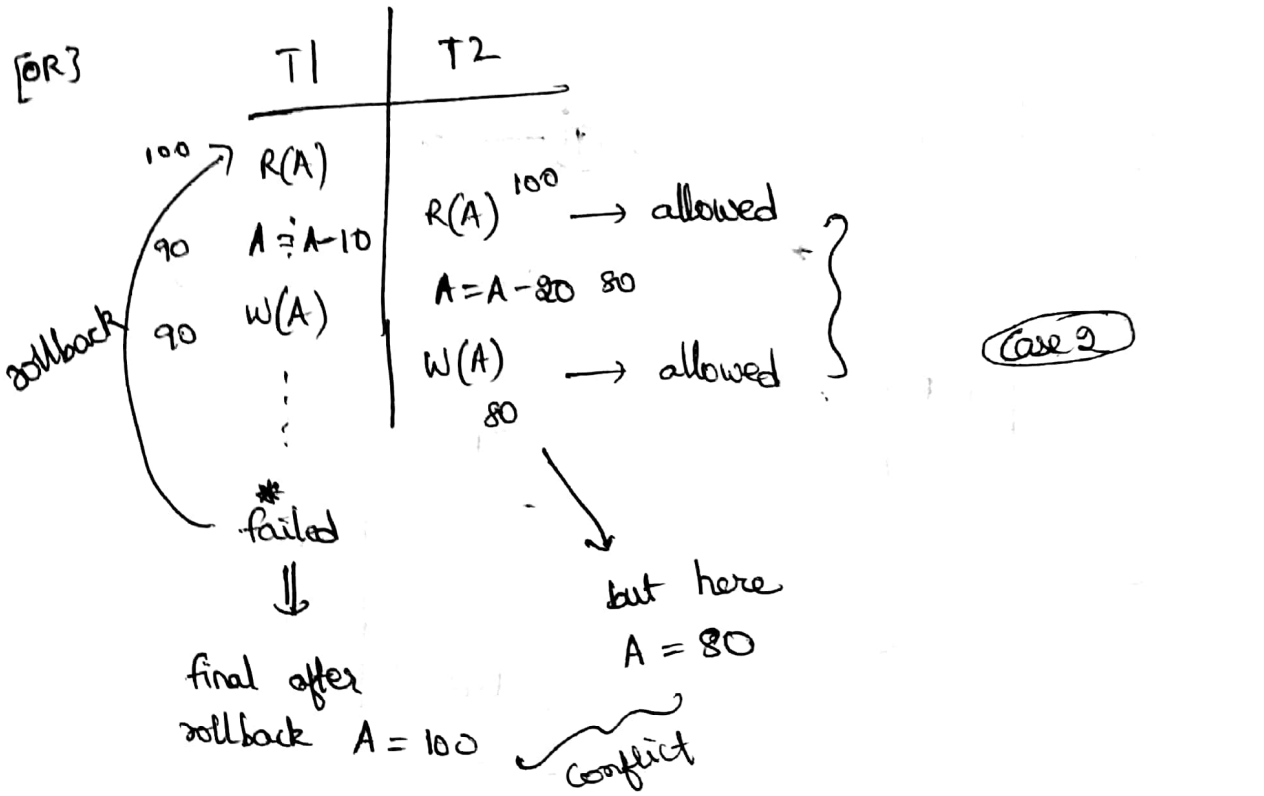
$\Longrightarrow$ If 1 fails then remaining all are forcefully rolled back

- CPU utilization is wasted / degraded.
- This is called Cascading Schedule.




Case 1

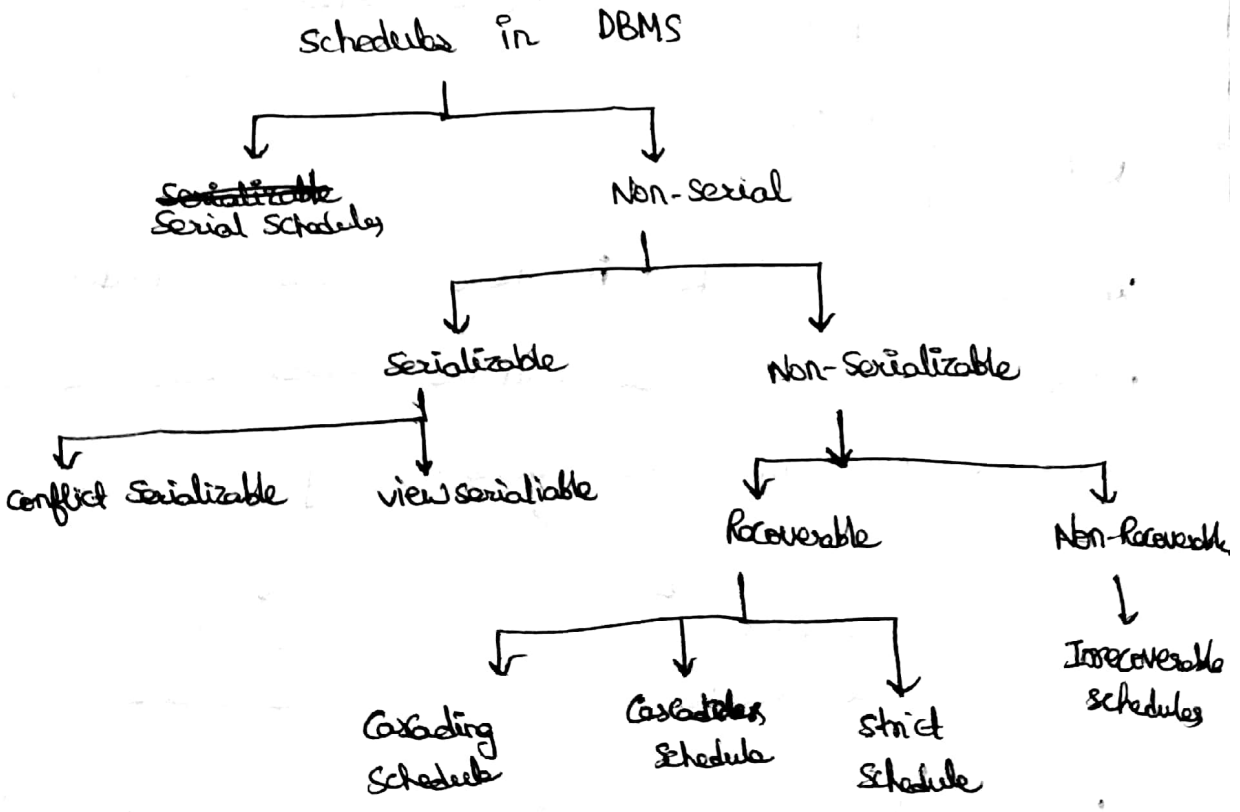
Not allowed until T1 is committed $\Longrightarrow$ other than R(A), remaining are allowed like R(B), R(C) W(A).




- This is called (2 cases) are called Cascadeless Schedule.

IMP
*. SERIALIZABILITY :

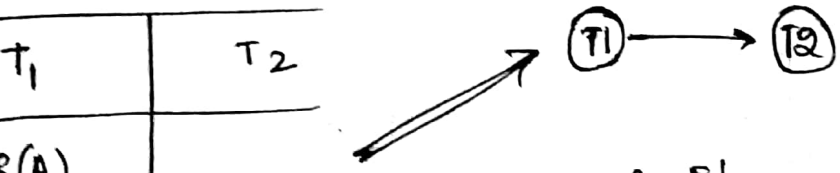
- In DBMS, schedules may be classified as:



Schedule S

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |

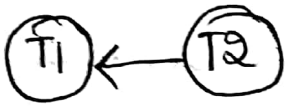
Serial Schedule




Consider another S

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

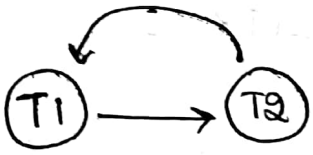
Serial Schedule




Now, Parallel Schedules

Schedule S

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




Parallel Schedules forms closed | Circular graph

We need to check if these Parallel Schedules equivalents exists that will be serial schedules or not. we call this as "Serializability."

- Conflict Serializable.
- View Serializable

Parallel schedule equivalent Serial Schedule is to be found.

Schedule S

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

Possible serial schedules for any 3 transactions is,

T1 → T2 → T3

T1 → T3 → T2

T2 → T3 → T1

T2 → T1 → T3 (3!)

T3 → T1 → T2

T3 → T2 → T1

*. CONFLICT EQUIVALENT :

R(A) R(A) } Non conflict pairs

R(A) W(A)
W(A) R(A)
W(A) W(A)
} conflict pair (problem here)

R(B) R(A)
W(B) R(A)
R(B) W(A)
W(A) W(B)
} Non-conflict pairs

Consider another schedule S'

S'

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

Consider a schedule S

S

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

- Adjacent Non-conflict pairs are to be swapped.
- No change in positions for a conflict pairs.

S' schedule

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) (non conflict) |
| R(B) | |
| | W(A) |

⟹

S'

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

This S' schedule is now equivalent to S schedule.

So, S ≡ S' are conflict equivalent schedules.

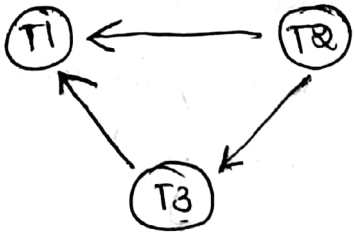
- **PRECEDENCE GRAPH :**

Consider a schedule S,

| T1 | T2 | T3 |
|---|---|---|
| R(x) | | |
| | | R(y) |
| | | R(x) |
| | R(y) | |
| | R(z) | |
| | | W(Y) |
| | W(z) | |
| R(z) | | |
| W(x) | | |
| W(z) | | |

- check the conflict pairs in other transactions and draw the edges

Precedence Graph




Loop/cycle - Not there → Conflict Serializable Schedule → Consistent

No

yes → view Serializable

indegree = 0

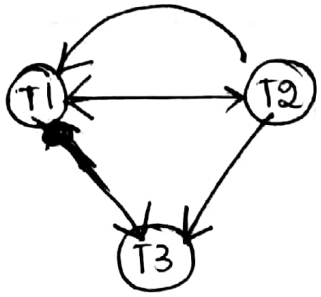
↓

T2 → T3 → T1

(sequence)

Consider A Schedule S,

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



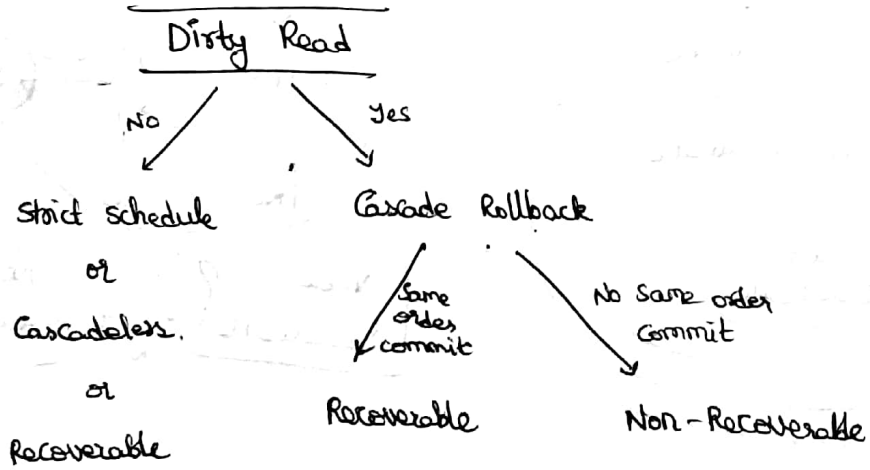

There is loop between T1 & T2

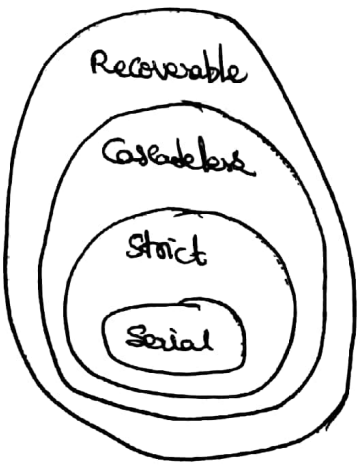
Non conflict ← Serializable

Both are view equivalent.

*. CONCURRENCY CONTROL :

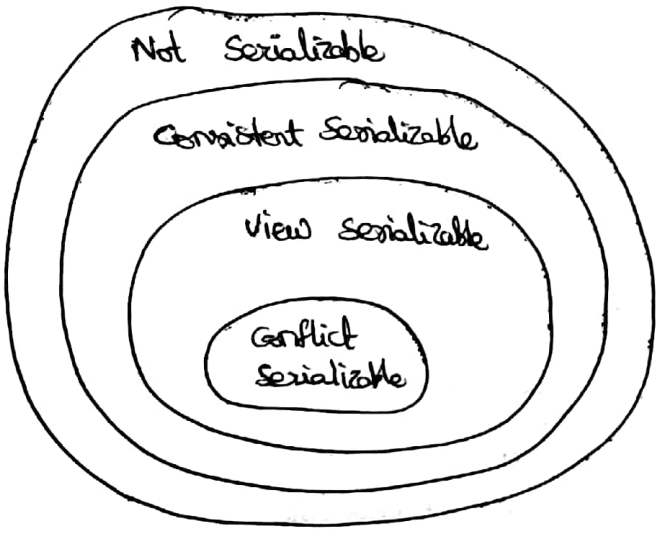
*. Characterizing Schedules :

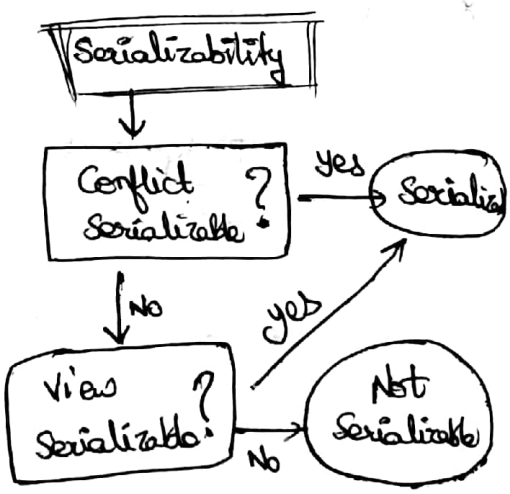


*. on a short note :

- Recoverable schedules : Recovery is possible
- Non-Recoverable schedules shouldnot be permitted by DBMS.
- No committed transactions ever needs to be rolled back
- Cascading rollback may occur in some recoverable schedules.
- Uncommitted transactions may need to be rolled back.
- Cascadeless schedule : Avoids cascading rollback.
- Strict schedule : More strict nature of parallel processing.



Module 5 : # TRANSACTION PROCESSING

* **TRANSACTION** : It is a set of operations used to perform a logical unit of work.

- A transaction generally represent change in database.
- R(A) ; Transfer of data read from A to B to access
- W(A) : change of data from A.

Ex: Suppose A = 1000 B = 2000. The performed transaction operations are :

1. R(A) = 1000
2. A = A - 500
3. W(A) = 500
4. R(B) = 2000
5. B = B + 500
6. W(B) = 500

Changes are made in RAM only not in the original DB

original DB
A = 1000
B = 2000

RAM
A = 500
B = 2500

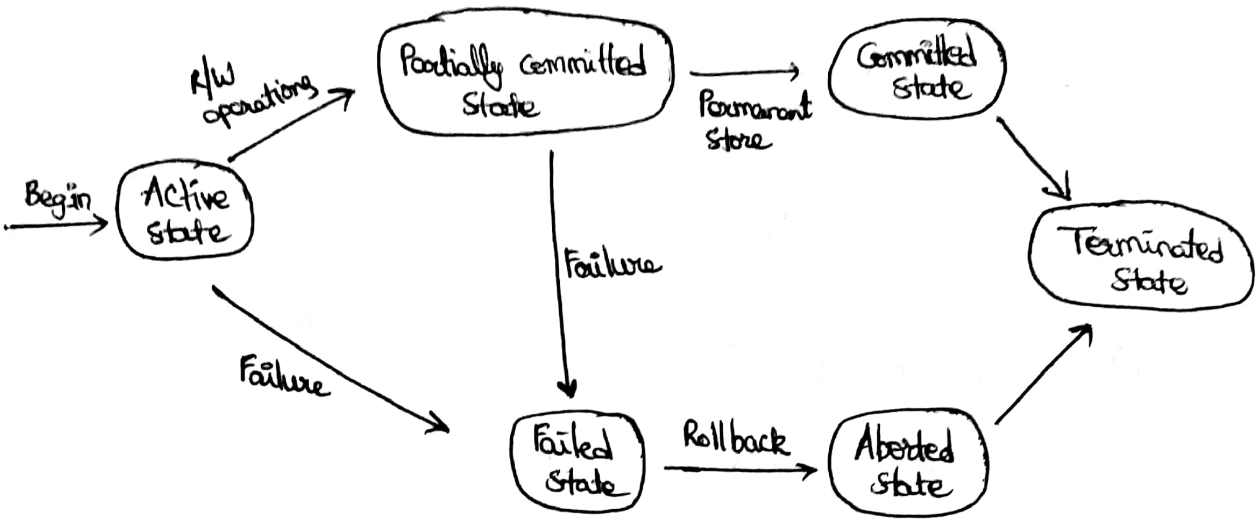
——— Transfer of Rs.500/- from A to B ———

Now to change the current in database, then we need to type `commit ;` ⇒ Now in DB also its updated.

*. ACID Properties of Transaction :

A - ~~Accuracy~~ Atomicity
C - Consistency
I - Isolation
D - Durability

*. Lifecycle of Transaction is :




1. Atomicity : either transaction occurs completely or it doesnot occur at all. (All or nothing rule).

2. Consistency : ensures that database remains consistent before and after the transaction.

3. Isolation : multiple transactions can occur without causing any inconsistency, (occurs simultaneously).

4. Durability : All the changes made by a transaction after its successful execution are written successfully to disk.

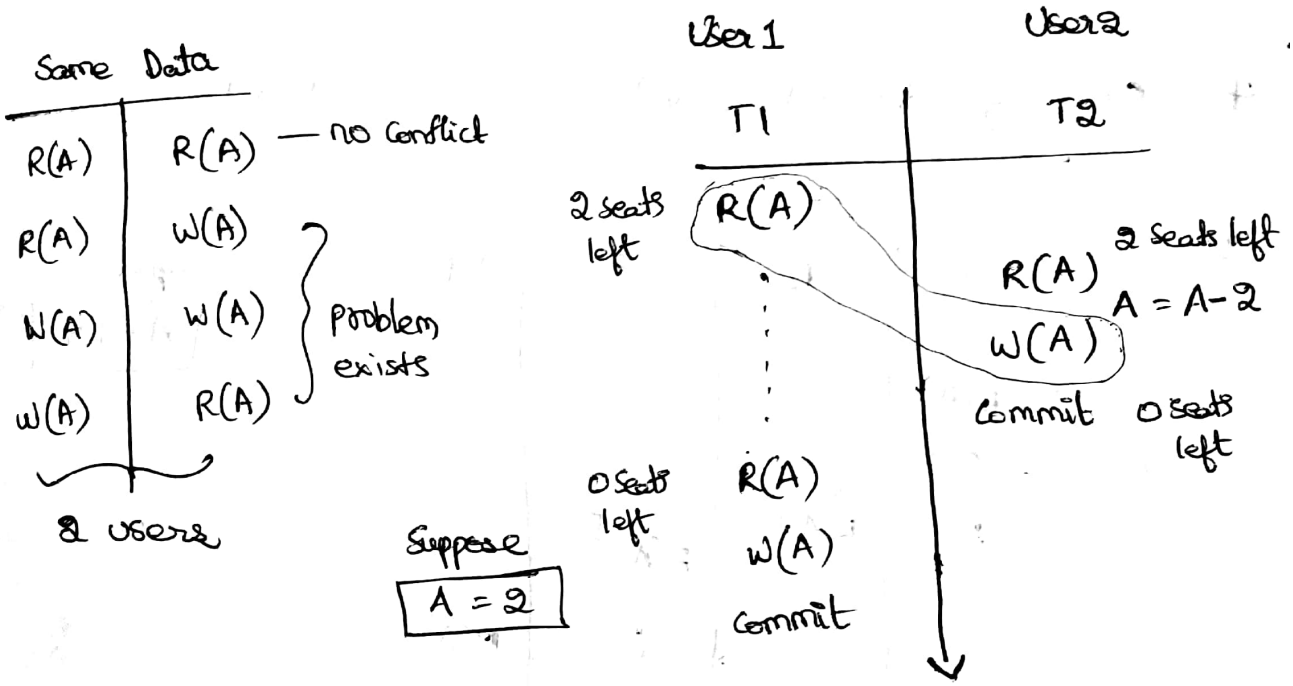
*. SCHEDULE :

- It is the chronological execution sequence of multiple transactions. There are two types of schedules

1. Serial : one after the other & only after completing current one, the next one starts. (ATM)

2. Parallel : multiple transactions at same time.
Throughput - No of transactions / time (online Payment Banks)

## *. READ-WRITE CONFLICTS :




## *. RECOVERABILITY :

- Irrecoverable schedule : ability to not getting recovered due to failure or improper abort.

Ex: Consider A = 10 & following schedule:

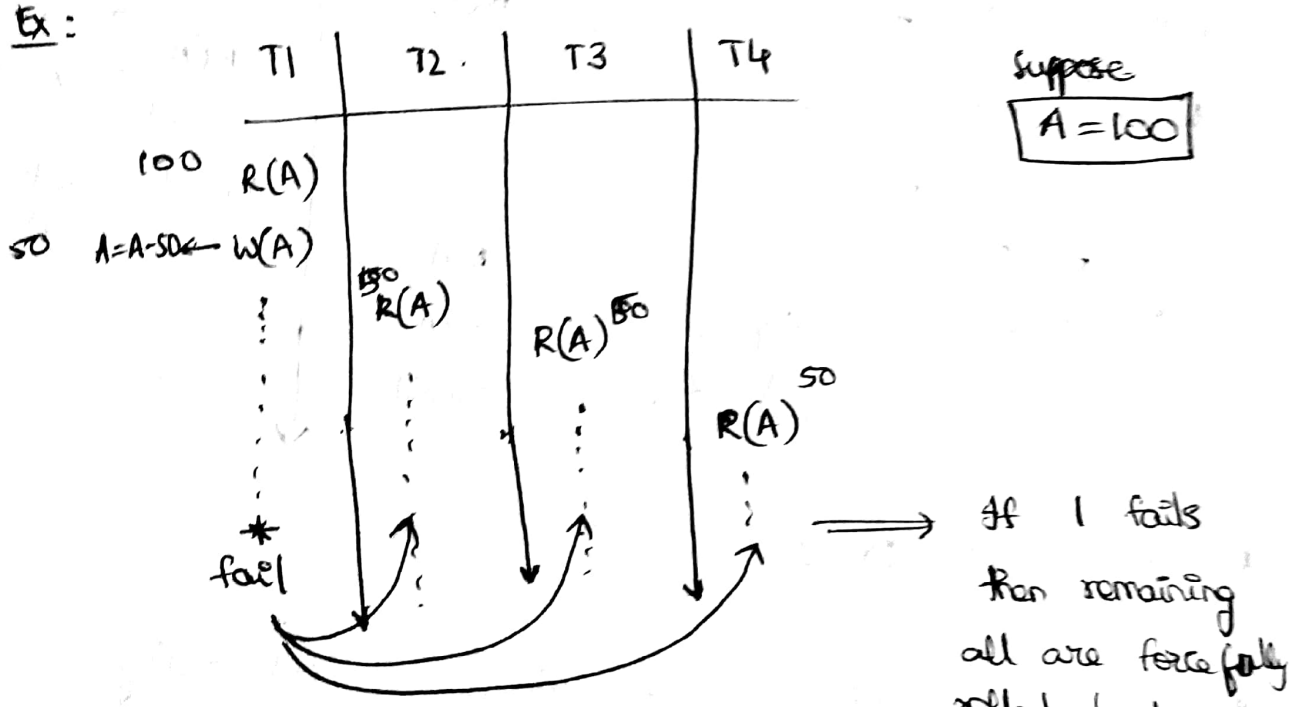
| | T1 | T2 | |
|---|---|---|---|
| 10 | R(A) | | |
| 5 | A = A-5 | | |
| 5 | W(A) | | |
| | : | R(A) | 5 |
| | : | A = A-2 | |
| 3 fail * | R(B) | W(A) | 3 |
| | | commit | 3 |

If it fails, the whole transactions will be undo / rollback. So, due to fail of T1, it will undo entire transactions operation & rollbacks to 1st point. ⟹ A = 10 is updated

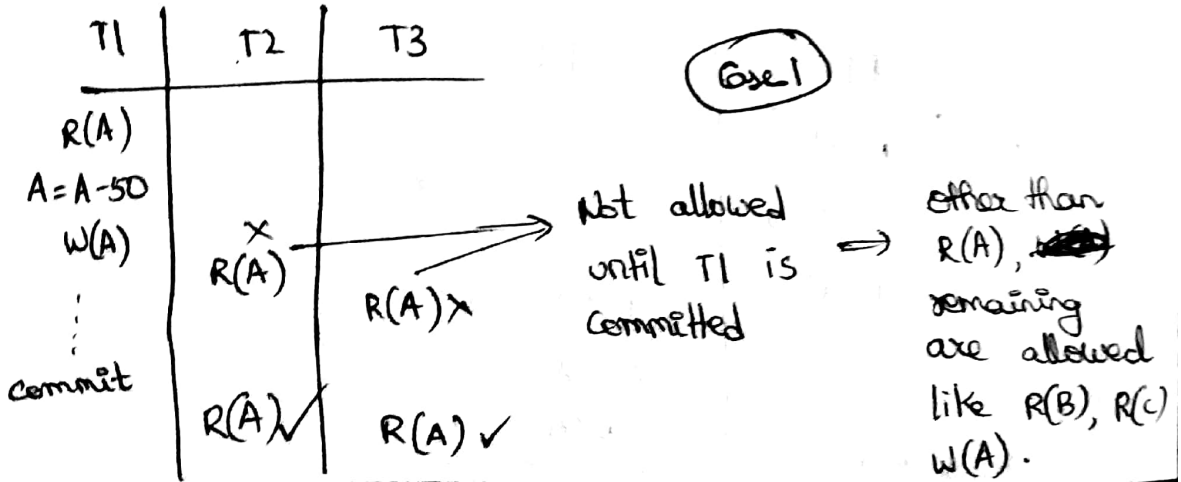
But T2 has performed something & it is not getting that ⟹ the schedule is not recoverable.

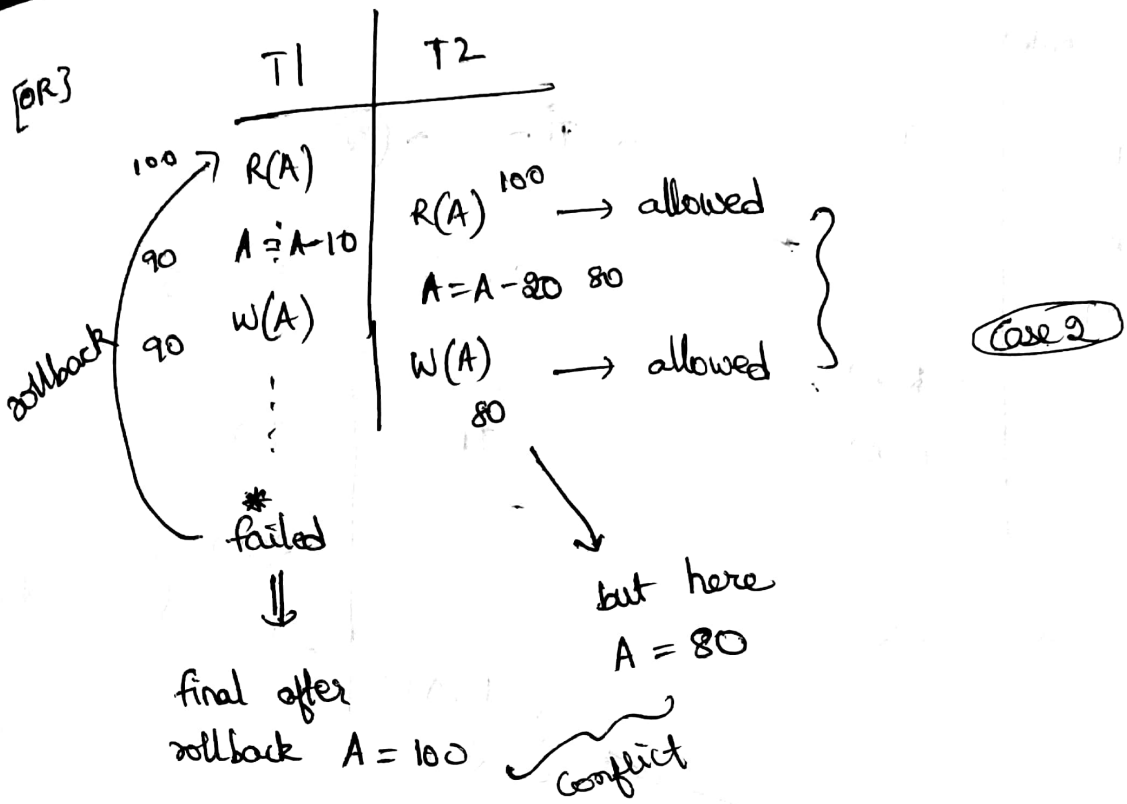
## *. Cascading schedule (vs) Cascadeless schedule :

Ex:




- CPU Utilization is wasted / degraded.
- This is called Cascading schedule.

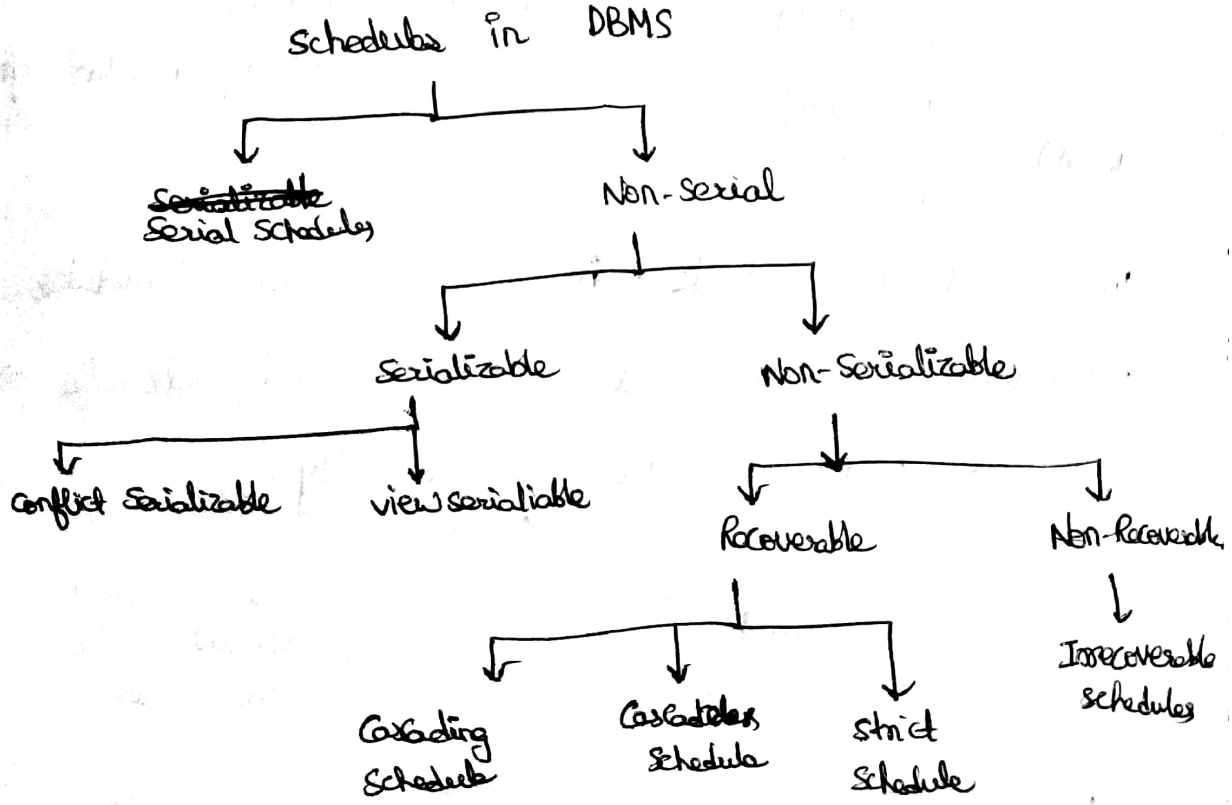





- This is called (2 cases) are called Cascadeless Schedule.

## IMP *. SERIALIZABILITY :

- In DBMS, schedules may be classified as:



Schedule S

| $T_1$ | $T_2$ |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) |
| | W(A) |

Serial Schedule


Consider another S

S

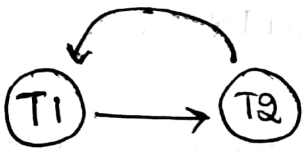
| T1 | T2 |
|---|---|
| | R(A) |
| | W(A) |
| R(A) | |
| W(A) | |

Serial Schedule


Now, Parallel Schedules

Schedule S

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




Parallel schedules forms closed | circular graph

We need to check if these Parallel Schedules equivalents exists that will be serial schedules or not. we call this as "Serializability."

- Conflict Serializable.
- View Serializable

Parallel schedule equivalent Serial Schedule is to be found.

Schedule S

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

Possible serial schedules for any 3 transactions is,

T1 → T2 → T3
T1 → T3 → T2
T2 → T3 → T1
T2 → T1 → T3
T3 → T1 → T2
T3 → T2 → T1

(3!)

*. CONFLICT EQUIVALENT :

R(A) R(A) } Non conflict pairs

R(A) W(A)
W(A) R(A)
W(A) W(A)
} conflict pair (problem here)

R(B) R(A)
W(B) R(A)
R(B) W(A)
W(A) W(B)
} Non-conflict pairs

Consider another schedule S'

S'

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

Consider a schedule S

S

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

- Adjacent Non-conflict pairs are to be swapped.
- No change in positions for a conflict pairs.

$S^1$ schedule

| T1 | T2 |
|---|---|
| R(A) | |
| W(A) | |
| | R(A) non conflict |
| R(B) | |
| | W(A) |

$\Longrightarrow$

$S^1$

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

This $S^1$ schedule is now equivalent to S schedule.

So, $S \equiv S^1$ are conflict equivalent schedules.

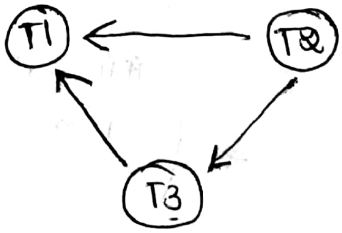
- **PRECEDENCE GRAPH :**

Consider a schedule S,

| T1 | T2 | T3 |
|---|---|---|
| R(x) | | |
| | | R(y) |
| | | R(x) |
| | R(y) | |
| | R(z) | |
| | | W(y) |
| | W(z) | |
| R(z) | | |
| W(x) | | |
| W(z) | | |

- check the conflict pairs in other transactions and draw the edges

Precedence Graph




Loop/cycle - Not there

No → conflict serializable schedule → Consistent

yes → view serializable

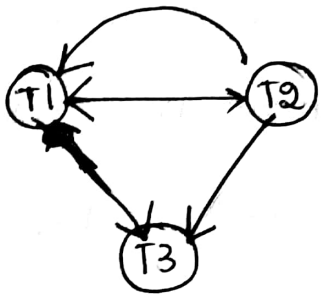
indegree = 0

$T_2 \rightarrow T_3 \rightarrow T_1$

(sequence)

Consider A schedule S,

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



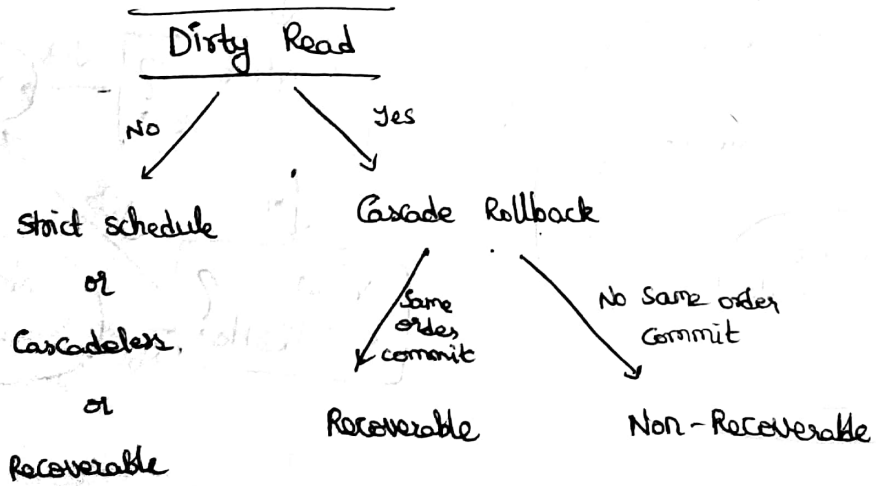

There is loop between T1 & T2

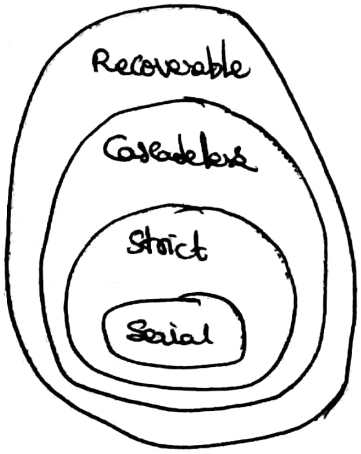
← Non conflict Serializable

Both are view equivalent.

*. CONCURRENCY CONTROL :

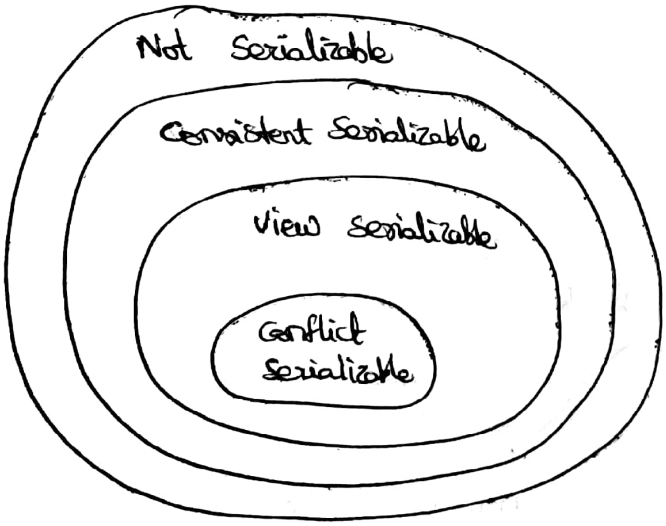
*. Characterizing Schedules :

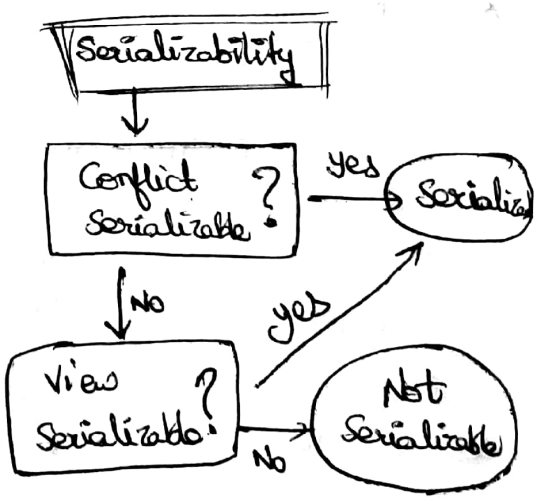


*. on a short note :

- Recoverable schedules : Recovery is possible
- Non-Recoverable schedules should not be permitted by DBMS.
- No committed transactions ever needs to be rolled back
- Cascading rollback may occur in some recoverable schedules.
- Uncommitted transactions may need to be rolled back.
- Cascadeless schedule : Avoids cascading rollback.
- Strict schedule : More strict nature of parallel processing.



# MODULE - 6 : PHYSICAL DATABASE DESIGN

## *. INDEXING :

- I/O cost is reduced by using ~~reducing~~ indexing.

Ex:
Consider a Hard disk in which Block Size = 1000 B, each record is of size = 250 B. If total no. of records are 10000 and the data entered in Hard disk without any order (unordered). What is the average time complexity to search a record from HD?

Sol:

$$\text{No. of records in each block} = \frac{\text{Size of Block}}{\text{Size of each record}}$$

$$= \frac{1000}{250} = 4$$

$$\text{No. of Blocks required} = \frac{\text{Total of records}}{\text{No. of records in each}} = \frac{10000}{4} = 2500$$

- Calculate complexity interms of I/O cost.
- Best case is found in 1st block = $O(1)$.
- Worst case is found in last block = $O(n)$
- Average case is linear search $O\left(\frac{n}{2}\right) \sim O(n)$

$\sim \frac{2500}{2} \sim 1250.$

→ This is for data is unordered.

But if the data is ordered, then it is Binary Search with time complexity $\log_2 N$

$N = 2500 \rightarrow \log_2 2500 \approx 11.288 \sim 12$ times

But when we use Indexing,

What is average time complexity to search a record from Index table if Index table entry $= 20\,B$ (Key 10B + Pointer 10B).

- Block size in Index table = Block size in Harddisk.
- Each Block in Index table contains = $\frac{\text{Size of Block}}{\text{Index Block entry}} = \frac{1000}{20} = 50$
- ordered → Sparse method, unordered → Dense method
- Total entries in Index Table = $\frac{\text{total blocks in harddisk}}{\text{each Block in Index table contains}} = \frac{2500}{50} = 50$
- Searching in Index table (Key) ~ Binary Search $= \log_2 50 \approx 5.649 \sim 6$ times.

∴ Just need to search **6 times** if indexing used But 12 times if not used.

If Dense method is used,

total entries in Index table = $\frac{\text{Total entries in HDD}}{\text{entries in each block}} = \frac{10000}{50}$

$= 200$

searching = $\log_2 200 \sim 8$ times + 1 time (↓ for searching in HDD) = 9 times

∴ Sparse (ordered data) Indexing used = 4 times search.

∴ Dense (unordered data) Indexing used = 9 times search.

∴ Sparse (ordered data) Indexing not used = 12 times search.

∴ Dense (unordered data) Indexing not used = 1250 times search.

---

## *. TYPES OF INDEXES :

1. Primary Index
2. Clustered Index
3. Secondary Index

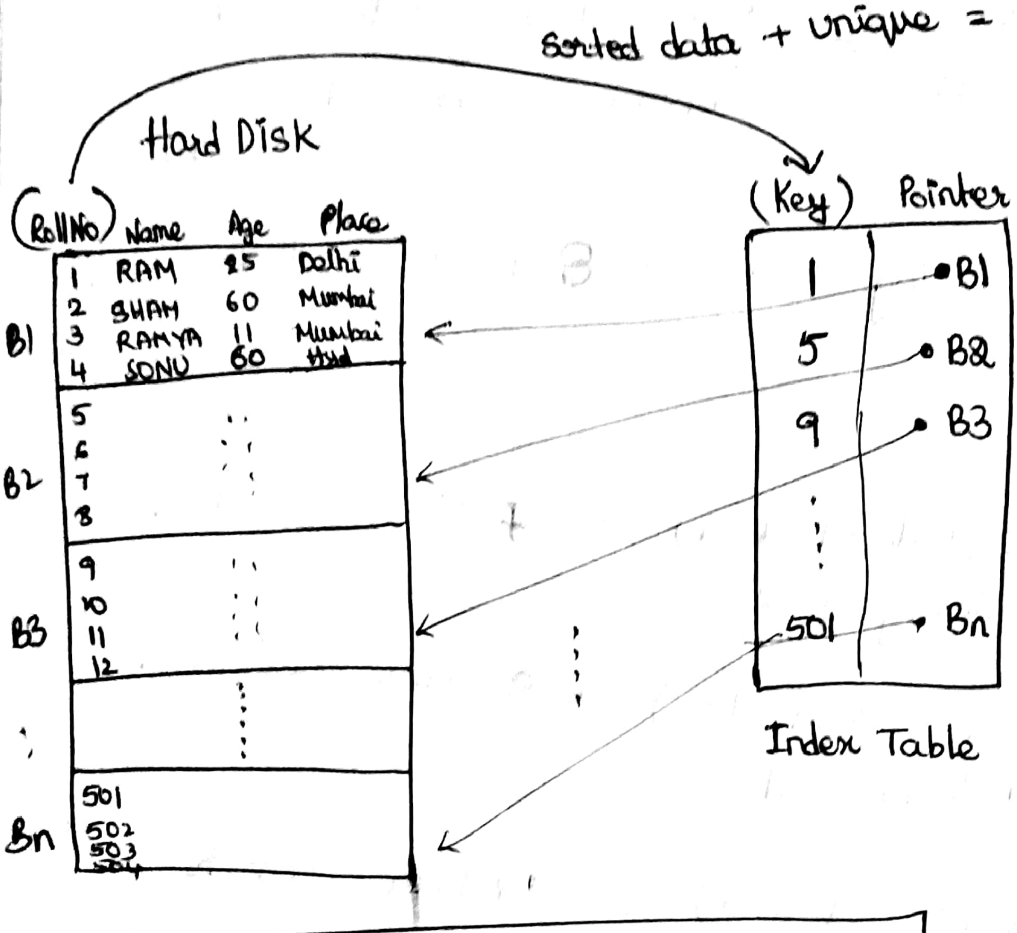
| ascending or descending | Key | Non Key |
|---|---|---|
| Ordered File | Primary Index | Clustered Index |
| Unordered File | Secondary Index | Secondary Index |

Key ↓ uniqueness

Non Key ↓ repeated values

Unordered File ↓ random order

## * PRIMARY INDEX : (Sparse) (ordered + Key)



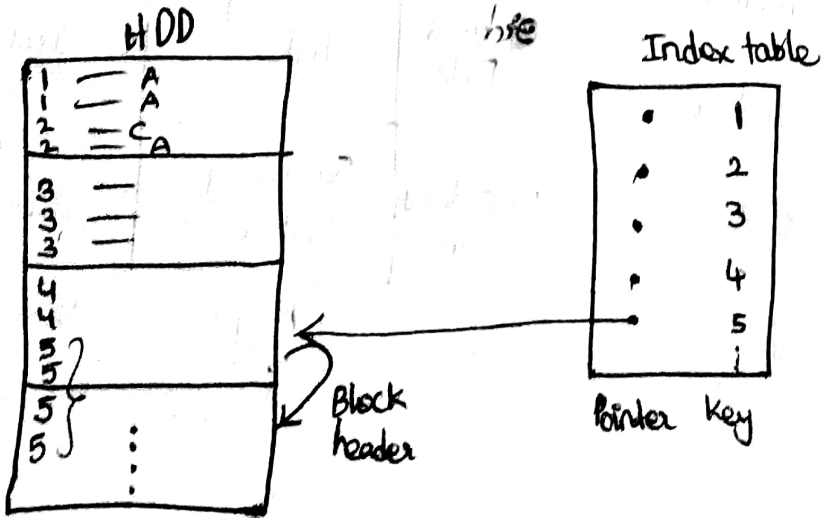

- No. of entries in Index table = No. of Blocks in HDD * Sparse Method

- Search time complex table = $\log_2 N + 1 \sim O(\log N)$

  where N is no. of blocks in Index table.

---

## * CLUSTERED INDEX : (ordered + Non Key)



- Search time complexity = $\log_2 N + 1 + K \sim O(\log N)$

  where N = No. of blocks in index table

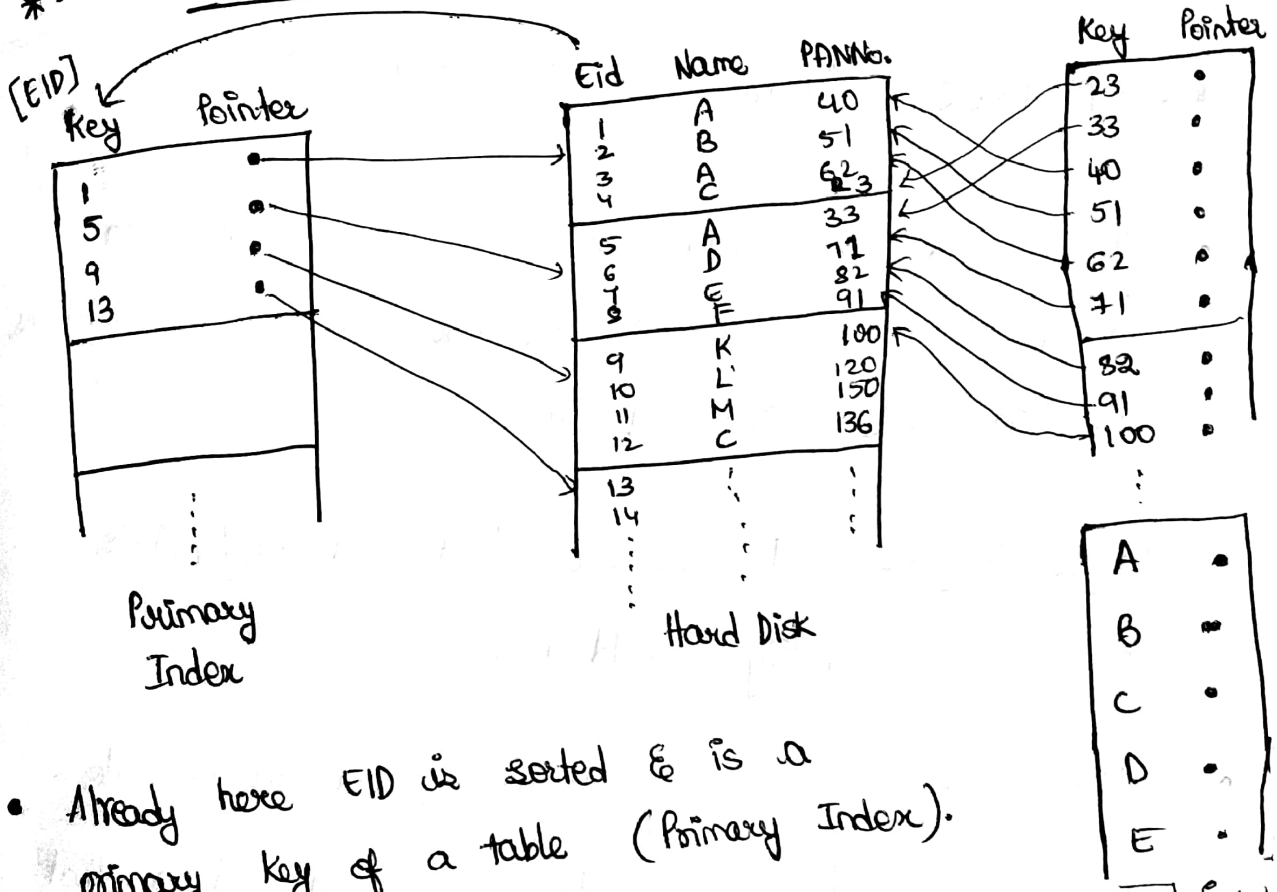
  K = No. of Block Headers

  $\log_2 N$ : Searching in Index table; 1 : Search in HDD; K : Block Header HDD

- Atmost (maximum) 1 cluster index is only possible in any table of Database.

---

## *. SECONDARY INDEX : (unordered + any type key)




- Already here EID is sorted & is a primary key of a table (Primary Index).

- Even after having primary Index, if the processed Query includes a non primary index that is also unique, then there needs the concept of Secondary Index.

Ex : SELECT * FROM EMPLOYEE WHERE PANNO = 82;

Primary Index ✗

✓ Secondary index

- No. of Records in Index = No. of records in Hard disk.
- Search time complexity = $\log_2 N + 1$ , N = No. of record in Index.
- Time complexity for unordered + Non key Searching = $(\log_2 N + K)N$
  - $\log_2 N$ ↓ Search in Index table
  - $K$ ↓ No. of Seaches in
  - $N$ ↓ intermediate layer for Block of Records

---

## *. B-TREE (Dynamic Multilevel Indexing):

- It is a tree data structure that keeps data stored and allows searches, insertions, and deletions in logarithmic amortized time.
- Block Pointer – A tree pointer that denotes/points to its corresponding child node
- Keys – Searching criteria placed in a node. Every key has a Data Pointer/Record pointer that points to its record present in Secondary Memory.
- order – Maximum child nodes in the tree for a root.

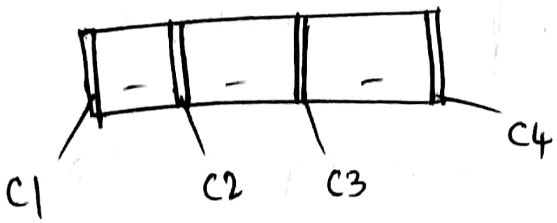
☆ Suppose order = P ⟹ keys = P-1 ,

Block pointer = P , record pointer = P-1 .

| children node | Root Node | Intermediate Node |
|---|---|---|
| Max | P | P |
| Min | $\lceil P/2 \rceil$ | $\lceil P/2 \rceil$ |

Ex: A tree has order = P = 4

Maximum no. of children = 4




Keys = P-1 = 3

Block pointers = 4

Record pointers = 3

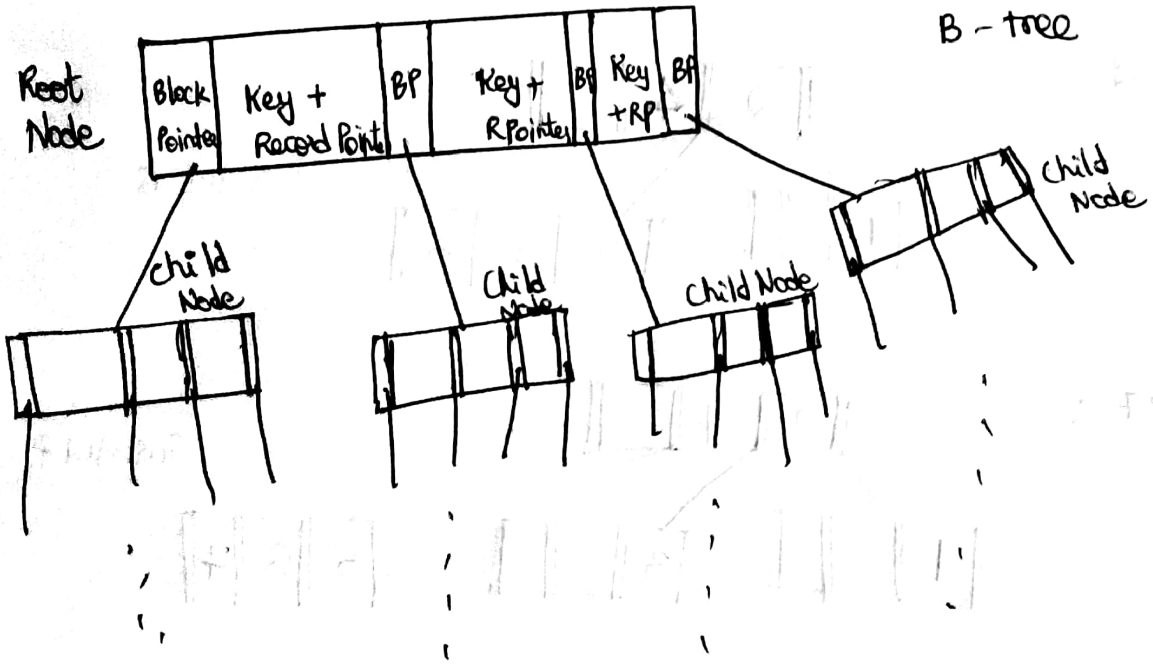
## *. INSERTION IN B-TREE :

Ex: Insert the following keys into B-tree if order of B-Tree = 4. & keys = 1,2,3,4,5,6,7,8,9,10

Max Keys = P - 1 = 4 - 1 = 3

Min Keys = $\lceil P/2 \rceil$ - 1 = 1

structure of B-tree



- Element < Root Element ⟹ Left side of tree
  Element > Root Element ⟹ Right side of tree

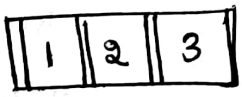
Step 1 :


inserted 1

Step 2 :

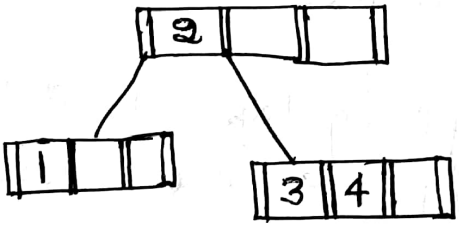

inserted 2

Step 3 :



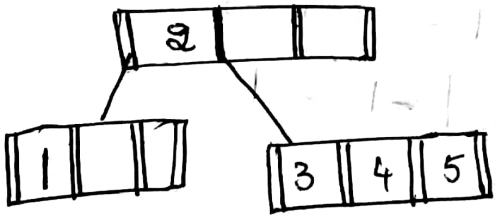

inserted 3

Step 4 : As the maximum key = 3 Now next key values will be in new child node



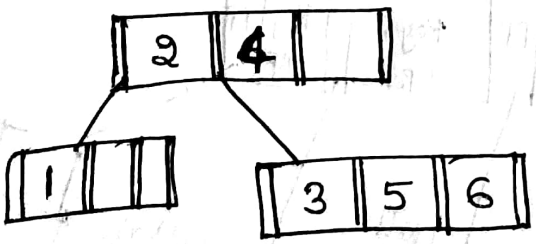

inserted 4

Step 5 :



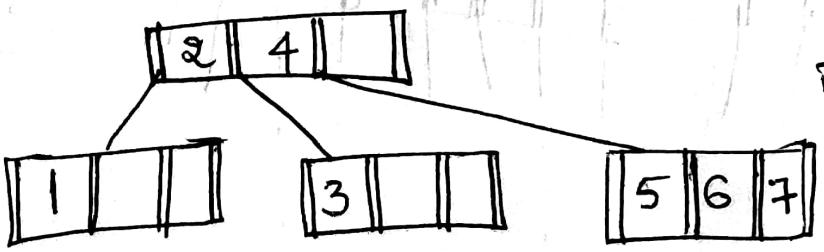

inserted 5

Step 6 :



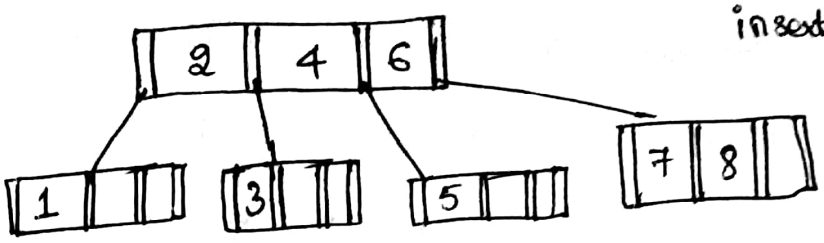

inserted 6

Step 7 :



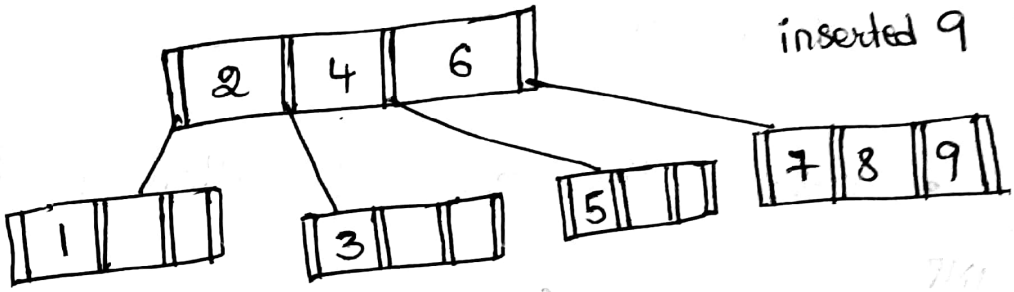

inserted 7

Step 8 :




inserted 8

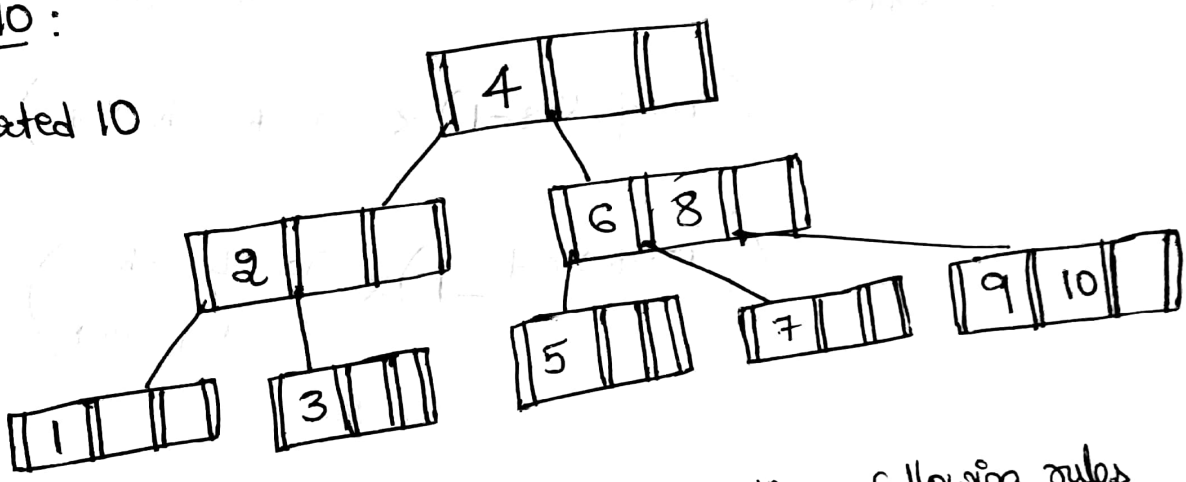
Step 9 :




inserted 9

step 10 :

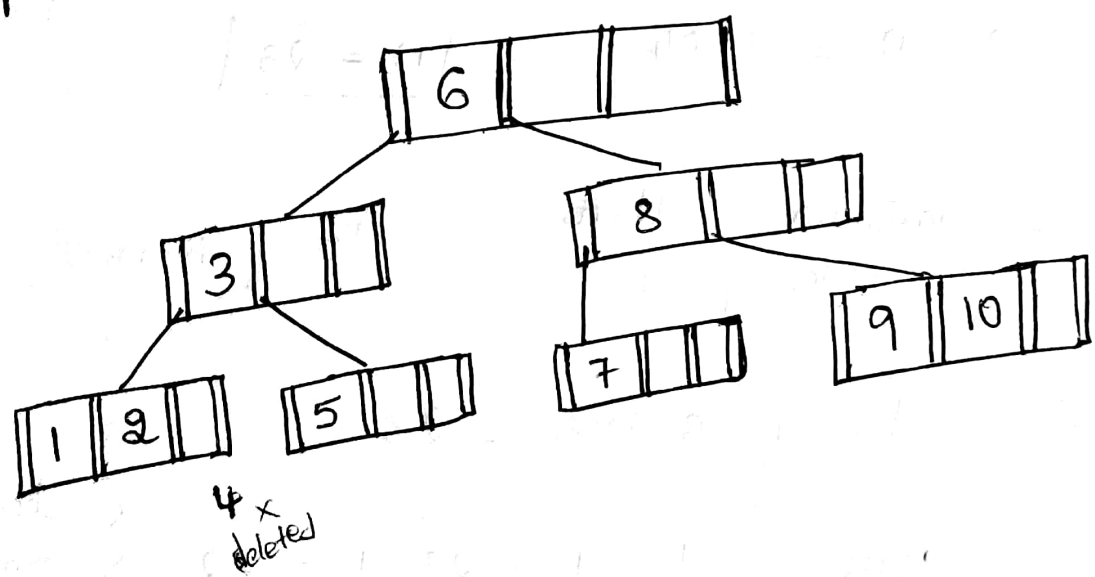
inserted 10




- Deletion will be same as insertion following rules

suppose delete value key 4

→



**Ex:** Consider a B-Tree with key size = 10 B, Block size = 512 B, data Pointer is of size 8 bytes and Block pointer is 5 Bytes. Find the order of B-Tree ?

- order = Maximum no. of children possible.

IMP

- Total size of B tree $\geq$ (no. of Block Pointers $\times$ size of each Block Pointer) + ((no. of Keys $-1$) $\times$ size of each key) + ((no. of record pointers $-1$) $\times$ size of each record pointer)

$$\Rightarrow \quad n \times 5 + (n-1)(10) + (n-1)\,8 \leq 512$$

$$5n + 18n - 18 \leq 512 \quad \Rightarrow \quad n \leq \frac{530}{23}$$

$$\Rightarrow \quad n \leq 23.04 \quad \Rightarrow \quad \boxed{n = 23}$$

$n$ = maximum children node possible = order of B-Tree

$\therefore$ order of B-Tree = 23.

Keys = order $-1$ $= 23 - 1 = 22$ } check formulas.

* Difference between B-Tree & B+-Tree :

| B-Tree | B+-Tree |
|---|---|
| • Data is stored in leaf as well as internal nodes. | • Data is stored only in leaf nodes. |
| • Searching is slower, Deletion is complex. | • Searching is faster, leaf node Deletion is easy (directly from ^) |
| • No redundant search key is present. | • Redundant keys may present. |
| • Leaf nodes not linked together | • Linked together like Linked List. |

Ex: Consider a B+ Tree with key size = 10 bytes block size = 512 bytes data pointer = 8 bytes and block pointer = 5 bytes. What is the order of leaf & non leaf node ?

IMP

• Total size of B+-Tree (non-leaf) $\geq$ (no. of Block Pointers $\times$ size of each Block Pointer) + ((no. of keys $-1$) $\times$ size of each key)

$\Rightarrow n \times 5 + (n-1) \times 10 \leq 512$

$5n + 10n - 10 \leq 512 \Rightarrow n \leq \frac{522}{15}$

$\Rightarrow n \leq 34.8 \sim \boxed{n = 34}$

order = 34 (internal nodes) = childrens

IMP

• Total size of B+-tree for (Leaf node) $\geq$ (No. of keys x size of each key) + (No. of data pointers x size of each data pointer) + 1.(Block pointer size)

$\Rightarrow \quad x \times (10) + x \times (8) + 1.(5) \leq 512$

$18x + 5 \leq 512 \quad \Rightarrow \quad x \leq \frac{507}{18}$

$\Rightarrow \quad x \leq 28.16 \quad \sim \quad \boxed{x = 28}$

∴ order of Leaf node = 28

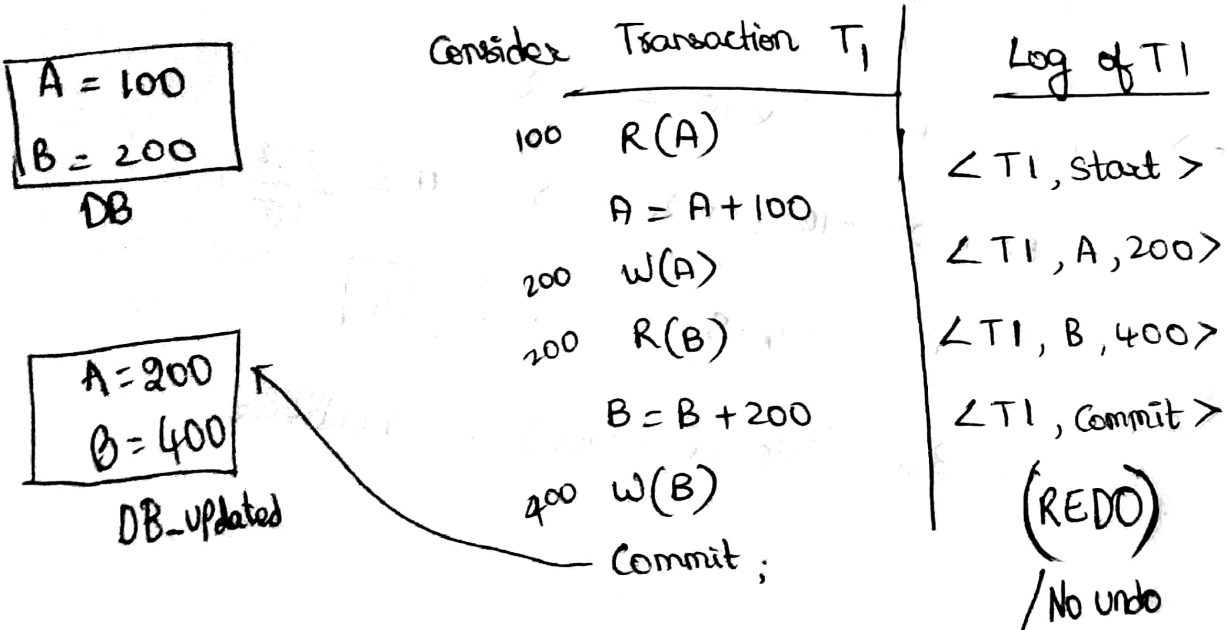
order of Non-leaf node = 34

---

## MODULE-5 : LOG BASED RECOVERY

(MISSED TOPIC)

### *. DEFERRED DATABASE MODIFICATION :



## * Immediate Database Modification :

| T1 | | |
|---|---|---|
| R(A) | 100 | A=100, B=200 |
| A = A + 100 | | |
| W(A) | 200 → | A=200, B=200 |
| R(B) | 200 | |
| B = B + 200 | | |
| W(B) | 400 → | A=200, B=400 |
| Commit ; | | |

- Before Commit, Write operation directly updates the value in database

**Log**

<T1, Start>

<T1, A, 100, 200>

<T1, B, 200, 400>

<T1, Commit>

Start + Commit = REDO

only Start = UNDO

- immediately old values are updated in database when failure occurs.