o.	Date	Title	Page No.	Teacher's Sign / Remarks

L1

Introduction to DBMS

What is Data -

data is a collection of raw, unorganised facts and details like text, observations, figures, symbols, and descriptions of things.

- in other words, data does not carry any specific purpose and has no significance by itself
- moreover, data is measured in terms of bits and bytes - which are basic units of information in the context of computer storage and processing
- data can be recorded and doesn't have any meaning unless processed

Types of data

- Quantitative - numerical form
  - weight, volume, cost of an item
- Qualitative - descriptive, but not numerical
  - name, gender, hair color of a person

What is Information

- information is processed, organized, and structured data
- it provides context of the data and enables decision making
- processed data that make sense to us
- information is extracted from the data, by analyzing and interpreting piece of data

Eg. - you have data of all the people living in your locality, its data, when you analyze and interpret the data and come to some conclusion that

- there are 100 senior citizens
- the sex ratio is 1:1
- new born babies are 100

} these are information

Data vs information

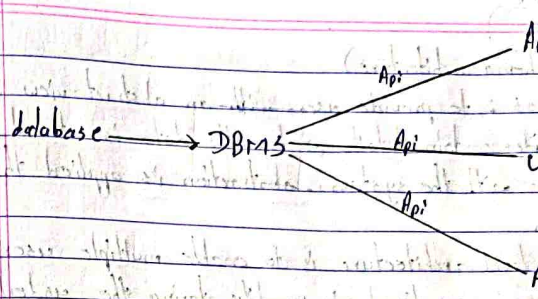
- data is collection of facts, while information puts those facts into context
- while data is raw and unorganized, information is organized
- data points are individual and sometimes unrelated, information maps out that data to provide a big-picture view of how it all fits together
- data, on its own, is meaningless. when it's analized and interpreted, it becomes meaningful information
- data does not depend on information. however, information depends on data
- data typically comes in the form of graphs, numbers, figures, or statistics. information is typically presented through words, language, thoughts, and ideas
- data isn't sufficient for decision-making but you can make decisions based on information.

What is database

- database is an electronic place/system where data is stored in a way that it can be easily accessed, managed, and updated
- to make real use data, we need database (DBMS)

What is DBMS

- a database management system is a collection of interrelated data and a set of programs to access those data. the collection of data, usually referred to as the database, contains information relevent to an interprise. the primary goal of a DBMS is to provide a way to store and retrive database information that is both convenient and efficient
- A DBMS is the database itself, along with all the software and functionality. it is used to perform different operations, like addition, access, updating and deletion of the data.



DBMS vs File system

- File-processing system has major disadvantages
  - data redundancy and inconsistancy
  - difficulty in accessing data
  - data isolation
  - Integrity problems
  - Atomicity problems
  - Concurrent-access anomalies
  - security problems

- above 7 are also the advantages of DBMS

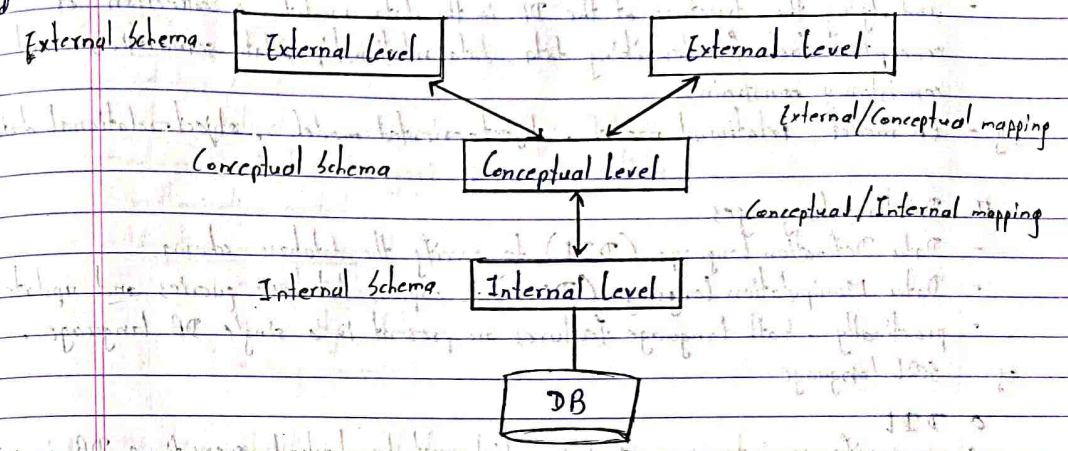
DBMS Architecture

- View of data (three schema architecture)
  - the major purpose of DBMS is to provide users with an abstract view of the data. that is, the system hides certain details of how the data is stored and maintained
  - to simplify user interaction with the system. abstraction is applied through several levels of abstraction
  - the main objective of three level architecture is to enable multiple users to access the same data with a personalized view while storing the underlying data only once.

- physical level / Internal level
  - the lowest level of abstraction describes how the data stored.
  - low-level data structure used.
  - it has physical schema which describes physical storage structure of DB
  - talks about - storage allocation (N-ary tree etc). data compression and encryption etc.
  - Goal - we must define algorithm that allow efficient access to data.

- logical level / Conceptual level
  - the conceptual schema describes the design of a database at the conceptual level. describes what data are stored in DB, and what relationship exist among those data.
  - user at logical level does not need to be aware about physical-level structures.
  - DBA, who must decide what information to keep in the DB use the logical level of abstraction
  - Goal - easy to use

- View level / External level
  - highest level of abstraction aims to simplify users' interaction with the system by providing different view to different end-user
  - each view schema describes the database part that a particular user group is interested and hides the remaining database from the user group
  - at the external level, a database contains several schemas that sometime called as subschema. the subschema is used to describe the different view of the database
  - at views also provide a security mechanism to prevent users from accessing certain parts of DB.



- Instances and schemas
  - the collection of information stored in DB at a particular moment is called an instance of DB
  - the overall design of the DB is called the DB schema
  - schema is structured discription of data. schema doesn't change frequently. data may change frequently
  - DB schema corresponds to the variable declarations (along with type) in a program
  - we have 3 types of schema ① physical ② logical. ③ view schema called subschemas.
  - logical schima is most important in terms of its effect on application programs, as programmers construct apps by using logical schema
  - physical data independance, physical schema change should not affect logical schema / application programs.

Data models
- provides a way to describe the design of a DB at logical level
- underlying the structure of the DB is the data model, a collection of conceptual tools for describing data, data relationship, data semantics and consistancy constraints

Eg - ER model, Relational model, object-oriented model, object-relational data model etc.

Database languages
- Data Definition language (DDL) to specify the database schema.
- Data Manipulation language (DML) to express database queries and updates
- practically, both language features are present in a single DB language,

eg - SQL language

DDL
- we specify consistency constrains, which must be checked, every time DB is updated.

DML
- Data manipulation involves
  - retrival of information stored in DB
  - Insertion of new information into DB
  - Ddetion of information from the DB
  - Updating existing information stored in DB
- Query language - a part of DML to specify statement requesting the retrival of information

How is Database accessed from Application programs ?
- Apps (written in host languages, C/C++, Java) interacts with DB

Eg - Banking system's module generating payrolls access DB by executing DML statements from the host language
- API is provided to send DML/DDL statements to DB and retrieve the results
  - Open database connectivity (ODBC), microsoft, "C"
  - Java database connectivity (JDBC), Java

Database Administrator DBA
- a person who has central control of both the data the programs that access those data

Functions of DBA
- schema definition
- storage structure and access methods
- schema and physical organization modifications
- authorization control
- Routine maintenance
  - periodic backups
  - security patches
  - any upgrades

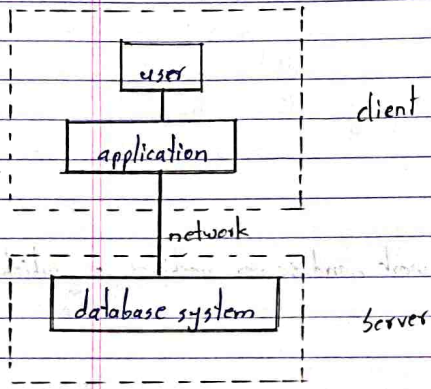
DBMS application architectures
- client machines, on which remote DB users work, and server machines on which DB system runs

① T_1 architecture
- the client, server and DB all present on the same machine

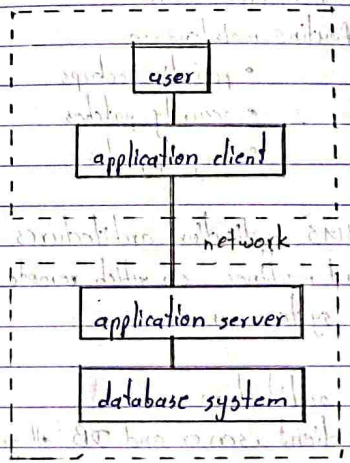
② T_2 architecture
- app is partitioned into 2-components
- client machine, which invokes DB system functionality at server end through query language statements
- API standards like ODBC & JDBC are used to interact between client & server

③ T_3 architecture
- app is partitioned into 3 logical components
- client machine is just frontend and doesn't contain any direct DB calls
- client machine communicates with App server, and app server communicated with DB system to access data.

- business logic, what action to take at that condition is in app server itself.
- T3 architecture are best for WWW applications
- Advantages
  - scalability due to distributed application server
  - data integrity, app server acts as a middle layer betwee client and DB which minimize the chances of data corruption
  - Security, client can't directly access DB, hence it is more secure



T_2 architecture



T_3 architecture

Entity - Relationship Model

- data model - collection of conceptual tools for describing data, data relationship, data semantics, and consistency constraints.
- ER model -
  - it is high level data model based on a perception of a real world that consists of a collection of basic objects, called entities and of relationship among these objects
  - graphical representation of ER model is ER diagram, which acts as a blueprint of DB

- Entity -

  an entity is a "thing" or "object" in the real world that is distinguishable from other all objects.
  - it has physical existance
  - each student in a collage is an entity
  - entity can be uniquely identified (by a primary attribute, aka primary key)
  - Strong entity - can be uniquely identified
  - Weak entity - can't be uniquely identified, depends on some other strong entity
    - it doesn't have sufficient attributes, to select a uniquely identifiable attribute
    - loan → strong entity, payment → weak, as instalments are sequential number counter can be generated separate for each loan
    - weak entity depends on strong entity for existance

- Entity set
  - it is a set of entities of the same type that share the same properties, or attributes.

Eg - student is an entity set
Eg - customer of a bank

Attributes

- an entity is represented by a set of attributes
- each entity has a value for each of it's attributes
- for each attribute, there is a set of permitted values, called the domain, or value set, of that attribute.

Eg — Student entity has following attributes.

- student ID
- name
- standard
- course
- batch
- contact number
- address

Types of Attributes

① simple
- attributes which can't divide further

Eg - customer's account number in a bank, student's roll number, etc.

② Composite
- can be divided into subparts (that is other attributes)

Eg - name of person, can be divided into first-name, middle-name, last-name
- if used wants to refer to an entire attribute or to only a component of the attribute
- address can also be divided. street, city, state, PIN code

③ Single-value
- only one value attribute

Eg - student ID, loan number for loan

④ Multi-valued
- attribute having more than one value

Eg - phone number, nominee-name on some insurance, dependent name etc.
- limit constraint may be applied, upper or lower limits

⑤ Derived
- value of this type of attribute can be divided from the value of other related attributes (derived)

Eg - age, loan-age, membership-period, etc.

⑥ Null value
- an attribute takes a null value when an entity does not have a value for it.
- it may indicate "not applicable", value doesn't exist. Eg - person having no middle name
- it may indicate "unknown"
  - unknown can indicate missing entry. Eg — name value of a customer is null. means it is missing as name must have some value
  - Not known, salary attribute value of an employee is null, means it is not known yet.

Relationship

- association among two or more entity

Eg - person has vehicle, parent has child, customer borrow loan, etc.
- Strong relationship, between two independent entities
- Weak relationship, between weak entity and its owner/strong entity

Eg - loan <instalment-payments> payment

degree of relationship

- number of entities participating in relationship
- Unary - only one entity participates. Eg - employee manages employee
- Binary - two entities participates. Eg - student take course
- Ternary - three entities participates. Eg - employee works-on branch, employee works-on job
- Binary and common

- Relationship constraints
  - mapping cardinality / cardinality ratio
    - number of entityes to which another entity can be associated via a relationship.
    - * one to one . entity in a associates with at most one entity in B . where A & B are entity sets . and an entity of B is associated with at most one entity of A

Eg - citizen has aadhar card

    - * one to many . entity in a associated with N entity in B . while entity in B is associated with at most one entity in A

Eg - citizen has vehicle

    - * Many to one . entity in a associated with at most one entity in B . while entity in B can be associated with N entity in A

Eg - course taken by professor

    - * many to many . entity in a associated with N entity in B . while entity in B also associated with N entity in A

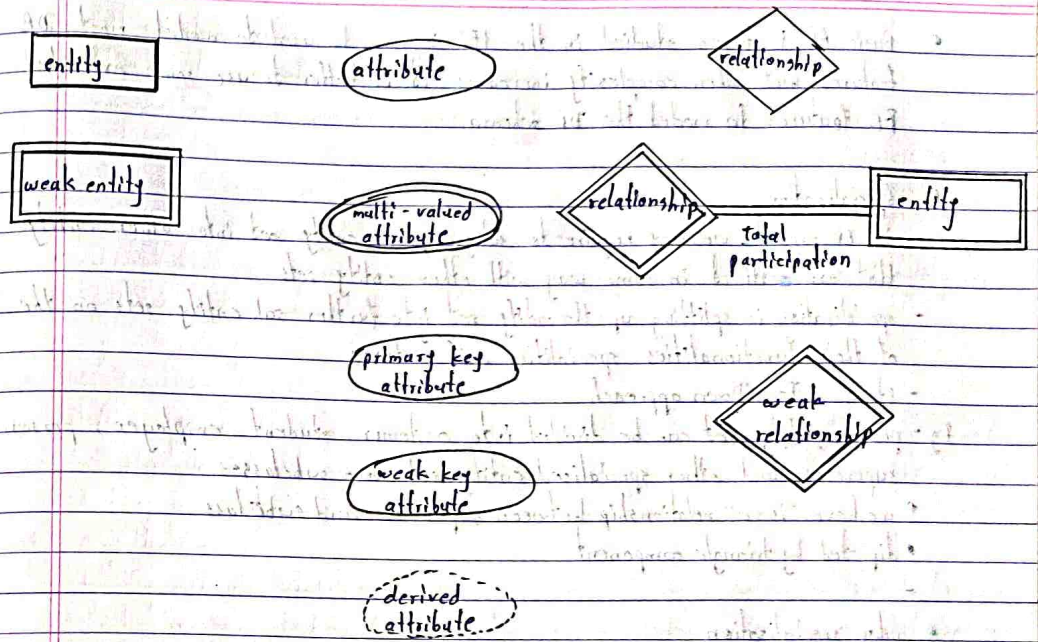
Eg - customer buy product
- student attend course

- Participation constraints
  - Aka . minimum cardinality constraint
  - types , partial & total participation
  - partial participation , not all intities are involved in the relationship instance
  - total participation , each entity must be involved in at least one relationship instance

Eg - customer borrow loan , loan has total participation as it can't exist without customer entity . and customer has partial participation

  - weak entity has total participation constraint , but strong may not have total

- ER notations



L4 Extended ER Features

- Basic ER Features studied in the LEC-3, can be used to model most DB features but when complexity increases, it is better to use some extended ER features to model the DB schema

Specialisation

- in ER model, we may require to subgroup an entity set into other entity sets that are distinct in some way with other entity sets
- specialisation is splitting up the entity set into further sub entity sets on the basis of their functionalities, specialities and features
- it is a Top-Down approach

Eg - person entity set can be divided into customer, student, employee, person is superclass and other specialised entity sets are subclasses

- we have "is-a" relationship between superclass and subclass
- dipicted by triangle component
-

- why specialisation
  - certain attributes may only be applicable to a few entities of the parent entity set
  - DB designer can show the distinctive features of the sub entities
  - To group some entities such entities we apply specialisation, to overall refine the DB blueprint

Generalisation

- it is just a reverse of spedalisation
- DB designer may encounter certain properties of two entities are overlapping designer may consider to make a new generalised entity set. that generalised entity set will be a super class
- "is-a" relationship is present between subclass and super class

Eg - car, jeep and bus all have some common attributes, to avoid data representation for the common attributes, DB designer may consider to generalise to a new entity set "vehicle"

- it is a Bottom-up approach

- why generalisation
  - make DB more refined and simpler
  - common attributes are not repeated

Attribute Inheritance

- both specialisation and generalisation, has attribute inheritance.
- the attribute of higher level entity sets are inherited by lower level entity sets.

Eg - customer and employee inherit the attributes of person.

Participation Inheritance

- if a parent entity set participates in relationship then its child entity sets will also participate in that relationship.

Aggregation

- how to show relationships among relationships & aggregation is the technique
- abstraction is applied to treat relationships as higher-level entities. we can call it abstract entity
- avoid redundancy by aggregating relationship as an entity set itself

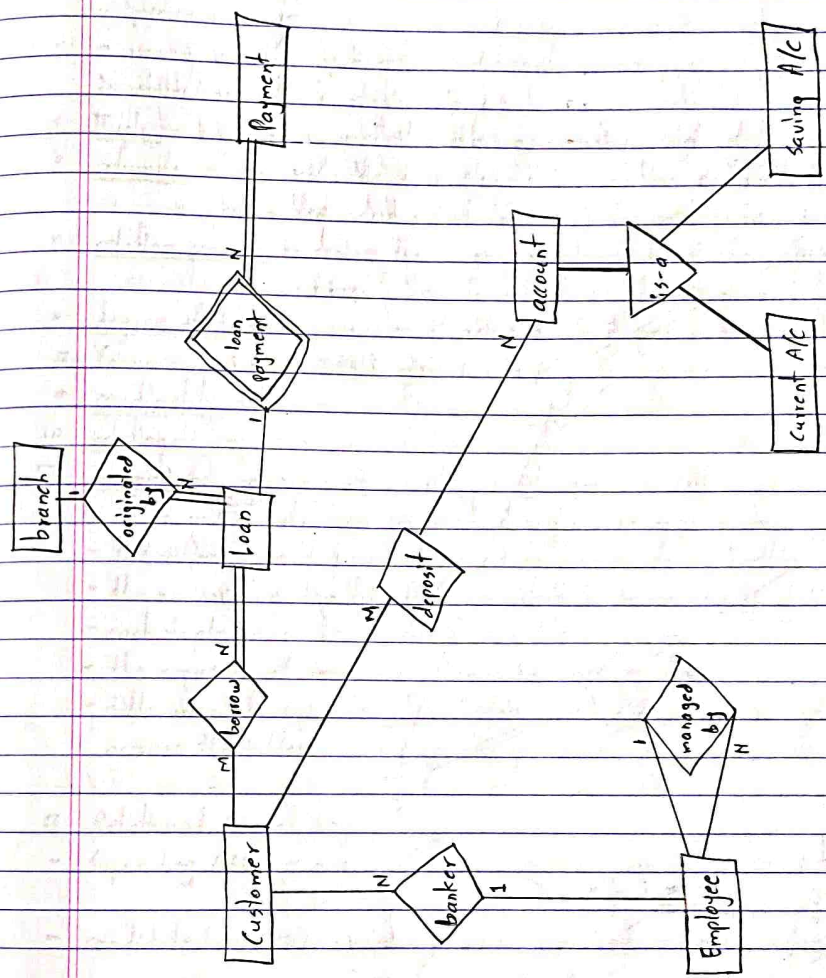
25

Steps to make ER diagram

① Identify Entity sets
② identify attributes and their types
③ Relationship and constraints
   - mapping
   - participation

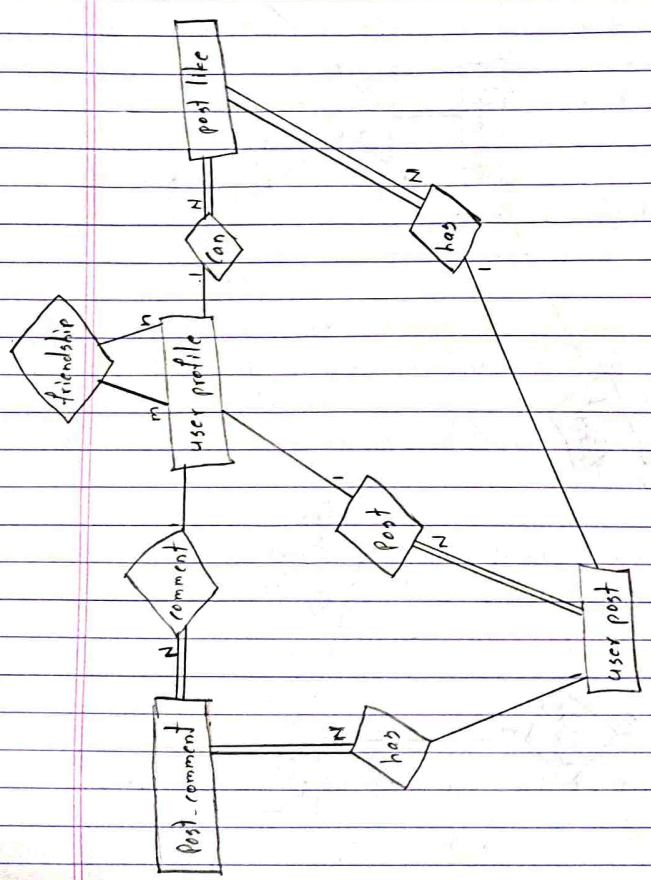
every entity have their attributes

ER model of Banking system



ER model of facebook

every entity - have attribut



L7

Relational model

- relational model (RM) organises the data in the form of relations (tables)
- a relational DB consists of collection of tables, each of which is assigned a uniqueue name
- a row in a table represents a relationship among a set of values. and table is collection of such relationship
- Tuple - a single row of the table representing single data point / a unique record
- columns - represents the attributes of the relation. each attribute, there is a permitted value, called domain of the attribute
- relation schema - defines the design and structure of the relation, contains the name of the relation and all the columns / attributes
- degree of table - number of attributes / columns in a given table / relation
- Common RM based DBMS system. aka RDBMS : oracle, IBM, MySQL, MS Access.
- cardinality - total no of tuples in a given relation
- relational key - set of attributes which can uniquely identify an each tuple
- important properties of a table in relational model
  - the name of relation is distinct among all other relation
  - the values have to be atomic, can't be broken down further
  - the name of each attribute / column must be unique
  - each tuple must be unique in a table
  - the sequence of row and column has no significance
  - table must follow integrity constraints - it helps to maintain data consistency accross the tables

- Relational Model keys
  - Super key (SK) - any P&C of attributes present in a table which can uniquely identify each tuple
  - candidate key (CK) - minimum subset of super keys, which can uniquely identify each tuple. it contains no redundent attribute
    - CK value shouldn't be NULL
  - Primary key (PK) - selected out of CK set, has the least no of attribute

- alternate key (AK) - all CK except PK
- foreign key (FK) -
  - it creates relation between two tables
  - a relation, say r_1 may include among its attributes the PK of an other relation, say r_2. this attribute is called FK from r_1 referencing r_2.
  - the relation r_1 is aka referencing (child) relation of the KK dependency, and r_2 is called referenced (parent) relation of the FK
  - FK helps to cross reference between two different relations
- Composite Key - PK formed using at least 2 attributes
- Compound Key - PK which is formed using 2 FK
- Surrogate Key -
  - Synthetic PK
  - generated automatically by DB, usually an integer value
  - may be used as PK

- Integrity constraints
  - CRUD operations must be done with some integrity policy so that DB is always constraint
  - introduced so that we do not accidentally corrupt the DB

- Domain constraints
  - restricts the value in the attribute of relation, specifies the domain
  - restrict the data types of every attribute

Eg - we want to specify that the enrolment should happen for candidate birth year <2002

- Entity constraints
  - every relation should have PK. PK != NULL

- Referential constraints
  - specified between two relations & helps maintain consistency among tuples of two relations
  - it requires that the value appearing in specified attributes of any tuple in referencing relation also appear in the specified attributes of at least one tuple in the referenced relation
  - if FK in referencing table refers to PK of referenced table then every value of the FK in referencing table must be NULL or available in referenced table
  - FK must have the matching PK for its each value in the parent table or it must be NULL

- Key constraints - the six types of key constraints present in the database management system, are -
  - NOT NULL - this constraint will restrict the user from not having NULL value. it ensure that every element in the database has a value
  - UNIQUE - it helps us to ensure that all the values consisting in a column are different from each other
  - DEFAULT - it is used to set the default value to the column. the default value is added to the columns if no value is specified for them
  - CHECK - it is one of the integrity constraints in DBMS. it keeps the check that integrity of data is maintained before the and after the complition of the CRUD
  - PRIMARY KEY - this is an attribute or set of attribute that can uniquely identify each entity in the entity set. the primary key must contain unique as well as not null values
  - FOREIGN KEY - whenever there is relationship between two entities, there must be some common attributes between them. this common attributes must be the primary key of an entity set and will become the foreign key of another entity set. this key will prevent every action which can result in loss of connection between tables

L8

Transform ER model to relational model

- Both ER model and relational model are abstract logical representation of real world enterprises. because the two models implies the similar design principles, we can convert ER model design into relational design.
- converting a DB representation from an ER diagram to a table format is the way we arrive at relational DB-design from an ER diagram
- ER diagram notations to relations

Strong Entity

- becomes an individual table with entity name, attributes becomes columns of the relation
- entities primary key (PK) is used as Relation's PK
- FK are added to establish relationships with other relations

Weak Entity

- a table is formed with all the attributes of an entity
- PK of it's corresponding strong entity will be added as FK
- PK of the relation will be a composite PK. {FK + Partial discriminator Key}

Single values Attributes

- represented as columns directly in the tables / relations

Composite Attributes

- handled by creating a separate attribute itself in the original relation for each composit attribute

Eg - Address {street-name, house-no}. is a composit attribute in customer relation, we add address-street-name & address-house-name as new columns in the attribute and ignore "address" as an attribute

Multivalued Attributes

- new tables (named as original attribute name) are created for each multivalue attribute

- PK of the entity is used as column FK in the new table
- multivalued attribute's similar name is added as column to define multiple values
- PK of the new table would be { FK + multivalued-name }

Eg - For strong entity Employee, dependent-name is a multivalued attribute
- new table named dependent-name will be formed with columns emp-id & name.
- PK { emp-id, name }
- FK { emp-id }

Derived Attributes - not considered in the tables

Generalisation
- Method 1

- create a table for the higher level entity set. for each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a common for each attribute of the primary key of the higher-level entity set
- Table 1 - account (account-number, balance)
- Table 2 - saving-account (account-number, interest-rate, daily-widrawl-limit)
- Table 3 - current-account (account-number, overdraft-amount, per-transaction-charges)

- Method 2

- an alternative representation is possible, if the generalisation is disjoint and complete - that is, if no entity is a member of two lower-level entity sets directly below a higher-level entity set, and if every entity in the higher level entity set is also a member of one of the lower-level entity sets. here, do not create a table for the higher-level entity set. instead, for each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the higher-level entity sets
- Table would be:

1) Table 1 - saving account (account-number, balance, interest-rate, daily-widrawl-limit)
2) Table 2 - current account (account-number, balance, overdraft-amount, per-transaction-charges)

- Drawbacks of Method 2

- if the second method were used for an overlapping generelisation, some values such as balance would be stored twice unnecessary. similarly, if the generalisation. some values such some accounts were neither saving nor current accounts - then such accounts could not be represented with the second method.

Aggregation
- table of the relationship set is made
- attributes includes primary keys of entity set and aggregation set's entities
- also add descriptive attribute of if any on the relationship.

SQL

- what is RDMS : relational database management system
  - software that enable us to implement designed relational model

Eg - MySQL, MS SQL, Oracle, IBM etc.

- table / relation in the simplest form of data storage object in R-DB
- MySQL is open-source RDMS, and it uses SQL for all CRUD operations

- MySQL used client-server model, where client is CLI or frontend that used services provided by MySQL server.

- Difference between SQL & MySQL
  - SQL is structured query language used to perform CRUD operations in R-DB, while MySQL is a RDBMS used to store, manage and administrate DB (provided by itself) using SQL

DATA TYPE	Description
CHAR	string (0-255), string with size = (0,255) eg - CHAR(25)
VARCHAR	string (0-255)
TINYTEXT	string (0-256)
TEXT	string (0-65535)
BLOB (binary large object)	string (0-65535)
MEDIUMTEXT	string (0-16777215)
MEDIUMBLOB	string (0-16777215)
LONGTEXT	string (0-4294967295)
LONGBLOB	string (0-4294967295)
TINYINT	integer (-128 to 127)
SMALLINT	integer (-32768 to 32767)
MEDIUMINT	integer (-8388608 to 8388607)
INT	integer (-2147483648 to 2147483647)
BIGINT	integer (-9223372036854775808 to -11-(-1)
FLOAL	decimal with precision to 23 digits
DOUBLE	decimal with 24 to 53 digits

DECIMAL	double stored as string
DATE	YYYY-MM-DD
DATETIME	YYYY-MM-DD HH:MM:SS
TIMESTAMP	YYYYMMDDHHMMSS
TIME	HH:MM:SS
ENUM	one of the preset values
SET	one or many of the preset values
BOOLEAN	0/1
BIT	eg - BIT(n), n upto 64, store values in bits

- size : TINY < SMALL < MEDIUM < INT < BIGINT
- variable length data types. Eg - VARCHAR are better to use because it uses only memory which ever are needed
- values can also be unsigned. Eg - INT UNSIGNED

Types of SQL commands

1. DDL data define language (define relation schema)
   - CREATE - create table, DB, view
   - ALTER TABLE - modification in table structure. Eg - change column datatype / add, remove
   - DROP - delete table, DB, view
   - TRUNCATE - remove all the tuples from the table
   - RENAME - rename DB name, table name, column name etc.

2. DRL/DQL data retrival language / data query language (retrive data from tables)
   - SELECT

3. DML data modification language (use to perform modification in DB)
   - INSERT - insert data into a relation
   - UPDATE - update relation data
   - DELETE - delete row(s) from the relation

4. DCL data control language (grant or revoke authorities from user)
   - GRANT - access privileges to the DB
   - REVOKE - revoke user access privileges

5. TCL translation control language (to manage translations done in the DB)
   - START TRANSACTION - begin a transaction
   - COMMIT - apply all the changes and end transection
   - ROLLBACK - discard changes and end transection
   - SAVEPOINT - checkout within the group of transaction in which to rollback

MANAGING DB (DDL)

creation of DB - CREATE DATABASE IF NOT EXISTS db_name;
USE db_name; to choose DB
DROP DATABASE IF EXISTS db_name; delete database
SHOW DATABASES; list all the DBs in the server
SHOW TABLES; list tables in the selected DB

DATA RETRIVAL LANGUAGE (DRL)

syntax - SELECT <set_of_column_name> FROM <table_name>;
- order of execution RIGHT to LEFT

Q. can we use SELECT keyword without using FROM clause?
- Yes, using DUAL Tables
- DUAL tables are dummy tables created by MySQL, help users to do certain obvious actions without referring to user defined Tables

Eg - SELECT 55+11;
SELECT now();
SELECT ucase(); / lcase();

- WHERE
  - reduce rows based on given conditions
- Eg - SELECT * FROM customer WHERE age >18;

- BETWEEN
- Eg - SELECT * FROM customer WHERE age between 0 AND 100;
  - in the abov 0 & 100 are inclusive, also they are part of reinge

- IN
  - reduces OR conditions
- Eg - SELECT * FROM officers WHERE officer_name IN ('Om', 'RAM', 'xx');

- AND / OR / NOT
  - AND - WHERE cond1 AND cond2
  - OR - WHERE cond1 OR cond2
  - NOT - WHERE col_name NOT IN (1,2,3,4);

- IS NULL
- Eg - SELECT * FROM customer WHERE prime_status is NULL;

- Pattern Searching / Wildcard ('%', '_')
  - '%' any number of character from 0 to n. similar to '*' asterisk in regex.
  - '_' any one character
  - SELECT * FROM customer WHERE name LIKE '%p_';

- ORDER BY
  - sorting the data retrived using WHERE clause.
  - ORDER BY <column_name> DESC; // ASC for ascending
- Eg - SELECT * FROM customer ORDER BY name DESC;

- GROUP BY
  - GROUP BY clause is used to collect data from multiple records and group the result by one or more column. it is generally used in SELECT statement
  - Groups into category based on column given
- Eg - SELECT c1, c2, c3 FROM sample_table WHERE cond GROUP BY c1, c2, c3
  - all the column names mentioned after SELECT statement shall be replaced in GROUP BY, in order to successfully execute the query.
  - used with aggregation functions to perform various actions:
    - COUNT()
    - SUM()
    - AVG()
    - MIN()
    - MAX()

- DISTINCT
  - find distinct values in the table
  - SELECT DISTINCT col_name FROM table_name;
  - GROUP BY can also be used for the same
    - SELECT col_name FROM table_name GROUP BY col_name; same o/p as above
    - SQL is smart enough to realise that if you are using GROUP BY and not using any aggregation function, then you mean "DISTINCT".

- GROUP BY HAVING
  - out of the categories made by GROUP BY, we would like to know only particular thing (cond)
  - similar to WHERE
  - SELECT COUNT(cust_id), country FROM customer GROUP BY country HAVING COUNT(cust_id) > 50;

* WHERE vs HAVING
1) both have same function of filtering the row base on certain conditions
2) WHERE clause is used to filter the rows from the table based on specified condition

3) HAVING clause is used to filter the rows from the groups based on the specified conditions
4) HAVING is used after GROUP BY while WHERE is used before GROUP BY clause
5) if you are using HAVING, GROUP BY is necessary
6) WHERE can be used with SELECT, UPDATE & DELETE keywords while GROUP BY used with SELECT.

```
CREATE DATABASE ORG;
SHOW DATABASES;
USE ORG;

CREATE TABLE worker(
WORKER_ID INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
FIRST_NAME VARCHAR(255),
LAST_NAME VARCHAR(255),
SALARY INT(15),
JOINING_DATE DATETIME,
DEPARTMENT VARCHAR(255)
);

INSERT INTO worker VALUES (1,'OM','KALDATE',10000,'11-12-13 11.12.01','HR');

INSERT INTO worker(WORKER_ID,FIRST_NAME,LAST_NAME,SALARY,JOINING_DATE,DEPARTMENT) VALUES
(2,'RAM','KALDATE',100,'11-12-13 10.12.10','CEO'),
(3,'GANDHI','MAHAN',150,'11-02-13 11.12.10','CEO'),
(4,'NAVNA','NARWAD',2000,'12-02-13 11.13.11','MANAGER'),
(5,'ABI','KALDATE',2500,'13-02-13 11.13.11','EMPLOYEE');

SELECT DEPARTMEN,COUNT(DEPARTMEN) FROM worker GROUP BY DEPARTMENT HAVING COUNT(DEPARTMENT) >1;
```

```
SELECT * FROM worker    //show entire table
```

WORKER_ID	FIRST_NAME	LAST_NAME	SALARY	JOINING_DATE	DEPARTMENT
1	OM	KALDATE	10000	11-12-13 11.12.01	HR
2	RAM	KALDATE	100	11-12-13 10.12.10	CEO
3	GANDHI	MAHAN	150	11-02-13 11.12.10	CEO
4	NAVNA	NARWAD	2000	12-02-13 11.13.11	MANAGER
5	ABI	KALDATE	2500	13-02-13 11.13.11	EMPLOYEE

```
DROP TABLE worker;    // delete table
SELECT WORKER_ID, DEPARTMENT FROM worker;    shows only column of worker_ID & DEPARTMENT
SELECT 1+4;    // make virtual table & show 16
SELECT now();    // show live time of server
SELECT lcase('OM');    // show in lower case    ucase() --> also can use
SELECT * FROM worker WHERE salary < 200;    // show all rows satisfy which
SELECT * FROM worker WHERE DEPARTMENT = 'HR';    // show 1 ID's row
SELECT * FROM worker WHERE salary BETWEEN 100 AND 200;    // id 3 Id show
SELECT * FROM worker WHERE salary = 100 OR salary = 150;
same SELECT * FROM worker WHERE DEPARTMENT IN ('HR','CEO');
    all the HR's & CEO's are printed
SELECT * FROM worker WHERE salary is NULL;    null salary employee prints
SELECT * FROM worker WHERE FIRST_NAME LIKE '_M';    # LIKE is IMP
    _ --> only one character    % --> any number of character from 0 to n
    _ _i --> third character must be i

SELECT * FROM worker ORDER BY salary;    --> default ascending (ASC)
SELECT * FROM worker ORDER BY salary DESC;    descending (DESC)
SELECT DISTINCT DEPARTMENT FROM worker;    shows distinct

SELECT DEPARTMENT, COUNT(DEPARTMENT) FROM worker GROUP BY DEPARTMENT;
SELECT DEPARTMENT, AVG(SALARY) FROM worker GROUP BY DEPARTMENT;
also can use MIN, MAX with GROUP BY
```

Constraints DDL

1) Primary key

```
CREATE TABLE customer (
    id INT PRIMARYKEY,
    branch_id INT,
    First_name CHAR(50),
    Last_name CHAR(50),
    DOB DATE,
    Gender CHAR(6),
    PRIMARY KEY(id)
);
```

two ways to define primary KEY.

- PK is not NULL, unique & only one per table.

2) Foreign key
- FK refers to PK of other table
- each relation can having any number of FK

```
CREATE TABLE ORDER (
    id INT PRIMARY KEY,
    delivery_date DATE,
    order_placed_date DATE,
    cust_id INT,
    FOREIGN KEY(cust_id) REPRESENTS customer(id)
);
```

3) UNIQUE
- unique, can be null, table can have multiple unique attributes

```
CREATE TABLE customer (
    email VARCHAR(1024) UNIQUE,
);
```

4) CHECK

```
CREATE TABLE customer (
    CONSTRAINT age_check CHECK(age>12),
);
```

- "age_check" can also avoid this, MySQL generates name of constraint automatically

5) DEFAULT
- set default value of the column.

```
CREATE TABLE account (
    saving_rate DOUBLE NOT NULL DEFAULT 4.25;
);
```

- An attribute can be PK and FK both in a table.

ALTER OPERATIONS changes schema

1) ADD
- add new column
- ALTER TABLE table_name ADD new_col_name datatype ADD new_col_name_datatype
Eg - ALTER TABLE customer ADD age INT NOT NULL;

2) MODIFY
- changes data type of an attribute
- ALTER TABLE table_name MODIFY col_name col_datatype;

Eg - VARCHAR TO CHAR
ALTER TABLE customer MODIFY name CHAR(1024);

3) CHANGE COLUMN
- rename column name
- ALTER TABLE table_name CHANGE COLUMN old_col new_col new_datatype;

Eg - ALTER TABLE customer CHANGE COLUMN name my_name VARCHAR(1024);

5) DROP COLUMN
- drop a column completely
- ALTER TABLE table_name DROP COLUMN col_name;
Eg - ALTER TABLE customer DROP COLUMN middle_name;

6) RENAME
- rename table name itself.
- ALTER TABLE table_name RENAME TO new_table_name;
Eg - ALTER TABLE customer RENAME TO customer_details;

DATA MANIPULATION LANGUAGE (DML)

① Insert
- INSERT INTO table_name (col1, col2, col3) VALUER (v1, v2, v3), (va1, va2, va3);

② Update
- UPDATE table_name SET col1 = 1, col2 = 'abc' WHERE id = 1;
- update multiple rows
Eg • UPDATE student SET standard = standard + 1;

* ON UPDATE CASCADE
- can be added to the table while creating constraints. suppose there is a situation where we have two tables such that primary key of one table is the foreign key for another table. if we update the primary key of the first table then using the ON UPDATE CASCADE foreign key of the second table automatically get updated

③ Delete
- DELETE FROM table_name WHERE id = 1;
- DELETE FROM table_name;

@ DELETE CASCADE - (to overcome DELETE constraint of referential constraints)
- what would happen to child entry if parent table's entry is deleted?
```
CREATE TABE ORDER(
        order_id int PRIMARY KEY;
        ddivery_date DATE;
        cust_id INT;
        FOREIGN KEY (cust_id) REFERENCES customer (id) ON DELETE CASCADE
    );
```

▲ ON DELETE NULL - (can FK have null values?..)
```
CREATE TABLE ORDER(
        order_id int PRIMARY KEY,
        delivery_date DATE,
        cust_id INT
        FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE SET NULL
    );
```

④ REPLACE
- primarily used for already present tuple in a table
- As UPDATE, using REPLACE with the help of WHERE clause in PK, then that row will be replaced.
- As INSERT, if there is no duplicate data new tuple will be inserted.
- REPLACE INTO student (id, class) VALUES (4, 3);
- REPLACE INTO table SET col1 = val1, col2 = val2;

```
CREATE TABLE order_details(
    order_id INT PRIMARY KEY,
    delivery_date DATE,
    cust_id INT,
    FOREIGN KEY (cust_id) REFERENCES customer(id)
    );
```

order_id	delivery_date	cust_id
100	2024-01-01	1
101	2024-01-02	1
102	2024-01-02	2

```
INSERT INTO order_details VALUES
(100, '2024-01-01', 1),
(101, '2024-01-02', 1);
INSERT INTO order_details VALUES (102,'2024-01-02',2);
```

- `INSERT INTO order_details (order_id, cust_id) VALUES (90, 2);`
- `ALTER TABLE order_details ADD interest FLOAT NOT NULL DEFAULT 0;`
- `ALTER TABLE order_details MODIFY interest INT NOT NULL DEFAULT 0;`
- `DESC order_details;`
- `ALTER TABLE order_details CHANGE COLUMN delivery_date kadhe-pahej VARCHAR(255) NOT NULL DEFAULT 'UDYA';`
- `ALTER TABLE order_details DROP COLUMN interest;`
- `ALTER TABLE order_details RENAME TO account_details;`
  `SET SQL_SAFE_UPDATES = 0;`
- `UPDATE order_details SET delivery_date = 2024-01-00, order_id = 1000;`
- `UPDATE customer SET pincode = 11000;`
- `UPDATE customer SET pincode = pincode + 1;`

- `DELETE FROM order_details WHERE order_id = 100;`
- `DELETE FROM order_details;` // all Table will delete (values)

4

First customer table implement then order_id table implement

```
CREATE TABLE customer(
    id INT PRIMARY KEY,
    c_name VARCHAR(255),
    gender CHAR(1),
    city VARCHAR(255),
    pincode INT NOT NULL DEFAULT 0
    CONSTRAINT CHECK (pincode > 1000)
    );
```

id	c_name	gender	city	pincode
1	OM	M	SAWARGAON	413624
2	NAVNATH	M	SAWARGAON	413624
3	TANVI	F	SOLAPUR	411004

```
INSERT INTO customer VALUES
(1, 'OM', 'M', 'SAWARGAON', 413624),
(2, 'NAVNATH', 'M', 'SAWARGAON', 413624),
(3, 'TANVI', 'F', 'SOLAPUR', 311004);
```

* Referential Constaint
① INSERT
② DELETE Constaint
→ On delete concate
→ On delete Set Null

```
CREATE TABLE cust (
    id INT PRIMARY KEY,
    C-name VARCHAR(255),
    gender CHAR(1),
    );
INSERT INTO cust VALUES (1, 'OM', 'M');

CREATE TABLE o-details (
    order_id INT PRIMARY KEY,
    delivery_date DATE,
    cust_id INT,
    FOREIGN KEY (cust_id) REFERENCES cust(id) ON DELETE CASCADE
    );
                                              SET NULL
INSERT INO o-details VALUES ('100', '2024-11-11', 1);

DELETE FROM cust WHERE id = 1;
```

ODC * here the o-details related to OM customer. when parent is deleted child will also delets, entry was deleted. entire tuple delets.

ODSN * here the o-details related to OM customer when parent is ddeleted the foreign key on the child will filled by null. data remains in the table

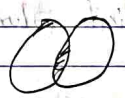
* Replace ① if data already present. replace it
② if data not present. INSERT it.

```
REPLACE INTO cust VALUES (5, 'RAM', 'M', 'sawargaon', 411006);
REPLACE INTO cust VALUES (5, 'xxx', 'F', 'LL', 414141); // id is INT (P.K)
REPLACE INTO cust (id, C_name) VALUES (5, 'sam');
REPLACE INTO cust SET id = 5, C-name = 'ramakant';

REPLACE INTO cust (id, C_name, gender) SELECT id, C_name, gender FROM
                                    cust WHERE id = 2;
```

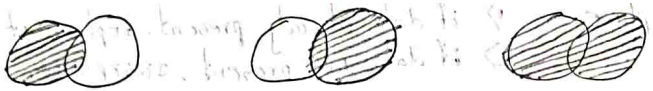
□ JOINING TABLES
- all RDBMS are relational in nature, we refer to other tables to get meaningful outcomes.
- FK are used to do reference to other table.

* INNER JOIN
returns a resultant table that has matching values from both the tables
→ ON DELETE CASCADE or all the tables
→ ON DELETE SET NULL - SELECT column-list FROM table1 INNER JOIN table2 ON condition1 INNER JOIN table 3 ON condition 2



- Aliasin MySQL (AS)
- Alias in mySQL is used to give a temperory name to table or column in a table for the purpose of perticular query, it work as nick-name, it works as a nickname for expressing the tables or column names, it makes the query short & neat.

```
SELECT col_name AS alias_name FROM table_name
SELECT colname1, col_name2, FROM table_name AS alias_name.
```



1) OUTER JOIN

1) LEFT JOIN
- this returns a resulting table that all the data from left table and the matched data from right table
- SELECT colums FROM table1 LEFT JOIN table2 ON join-condition;

2) RIGHT JOIN
- this returns a resulting table that all the data from right table and the matched data from left table
- SELECT colums FROM table RIGHT JOIN table2 ON join-condition;

3) FULL JOIN
- this returns a resulting table that contains all the data when there is a match on left or right table data.
- Emulated in MySQL using LEFT and RIGHT JOIN
- LEFT JOIN UNION RIGHT JOIN.

* SELECT columns FROM table1 as t_1 LEFT JOIN table2 as t_2 ON t_1.id = t_2.id
UNION
SELECT columns FROM table1 as t_1 RIGHT JOIN table2 as t_2 ON t_1.id = t_2.id;

* UNION ALL can also be used this will duplicate values as well while UNION gives unique values

4) CROSS JOIN
- this returns all the cartesian products of the data present in both tables. hence all posible variations are reflected in the output
- used rearely in practical purpose
- table1 has 10 rows and table2 has 5, then resultant would have 50 rows
- SELECT column_list FROM table1 CROSS JOIN table2;

5) SELF JOIN
- it is used to get the output from a particular table when the same table is joined to itself.
- Emulated using INNER JOIN
- SELECT columns FROM table1 as t_1 INNER JOIN table2 as t_2 ON t_1.id = t_2.id

- join without using JOIN keywords
- SELECT * FROM table1, table2 WHERE condition.

Eg - SELECT artistname, album_name, year_recorded FROM artist, album WHERE artist.id = album.artist_id;

- SET operations
- used to combine multiple select statements.
- always gives distinct rows.

JOIN	SET operation.
- combines multiple tables based on matching condition	combination is resulting set from two or more SELECT statements.
- column wise combination	Row wise combination.
- data type of two tables can be different	datatype of corresponding columns from each table should be the same
- can generate distinct or duplicate rows	generate distinct rows
- the number of columns selected may or may not be the same for each table	the number of columns selected must be the same from each table
- combines results horizontally	combines results vertically

0 UNION
- combines two or more SELECT statements
- SELECT * FROM table1
UNION
SELECT * FROM table2
- number of column, order of column must be same for table1 and table2

1 INTERSECT
- returns common values of the tables
- Emulated
* SELECT DISTINCT column_list FROM table1 INNER JOIN table2 USING (join_col);
* SELECT DISTINCT * FROM table1 INNER JOIN table2 ON USING (id);

2 MINUS
- this operation returns the distinct row from the first table that does not occure in the second table
- Emulated
* SELECT column_list FROM table1 LEFT JOIN table2 ON condition WHERE table2.column_name IS NULL;
eg - SELECT id FROM table1 LEFT JOIN table2 USING(id) WHERE table2.id IS NULL;

customer

order_details

INNER JOINS
```
SELECT C.*, O.* FROM customer AS C INNER JOIN order_details AS O ON
C.id = O.cust_id;
```

LEFT JOINS
```
SELECT C.C_name, C.gender, O.order_id, O.cust_id FROM customes AS C
LEFT JOIN order_details AS O. ON C.id = O.cust_id;
```

RIGHT JOINS
```
SELECT C.C_name, C.gender, O.order_id, O.cust_id FROM customer AS C
RIGHT JOIN order_details AS O ON C.id = O.cust_id;
```

INNER JOIN WITH CONDI
```
SELECT C.C_name, O.* FROM customer AS C INNER JOIN order_details AS O
ON C.id = O.cust_id WHERE C.pincode = 413624 OR O.order_id = 100;
```

SELF JOIN
```
SELECT C.*, O.* FROM customer AS C INNER JOIN order_details
id's wrond in SELF JOIN the Two tables are only for name the only
one table is used
SELECT C.C_name, C1.city FROM customer AS C INNER JOIN customer
AS C1 ON C.id = C1.id;
```

FULL JOIN
```
SELECT C.*, O.* FROM customer AS C LEFT JOIN order_details AS O ON C.id=O.cust_id
```

LJ + UNION + RJ
```
UNION
SELECT C.*, O.* FROM customer AS C RIGHT JOIN order_details AS O ON C.id = O.cust_id;
```

CROSS JOIN
```
SELECT customer_id, order_details.order_id FROM customer CROSS JOIN
order_details;
```

Q. can we JOIN without JOIN keyword..?

→ Yes

```
SELECT * FROM customer, order_details WHERE customer.id = order_details.cust_
id;
```

A = 1 4 8 10
B = 1 11 12 13
A∪B = 1

UNION Select * FROM T_1 where same columns are needed in T_1 & T_2
UNION
Select * FROM T_2;

INTERSECTION SELECT DISTINCT id FROM T_1 INNER JOIN T_2 USING (id)
we use INNER JOIN for INTERSECTION

MINUS SELECT id FROM T_1 LEFT JOIN T_2 USING(id) WHERE T_2.id is NULL;

DEPT 1

empid	name	role
1	A	enginner
2	B	salesman
3	C	manager
4	D	salesman
5	E	engineer

DEPT 2

empid	name	role
3	C	manager
6	F	marketing
7	G	salesman

Q. List out all the employee in the company
→ SELECT * FROM dept1
UNION
SELECT * FROM dept2;

UNION

Q. list out all the employee in all department who work as salesman
→ SELECT * FROM dept1 WHERE role = 'SALESMAN'
UNION
SELECT * FROM dept2 WHERE role = 'SALESMAN';

Q list out all the employee who work for both the department
→ SELECT dept1.*, dept2.* FROM dept1 INNER JOIN dept2 USING(empid);

Q list out all the employee working id dept1 but not in dept2
→ SELECT dept1.* FROM dept1 LEFT JOIN dept2 USING (empid) WHERE dept2.empid IS NULL;

■ SUB QUERIES

- SELECT * FROM customer WHERE id IN (SELECT id FROM customer WHERE c_name = 'TANVI');

- SELECT * FROM employee WHERE age IN (SELECT age FROM employee WHERE age >30);

Q Find employee details working in more than 1 project
- SELECT * FROM employee WHERE id IN (
SELECT empID FROM project group by empID having count(empID)>1);

Q select employee details having age > avg (age)
- SELECT * FROM employee WHERE age > (seled avg(age) FROM employee);

Q select max age person whose first name contains 'a'
- SELECT max(age) FROM (SELECT * FROM employee WHERE fname like '%a%')
AS temp; alias
↓
new table name

32
44
22
31
21

* find 3rd oldest employee

SELECT * FROM employee e1 WHERE 3 = (
SELECT COUNT(e2.age)
FROM employee e2
WHERE e2.age >= e1.age);

1) e1.age = 32
→ 2

2) e1.age = 44
→ 1

3) e1.age = 22
→ 4

4) e1.age = 31
→ 3 satisfy 3 = 3

VIEW

Create VIEW
- CREATE VIEW customer_view AS SELECT fname, age FROM employee;

SHOW view
- SELECT * FROM customer_view;

ALTER OR UPDATE
- ALTER VIEW customer_view AS SELECT fname FROM employee;

- DROP VIEW IF EXISTS customer_view;

SUB QUERIES

- outer query depends upon inner query
- alternatives to joins
- nested queries
- SELECT column_list FROM table_name WHERE column_name OPERATOR (SELECT column_list FROM table_name [WHERE]);

eg - SELECT * FROM table1 WHERE col1 IN (SELECT col1 FROM table1);

0. Sub queries exist mainly in 3 clause
1. inside a WHERE clause
2. inside a FROM clause
3. inside a SELECT clause

▷ Subquery using FROM clause
- SELECT MAX(rating) FROM (SELECT * FROM movie WHERE country = 'India') as temp;

▷ Subquery using SELECT
- SELECT (SELECT column_list FROM T_name WHERE condition), column_list FROM T2_name WHERE condition.

▷ Derived subquery
- SELECT columnlist FROM (SELECT columnlist FROM table_name WHERE [condition]) as new_table_name;

□ Co-related sub-query
- With a normal nested subquery, the inner select query runs first and executes once, returning values to be used by the main query. A corellated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.

JOINS	SUBQUERIES
- Faster	- slower
- Joins maximise calculation burden on DBMS	- Keeps responsibility of calculation on user
- complex, difficult to understand and implement	- comparitively easy to understand and implement
- choosing optimal join for optimal use case is difficult.	- easy

MySQL VIEWS

- a view is database object that has no values. its contents are based on the base table. it contains row and columns similar to the real table
- in MySQL, the view is a virtual table created by a query by joining one or more tables. it is operated similarly to the base table but does not contain any data of it's own.
- the view and table have one main difference that the view are definitions built on top of other tables (or views). if any changes occur in the underlying table, the same changes reflected in the view also

- CREATE VIEW view_name AS SELECT columns FROM tables [WHERE conditions];
- ALTER VIEW view_name AS SELECT columns FROM table WHERE conditions;
- DROP VIEW IF EXISTS view_name;
- CREATE VIEW trainer AS SELECT c.course_name, c.trainer, t.email FROM courses c, contact t WHERE c.id = t.id ; view using join clause

note : We can also import/export table schema from files (.csv or json)

Lovebabbar IMP 50 questions

Worker

WORKER_ID	FIRST_NAME	LAST_NAME	SALARY	JOINING_DATE	DEPARTMENT
1	Monika	Arora	100000	2014-02-20 09.00.00	HR
2	Niharika	Verma	80000	2014-06-11 09.00.00	Admin
3	Vishal	Singhal	300000	2014-02-20 09.00.00	HR
4	Amitabh	Singh	500000	2014-02-20 09.00.00	Admin
5	Vivek	Bhati	500000	2014-06-11 09.00.00	Admin
6	Vipul	Diwan	200000	2014-06-11 09.00.00	Account
7	Satish	Kumar	75000	2014-01-20 09.00.00	Account
8	Geetika	Chauhan	90.000	2014-04-11 09.00.00	Admin

Bonus

WORKER_REF_ID	BONUS_AMOUNT	BONUS_DATE
1	5000	2016-02-20 00.00.00
2	3000	2016-06-11 00.00.00
3	4000	2016-02-20 00.00.00
1	4500	2016-02-20 00.00.00
2	3500	2016-06-11 00.00.00

Title

WORKER_REF_ID	WORKER_TITLE	AFFECTED_FROM
1	Manager	2016-02-20 00.00.00
2	Executive	2016-06-11 00.00.00
8	Executive	2016-06-11 00.00.00
5	Manager	2016-06-11 00.00.00
4	Asst. Manager	2016-06-11 00.00.00
7	Executive	2016-06-11 00.00.00
6	Lead	2016-06-11 00.00.00
3	Lead	2016-06-11 00.00.00

Q.1 write an sql query to fetch "FIRST_NAME" from worker table using the alias name as <WORKER_NAME>.
→ SELECT FIRST_NAME AS WORKER_NAME FROM worker;

Q.2 write query to fetch "FIRST_NAME" from worker table in upper case
→ SELECT ucase(FIRST_NAME) FROM worker;

Q.3 write query to fetch unique values of DEPARTMENT from worker table
→ SELECT DISTINCT DEPARTMENT FROM worker;

Q.4 write query to print first three characters of FIRST_NAME from worker table
→ SELECT substring(FIRST_NAME, 1, 3) FROM worker;

Q.5 write query to find the position of the alphabate ('b') in the first name column 'Amitabh' from worker table
→ SELECT instr(FIRST_NAME, 'B') FROM worker WHERE FIRST_NAME = 'Amitabh';

Q.6 write query to print the FIRST_NAME from worker table after removing white spaces from the right side
→ SELECT rtrim(FIRST_NAME) FROM worker;

Q.7 write query to print the DEPARTMENT from worker table after removing white spaces from the left side
→ SELECT ltrim(FIRST_NAME) FROM worker;

Q.8 write query that fetches the unique values of DEPARTMENT from worker table and prints its length.
→ SELECT DISTINCT DEPARTMENT, length(DEPARTMENT) FROM worker;

Q.9 write query to print the FIRST_NAME from worker after replacing 'a' with 'A'
→ SELECT replace(FIRST_NAME, 'a', 'A') FROM worker;

Q.10 write query to print the FIRST_NAME and LAST_NAME from worker table into a single column COMPLETE_NAME.

→ SELECT concat(FIRST_NAME,' ',LAST_NAME) AS COMPLETE_NAME FROM worker;

Q.11 write query to print all worker details from the worker table order by FIRST_NAME Ascending.

→ SELECT * FROM worker ORDER BY FIRST_NAME;

Q.12 write query to print all worker details from the worker table order by FIRST_NAME ascending and DEPARTMENT descending.

→ SELECT * FROM worker ORDER BY FIRST_NAME, DEPARTMENT DESC;

Q.13 write query to print details for workers with the FIRST_NAME as "Vipul" and "Satish" from worker table.

→ SELECT * FROM worker WHERE FIRST_NAME IN ('Vipul', 'Satish');

Q.14 write query to print details of workers excluding FIRST_NAME "Vipul" and "Satish" from worker table

→ SELECT * FROM worker WHERE FIRST_NAME NOT IN ('Vipul', 'Satish');

→ SELECT * FROM worker WHERE FIRST_NAME <> 'Vipul' AND FIRST_NAME <> 'Satish';

Q.15 write query to print details of workers with department name as "Admin*"

→ SELECT * FROM worker WHERE DEPARTMENT LIKE 'Admin%';

Q.16 write query to print details of the worker whose FIRST_NAME contains 'a'

→ SELECT * FROM worker WHERE FIRST_NAME LIKE '%a%';

Q.17 write query to print details of the workers whose FIRST_NAME ends with 'a'.

→ SELECT * FROM worker WHERE FIRST_NAME LIKE '%a';

Q.18 write query to print details of the workers whose first name ends with 'h' and contains 6 alphabates

→ SELECT * FROM worker WHERE FIRST_NAME LIKE '_____h';

Q.19 write query to print details of the workers whos salary lie between 1.00.000 AND 5.00.000

→ SELECT * FROM worker WHERE SALARY BETWEEN 1.00.000 AND 5.00.000;

Q. write query to print details of the workers who have joined in Feb'2014.

→ SELECT * FROM worker WHERE YEAR(joining_date) = 2014 AND MONTH(joining_date) = 02;

Q.21 write query to fetch the count of employee working in the department 'Admin'

→ SELECT count(DEPARTMENT) FROM worker WHERE DEPARTMENT = 'Admin';

Q.22 write query to fetch worker full names with salaries >= 50.000 and <= 1.00.000;

→ SELECT concat(FIRST_NAME,' ',LAST_NAME) FROM worker WHERE SALARY BETWEEN 50.000 AND 1.00.000;

Q.23 write query to fetch the no. of workers for each department in the descending order.

→ SELECT DEPARTMENT, count(WORKER_ID) FROM worker GROUP BY DEPARTMENT ORDER BY count(WORKER_ID) DESC;

Q.24 write query to print details of the worker who are also manager.

→ SELECT w.* FROM worker AS w INNER JOIN title AS t ON w.worker_id = t.worker_ref_id WHERE t.worker_title = 'Manager';

Q.25 write query to fetch number (more than 1) of same titles in the emp of different types.

→ SELECT WORKER_TITLE, count(WORKER_TITLE) FROM title GROUP BY WORKER_TITLE HAVING count(WORKER_TITLE) > 1;

Q. 26 write query to show only odd rows from a table (<>)
→ SELECT * FROM worker where MOD (WORKER_ID, 2) != 0;

Q. 27 write query to show only even rows from a table
→ SELECT * FROM worker where MOD (WORKER_ID, 2) = 0;

Q. 28 write query to clone a new table from another table
→ CREATE TABLE worker_clone LIKE worker;
INSERT INTO worker_clone SELECT * FROM worker;
SELECT * FROM worker_clone;

Q. 29 write query to fetch intersecting records of two tables.
→ SELECT worker.* FROM worker INNER JOIN worker_clone USING (WORKER_ID);

Q. 30 write query to show records from one table that another table does not have
→ SELECT worker.* FROM worker LEFT JOIN worker_clone using (worker_id)
WHERE worker_clone.worker_id is NULL; minus

Q. 31 write query to show the current date and time.
→ SELECT curdate();
SELECT now();

Q. 32 Write query to show the top n (say 5) records of a table order by decreasing salary
→ SELECT * FROM worker ORDER BY SALARY DESC LIMIT 5;

Q. 33 write query to determine the n^{th} (say n=5) highest salary from a table
→ SELECT * FROM worker ORDER BY SALARY DESC LIMIT 4, 1;

4	500 000	Amit	not take
5	500 000	vivek	
3	300 000	vishal	
6	200 000	Vipul	→ i take → if there is 4, 2
1	100 000	monika	

means first 4 not take from it to further 2 take

] → 2 take

Q. 34 write query to determine the 5th highest salary without using LIMIT keyword
→ SELECT WORKER_ID, SALARY FROM worker AS w1 WHERE 4 = (
SELECT count(DISTINCT(w2.salary)) FROM worker AS w2
WHERE w2.salary >= w1.salary);

Q. 35 write query to fetch the list of employee with the same salary
→ SELECT w1.* FROM worker AS w1 INNER JOIN worker AS w2 USING (salary)
WHERE w1.Worker_id <> w2.worker_id;

→ SELECT w1.* FROM worker AS w1, worker AS w2 WHERE
w1.salary = w2.salary AND w1.worker_id != w2.worker_id;

Q. 36 write query to show second highes salary from table using sub-query
→ SELECT max(salary) FROM worker
WHERE salary not in (SELECT max(salary) FROM worker);

Q. 37 write query to show one row twice in result from a table
→ SELECT * FROM worker UNION ALL SELECT * FROM worker ORDER BY worker_id;
union → common once
Union All → common how many occured in both

Q. 38 write query to list worker_id who does not get bonus.
→ SELECT WORKER_ID FROM worker WHERE WORKER_ID NOT IN (
SELECT WORKER_REF_ID FROM bonus);

Q. 39 write query to fetch the first 50% records from a table.
→ SELECT * FROM WHERE WORKER_ID <= (SELECT count(WORKER_ID)/2 FROM
worker);

Q. 40 write query to fetch the departments that have less than 4 people in it.

→ SELECT DEPARTMENT, count(DEPARTMENT) FROM worker GROUP BY DEPARTMENT HAVING COUNT(DEPARTMENT) < 4;

Q. 41 write query to show all departments along with the number of people there

→ SELECT DEPARTMENT, count(DEPARTMENT) FROM worker GROUP BY DEPARTMENT;

Q. 42 write query to show the last record from a table.

→ SELECT * FROM worker WHERE WORKER_ID = (SELECT max(WORKER_ID) FROM worker);

Q. 43 write query to fetch the first row of a table

→ SELECT * FROM worker WHERE WORKER_ID = (SELECT min(WORKER_ID) FROM worker);

Q. 44 write query to fetch the last five records from a table

→ (SELECT * FROM worker ORDER BY WORKER_ID DESC LIMIT 5) ORDER BY WORKER_ID;

* Q. 45 write query to print the name of employee having the highest salary in each department.

→ SELECT w.department, w.first_name, w.salary FROM
(SELECT max(SALARY) AS maxsal, department FROM worker GROUP BY department) temp
INNER JOIN worker w on temp.department = w.department AND temp.maxsal = w.salary;

* Q. 46 write query to fetch three max salaries from a table using co-related subquery

→ SELECT distinct salary FROM worker AS w1 WHERE 3 >= (
SELECT count(distinct salary) FROM worker AS w2 WHERE w1.salary <= w2.salary
ORDER BY w1.salary DESC;

→ SELECT DISTINCT (SALARY) FROM worker ORDER BY SALARY DESC LIMIT 3;

* Q. 47 write query to fetch three min salaries from a table using co-related subquery

→ SELECT distinct salary FROM worker w1 WHERE 3 >= (
SELECT count(distinct salary) FROM worker w2 WHERE w1.salary >= w2.salary)
ORDER BY w1.salary DESC;

Q. 48 write query to fetch the n^{th} max salaries from a table

→ SELECT distinct salary FROM worker w1 WHERE n >= (
SELECT count(distinct salary) FROM worker w2 WHERE w1.salary <= w2.salary)
ORDER BY w1.salary DESC;

Q. 49 write query to fetch departments along with the total salaries paid for each of them

→ select department, sum(salary) FROM worker GROUP BY department ORDER BY sum(salary);

Q. 50 write query to fetch the name of workers who earn the highest salary

→ SELECT first_name, salary FROM worker WHERE salary = (
select max(salary) FROM worker);

Last question

Pairs

A	B
1	2
2	4
2	1
3	2
4	2
5	6
6	5
7	8

A	B
1	2
2	4
5	6

A	B
2	1
4	2
6	5

A	B
1	2
2	4
3	2
5	6
7	8

Q.1 remove reversed pairs JOINS

→ SELECT lt.* FROM pairs AS ls LEFT JOIN pairs AS rs ON lt.A = rt.B AND lt.B = rt.A WHERE rt.A IS NULL OR lt.A < rt.A;

Q.2 print only that pairs which are not repeated

→ SELECT lt.* FROM pairs AS ls LEFT JOIN pairs AS rs ON lt.A = rt.B AND lt.B = rt.A WHERE rt.A IS NULL AND rt.B IS NULL;

A	B
3	2
7	8

Q.3 remove reversed pairs (Correlated subquery)

→ SELECT * FROM pairs AS P1 WHERE NOT EXISTS (
SELECT * FROM pairs AS P2 WHERE P1.B = P2.A AND P1.A = P2.B AND P1.A > P2.A);

Normalisation

Union A∪B

intersection A∩B

- normalisation is a step towards DB optimisation

Functional Dependency FD

- it's relationship between the primary key attribute (usually) of the relation to that of the other attribute of the relation
- $X \rightarrow Y$ left side X = Determinant, right side Y = Dependent

Types of FD

① Trivial FD : $A \rightarrow B$ has trivial functional dependency if B is a subset of A
$A \rightarrow A$, $B \rightarrow B$ are also trivial FD

② Non-trivial FD : $A \rightarrow B$ has a non-trivial functional dependency if B is not a subset of A. (A intersection B is NULL)

Rules of FD Armstrong's axioms

1) Reflexive
- if 'A' is a set of attributes and 'B' is subset of 'A', then $A \rightarrow B$ holds
- if $A \supseteq B$ then $A \rightarrow B$

2) Augmentation
- if B can be determined by A, then adding an attribute to this functional dependency won't change anything
- if $A \rightarrow B$ holds, then $AX \rightarrow BX$ holds too. 'X' being a set of attributes

3) Transitivity
- if A determines B and B determines C, we can say that A determines C.
- if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

Why normalisation

- to avoid redundency in DB, not to store redundent data.

- what happen if we have redundent data
  - insertion, deletion, updation anomalies arises

Anomalies

1) Insertion anomaly
- when certain data (attribute) can not be inserted into the DB without the presence of other data

2) Deletion anomaly
- the deletion anomaly refers to that situation where the deletion of data results in the unintended loss of some other important data.

3) Updation anomaly or modification anomaly
- the update anomaly is when an update of a single data value requires multiple rows of data to be updated.
- due to updation to many places, may be data inconsistency arises, if one forgets to update the data at all the intended places.

- due to this anomalies, DB size increases and DB performence become slow
- to rectify these anomalies and the effect of these of DB, will we use database optimisation techniques called normalisation.

What is normalisation

- normalisation is used to minimise the redundency from a relation. it is also used to eleminate undesirable characteristics like insertion, update, deletion anomalies
- normalisation divides the composite attributes into individual attributes or larger table into smaller and links them using relationship
- the normal form is used to reduce redundency from the database table.

Types of normal forms

1NF, 2NF, 3NF, BCNF

1 NF

- every relation cell must have atomic value
- relation must not have multi-valued attributes

2 NF

- relation must be in 1NF
- there should not be any partial dependency
  - all non-prime attributes must be fully dependent on PK
  - non-prime attributes cannot depend on the part of the PK

3 NF

- relation must be in 2NF.
- no transitivity dependency exists
- non-prime attributes should not find a non-prime attributes.

BCNF boyce-codd normal form

- relation must be in 3NF
- FD : A → B , A must be a super key
- we must not derive prime attribute from any prime or non-prime attribute

Advantages of normalisation

- normalisation helps to minimise data redundancy
- greater overall database organisation
- data consistency is maintained in DB

Transaction

Transaction

- A unit of work done against the DB in a logical sequence
- sequence is very important in transaction
- it is a logical unit of work that contains one or more SQL statements, the result in all this statements in a transaction either gets completed sucessfully (all the changes made to the DB are perment) or if at any point any failure happens it gets rollbacked (all the changes being done are undone)

```
read (A)
A = A-50
write (A)
read (B)
B = B+50
write (B)
```

ACID properties

- to ensure entigrity of the data, we require that the DB system maintain the following properties of the transaction.

1) Atomicity

- either all operations of transaction are reflected properly in the DB, or none

2) Consistency

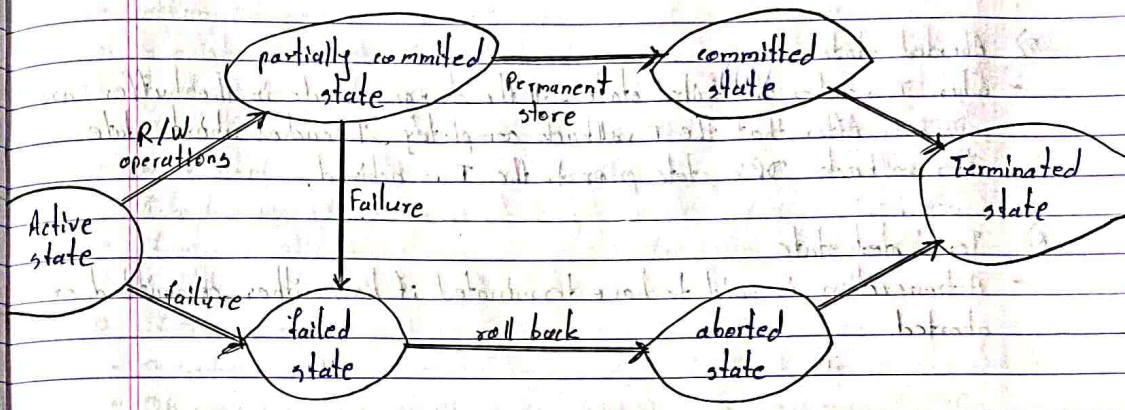
- integrity constraints must be maintained before and after transaction.
- DB must be consistent after transaction happens

3) Isolation

- even though multiple transaction may execute correctly, the system guarentees that, for every pair of transaction T_i and T_j, it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished. Thus, each transaction is unaware of other transaction executing concurrently in the system.
- multiple transaction can happen in the system in isolation, without interfering each other.

4) Durability

- after transaction completes sucessfully, the changes it had made to the database persist, even if there are system failure.



1) Active state

- first step. all the read and write operations are being performed. if they execute without any error the T comes to partially committed state, although if any error occurs the it leads to a failed state.

2) Partially committed state

- after transaction is executed the changes are saved in the buffer in the main memory. if the changes made are permanent on the DB then the state will transfer to committ state and if there is any failure, the T will go to fail state.

3) comitted state

- when updates are made permanent on the DB. the the T is said to be in comitted state. rollback can't be done from the commit state. new consistent state is achived at this stage.

4) Failed state
- when T is being executed and some failure occurs due to this, it is impossible to continue the execution of the T.

5) Aborted state
- when T reaches the failed state, all the changes made in the buffer are reversed. After that the T rollback completely. T reaches abort state after rollback. DB's state prior to the T is achived.

6) Terminated state
- A transaction is said to have terminated if has either committed or aborted.

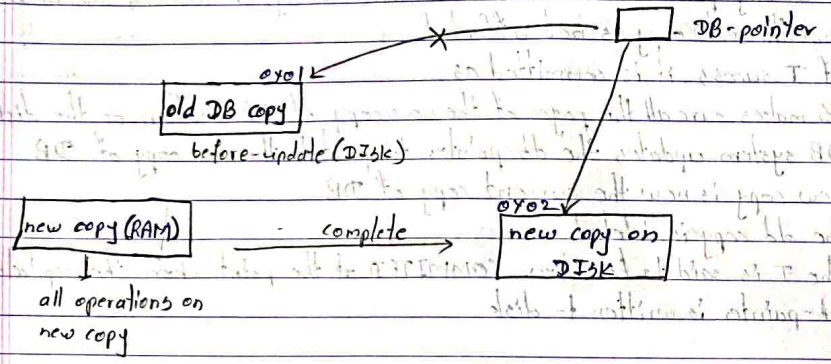
How to implement atomicity and durability in transactions
- Recovery mechanism component of DBMS supports atomicity and durability

a. Shadow-copy schema
- based on making copies of DB aka, shadow copies.
- assumption only one transaction (T) is active at a time
- a pointer called DB pointer is maintained on the disk, which at any instante points to current copy of DB
- T, that wants to update DB first create a complete copy of DB.
- all further updates are done on new DB copy leaving the original copy (shadow copy) untouched.
- if at any point the T has to be aborted the system deletes the new copy. and the old copy is not affected
- if T sucess, it is committed as,
- OS makes sure all the pages of the new copy of DB written on the disk
- DB system updates the db-pointer to point to the new copy of DB
- new copy is now the current copy of DB
- the old copy is deleted
- the T is said to have been COMMITTED at the point where the updated db-pointer is written to disk

a. Atomicity
- if T falls at any time before db-pointer is updated, the old content of DB are not affected
- T abort can be done by just deleting the new copy of DB
- Hence, either all updates are reflected or none

a. Durability
- suppose, system fails are any time before the updated db-pointer is written to disk
- when the system restarts, it will read db-pointer and will thus, see the original content of DB and none of the effects of T will be visible
- T is assumed to be successful only when db-pointer is updated.

- if system fails after db-pointer has been updated. before that all the pages of the new copy were written to disk. hence, when system restarts, it will read new db copy
-
- the implementation is dependent on write to the db-pointer being atomic. luckily, disk system provide atomic updates to entire block or at least a disk sector. so we make sure db-pointer lies entirely in a single sector. By sorting db-pointer at the beginning of a block
- inefficient, as entire DB is copied for every transaction



- Log-based recovery methods
- the log is a sequence of records. log of each transaction is maintained in some, stable storage so that any failure occurs, then it can be recovered from there
- if any operation is performed on the database, then it will be recorded in the log
- but the process of sorting the logs should be done before the actual transaction is applied in the database.
- stable storage is a classification of computer data storage technology that guarantees atomicity for any given write operation and allows software to be written that is robust against some hardware and power failures

A	B	logs (stable storage)	Immediate DB modifications.
₹ 1000	₹ 2000		
read (A)		< To start >	< To start >
A = A - 50		< To, A, 950 >	< To, A, 1000, 9500 > 1000 → old time value
write (A)		< To, B, 2050 >	< To, B, 2000, 2050 > 9500 → new time value
B = B + 50		< To commit >	< To commit >
write (B)			

first write logs the execute the operations. if failure occurs we have to know which step are performing and if fails

- Deferred DB modifications
- ensuring atomicity by recording all the DB modification in the log but deferring the execution of all the write operation until the final action of the T has been executed
- log information is used to executing deferred writes when T is completed
- if system crashed before the T completes, or if T aborted, the information in the logs are ignored
- if T complets, the records associated to it in the log file are used in executing the ~~execution of all the write operations~~ deferred writes
- if failure occurs while this updating is taking place, we perform redo

- Immediate DB modifications
- DB modifications to be output to the DB while the T is still in active state
- DB modifications written by active T are called uncommitted modifications.
- in the event of crash or T failure, system uses old value field of the log records to restore modified values.
- update takes place only after log records in a stable storage

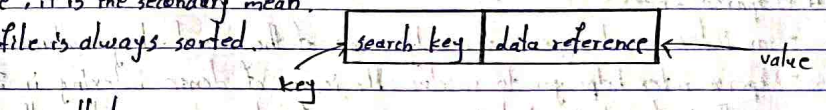
- Failure handling
- system failure before T completes, or if T aborted, then old value field is used to undo the T.
- if T completes and system crashes, then new value field is used to redo T having commit logs in the logs

□ Check points

- the check point is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk
- the check point is like a bookmark. while the execution of transection, such check points are marked and transection is executed then using the steps of the transaction, the log files will be created
- when it reaches to the check point, then the transaction will be updated in to the DB. and till that point. the entire log file will be removed from the file then the log file is updated with the new step of transaction till next checkpoint and so on
- the checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transaction were comitted.

Indexing in DBMS

- Indexing is used to optimise the performence of database by minimising the number of disk access required when a query is processed
- the index is a type of data structure. it is used to locate and access the data in database table quickly
- Speeds up operation with read operations like SELECT query, WHERE clause etc.
- Search Key : contains copy of primary key or candidate key of the table or something else.
- Data Reference : pointer holding the address of disk block where the value of the corresponding key is stored.
- indexing is optional. but increases access speed. it is not the primary meat to access the tuple, it is the secondary mean.
- Index file is always sorted.



□ Indexing methods.

① Primary Index - clustering index

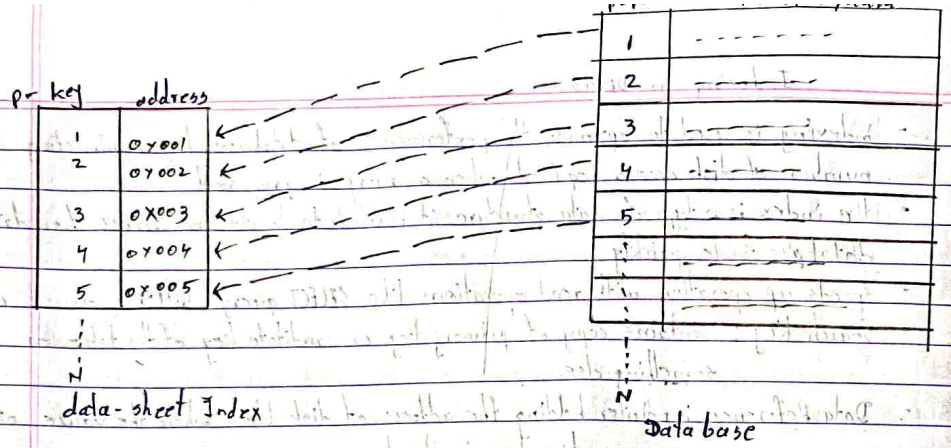
- A file may have several indices, on different search keys. if the data file containing the records is sequentially ordered, a primary index is an index whose search key also defines the sequential order of the file

Note - the term primary index is sometimes used to mean an index on primary key. However such usage is nonstandard and should be avoided.

- all files are ordered sequentially on some search key. it could be primary key or non-primary key.

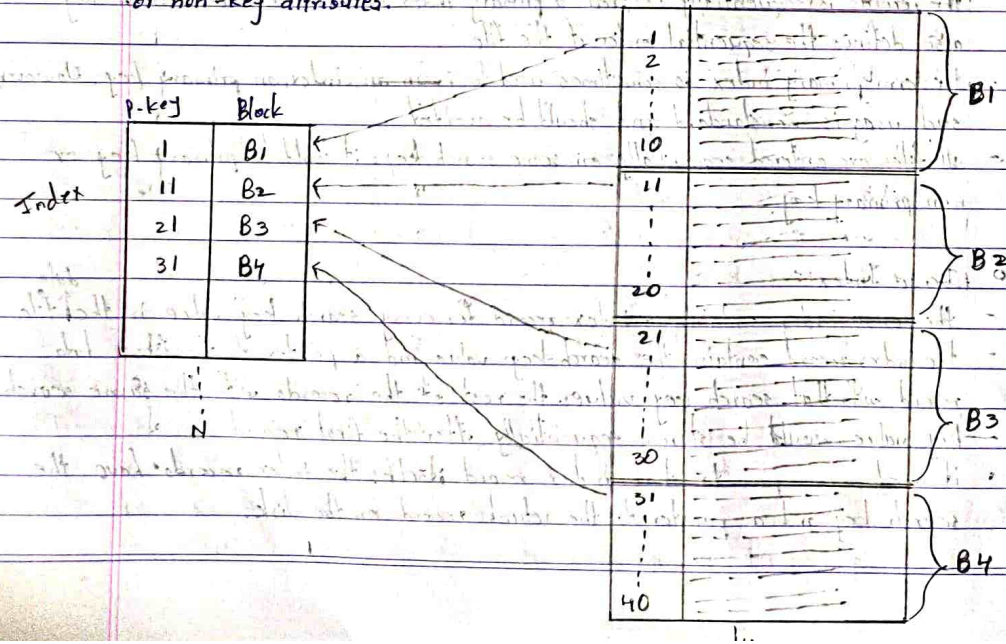
○ Dense Index

- the dense index contains an index record for every search key value in the data file
- the index record contains the search-key value and a pointer to the first data record with that search key value. the rest of the records with the same search key value would be stored sequentially after the first record
- it needs more space to store index record itself. the index records have the search key and a pointer to the actual record on the disk.



4 Sparse Index

- an index record appears for only some of the search-key values
- sparse index helps you to resolve the issues of dense indexing in DBMS, in this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrived, the block address will be fetched
- primary indexing can be based on data file is sorted w.r.t primary key attribute or non-key attributes.

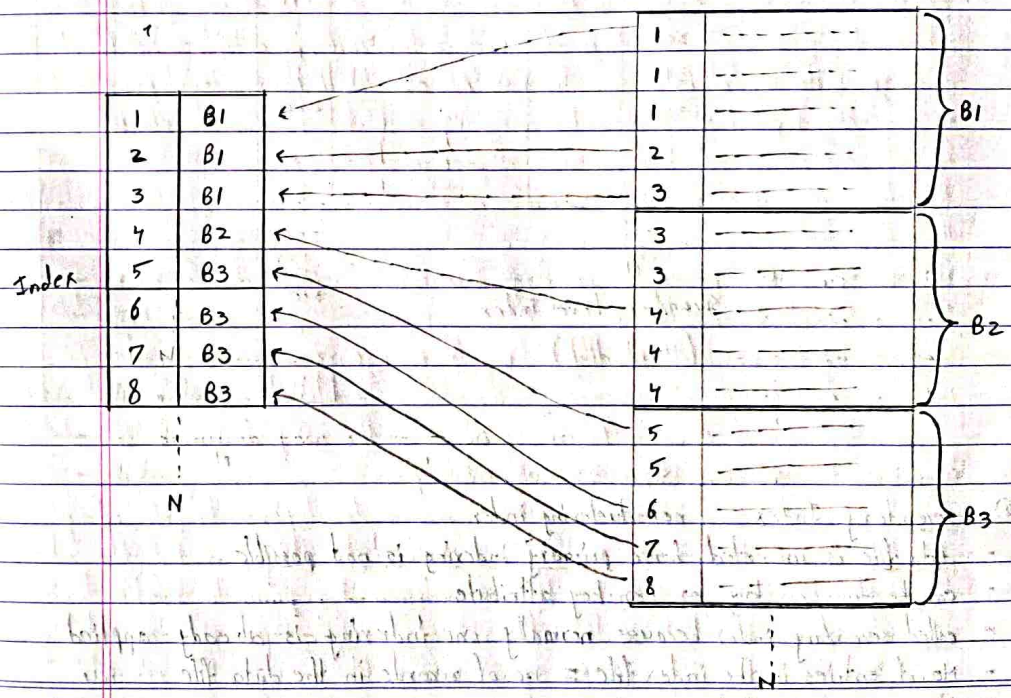


- Based on key attribute
  - data file is sorted w.r.t primary key attribute
  - P-K will be used as search-key in index
  - sparse index will be formed, i.e. no. of entries in the index file = no. of blocks in data

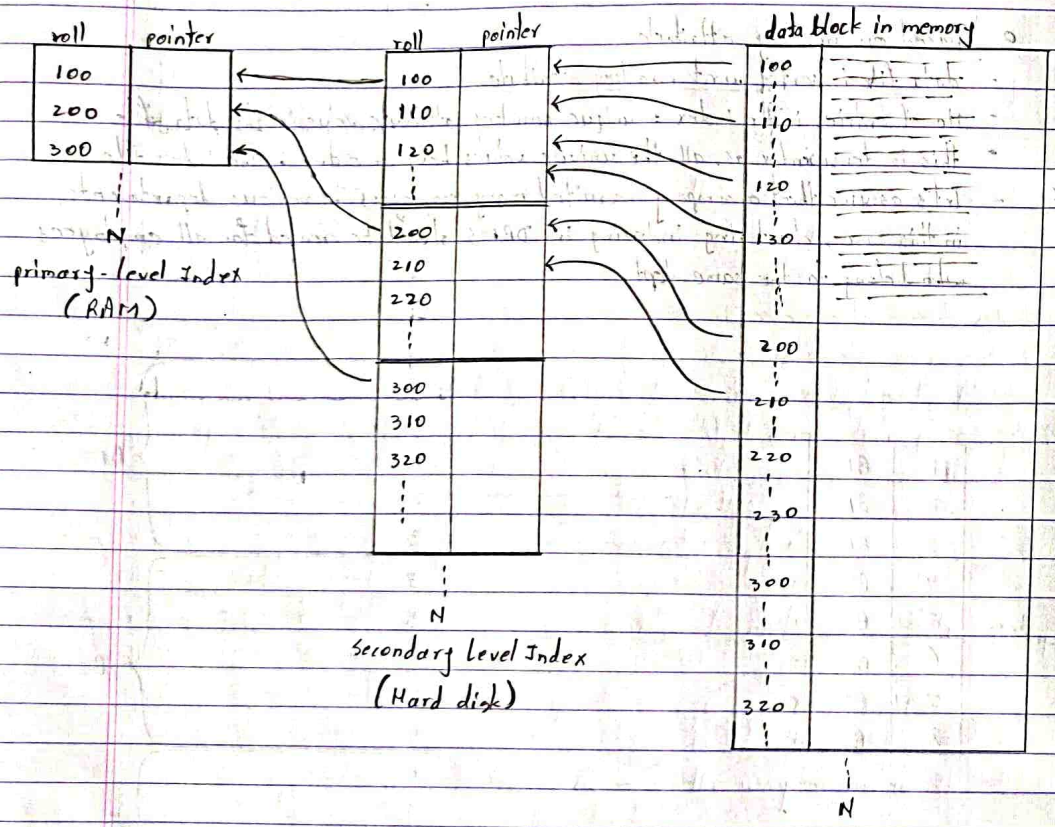
- Based on non-key attribute
  - data file is sorted w.r.t non key attribute.
  - No. of entries in the index = unique non-key attribute value in the data file
  - this is dense index as, all the unique values have an entry in the index file

Eg - let's assume that a company recruited many employees in various departments. in this case, clustering indexing in DBMS should be created for all employees who belong to the same dept.

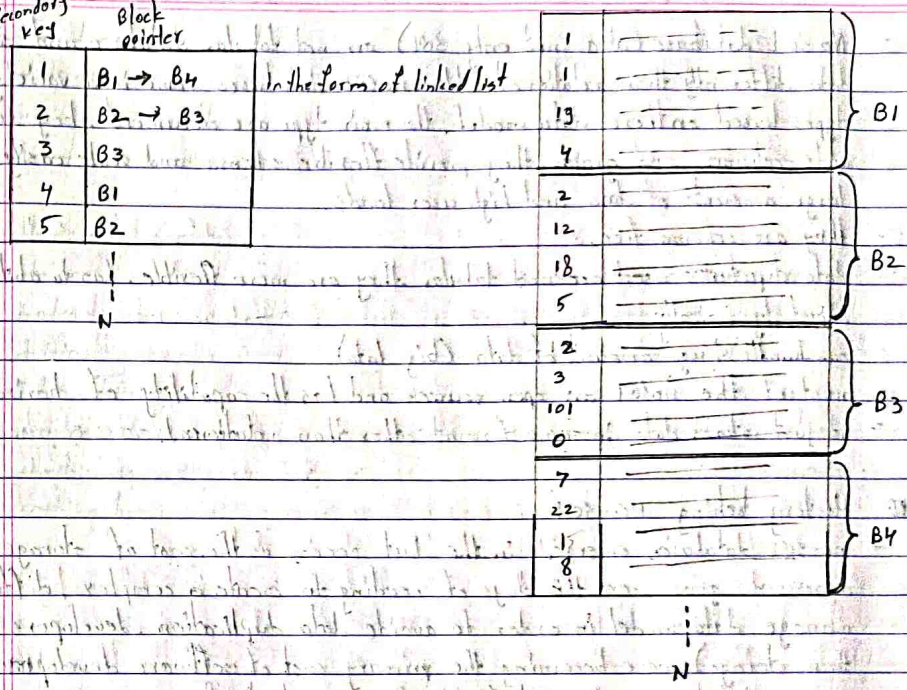
index stores first occurence in DB



- multi-level index
- index with two or more levels
- if the single level index become enough large that the binary search it self would take much time, we can break down indexing into multiple levels.



② Secondary Index non-clustering index
- data file is unsorted. hence primary indexing is not possible
- can be done on key or non-key attribute
- called secondary index because normally one indexing is already applied
- No. of entries in the index file = no. of records in the data file
- it's an example of dense index



- Advantages of Indexing
  - fastest access and retrival of data
  - IO is less
- Limitations of Indexing
  - additional space to store index table
  - indexing decreases performence in INSERT, DELETE, and UPDATE query

No SQL

- No SQL database (aka "not only SQL") are not tabular database and store data differently than relational table. nosql databases come in a variety of types based on there data model. the main type are document, key-value, wide-column, and graph. they provide flexible schema and scale easily with large amounts of data and high user loads.
- they are schema free
- data structures used are not tabular. they are more flexible. has to ability to adjust dynamically
- can handle huge amount of data (big data)
- most of the nosql are open sources and has the capability of horizontal scaling
- it just stores data in some format other than relational.

History behing NO-SQL

- no-SQL databases emerged in the last 2000s as the cost of storage dramatically decreased. gone were the days of needing to create a complex, difficult-to-manage data model in order to avoide data duplication. developers (rather than storage) were becoming the primary cost of software development, so NOSQL databases optimized for developer productivity
- date becoming unstructured more. hence structuring (defining schema in advance) them had becoming costly
- NOSQL allows developer to store huge amount of unstrudured data, giving them a lot of flexibility
- recognising the need to rapidly adapt to changeing requirements in a software system, developers needed the ability to iterate quickly and make changes throughout their software stack - all the way down to the database. NOSQL databases gave them their flexibility
- cloud computing also rose in popularity, and developers began using public clouds to host their applications and data. they wanted the ability to distribute data across multiple servers and regions to make their applications resilient, to scale out instead of scale up, and to intelligently geo-place their data. Some NOSQL databases like mongo-DB provide these capabilities.

NO-SQL databases Advantages.

① Flexible Schema
- RDBMS has pre-defined schema, which become an issue when we do not have all the data with us or we need to change the schema. it's a huge task to change schema on the go.

② Horizantal Scalling
- horizontal scalling, also known as scale-out, refers to bringing on additional nodes to share the load. this is difficult with relational databases due to the difficulty in spreading out related data accross nodes. with non-related DB's. this is made simpler since collections are self-contained and not coupled relationally; this allows them to be distributed across nodes more simply, as queries do not have to "join" them together across nodes
- scalling horizontally is achived through Sharding OR Replica-sets

③ High Availability
- NOSQL DB's are highly available due to it's auto replication feature. i.e. whenever any kind of failure happens data replicates itself to the preceding consistent state.
- if a server fails, we can access that data from another server as well, as in NOSQL database data is stored at multiple servers

④ Easy Insert & Read operations
- queries in NOSQL databases can be faster than SQL database. why? data in SQL databases is typically normalised. so queries for a single object or entity require you to join data from multiple tables. as your table grow in size, the joins can become expensive. however, data in NOSQL database is typically stored in a way that is optimised for queries. the rule of thumb when you use MongoDB is data that is accessed together should be stored together, queries typically do not have or require joins. so the queries are very fast.

- but difficulty delete or update operations.

- Caching mechanism
- NO-SQL use case is more for cloud applications.

When to use NO-SQL ?
- fast-paced agile development
- storage of structured and semi-structured data
- huge volume of data
- requirements for scale-out architecture.
- modern application paradigms like micro-services & real-time streaming

NO-SQL DB Misconceptions
roomer - relationship data is best suited for relational databases
reality - A common misconcept is that NO-SQL databases or non-relational databases don't store relationship data well. NO-SQL databases can store relationship data. - they just store it differently than relational databases do. in fact, when compared with relational databases, because related data doesn't have to split between tables. NO-SQL data models allow nested within a single data structure.
roomer - NO-SQL databases don't support ACID transactions.
reality - another common misconcept is that NO-SQL databases don't support ACID transactions. some NO-SQL databases like mongoDB do, in fact, support ACID transactions.

Types of NO-SQL data models
① Key-value stores
- the simplest type of NO-SQL database is a key-value store. every data dement in the database is stored as a key value pair consisting of an attribute name (or "key") and a value. in a sense, a key-value store is like a relational database with only two columns: the key or attribute name (such as "state") and the value (such as "Alaska")
- use cases include shopping carts, user preferences, and user profiles.
- Eg. Oracle NO-SQL, Amazon Dynamo DB, Mongo-DB also supports key-value store, redis.
- a key-value associates a value (which can be anything from a number or simple string to a complex object) with a key, which is used to keep track of the object. in it's simplest form, a key-value store is like a dictionary / array / map object as it exists in most programming paradigms, but which is stored in a persistent way and managed by a database management system (DBMS)
- key-value databases use compact, efficient index structures to be able to quickly and reliable locate a value by its key, making them ideal for systems that need to be able find an receive data in constent time
- there are several use-cases where choosing a key value store approach is an optimal solution.
1) real time random data access eg - user session attributes in an online application such as gaming of finance
2) caching mechanism from for frequently accessed data or configuration based on key
3) application is designed on simple key-based queries

② Column-Oriented / Columnar / C-store / Wide-column
- the data is stored such that each row of the column will be next to other rows from that same column.
- while a relational database stores data in rows and reads data row by row, a column store is organised as a set of columns. this means that when you want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data. columns are often of the same type and benifit from more efficient compression. making reads even faster. columnar databases can quickly aggregate the value of a given column (adding up the total sale for the year, for example)
Use cases include analytics
Eg - Cassandra, Redshift, Snowflake.

③ Document based storage
- this DB stores data in documents similar to JSON (Java-script object notation) objects. each document contains pairs of fields and values. the values can typically be a variety of types including things like strings, numbers, boolean, arrays or objects
- use cases include e-commerce platforms, trading platform, and mobile app development across industries.
- supports ACID properties hence, suitable for transactions
Eg - Mongo DB, Couch DB

④ Graph based storage
- a graph database focuses on the relationship between data elements. each element is stored as a node (such as a person in a social media graph) the connections between elements are called links or relationships. In a graph database, connections are first-class elements of the database, stored directly. in relational databases, links are implied, using data to express the relationships
- a graph database is optimized to capture and search the connections between data elements, overcoming the overhead associated with JOINing multiple tables in SQL
- very few real-world business systems can survive solely on graph queries. As a result graph database are usually run alongside other more traditional databases.
- use cases include fraud detection, social networks, and knowledge graphs.

NO-SQL database Dis-advantages
- Data Redundancy
- since data models in NO-SQL databases are typically optimized for queries and not for reducing data duplication. NO-SQL databases can be larger than SQL databases. storage is currently so cheap that most consider this a minor drawback, and some NO-SQL databases also support compression to reduce the storage footprint.
- Update & Delete operations are costly
- all types of NO-SQL data model doesn't fulfill all of your application needs,
- depending on the NO-SQL database type you select, you may not be able to do all of your use case in a single database. for Eg → graph DB are excellent for analysing relationships in your data but may not provide what you need for everyday retrival of data such as range queries. when selecting a NO-SQL DB, consider what you use cases will be and if a general purpose DB like MongoDB would be a better option.
- Doesn't support ACID properties in general.
- Doesn't support data entry with consistency constraints

Difference between SQL & NO-SQL

	SQL DB	NO-SQL DB
Data storage model	tables with fixed rows and columns	Document : JSON documents, Key value : key-value pairs Wide column : tables with rows & dynamic columns Graph : nodes & edges
development history	development in the 1970s with a focus on reducing data duplication	developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile & DevOps practices
Examples	Oracle, MySQL, Microsoft SQL server, PostgreSQL	Document : Mongo DB & couchDB key-value : Redis & DynamoDB Wide-column : Cassandra & HBase Graph : Neo4j & Amazon Neptune
Schema	Fixed	Flexible
Scaling	Vertical (scale-up)	Horizontal (scale-out across commodity servers)

Primary Purpose	General Purpose	Document : general purpose, Key-value : large amounts of data with simple look up queries Wide column : large amounts of data with predictable query patterns. Graph : analyzing & traversing relationships between connected data.
ACID properties	Supported	Not supported, except in DB like MongoDB etc.
JOINS	Typically required	Typically not required
Data to object mapping	Required object-relational mapping	many do not require ORMs. MongoDB documents, map directly to data structures in most popular programming languages

L16

Types of database

① Relational databases
- based on relational model
- relational databases are quite popular. even though it was a system designed in the 1970s also. also known as relational database management system (RDBMS). relational databases commonly use structured query language (SQL) for operations such as creating, reading, updating, deleting data. relational databases store information is discrete tables, which can be JOINed together by fields known as foreign keys. for example, you might have a user table which contains information about all your users, and join it to a purchases table, which contains information about all the purchases they've made. MySQL. Microsoft SQL server, and Oracle are types of relational DB's
- they are ubiquitous, having acquired a steady user base since the 1970s
- they are highly optimized with working with structured data.
- they provide a stronger guarantee of data normalisation
- they use a well-known querying language through SQL
- Scalability issues (Horizontal scaling)
- data become huge. system become more complex

② Object oriented databases
- the object-oriented data model, is based on the object-oriented-programming paradigm, which is now in wide use. Inheritance, Object-identity and Encapsulation (information hiding), with methods to provide an interface to objects, are among the key concepts of Object-oriented programming that have found applications in data modeling. the object-oriented data model also supports a rich type system, including structured and collection types. while inheritance and, to some extent, complex types are also present in the E-R model, encapsulation and object-identity distinguish the object-oriented data model from the E-R model
- sometime the database can be very complex. having multiple relations, so, maintaining a relationship between them can be tedious at times.

- In object-oriented DB's data is treated as an object
- all bits of information come in one instantly available object package instead of multiple tables

* Advantages
- data storage and retrival is easy and quick
- can handle complex data relations and more variety of data types that standard relational databases.
- relatively friendly to model the advance real world problems.
- works with functionality of OOPs and Object Oriented languages

* Di's - advantages
- High complexity causes performance issues like read, write, update and delete operations are slowed down.
- Not much of a community support as isn't widely adopted as relational databases
- Does not support views like relational databases.

Eg - Object DB, Gemstone etc.

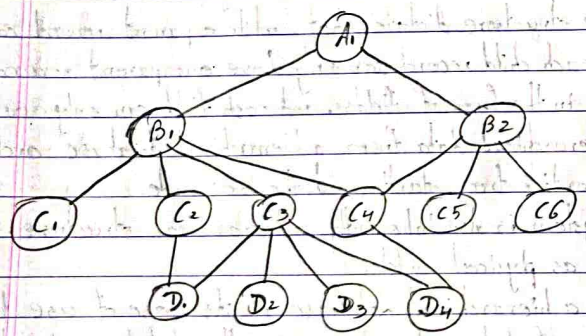
(3) NO-SQL databases
- NO-only SQL are non tabular database and store data differently than relational tables. NOSQL databases come in variety of types based on their data model. the main types are document, key-value, wide-column, and graph. they provide flexible schema and scale easily with large amounts of data and high user loads.
- they are schema free
- data structures used are not tabular, they are more flexible, has the ability to adjust dynamically
- can handle huge amount of data. (big data)
- most of the NOSQL are open sources and has the capability of horizontal scaling
- it just stores data in some format other than relational
-

(4) Hierarchical database
- as the name suggests, the hierarchical database is most appropriate for use cases in which the main focus of information gathering is based on a concerent hierarchy, such as several individual employees reporting to a single department at a company
- the schema for hierarchical database is defined by it's tree-like organisation. in which there is typically a root "parent" directory of data stored as records that links to various other subdirectory branches, and each subdirectory branch, or child record, may link to various other subdirectory branches
- the hierarchical database structure dictates that, while a parent record can have several child records, each child record can only have one parent record. data within records is stored in the form of fields, and each field can only contain one value. retrieving hierarchical data from a hierarchical database architecture requires traversing the entire tree, starting at the root node.
- since the disk storage system is also inherently a hierarchical structure, these models can also be used as physical models.
- the key advantage of a hierarchical database is its ease of use. the one-to-many organisation of data makes traversing the database simple and fast, which is ideal for use cases such as website drop-down menus or computer folders in systems like Microsoft Windows OS. due to the separation of the tables from physical storage structures, information can easily be added or deleted without affecting the entirety of the database. and most major programming languages offer functionality for reading tree structure databases
- the major disadvantage of hierarchical databases is their inflexible nature. the one-to-many structure is not ideal for complex structures as it cannot describe relationships in which each child node has multiple parents nodes. also the tree like organisation of data requires top-to-bottom sequential searching which is time consuming, and require repetitive storage of data in multiple different entities, which can be redundent

Eg - IBM, IMS

(5) Network databases
- extension of hierarchical databases
- the child records are given the freedom to associate with multiple parent records.
- Organised in a Graph structure
- can handle complex relations
- Maintenance is tedious
- M & N links may cause slow retrival
- Not much web community support

Eg - Integrated Data Store (IDS), IDMS (Integrated database management system), raima database manager, TurboIMAGE etc.



L17 Clustering in DBMS

- database clustering (making replica-sets) is the process of combining more than one server or instance connecting a single database. some time one server may not be adequate to manage the amount of data or the number of requests, that is when a data cluster is needed. DB clustering, SQL server clustering, and SQL clustering are closely associated with SQL is the language used to manage the databases information.
- replicate the same data set on different server.

Q Advantages

1) Data Redundency -
clustering with databases helps with data redundency, as we store the same data at multiple servers. don't confuse this data redundency repetition of the same data that might lead to some anomalies. the redundency that clustering offers is required and is quite certain due to the synchronisation. in case any of the servers had to face a failure due to any possible reason, the data is available at other servers to access

2) Load balancing -
or scalability doesn't come by default with the database. it has to be brought by clustering regularly. it also depends on the setup. basically, what load balancing does is allocating the workload among the different servers that are part of cluster. this indicates that more users can be supported and if for some reasons if a huge spike in the traffic appears, there is a higher assurance that it will be able to support the new traffic. one machine is not going to all get of the hits. this can provide scaling seamlessly as required. this links directly to high availability. without load balancing. a particular machine could get overworked and traffic would slow down, leading to decrement of the traffic to zero

3) High availability

- when you can access a DB. it implies that it is available. high availability refers the amount of time a database is considered available. the amount of availability you need greatly depends on the number of transactions you are running on your database and how often you are running any kind of analytics on your data. with database clustering. we can reach extremely high levels of availability due to load balancing and have extra machines. in case a server got shut down the databases will, however, be available

- How does clustering work
- in cluster architecture, all requests are split with many computers so that an individual user request is executed and produced by a number of computer systems. the clustering is servicable difinitely by the ability of load balancing and high-availability. if no node collapses, the request is handled by another node. consequently. there are few or no possibilities of absolute system failures.

Partitioning & Sharding in DBMS - DB optimisation

- A big problem can solve easily when it is chopped into several smaller sub-problems. that is what the partitioning technique does. it divides a big database containing data materials matrix and indxes into smaller and hardly slices of data called partitions. the partitioned tables are directly used by SQL queries without any alteration. once the database partitioned. the data definition language easily work on the smaller partitioned slices, instead of handling giant database altogather. this is how partitioning cuts down the problems in managing large DB problem table

- Partitioning
- is the technique used to divides stored database objects into separate server due to this there is an increase in performance, controllability of data. we can manage huge chunks of data optimally, when we horizontally scale our machines/servers, we know that it gives us a challanging time dealing with relational databases. as it's quite tough to maintain the relations. but if we apply partitioning to the database that is already scaled out. ie. equipped with multiple servers, we can partition our database among those servers and handle the big data easily.

- Vertical Partitioning
- Slicing relation vertically / column-wise
- need to access different servers to get complete tuples.

- Horizontal Partitioning
- slicing relation horizontally / row-wise
- independing chunks of data tuples are stored in different servers.

- When Partitioning is Applied
- data become much huge that managing and dealing with it become a tedious task
- the number of requests are enough larger that the single DB server access is taking huge time and hence the system's response time become high

- Advantages of partitioning
  - parallelism
  - Availability
  - Performance
  - Manageability
  - Reduce cost, as scalling-up or vertical scalling might be costly

- Distributed database
  - A single logical database that is spread accross multiple locations (servers) and logically interconnected by network
  - this is the product of applying DB optimisation techniques like Clustering, Partitioning, Sharding

- Sharding
  - technique to impliment horizontal partitioning
  - the fundamental idea of sharding is the idea that instead of having all the data sit on one DB instance, we split it up and introduce a routing layer so that we can forward the request to the right instances that actually contain the data.

- Pros
  - Scalibility
  - Availability

- Cons
  - complexity, making partition mapping, routing layer to be implemented in the system, Non-uniformality that creates the necessity of Re-Sharding
  - Nent well suited for analytical type of queries, as the data is spread accross different DB instances (Scatter-Gather problem)

Database Scalling Patterns

- A Case Study - cab booking App

  - Tiny startup
  - ~ 10 customer onboard
  - A single small machine DB stores all customers, trips, locations, booking data, and customer trip history
  - ~ 1 trip booking in 5 mins

- Your app becoming famous, but... the problem begins
  - requests scale upto 30 bookings per minute
  - Your tiny DB system has started performing poorly
  - API latency has increased a lot
  - Transactions facing deadlock, Starvation, frequent failure
  - Sluggish App experience
  - Customer dis-satisfaction

- Solutions
  - we have to apply some kind of performing optimisation measures
  - We might have to scaler our system going forward.

1. ✱ Query Optimisation & Connection Pool Implementation

  - cache frequently used non-dynamic data like booking history, payment history user profile etc.
  - introduce DB redundancy (Or may be use NOSQL).
  - use connection pool libraries to cache DB connections
  - multiple application threads can use same DB connection.
  - Good optimisation as of now

- Scaled the business to one more city, and now getting ~ 100 booking per minute

2. * Vertical Scaling or Scale-up

- Upgrading our initial tiny machine
- RAM by 2x & ssD by 3x etc.
- Scale up is pocket friendly till a point only
- more you scale up, cost increases exponentially
- Good optimisation as of now

• buisness is growing, you decided to scale it to 3 more cities and now getting 300 booking per minute

3. * Command Query Responsibility Segregation CQRS

- the scaled up big machine is not able to handle all read/write requests.
- separated read/write operation physical machine wise.
- 2 more machines as replica to the primary machine.
- all read queries to replicas
- all write queries to primary
- buisness is growing, you dicided to scale it 2 more cities.
- primary is not able to handle all write requests
- lag between primary and replica is impacting user experience.

4. * Multi Primary Replication

- why not distribute write request to replica also?
- all machines can work as primary & replica
- multi primary configuration is a logical circular ring
- write data to any node
- read data from any node that replies to the broadcast first

• you scale to 5 more cities and your system is in pain again @ 50 request/sec

5. * Partitioning of DATA by Functionality

- what about separating the location table in separate DB schema?
- what about putting that DB in separate machines with primary-replica or multi-primary configuration?.
- Different DB can host data categorised by different functionality.
- backend or application layer has to take responsibility to join the results
-

• planning to expand your business to other country.

6. * Horizontal Scaling or Scale-out

- Sharding - multiple shards
- allocate 50 machines - all having same DB schema - each machine just hold a part of data
- locality of data should be there
- each machine can have their own replicas, may be used in failure recovery.
- Sharding is generally hard to apply. But "No pain, No Gain"

• Scaling the business across continents

7 * Data Centre Wise Partition

- requests travelling across continents are having high latency
- what about distributing traffic accross data centres?.
- Data centres across continents
- Enable cross data centre replication which helps disaster recovery
- this always maintain availability of your system.
- Now, plan for an IPO

120 CAP theorem

- basic and one of the most imp concept in Distributed databases
- Useful to know this to design efficient distribution system for your given business logic
-

□ let's first breakdown CAP

● Consistency :
- in a consistent system, all node see the data simultaneously. if we perform a read operation on a consistent system, it should return the value of the most recent write operation. the read should causes all nodes to return the same data. all users see the same data at the same time, regardless of the node they conect to. when data is written to a single node, it is then replicated across the other nodes in the system.

● Availability :
- when availability present in a distributed system, it means that the system remains operational all of the time. every request will get response regardless of the individual state of the nodes. this mean that the system will operate even if there multiple nodes down, unlike consistent system, there's no guarante that the response will be the most recent write operation.

● Partition Tolerence :
- when a distributed system encounters a partition, it means that there's break in communication between nodes. if a system is partition-tolarent, the system doe's not fail, regardless of whether message are dropped or delayed between nodes within the system. to have partition tolarence, the system must replicate records across combinations of nodes and networks

□ What doe's the CAP Theorem says
- a distributed system can only provide two of three properties simultaneously - consistency, availability, partition tolarence. the theorem formalises the tradeoff between consistency & availability when there is a partition.

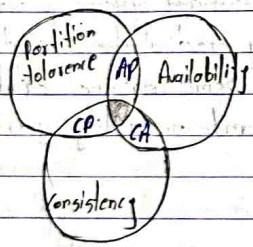
□ CAP Theorem NO SQL Databases
- nosql databases are great for distributed networks. they allow for horizontal scaling, and they can quickly scale across multiple nodes. when dicided which noSQL database to use, it's important to keep the CAP theorem in mind.

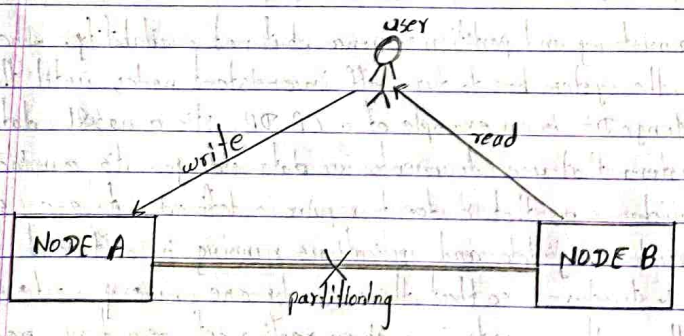
1) CA DB -
- CA DB enables consistency and availability across all nodes. unfortunately, CA DB can't deliver fault tolerence, in any distributed system, partition are bound to happen, which means this type of DB isn't very practical choice. that being said, you still can find a CA DB if you need one. Some relational databases, such as MySQL or PostgreSQL, allow for consistency and availability. you can deploy them to nodes using replication.

2) CP DB -
- CP DB enables consistency and partition tolarence, but not availability. when a partition occurs, the system has to turn off inconsistent nodes until the partition can be fixed. MongoDB is an example of a CP DB. it's a noSQL database management system. that uses documents for data storage. it's considered schema-less, which means that it doesn't require a defined database schema. it's commonly used in big data and applications running in different locations. the CP system is structured so that there's only one primary node that receives all of the write requests in a given replica set. secondary nodes replicate the data in the primary nodes, so if the primary node fails, a secondary node can stand-in. in banking system availability is not as imporlent as consistency, so we can opt it (Mongo DB)

3) AP DB -

- AP DB enable availability and partition tolerence. but not consistency. in the event of partition, all nodes are available. but they're not all updated. for Ex - if a user tries to access data from a bad node. they won't receive the most up-to-date version of the data. when the partition is eventually resolved, most AP databases will sync the nodes to ensure consistency across them. Apache Cassandra is an example of an AP database. it's NOSQL database with no priority node, meaning that all of the nodes remain available. cassandra allows for eventual consistency because users can re-sync their data right after a partition is resolved. for apps like Facebook, we value availability more than consistency, we'd opt for AP databases like Cassandra or Amazon DynamoDB





1) Consistency lost → both request should process
2) Availability lost → if we fail one request

ACID & Base Model

ACID Model

- collections of operations that form a single logical unit of work are called transactions and the DB system must ensure proper execution of transactions and the ACID DB transaction model ensures that a performed transaction is always consistent. to explain ACID in more detail and easy way is to understand through breaking down the acronym, ACID

1) Atomicity -

- this property states transaction must be treated as atomic unit, that is, either all of its operations are executed or non, and there must be no state in database where a transaction is left partially completed also the states should be defined either before the execution of the transaction or after the execution of the transaction

2) Consistency

- the DB must remain in a consistent state after any transaction also no transaction should have any adverse effect on the data residing in the DB and if the DB was in a consistent state before the execution of a transaction then it must remain consistent after the execution of the transaction as well

3) Isolation

- in a DB system where more than one transaction is being executed simultaneously and in parallel, the property of isolation states that each one of the transaction is going to be administered and executed as it is the only transaction in the system also no transaction will affect the existance of any other transaction

4) Durability

- the DB should be durable enough to hold all it's latest updates even if the system fails or restarts so, in a practical way of saying that if a transaction updates a chunk of data in a DB and commit is performed then the DB will hold the modified data but if the commit is not performed then no data is

modified and it can only be done when the system start. the DB's that are ACID compliant are the one safe way to make sure your DB's is ACID compliant is to choose a relational DB management system like MySQL, PostgreSQL, Oracle, SQlite, and microsoft SQL server. some NoSQL DBMSs, such as Apache's Couch DB or IBM's Db2, also processes a certain degree of ACID compliance (there are no recomendations of No SQL database if the requirement is of strict ACID rules)

□ Base Model

- the rise in popularity of NoSQL DB's provided a flexible and fluidity with ease to manipulate data and as a result, a new DB model was designed, reflecting these properties. the acronym BASE is slightly more confusing than ACID but however, the words behind it suggest ways in which the BASE model is different and acronym BASE stands for

1) Basically Available

- instead of making it compulsary for immediate consistency, BASE-modelled NO-SQL database will ensure the availability of data by spreading and replicating it across the nodes of the DB duster

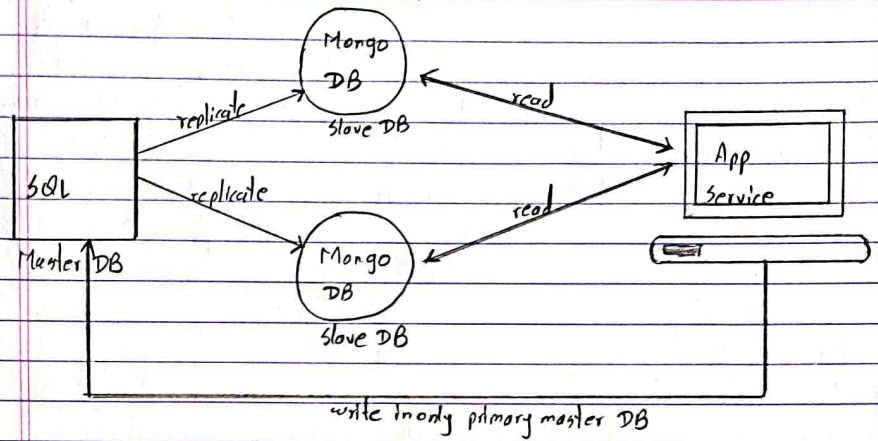
2) Soft State -

- due to the lack of immediate consistency, the data values may change over time the BASE value may change over. the BASE model breaks-off with the concept of a DB that obligates its own consistency, delegating that responsibility to developers

3) Eventually Consistent

- the fact that BASE does not obligates immediate consistency but it does not mean that it never achieves, it However, until it does, the data reads are still possible (even through they might not reflect reality)

The Master-Slave DB concept



- master slave is a general way to optimise IO in a system where number of requests goes way high that a single DB server is not able to handle it efficiently
- it's a pattern 3 in LEC-19 (DB scalling pattern) Command Query Responsibility Segregation
- the true and latest data is kept in the Master DB thus write operations are directed there. reading ops are done only from slaves. this architecture serves the purpose of safeguarding site Reliability, Availability, Reduce latency etc. if a site receives a lot of traffic and the only available DB is one master, it will be overloaded with reading & writting requests, making the entire system slow for everyone on the site.
- DB replication will take care of distributing data from master machine to slave machile. this can be Synchronous or Asynchronous depending upon the systems need.

banking — facebook

- in master slave it's strictly that we can't write is slave DB. M to S
- if it's able to write on slave the it called's M to M