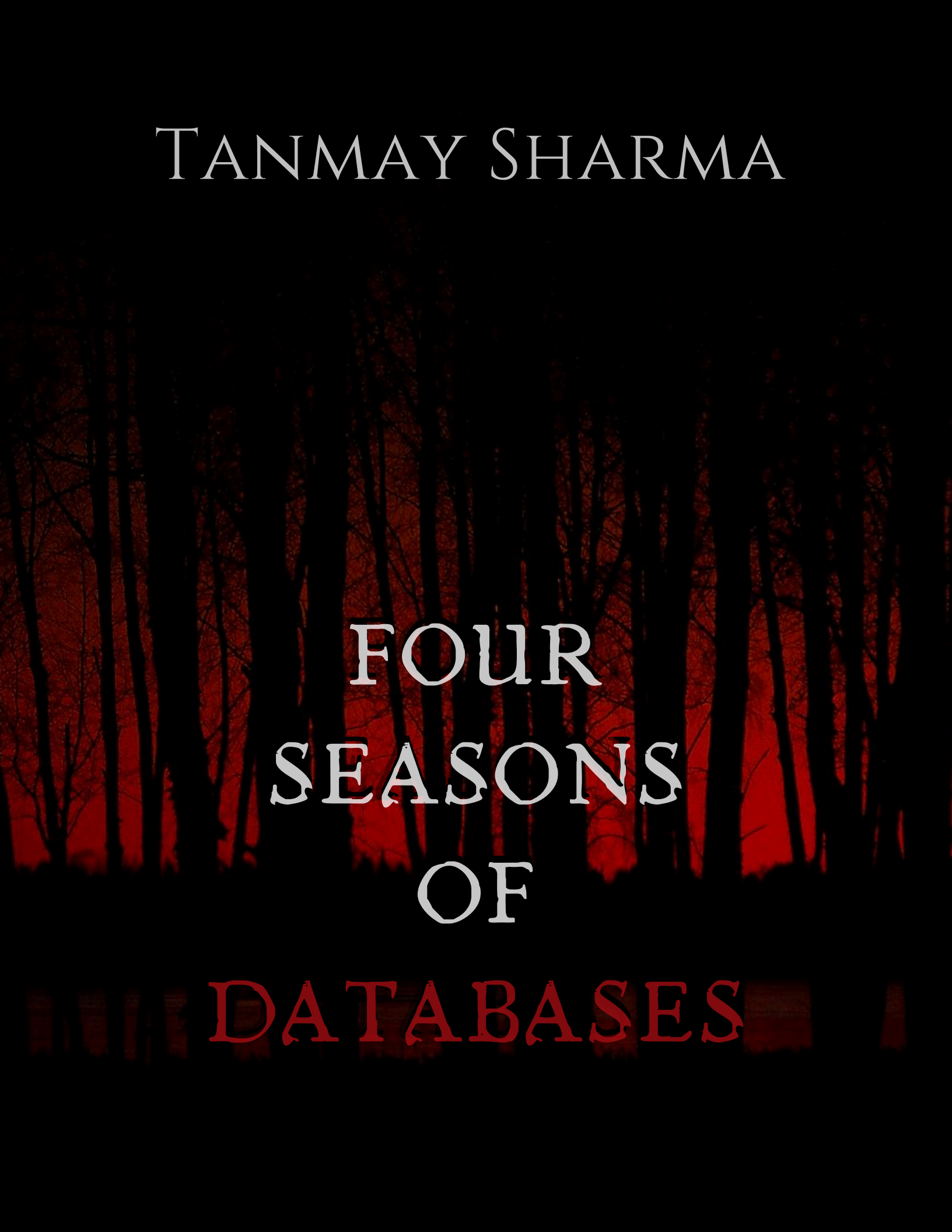
Tanmay Sharma



Four Seasons of Databases

Index

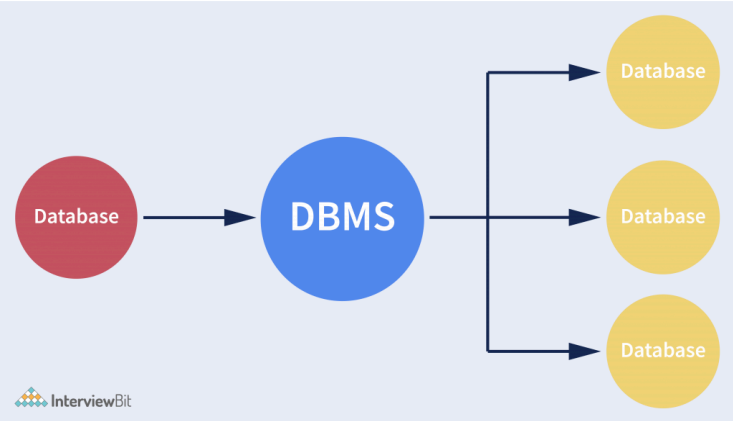
Introduction to Database Management Systems

A database management system (DBMS) allows users to manage and organize their data in a structured manner. It simplifies the processes of creating, reading, updating and deleting data entries using querying, as well as allowing control and access of specific data to specific users.

It provides a wholehearted environment where the user does not have to worry about designing algorithms to manipulate data and the location of their data, DBMS does that for them.

Key Features of DBMS

- **Data Modeling:**
  - DBMS provides tools to create and modify the models according to which the data is kept, which define the relations and structures of data.
- **Data Storage & Retrieval:**
  - DBMS is responsible for storing and retrieving data from a database.
- **Concurrency Control:**
  - DBMS provides services to handle concurrent requests happening at the same time to ensure that database stays in consistent state.
- **Data Integrity & Security:**
  - DBMS ensures that only right user can access the data and restrict those who are not allowed.
- **Backup & Recovery:**
  - DBMS provides mechanisms where copies of database are kept so as to ensure recovery of data in case of any failure.



We can say that a database is a collection of inter-related data stored in a schema. For eg, your university database stores your information, your name, enrollment, DoB, Aadhar number, PAN number, etc.

Types of DBMS

- **RDBMS:**
    - RDBMS is Relational Database Management System, where the related data is stored in a table based structure. Each row indicate a unique record and column indicates the attribute. If you change the database structure, you change it for all users.
- **OODBMS:**
    - ODBMS is Object-Oriented DBMS, it stores information in arbitrary structures, nested structures, and dynamically varying structures. These structures make up software objects or subobjects.
- **NoSQL:**
    - NoSQL database stores data in key-value pairs. Where each item denotes a unique record and every item can contain different-different attributes unlike RDBMS which has fixed schema for all records.

Database Languages

DBMS utilizes database languages that enable a user access and control a database. These languages are subdivided into four types based on their scope and usage. You can consider the roles of database languages as a functionality of a database.
Their types:

1. Data Definition Language
2. Data Manipulation Language
    a. Data Querying Language
3. Data Control Language
4. Transaction Control Language

Data Definition Language

DDL is the short name for Data Definition Language, which deals with database schemas and descriptions, of how the data should reside in the database.

CREATE: to create a database and its objects like (table, index, views, store procedure, function, and triggers).
ALTER: alters the structure of the existing database.
DROP: delete objects from the database.
TRUNCATE: remove all records from a table, including all spaces allocated for the records are removed.
COMMENT: add comments to the data dictionary.
RENAME: rename an object.

Data Manipulation Language

DML is the short name for Data Manipulation Language which deals with data manipulation and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE, etc., and it is used to store, modify, retrieve, delete and update data in a database. Data query language(DQL) is the subset of “Data Manipulation Language”. The most common command of DQL is SELECT statement. SELECT statement help on retrieving the data from the table without changing anything in the table.

SELECT: retrieve data from a database
INSERT: insert data into a table
UPDATE: updates existing data within a table
DELETE: Delete all records from a database table
MERGE: UPSERT operation (insert or update)
CALL: call a PL/SQL or Java subprogram
EXPLAIN PLAN: interpretation of the data access path
LOCK TABLE: concurrency Control

Data Query Language

Data query language(DQL) is the subset of “Data Manipulation Language”. The most common command of DQL is the SELECT statement. SELECT statement helps us in retrieving the data from the table without changing anything or modifying the table. DQL is very important for retrieval of essential data from a database.

Data Control Language

DCL is short for Data Control Language which acts as an access specifier to the database.(basically to grant and revoke permissions to users in the database

GRANT: grant permissions to the user for running DML(SELECT, INSERT, DELETE,…) commands on the table
REVOKE: revoke permissions to the user for running DML(SELECT, INSERT, DELETE,…) command on the specified table

Transactional Control Language

TCL is short for Transactional Control Language which acts as a manager for all types of transactional data and all transactions. Some of the command of TCL are

ROLLBACK: Used to cancel or Undo changes made in the database
COMMIT: It is used to apply or save changes in the database
SAVEPOINT: It is used to save the data on the temporary basis in the database

Applications of DBMS

DBMS is utilized everywhere, literally everywhere. You book your train tickets, flights, even your Aadhar information, in your university, finance industry, telecomm industry, amazon, flipkart, google, everywhere. Wherever data is, there is a DBMS.

Why not use File Systems?

Imagine you are using a file to store data about students in a university. You store important information about students (name, age, dob, enrollment number, contact info, grades, payment history and even in some cases medical records) in a file. And since there's no way you can structure a file so data is cluttered and only way to separate them is using a delimiter (as in comma-separated values, csv file).

The student is say studying in Computer Engineering department, and the department only want personal info and grades of that student, so there's no way you can select out some informations from the file easily. You have to create separate programs to access specific information.
To access personal information only, you will create a program.
To access contact information only, you will create another program.
To access grades only, you have to build a program again.
This is such a hectic task if you utilize a file.

Now, to cope up with the mess you've created above, you decide to copy data from central database and give it to all the departments. If the central database contains 10GB of data, and considering 15 separate departments, you will create 15 copies, 150GB of extra space.

Your one student in the university is quite unhappy with his grades and decides to hack into your so called "file-system" database. He can easily do that, cause there is no lock mechanism in linux or windows operating systems. All you can do is encryption.

Now, Computer Department just uploaded the grades of their students, and since you have multiple copies of same data, you have to apply changes in each copy.

With above example, I think you understood the importance of using a DBMS instead of file system. A file system has,

- No structure to store data.
- Inconsistent copies of data.
- Difficult data access (write program for each query).
- Unauthorized access.
- No concurrency among data.
- No service for backup and recovery.

A DBMS has all the services that aren't present in a file system.

History of Development of DBMS: https://www.geeksforgeeks.org/history-of-dbms/

Disadvantages of DBMS

1. **Complex Design:**
    a. Users have to learn to design views to implement a feature.
2. **Learn Querying:**
    a. To implement dbms and store data you have to learn querying language.
3. **SQL Injection:**
    a. Some may inject queries in their responses to attack a dbms server.
4. **Cost of Maintenance:**
    a. You have to hire people who write database languages and people who administer your data.
5. **Compatibility:**
    a. DBMS might not be available to all the required technologies to implement.

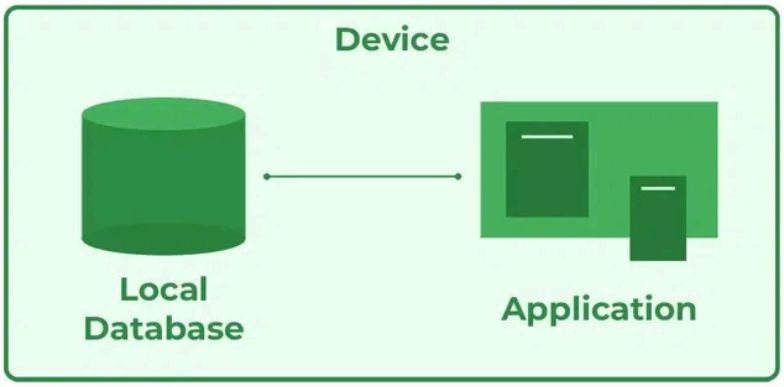
DBMS Architecture

Usage of DBMS can be different for every user, so it's not advised for every user to utilize same architecture. A corporation has different needs while a small shopkeeper has different. To tackle this problem, a user is advised to go through their needs and requirements and use the architecture beneficial for them.
Generally, DBMS Architectures are divided into three categories:

1. Tier 1
2. Tier 2
3. Tier 3

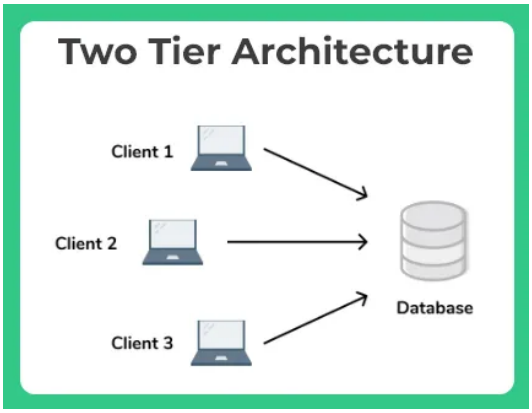
One-Tier Architecture

In tier 1 architecture, there is no need to set up a server elsewhere. The application and the database are available locally to the user. In here, same machine handles application processing and dbms processing.
Usecases, a shopkeeper who doesn't wants to use his red-book ledger and write entries manually. Eg, you go to buy medicines from local pharmacy, well they utilize a local database.
In some cases they might set up a local server on one machine and utilize that database through distributed machines locally.
Advantages, it's cost effective, easy to implement and quite simple architecture.



Two-Tier Architecture

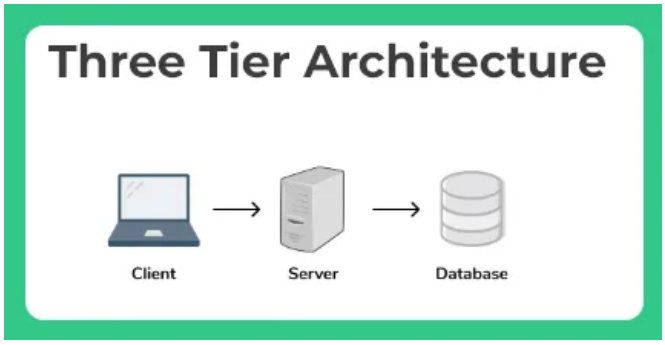
This architecture is like the **client-server model**, where a client queries through an application having UI and those queries are handled by a server remotely. Data is shared by utilizing APIs as in ODBC (Open Database Connectivity) and JDBC (Java Database Connectivity). In here, server handles dbms processing and client handles application processing.
The **disadvantage** with this type is that it cannot handle large number of requests as application will slow down in handling those number of requests. There is single server and no logic to redirect clients to other servers in case of large number of requests.



Three-Tier Architecture

This level of architecture is kept more distinct from tier 2 architecture. Here, the client-server model is made more simple and separated. There are three layers called, **Presentation-Application-Data** layers. This doesn't mean that there are three machines to handle processing. Machines can or cannot be three, here we divided the logistics into three logical layers to understand in a better manner.
Presentation layer presents the user interface, collects data (which is sent to Application layer). For example, your browser. It gives you UI, tables, forms, etc.
Application layer handles the business logic, the data collected from presentation layer, APIs and server redirections. This layer can be present along with presentation layer on local machine and as well with third layer or even kept separate. It uses caching and load balancing and then sends the data to nearest or most optimal server.

Data layer, is the database layer. It handles the simplified requests it receives from application layer and stores data or gives response as needed.



Let's take the example of **Google.com's architecture** to understand tier-three,

How Google Handles Millions of Users

Google's architecture is a complex and highly optimized system, but the general principles align with the concepts of Three-Tier Architecture and beyond. Here's how it manages millions of users:

- **Presentation Layer (Client Tier):**
  - **Role:** This is the user-facing part of Google, which includes the Google search page and other interfaces (e.g., Google Maps, Gmail).
  - **Users' Devices:** Users access Google's services through their web browsers or apps.

- **Application Layer (Business Logic Tier):**
  - **Role:** This layer handles the search queries, processes requests, and applies the business logic to generate search results. It includes the backend systems that process and rank search results, manage user sessions, and more.
  - **Load Balancing:** The application layer uses load balancers to distribute incoming search queries across multiple servers. This ensures that no single server gets overwhelmed and that users receive responses quickly.

- **Data Layer (Database Tier):**
  - **Role:** This layer stores the data that Google needs to perform searches, including web page indexes, user data, and other information.
  - **Data Management:** Google uses distributed databases and storage systems to handle vast amounts of data. Data is replicated and sharded across multiple servers to ensure high availability and fast access.

How Google Handles Scalability and Performance

- **Load Balancing:**
  - **Request Distribution:** When users search on Google, their queries are first routed to a load balancer. The load balancer distributes these queries to different application servers based on factors like current load, server health, and geographical location.
  - **Global Distribution:** Google has data centers around the world. The load balancer directs users to the nearest or most optimal data center, improving response times and reducing latency.

- **Caching:**
  - **Result Caching:** To speed up responses, Google caches frequently accessed data and search results. This reduces the load on the database and application servers by serving cached results for common queries.

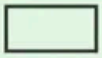
- **Distributed Systems:**
  - **Data Distribution:** Google's data is stored in a distributed manner across many servers. This allows for handling large volumes of data and ensures that the system can continue to operate even if some servers fail.
  - **Fault Tolerance:** If one server or data center fails, others can take over, ensuring continuous availability and reliability.

- **High Availability:**
  - **Redundancy:** Critical components are duplicated across multiple servers or data centers. This redundancy ensures that services remain available even in case of hardware failures or high traffic.

- **Continuous Optimization:**
  - **Performance Tuning:** Google continuously monitors and optimizes its infrastructure to handle increasing loads and improve efficiency. This includes upgrading hardware, refining algorithms, and deploying new technologies.

ER Models

Entity Relationship Models are a way to represent and identifying entities in a database. Entities represent real-world objects and their relations represent relationship among other entities. Attributes represent their properties each object hold. For eg, in a ***employee*** table, they store name, address, contact, email, employee id of a person and this holds true for each employee. Thus ***employee*** is an entity.
These ER diagrams have evolved into Enhanced ER Models and Object Relationship Models.
ER diagrams can easily be converted into table schema that we use in SQL.
These are quite easy to understand and does not require any high-level knowledge as pre-requisite.

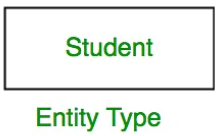
Figures	Symbols	Represents
Rectangle		Entities in ER Model
Ellipse		Attributes in ER Model
Diamond		Relationships among Entities
Line		Attributes to Entities and Entity Sets with Other Relationship Types
Double Ellipse		Multi-Valued Attributes
Double Rectangle		Weak Entity

Components of ER model,

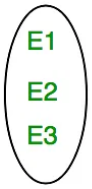
ER Model		
Entities	**Attributes**	**Relationships**
● Strong Entity ● Weak Entity	● Key Attribute ● Composite Attribute ● Multivalued Attribute ● Derived Attribute	● One to One ● One to Many ● Many to One ● Many to Many

Entity

Entity represents a real-world object, a single object. Complete set of entities of same type is called Entity-Set. Tangible entities represent real-world object whereas intangible does not.



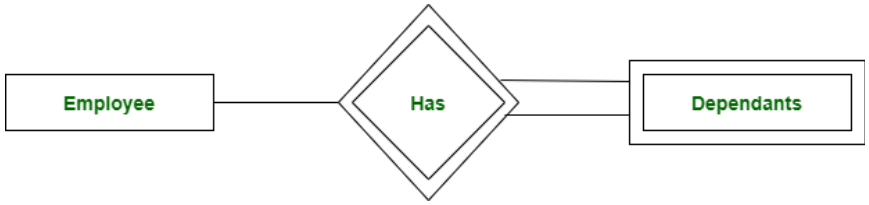
Entity Type



Entity Set

Types of Entities

1. **Strong Entity:**
    a. Strong entities are those which has a primary attribute of their own.
    b. They can uniquely be identified among a entity set.
    c. These are represented using a rectangle.
2. **Weak Entity:**
    a. Weak entities does not have primary attribute.
    b. They depend on a strong entity for their existence.
    c. They are represented using double rectangle.
    d. Participation of weak entity is always total.



Attributes

Attributes are the properties of an entity.

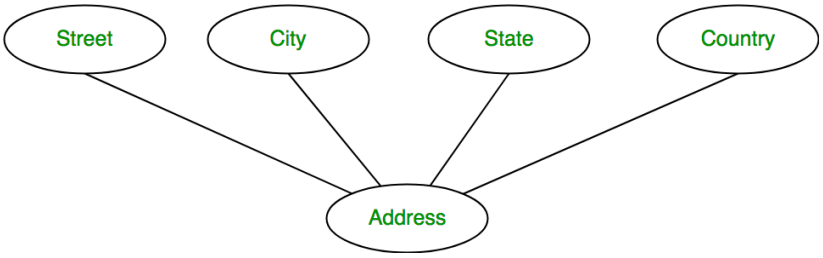
Types of Attributes

1. **Key Attributes:**
    a. Which uniquely identifies an entity.
    b. Eg, roll number.
2. **Composite Attribute:**
    a. Which is composed of many other attributes.
    b. Eg, address: house no., street, city, pin.
3. **Multivalued Attribute:**
    a. Which contains more than one value for same type.
    b. Eg, phone number.
4. **Derived Attribute:**
    a. Which can be derived from other attributes.
    b. Eg, age can be derived from DoB.

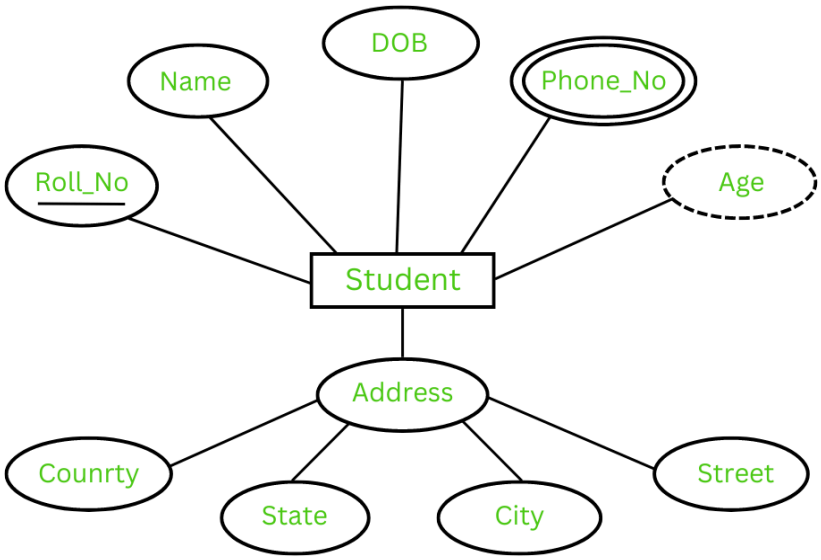
Key Attribute

Multivalued Attribute

Derived Attribute



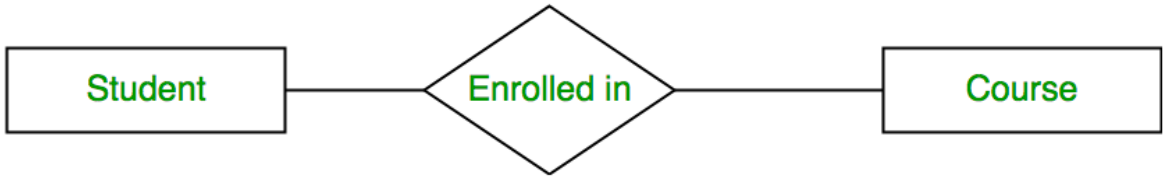
Composite Attribute



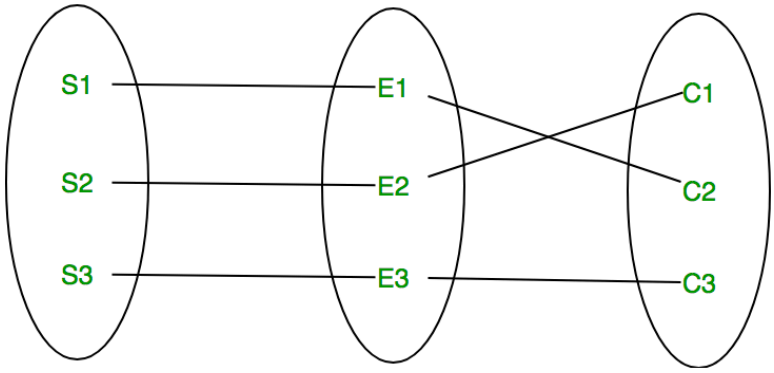
Complete Entity Type Student

Relationship Type and Relationship Set

A Relationship Type represents the association between entity types. For example, 'Enrolled in' is a relationship type that exists between entity type Student and Course. In ER diagram, the relationship type is represented by a diamond and connecting the entities with lines.



A set of relationships of the same type is known as a relationship set. The following relationship set depicts S1 as enrolled in C2, S2 as enrolled in C1, and S3 as registered in C3.

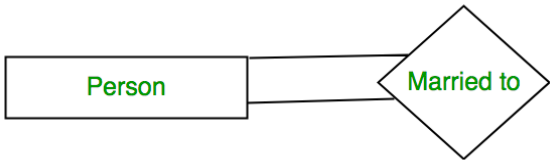


Degree of a Relationship Set

The number of different entity sets participating in a relationship set is called the **degree of a relationship set.**

Unary Relation

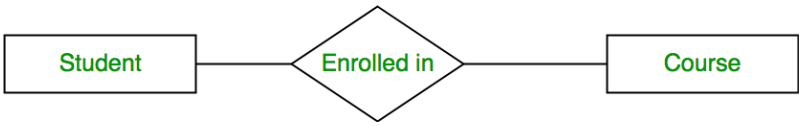
1. When *ONE* entity set is participating in a relation, i.e., ***it is related to itself***.



2. These are also called Recursive Relationships in ER model.
3. Generally, these are used to denote hierarchies or networks.

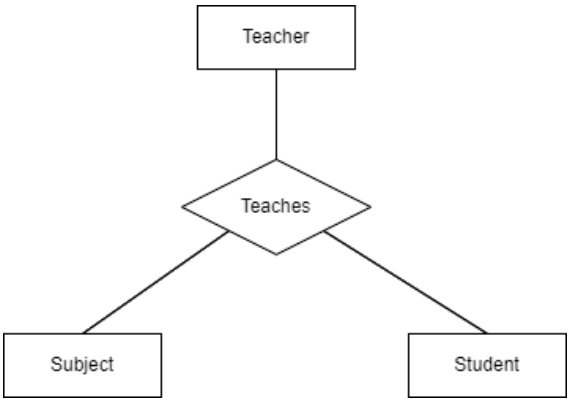
Binary Relation

When *ONE* entity is related to *ANOTHER* entity in a relation.



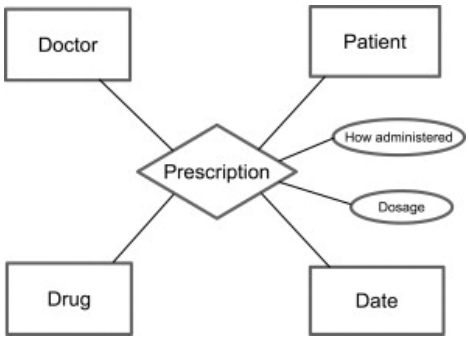
Ternary Relation

When *ONE* entity is related to *THREE* entities.



N-ary Relation

When *N* entities are participating in a relation.



Cardinality

Cardinality simply represents ***"How many times can one entity take part in a relationship"***.

1. **One to One (1 : 1)**
    a. Each entity takes part in a relationship only once.
    b. Eg, each person has one Aadhar only.
2. **One to Many (1 : N)**
    a. One entity is related to many entities.
    b. Eg, a company have many employees, but single employee doesn't have many companies.
3. **Many to One (N : 1)**
    a. Many entities are related to one.
    b. Eg, each book in a university belongs to a single library, but vice versa is not true.
4. **Many to Many (N : M)**
    a. Many entities are related to many other entities.
    b. Eg, student can enroll in many courses and a single course can be enrolled by many students.

Participation Constraint

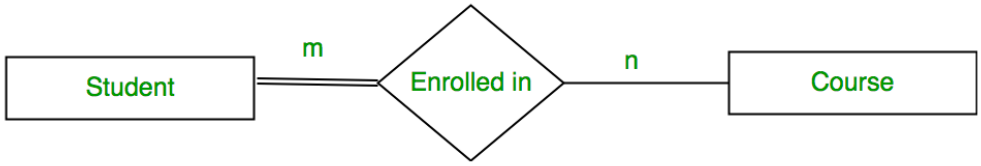
This constraint is applied to entities participating in a relationship set.

Total Participation

1. Every entity of one set has to be related in a relation.
2. Eg, every student must be enrolled in a course. No student can roam having no course enrolled in.

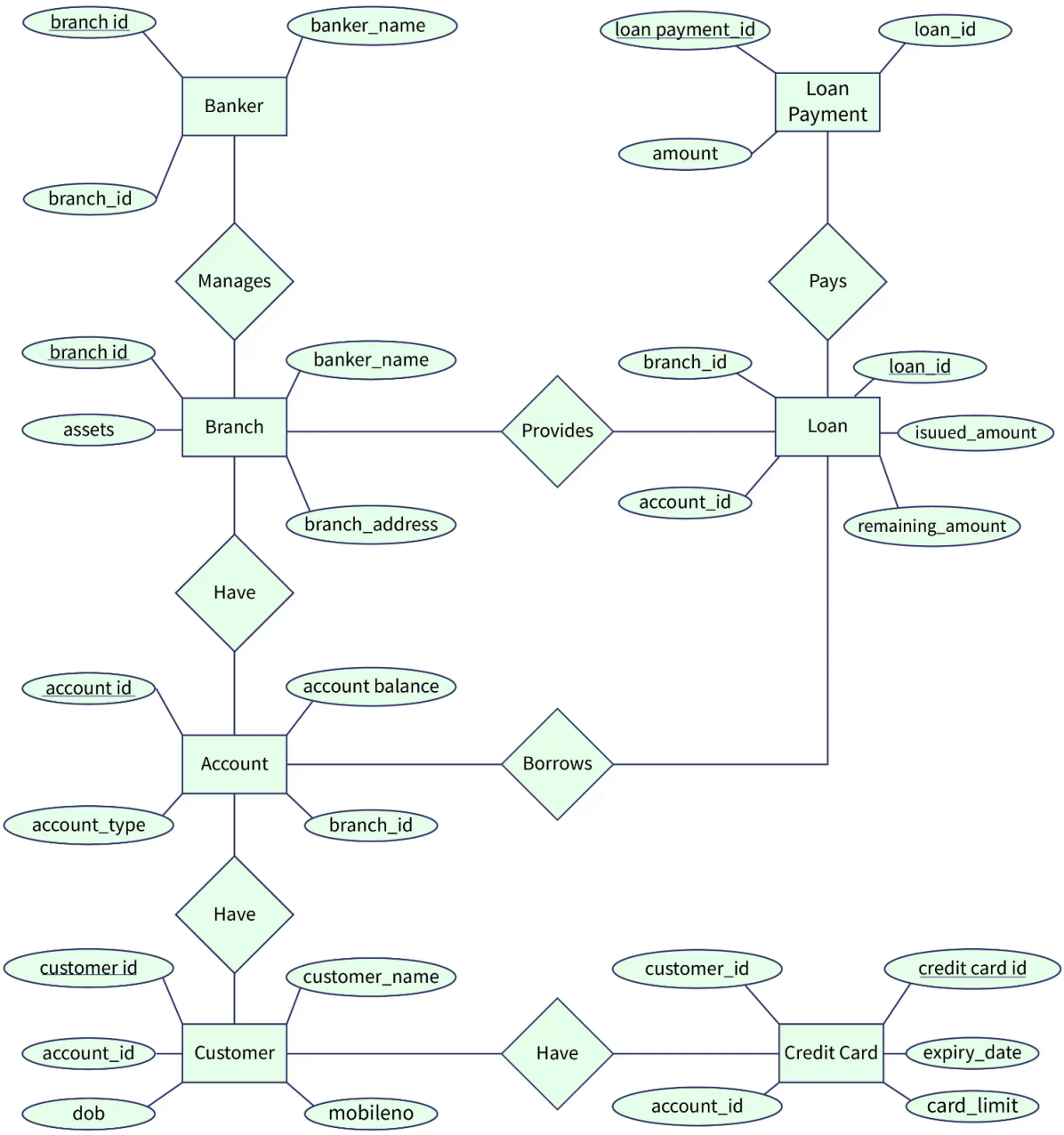
Partial Participation

1. Here every entity doesn't have to participate in a relation.
2. Eg, a course can exist in a university without being enrolled by any student.



Steps to Draw ER Diagram?

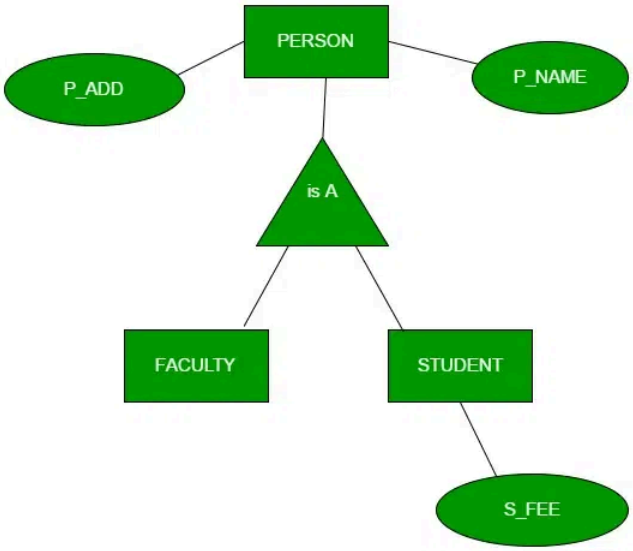
1. The very first step is Identifying all the Entities, and place them in a Rectangle, and labeling them accordingly.
2. The next step is to identify the relationship between them and place them accordingly using the Diamond, and make sure that, Relationships are not connected to each other.
3. Attach attributes to the entities properly.
4. Remove redundant entities and relationships.



Banks' account ER diagram.

Generalization in ER Models

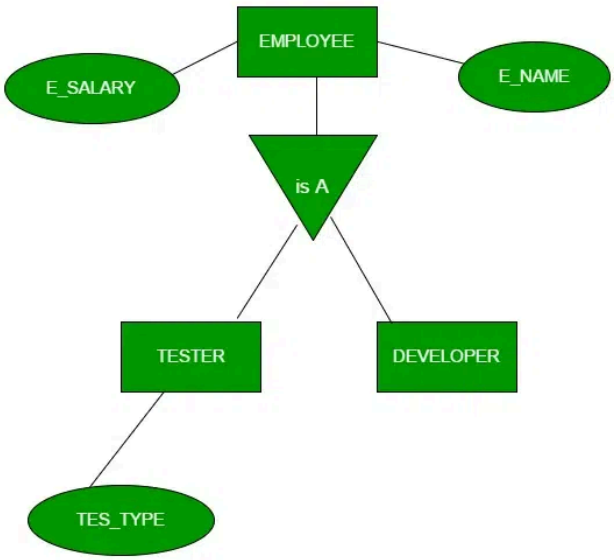
Sometimes ER diagrams turn complex because of so many entities and their attributes, and it becomes difficult to study them as well. To simplify the ER diagram, we use **Generalization** method to separate out common attributes of two or more entities and combine them into one. In generalization, we use ***is-a*** relationship and this approach is a ***bottom-up approach***.



For eg, in a university database we have separate entities for STUDENT and PROFESSOR but can have similar attributes like, name, id, address, contact, email, etc. Since, these attributes are basically redundant when creating an ER diagram, we can simplify them into one super entity called PERSON.

Specialization in ER Models

Specialization is just opposite of generalization. Here a super entity is sub-divided into lower-level entities so as to simplify their roles. For eg, the entity EMPLOYEE can be sub-divided into TESTER and DEVELOPER in a corporation just because they have different roles and responsibilities.

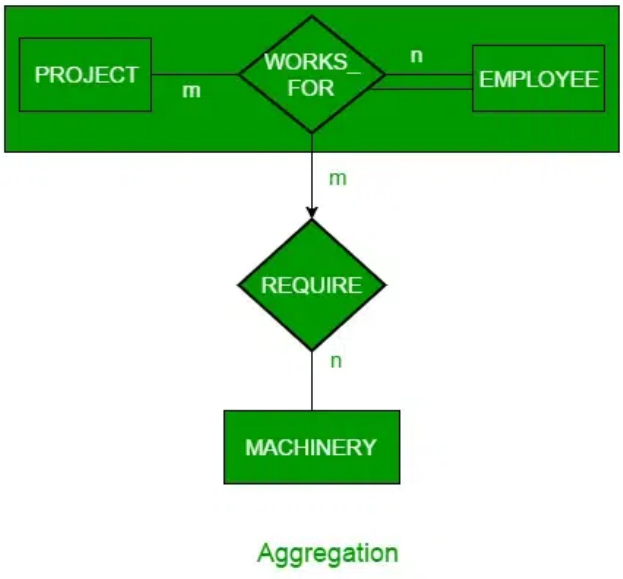


Even if these are divided they still have common attributes like employee_id and salary.
Specialization is a ***top-down approach***.
In specialization too, we use ***is-a*** relationship.

Aggregation in ER Models

Sometimes in an ER diagram it might occur that we need to specify a relationship between an entity and a relationship, which is specifically done using aggregation. We combine the entity and relation into a higher-level entity and specifies the relation between another relation and that higher entity.

For eg, an EMPLOYEE works_for a PROJECT. Here employee and project are entities and works_for is a relation between them. Now, it's possible that the EMPLOYEE working for a PROJECT might require some MACHINERY. And specifically he/she is working for some PROJECT, project also plays an important role in deciding MACHINERY.
So, we aggregate EMPLOYEE and PROJECT into a higher-level entity and build a relation with MACHINERY using a REQUIRE relationship.



Aggregation

Generalization, Specialization and Aggregation might seem difficult to understand here, but these can easily be implemented in SQL.

Introduction to Relational Models and Operators

It was proposed by Dr. E.F. Codd. It uses the concept of relations to represent each and every file. Relations are Two-Dimensional Tables. It is easy to implement and easy to simplification in the operations to manipulate the data. This is the most popular data model. It is simple to implement. It uses the primary key and secondary key to connect any two files.

Eg, Relational Model can be represented as shown below,

```
STUDENT (StudNo, Sname, Special)
ENROLLMENT (StudNo, Subcode, marks)
SUBJECT (Subcode, Subname, Maxmarks, Faccode)
FACULTY (Faccode, Fname, Dept)
```

Codd made twelve-rules for some data to be represented in form of relational models, but there hasn't been any type of data which have followed more than eight to nine of these rules. You don't need to remember these rules. They are just here to understand what he said about data.

Codd's Twelve Rules

1. **Information Rule:**
    a. Definition: All information in a relational database must be represented explicitly at the logical level and in the same way as values in tables (relations).
    b. Implication: Data should be stored in tables, with data items represented as values in rows and columns.
2. **Guaranteed Access Rule:**
    a. Definition: Every data element (atomic value) must be logically accessible by using a combination of table name, primary key, and column name.
    b. Implication: Data should be easily retrievable through simple queries without needing access to the underlying physical storage.
3. **Systematic Treatment of Null Values:**
    a. Definition: Null values (representing missing or inapplicable information) must be uniformly treated and distinguished from other values like zero or an empty string.
    b. Implication: The system must handle null values consistently and provide mechanisms to query and manage them.
4. **Dynamic On-Line Catalog Based on the Relational Model:**
    a. Definition: The database's catalog (metadata) should be stored in the same relational format as user data and accessible using the same relational language.
    b. Implication: The system should use its own relational model to manage metadata, and this metadata should be queryable.

5. **Comprehensive Data Sublanguage Rule:**
    a. Definition: The system must support a comprehensive language for data definition, manipulation, and transaction management, with a single, uniform interface.
    b. Implication: Users should be able to perform all database operations using a single language that supports both queries and data management.
6. **View Updating Rule:**
    a. Definition: Any view (virtual table) that is theoretically updateable must also be updateable through the system, allowing modifications to the underlying base tables.
    b. Implication: Views should support operations like insertions, updates, and deletions if the underlying data supports it.
7. **High-Level Insert, Update, and Delete:**
    a. Definition: The system must support set-based operations for inserting, updating, and deleting data, allowing users to handle multiple rows at a time.
    b. Implication: The database should allow bulk operations and not require individual row manipulation for every change.
8. **Physical Data Independence:**
    a. Definition: Changes to the physical storage of data should not affect the way data is accessed or queried by users.
    b. Implication: Users and applications should not need to know or adapt to changes in how data is physically stored.
9. **Logical Data Independence:**
    a. Definition: Changes to the logical schema (structure of the database) should not affect the applications or users' ability to access data.
    b. Implication: Alterations to tables, such as adding or removing columns, should not impact existing applications that use the data.
10. **Integrity Independence:**
    a. Definition: Integrity constraints (rules that ensure data accuracy) should be stored and managed separately from application programs.
    b. Implication: Constraints should be enforced by the DBMS itself rather than being embedded in application code.
11. **Distribution Independence:**
    a. Definition: The system should provide a unified view of the data regardless of its physical distribution across multiple locations.
    b. Implication: Users should interact with the data as if it were stored in a single location, even if it is distributed across multiple sites.
12. **Non-Subversion Rule:**
    a. Definition: If the system provides a low-level access method (like direct file access), it should not bypass or subvert the integrity rules of the relational model.
    b. Implication: Even when using low-level operations, the system must maintain the integrity constraints and relational principles.

Types of Keys

Keys are a basic requirement of any entity and relation in a relational model. They are used to uniquely identify among different tuples or records.

Different Types of Database Keys

1. Candidate Key
2. Primary Key
3. Super Key
4. Alternate Key
5. Foreign Key
6. Composite Key

Candidate Keys

A candidate key is a minimal set of attributes (columns) in a table that can uniquely identify each record (row) in that table. In other words, no subset of a candidate key can uniquely identify records; it is the smallest combination of attributes that can do so.

Properties:

1. Candidate key must contain unique values.
2. Candidate key is a minimal set of attributes.
3. It must not contain null values.
4. It is irreducible, i.e., no proper subset of candidate key can uniquely identify a record.

Primary Key

Among the candidate keys, one is chosen as the primary key. The primary key is the key that will be used as the main unique identifier for records in the table. The choice of primary key is often based on practical considerations, such as simplicity, stability, and ease of use.

Properties, same as candidate keys.

Super Keys

A superkey is a set of one or more attributes (columns) in a table that can uniquely identify each record (row) in that table. A superkey may contain extra attributes beyond what is strictly necessary to achieve uniqueness.

Properties,

1. Uniquely identifies each record.
2. It may contain additional attributes beyond what's necessary.

3. Every candidate key is a super key but vice versa is not true.
4. It may contain null values but only in those attributes which do not contribute to unique identification.

Alternate Keys

An alternate key is any candidate key that is not chosen as the primary key. It is an alternative key that can uniquely identify records in a table but is not selected as the main key for the table.
Properties,

1. Uniquely identifies each record.
2. Not selected as primary key.
3. Key is minimal, as it is a candidate key.
4. It can server it's purpose as primary key.
5. Doesn't contain null values.

Foreign Keys

A foreign key is a column or a set of columns in a table that is used to reference the primary key of another table. This relationship establishes a linkage between the data in the two tables, ensuring that the data in the foreign key column corresponds to valid entries in the referenced primary key column.
Properties,

1. Referential Integrity (more later).
2. It may contain non unique values.
3. It allows null values as well.
4. Cascading actions (more later).

Composite Key

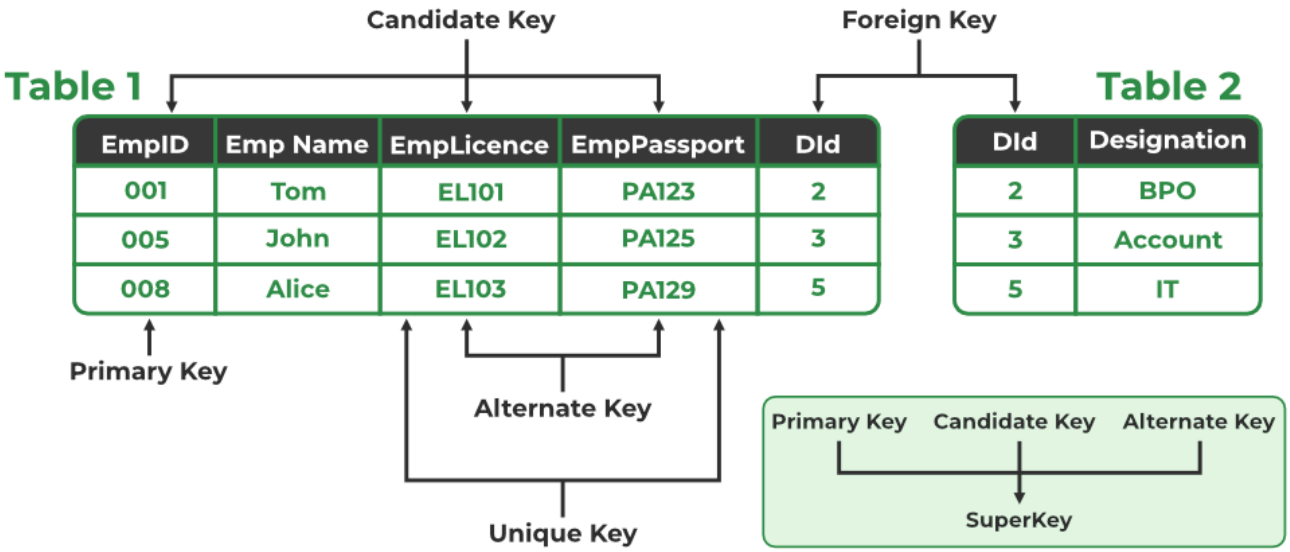
A composite key is a type of primary key that consists of two or more attributes (columns) in a table, combined together to uniquely identify a record. Unlike a single attribute primary key, which uses just one column for uniqueness, a composite key uses a combination of multiple columns.
Properties, same as that of primary key. And additionally, all the columns involved in a composite key cannot be null.

Artificial Keys

An artificial key (or **surrogate key**) is a unique identifier that is created and assigned to each record in a table, often using a sequential number or an arbitrary value. It does not have any business meaning or relevance outside of its role as a unique identifier.

Properties, same as that of primary key.

These are generated by the database when determining a PK is difficult or PK is not declared by the developer.



Integrity Constraints

Integrity constraints are the set of predefined rules that are used to maintain the quality of information. Integrity constraints ensure that the data insertion, data updating, data deleting and other processes have to be performed in such a way that the data integrity is not affected. They act as guidelines ensuring that data in the database remain accurate and consistent. So, integrity constraints are used to protect databases. The various types of integrity constraints are

Types of Integrity Constraints

1. Domain Constraints
2. Entity integrity Constraints
3. Key Constraints
    a. Not-Null
    b. Unique
    c. Default
    d. Check
    e. PK
    f. FK

Domain Constraints

These constraints define the type and range of values an attribute can take. For eg, age should be an integer, generally between 0 to 110. Someone should not enter *'a'* as their age.
Domain Constraints are implemented in dbms using type of data (which also accompanies the range of data). If there's any voilation while inserting or updating, dbms will throw an error.

Entity Integrity Constaint

An entity integrity constraint is a rule in a relational database that ensures that each record in a table is unique and has a unique value to identify it. This constraint is usually enforced by primary keys and UNIQUE constraints.
Every relation should have a PK.
And that PK cannot be NULL.

Key Constraints

Not-Null Constraint

These type of constraints does not allow any cell to hold null value. It is implemented generally in dbms using NOT NULL constraint.

Unique Constaint

Unique constraint does not allow two tuples to have same values, i.e., values in a column should always be different.

Default

It is used to set default values to column if no value is available.

Check

A check constraint ensures that all values in a column or a set of columns meet a specific condition or criteria.

Primary Key

A primary key constraint uniquely identifies each record in a table. It ensures that the value in the primary key column(s) is unique for each record and that no null values are allowed.
PK cannot be NULL and it's unique for each tuple.

Foreign Key

A foreign key constraint ensures that the value in one table (the child table) matches a value in another table (the parent table). It maintains referential integrity between the two tables.
It should follow,

1. **Referential Integrity**
    a. Referential integrity ensures that relationships between tables are maintained consistently. It guarantees that a foreign key in one table matches an existing primary key in another table, thereby preserving the accuracy of links between related data.
2. **Cascading Actions**
    a. Cascading actions are operations defined in the context of foreign key constraints that specify how changes in the parent table should propagate to the child table. They help automate updates and deletions across related records, ensuring data consistency.
    b. ON DELETE/UPDATE CASCADE

    i. When a row is deleted or updated in parent table, then delete or update the rows in child table as well.

c. ON DELETE/UPDATE SET NULL

    i. Set null in child table cells.

d. ON DELETE/UPDATE SET DEFAULT

    i. Set default value if defined.

e. ON DELETE/UPDATE RESTRICT

    i. Prevents the deletion or updation of parent records if there are dependent child records.

3. A foreign key may contain null values.

Anomalies in Relational Model

Anomalies in the relational model refer to inconsistencies or errors that can arise when working with relational databases, specifically in the context of data insertion, deletion, and modification.
These anomalies can be categorized into three types:

- Insertion Anomalies
- Deletion Anomalies
- Update Anomalies.

Database anomalies are the faults in the database caused due to poor management of storing everything in the flat database. It can be removed with the process of Normalization, which generally splits the database which results in reducing the anomalies in the database.

Insertion Anomaly

Imagine a university database with a single table StudentCourses:

StudentID	*StudentName*	*CourseID*	*CourseName*	*Instructor*
1	Alice Smith	CS101	Intro to CS	Dr. Brown
2	Bob Jones	CS101	Intro to CS	Dr. Brown
3	Alice Smith	MATH101	Calculus I	Dr. Green

StudentID	*StudentName*	*CourseID*	*CourseName*	*Instructor*
NULL	NULL	CS102	Data Structures	Dr. White

Suppose a new course, CS102 - Data Structures, is introduced, but no students have enrolled yet. In the existing table structure, you would need to insert a row with a placeholder student for the new course:
Another case of insertion anomaly is that,
In the child table where some FK references a PK in parent table, we cannot add another tuple into the child table as it might be possible that the value of FK we might be adding is not present in PK of parent table.

Updation Anomaly and Deletion Anomaly

An updation/deletion anomaly occurs when changes to data require updates/deletes to multiple rows in a table and also in the referencing table. This can lead to inconsistencies and loss of data if all instances of the data are not updated correctly.
Eg, If a tuple is deleted or updated from referenced relation and the referenced attribute value is used by referencing attribute in referencing relation, it will not allow deleting the tuple from referenced relation.
To avoid these we specify cascading actions.
In a Nutshell, we can say that,
Insertion Anomalies: These anomalies occur when it is not possible to insert data into a database because the required fields are missing or because the data is incomplete. For example, if a database requires that every record has a primary key, but no value is provided for a particular record, it cannot be inserted into the database.

Deletion anomalies: These anomalies occur when deleting a record from a database and can result in the unintentional loss of data. For example, if a database contains information about customers and orders, deleting a customer record may also delete all the orders associated with that customer.

Update anomalies: These anomalies occur when modifying data in a database and can result in inconsistencies or errors. For example, if a database contains information about employees and their salaries, updating an employee's salary in one record but not in all related records could lead to incorrect calculations and reporting.

Removal of anomalies is done by incorporating **Normalization**.

How to create database?

1. Requirement gathering (most important step, people forget to tell). You try to understand the scope and purpose of your database.
2. Remove redundant data through Normalization.
3. Select appropriate datatypes.
4. Establish relationship between tables.
5. Specify constraints to ensure data integrity.
6. Optimize after building.

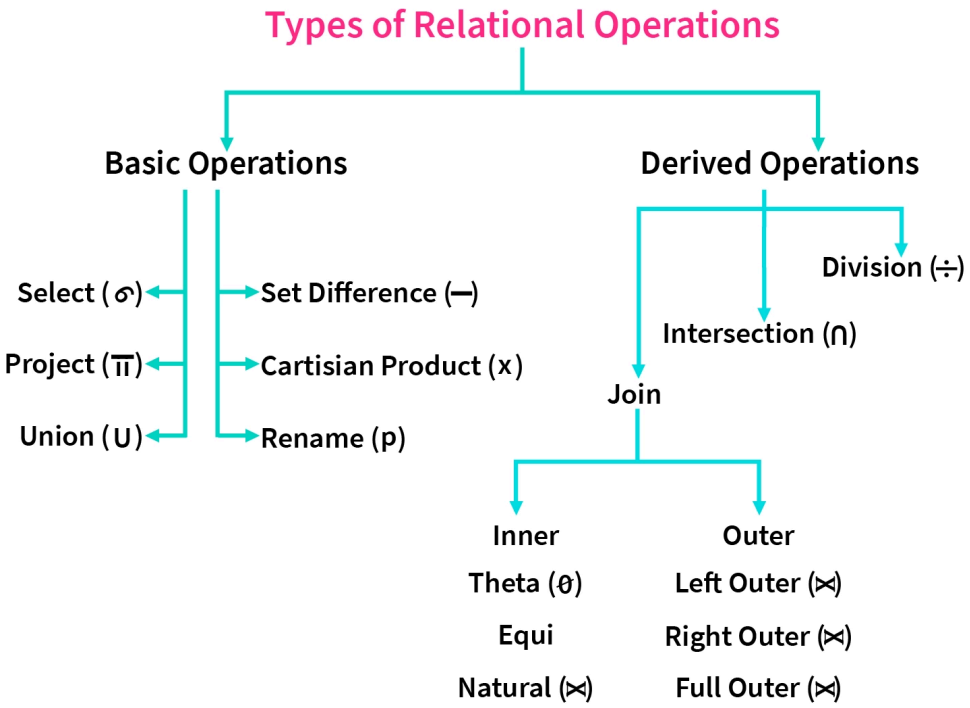
Introduction to Relational Algebra in DBMS

Relational Algebra is a procedural query language. Relational algebra mainly provides a theoretical foundation for relational databases and SQL. The main purpose of using Relational Algebra is to define operators that transform one or more input relations into an output relation. Given that these operators accept relations as input and produce relations as output, they can be combined and used to express potentially complex queries that transform potentially many input relations (whose data are stored in the database) into a single output relation (the query results). As it is pure mathematics, there is no use of English Keywords in Relational Algebra and operators are represented using symbols.

Fundamental Operators

These are the basic/fundamental operators used in Relational Algebra.

1. Selection(σ)
2. Projection(π)
3. Union(U)
4. Set Difference(-)
5. Rename(ρ)
6. Cartesian Product(X)



Selection(σ)

1. It is used to select required tuples of the relations.
2. The selection operator only selects the required tuples but does not display them.
3. For display, the data projection operator is used.
4. σ (condition) selects those tuples which fall under the condition specified.
5. σ (department='IT') selects those tuples with department *'IT'*.

Eg,

Employee ID	***Name***	***Age***	***Department***
1	Alice Smith	30	HR
2	Bob Jones	40	IT
3	Carol Davis	25	IT
4	David Brown	35	Finance

Output,

Employee ID	***Name***	***Age***	***Department***
2	Bob Jones	40	IT
3	Carol Davis	25	IT

Projection(π)

1. It is used to project required column data from a relation.
2. π (name, department) will return a table containing only names and department of employees.

Eg,

Employee ID	*Name*	*Age*	*Department*
1	Alice Smith	30	HR
2	Bob Jones	40	IT
3	Carol Davis	25	IT
4	David Brown	35	Finance

Output,

Name	*Department*
Alice Smith	HR
Bob Jones	IT
Carol Davis	IT
David Brown	Finance

Union(U)

1. Union operation in relational algebra is the same as union operation in set theory.
2. The only constraint in the union of two relations is that both relations must have the same set of Attributes.
3. It removes duplicates if any.
4. FullTimeEmployees U PartTimeEmployees, this will give a common result table.

Full Employees Table

Employee ID	***Name***	***Position***
1	Alice Smith	Manager
2	Bob Jones	Developer

Part Time Employees Table

Employee ID	***Name***	***Position***
3	Carol Davis	Consultant
4	David Brown	Intern

Output Table,

Employee ID	***Name***	***Position***
1	Alice Smith	Manager
2	Bob Jones	Developer
3	Carol Davis	Consultant
4	David Brown	Intern

Set difference (-)

1. Set Difference in relational algebra is the same set difference operation as in set theory.
2. The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.
3. Employees - Managers, this will give Employees who are not listed in Managers.

Students,

Student ID	***Name***	***Grade***
1	Alice	10
2	Bob	11
3	Carol	12

Honors Students,

Student ID	***Name***	***Grade***
1	Alice	10
3	Carol	12

Non-Honors Students Output,

Student ID	***Name***	***Grade***
2	Bob	11

Rename(ρ)

1. Rename is a unary operation used for renaming attributes of a relation.
2. ρ(a/b)R will rename the attribute 'b' of the relation by 'a'.

Cross Product (X)

1. Cross-product between two relations. Let's say A and B, so the cross product between A X B will result in all the attributes of A followed by each attribute of B.
2. Each record of A will pair with every record of B.

Employees Table,

EmployeeID	*Name*
1	Alice Smith
2	Bob Jones

Projects Table,

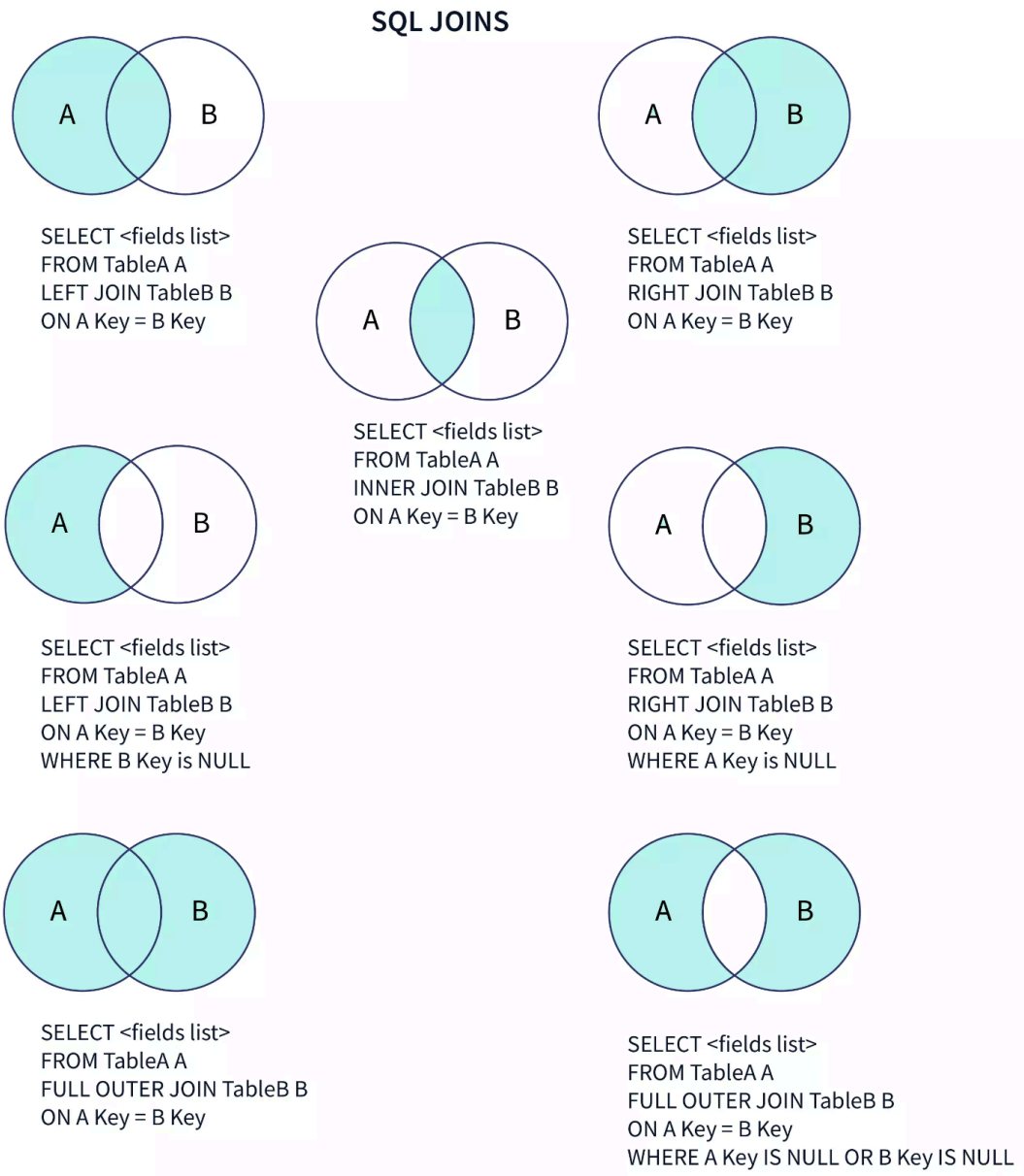
ProjectID	*ProjectName*
101	Project A
102	Project B

Final Output Table,

EmployeeID	*Name*	*ProjectID*	*ProjectName*
1	Alice Smith	101	Project A
1	Alice Smith	102	Project B
2	Bob Jones	101	Project A
2	Bob Jones	102	Project B

Derived Operators

1. Set Intersection (∩)
2. Division (/)
3. Inner Join
4. Left Join
5. Right Join
6. Full Outer Join



Set Intersection (∩)

1. Set Intersection in relational algebra is the same set intersection operation in set theory.
2. The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.
3. (π_EmployeeID, Name(Employees) ∩ π_EmployeeID, Name(ProjectTeam)), this will give employeeID, Name of tuples which are common to both tables.

Employees Table,

Employee ID	***Name***	***Age***	***Department***
1	Alice Smith	30	HR
2	Bob Jones	40	IT
3	Carol Davis	25	IT
4	David Brown	35	Finance

Project Team Table,

EmployeeID	***Name***	***Role***
2	Bob Jones	Developer
3	Carol Davis	Analyst
5	Eva Green	Tester

Final Output Table,

Employee ID	***Name***
2	Bob Jones
3	Carol Davis

Inner Join

An Inner Join returns only the rows that have matching values in both tables. If there's no match, the row is excluded from the result.

Customers Table,

CustomerID	*CustomerName*
1	Alice
2	Bob
3	Carol

Orders Table,

OrderID	*CustomerID*	*Product*
101	1	Laptop
102	2	Smartphone
103	4	Tablet

Final Output Table,

CustomerName	*Product*
Alice	Laptop
Bob	Smartphone

Left Join

A Left Join returns all rows from the left table and the matched rows from the right table. If there is no match, the result is NULL for columns from the right table.

Customers Table,

CustomerID	CustomerName
1	Alice
2	Bob
3	Carol

Orders Table,

OrderID	CustomerID	Product
101	1	Laptop
102	2	Smartphone
103	4	Tablet

Final Output Table,

CustomerName	Product
Alice	Laptop
Bob	Smartphone
Carol	NULL

Right Join

A Right Join returns all rows from the right table and the matched rows from the left table. If there is no match, the result is NULL for columns from the left table.

Customers Table,

CustomerID	*CustomerName*
1	Alice
2	Bob
3	Carol

Orders Table,

OrderID	*CustomerID*	*Product*
101	1	Laptop
102	2	Smartphone
103	4	Tablet

Final Output Table,

CustomerName	*Product*
Alice	Laptop
Bob	Smartphone
NULL	Tablet

Full Outer Join

A Full Outer Join returns all rows when there is a match in either left or right table. If there is no match, the result will contain NULL for columns from the table without a match.
Basically it's a union of left join and right join.

Customers Table,

CustomerID	*CustomerName*
1	Alice
2	Bob
3	Carol

Orders Table,

OrderID	*CustomerID*	*Product*
101	1	Laptop
102	2	Smartphone
103	4	Tablet

Final Output Table,

CustomerName	*Product*
Alice	Laptop
Bob	Smartphone
Carol	NULL
NULL	Tablet

Functional Dependency

A functional dependency (FD) between two sets of attributes in a relational database is a constraint that describes a relationship between attributes. Specifically, a functional dependency X -> Y means that if two tuples (rows) agree on the values of attributes in set X, then they must also agree on the values of attributes in set Y. In other words, X functionally determines Y.

Notation

X and Y are sets of attributes.
X -> Y indicates that Y is functionally dependent on X.

Properties of FD

1. **Reflexivity**
    a. If Y ⊆ X (Y is a subset of X) then, X -> Y is always true.
2. **Augmentation**
    a. If X -> Y then, XZ -> YZ.
3. **Transitive**
    a. If X -> Y and Y -> Z then, X -> Z.
4. **Union**
    a. If X -> Y and X -> Z then, X -> YZ.
5. **Decomposition**
    a. If X -> YZ then, X -> Y and X -> Z.
6. **Pseudo-transitivity**
    a. If X -> Y and WY -> Z then, WX -> Z.
7. **Composition**
    a. If X -> Y and W -> Z then, WX -> ZY.

Attribute Closure

Attribute Closure method is the only way to determine Candidate Keys, Prime Attributes and Non-Prime Attributes from a relation. We utilize the properties of FDs to find the candidate keys.

R (A, B, C, D)
FD = {A -> B, B -> C, C -> D}
Now,

A^+ = ABCD
B^+ = BCD
C^+ = CD
D^+ = D

Since, A can determine all attributes of relation R. A is the Candidate Key. And A is the prime attribute others are non-prime attributes.

CK = {A}

R (A, B, C, D)
FD = {A -> B, B -> C, C -> D, D -> A}

Now,

A^+ = ABCD
B^+ = BCDA
C^+ = CDAB
D^+ = DABC

Since, each member of relation R can determine all, So all of them are candidate keys and in fact they all are prime attributes.

CK = {A, B, C, D}

R (A, B, C, D, E)
FD = {A -> B, BC -> D, E -> C, D -> A}

Now, to find which attribute is going to be a part of candidate key, check on the RHS of each FD.
Select the attribute which cannot be derived from any one, in this case it is E.
This means that E is going to be a part of every Candidate Key.

So,

AE^+ = AEBCD

As AE is part of CK, check which attributes derive A and E.
Here, D can derive A.

So,

DE^+ = DEABC

As DE is also a part of CK, check which attributes derive D and E.
Here, D can be derived using BC.

So,

BCE^+ = BCEDA

If BCE is deriving all attributes, check for minimal set,
Hence,

BE^+ = BECDA

CE^+ = CE (Not a CK)

So, BCE can be a SK but not a CK.

Similarly, check for each attribute which can derive all attributes.
Finally,

CK = {AE, DE, BE}

Prime Attributes = {A, B, D, E}

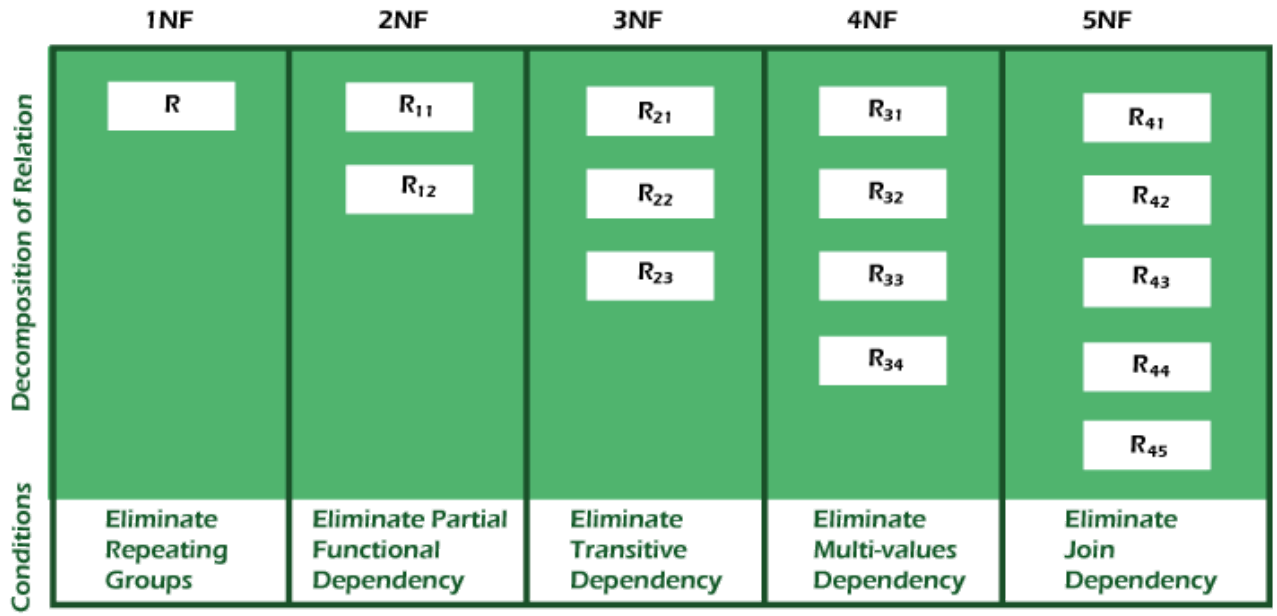
Non-Prime Attributes = {C}

You don't need to study **Armstrong's axioms** as,
Armstrong's axioms state following properties,

1. Reflexivity
2. Augmentation and
3. Transitivity

Normalization

It is a method to reduce redundancy from a database (basically remove or reduce duplicacy as much as possible).
If we stored all the data in a single table, for example in a university database, data of students, course and faculties is stored in a single table. This would lead to redundant data. Even if the student data is kept singular, course and faculty data would be redundant as a course is taken by multiple students and a single faculty teaches hundreds of students.
If not resolved then this would lead to insertion, updation and deletion anomalies as we discussed earlier.
There are two types of redundancies, **row wise** and **column wise**.
Row wise redundancies are solved using **Primary Key**.
For **column wise** redundancies we need **Normal Forms** to reduce.



First Normal Form

No attribute can contain multiple values.

Table not in 1NF,

OrderID	*CustomerName*	*Products*
1001	Alice	Laptop, Mouse, Keyboard
1002	Bob	Smartphone
1003	Charlie	Tablet, Headphones
1004	David	Laptop, Printer
1005	Eve	Smartphone, Headphones
1006	Frank	Laptop, Printer, Scanner
1007	Grace	Tablet

Table in 1 NF,

OrderID	*CustomerName*	*Product*
1001	Alice	Laptop
1001	Alice	Mouse
1001	Alice	Keyboard
1002	Bob	Smartphone
1003	Charlie	Tablet
1003	Charlie	Headphones
1004	David	Laptop
1004	David	Printer
1005	Eve	Smartphone
1005	Eve	Headphones
1006	Frank	Laptop
1006	Frank	Printer
1006	Frank	Scanner
1007	Grace	Tablet

Second Normal Form

1. Table should be in 1 NF.
2. And there should not be any partial dependency, i.e., no proper subset of candidate key should determine any of non prime attributes.

Partial dependency is defined as,
say AB is a CK in a relation R (A, B, C, D, E).
Hence, {A, B} are prime attributes then this gives {C, D, E} as non-prime attributes.
Now, in the FDs, if either A or B solely is determining some non-prime attributes, say A -> C. This is called as Partial Dependency. And this should not be present in 2 NF.

Eg,
R (A, B, C, D, E, F)
FD = {C -> F, E -> A, EC -> D, A -> B}
When we solve it, we find
CK = {EC}
Prime A = {E, C}
Non-P A = {A, B, D, F}

But here, we can see that,
C -> F (partial dependency)
E -> A (partial dependency)
EC -> D (this is NOT a partial dependency)

Hence, this is **not in 2 NF form**.

To convert it to 2 NF, we decompose the original table such that,
Since, C -> F (we make it a new relation)
E -> A and A -> B, so E -> B and this is also true that E -> AB (this becomes another relation)
Also, EC -> D (this becomes third relation).

R1 (C, F)
R2 (E, A, B)
R3 (E, C, D)

To check for a partial dependency,
LHS mus be proper subset of CK and RHS must be a non-prime attribute.

Third Normal Form

1. Table should be in 2 NF.
2. There shouldn't be any transitive dependency.

Transitive dependency,
When an attribute is indirectly dependent on the primary key through another non-key attribute, it's called a transitive dependency. In simple words, LHS should be CK if RHS is non-prime, i.e.,
Non-prime -> Non-prime is NOT ALLOWED.

Eg,
In relation **R (A, B, C, D, E)**
FD = {A -> B, B -> C, CD -> E}

CK = {AD}
PA = {A, D}
NPA = {B, C, E}

Not in 2NF, as A -> B (partial dependency)
So, converting to 2NF,

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

But if you look closely,
AD^+ = ADBCE (AD is dependent on CD to determine E)
Hence, this is a **transitive dependency (CD -> E)**, CD ain't Candidate Key.

To solve this we decompose relation R.
R1 (<u>A</u>, <u>D</u>, B, C)
R2 (<u>C</u>, <u>D</u>, E)

Boyce Codd Normal Form (BCNF)

1. Table should be in 3 NF.
2. If X -> Y, then X should be super key.

Eg,
Let's say we have relation,
R (A, B, C, D)
FD = {AB -> CD, D -> A}

We got,
CK = {AB, BD}
PA = {A, B, D}
NPA = {C}

There's no transitive dependency as in AB in first is CK and in D -> A, A is prime attribute.
So, this is in 3 NF.

But, in **D -> A**, **D is not Super Key**, so this is not in BCNF.

We decompose it to make in BCNF form such that,
R1 (<u>D</u>, C)
R2 (<u>A</u>, <u>B</u>, D)

Lossless Decomposition

Lossless join decomposition is a decomposition of a relation R into relations R1, and R2 such that if we perform a natural join of relation R1 and R2, it will return the original relation R. This is effective in removing redundancy from databases while preserving the original data.

1. The Union of Attributes of R1 and R2 must be equal to the attribute of R. Each attribute of R must be either in R1 or in R2.
2. The intersection of Attributes of R1 and R2 must not be NULL.
3. The common attribute must be a key for at least one relation (R1 or R2).

Dependency Preserving Decomposition

If we decompose a relation R into relations R1 and R2, All dependencies of R either must be a part of R1 or R2 or must be derivable from a combination of functional dependency of R1 and R2.
For Example in a relation,
R (A, B, C, D, E)
FD = {A -> BC, CD -> E}
is decomposed into **R1(ABC)** and **R2(CDE)**
which is dependency preserving because A -> BC is a part of R1(ABC).

Denormalization

Denormalization is a database optimization technique in which we add redundant data to one or more tables. This can help us avoid costly joins in a relational database. Note that denormalization does not mean ‘reversing normalization’ or ‘not to normalize’. It is an optimization technique that is applied after normalization.
Basically, The process of taking a normalized schema and making it non-normalized is called denormalization, and designers use it to tune the performance of systems to support time-critical operations.
In a traditional normalized database, we store data in separate logical tables and attempt to minimize redundant data. We may strive to have only one copy of each piece of data in a database.

For example, in a normalized database, we might have a Courses table and a Teachers table. Each entry in Courses would store the teacherID for a Course but not the teacherName. When we need to retrieve a list of all Courses with the Teacher’s name, we would do a join between these two tables.

In some ways, this is great; if a teacher changes his or her name, we only have to update the name in one place. The drawback is that if tables are large, we may spend an unnecessarily long time doing joins on tables.
Denormalization, then, strikes a different compromise. Under denormalization, we decide that we're okay with some redundancy and some extra effort to update the database in order to get the efficiency advantages of fewer joins.

Pros of Denormalization

1. Retrieving data is faster since we do fewer joins
2. Queries to retrieve can be simpler(and therefore less likely to have bugs),
3. since we need to look at fewer tables.

Cons of Denormalization

1. Updates and inserts are more expensive.
2. Denormalization can make update and insert code harder to write.
3. Data may be inconsistent.
4. Data redundancy necessitates more storage.

Instruction Cycle

1. Fetch
2. Decode
3. Execute
4. Store

Transactions

Transactions are set of instructions which perform a logical unit of work. The instructions defined under a transactions are Read and Write instructions.
Eg, for debit transaction from account A,
Read (A): Reads value of A from database and store it to buffer in main memory.
A = A - 1000
Write (A): Write value of A from buffer in main memory to database.

Commit
Changes made to temporary values are made permanent to database.

Rollback
Going back to the previous consistent state is called rolling back.

Properties of Transactions

1. Atomic
2. Consistent
3. Isolated
4. Durable

Atomic

Simplest definition of Atomic is, *"either all or none"*.
It is the duty of DBMS to try and execute all the instructions of a transaction. If at some point it fails, then it should rollback to previous consistent state undoing all the changes.

Consistent

Sum of all logical variables shall remain same before the transaction starts execution and after the transaction completes it's execution.
Eg, A wants to transfer 1000 from his account to B. Initially A has 5000 and B has 4000.
Total sum before transfer is 5000 + 4000 = 9000.
Now, when the transaction is completed, A has 4000 and B now has 5000.
Total sum after transfer is 4000 + 5000 = 9000.

Isolation

Multiple Transactions exist concurrently without knowing existence of each other.

Durability

Changes made to the database should remain persistent. They should not diminish if some system failure occurs.

The most important thing to note here is that, Atomicity, Isolation and Durability are properties of database and if they all are good then Consistency holds true automatically.

Property	**Responsible Person**
Atomicity	Transaction Manager
Consistency	Application Programmer
Isolation	Concurrency Control Manager
Durability	Recovery Manager

Advantages of ACID Properties in DBMS

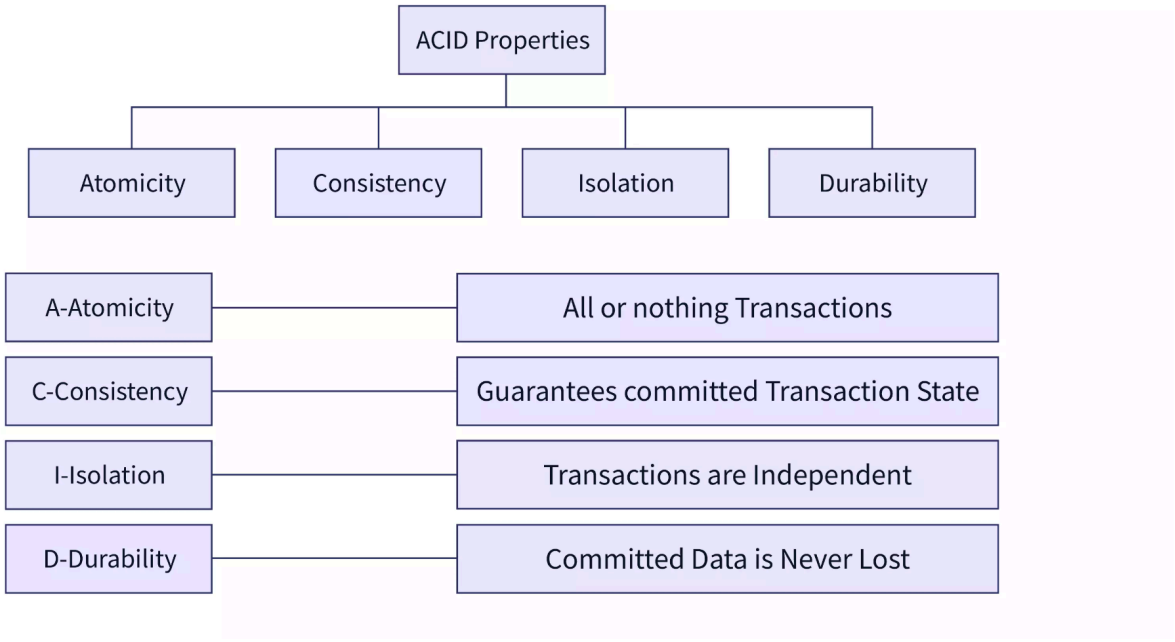
1. **Data Consistency**: ACID properties ensure that the data remains consistent and accurate after any transaction execution.
2. **Data Integrity**: ACID properties maintain the integrity of the data by ensuring that any changes to the database are permanent and cannot be lost.
3. **Concurrency Control**: ACID properties help to manage multiple transactions occurring concurrently by preventing interference between them.
4. **Recovery**: ACID properties ensure that in case of any failure or crash, the system can recover the data up to the point of failure or crash.

Disadvantages of ACID Properties in DBMS

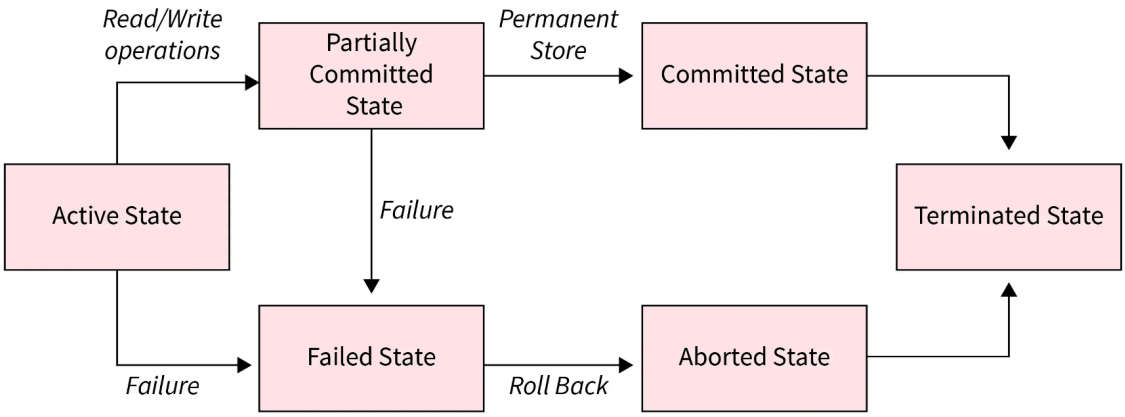
1. **Performance**: The ACID properties can cause a performance overhead in the system, as they require additional processing to ensure data consistency and integrity.
2. **Scalability**: The ACID properties may cause scalability issues in large distributed systems where multiple transactions occur concurrently.
3. **Complexity**: Implementing the ACID properties can increase the complexity of the system and require significant expertise and resources.

Overall, the advantages of ACID properties in DBMS outweigh the disadvantages. They provide a reliable and consistent approach to data management, ensuring data integrity, accuracy, and reliability.

However, in some cases, the overhead of implementing ACID properties can cause performance and scalability issues. Therefore, it’s important to balance the benefits of ACID properties against the specific needs and requirements of the system.



Transaction States



The only operations that concern us in case of DBMS are Read and Write operations.

In a DBMS, we don't actually work on the original database, we create a copy of it in main memory and perform operations on it. If the operations execute successfully, we commit our transactions for making changes in original copy. If somehow there's a failure while executing transaction, database is sent to failed state and rolled back to it's previous consistent state.

When there's a failure, we can opt to try executing the transaction again.

Transactions when committed are permanent changes, you cannot rollback after committing.
In case you want to go back to previous state, you run another transaction called as "Compensating Transaction", which doesn't rollback the database but takes the database to another consistent state similar to previously existing.

Concurrency Control

Concurrency control is a very important concept of DBMS which ensures the simultaneous execution or manipulation of data by several processes or user without resulting in data inconsistency. Concurrency Control deals with interleaved execution of more than one transaction.
Advantages of Concurrency Control are,

1. Decreased Wait Time
2. Decreased Response Time
3. Increased Resource Utilization
4. Increased Efficiency

However, when multiple transactions are executing parallelly then there might be scenarios where the database might get into inconsistent state. And they are,

Concurrency Problems in Transactions

Dirty Read

A transaction reads a value that has been modified by another transaction but not yet committed.

Unrepeatable Read

A transaction reads the same data multiple times and gets different results.

Lost Update

The changes made by one transaction are overwritten by another transaction, resulting in the loss of the first transaction's updates.

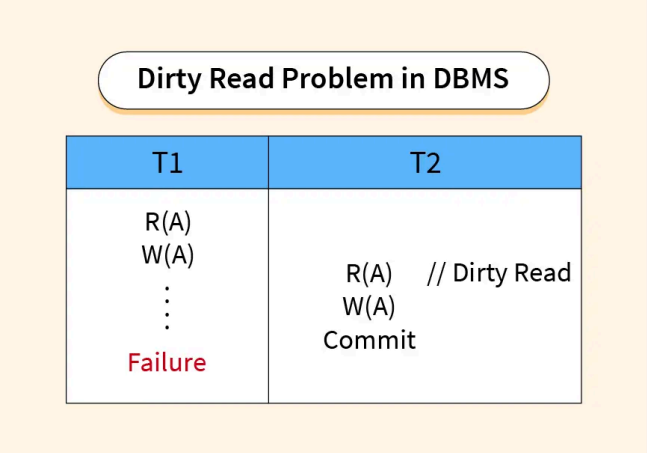
Phantom Read

A transaction reads a set of data multiple times and finds new rows that were inserted by another transaction.

Dirty Read Problem

Let's understand this problem through a scenario.
There were two friends Ram and Shyam. They both had a Mathematics examination and since Shyam didn't studied anything, he decided to copy whatever Ram wrote in his exam sheets.
Ram wrote his answers and Shyam copied everything. Since Shyam already wrote all the answers he submits his exam sheets and leaves the room.
But hey, Ram is not done. He looks at some of his answers and finds they are wrong!
He corrects them and submits his sheets to the invigilator.
After a week, the result was declared and Shyam was hoping to get same marks as Ram. But he got less than average. He was confused. How can anything like this happen, they both wrote the same answers.
Shyam asked Ram and concluded that he should have waited for Ram to submit his sheet first, before submitting his own

You see the problem here?
Problem is not cheating, problem is cheating wrong data. And that's what Dirty Read is all about.
A transaction when reads the modified value of an uncommitted transaction, it is called Dirty Read.
Dirty Read does not happen always but why take the risk.



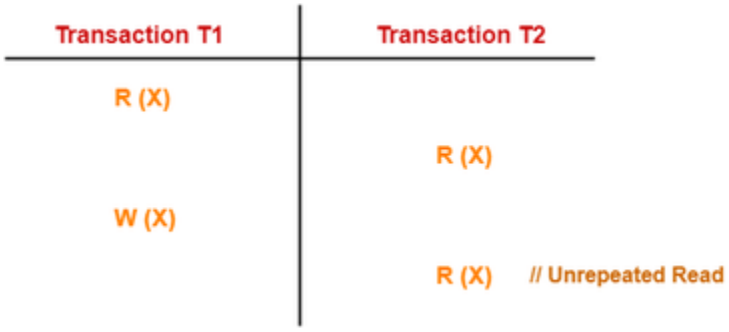
Transaction T2 works on a value which was not committed by T1.
Hence, the solution for this is that if a transaction does a dirty read then it should wait for the first transaction to commit. In that case if any changes occur or any failure occur, T2 won't commit on a false value.

Problem, T2 read uncommitted value of a variable modified by T1.
Solution, T2 should commit after T1 has committed already.

Unrepeatable Read

Let's hear another story of a boy whose parents have gone out. He's all alone in the house feeling the independence that many 12 year olds have. He decides to cook some Maggi for himself. He puts on the pan and add the Maggi in it. He covers the pan and goes outside to lock the door.
When he comes back and looks out of the window, there were high winds blowing outside the house. It was complete dark, branches of the tree nearby banged to the windows.
Aside from all of this, he decides to take his Maggi and go to watch TV. He removes the lid and is shocked!
There was no Maggi inside the pan, it was having Fried Rice in it. He was confused. How did that happen, he was all alone in the house. Fear shivers his spine when he hears someone breathing in the kitchen.
He looks up and there was this spirit with dark feathers, long claws. Ghost said, "leave it there, I like it".

Something like that happens in a unrepeatable read problem.
Every transaction believes that they are isolated and all alone executing in the system.



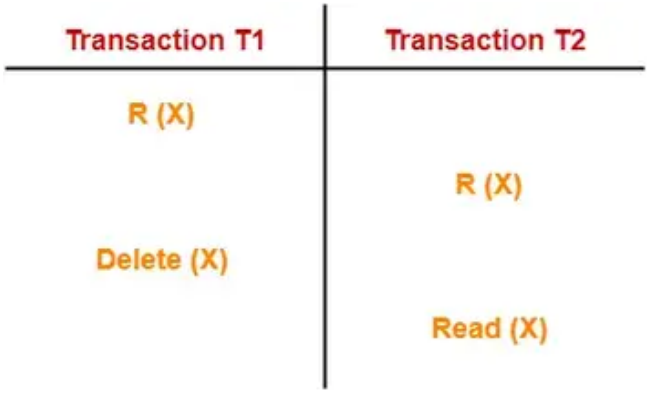
Transaction T2 reads X two times but both the times value is different.
This happened because T1 modified X before T2 read it for second time.

Solution to unrepeatable read is locking mechanism, which is discussed later.

Phantom Read

Let’s continue with the story of the boy. After his parents heard from him they comforted him in every way possible. They told him that nothing like this happens, it was all just something he might have made in his mind.
Three years have passed since that incident and the boy knows it wasn’t something he made up.
Today, his parents have to go meet his aunt and he stays at the house, once again, all alone.
Again he decides to cook Maggi and leaves it in the pan.
He comes back again and is shocked to see that nothing is there in the house. He calls the police because everything in the kitchen is stolen, even the Maggi.

Something like this happens in Phantom Read,
A transaction tries to read a variable that does not exist anymore.



Here Transaction T1 deletes X before T2 can read it once again.
This causes error related to variable doesn’t existing.

Lost Update

Lost update also known as Write-Write conflict happens when a transaction performs a blind write on a variable.
Blind write occurs when a transaction writes a value without reading it first. If it would have read it first, then it would be a Dirty read problem.

Transaction T1	***Transaction T2***
R (A) W (A) Commit	 W (A) Commit

Solution to Lost update or Write-Write problem is achieved by providing more isolation to the transactions.

Schedules

Schedule is an arrangement of logical instructions that are present in a transaction.

1. Serial Schedule
2. Concurrent Schedule also known as Non-serial Schedules

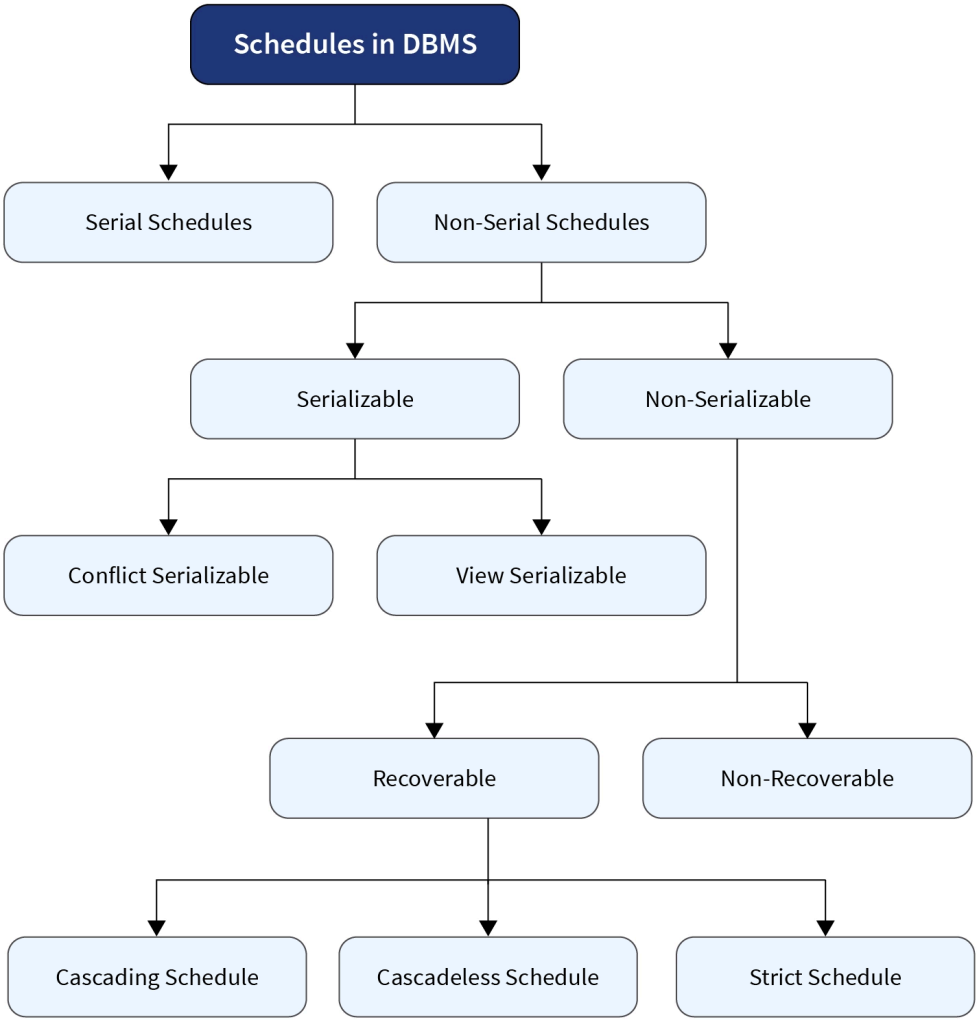
Serial Schedule	**Concurrent Schedules**
In serial scheduling, transactions are executed one after the other.	In concurrent schedules, transactions are executed concurrently.
Serial scheduling consumes more amount of time.	They takes less amount of time as CPU is not idle here.
Here, only one transaction is executed at a time.	Multiple transactions are executed based on number of cores available.
They are always consistent.	These are not consistent, when multiple transactions work on shared memory.
Serial Schedules are less efficient.	Concurrent Schedules are very much efficient.

Since a serial schedule is always consistent, what we try is to prove *"our devised"* concurrent schedule equivalent to a serial one. In that way our concurrent schedule also becomes consistent.

Transaction T1	Transaction T2
R(A)	
W(A)	
R(B)	
W(B)	
commit	
	R(A)
	W(B)
	commit

Transaction (Ta)	Transaction (Tb)
R(X)	
	R(X)
W(X)	
Commit	
	W(X)
	R(X)
	commit

Serial & Non-Serial Schedules


Schedules in DBMS
Serial Schedules
Non-Serial Schedules
Serializable
Non-Serializable
Conflict Serializable
View Serializable
Recoverable
Non-Recoverable
Cascading Schedule
Cascadeless Schedule
Strict Schedule

Conflict Serializability

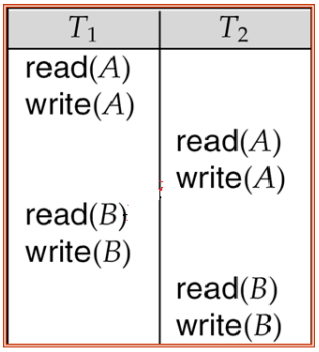
Let's first understand what actually causes the conflict among transactions.

*When there are instructions which belong to different transactions and are working on **same data** and one of them is a **write operation**, this makes the transactions go inconsistent.*
Thus, making the schedule conflicting.

In order to decide which non-serial schedule is consistent, we compare it with the corresponding serial schedule. If they match perfect after swapping those instructions, our non-serial schedule is also consistent.

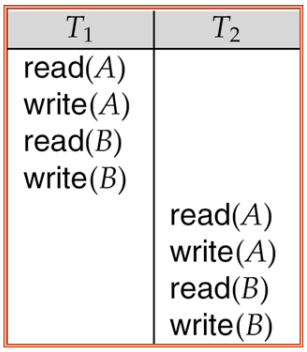
Now, let's see how to swap instructions.

1. They should belong to different transactions. In any case you cannot change logical order of instructions in a transactions.
2. They should belong to same data they are working on.



T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Before swapping



T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

After swapping

Now, we don't actually compare each transaction with the serial schedule because it might become a tedious task to do so if I have many number of transactions.

What we utilize is called as Precedence Graph.

Just so a schedule fails to be conflict serializable does not mean it cannot be a serial schedule. If it fails conflict serializability then we go on and check for View Serializability.

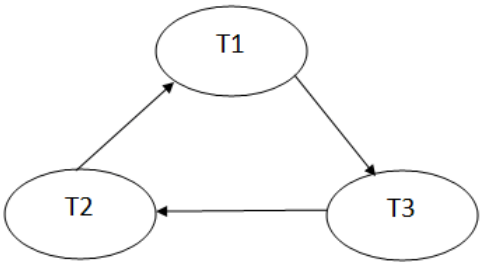
Precedence Graph

It is a graph structure formed for each transaction as we go in a top-to-bottom manner through the logical instructions. This is utilized to check conflict serializability in a schedule.
As soon as we get two transactions working on same variable and one of the instruction is a write operation, we draw it in the graph keeping the transaction above as starting point and transaction below as ending point.
If a cycle exists then the schedule cannot be conflict serializable.

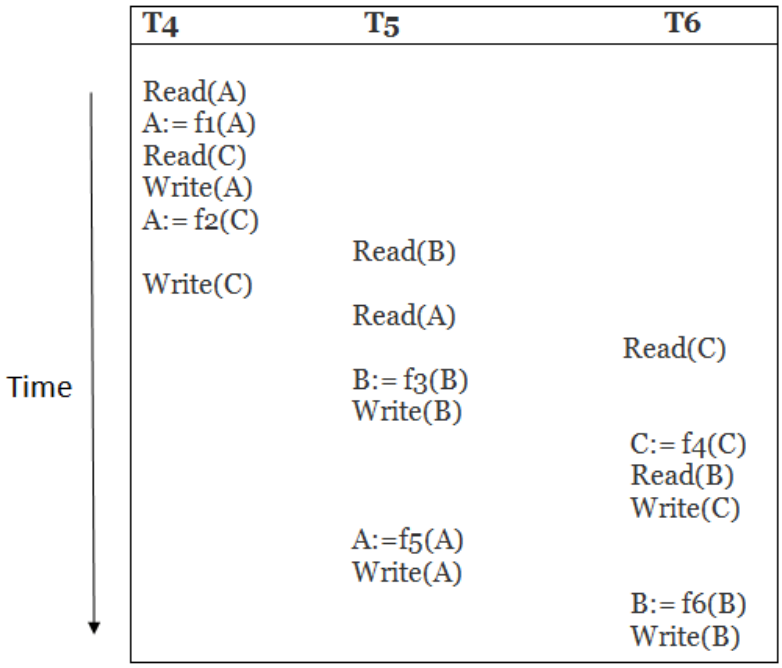
Time

T1	T2	T3
Read(A)		
	Read(B)	
$A := f_1(A)$		
		Read(C)
	$B := f_2(B)$	
	Write(B)	
		$C := f_3(C)$
		Write(C)
Write(A)		
		Read(B)
	Read(A)	
	$A := f_4(A)$	
Read(C)		
	Write(A)	
$C := f_5(C)$		
Write(C)		
		$B := f_6(B)$
		Write(B)

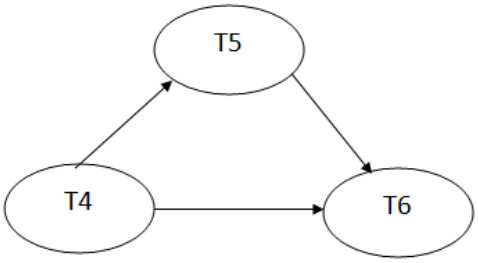
Schedule S1



In here, you can run for yourself and check that this schedule is not serializable. More explaination in second example on next page.



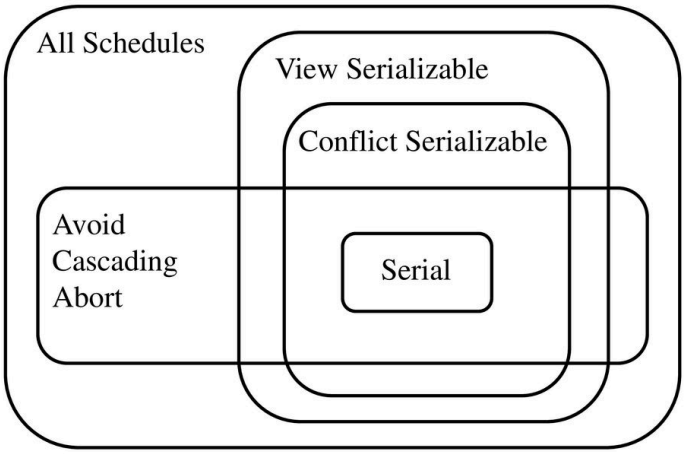
Schedule S2



Explanation

1. **Read(A):** In T4,no subsequent writes to A, so no new edges
2. **Read(C):** In T4, no subsequent writes to C, so no new edges
3. **Write(A):** A is subsequently read by T5, so add edge T4 -> T5
4. **Read(B):** In T5,no subsequent writes to B, so no new edges
5. **Write(C):** C is subsequently read by T6, so add edge T4 -> T6
6. **Write(B):** A is subsequently read by T6, so add edge T5 -> T6
7. **Write(C):** In T6, no subsequent reads to C, so no new edges
8. **Write(A):** In T5, no subsequent reads to A, so no new edges
9. **Write(B):** In T6, no subsequent reads to B, so no new edges

View Serializibality



View Serializability is a procedure to check if the given schedule is consistent or not. It is performed only if the given schedule is not conflict-serializable. It is important to check the consistency of the schedule as an inconsistent schedule leads to an inconsistent state of the program which is undesirable.

1. If a schedule is conflict serializable, then it is view serializable.
2. If a schedule is not view serializable, then it is not a consistent schedule.

How do we check if a schedule is view serializable or not?
First, we check for conflict serializability.
If it is not in conflict serializability then we check if there is any blind write condition among the transactions. If blind write is not found then this schedule cannot be view serializable.
If blind write is found, then we check for it's view equivalence with all the possible orders of serial schedules. If there are ***n*** transactions, then total possible serial schedules are ***n!***.

And we check following conditions between the original and possible serial schedules one by one,

Initial Read

This condition ensures that the initial read of a data item by a transaction is the same in both schedules. It prevents anomalies where a transaction reads a value that has been updated by another transaction in one schedule but not in the other.
Eg, Consider two transactions, T1 and T2. If T1 reads a value of 100 from a data item in one schedule and reads a value of 200 in the other schedule, the initial read condition is violated.

Intermediate Read

This condition guarantees that intermediate reads of a data item by a transaction are consistent in both schedules. It ensures that a transaction's intermediate calculations and decisions are based on the same data in both schedules, preventing inconsistencies in the final results.
Eg, If T1 reads a value of 100, updates it to 200, and then reads it again in one schedule, but reads the original value of 100 in the other schedule, the intermediate read condition is violated.

Final Write

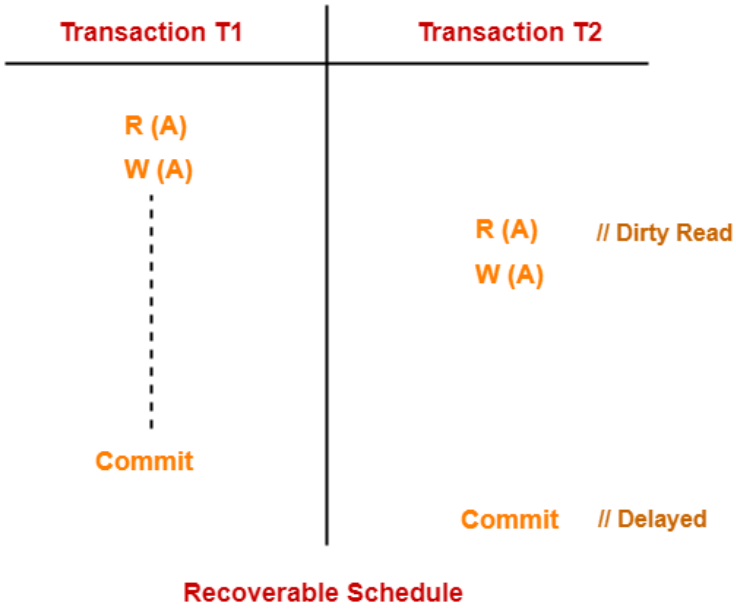
This condition requires that the final write of a data item by a transaction is the same in both schedules. It ensures that the final state of the database is consistent, regardless of the order in which transactions execute.
Eg, If T1 writes a value of 200 to a data item in one schedule and writes a value of 300 in the other schedule, the final write condition is violated.

Great Example, check out here: 8.16 Practice Problem on View Serializability

Recoverable Schedule

A schedule is called recoverable if it allows for the recovery of the database to a consistent state after a transaction failure.
If there is a transaction which is dependent on another transaction then there might pose a situation which makes our database inconsistent. If there is no dependency, then the database is always consistent.

How to decide which transaction is recoverable or not?
Always check for Dirty Read problem occurring between two or more transactions. If there is a dirty read problem, there is dependency and there might be a need to resolve it.



Recoverable Schedule

Just look again at the example we saw of Ram and Shyam in Dirty Read problem.
When Shyam submits his answer sheet after Ram, there won't be any problem. The changes that Ram makes in his sheet can also be copied by Shyam. But when vice versa happens, this puts Shyam in inconsistent situation.

Dirty Read can be resolved by delaying the commit of dependent transaction and committing the non-dependent transaction first.

Cascadeless Schedule

Let's first understand what is a cascading schedule then we'll get to know about cascadelessness.
If there are multiple transactions with multiple depending dirty reads, this creates a chain. If the main transaction rollbacks all the depending transactions also rollback, creating a cascading schedule.

We don't want our schedule to be cascading because in a real-world database there might be many number of depending transactions existing alongside main transaction. And if the main transaction rollback, they rollback too. Thus consuming lots of resources and time for nothing.

Eg, in here if any point *T1* fails before committing, both *T2* and *T3* also rolls back.

T1	***T2***	***T3***

R(A)
W(A)
 	
Commit
	R(A) // Dirty Read
	W(A)
	
	Commit
		R(A) // Dirty Read
		
		Commit

Solution,

T1	***T2***	***T3***

R(A)
W(A)
Commit
	R(A) // Dirty Read
	W(A)
	Commit
		R(A) // Dirty Read
		Commit

In this way, every transaction always reads the updated value brought from database to the main memory.

Cascadeless schedules may contain blind write instructions as they are judged on dirty reads only.

Strict Schedule

Since we looked into all other types of schedules available to us but they don't take into account the problem of blind write (which is one of the biggest issues).
In following example we see that a schedule can be conflict serializable, view serializable, recoverable and cascadeless as well but still suffer from blind write problem.

Eg,

Transaction T1	***Transaction T2***
R (A) W (A) Commit	W (A) // Blind Write R (A) Commit

You can see this is conflict serializable (hence view serializable), it is recoverable as when T1 rollbacks before committing, T2 rollbacks as well. It is also cascadeless as there is no dirty read here.

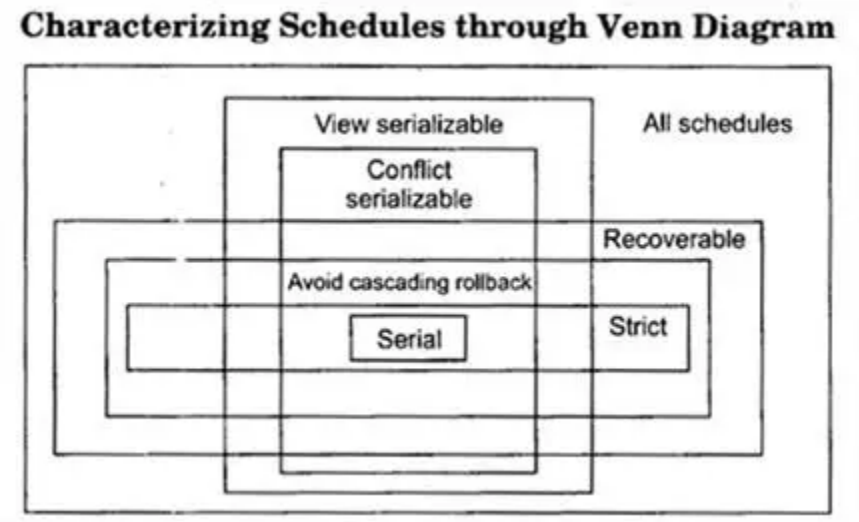
Corrected schedule,

Transaction T1	***Transaction T2***
R (A) W (A) Commit	W (A) R (A) Commit

And in this as well, look R (A) in T2 works only after T1 has committed. Hence, preserving concurrency.

Transaction T1	***Transaction T2***
R (A) W (A) Commit	R (B) W (B) R (A) Commit

A strict schedule requires that if a transaction performs a write operation on a data item, no other transaction can either read or write that data item until the first transaction commits.
Strict schedule does not mean that our schedule can only work in sequential serial manner.



Concurrency Control Protocols

We know how to check if a schedule is consistent or not but doing that is a tedious process. First, you have to check for conflict serializability, then for view serializability, is it recoverable, cascadeless and then so on for a lot more things.
We don't have that much resources and algorithms to do such hectic task.
But what we can do is simply design algorithm which by itself guarantees to follow above mentioned properties (specifically conflict serializability).

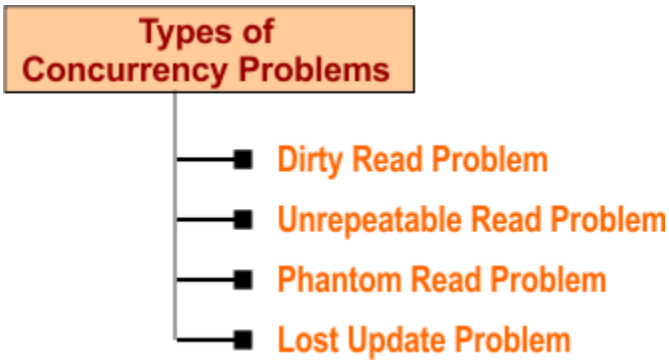
And protocols that ensure conflict serializability are,

1. Time Stamping Protocol
2. Lock Based Protocol
    a. 2PL (basic, conservative, strict, rigorous)
    b. Graph based
3. Validation Protocol

Advantages of Concurrency Control,

1. Reduced Waiting Time
2. Reduced Response Time
3. Increased Resource Utilization
4. Increased Efficiency in handling multiple tasks at the same time

Concurrency control helps in dealing with following problems as well.



Time Stamping Protocol

Have you ever stood in a queue? If yes, then its highly possible you would have seen conflicts or fights occuring between people standing in a queue, it's very common.
Now, what would happen if I replace the queue by giving tokens to people. Everyone would be assured of getting their chance, there won't be any rush or conflicts.

This is what we do in Time Stamping Protocol.
Instead of any normal token, we give each transaction a time stamp based token. And the token value is their entry time when they entered the system. And no two transactions have same entry moments, thus we also ensured uniqueness among them as well.

Why not just use integer values?

Take the example of Air Traffic Controller, their computers run 24x7x365. And the data they deal with is huge, imagine running so many transactions. So, if we used "just integer" values, we'll lose track of numbers and storing such large numbers is a problem in itself.

Properties associated with TS protocol,

1. It always stays fixed and unique for a transaction. Eg, your DoB is going to be what it is forever. Welcome to the SYSTEM!
2. Since, no timestamp repeat itself it becomes easy to decide order of execution between transactions.
3. Time Stamp for a transaction Ti is denoted as *TS (Ti)*.
4. For a data item, we define two values.
    a. Write TS, it is the largest timestamp value of any transaction that has successfully written to that data item.
    b. Read TS, it is the largest timestamp value of any transaction that has successfully read from that data item.
5. Thus if two transactions enter into the system, such that Ti enters before Tj then, *TS (i) < TS (j).*

Advantages of Timestamping Protocol,

1. Timestamp-based protocols in dbms ensure serializability as the transaction is ordered on their creation timestamp.
2. No deadlock occurs when timestamp ordering protocol is used as no transaction waits.
3. No older transaction waits for a longer period of time so the protocol is free from deadlock.
4. Timestamp based protocol in dbms ensures that there are no conflicting items in the transaction execution.

Disadvantages of TS Protocol,

1. Timestamp-based protocols in dbms may not be cascade free or recoverable.
2. In timestamp based protocol in dbms there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction.

Conditions to Request

In every case, **Read-Read** never becomes a condition of conflict. But whenever two transactions work on same data item and any one of them is a **Write**, it creates conflict.
So when a transaction wants to read a value, it should check for write timestamp whereas if the transaction wants to write a value, then it should check for both read and write timestamps.

Ti request for Read (Q)
If *TS (Ti) < WTS (Q)*,
Means that *Ti* wants to read a value which was already overridden by another junior transaction.
Hence, this request must be rejected and *Ti* should rollback and come as junior into the system.

TS (Ti) = 5	***TS (Tx) = 10***
	W (Q)
R (Q) // reading modified value of junior transactions ***Tx***	

If *TS (Ti) >= WTS (Q)*,
Means that *Ti* is a junior transaction and should be allowed to read value.
Also, new value of *RTS (Q)* will be ***max (RTS (Q), TS (Ti))***.

TS (Ti) = 15	***TS (Tx) = 10***
	W (Q)
R (Q) // reading modified value of senior transaction ***Tx***	

Ti request for Write (Q)

If *TS (Ti) < RTS (Q)*,
Means that the value of *Q* was needed previously and *Ti* lost its chance. So, system would reject this request and *Ti* should rollback.
Junior already read a value and is currently working on it. Hence, senior should have made changes before junior came.

TS (Ti) = 5	***TS (Tx) = 10***
W (Q) // writing, already read value of junior transaction ***Tx***	R (Q)

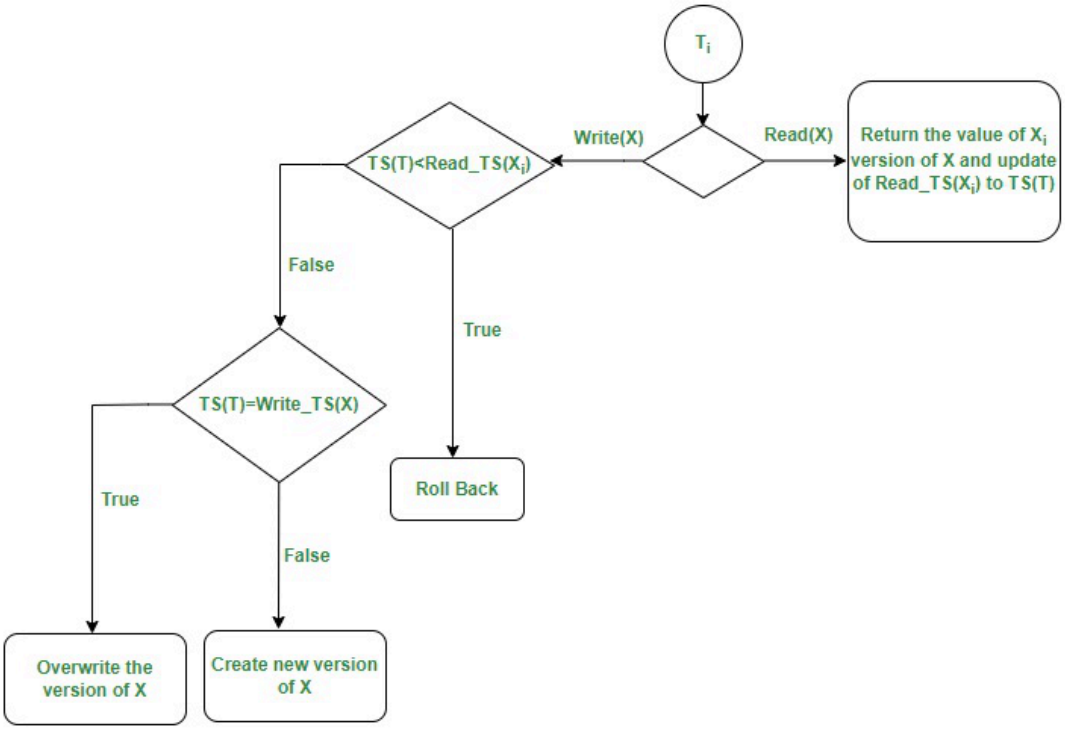
If *TS (Ti) < WTS (Q)*,
Means that *Ti* is trying to write obsolete value of *Q*. It should be rejected and rollback.
Here, junior is working and modifying a value thinking that senior would've done their part.

TS (Ti) = 5	***TS (Tx) = 10***
R (Q) // writing, already modified value of junior transaction ***Tx***	W (Q)

Otherwise,
If above conditions fails, this means we're ready to go. And *Ti* should be given the permission to modify the value of *Q*. Also, new value of *WTS (Q)* will be ***max (WTS (Q), TS (Ti))***.

Properties of Timestamping Protocol

1. It ensures conflict serializability, we never allow the swap requests of conflicting instructions.
2. It ensures view serializability.
3. Possibility of dirty reads as no restriction on commit.
4. Irrecoverable schedules and cascading rollbacks are possible.
5. Deadlocks are not possible, either we allow or reject a transaction.



Thomas Write Rule in TS Protocol

Thomas Write Rule is a modification in the classic Time Stamping Protocol.
Originally, TS Protocol only allowed conflict serializable transactions to execute, which in turn was responsible for multiple rollbacks for senior transactions.
Thomas Write rule modified the original protocol to allow View Serializable transactions to execute as well. This is genius because we only care about consistent state of our database and if that can be achieved even by neglecting some condition then why not do that.

When ***Ti request to Write (Q)*** and
If *TS (Ti) < WTS (Q)*, this should rollback if we follow original TS protocol.
But according to Thomas Write modification, there shouldn't be any rollback, we just simply ignore this operation. Because, final value of *Q* is depends on junior transaction, not on senior. So, if senior wants to write just ignore it and move with the previous value that junior already wrote.

See for yourself in this example,
Original Case (what should've happened), say *Q = 10* and both *Ti* and *Tx* wants to change it.

TS (Ti) = 5	***TS (Tx) = 10***
W (Q), Q = 12	W (Q), Q = 14

Hence, final value of ***Q* should be *14***.

Now, If we ignore *W (Q)* of *Ti*, when *Ti* comes after *Tx*.

TS (Ti) = 5	***TS (Tx) = 10***
W (Q), Q = 12 // ignoring this	W (Q), Q = 14

Final value of ***Q* is still *14***.
So, instead of rolling back, Ti should've continues executing as it's effect is ignored. We got the same result in both scenarios.

Lock Based Protocol

To achieve better concurrency, isolation is most important. This can be achieved simply by using locks. Idea is simple, before doing any update (read or write) just obtain a lock on the data item so that another transaction cannot modify it while first one is still in use.
Above statement simply undermines the idea of concurrency. We can't have better concurrency if all transactions are just acquiring locks every now and then.
That's why locking mechanism is improved and we use two different types of locks for two different purposes so that concurrency doesn't get affected more.

Shared Lock

Shared lock is denoted as *lock_S (Q)*. If a transaction wants to read a data item's value then they acquire shared lock and while they have it, no other transaction can write value for the data item. The most interesting part is that multiple transactions can acquire shared lock at the same time.

Exclusive Lock

Exclusive lock is denoted as *lock_X (Q)*. If a transaction wants to write a data item's value, then they acquire an exclusive lock. And while they have this lock, no other transaction can read or write the data item's value. Another thing is that no two transactions can acquire the exclusive lock at the same time.

Just locking is not enough for proper scheduling among transactions.

T1	***T2***
lock_X (A) R (A) W (A) *unlock (A)*	
	lock_S (B) R (B) *unlock (B)*
lock_X (B) R (B) W (B) *unlock (B)*	
	lock_S (A) R (A) // Dirty Read of A *unlock (A)*
Commit	
	Commit

You can clearly see that T2 is dirty reading the value of A modified by T1. Also, we didn't mentioned the order of commit for these transactions in a simple locking protocol, hence this can lead to problems arising in recoverability and cascadelessness of transactions.
Now, if you check using graph precedence you'll find that above schedule is not conflict serializable as well.
These are the reasons that we use modified versions of Locking Protocols.

Basic 2 Phase Locking Protocol

The Two-Phase Locking (2PL) protocol is one of the most commonly used concurrency control mechanisms in databases. It ensures conflict serializability by regulating how transactions acquire and release locks. The protocol operates in two distinct phases:

Growing Phase

1. In this phase, a transaction can acquire locks (either shared or exclusive), but it cannot release any locks.
2. This phase continues until the transaction acquires all the locks it needs.

Shrinking Phase

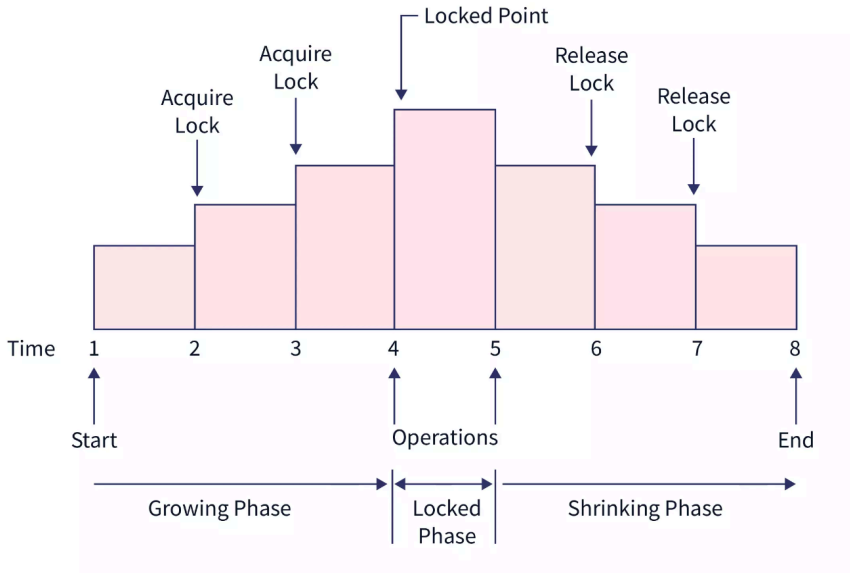
1. Once a transaction releases its first lock, it enters the shrinking phase.
2. In this phase, the transaction can only release locks, but it cannot acquire any new locks.

Properties of the Basic 2PL Protocol

1. Growing and Shrinking Phases
    a. A transaction must acquire all its necessary locks during the growing phase.
    b. Once the transaction starts releasing locks (enters the shrinking phase), it cannot acquire any more locks.
2. Guarantees Conflict Serializability
    a. The key feature of 2PL is that it guarantees conflict serializability. This is achieved by preventing transactions from overlapping their locking and unlocking operations in such a way that leads to non-serializable schedules.
3. Prevents Lost Updates
    a. The protocol ensures that transactions do not overwrite each other's updates or read uncommitted values.
4. Non-Deadlock Free
    a. Basic 2PL can lead to deadlocks because transactions might be waiting for each other to release locks, forming a cycle of dependency.
5. Non-recoverable and Cascading Rollbacks are allowed here.

T1	*T2*
lock_X (A) R (A) W (A)	
	lock_S (B) R (B)
lock_X (B) *// denied lock, gone to wait state*	*lock_S (A)* *// denied lock, gone to wait state*

Now, both T1 and T2 are in wait state, hence they got stuck up in deadlock.



Conservative 2 PL Protocol

The Conservative Two-Phase Locking (C2PL) protocol is a variant of the basic Two-Phase Locking (2PL) protocol designed to prevent deadlocks. In Conservative 2PL, a transaction must acquire all the locks it needs at the very beginning of the transaction, before it starts executing any operations. If it cannot acquire all the necessary locks, it waits until it can.

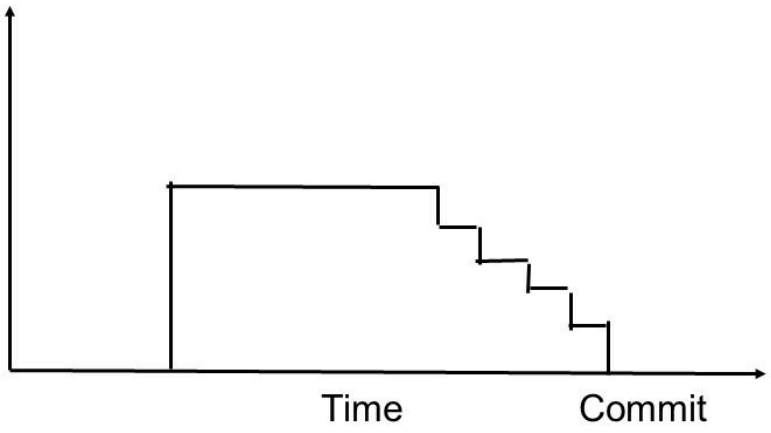
How Conservative 2PL Works

1. ***Lock Preacquisition***
    a. Before the transaction begins executing any operations, it must attempt to acquire all the locks (both shared and exclusive) it will need throughout its lifetime.

2. ***No Partial Lock Acquisition***
    a. If a transaction cannot acquire all of the locks it requires, it waits until all the locks are available. This prevents the situation where a transaction acquires some locks and waits for others, which could lead to deadlocks in other 2PL variants.

3. ***Execution Phase***
    a. Once the transaction successfully acquires all the required locks, it proceeds with executing its operations.

4. ***Release of Locks***
    a. As in basic 2PL, once a transaction starts releasing locks, it cannot acquire any new locks. Thus, after the lock preacquisition phase, the transaction follows the usual rules of growing and shrinking phases.



Properties of Conservative 2PL

1. Deadlock Prevention
    a. The primary advantage of Conservative 2PL is that it prevents deadlocks entirely. By ensuring that transactions acquire all required locks at the beginning, there is no circular wait situation that leads to deadlocks.
2. Ensures Conflict Serializability
    a. Since, every transaction is isolated because of acquiring and releasing locks, it always ensures conflict serializability.
3. Non-recoverable and Cascading Rollbacks are allowed here.

Rigorous 2PL Protocol

The Rigorous Two-Phase Locking (Rigorous 2PL) protocol is a variant of the Two-Phase Locking (2PL) mechanism used in concurrency control in database systems to ensure serializability of transactions. It is called "rigorous" because of the stricter rules it enforces, compared to basic 2PL, to maintain consistency and avoid issues like cascading rollbacks and irrecoverability of schedules.

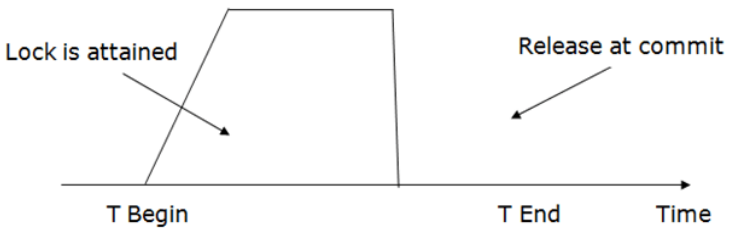
Rigorous 2PL goes a step further by requiring transactions to hold both exclusive (X) and shared (S) locks until the transaction commits or aborts. In other words, no locks are released before the transaction completes, regardless of whether they are shared or exclusive locks.

Steps of Rigorous 2PL Protocol

1. ***Lock Acquisition (Growing Phase)***
    a. When a transaction needs to read or write an item, it requests the appropriate lock (shared for reading, exclusive for writing).
    b. The transaction keeps acquiring locks as necessary without releasing any.
2. ***No Lock Release (Growing Phase Continues)***
    a. Unlike basic 2PL, a transaction cannot release any lock (shared or exclusive) during its execution. Locks are held until the end of the transaction.
3. ***Lock Release (Only After Completion)***
    a. Once the transaction either commits or aborts, it releases all locks at once (this is sometimes called the "atomic release" of locks).
4. ***Commit or Abort***
    a. If the transaction successfully completes (commit), it releases all the locks.
    b. If the transaction fails or is aborted, it rolls back, then releases all locks.

Properties of Rigorous 2PL,

1. Conflict Serializability is ensured.
2. Cascading Rollbacks are prevented.
3. Schedules are recoverable.
4. Deadlocks cannot be prevented.
5. Efficiency is reduced.



Strict 2 PL Protocol

The Strict Two-Phase Locking (Strict 2PL) protocol is a variant of the Two-Phase Locking (2PL) mechanism used in database systems for concurrency control. It ensures serializability while also preventing cascading aborts, making it one of the most commonly used locking protocols in database management systems (DBMS). It is an improvement over Rigorous 2PL.

How Strict 2PL Works,

1. ***Lock Acquisition (Growing Phase)***
    a. A transaction can acquire both shared and exclusive locks as needed during its execution.
    b. For example, a shared lock (*lock_S*) is requested to read a data item, and an exclusive lock (*lock_X*) is requested to modify a data item.
2. ***No Lock Release (Growing Phase)***
    a. During this phase, the transaction cannot release any lock. It must continue to hold all locks until the second phase begins.
3. ***Exclusive Lock Holding (Until Commit/Abort)***
    a. Exclusive locks (*lock_X*) are held until the transaction completes.
    b. If a transaction wants to modify data, it must hold the *lock_X* until it either commits (successfully completes) or aborts (rolls back).
    c. Shared locks (*lock_S*), used for reading, can be released earlier, potentially allowing for more concurrency than in Rigorous 2PL, where even *lock_S* are held until commit.
4. ***Lock Release (Shrinking Phase)***
    a. Once the transaction commits or aborts, it releases all locks it holds at once (including both exclusive and shared locks).

Characteristics of Strict 2PL,

1. Ensures Conflict Serializability.
2. Ensures Recoverability.
3. Ensures Cascadeless Rollbacks.
4. Does not ensures Deadlocks.

Graph Based Protocol

The Graph-Based Locking Protocol is a concurrency control mechanism used in database management systems to ensure the serializability of transactions. Unlike traditional locking protocols such as Two-Phase Locking (2PL), the graph-based approach provides a more structured way of controlling the order in which transactions access data objects.

In this protocol, the data items (database objects) are organized in the form of a Directed Acyclic Graph (DAG), and the locking rules are defined based on the structure of the graph. This ensures that there is a clear and consistent ordering of how transactions acquire locks, which helps prevent deadlocks and maintain serializability.

Key Concepts of Graph-Based Locking Protocol

1. ***Directed Acyclic Graph (DAG)***
    a. The data items (or resources) in the database are represented as nodes in a DAG.
    b. A DAG is a directed graph with no cycles, meaning there is a predefined direction for accessing data items, and transactions must follow this direction.
    c. The edges (directed arcs) between nodes define the allowable access sequence of data items.
2. ***Lock Acquisition Rules***
    a. A transaction can lock any node (data item) if it has locked all its predecessor nodes (in the graph).
    b. Once a transaction has locked a node, it is allowed to request locks on its successor nodes in the graph.
    c. The transaction cannot request locks on a node that precedes the currently locked node (ensuring acyclic behavior).
3. ***Lock Release Rules***
    a. Transactions are allowed to release locks in any order, but once a lock on a node is released, the transaction cannot request new locks on any of its successor nodes in the graph.
    b. This rule prevents cycles and ensures that transactions follow a hierarchical structure in accessing the data.

Graph-Based Locking Process

1. ***Graph Setup***
    a. A directed acyclic graph is created where each node represents a data item, and directed edges represent the allowed order of access.
    b. For example, if there is an edge from node A to node B, a transaction can access A first and then B, but not the other way around.
2. ***Transaction Execution***
    a. A transaction must follow the directed edges of the graph when acquiring locks.
    b. For example, if a transaction wants to lock both data items A and B, and if A precedes B in the graph (i.e., there is a directed edge from A to B), the transaction must lock A before it can lock B.
3. ***Lock Acquisition***
    a. When a transaction needs to access a data item, it checks whether it has locked all the predecessor nodes in the graph.
    b. If all predecessors are locked, the transaction can proceed to acquire the lock on the desired node (data item).
    c. If not, the transaction must wait until it acquires the necessary predecessor locks.
4. ***Lock Release***
    a. Transactions can release locks in any order.
    b. Once a lock on a node is released, the transaction is not allowed to acquire any locks on its successors (nodes that follow the released node in the graph).

Advantages of Graph-Based Locking Protocol

1. **Deadlock-Free:** Since the transactions must follow the directed acyclic structure of the graph when acquiring locks, cycles cannot occur, thereby eliminating deadlocks.
2. **Serializability:** The protocol enforces a global order on the access to data items, ensuring that the transaction schedule is serializable.
3. **More Concurrency:** Since locks can be released at any time and transactions can lock multiple nodes simultaneously (as long as they follow the graph structure), there can be higher concurrency compared to strict protocols like 2PL.

Disadvantages of Graph-Based Locking Protocol

1. **Complex Setup:** Defining the graph of data items (DAG) can be complex, especially in large systems with many data items.
2. **Limited Flexibility:** The rigid structure of the graph limits the flexibility in how transactions can acquire locks. Transactions are forced to follow a predefined order, which may not always align with the natural order of access for all transactions.
3. **Static Structure:** The graph-based protocol assumes a static structure for lock acquisition. If the relationships between data items or access patterns change dynamically, the DAG might need frequent adjustments, which can be impractical.

Deadlocks in DBMS

Deadlock in a Database Management System (DBMS) occurs when two or more transactions block each other by holding locks on resources the other transactions need. As a result, none of the transactions can proceed, leading to a standstill.

Example of Deadlock

1. Transaction T1 locks resource A and waits for resource B.
2. Transaction T2 locks resource B and waits for resource A.

Both transactions are stuck, creating a circular wait, and neither can finish.

Deadlock Handling

DBMSs use various techniques to detect, prevent, or resolve deadlocks. Deadlock handling involves either detecting a deadlock once it occurs and then resolving it or taking preventive measures to avoid deadlock situations.

There are two main strategies to deal with deadlocks

1. **Deadlock Prevention:** Ensure that the system never enters a deadlock state.
2. **Deadlock Detection and Recovery:** Allow the system to enter a deadlock state and then detect and resolve it.

Deadlock Prevention

Deadlock prevention techniques ensure that the system avoids any situation that could potentially cause a deadlock by carefully regulating how locks are acquired.

Four Necessary Conditions for Deadlock,

1. **Mutual Exclusion:** At least one resource is held in a non-shareable mode (i.e., only one transaction can use it at a time).
2. **Hold and Wait:** Transactions holding at least one resource are allowed to request additional resources.
3. **No Preemption:** Resources cannot be forcibly removed from a transaction holding them.
4. **Circular Wait:** A closed chain of transactions exists, where each transaction holds a resource that the next transaction in the chain is waiting for.

By ensuring one or more of these conditions do not occur, deadlock can be prevented.

Deadlock Prevention Strategies

Prevention of Circular Wait

Impose a global order on resource acquisition

1. Assign a numerical or hierarchical order to all resources.
2. A transaction can only request a resource if it has locked all resources with lower order numbers first.
3. This prevents circular waiting because transactions can only request higher-order resources, breaking the cycle.

Example,
If resource A has a lower order than resource B, transactions must acquire A before B.

Eliminate Hold and Wait

Require all resources to be requested at the beginning

1. Transactions must acquire all the resources they will need before starting, or they release all locks and try again later with all required locks.
2. This prevents situations where a transaction holds some resources and waits for others, thus eliminating the possibility of deadlock.

Disadvantage, It reduces concurrency, as transactions may have to wait longer to get all resources at once.

No Preemption

Allow resource preemption

1. If a transaction holding some resources requests a new resource and cannot acquire it, it is forced to release its current resources and restart later.
2. This avoids deadlock by forcibly removing resources from transactions that are unable to proceed.

Prevention of Mutual Exclusion

1. In certain cases, deadlocks can be avoided by allowing multiple transactions to share resources simultaneously. However, this is only possible for resources that can be safely shared (i.e., those not involved in modifications).

Deadlock Detection and Recovery

Deadlock detection allows the system to enter a deadlock state and then uses mechanisms to detect and recover from the deadlock.

Deadlock Detection

1. In this approach, the DBMS allows transactions to proceed, assuming deadlocks might occur, and periodically checks the system for deadlocks.
2. The system uses a Wait-For Graph (WFG) to detect deadlocks.

Wait-For Graph (WFG)

1. Nodes in the graph represent active transactions.
2. Directed edges between nodes represent that one transaction is waiting for a resource held by another transaction.
3. If a cycle is detected in the graph, a deadlock exists.

Algorithm

1. The system builds a WFG by analyzing which transactions are waiting for which resources.
2. It periodically checks the WFG for cycles.
3. If a cycle is found, it indicates a deadlock, and the system must recover.

Deadlock Recovery

Once a deadlock is detected, the system needs to recover by breaking the deadlock. There are a few recovery techniques.

Transaction Rollback (Victim Selection)

1. The system selects one or more transactions to be rolled back (aborted) to break the deadlock cycle.
2. *The victim transaction is chosen based on certain criteria,*
    a. Least amount of work done: Prefer to abort transactions that have made the least progress.
    b. Least cost: Choose the transaction that will incur the least cost in terms of rollback (fewest updates made).
    c. Priority: Some transactions may be given higher priority based on their importance.

Disadvantage, The chosen transaction will lose its progress, and the system will need to restart it later.

Transaction Preemption

1. The system can preempt some transactions by releasing their locks on resources. This allows other transactions to proceed. The preempted transactions will then need to reattempt later.

Timeouts

1. A simpler way of detecting potential deadlocks is using timeouts. If a transaction waits too long for a resource, the system assumes that it is part of a deadlock and aborts the transaction.

Disadvantage, Timeouts can result in unnecessary rollbacks if the transaction is not part of a deadlock but is just waiting longer than usual.

Aspect	**Deadlock Prevention**	**Deadlock Detection and Recovery**
Approach	Avoid deadlock situations from arising.	Detect deadlock after it occurs, then resolve it.
Overhead	Imposes extra constraints on transactions to prevent deadlock (e.g., locking order, no hold-and-wait).	Periodically check for deadlocks, which can have performance costs.
Resource Utilization	Can reduce concurrency and system utilization by enforcing restrictive locking behavior.	Allows more concurrency until a deadlock actually occurs.
Complexity	Complex to implement, especially for large, dynamic systems.	Requires deadlock detection algorithms like Wait-For Graph analysis.
Transaction Aborts	May prevent deadlock by delaying transactions, but transactions are not aborted unnecessarily.	Involves aborting transactions to resolve deadlock after detection.

MySQL

We'll study SQL queries using MySQL.

Installation

Download MySQL from here: [Download MySQL Installer]

Take help from here (Windows): How to install MySQL 8.0.39 Server and Workbench latest versio…

For (Linux/ Ubuntu): How To Install MySQL on Ubuntu 22.04 LTS (Linux)

For (macOS): How to Install MySQL on Mac | Install MySQL on macOS (2024)

Create user, password (remember all along).

Connecting & Disconnecting to Database

```
mysql -u <username> -p
<password>

exit;
```

Learn tutorial from here: [MySQL SQL (w3schools.com)]

Or watch from here: SQL Tutorial for Beginners [Full Course]

Data Types in MySQL

Integer Data Types

Data Type	Storage (Bytes)	Minimum Value (Signed)	Maximum Value (Signed)	Minimum Value (Unsigned)	Maximum Value (Unsigned)
TINYINT	1	-128	127	0	255
SMALLINT	2	-32,768	32,767	0	65,535
MEDIUMINT	3	-8,388,608	8,388,607	0	16,777,215
INT	4	-2,147,483,648	2,147,483,647	0	4,294,967,295
BIGINT	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807	0	18,446,744,073,709,551,615

Floating Point Types

Data Type	Storage (Bytes)	Description
FLOAT	4	Single-precision floating-point number. Approximate range: **±1.175494351e-38** to **±3.402823466e+38**
DOUBLE	8	Double-precision floating-point number. Approximate range: **±2.2250738585072014e-308** to **±1.7976931348623157e+308**
DECIMAL	Varies	Fixed-point exact number. Can specify precision (number of digits) and scale (digits to right of the decimal point). Example: DECIMAL(5, 2) allows numbers like 999.99. Default: DECIMAL(10, 0)

Boolean Types

Data Type	Storage (Bytes)	Description
BOOLEAN	1	Synonym for TINYINT(1), values are 0 (false) and 1 (true).

Character Types

Data Type	Storage (Bytes)	Description
CHAR(n)	n bytes (fixed)	Fixed-length string of length n (0 to 255 characters). Pads with spaces for unused length.
VARCHAR(n)	Varies	Variable-length string, up to n characters (maximum 65,535 bytes depending on encoding and table row size).

Text Data Types

Data Type	Storage (Bytes)	Description
TINYTEXT	Up to 255 bytes	Holds up to 255 characters.
TEXT	Up to 65,535 bytes	Holds up to 65,535 characters (64 KB).
MEDIUMTEXT	Up to 16,777,215 bytes	Holds up to 16,777,215 characters (16 MB).
LONGTEXT	Up to 4,294,967,295 bytes	Holds up to 4,294,967,295 characters (4 GB).

Binary Types

Data Type	Storage (Bytes)	Description
BINARY(n)	n bytes (fixed)	Fixed-length binary string. Similar to CHAR but stores binary data.
VARBINARY(n)	Varies	Variable-length binary string. Similar to VARCHAR but for binary data.
TINYBLOB	Up to 255 bytes	Holds up to 255 bytes of binary data.
BLOB	Up to 65,535 bytes	Holds up to 65,535 bytes (64 KB) of binary data.
MEDIUMBLOB	Up to 16,777,215 bytes	Holds up to 16,777,215 bytes (16 MB) of binary data.
LONGBLOB	Up to 4,294,967,295 bytes	Holds up to 4,294,967,295 bytes (4 GB) of binary data.

Enum and Set Types

Data Type	Description
ENUM('val1', 'val2', ...)	A string object that holds one value chosen from a list of predefined values. Can store up to 65,535 distinct values.
SET('val1', 'val2', ...)	A string object that can hold multiple values chosen from a list of predefined values. Can store up to 64 distinct values.

Date and Time Data Types

Data Type	Storage (Bytes)	Description
DATE	3	Stores a date in the format YYYY-MM-DD. Range: 1000-01-01 to 9999-12-31.
TIME	3	Stores a time in the format HH:MM:SS. Range: -838:59:59 to 838:59:59.
DATETIME	5	Stores a date and time in the format YYYY-MM-DD HH:MM:SS. Range: 1000-01-01 00:00:00 to 9999-12-31 23:59:59.
TIMESTAMP	4	Stores a timestamp in the format YYYY-MM-DD HH:MM:SS. Range: 1970-01-01 00:00:01 UTC to 2038-01-19 03:14:07 UTC. Auto-updates on row changes.
YEAR	1	Stores a year in 2 or 4 digits. Range: 1901 to 2155 (4-digit format), 70 to 69 (2-digit format: represents years 1970 to 2069).

Basic MySQL Queries

```
    -- To get the current date and time, you can use the NOW() function.
    SELECT NOW();

    -- To get only the current date, use the CURDATE() function.
    SELECT CURDATE();

    -- To retrieve only the time portion, use the CURTIME() function.
    SELECT CURTIME();

    -- To get the currently authenticated MySQL user, use the USER() or
CURRENT_USER() function.
    SELECT USER();
    SELECT CURRENT_USER();

    -- To retrieve the version of the MySQL you're connected to, use the
VERSION() function.
    SELECT VERSION();

    -- To see a list of all databases available on the server:
    SHOW DATABASES;

    -- To list all the tables in the currently selected database:
    SHOW TABLES;

    -- To see the name of the database you are currently using:
    SELECT DATABASE();
```

Create Database & Create Table

```sql
    -- To create a new database, use the CREATE DATABASE statement.
    CREATE DATABASE database_name;

    -- Example,
    CREATE DATABASE employees;

    -- After creating the database, you need to select it for use with the USE
command:
    USE my_database;

    -- To create a table within the selected database, use the CREATE TABLE
statement. A table requires specifying columns and their data types.
    CREATE TABLE table_name (
        column1_name column1_data_type,
        column2_name column2_data_type,
        ...
    );

    -- Example,
    CREATE TABLE employees (
        employee_id INT AUTO_INCREMENT PRIMARY KEY,
        first_name VARCHAR(50),
        last_name VARCHAR(50),
        hire_date DATE,
        salary DECIMAL(10, 2)
    );
```

Constraints in MySQL

```
    -- UNIQUE Constraint
    -- Ensures that all values in a column are unique
    CREATE TABLE products (
        product_id INT PRIMARY KEY,
        product_name VARCHAR(100) UNIQUE,
        price DECIMAL(10, 2) NOT NULL
    );

    -- PRIMARY KEY Constraint
    -- A special type of UNIQUE constraint that uniquely identifies each row in a
table. It can be defined on a single column or a combination of columns
    CREATE TABLE customers (
        customer_id INT PRIMARY KEY,
        first_name VARCHAR(50),
        last_name VARCHAR(50)
    );

    -- AUTO_INCREMENT Attribute
    -- Automatically generates a unique integer value for each new row. It's
often used with PRIMARY KEY constraints
    CREATE TABLE orders (
        order_id INT PRIMARY KEY AUTO_INCREMENT,
        customer_id INT,
        order_date DATE
    );

    -- FOREIGN KEY Constraint
    -- Defines a relationship between columns in two tables. It ensures that the
values in the foreign key column match existing values in the referenced primary
key column
    CREATE TABLE order_items (
        order_item_id INT PRIMARY KEY,
        order_id INT,
        product_id INT,
        FOREIGN KEY (order_id) REFERENCES orders(order_id),
        FOREIGN KEY (product_id) REFERENCES products(product_id)
    );
```

```sql
    -- NOT NULL Constraint
    -- Ensures that a column cannot contain NULL values
    CREATE TABLE customers (
        customer_id INT PRIMARY KEY,
        first_name VARCHAR(50) NOT NULL,
        last_name VARCHAR(50) NOT NULL
    );

    -- CHECK Constraint
    -- Verifies that data in a column meets a specified condition
    CREATE TABLE products (
        product_id INT PRIMARY KEY,
        product_name VARCHAR(100),
        price DECIMAL(10, 2) CHECK (price >= 0)
    );

    -- DEFAULT Constraint
    -- Specifies a default value for a column if no value is provided when
inserting a new row
    CREATE TABLE orders (
        order_id INT PRIMARY KEY AUTO_INCREMENT,
        order_date DATE DEFAULT CURDATE()
    );
```

Altering with Table Structure

```
    -- Renaming a Table Name
    ALTER TABLE old_table_name RENAME TO new_table_name;

    -- Renaming a Column Name
    ALTER TABLE table_name RENAME COLUMN old_column_name TO new_column_name;

    -- Modifying Column Structure
    ALTER TABLE table_name MODIFY COLUMN column_name data_type [NOT NULL]
[DEFAULT value];

    -- Deleting a Column
    ALTER TABLE table_name DROP COLUMN column_name;

    -- Adding a Column
    ALTER TABLE table_name ADD COLUMN column_name data_type [AFTER
existing_column];

    -- Deleting Complete Table Entries Plus Structure
    DROP TABLE table_name;

    -- Deleting Only Entries from a Table
    TRUNCATE TABLE table_name;
```

SELECT Statement

```sql
-- Basic SELECT Statement
SELECT column1, column2, ...
FROM table_name;

-- Example,
SELECT customer_id, first_name, last_name
FROM customers;
```

SELECT with WHERE Clause

```sql
-- SELECT Statement with WHERE Clause
SELECT column1, column2, ...
FROM table_name
WHERE condition;

-- Example,
SELECT *
FROM orders
WHERE order_date > '2023-12-31';
```

SELECT with AND, OR, NOT

```sql
-- SELECT Using AND, OR, NOT
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2;

SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2;
```

```sql
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;

-- Example,
SELECT *
FROM products
WHERE price > 100 AND category = 'Electronics';

SELECT *
FROM customers
WHERE country = 'USA' OR country = 'Canada';

SELECT *
FROM orders
WHERE NOT order_status = 'Cancelled';
```

INSERT, UPDATE and DELETE

```sql
-- INSERT Query
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);

-- Example,
INSERT INTO customers (customer_id, first_name, last_name, email)
VALUES (1001, 'John', 'Doe', 'johndoe@example.com');


-- UPDATE Query
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;

-- Example,
UPDATE orders
SET order_status = 'Shipped'
WHERE order_id = 1000;


-- DELETE Query
DELETE FROM table_name
WHERE condition;

-- Example,
DELETE FROM products
WHERE product_id = 101;
```

ORDER BY, GROUP BY, LIMIT and HAVING

```sql
-- ORDER BY Query
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 ASC|DESC, column2 ASC|DESC, ...;

-- Example,
SELECT customer_id, first_name, last_name
FROM customers
ORDER BY last_name ASC;


-- GROUP BY Query
SELECT column1, column2, ...
FROM table_name
GROUP BY column1, column2, ...;

-- Example,
SELECT order_status, COUNT(*) AS total_orders
FROM orders
GROUP BY order_status;


-- LIMIT Query
SELECT column1, column2, ...
FROM table_name
LIMIT offset, count;

-- Example,
SELECT product_name, price
FROM products
LIMIT 5;


-- HAVING Query
SELECT column1, column2, ...
FROM table_name
GROUP BY column1, column2, ...
HAVING condition;
```

```sql
-- Example,
SELECT order_status, COUNT(*) AS total_orders
FROM orders
GROUP BY order_status
HAVING total_orders > 10;
```

String Operations

```sql
    -- Length of a String
    SELECT LENGTH(column_name)
    FROM table_name;

    -- Uppercasing a String
    SELECT UPPER(column_name)
    FROM table_name;

    -- Lowercasing a String
    SELECT LOWER(column_name)
    FROM table_name;

    -- Extract a substring
    SELECT SUBSTRING(product_name, 1, 5) AS first_5_chars
    FROM products;

    -- Remove leading and trailing whitespace
    SELECT TRIM(description)
    FROM products;

    -- Replace all occurrences of old_string with new_string
    SELECT REPLACE(product_name, 'old', 'new')
    FROM products;

    -- Returns the position of substring within string
    SELECT LOCATE('product', product_name) AS product_position
    FROM products;
    -- This will find the position of the substring "product" within the
product_name column. If "product" doesn't exist, it will return 0.
```

Aggregate Functions

```sql
-- Returns the minimum value of a column
SELECT MIN(price) AS lowest_price
FROM products;

-- Returns the maximum value of a column
SELECT MAX(order_date) AS latest_order
FROM orders;

-- Counts the number of rows in a table or the number of non-NULL values in a column
SELECT COUNT(*) AS total_customers
FROM customers;

SELECT COUNT(city) AS total_cities
FROM customers;

-- Calculates the average value of a column
SELECT AVG(price) AS average_price
FROM products;

-- Calculates the sum of the values in a column
SELECT SUM(quantity) AS total_quantity
FROM order_items;
```

Additional Notes

1. You can use these functions with or without the GROUP BY clause.
2. When used with GROUP BY, these functions calculate the values for each group.
3. You can combine these functions with other SQL statements to perform complex calculations.

```sql
SELECT customer_id, SUM(total_amount) AS total_spent
FROM orders
GROUP BY customer_id;
```

Comparison Operators and Logical Expressions

```
    -- Used for pattern matching
    SELECT *
    FROM products
    WHERE product_name LIKE '%laptop%';

    -- Checks if a value is within a list of values, more like multiple OR
    SELECT *
    FROM orders
    WHERE order_status IN ('Shipped', 'Delivered');

    -- Checks if a value is within a range inclusive
    SELECT *
    FROM products
    WHERE price BETWEEN 100 AND 200;

    -- Checks if a value is NULL
    SELECT *
    FROM customers
    WHERE email IS NULL;

    -- Checks if a subquery returns any rows
    SELECT customer_id
    FROM customers
    WHERE EXISTS (SELECT 1 FROM orders WHERE orders.customer_id =
customers.customer_id);
```

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%'	Finds any values that start with "a" and are at least 2 characters in length
WHERE CustomerName LIKE 'a__%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

Joins

```
    -- INNER JOIN
    -- Returns rows that have matching values in both tables
    SELECT customers.customer_id, customers.first_name, orders.order_id
    FROM customers
    INNER JOIN orders ON customers.customer_id = orders.customer_id;

    -- LEFT JOIN
    -- Returns all rows from the left table, even if there are no matches in the
right table
    SELECT customers.customer_id, customers.first_name, orders.order_id
    FROM customers
    LEFT JOIN orders ON customers.customer_id = orders.customer_id;

    -- RIGHT JOIN
    -- Returns all rows from the right table, even if there are no matches in
the left table
    SELECT customers.customer_id, customers.first_name, orders.order_id
    FROM customers
    RIGHT JOIN orders ON customers.customer_id = orders.customer_id;

    -- FULL OUTER JOIN
    -- Returns all rows when there is a match in either left or right table
    SELECT customers.customer_id, customers.first_name, orders.order_id
    FROM customers
    FULL OUTER JOIN orders ON customers.customer_id = orders.customer_id;

    -- SELF JOIN
    -- Joins a table with itself
    SELECT e1.employee_id, e1.first_name, e2.employee_id, e2.first_name
    FROM employees e1
    JOIN employees e2 ON e1.manager_id = e2.employee_id;
```

CASE Expression

The CASE expression is used to conditionally select different values based on specific conditions. It's similar to an IF-ELSE statement in programming languages.

```
SELECT product_id, product_name,
    CASE
        WHEN price > 1000 THEN 'Expensive'
        WHEN price BETWEEN 500 AND 1000 THEN 'Moderate'
        ELSE 'Budget-friendly'
    END AS price_category
FROM products;
```

IFNULL() and COALESCE() Functions

Both IFNULL() and COALESCE() functions are used to replace NULL values with a specified alternative.

```
-- Returns expr1 if it's not NULL, otherwise returns expr2
SELECT customer_name, IFNULL(email, 'Unknown Email') AS email_address
FROM customers;

-- Returns the first non-NULL expression from the list
SELECT product_name, COALESCE(price, 0) AS price_or_zero
FROM products;
```

1. IFNULL() takes only two arguments.
2. COALESCE() can take multiple arguments.

Views in MySQL

Views are virtual tables derived from the results of a SELECT statement. They provide a way to present data in a specific format or to simplify complex queries.

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table1, table2, ...
WHERE condition;
```

User Defined Functions

```
CREATE FUNCTION calculate_discount(price DECIMAL(10, 2))
RETURNS DECIMAL(10, 2)
BEGIN
    DECLARE discount DECIMAL(10, 2);
    IF price > 1000 THEN
        SET discount = price * 0.1;
    ELSE
        SET discount = price * 0.05;
    END IF;
    RETURN discount;
END;
```

Stored Procedures

Stored procedures are reusable code blocks that can be executed as a single unit. They can take input parameters and return output parameters.

```
CREATE PROCEDURE get_customer_by_id(IN customer_id INT)
BEGIN
    SELECT customer_id, first_name, last_name
    FROM customers
    WHERE customer_id = customer_id;
END;
```

Introduction to File Structures in DBMS

File Structures in a Database Management System (DBMS) refer to the organization and storage of data in files on secondary storage devices like hard drives or SSDs. The file structure defines how data is physically stored, accessed, and managed in the DBMS, allowing for efficient retrieval and modification of data.

The performance and efficiency of any database system largely depend on how files are structured and accessed. File structures in DBMS can vary based on factors like access methods, data organization, and storage medium.

Hierarchy Flow of developing a DBMS for product,

SRS ER Model Relational Model Normalization File Structures (indexes & physical structures)

Why do indexing? *Because, indexing can make the fetch of results very fast.*
What physical structures are used? *B Trees and B+ Trees.*

Components of File Structures in DBMS

1. ***Records***
    a. The basic unit of data storage in a file. A record is a collection of fields that represent an entity (e.g., a row in a relational database table).
2. ***Fields***
    a. A field is a single piece of information in a record (e.g., a column in a relational table). Each field stores a specific attribute of an entity.
3. ***Blocks***
    a. Files are stored on disk in units called blocks. A block is the smallest unit of data that can be transferred between the disk and memory. Multiple records are stored in one block.
4. ***Pages***
    a. In some systems, blocks are referred to as pages. A page is typically the unit of data that is read from or written to the disk.

*In a Database Management System (DBMS), the way records are stored in files can vary based on whether they are **sorted** or **unsorted**, and whether the records are **spanned** or **unspanned**. These characteristics impact how data is accessed, modified, and managed.*

Sorted File Structure

A sorted file is a file where the records are stored in a specific order, typically based on a key field (e.g., Employee ID, Account Number).

1. **Organization:** Records are physically arranged on disk in ascending or descending order of the key attribute. If a record is inserted, the system ensures that the order is maintained by placing the record in its appropriate position.
2. **Characteristics**
    a. **Search Efficiency:** Efficient for searching, especially for range queries, as binary search or sequential search can be used.
    b. **Insertion Overhead:** Inserting a new record requires finding the correct position and shifting records if necessary, which may incur additional overhead.
    c. **Sequential Access:** Highly efficient for sequential access, where records are retrieved in order.

Use Case, Sorted file structures are commonly used when records need to be accessed in a specific order or for applications with frequent range queries (e.g., retrieving all employee records between two employee IDs).

Unsorted (Heap) File Structure

An unsorted file, also known as a heap file, stores records in the order they are inserted, without any specific arrangement based on key values.

1. **Organization:** When a new record is added, it is simply placed in the next available space (end of the file or in a gap created by a deleted record). There is no particular order in which records are stored.
2. **Characteristics**
    a. **Search Inefficiency:** Searching for a specific record involves scanning the entire file (sequential search), making retrieval inefficient for large datasets.
    b. **Efficient Insertion:** Inserting new records is fast, as the system simply adds the new record to the next available location.
    c. **Deletion Overhead:** Deleting records can leave gaps, which may cause fragmentation. Compacting the file to eliminate gaps can be costly.

Use Case, Unsorted files are used when there is no need for ordered retrieval of records, and the primary focus is on efficient insertions (e.g., logging systems, bulk data storage).

Spanned File Structure

A spanned file allows a single record to span across multiple blocks (or pages). This is useful when a record is too large to fit into a single block.

1. **Organization:** If a record is larger than the block size, it is split, and its parts are stored across multiple blocks. The DBMS maintains pointers to link these blocks together, allowing the entire record to be reconstructed when accessed.
2. **Characteristics**
    a. **Space Efficiency:** Makes efficient use of storage space, as large records do not need to be restricted to block size limits.
    b. **Complex Access:** Accessing a spanned record may involve reading multiple blocks and assembling the record in memory, which can introduce overhead.
    c. **Record Size Flexibility:** Spanned files are ideal for databases with variable-length records, especially when some records are significantly larger than others.

Use Case, Spanned file structures are used when records are variable in size, and some records are too large to fit into a single block (e.g., multimedia data, documents, or records with large text fields).

Unspanned File Structure

An unspanned file does not allow records to span across multiple blocks. Each record must fit entirely within a single block.

1. **Organization:** If a record is too large to fit into a single block, the system either rejects the record or requires splitting it into smaller records. Each block is treated independently, and no record spans multiple blocks.
2. **Characteristics**
    a. **Simpler Access:** Accessing records is straightforward since each record is entirely contained within one block. This minimizes the need for multiple I/O operations when retrieving a single record.
    b. **Wasted Space:** If the record is smaller than the block size, it may lead to unused space in the block. Additionally, if a record is slightly larger than the block size, it cannot be stored without modification.
    c. **Record Size Limit:** There is a limit to how large a record can be, which is dictated by the block size.

Use Case, Unspanned file structures are used when all records are of fixed or relatively small sizes, and the system benefits from simple record access without the need for handling multiple blocks (e.g., when storing metadata or small-sized records like fixed-length fields).

Indexing

Indexing in a Database Management System (DBMS) is a technique used to improve the speed of data retrieval operations. An index is a data structure that allows for quick lookups of records in a database table. By creating an index on one or more columns, the DBMS can avoid scanning the entire table when searching for records, thus improving query performance.

Why Indexing is Important

1. **Faster Search:** Without an index, searching for a specific record may require scanning the entire table (a full table scan), which can be inefficient, especially for large datasets.
2. **Efficient Sorting:** Indexes maintain data in a sorted order, which makes operations like range queries or sorting much faster.
3. **Speeds Up Queries:** Indexing speeds up queries that involve selection, joins, and aggregation by reducing the number of disk I/O operations required.

Structure of an Index

1. An index typically consists of **keys** (values from one or more columns in a table) and **pointers** (references to the actual records in the table).
2. When a query is executed, the DBMS searches the index first to quickly locate the records in the table.

Types of Indexing in DBMS

Indexes can be categorized based on different criteria, such as the number of keys used, the data structure used to implement them, and their physical storage.

1. Primary Indexing
2. Secondary Indexing
3. Clustered Indexing

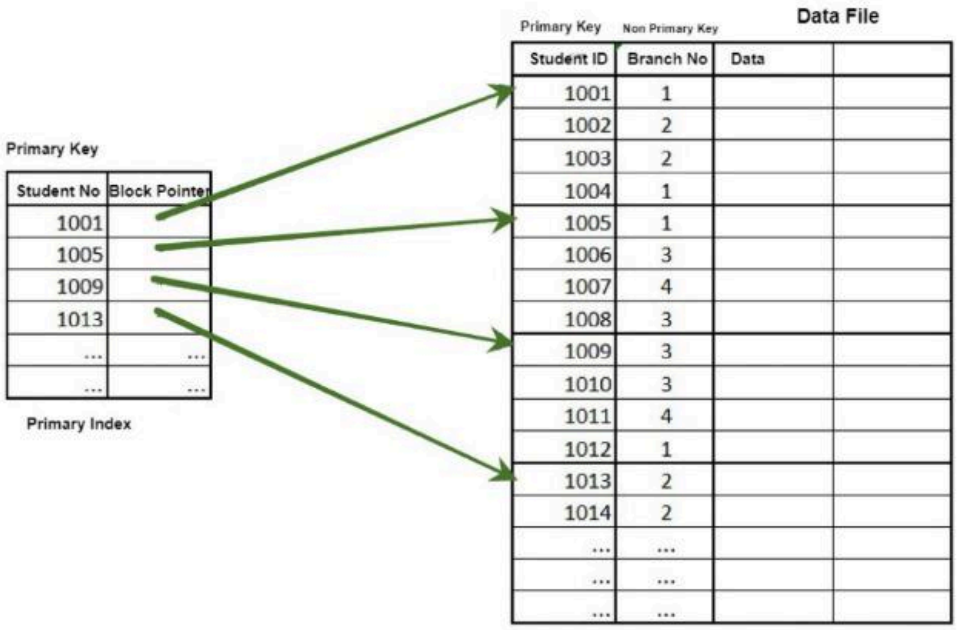
Primary Index

1. **Definition:** A primary index is created on a table's primary key field. Since the primary key is unique for each record, the index contains one entry per record.
2. **Structure:** The records in the data file are sorted on the primary key, and the index contains the primary key values and pointers to the records.

Types of Primary Index

1. **Dense Primary Index:** Every search key (primary key) in the data table has an entry in the index. Each index entry points to a specific record.
2. **Sparse Primary Index:** The index does not have entries for every record. Instead, it only has entries for some of the keys, with pointers to blocks. The first record in each block is indexed.

Use Case, A primary index is used for fast lookups in tables where the records are physically sorted based on the primary key.
Example, In a table of employee records where each employee has a unique EmployeeID, a primary index on the EmployeeID allows for efficient searching of employee data.



Sparse Primary Index

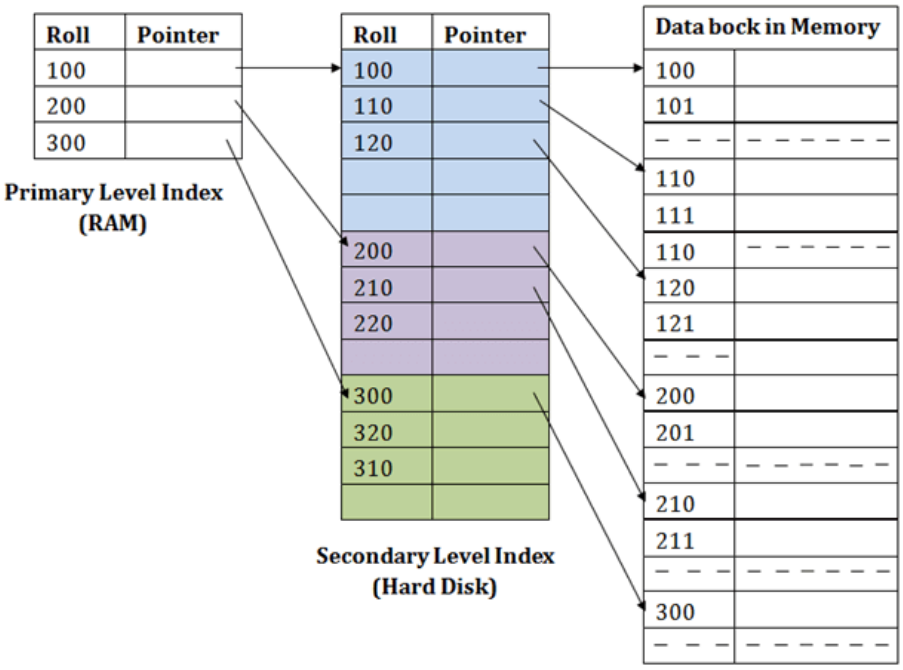
Secondary Index

1. **Definition:** A secondary index is created on non-primary key fields (i.e., attributes that are not unique for each record). This index is used to improve the speed of searches that involve attributes other than the primary key.
2. **Structure:** The secondary index contains the values of the indexed field and pointers to the records.

Types of Secondary Index

1. **Dense Secondary Index:** Like the dense primary index, it has an index entry for every record.
2. **Sparse Secondary Index:** A sparse secondary index contains index entries only for some records.

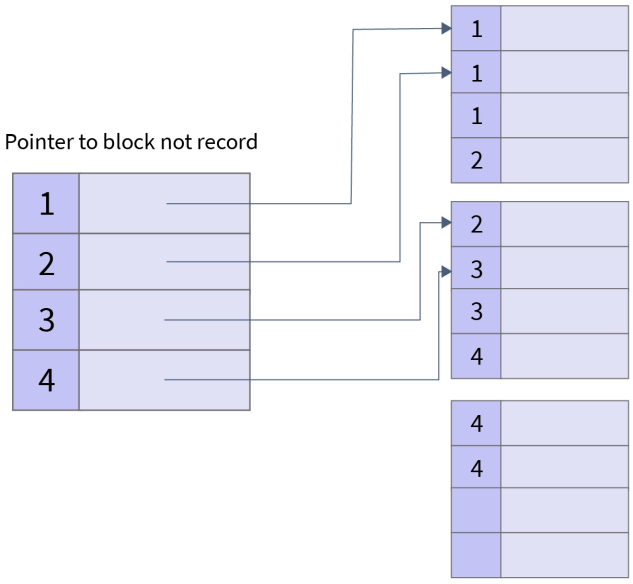
Use Case, Secondary indexes are useful when queries frequently involve fields that are not part of the primary key.
Example, In a database where customer orders are stored, a secondary index on OrderDate can help quickly retrieve all orders from a specific date.



Clustered Index

1. **Definition:** A clustered index determines the physical order of data in the table. The table is stored in the same order as the index. A table can have only one clustered index because records can only be stored in one specific order.
2. **Structure:** In a clustered index, the rows are sorted based on the index key values, and the data is stored in that sorted order.

Use Case, Clustered indexes are useful when queries often retrieve a range of values, as the data is stored in sorted order.
Example, In a customer database, a clustered index on CustomerID ensures that the rows are stored in the order of CustomerID.
Note, In most relational databases, the primary key automatically creates a clustered index unless specified otherwise.



Aspect	Primary Index	Secondary Index	Clustered Index
Definition	Created on the primary key of the table.	Created on non-primary key columns.	Determines the physical order of rows in the table.
Uniqueness	Unique by default, as it is based on the primary key.	May or may not be unique.	Can be unique or non-unique (typically based on the primary key).
Physical Storage	Does not affect the physical storage of data.	Does not affect the physical storage of data.	Affects the physical order of the data in the table.
Number per Table	One per table (as there is only one primary key).	Multiple allowed per table.	One per table (since it determines physical order).
Data Retrieval	Efficient for queries using the primary key.	Efficient for queries on the indexed columns but not on the primary key.	Efficient for queries where sorting or range retrieval is needed.
Index Type	Typically a dense index.	Can be dense or sparse.	Typically a dense index.
Usage	Primary means of identifying records.	Provides an additional access path.	Determines the order of records, optimizing range queries.
Example	`CREATE UNIQUE INDEX idx_primary_key ON table(pk);`	`CREATE INDEX idx_column ON table(column);`	`CREATE CLUSTERED INDEX idx_primary_key ON table(pk);`

Case Study of any Contemporary Database

Hospital Management System

Aim

XYZ hospital is a multi specialty hospital that includes a number of departments, rooms, doctors, nurses, compounders, and other staff working in the hospital. Patients having different kinds of ailments come to the hospital and get checkup done from the concerned doctors. If required they are admitted in the hospital and discharged after treatment.

The aim of this case study is to design and develop a database for the hospital to maintain the records of various departments, rooms, and doctors in the hospital. It also maintains records of the regular patients, patients admitted in the hospital, the check up of patients done by the doctors, the patients that have been operated, and patients discharged from the hospital.

Description

In hospital, there are many departments like Orthopedic, Pathology, Emergency, Dental, Gynecology, Anesthetics, I.C.U., Blood Bank, Operation Theater, Laboratory, M.R.I., Neurology, Cardiology, Cancer Department, Corpse, etc. There is an OPD where patients come and get a card (that is, entry card of the patient) for check up from the concerned doctor. After making entry in the card, they go to the concerned doctor's room and the doctor checks up their ailments. According to the ailments, the doctor either prescribes medicine or admits the patient in the concerned department. The patient may choose either private or general room according to his/her need. But before getting admission in the hospital, the patient has to fulfill certain formalities of the hospital like room charges, etc. After the treatment is completed, the doctor discharges the patient. Before discharging from the hospital, the patient again has to complete certain formalities of the hospital like balance charges, test charges, operation charges (if any), blood charges, doctors' charges, etc.

Next we talk about the doctors of the hospital. There are two types of the doctors in the hospital, namely, regular doctors and call on doctors. Regular doctors are those doctors who come to the hospital daily. Calls on doctors are those doctors who are called by the hospital if the concerned doctor is not available.

Table Description

Following are the tables along with constraints used in *Hospital Management* database.

DEPARTMENT: This table consists of details about the various departments in the hospital. The information stored in this table includes department name, department location, and facilities available in that department.

Constraint: Department name will be unique for each department.

ALL_DOCTORS: This table stores information about all the doctors working for the hospital and the departments they are associated with. Each doctor is given an identity number starting with DR or DC prefixes only.

Constraint: Identity number is unique for each doctor and the corresponding department should exist in `DEPARTMENT` table.

DOC_REG: This table stores details of regular doctors working in the hospital. Doctors are referred to by their doctor number. This table also stores personal details of doctors like name, qualification, address, phone number, salary, date of joining, etc.

Constraint: Doctor's number entered should contain DR only as a prefix and must exist in `ALL_DOCTORS` table.

DOC_ON_CALL: This table stores details of doctors called by hospital when additional doctors are required. Doctors are referred to by their doctor number. Other personal details like name, qualification, fees per call, payment due, address, phone number, etc., are also stored.

Constraint: Doctor's number entered should contain DC only as a prefix and must exist in `ALL_DOCTORS` table.

PAT_ENTRY: The record in this table is created when any patient arrives in the hospital for a check up. When patient arrives, a patient number is generated which acts as a primary key. Other details like name, age, sex, address, city, phone number, entry date, name of the doctor referred to, diagnosis, and department name are also stored. After storing the necessary details patient is sent to the doctor for check up.

Constraint: Patient number should begin with prefix PT. Sex should be M or F only. Doctor's name and department referred must exist.

PAT_CHKUP: This table stores the details about the patients who get treatment from the doctor referred to. Details like patient number from patient entry table, doctor number, date of check up, diagnosis, and treatment are stored. One more field status is used to indicate whether patient is admitted, referred for operation or is a regular patient to the hospital. If patient is admitted, further

details are stored in `PAT_ADMIT` table. If patient is referred for operation, the further details are stored in `PAT_OPR` table and if patient is a regular patient to the hospital, the further details are stored in `PAT_REG` table.

Constraint: Patient number should exist in `PAT_ENTRY` table and it should be unique.

PAT_ADMIT: When patient is admitted, his/her related details are stored in this table. Information stored includes patient number, advance payment, mode of payment, room number, department, date of admission, initial condition, diagnosis, treatment, number of the doctor under whom treatment is done, attendant name, etc.

Constraint: Patient number should exist in `PAT_ENTRY` table. Department, doctor number, room number must be valid.

PAT_DIS: An entry is made in this table whenever a patient gets discharged from the hospital. Each entry includes details like patient number, treatment given, treatment advice, payment made, mode of payment, date of discharge, etc.

Constraint: Patient number should exist in `PAT_ENTRY` table.

PAT_REG: Details of regular patients are stored in this table. Information stored includes date of visit, diagnosis, treatment, medicine recommended, status of treatment, etc.

Constraint: Patient number should exist in patient entry table. There can be multiple entries of one patient as patient might be visiting hospital repeatedly for check up and there will be entry for patient's each visit.

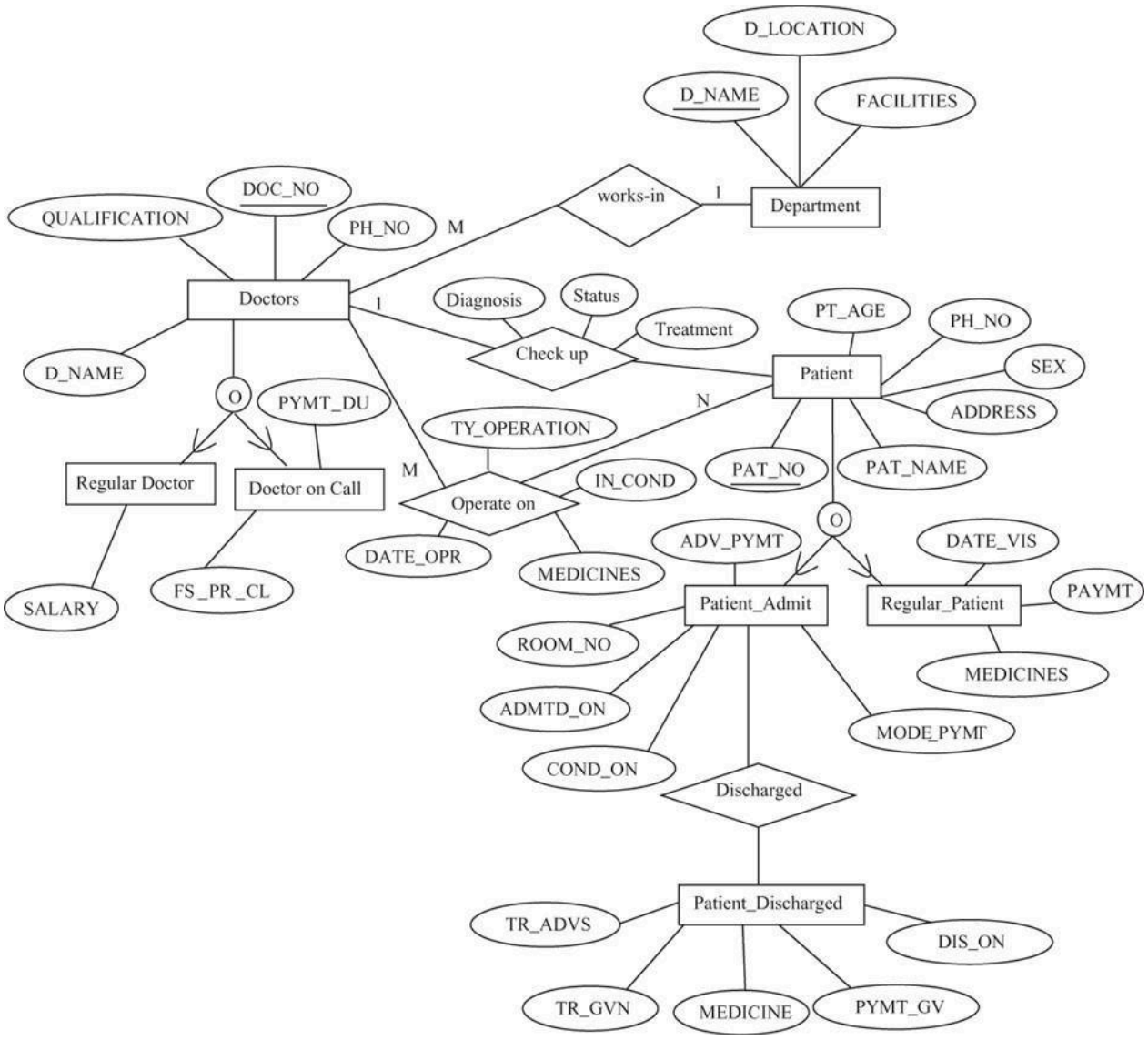
PAT_OPR: If patient is operated in the hospital, his/her details are stored in this table. Information stored includes patient number, date of admission, date of operation, number of the doctor who conducted the operation, number of the operation theater in which operation was carried out, type of operation, patient's condition before and after operation, treatment advice, etc.

Constraint: Patient number should exist in `PAT_ENTRY` table. Department, doctor number should exist or should be valid.

ROOM_DETAILS: It contains details of all rooms in the hospital. The details stored in this table include room number, room type (general or private), status (whether occupied or not), if occupied, then patient number, patient name, charges per day, etc.

Constraint: Room number should be unique. Room type can only be G or P and status can only be Y or N

ER Diagram


D_LOCATION
D_NAME
FACILITIES
Department
works-in
1
M
DOC_NO
QUALIFICATION
PH_NO
Doctors
1
Diagnosis
Status
Treatment
Check up
PT_AGE
PH_NO
SEX
ADDRESS
Patient
N
D_NAME
O
PYMT_DU
Regular Doctor
Doctor on Call
M
TY_OPERATION
Operate on
IN_COND
DATE_OPR
MEDICINES
PAT_NO
PAT_NAME
O
ADV_PYMT
DATE_VIS
PAYMT
Patient_Admit
Regular_Patient
ROOM_NO
MEDICINES
ADMTD_ON
MODE_PYMT
COND_ON
Discharged
SALARY
FS_PR_CL
Patient_Discharged
TR_ADVS
DIS_ON
TR_GVN
MEDICINE
PYMT_GV

Relational Schema

The relational database schema for *Hospital Management* database is as follows:

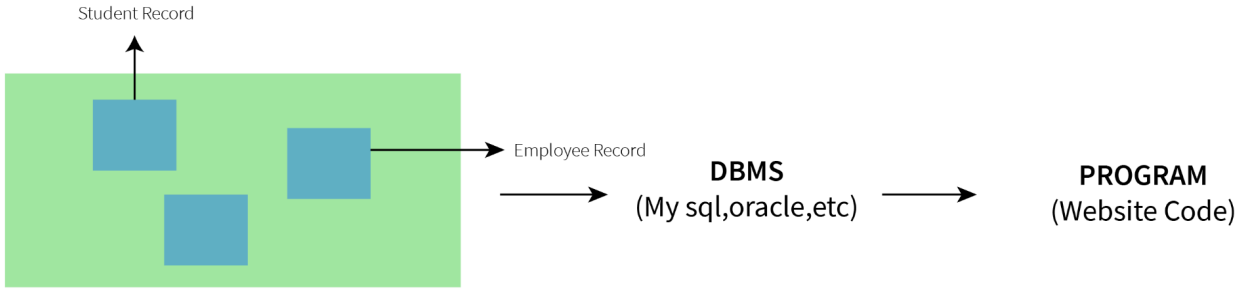
1. DEPARTMENT (D_NAME, D_LOCATION, FACILITIES)
2. ALL_DOCTORS (DOC_NO, DEPARTMENT)
3. DOC_REG(DOC_NO, D_NAME, QUALIFICATION, SALARY, EN_TIME, EX_TIME, ADDRESS, PH_NO, DOJ)
4. DOC_ON_CALL (DOC_NO, D_NAME, QUALIFICATION, FS_PR_CL, PYMT_DU, ADDRESS, PH_NO)
5. PAT_ENTRY (PAT_NO, PAT_NAME, CHKUP_DT, PT_AGE, SEX, RFRG_CSTNT, DIAGNOSIS, RFD, ADDRESS, CITY, PH_NO, DEPARTMENT)
6. PAT_CHKUP (PAT_NO, DOC_NO, DIAGNOSIS, STATUS, TREATMENT)
7. PAT_ADMIT (PAT_NO, ADV_PYMT, MODE_PYMT, ROOM_NO, DEPTNAME, ADMTD_ON, COND_ON, INVSTGTN_DN, TRMT_SDT, ATTDNT_NM)
8. PAT_DIS (PAT_NO, TR_ADVS, TR_GVN, MEDICINES, PYMT_GV, DIS_ON)
9. PAT_REG (PAT_NO, DATE_VIS, CONDITION, TREATMENT, MEDICINES, DOC_NO, PAYMT)
10. PAT_OPR (PAT_NO, DATE_OPR, IN_COND, AFOP_COND, TY_OPERATION, MEDICINES, DOC_NO, OPTH_NO, OTHER_SUG)
11. ROOM_DETAILS (ROOM_NO, TYPE, STATUS, RM_DL_CRG, OTHER_CRG)

DBMS Interview Questions

Basic Interview Questions

What is DBMS and what is its utility? Explain RDBMS with examples.
DBMS stands for Database Management System, is a set of applications or programs that enable users to create and maintain a database. DBMS provides a tool or an interface for performing various operations such as inserting, deleting, updating, etc. into a database. It is software that enables the storage of data more compactly and securely as compared to a file-based system. A DBMS system helps a user to overcome problems like data inconsistency, data redundancy, etc. in a database and makes it more convenient and organized to use it. Check this DBMS Tutorial by Scaler Topics.

Examples of popular DBMS systems are file systems, XML, Windows Registry, etc.



RDBMS stands for Relational Database Management System and was introduced in the 1970s to access and store data more efficiently than DBMS. RDBMS stores data in the form of tables as compared to DBMS which stores data as files. Storing data as rows and columns makes it easier to locate specific values in the database and makes it more efficient as compared to DBMS.

Examples of popular RDBMS systems are MySQL, Oracle DB, etc.

What is a Database?

A Database is an organized, consistent, and logical collection of data that can easily be updated, accessed, and managed. Database mostly contains sets of tables or objects (anything created using create command is a database object) which consist of records and fields. A tuple or a row represents a single entry in a table. An attribute or a column represents the basic units of data storage, which contain information about a particular aspect of the table. DBMS extracts data from a database in the form of queries given by the user.

Mention the issues with traditional file-based systems that make DBMS a better choice?

The absence of indexing in a traditional file-based system leaves us with the only option of scanning the full page and hence making the access of content tedious and super slow. The other issue is redundancy and inconsistency as files have many duplicate and redundant data and changing one of them makes all of them inconsistent. Accessing data is harder in traditional file-based systems because data is unorganized in them.

Another issue is the lack of concurrency control, which leads to one operation locking the entire page, as compared to DBMS where multiple operations can work on a single file simultaneously.

Integrity check, data isolation, atomicity, security, etc. are some other issues with traditional file-based systems for which DBMSs have provided some good solutions.

Explain a few advantages of a DBMS.

Following are the few advantages of using a DBMS.

1. **Data Sharing**: Data from a single database can be simultaneously shared by multiple users. Such sharing also enables end-users to react to changes quickly in the database environment.
2. **Integrity constraints**: The existence of such constraints allows storing of data in an organized and refined manner.
3. **Controlling redundancy in a database**: Eliminates redundancy in a database by providing a mechanism that integrates all the data in a single database.
4. **Data Independence**: This allows changing the data structure without altering the composition of any of the executing application programs.
5. **Provides backup and recovery facility**: It can be configured to automatically create the backup of the data and restore the data in the database whenever required.
6. **Data Security**: DBMS provides the necessary tools to make the storage and transfer of data more reliable and secure. Authentication (the process of giving restricted access to a user) and encryption (encrypting sensitive data such as OTP, credit card information, etc.) are some popular tools used to secure data in a DBMS.

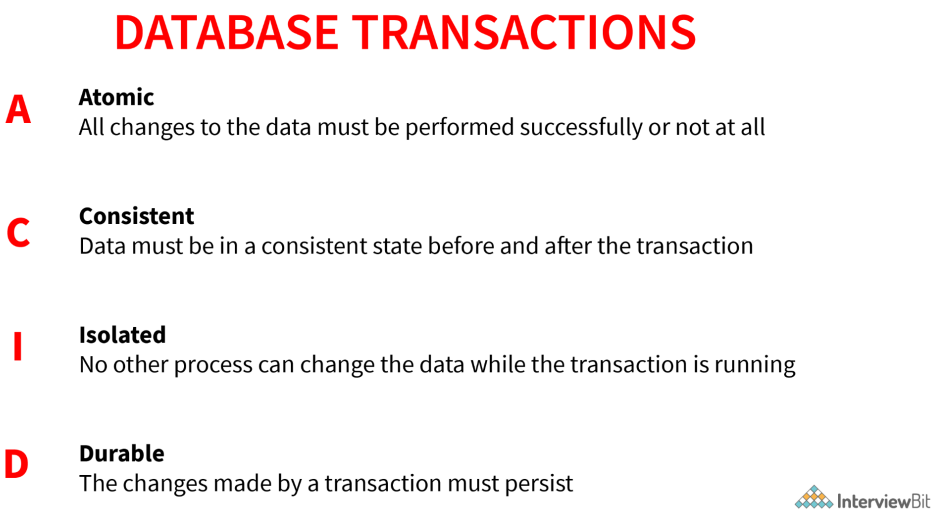
Explain different languages present in DBMS.

Following are various languages present in DBMS:

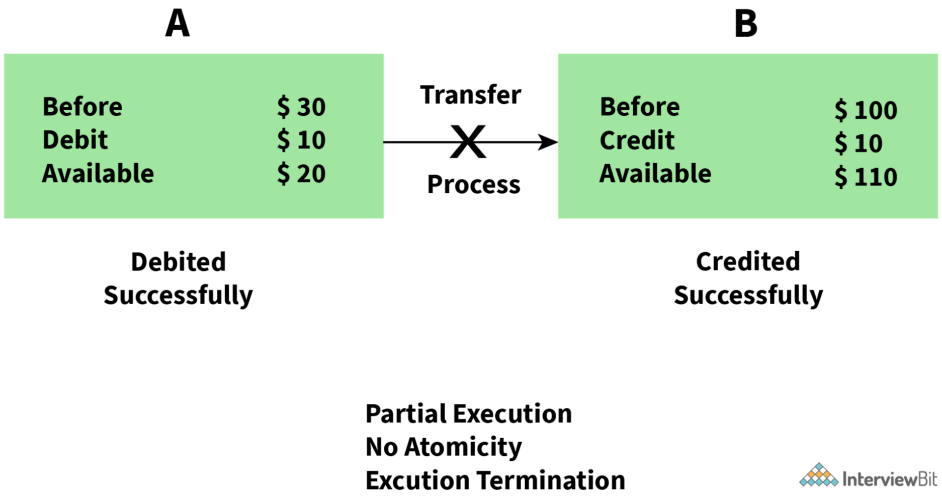
1. **DDL(Data Definition Language):** It contains commands which are required to define the database. E.g., CREATE, ALTER, DROP, TRUNCATE, RENAME, etc.
2. **DML(Data Manipulation Language):** It contains commands which are required to manipulate the data present in the database. E.g., SELECT, UPDATE, INSERT, DELETE, etc.
3. **DCL(Data Control Language):** It contains commands which are required to deal with the user permissions and controls of the database system. E.g., GRANT and REVOKE.
4. **TCL(Transaction Control Language):** It contains commands which are required to deal with the transaction of the database. E.g., COMMIT, ROLLBACK, and SAVEPOINT.

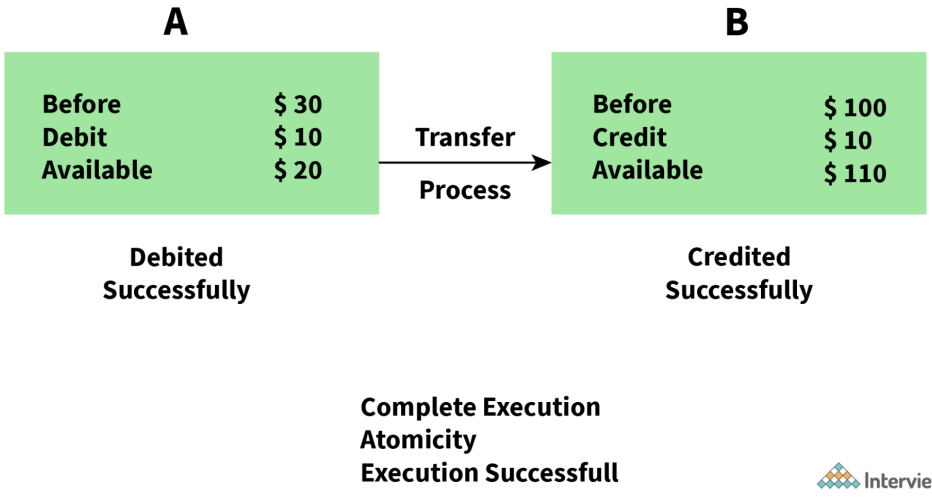
What is meant by ACID properties in DBMS?

ACID stands for Atomicity, Consistency, Isolation, and Durability in a DBMS these are those properties that ensure a safe and secure way of sharing data among multiple users.

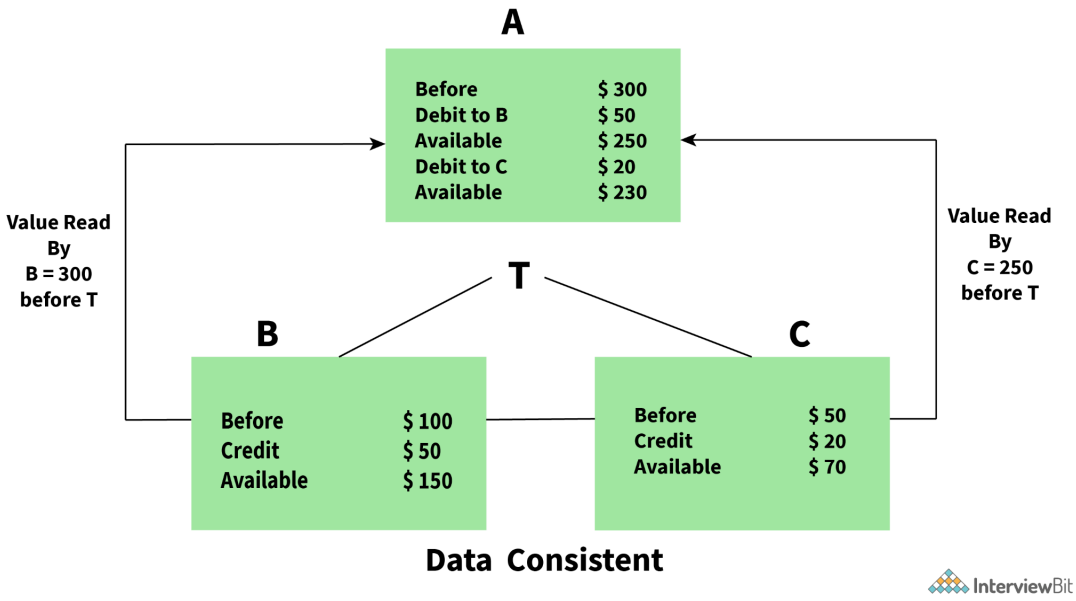


1. **Atomicity:** This property reflects the concept of either executing the whole query or executing nothing at all, which implies that if an update occurs in a database then that update should either be reflected in the whole database or should not be reflected at all.

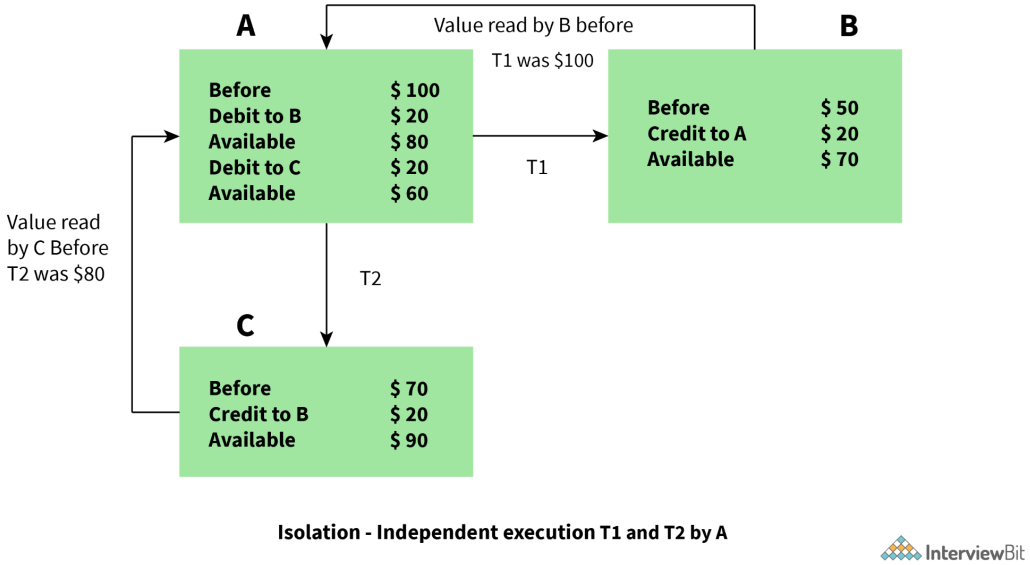




2. **Consistency:** This property ensures that the data remains consistent before and after a transaction in a database.



3. **Isolation:** This property ensures that each transaction is occurring independently of the others. This implies that the state of an ongoing transaction doesn't affect the state of another ongoing transaction.



4. **Durability:** This property ensures that the data is not lost in cases of a system failure or restart and is present in the same state as it was before the system failure or restart.

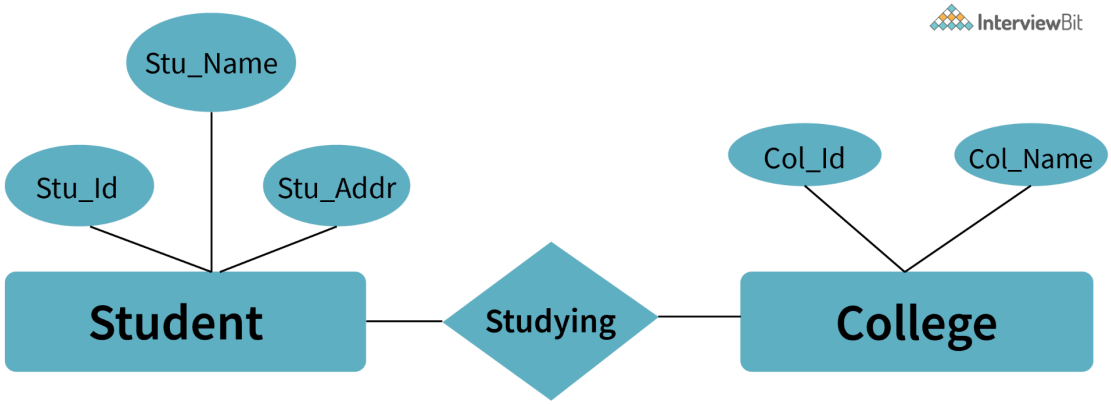
Are NULL values in a database the same as that of blank space or zero?
No, a NULL value is very different from that of zero and blank space as it represents a value that is assigned, unknown, unavailable, or not applicable as compared to blank space which represents a character and zero represents a number.

Example: NULL value in “number_of_courses” taken by a student represents that its value is unknown whereas 0 in it means that the student hasn’t taken any courses.

Intermediate DBMS Interview Questions

What is meant by an entity-relationship (E-R) model? Explain the terms Entity, Entity Type, and Entity Set in DBMS?

An entity-relationship model is a diagrammatic approach to a database design where real-world objects are represented as entities and relationships between them are mentioned.



Sample E-R Diagram

1. Entity: An entity is defined as a real-world object having attributes that represent characteristics of that particular object. For example, a student, an employee, or a teacher represents an entity.
2. Entity Type: An entity type is defined as a collection of entities that have the same attributes. One or more related tables in a database represent an entity type. Entity type or attributes can be understood as a characteristic which uniquely identifies the entity. For example, a student represents an entity that has attributes such as student_id, student_name, etc.
3. Entity Set: An entity set can be defined as a set of all the entities present in a specific entity type in a database. For example, a set of all the students, employees, teachers, etc. represent an entity set.

What is meant by normalization and denormalization?

Normalization is a process of reducing redundancy by organizing the data into multiple tables. Normalization leads to better usage of disk spaces and makes it easier to maintain the integrity of the database.

Denormalization is the reverse process of normalization as it combines the tables which have been normalized into a single table so that data retrieval becomes faster. JOIN operation allows us to create a denormalized form of the data by reversing the normalization.

What is a lock. Explain the major difference between a shared lock and an exclusive lock during a transaction in a database?

A database lock is a mechanism to protect a shared piece of data from getting updated by two or more database users at the same time. When a single database user or session has acquired a lock then no other database user or session can modify that data until the lock is released.

1. **Shared Lock:** A shared lock is required for reading a data item and many transactions may hold a lock on the same data item in a shared lock. Multiple transactions are allowed to read the data items in a shared lock.
2. **Exclusive lock:** An exclusive lock is a lock on any transaction that is about to perform a write operation. This type of lock doesn't allow more than one transaction and hence prevents any inconsistency in the database.

Explain the difference between the DELETE and TRUNCATE command in a DBMS.

DELETE command: this command is needed to delete rows from a table based on the condition provided by the WHERE clause.

1. It deletes only the rows which are specified by the WHERE clause.
2. It can be rolled back if required.
3. It maintains a log to lock the row of the table before deleting it and hence it's slow.

TRUNCATE command: this command is needed to remove complete data from a table in a database. It is like a DELETE command which has no WHERE clause.

It removes complete data from a table in a database.

1. It can't be rolled back even if required. (truncate can be rolled back in some databases depending on their version but it can be tricky and can lead to data loss).
2. It doesn't maintain a log and deletes the whole table at once and hence it's fast.

Explain the difference between intension and extension in a database.

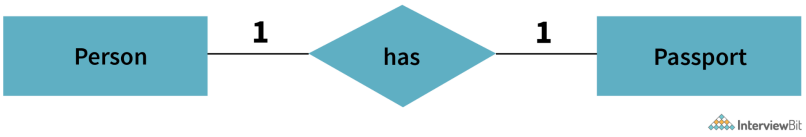
Following is the major difference between intension and extension in a database

1. **Intension:** Intension or popularly known as database schema is used to define the description of the database and is specified during the design of the database and mostly remains unchanged.
2. **Extension:** Extension on the other hand is the measure of the number of tuples present in the database at any given point in time. The extension of a database is also referred to as the snapshot of the database and its value keeps changing as and when the tuples are created, updated, or destroyed in a database.

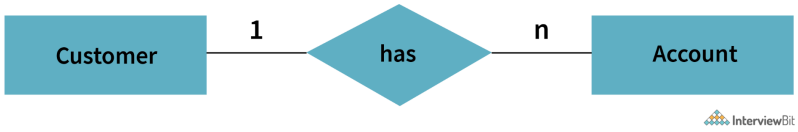
Explain different types of relationships amongst tables in a DBMS.

Following are different types of relationship amongst tables in a DBMS system:

1. **One to One Relationship:** This type of relationship is applied when a particular row in table X is linked to a singular row in table Y.

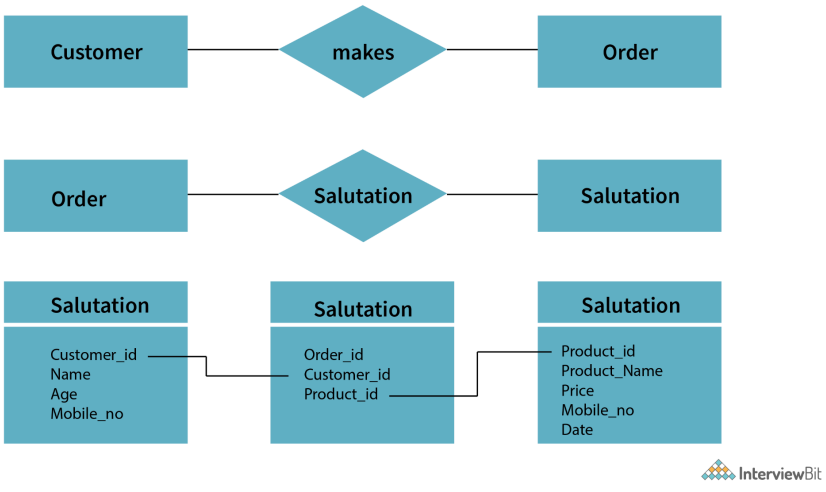


2. **One to Many Relationship:** This type of relationship is applied when a single row in table X is related to many rows in table Y.



3. **Many to Many Relationship:** This type of relationship is applied when multiple rows in table X can be linked to multiple rows in table Y.

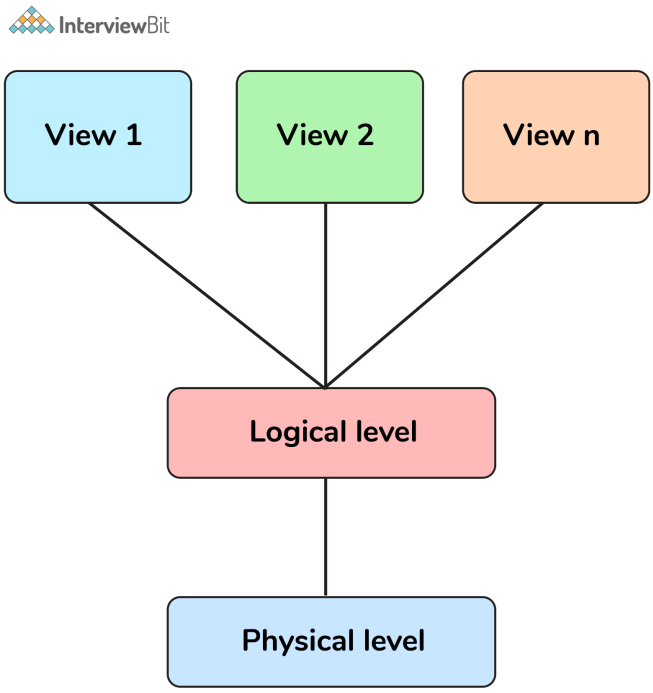
4. **Self Referencing Relationship:** This type of relationship is applied when a particular row in table X is associated with the same table.



Explain different levels of data abstraction in a DBMS.

The process of hiding irrelevant details from users is known as data abstraction. Data abstraction can be divided into 3 levels,

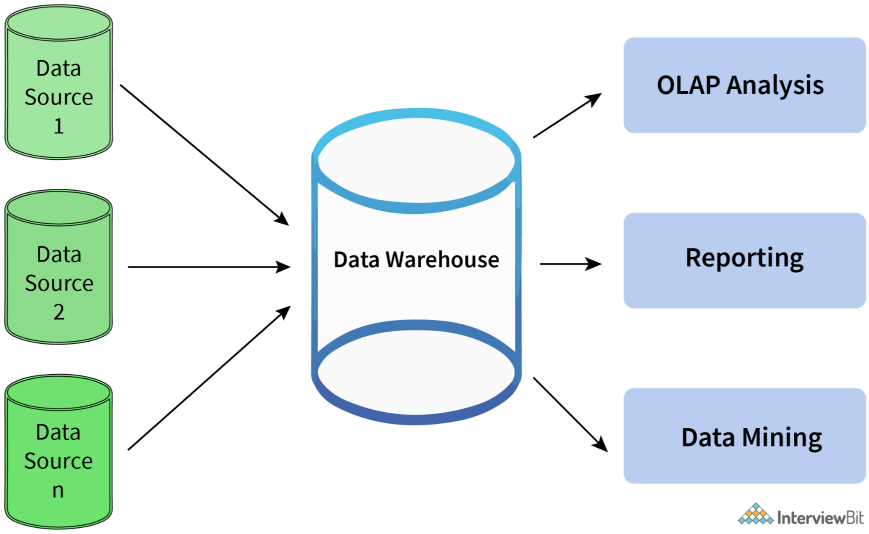
1. **Physical Level:** it is the lowest level and is managed by DBMS. This level consists of data storage descriptions and the details of this level are typically hidden from system admins, developers, and users.
2. **Conceptual or Logical level:** it is the level on which developers and system admins work and it determines what data is stored in the database and what is the relationship between the data points.
3. **External or View level:** it is the level that describes only part of the database and hides the details of the table schema and its physical storage from the users. The result of a query is an example of View level data abstraction. A view is a virtual table created by selecting fields from one or more tables present in the database.



Three levels of data abstraction

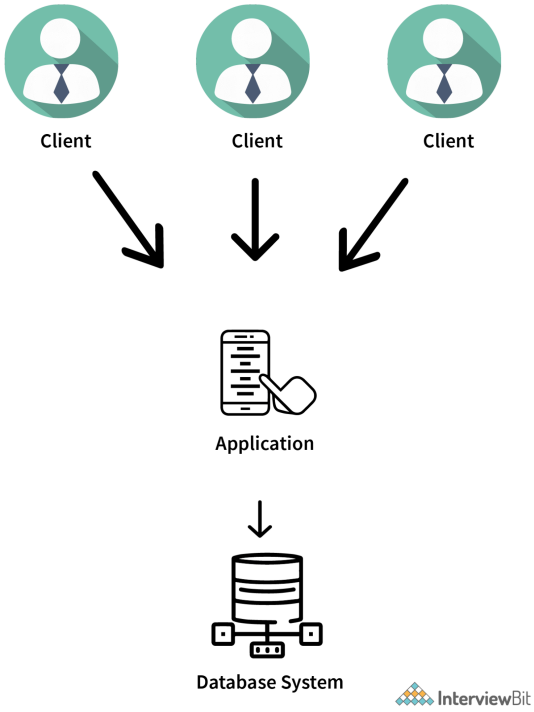
What is Data Warehousing?

The process of collecting, extracting, transforming, and loading data from multiple sources and storing them in one database is known as data warehousing. A data warehouse can be considered as a central repository where data flows from transactional systems and other relational databases and is used for data analytics. A data warehouse comprises a wide variety of an organization's historical data that supports the decision-making process in an organization.



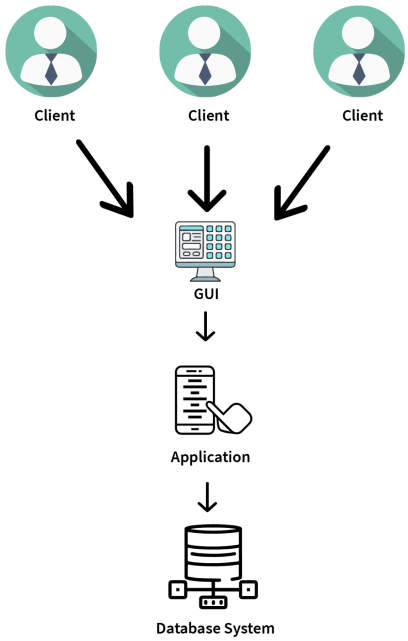
Advanced DBMS Interview Questions

Explain the difference between a 2-tier and 3-tier architecture in a DBMS.
The 2-tier architecture refers to the client-server architecture in which applications at the client end directly communicate with the database at the server end without any middleware involved.
Example – Contact Management System created using MS-Access or Railway Reservation System, etc.



The 3-tier architecture contains another layer between the client and the server to provide GUI to the users and make the system much more secure and accessible. In this type of architecture, the application present on the client end interacts with an application on the server end which further communicates with the database system.

Example – Designing registration form which contains a text box, label, button or a large website on the Internet, etc.

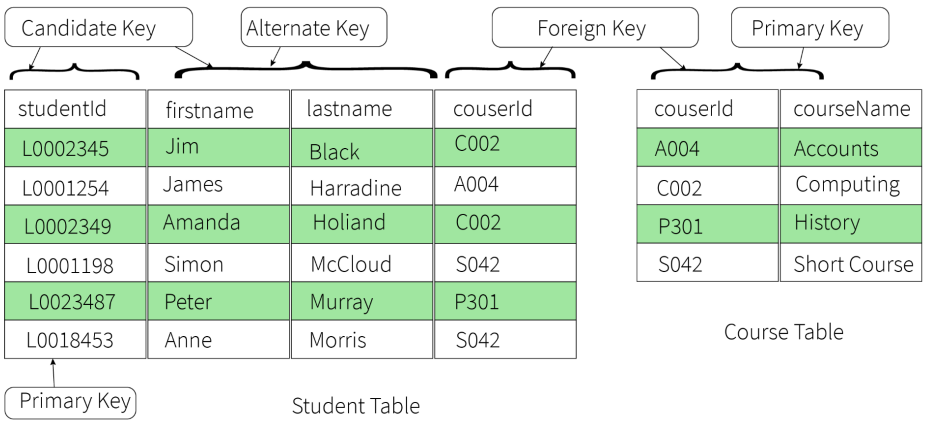


InterviewBit

Explain different types of keys in a database.

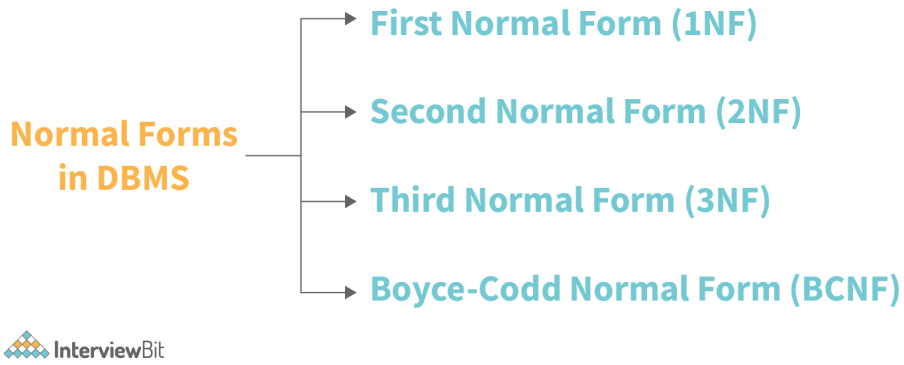
There are mainly 7 types of keys in a database,

1. **Candidate Key:** The candidate key represents a set of properties that can uniquely identify a table. Each table may have multiple candidate keys. One key amongst all candidate keys can be chosen as a primary key. In the below example since studentId and firstName can be considered as a Candidate Key since they can uniquely identify every tuple.
2. **Super Key:** The super key defines a set of attributes that can uniquely identify a tuple. Candidate key and primary key are subsets of the super key, in other words, the super key is their superset.
3. **Primary Key:** The primary key defines a set of attributes that are used to uniquely identify every tuple. In the below example studentId and firstName are candidate keys and any one of them can be chosen as a Primary Key. In the given example studentId is chosen as the primary key for the student table.
4. **Unique Key:** The unique key is very similar to the primary key except that primary keys don't allow NULL values in the column but unique keys allow them. So essentially unique keys are primary keys with NULL values.
5. **Alternate Key:** All the candidate keys which are not chosen as primary keys are considered as alternate Keys. In the below example, firstname and lastname are alternate keys in the database.
6. **Foreign Key:** The foreign key defines an attribute that can only take the values present in one table common to the attribute present in another table. In the below example courseId from the Student table is a foreign key to the Course table, as both, the tables contain courseId as one of their attributes.
7. **Composite Key:** A composite key refers to a combination of two or more columns that can uniquely identify each tuple in a table. In the below example the studentId and firstname can be grouped to uniquely identify every tuple in the table.



Relationship Between Keys

Explain different types of Normalization forms in a DBMS.
Following are the major normalization forms in a DBMS:



Full Names	Physical Address	Movies Rented	Salutation
Janet Jones	First street Plot No 4	Pirates. the Caribbean Clash of the Titans	Ms.
Robert Phil	3rd street 34	Forgetting Sarah Marshal Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

InterviewBit

Considering the above Table-1 as the reference example for understanding different normalization forms.

1NF: It is known as the first normal form and is the simplest type of normalization that you can implement in a database. A table to be in its first normal form should satisfy the following conditions:

1. Every column must have a single value and should be atomic.
2. Duplicate columns from the same table should be removed.
3. Separate tables should be created for each group of related data and each row should be identified with a unique column.

Full Names	Physical Address	Movies Rented	Salutation
Janet Jones	First street Plot No 4	Pirates. the Caribbean	Ms.
Janet Jones	First street Plot No 4	Clash of the Titans	Ms.
Robert Phil	3rd street 34	Forgetting Sarah Marshal	Mr.
Robert Phil	3rd street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

2NF: It is known as the second normal form. A table to be in its second normal form should satisfy the following conditions:

1. The table should be in its 1NF i.e. satisfy all the conditions of 1NF.
2. Every non-prime attribute of the table should be fully functionally dependent on the primary key i.e. every non-key attribute should be dependent on the primary key in such a way that if any key element is deleted then even the non_key element will be saved in the database.

Membership ID	Full Names	Physical Address	Salutation
1	Janet Jones	First street Plot No 4	Ms.
2	Robert Phil	3rd street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

Membership ID	Movies Rented
1	Pirates. the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

3NF: It is known as the third normal form. A table to be in its third normal form should satisfy the following conditions:

1. The table should be in its 2NF i.e. satisfy all the conditions of 2NF.
2. There is no transitive functional dependency of one attribute on any attribute in the same table.

Membership ID	Full Names	Physical Address	SalutationID
1	Janet Jones	First street Plot No 4	2
2	Robert Phil	3rd street 34	1
3	Robert Phil	5th Avenue	1

Membership ID	Movies Rented
1	Pirates. the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Salutation ID	Salutation
1	Mr.
2	Ms.
3	Mrs.
4	Dr.

BCNF: BCNF stands for Boyce-Codd Normal Form and is an advanced form of 3NF. It is also referred to as 3.5NF for the same reason. A table to be in its BCNF normal form should satisfy the following conditions:

1. The table should be in its 3NF i.e. satisfy all the conditions of 3NF.
2. For every functional dependency of any attribute A on B
3. (A->B), A should be the super key of the table. It simply implies that A can't be a non-prime attribute if B is a prime attribute.