MODULE 1

Introductory Concepts of Databases and Information Systems:

1.1 INTRODUCTION

In the present time, database systems are used to electronically store all the data of an organization. Databases allow data sharing and integrate data of an organization. In this unit, we are going to be introduced to the basic concepts of database systems. This is an introductory unit where we are going to learn about the definition of a database, its characteristics, classification, functions and the user types which are involved in using the database.

1.2 OBJECTIVES

Upon completion of this unit you should be able to:

· · Describe what a database is and how it functions. Outline various properties of a database.

9 Database System-Basic Concepts and Models

· · Explain the roles of different database end users. Compare and contrast the different databases based on their classification.

1.3 DATABASES: DEFINITION AND CHARACTERISTICS

database introduction and concepts:

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS).

concepts of database management system:

A DBMS serves as an interface between an end-user and a database, allowing users to create, read, update, and delete data in the database. DBMS manage the data, the database engine, and the database schema, allowing for data to be manipulated or extracted by users and other programs.

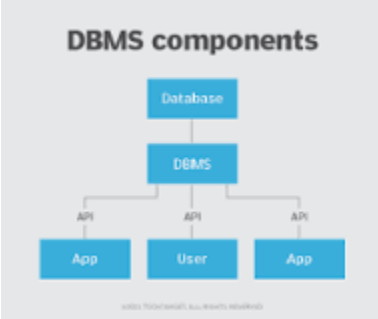
concepts of database and database design:

Database design concepts. A database is better defined as a set of entities and relations as shown in the next figure. Entities are objects with a physical (car) or conceptual (culture) existence. Database design is the process to define and represent entities and relations.

important concepts of database:

Then, there are a bunch of important database concepts for a software engineer to know: normalization, denormalization, SQL, No-SQL, ERDs, query optimization, etc. The list goes on! In short, software engineering is for those who can do a little bit of everything while paying a lot of attention and care to each task.

concept of database with example:



Databases often store information about people, such as customers or users. For example, social media platforms use databases to store user information, such as names, email addresses and user behavior. The data is used to recommend content to users and improve the user experience. Secure personal health information

database concept and structure:



A database is an organized collection of data. Instead of having all the data in a list with a random order, a database provides a structure to organize the data. One of the most common data structures is a database table. A database table consists of rows and columns.

concept of database design:

Database design is a collection of steps that help create, implement, and maintain a business's data management systems. The primary purpose of designing a database is to produce physical and logical models of designs for the proposed database system.

Modern information management rely heavily on database systems because they make it possible to store, retrieve, and manipulate massive volumes of data effectively. Designing reliable and scalable databases requires a thorough understanding of the principles and architecture of database systems. The essential ideas and complexities of database systems will be covered in detail in this article, along with examples from everyday life to show how they might be used in real-world situations.

Concepts of Database Systems

- **Data** − Data is the central component of every database system. The information that has to be handled and saved is represented by data. It could be structured, somewhat structured, or not at all. Structured data is arranged into tables with rows and columns

according to a predetermined pattern. Examples include financial data, product specifications, and client information. Data that is semi-structured, like JSON or XML, has some structure but does not follow a strict standard. Text documents, photos, and multimedia files are examples of unstructured data since they don't have a predetermined structure.

- **Database Management System (DBMS)**− Software that makes it easier to create, organize, and manipulate databases is known as a database management system (DBMS). It offers a selection of tools and user interfaces for effective data management. Data storage, data retrieval, data manipulation, data security, and concurrency control are among a DBMS's essential features. Popular DBMS s include PostgreSQL, Oracle, MySQL, and Microsoft SQL Server.
- **Database** − A database is a structured collection of data that is maintained and organized by a database management system (DBMS). It is made up of one or more tables, each of which represents a different entity or idea. Each row in a table represents one instance of the entity, and each column in a table represents a particular quality or trait. Rows and columns make up a table.
- **Schema** − A database schema outlines the logical organization and structure of a database. The tables, connections between tables, restrictions, and other information are all described. A schema outlines the structure and storage requirements for the data. Let's look at an illustration.
  Assume we have an online storefront with the following tables −
  Customers (Customer Id, Name, Email)
  Orders (Order Id, Customer Id, Order Date, Total Amount)
  Products (Product Id, Name, Price)
  The tables, their columns, and any connections or restrictions between them would all be specified by the schema.
- **Query Language** − : A query language enables users to access, manage, and alter data from databases by sending queries. The most used query language for relational databases is Structured Query Language (SQL). For building, editing, and querying databases, it offers a set of commands and syntax.

Take the following SQL query, for instance −

```
SELECT Customers.Name, Orders.OrderDate, Orders.TotalAmount
FROM Customers
JOIN Orders ON Customers.CustomerId = Orders.CustomerId
WHERE Customers.Country = 'USA'
```

Input Table- Customers

```
CustomerId | Name          | Country
-----------------------------------
1          | John Doe      | USA
2          | Jane Smith    | USA
3          | Mark Johnson  | Canada
4          | Sarah Wilson  | USA
5          | Robert Brown  | USA
6          | Lisa Thompson | Canada
7          | James Lee     | USA
8          | Emily Davis   | USA
9          | Michael Clark | Canada
10         | Emma Harris   | USA
```

Input Table- Orders

```
OrderId | CustomerId | OrderDate  | TotalAmount
-----------------------------------------------
1       | 1          | 2023-05-01 | $100.00
2       | 2          | 2023-05-10 | $250.00
3       | 2          | 2023-05-15 | $180.00
4       | 3          | 2023-05-20 | $300.00
5       | 4          | 2023-05-05 | $150.00
6       | 4          | 2023-05-12 | $220.00
7       | 5          | 2023-05-03 | $180.00
8       | 5          | 2023-05-18 | $280.00
9       | 7          | 2023-05-07 | $120.00
10      | 8          | 2023-05-09 | $200.00
11      | 8          | 2023-05-22 | $350.00
12      | 10         | 2023-05-14 | $190.00
```

Output Table

```
Customers.Name | Orders.OrderDate | Orders.TotalAmount
-------------------------------------------------------
John Doe       | 2023-05-01       | $100.00
Jane Smith     | 2023-05-10       | $250.00
Jane Smith     | 2023-05-15       | $180.00
Sarah Wilson   | 2023-05-05       | $150.00
Sarah Wilson   | 2023-05-12       | $220.00
Robert Brown   | 2023-05-03       | $180.00
Robert Brown   | 2023-05-18       | $280.00
```

```
James Lee      | 2023-05-07       | $120.00
Emily Davis    | 2023-05-09       | $200.00
Emily Davis    | 2023-05-22       | $350.00
Emma Harris    | 2023-05-14       | $190.00
```

For all orders placed by clients in the USA, this query returns the customer name, order date, and order total.

For all orders placed by clients in the USA, this query returns the customer name, order date, and order total.

Database System Architecture

The general structure and parts of a database system are described by the database system architecture. It includes the following essential elements −

- **User Interface** − Users can communicate with the database system using the user interface. It could take the form of a web-based interface, a GUI, or a command-line interface. Users may submit queries, enter data, and see query results or reports via the user interface.
  A web-based e-commerce program, for instance, may offer a user interface that enables users to look for items, make orders, and check their order histories.
- **Query Processor** − The query processor executes and optimizes SQL queries after receiving them from users or applications. In order to get the required data and carry out any necessary activities, it analyses the query, chooses the most effective execution plan and communicates with other components. In order to reduce resource consumption and boost speed, the query processor makes sure that queries are processed as effectively as possible.

Take the prior SQL query, for instance −

```
SELECT Customers.Name, Orders.OrderDate, Orders.TotalAmount
FROM Customers
JOIN Orders ON Customers.CustomerId = Orders.CustomerId
WHERE Customers.Country = 'USA'
```

Input Table- Customers

```
CustomerId | Name          | Country
-----------------------------------
1          | Adam Johnson  | USA
2          | Emma Thompson | UK
```

```
3         | Sophia Lee    | Canada
4         | Oliver Smith  | Australia
5         | Mia Davis     | USA
6         | Ethan Wilson  | UK
7         | Ava Brown     | Canada
8         | Noah Taylor   | Australia
9         | Isabella Chen | USA
10         | Liam Hall     | UK
```

Input Table- Orders

```
OrderId | CustomerId | OrderDate  | TotalAmount
-------------------------------------------
1       | 1          | 2023-06-01 | $150.00
2       | 2          | 2023-06-05 | $200.00
3       | 3          | 2023-06-10 | $120.00
4       | 4          | 2023-06-15 | $250.00
5       | 5          | 2023-06-20 | $180.00
6       | 6          | 2023-06-25 | $300.00
7       | 7          | 2023-06-02 | $210.00
8       | 8          | 2023-06-07 | $160.00
9       | 9          | 2023-06-12 | $190.00
10      | 10         | 2023-06-18 | $230.00
```

Output Table

```
Customers.Name  | Orders.OrderDate | Orders.TotalAmount
---------------------------------------------------
Adam Johnson    | 2023-06-01       | $150.00
Emma Thompson   | 2023-06-05       | $200.00
Sophia Lee      | 2023-06-10       | $120.00
Oliver Smith    | 2023-06-15       | $250.00
Mia Davis       | 2023-06-20       | $180.00
Ethan Wilson    | 2023-06-25       | $300.00
Ava Brown       | 2023-06-02       | $210.00
Noah Taylor     | 2023-06-07       | $160.00
Isabella Chen   | 2023-06-12       | $190.00
Liam Hall       | 2023-06-18       | $230.00
```

The "Customers" and "Orders" tables' necessary data is efficiently retrieved by the query processor, which also analyses the query and chooses the best join technique.

- **Storage Manager** − Managing the actual physical storage of data on discs or other storage media is the responsibility of the storage manager. To read and write data, it communicates with the file system or storage subsystem. To facilitate data access and guarantee data integrity, the storage manager manages data archiving, retrieval, and indexin
  For instance, the storage manager oversees the allocation of disc space to guarantee effective storage when a new order is placed in the e-commerce application. It also saves the order details in the relevant tables.
- **Buffer Manager** − Data transfer between memory and disc storage is controlled by the buffer manager, an important component. It reduces disc I/O operations and boosts efficiency by using a buffer cache to keep frequently used data pages in memory. The buffer manager makes sure that data caching and replacement procedures are effective in order to maximize memory consumption.
  For instance, when a query is run that needs to access data from the disc, the buffer manager pulls the necessary data pages into the buffer cache from the disc. The need for disc access can be avoided by serving subsequent requests that access the same data from memory.
- **Transactions Manager** − Database transactions' atomicity, consistency, isolation, and durability are all guaranteed by the transaction manager. To maintain data integrity and concurrency management, it maintains concurrent access to the data, takes care of transaction execution, and enforces transaction isolation levels.
  For instance, the transaction manager makes sure that each order is executed as a separate transaction when several clients place orders at once, ensuring data integrity and avoiding conflicts.
- **Data Dictionary** − The metadata regarding the database schema and objects are stored in the data dictionary, sometimes referred to as the metadata repository. It includes details on various database structures, including tables, columns, data types, constraints, indexes, and more. The DBMS uses the data dictionary to verify queries, uphold data integrity, and offer details on the database structure
  For instance, the data dictionary keeps tabs on the names, columns, data types, and constraints of the tables in the e-commerce application.
- **Concurrency Control** − Multiple transactions can access and edit the database simultaneously without resulting in inconsistent data thanks to concurrency control methods. To regulate concurrent access and preserve data integrity, methods including locking, timestamp ordering, and multi-version concurrency control (MVCC) are utilized. Concurrency control measures, for instance, make sure that two consumers updating their profiles in the same e-commerce application at the same time are serialized and applied appropriately to maintain data consistency.
- **Backup and recovery** − In order to safeguard against data loss and guarantee data availability, database systems must have backup and recovery processes. In the case of

system failures or data corruption, recovery procedures are employed to restore the database to a consistent condition. Regular backups are performed to create copies of the database.
To guarantee that data can be restored in the event of hardware problems or unintentional data loss, for instance, frequent backups of the e-commerce database are made.

Conclusion

In conclusion, building, implementing, and maintaining reliable and scalable databases requires an understanding of the principles and architecture of database systems. We looked at the essential ideas of data, DBMS, database, schema, and query language in this post. The architecture of database systems was also covered in detail, with topics covered including the user interface, query processor, storage manager, buffer manager, transaction manager, data dictionary, concurrency management, and backup and recovery procedures.

MODULE 2:

Semantic Database Design:

Data is essentially facts or numbers stored electronically. However, to extract valuable insights that drive business decisions, data must go through processes like collection, storage, transformation, and processing. Different use cases involve diverse datasets, and comprehending the interconnected relationships between these datasets allows organizations to leverage their data more effectively. This is where semantic data modeling emerges as a powerful solution, empowering organizations to unlock the true value of their data assets.

Semantic Data Modeling:

Semantic data modeling is a paradigm that focuses on capturing the meaning and context of data, rather than just its structure. By leveraging ontologies and formal knowledge representation

techniques, semantic models can precisely define concepts, relationships, and rules within a given domain. This approach not only facilitates data integration and interoperability but also enables advanced reasoning and inference capabilities.

In this context, understanding "what is data management" becomes pivotal, as it underscores the significance of semantic data modeling in efficiently organizing, interpreting, and extracting actionable insights from diverse datasets.

How Do Semantic Data Models Work?

Semantic data models (SDMs) combine semantic elements with graphical visualization, enhancing the value proposition of various data modeling approaches. The process of analyzing input data necessitates an abstraction process, wherein specific qualities and aspects of reality are selected while irrelevant ones are disregarded, aligning with the requirements of the specific solution, project, model, or schema.

An SDM leverages three distinct types of abstraction:

- Classification: This abstraction technique categorizes different objects in objective reality using "instance of" relations. It involves grouping objects based on shared characteristics, such as creating a group of employees.
- Aggregation: Aggregation defines a new object by combining a set of component objects, using "has a" relations. For example, an employer entity can be an aggregation of attributes like name, age, or contact information.
- Generalization: Generalization establishes a subset relationship between occurrences of two or more objects using "is a" relations. For instance, an employer can be a generalization of the concept of managers.

By employing these three abstraction methods, SDMs provide a robust framework for representing and comprehending the intricate relationships and semantics within complex data landscapes.

Benefits of Semantic Data Modeling

The semantic model helps in overseeing the company's comprehensive data management services, thereby enhancing decision-making abilities. Its benefits include:

Enhanced Data Integration

Semantic models provide a common, shared understanding of data across different systems, applications, and domains. This semantic interoperability enables seamless data integration, reducing the need for complex mappings and transformations.

Improved Data Quality

By explicitly defining data semantics, semantic models help ensure data consistency, accuracy, and completeness. They provide a framework for validating data against domain-specific rules and constraints, minimizing errors and redundancies.

Richer Data Insights

Semantic models capture the contextual meaning of data, allowing for more sophisticated data analysis and interpretation. This enables organizations to derive deeper insights and uncover hidden relationships within their data assets.

Increased Flexibility and Adaptability

Semantic models are designed to be extensible and adaptable, making it easier to incorporate changes and evolve data structures as business requirements evolve. This flexibility ensures that data models remain relevant and aligned with organizational needs.

Facilitated Knowledge Sharing and Reuse

By formalizing domain knowledge in a machine-readable format, semantic models foster knowledge sharing and reuse across teams, departments, and even organizations. This collaborative approach accelerates development cycles and promotes consistency.

Explore "5 Key Reasons Why Businesses Need Data Management Platforms" for deeper insights into the significance of semantic data modeling in optimizing business operations.

Semantic Data Modeling Approaches

Semantic data modeling involves representing data and its relationships in a way that captures the underlying meaning or semantics of the information. There are several approaches to semantic data modeling, each with its strengths and use cases. Here are some common approaches:

- One approach is the **Ontology Model**, which focuses on identifying and describing business data entities, while also establishing the existing relationships among these data elements. Ontologies provide a formal framework for defining the concepts, properties, and interrelationships within a specific domain.
- Another approach is the **Knowledge Graph data model**, which offers a visual representation of real-world entities and their interdependencies. Data Knowledge Graphs depict entities as nodes and the relationships between them as edges, creating a graphical model that facilitates understanding and exploration of the data landscape.

  Both Ontology Models and Knowledge Graphs aim to capture the semantics and context of data, enabling more effective data integration, analysis, and utilization within an organization's decision-making processes.

Other approaches include:

- **Entity-Relationship Modeling (ER):** ER modeling is a widely used approach in database design, where entities (objects or concepts) and their relationships are represented graphically. ER models can capture semantic relationships like inheritance, composition, and associations between entities. While ER modeling is primarily used for database design, it can also be employed for semantic data modeling by incorporating additional semantic constraints and rules.
- **Object-Role Modeling (ORM):** ORM is a semantic data modeling approach that focuses on representing the roles played by objects in various relationships. It uses natural language-based representations and can capture complex relationships, constraints, and rules. ORM is particularly useful for conceptual modeling and can be mapped to logical data models or ontologies.
- **Topic Maps:** Topic Maps are a semantic data modeling approach based on the concept of topics (representing subjects or concepts), associations (relationships between topics), and occurrences

(information resources relevant to a topic). Topic Maps are often used for knowledge representation, information integration, and information retrieval applications.

- **Semantic Data Models for Specific Domains:** Several semantic data modeling approaches have been developed specifically for certain domains, such as the Gene Ontology (GO) for biology, the SNOMED CT (Systematized Nomenclature of Medicine – Clinical Terms) for healthcare, and the Friend of a Friend (FOAF) vocabulary for describing people and their relationships in social networks.

The choice of the semantic data modeling approach depends on factors such as the complexity of the domain, the intended use cases (e.g., data integration, knowledge representation, reasoning), the required level of formality, and the existing standards or vocabularies in the domain. In some cases, a combination of approaches may be used to leverage the strengths of different methods. It's worth noting that semantic data modeling is often accompanied by techniques like ontology mapping, data integration, and reasoning engines to fully leverage the semantic representations and derive meaningful insights from the data.

Real-World Examples of Semantic Data Modeling

Banking and Finance

Problem: Banks struggle to provide a consistent customer experience due to siloed data sources.
Solution: A semantic data model using knowledge graphs and ontologies like FIBO helps create a common business language across the organization.
This enhances customer experience through personalized recommendations, self-service portals, and automated query resolution.

Healthcare

Problem: Healthcare data comes from disparate sources, making it challenging to create a single definition of master data.
An example is "Electronic Health Records" which are created from varied sources. This data could come from the patient's clinical records, hospital lab results, wearable devices, and more. Here, a medical condition can be described in different ways.
For instance, what I call a "backache" might be referred to as "spondylitis" in medical terminology. If the data doesn't link these semantically equivalent terms, it could mistakenly seem like I'm suffering from two distinct conditions instead of just one.
Solution: Semantic models bring strong interoperability by creating a common vocabulary, defining synonyms, and standardizing terminology across systems.
This ensures the accurate exchange of health records and prevents misdiagnosis due to terminology differences.

Automation

Problem: Robotic Process Automation excels in automating simple and repetitive tasks, such as back-office work. However, it struggles with unstructured, complex data that carries implicit meanings. This leads to scalability issues as the data volume increases. This is where semantics becomes crucial.
Solution: Semantic models provide structure and meaning to integrated data by analyzing unstructured information, establishing relationships among data points, and feeding this semantic understanding to automation tools, enabling them to scale.

Using High-Level Conceptual Data Models for Database Design

A simplified overview of the database design process. The first step shown is **requirements collection and analysis**. During this step, the database designers interview prospective database users to understand and document their **data requirements**. The result of this step is a concisely written set of users' requirements. These requirements should be specified in as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify the known **functional requirements** of the application.

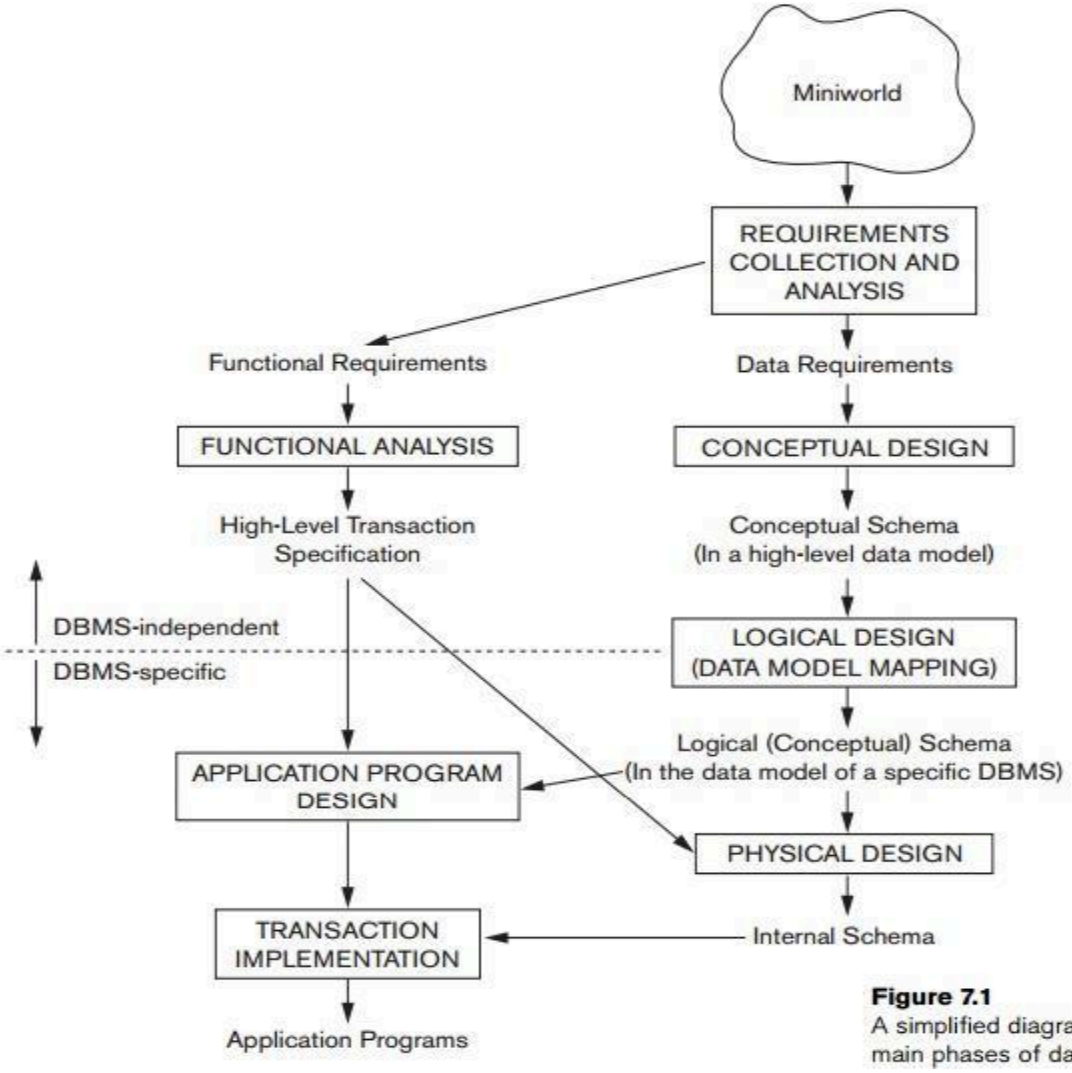


Figure 7.1
A simplified diagram to illustrate the main phases of database design.

These consist of the user-defined **operations** (or **transactions**) that will be applied to the database, including both retrievals and updates. In software design, it is common to use *data flow diagrams*, *sequence diagrams*, *scenarios*, and other techniques to specify functional requirements. We will not discuss any of these techniques here; they are usually described in detail in software engineering texts.

Once the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design**. The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include implementation details, they are usually easier to understand and can be used to communicate with nontechnical users. The high-level conceptual schema can also be used as a reference to ensure that all users' data requirements are met and that the requirements do not conflict. This approach enables database designers to concentrate on specifying the properties of the data, without being concerned with storage and implementation details. This makes it is easier to create a good conceptual data-base design.

During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user queries and operations identified during functional analysis. This also serves to confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements cannot be specified using the initial schema.

The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational or the object-relational database model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping**; its result is a database schema in the implementation data model of the DBMS. Data model mapping is often automated or semiautomated within the database design tools.

The last step is the **physical design** phase, during which the internal storage structures, file organizations, indexes, access paths, and physical design parameters for the database files are specified. In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications. We discuss the database design process in more detail in Chapter 10.

We present only the basic ER model concepts for conceptual schema design in this chapter. Additional modeling concepts are discussed in Chapter 8, when we introduce the EER model.

Introduction of ER Model



Peter Chen developed the ER diagram in 1976 .The ER model was created to provide a simple and understandable model for representing the structure and logic of databases. It has since evolved into variations such as the Enhanced ER Model and the Object Relationship Model
The Entity Relational Model is a model for identifying entities to be represented in the database and representation of how those entities are related. The ER data model specifies enterprise schema that represents the overall logical structure of a database graphically.
The Entity Relationship Diagram explains the relationship among the entities present in the database. ER models are used to model real-world objects like a person, a car, or a company and the relation between these real-world objects. In short, the ER Diagram is the structural format of the database.

Why Use ER Diagrams In DBMS?

- ER diagrams are used to represent the E-R model in a database, which makes them easy to convert into relations (tables).
- ER diagrams provide the purpose of real-world modeling of objects which makes them intently useful.
- ER diagrams require no technical knowledge and no hardware support.
- These diagrams are very easy to understand and easy to create even for a naive user.
- It gives a standard solution for visualizing the data logically.

Symbols Used in ER Model

ER Model is used to model the logical view of the system from a data perspective which consists of these symbols:

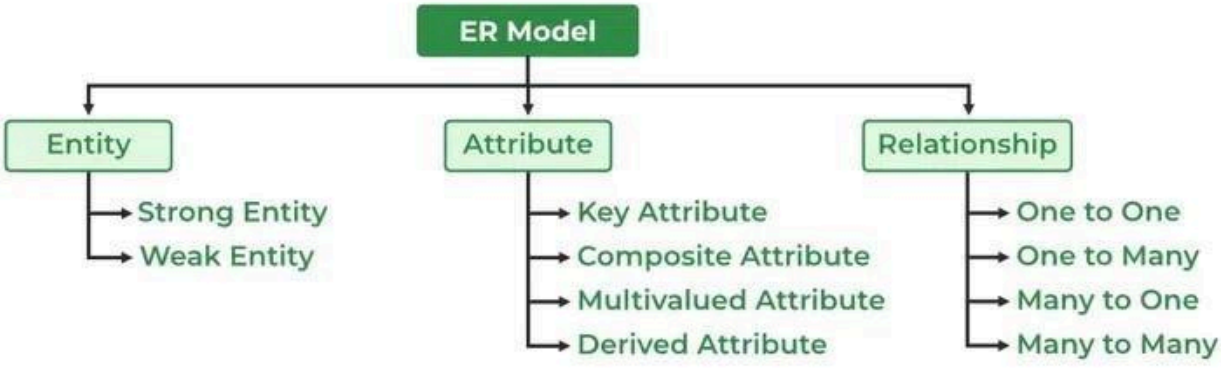
- **Rectangles:** Rectangles represent Entities in the ER Model.
- **Ellipses:** Ellipses represent Attributes in the ER Model.
- **Diamond:** Diamonds represent Relationships among Entities.
- **Lines:** Lines represent attributes to entities and entity sets with other relationship types.
- **Double Ellipse:** Double Ellipses represent Multi-Valued Attributes.
- **Double Rectangle:** Double Rectangle represents a Weak Entity.

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

-

Symbols used in ER Diagram

Components of ER Diagram
ER Model consists of Entities, Attributes, and Relationships among Entities in a Database System.



Components of ER Diagram

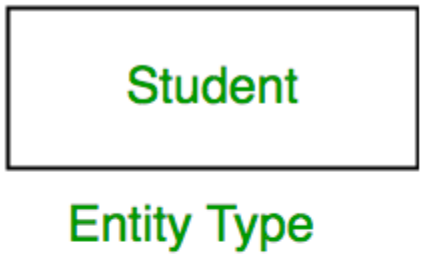
Entity

An Entity may be an object with a physical existence – a particular person, car, house, or employee – or it may be an object with a conceptual existence – a company, a job, or a university course.

Entity are of two types

1.Tangible Entity – Which can be touched like car , person etc.
2.Non – tangible Entity – Which can't be touched like air , bank account etc.

Entity Set: An Entity is an object of Entity Type and a set of all entities is called an entity set. For Example, E1 is an entity having Entity Type Student and the set of all students is called Entity Set. In ER diagram, Entity Type is represented as:



Entity Type



Entity Set

Entity Set

We can represent the entity set in ER Diagram but can't represent entity in ER Diagram because entity is row and column in the relation and ER Diagram is graphical representation of data.

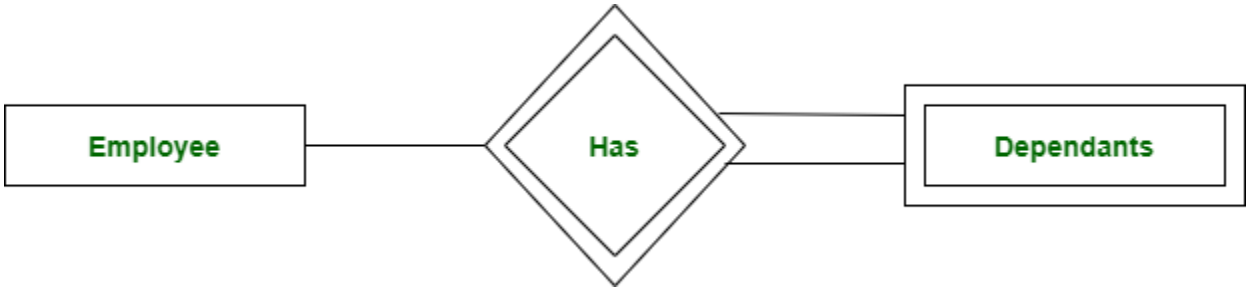
1. Strong Entity

A Strong Entity is a type of entity that has a key Attribute. Strong Entity does not depend on other Entity in the Schema. It has a primary key, that helps in identifying it uniquely, and it is represented by a rectangle. These are called Strong Entity Types.

2. Weak Entity

An Entity type has a key attribute that uniquely identifies each entity in the entity set. But some entity type exists for which key attributes can't be defined. These are called Weak Entity types.
For Example, A company may store the information of dependents (Parents, Children, Spouse) of an Employee. But the dependents can't exist without the employee. So Dependent will be a **Weak Entity Type** and Employee will be Identifying Entity type for Dependent, which means it is **Strong Entity Type**.

A weak entity type is represented by a Double Rectangle. The participation of weak entity types is always total. The relationship between the weak entity type and its identifying strong entity type is called identifying relationship and it is represented by a double diamond.



Strong Entity and Weak Entity

Attributes

Attributes are the properties that define the entity type. For example, Roll_No, Name, DOB, Age, Address, and Mobile_No are the attributes that define entity type Student. In ER diagram, the attribute is represented by an oval.



Attribute

1. Key Attribute

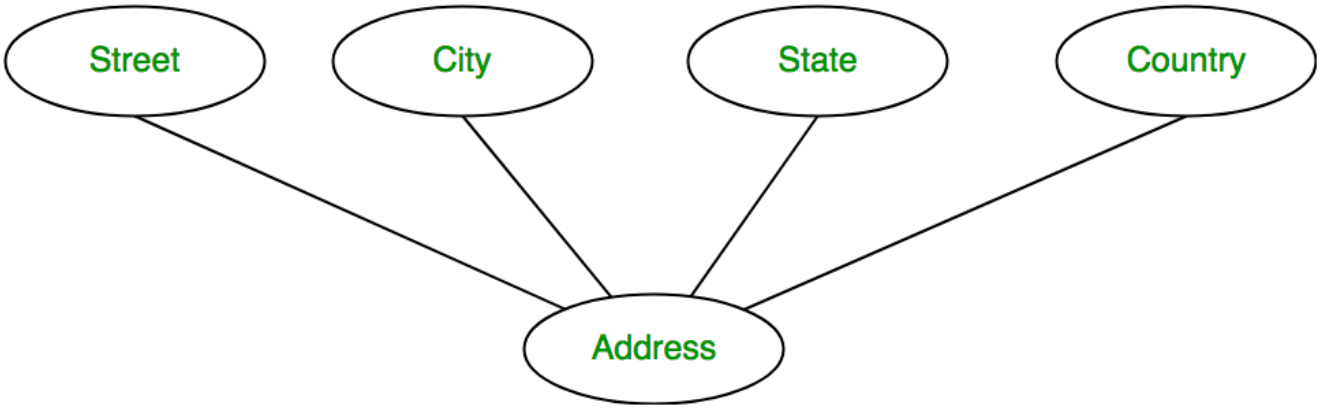
The attribute which **uniquely identifies each entity** in the entity set is called the key attribute. For example, Roll_No will be unique for each student. In ER diagram, the key attribute is represented by an oval with underlying lines.



Key Attribute

2. Composite Attribute

An attribute **composed of many other attributes** is called a composite attribute. For example, the Address attribute of the student Entity type consists of Street, City, State, and Country. In ER diagram, the composite attribute is represented by an oval comprising of ovals.



Composite Attribute

3. Multivalued Attribute

An attribute consisting of more than one value for a given entity. For example, Phone_No (can be more than one for a given student). In ER diagram, a multivalued attribute is represented by a double oval.



Multivalued Attribute

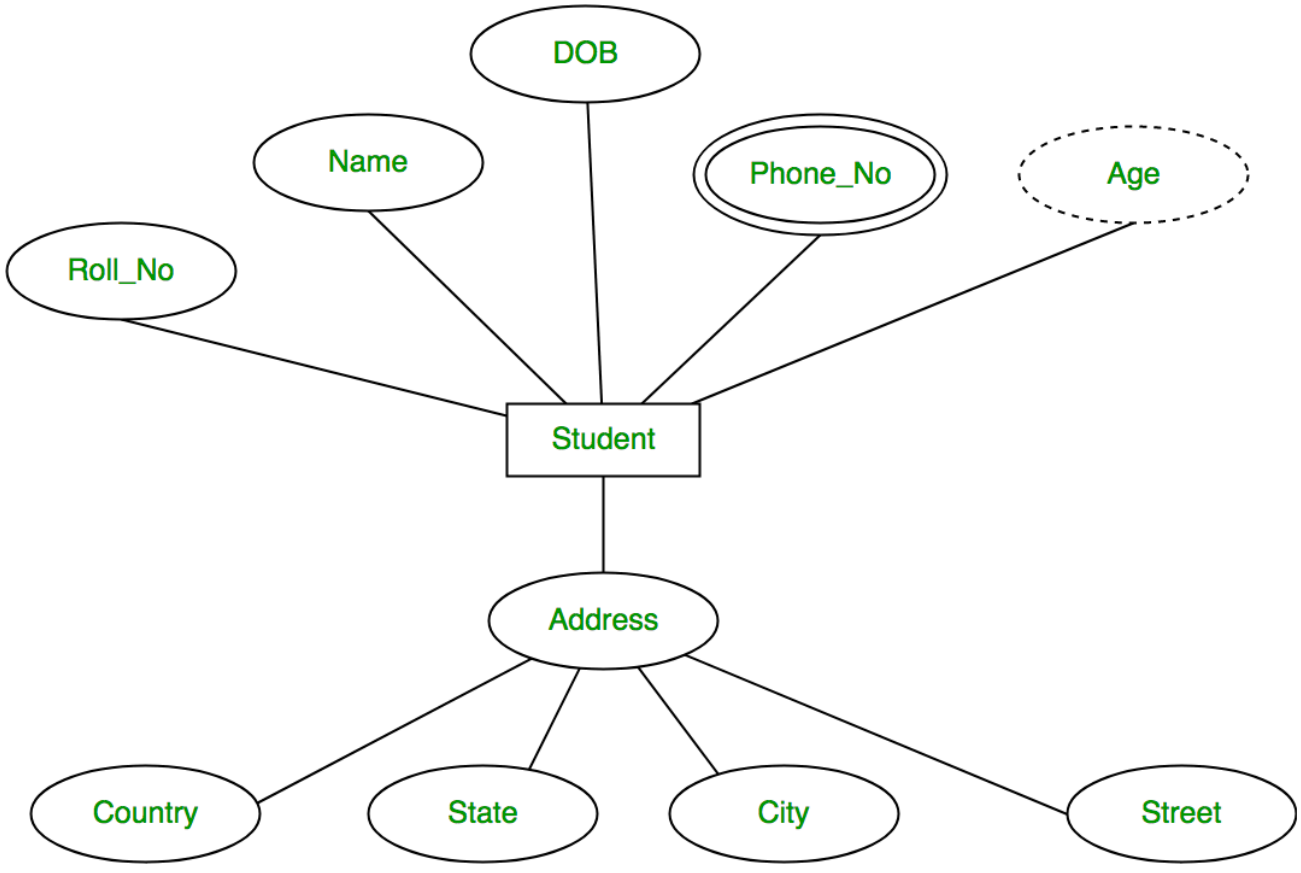
4. Derived Attribute

An attribute that can be derived from other attributes of the entity type is known as a derived attribute. e.g.; Age (can be derived from DOB). In ER diagram, the derived attribute is represented by a dashed oval.



Derived Attribute

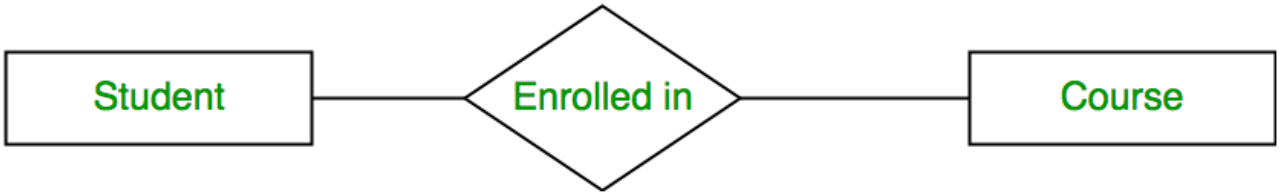
The Complete Entity Type Student with its Attributes can be represented as:



Entity and Attributes

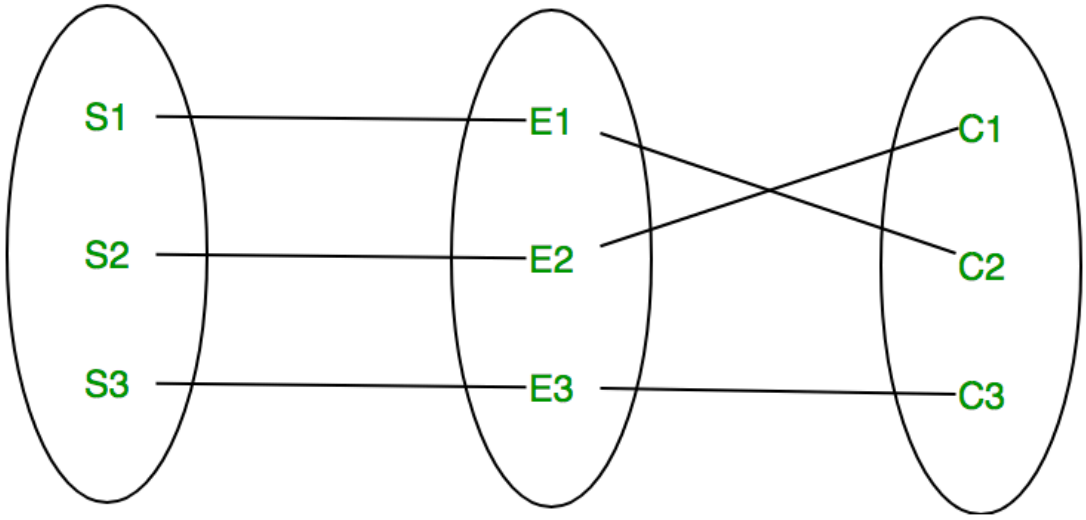
Relationship Type and Relationship Set

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.



Entity-Relationship Set

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.

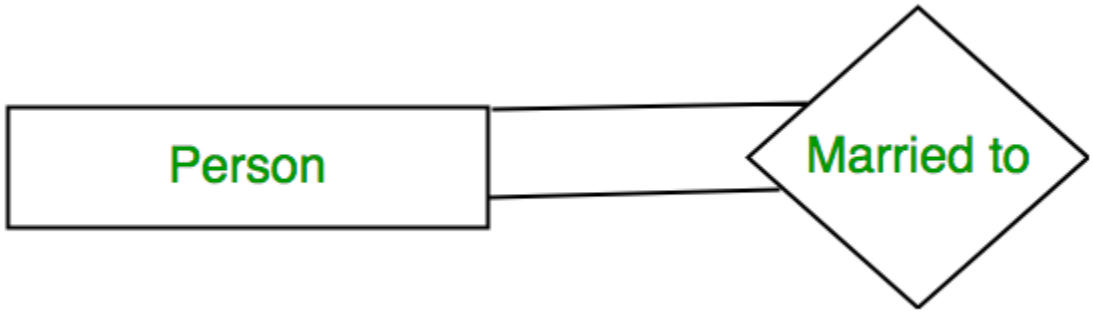


Relationship Set

Degree of a Relationship Set

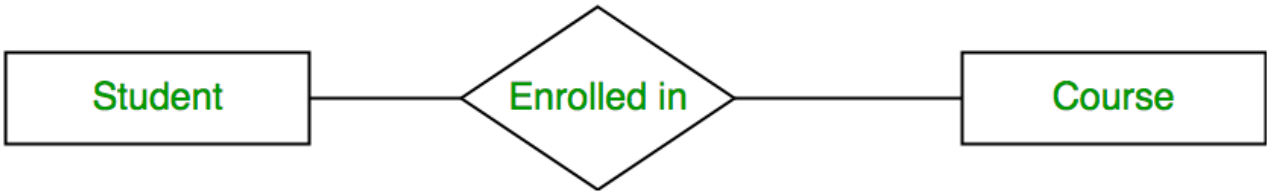
The number of different entity sets participating in a relationship set is called the degree of a relationship set.

1. Unary Relationship: When there is only ONE entity set participating in a relation, the relationship is called a unary relationship. For example, one person is married to only one person.



Unary Relationship

2. Binary Relationship: When there are TWO entities set participating in a relationship, the relationship is called a binary relationship. For example, a Student is enrolled in a Course.



Binary Relationship

3. Ternary Relationship: When there are n entities set participating in a relation, the relationship is called an n-ary relationship.

Cardinality

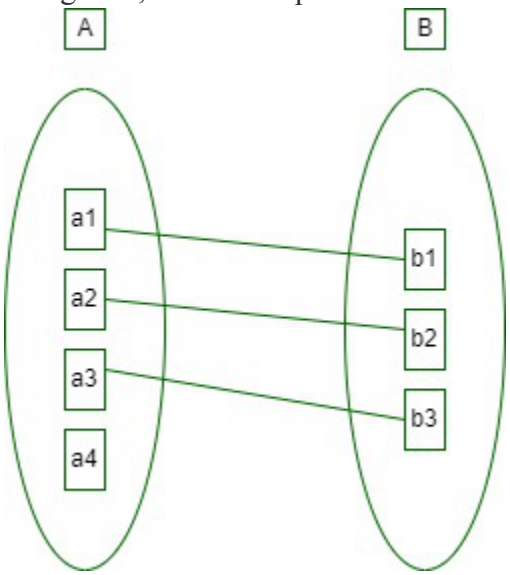
The number of times an entity of an entity set participates in a relationship set is known as cardinality. Cardinality can be of different types:

1. One-to-One: When each entity in each entity set can take part only once in the relationship, the cardinality is one-to-one. Let us assume that a male can marry one female and a female can marry one male. So the relationship will be one-to-one.
the total number of tables that can be used in this is 2.



one to one cardinality

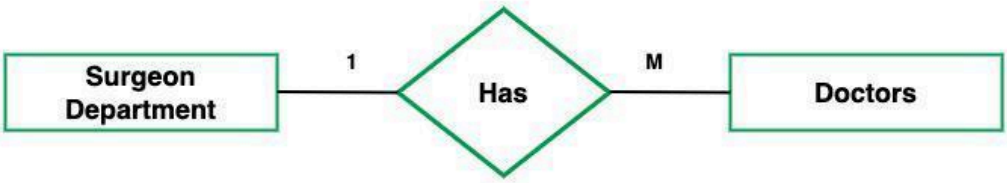
Using Sets, it can be represented as:



Set Representation of One-to-One

2. One-to-Many: In one-to-many mapping as well where each entity can be related to more than one entity and the total number of tables that can be used in this is 2. Let us assume that one

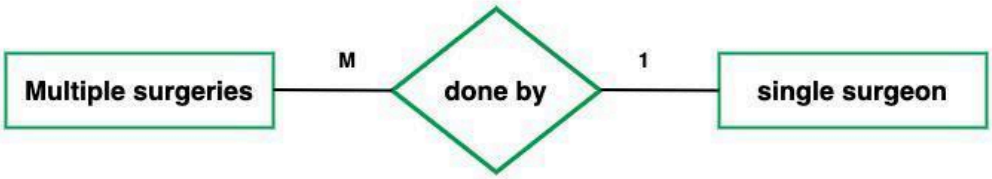
surgeon department can accommodate many doctors. So the Cardinality will be 1 to M. It means one department has many Doctors.



one to many cardinality

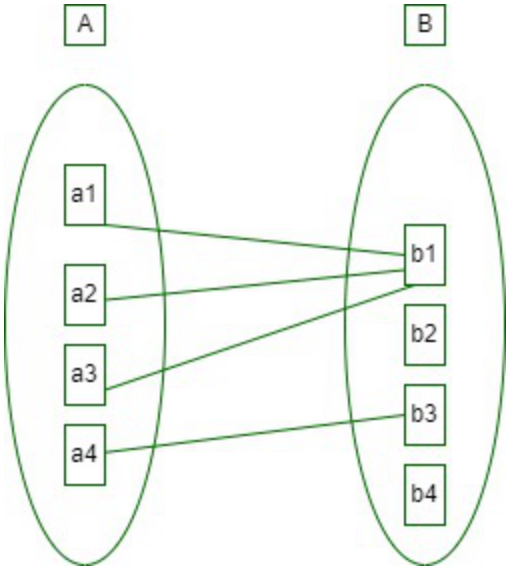
3. Many-to-One: When entities in one entity set can take part only once in the relationship set and entities in other entity sets can take part more than once in the relationship set, cardinality is many to one. Let us assume that a student can take only one course but one course can be taken by many students. So the cardinality will be n to 1. It means that for one course there can be n students but for one student, there will be only one course.

The total number of tables that can be used in this is 3.



many to one cardinality

Using Sets, it can be represented as:



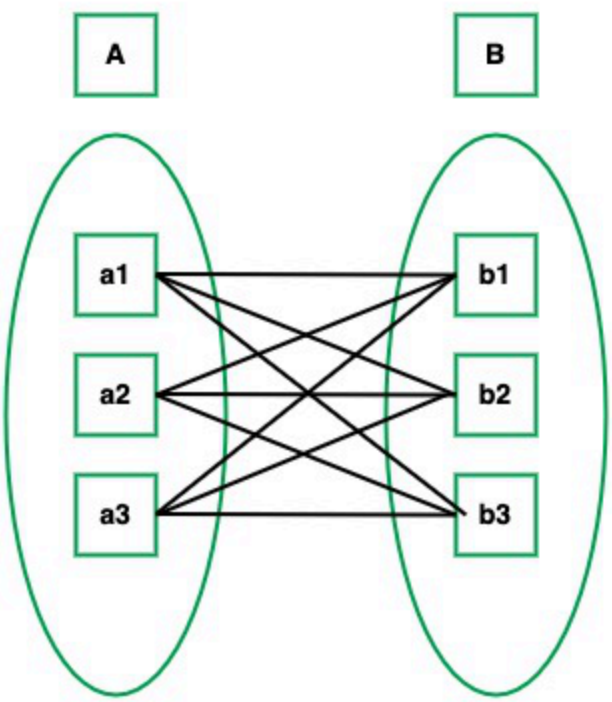
Set Representation of Many-to-One

In this case, each student is taking only 1 course but 1 course has been taken by many students.
4. Many-to-Many: When entities in all entity sets can take part more than once in the relationship cardinality is many to many. Let us assume that a student can take more than one course and one course can be taken by many students. So the relationship will be many to many. the total number of tables that can be used in this is 3.



many to many cardinality

Using Sets, it can be represented as:



Many-to-Many Set Representation

In this example, student S1 is enrolled in C1 and C3 and Course C3 is enrolled by S1, S3, and S4. So it is many-to-many relationships.

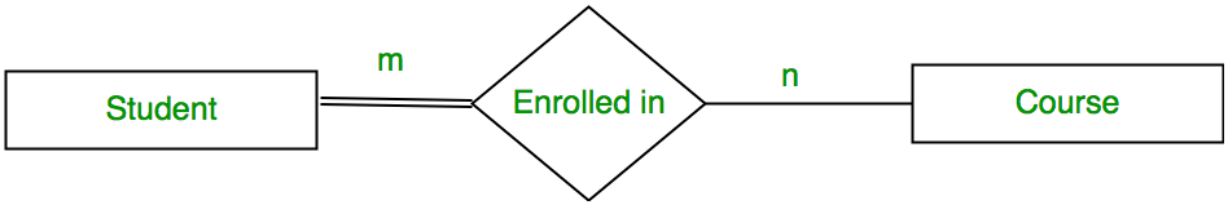
Participation Constraint

Participation Constraint is applied to the entity participating in the relationship set.

1. Total Participation – Each entity in the entity set must participate in the relationship. If each student must enroll in a course, the participation of students will be total. Total participation is shown by a double line in the ER diagram.

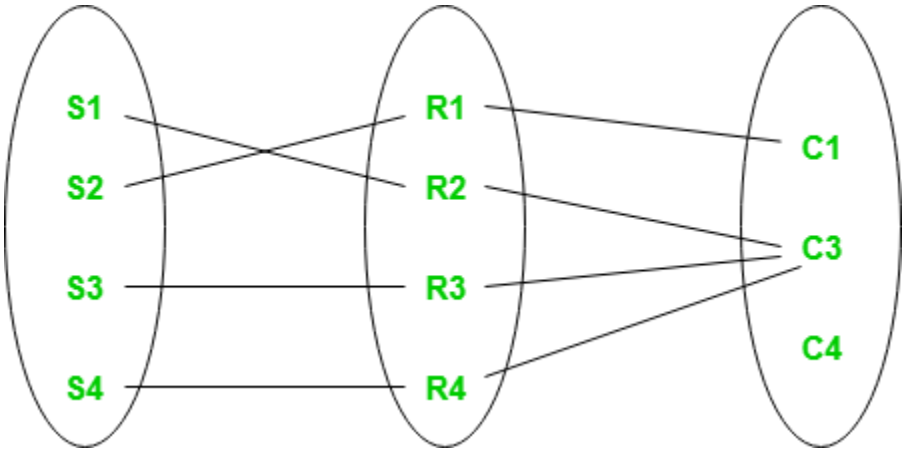
2. Partial Participation – The entity in the entity set may or may NOT participate in the relationship. If some courses are not enrolled by any of the students, the participation in the course will be partial.

The diagram depicts the 'Enrolled in' relationship set with Student Entity set having total participation and Course Entity set having partial participation.



Total Participation and Partial Participation

Using Set, it can be represented as,



Set representation of Total Participation and Partial Participation

Every student in the Student Entity set participates in a relationship but there exists a course C4 that is not taking part in the relationship.

How to Draw ER Diagram?

- The very first step is Identifying all the Entities, and place them in a Rectangle, and labeling them accordingly.
- 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.
- Attach attributes to the entities properly.
- Remove redundant entities and relationships.
- Add proper colors to highlight the data present in the database.

Enhanced ER Model



Introduction of ER Model

Today the complexity of the data is increasing so it becomes more and more difficult to use the traditional ER model for database modeling. To reduce this complexity of modeling we have to make improvements or enhancements to the existing ER model to make it able to handle the complex application in a better way.

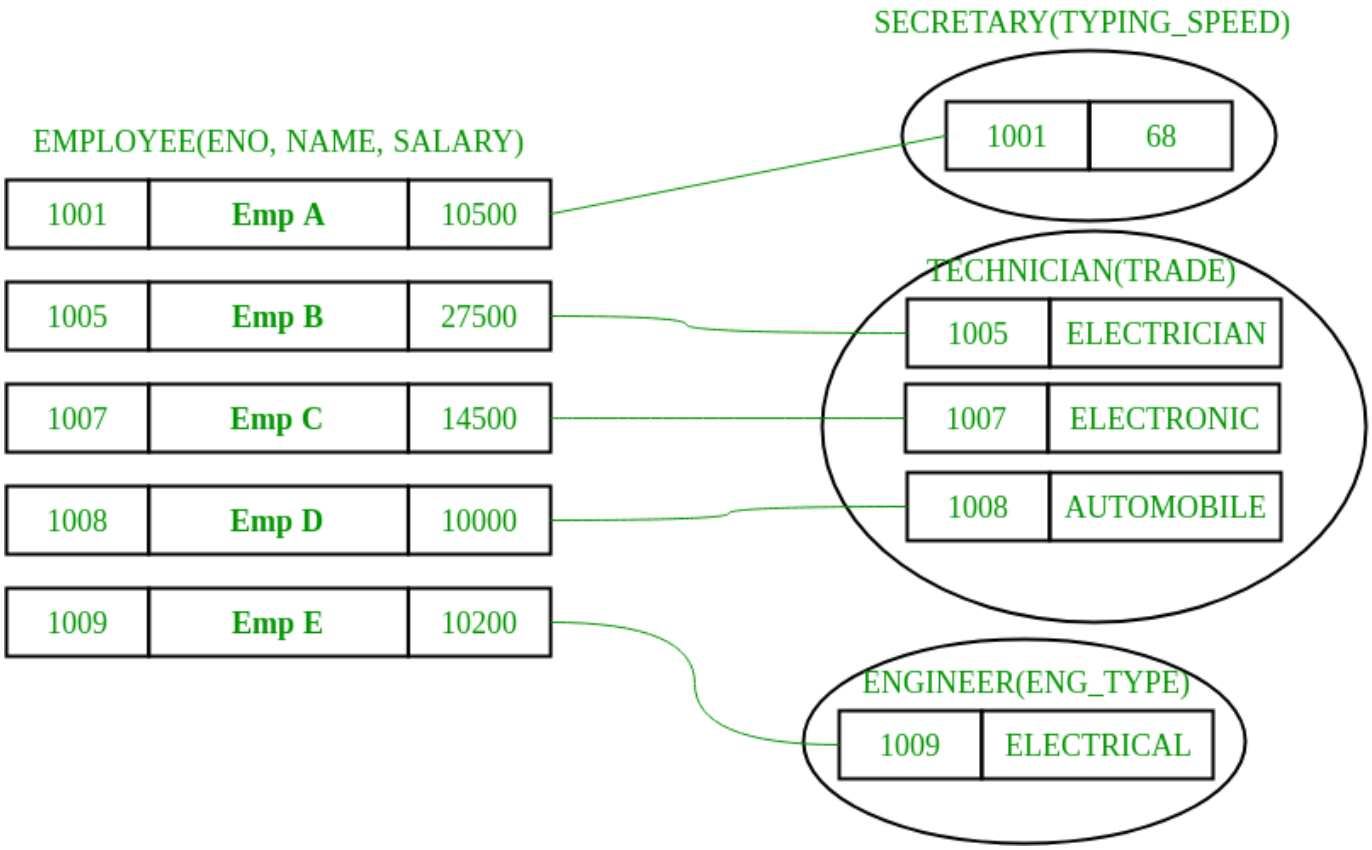
Enhanced entity-relationship diagrams are advanced database diagrams very similar to regular ER diagrams which represent the requirements and complexities of complex databases.

It is a diagrammatic technique for displaying the Sub Class and Super Class; Specialization and Generalization; Union or Category; Aggregation etc.

Generalization and Specialization: These are very common relationships found in real entities. However, this kind of relationship was added later as an enhanced extension to the classical ER model. **Specialized** classes are often called **subclass** while a **generalized class** is called a superclass, probably inspired by object-oriented programming. A sub-class is best understood by **"IS-A analysis"**. The following statements hopefully make some sense to your mind "Technician IS-A Employee", and "Laptop IS-A Computer".

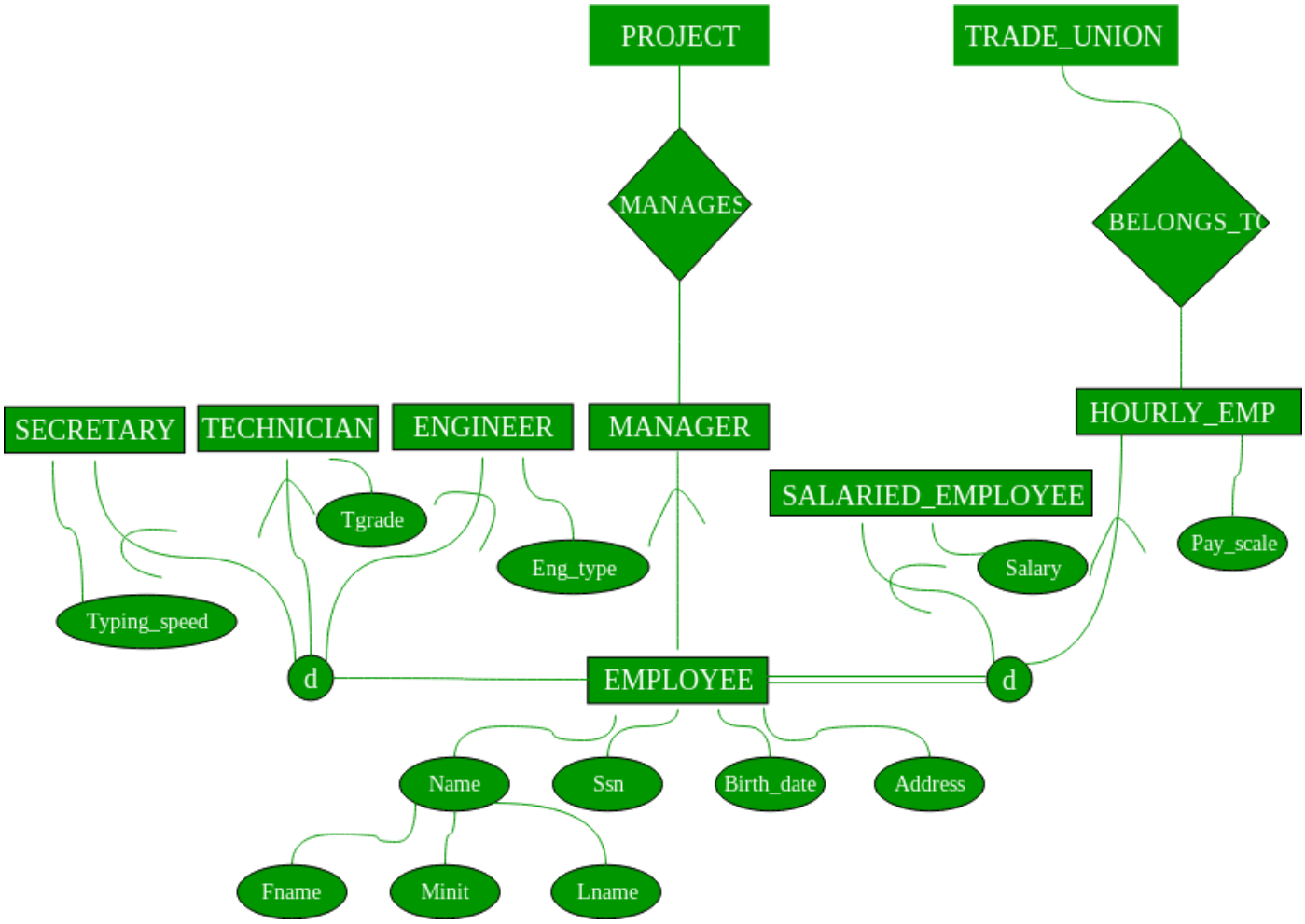
An entity is a specialized type/class of another entity. For example, a Technician is a special Employee in a university system Faculty is a special class of Employees. We call this phenomenon generalization/specialization. In the example here Employee is a generalized entity class while the Technician and Faculty are specialized classes of Employee.

Example:
This example instance of **"sub-class"** relationships. Here we have four sets of employees: Secretary, Technician, and Engineer. The employee is a super-class of the rest three sets of individual sub-class is a subset of Employee set.



- An entity belonging to a sub-class is related to some super-class entity. For instance emp, no 1001 is a secretary, and his typing speed is 68. Emp no 1009 is an engineer (sub-class) and her trade is “Electrical”, so forth.
- Sub-class entity “inherits” all attributes of super-class; for example, employee 1001 will have attributes eno, name, salary, and typing speed.

Enhanced ER model of above example



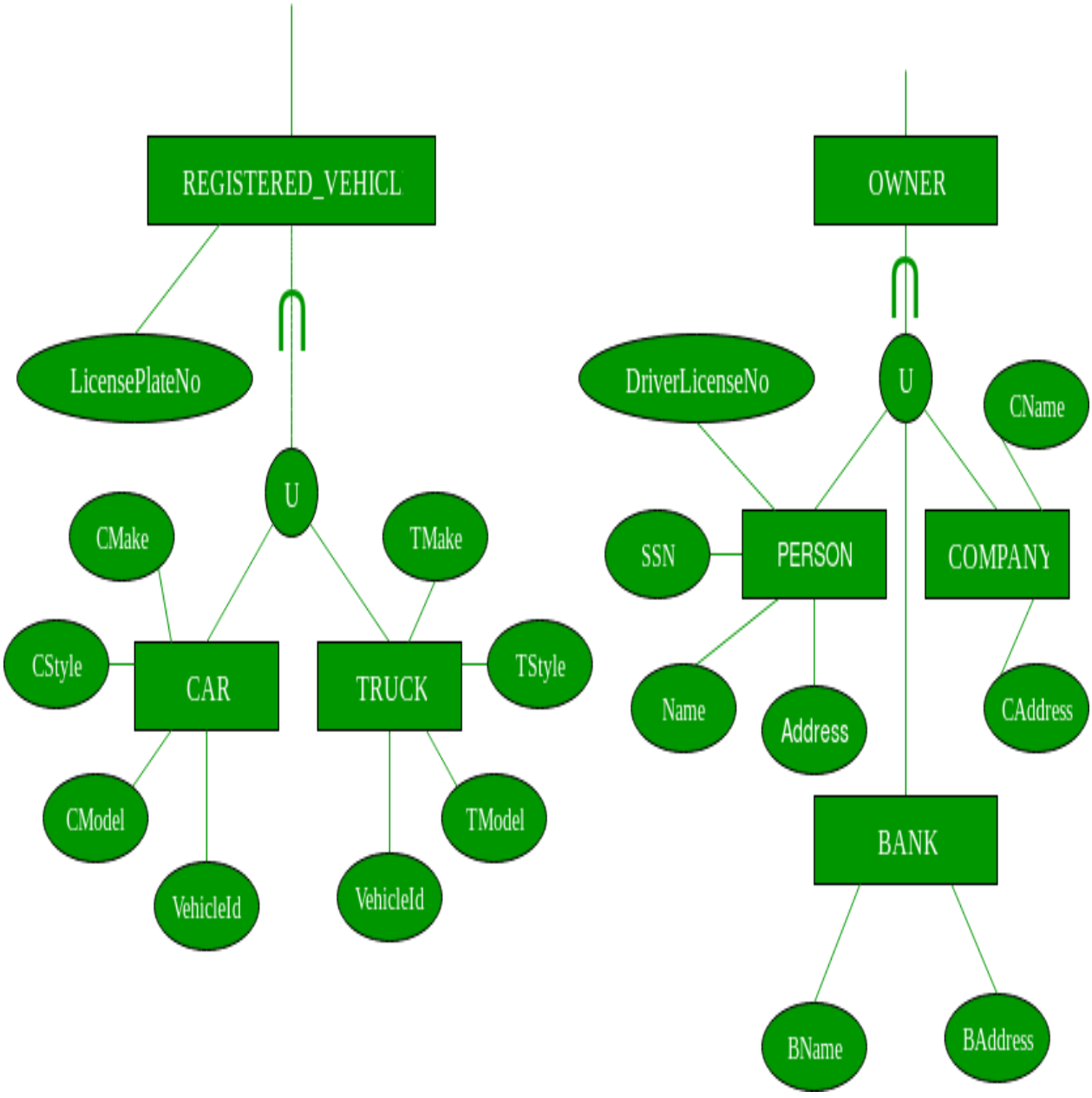
Constraints – There are two types of constraints on the "Sub-class" relationship.

1. **Total or Partial –** A sub-classing relationship is total if every super-class entity is to be associated with some sub-class entity, otherwise partial. Sub-class "job type based employee category" is partial sub-classing – not necessary every employee is one of (secretary, engineer, and technician), i.e. union of these three types is a proper subset of all employees. Whereas other sub-classing "Salaried Employee AND Hourly Employee" is total; the union of entities from sub-classes is equal to the total employee set, i.e. every employee necessarily has to be one of them.
2. **Overlapped or Disjoint –** If an entity from a super-set can be related (can occur) in multiple sub-class sets, then it is overlapped sub-classing, otherwise disjoint. Both the examples: job-type based and salaries/hourly employee sub-classing are disjoint.

Note – *These constraints are independent of each other: can be "overlapped and total or partial" or "disjoint and total or partial". Also, sub-classing has transitive properties.*

Multiple Inheritance (sub-class of multiple superclasses**) –**
An entity can be a sub-class of multiple entity types; such entities are sub-class of multiple entities and have multiple super-classes; Teaching Assistant can subclass of Employee and Student both. A faculty in a university system can be a subclass of Employee and Alumnus. In multiple inheritances, attributes of sub-class are the union of attributes of all super-classes.
Union –

- Set of Library Members is **UNION** of Faculty, Student, and Staff. A union relationship indicates either type; for example, a library member is either Faculty or Staff or Student.
- Below are two examples that show how **UNION** can be depicted in ERD – Vehicle Owner is UNION of PERSON and Company, and RTO Registered Vehicle is UNION of Car and Truck.



You might see some confusion in Sub-class and UNION; consider an example in above figure Vehicle is super-class of CAR and Truck; this is very much the correct example of the subclass as well but here use it differently we are saying RTO Registered vehicle is UNION of Car and Vehicle, they do not inherit any attribute of Vehicle, attributes of car and truck are altogether independent set, where is in sub-classing situation car and truck would be inheriting the attribute of vehicle class.

An Enhanced Entity-Relationship (EER) model is an extension of the original Entity-Relationship (ER) model that includes additional concepts and features to support more complex data modeling requirements. The EER model includes all the elements of the ER model and adds new constructs, such as subtypes and supertypes, generalization and specialization, and inheritance.

Here are some of the key features of the EER model:

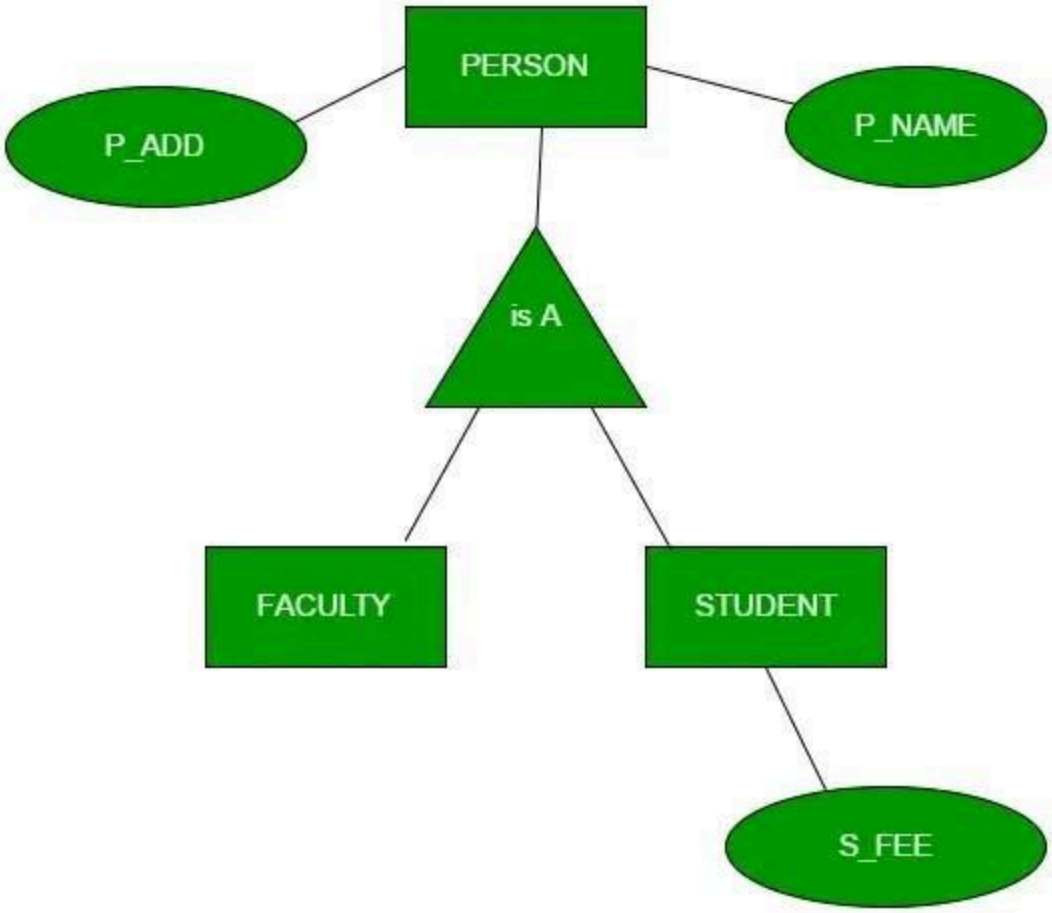
- **Subtypes and Supertypes:** The EER model allows for the creation of subtypes and supertypes. A supertype is a generalization of one or more subtypes, while a subtype is a specialization of a supertype. For example, a vehicle could be a supertype, while car, truck, and motorcycle could be subtypes.
- **Generalization and Specialization:** Generalization is the process of identifying common attributes and relationships between entities and creating a supertype based on these common features. Specialization is the process of identifying unique attributes and relationships between entities and creating subtypes based on these unique features.
- **Inheritance:** Inheritance is a mechanism that allows subtypes to inherit attributes and relationships from their supertype. This means that any attribute or relationship defined for a supertype is automatically inherited by all its subtypes.
- **Constraints:** The EER model allows for the specification of constraints that must be satisfied by entities and relationships. Examples of constraints include cardinality constraints, which specify the number of relationships that can exist between entities, and participation constraints, which specify whether an entity is required to participate in a relationship.
- Overall, the EER model provides a powerful and flexible way to model complex data relationships, making it a popular choice for database design. An Enhanced Entity-Relationship (EER) model is an extension of the traditional Entity-Relationship (ER) model that includes additional features to represent complex relationships between entities more accurately. Some of the main features of the EER model are:
- **Subclasses and Superclasses:** EER model allows for the creation of a hierarchical structure of entities where a superclass can have one or more subclasses. Each subclass inherits attributes and relationships from its superclass, and it can also have its unique attributes and relationships.
- **Specialization and Generalization:** EER model uses the concepts of specialization and generalization to create a hierarchy of entities. Specialization is the process of defining subclasses from a superclass, while generalization is the process of defining a superclass from two or more subclasses.
- **Attribute Inheritance:** EER model allows attributes to be inherited from a superclass to its subclasses. This means that attributes defined in the superclass are automatically inherited by all its subclasses.

- **Union Types: E**ER model allows for the creation of a union type, which is a combination of two or more entity types. The union type can have attributes and relationships that are common to all the entity types that make up the union.
- **Aggregation:** EER model allows for the creation of an aggregate entity that represents a group of entities as a single entity. The aggregate entity has its unique attributes and relationships.
- **Multi-valued Attributes:** EER model allows an attribute to have multiple values for a single entity instance. For example, an entity representing a person may have multiple phone numbers.
- **Relationships with Attributes:** EER model allows relationships between entities to have attributes. These attributes can describe the nature of the relationship or provide additional information about the relationship.

Generalization, Specialization and Aggregation in ER Model

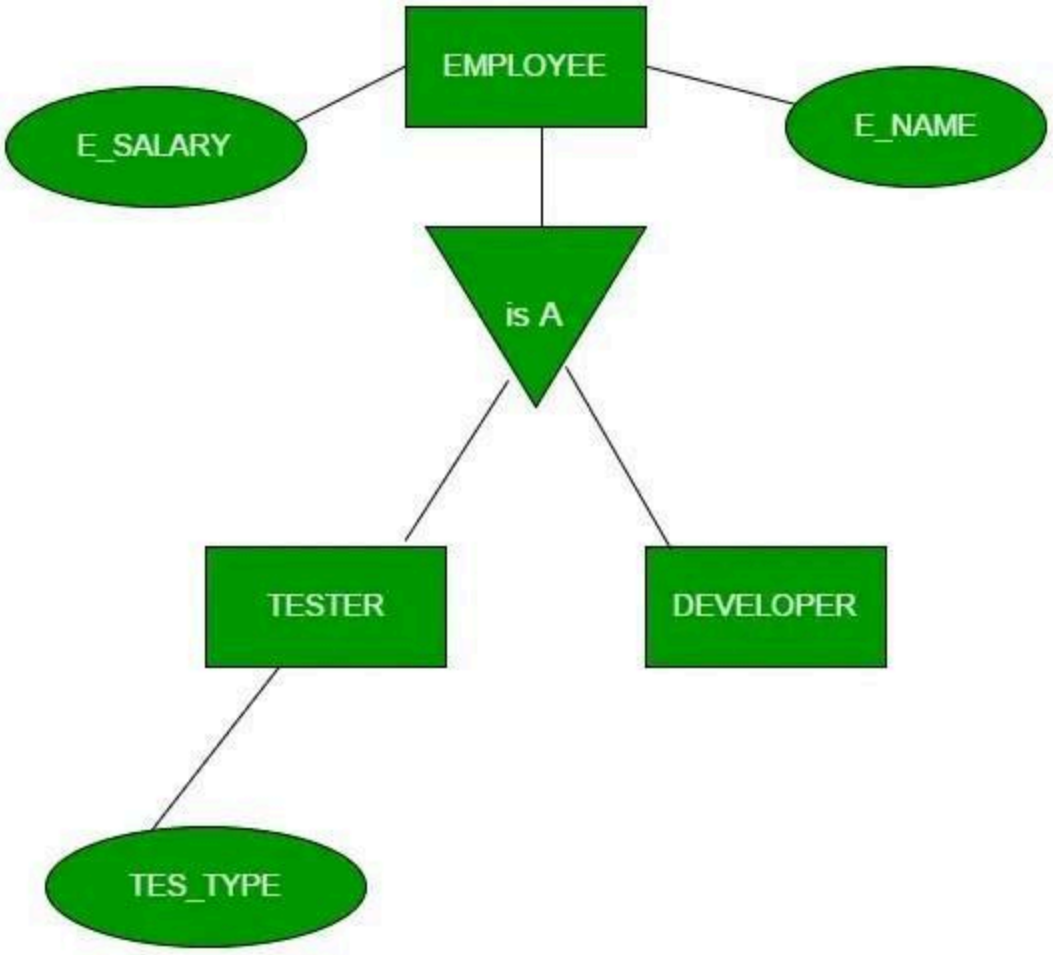
Using the ER model for bigger data creates a lot of complexity while designing a database model, So in order to minimize the complexity Generalization, Specialization, and Aggregation were introduced in the ER model and these were used for data abstraction in which an abstraction mechanism is used to hide details of a set of objects. Some of the terms were added to the Enhanced ER Model, where some new concepts were added. These new concepts are:

- Generalization
- Specialization
- Aggregation
    - **Generalization**
    - Generalization is the process of extracting common properties from a set of entities and creating a generalized entity from it. It is a bottom-up approach in which two or more entities can be generalized to a higher-level entity if they have some attributes in common. For Example, STUDENT and FACULTY can be generalized to a higher-level entity called PERSON as shown in Figure 1. In this case, common attributes like P_NAME, and P_ADD become part of a higher entity (PERSON), and specialized attributes like S_FEE become part of a specialized entity (STUDENT).



Specialization

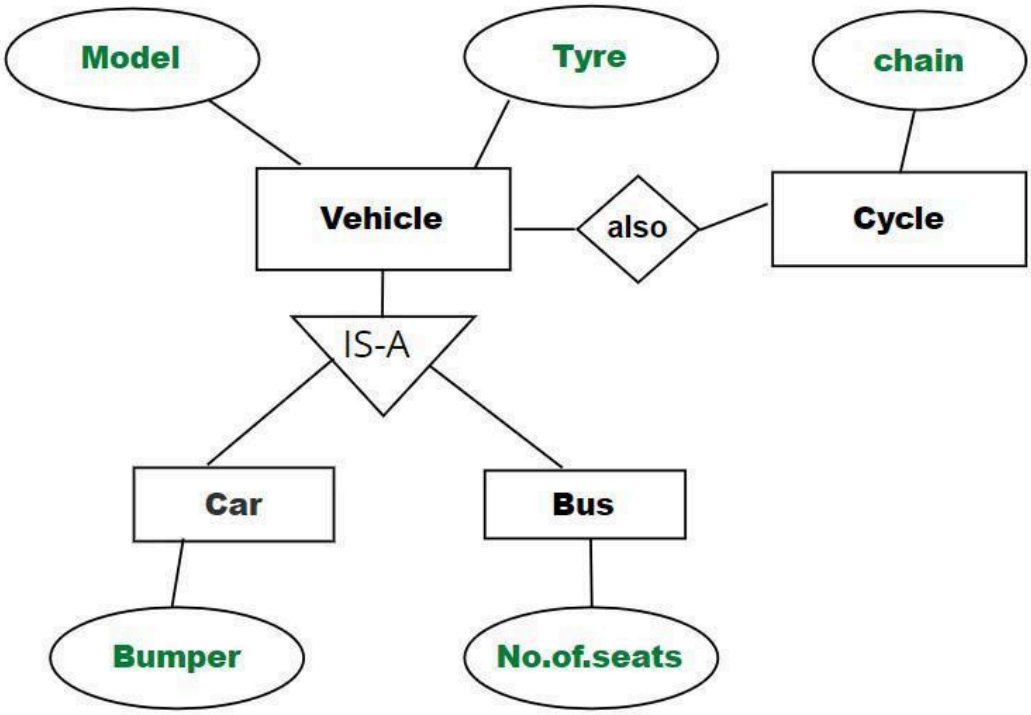
In specialization, an entity is divided into sub-entities based on its characteristics. It is a top-down approach where the higher-level entity is specialized into two or more lower-level entities. For Example, an EMPLOYEE entity in an Employee management system can be specialized into DEVELOPER, TESTER, etc. as shown in Figure 2. In this case, common attributes like E_NAME, E_SAL, etc. become part of a higher entity (EMPLOYEE), and specialized attributes like TES_TYPE become part of a specialized entity (TESTER).



Specialization

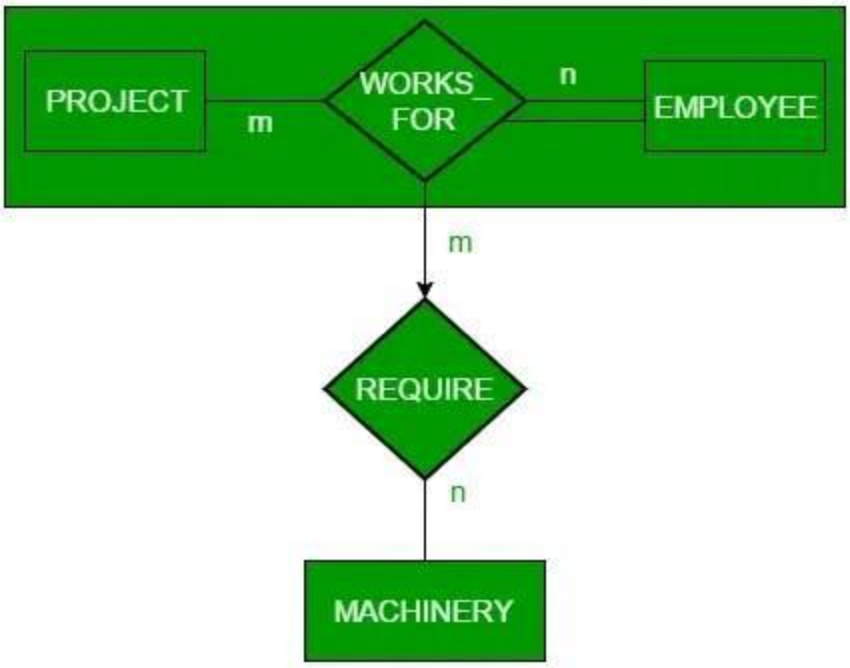
Inheritance: It is an important feature of generalization and specialization

- **Attribute inheritance**: allows lower level entities to inherit the attributes of higher level entities and vice versa.
- in diagram: **Car** entity is an inheritance of **Vehicle** entity ,So Car can acquire attributes of **Vehicle** example:car can acquire **Model** attribute of **Vehicle**.
- **Participation inheritance**: In participation inheritance, relationships involving higher level entity set also inherited by lower level entity and vice versa.
- in diagram: Vehicle entity has an relationship with Cycle entity ,So **Cycle entity** can acquire attributes of lower level entities i.e **Car** and **Bus** since it is inheritance of **Vehicle.**



Aggregation

An ER diagram is not capable of representing the relationship between an entity and a relationship which may be required in some scenarios. In those cases, a relationship with its corresponding entities is aggregated into a higher-level entity. Aggregation is an abstraction through which we can represent relationships as higher-level entity sets.
For Example, an Employee working on a project may require some machinery. So, REQUIRE relationship is needed between the relationship WORKS_FOR and entity MACHINERY. Using aggregation, WORKS_FOR relationship with its entities EMPLOYEE and PROJECT is aggregated into a single entity and relationship REQUIRE is created between the aggregated entity and MACHINERY.



Aggregation

Representing Aggregation Via Schema

To represent aggregation, create a schema containing the following things.

- the primary key to the aggregated relationship
- the primary key to the associated entity set
- descriptive attribute, if exists

Modeling of UNION types using categories

In DBMS, Superclass/subclass relationships with a single super-class. A shared subclass may be represented in multiple superclass/subclass relationships, where each relationship has a single superclass.

Understanding Superclass/Subclass Relationships in DBMS

A single superclass/subclass relationship using more than one superclass. Each superclass represents a distinct entity type. Subclass represents a group of objects that is a subset of the UNION of the distinct entity types. This subclass is known as union type or a category.

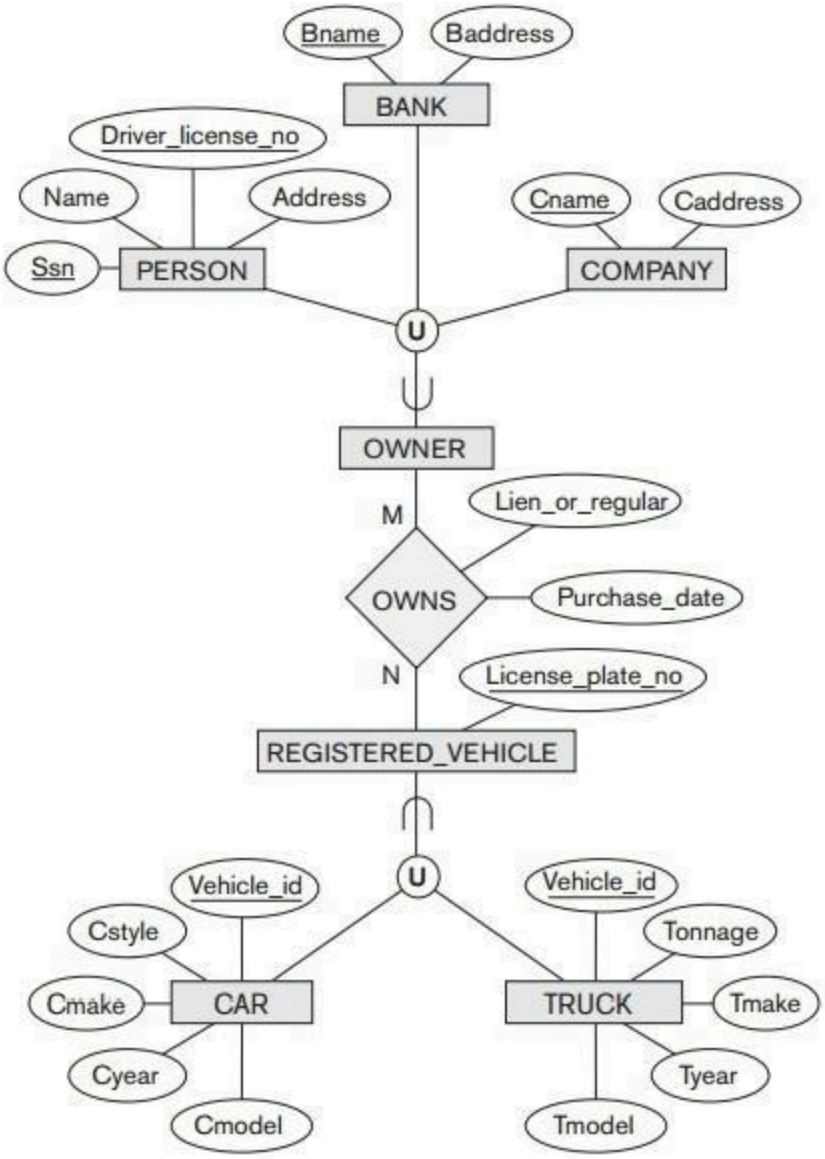
Example

Consider the scenario where we have three distinct entity types: PERSON, BANK, and COMPANY. In a motor vehicle registration database, a vehicle owner can be classified as a person, a bank holding a lien on the vehicle, or a company. To create a class (i.e., a collection of entities). That encompasses all three entity types, we need to construct a subclass that represents the union of the three sets, which we will call OWNER. This subclass is a category, also known as a union type.

Category in an Entity-Relationship Diagram

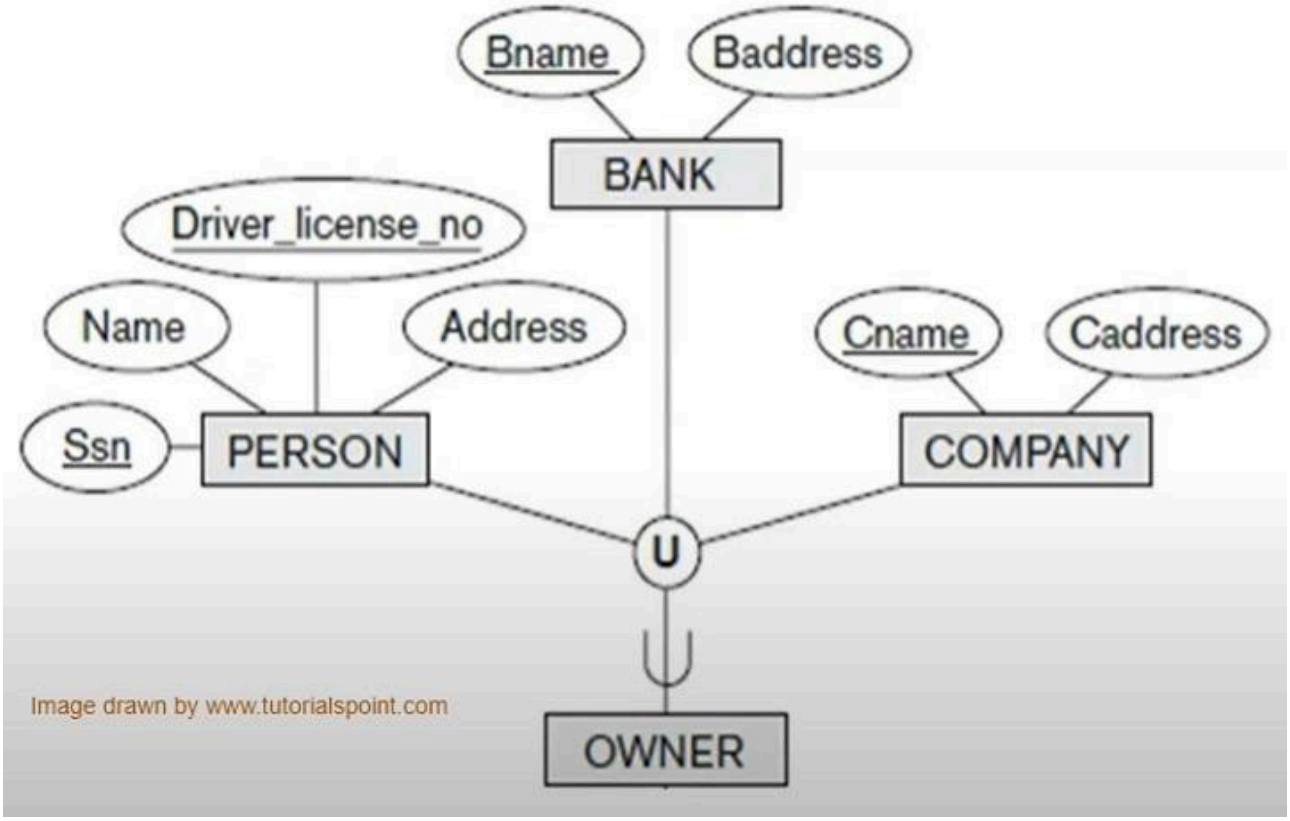
To visually represent this subclass in an Entity-Relationship (EER) diagram, we use a circle with the symbol "∪" to denote the set union operation, which is connected to the superclasses COMPANY, BANK, and PERSON. An arc with the subset symbol connects the circle to the OWNER category, indicating that the OWNER category is a subclass of the union of the three entity types. If necessary, we can display a defining predicate next to the line from the superclass to which the predicate applies.

In Figure below, we can see two categories displayed in the EER diagram: OWNER and REGISTERED_VEHICLE. OWNER is a subclass of the union of PERSON, BANK, and COMPANY, while REGISTERED_VEHICLE is a subclass of the union of CAR and TRUCK.



Comparison of Category and Shared Subclass: OWNER and ENGINEERING_MANAGER

Category is a subclass. It has two or more superclasses representing distinct entity types. Superclass/subclass relationships only have a single superclass. To better understand the distinction between a category and other subclass relationships, let us compare the OWNER category in Figure above with the shared subclass ENGINEERING_MANAGER.



ENGINEERING_MANAGER subclass is member of each of these superclasses: ENGINEER, MANAGER, and SALARIED_EMPLOYEE. Entity which is part of ENGINEERING_MANAGER must also exist in all three of its superclasses. So, ENGINEERING_MANAGER is subset of the intersection of the three classes. Engineering manager must be an ENGINEER, MANAGER, and a SALARIED_EMPLOYEE.

On the other hand, a category such as OWNER is a subset of the union of its superclasses. Entity that is member of OWNER must exist in only one of its superclasses. In Figure above, an OWNER entity may be COMPANY, BANK, or PERSON.

Attribute inheritance operates in the case of categories. In Figure above, each OWNER entity inherits the attributes of COMPANY, PERSON, or BANK. It depends on superclass to which entity belongs. In a shared subclass such as ENGINEERING_MANAGER (Figure above), the subclass inherits all the attributes of its superclasses SALARIED_EMPLOYEE, ENGINEER, and MANAGER.

MODULE 3:

Introduction of 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. Set Intersection(∩)
6. Rename(ρ)
7. Cartesian Product(X)

1. Selection(σ): It is used to select required tuples of the relations.
Example:

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

For the above relation, **σ(c>3)R** will select the tuples which have c more than 3.

A	B	C
1	2	4
4	3	4

Note: The selection operator only selects the required tuples but does not display them. For display, the data projection operator is used.

2. Projection(π): It is used to project required column data from a relation.
Example: Consider Table 1. Suppose we want columns B and C from Relation R.
π(B,C)R will show following columns.

B	C
2	4
2	3
3	4

Note: By Default, projection removes duplicate data.

3. Union(U): Union operation in relational algebra is the same as union operation in set theory.
Example:
FRENCH

Student_Name	Roll_Number
Ram	01
Mohan	02
Vivek	13
Geeta	17

GERMAN

Student_Name	Roll_Number
Vivek	13
Geeta	17
Shyam	21
Rohan	25

Consider the following table of Students having different optional subjects in their course.
π(Student_Name)FRENCH U π(Student_Name)GERMAN

Student_Name
Ram
Mohan
Vivek
Geeta
Shyam
Rohan

Note: The only constraint in the union of two relations is that both relations must have the same set of Attributes.

4. Set Difference(-): Set Difference in relational algebra is the same set difference operation as in set theory.
Example: From the above table of FRENCH and GERMAN, Set Difference is used as follows
π(Student_Name)FRENCH - π(Student_Name)GERMAN

Student_Name
Ram
Mohan

Note: The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.

5. Set Intersection(∩): Set Intersection in relational algebra is the same set intersection operation in set theory.
Example: From the above table of FRENCH and GERMAN, the Set Intersection is used as follows
π(Student_Name)FRENCH ∩ π(Student_Name)GERMAN

Student_Name
Vivek
Geeta

Note: The only constraint in the Set Difference between two relations is that both relations must have the same set of Attributes.

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

7. Cross Product(X): 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. Each record of A will pair with every record of B.
Example:
A

Name	Age	Sex
Ram	14	M
Sona	15	F
Kim	20	M

B

ID	Course
1	DS
2	DBMS

A X B

Name	Age	Sex	ID	Course
Ram	14	M	1	DS
Ram	14	M	2	DBMS
Sona	15	F	1	DS
Sona	15	F	2	DBMS
Kim	20	M	1	DS
Kim	20	M	2	DBMS

Note: If A has 'n' tuples and B has 'm' tuples then A X B will have ' n*m ' tuples.

Derived Operators

These are some of the derived operators, which are derived from the fundamental operators.

1. Natural Join(⋈)
2. Conditional Join

1. Natural Join(⋈): Natural join is a binary operator. Natural join between two or more relations will result in a set of all combinations of tuples where they have an equal common attribute.

Example:

EMP

Name	ID	Dept_Name
A	120	IT
B	125	HR
C	110	Sales
D	111	IT

DEPT

Dept_Name	Manager
Sales	Y
Production	Z
IT	A

Natural join between EMP and DEPT with condition :

EMP.Dept_Name = DEPT.Dept_Name

EMP ⋈ DEPT

Name	ID	Dept_Name	Manager
A	120	IT	A
C	110	Sales	Y
D	111	IT	A

2. Conditional Join: Conditional join works similarly to natural join. In natural join, by default condition is equal between common attributes while in conditional join we can specify any condition such as greater than, less than, or not equal.

Example:

R

ID	Sex	Marks
1	F	45
2	F	55
3	F	60

S

ID	Sex	Marks
10	M	20
11	M	22
12	M	59

Join between R and S with condition **R.marks >= S.marks**

R.ID	R.Sex	R.Marks	S.ID	S.Sex	S.Marks
1	F	45	10	M	20
1	F	45	11	M	22
2	F	55	10	M	20
2	F	55	11	M	22
3	F	60	10	M	20
3	F	60	11	M	22
3	F	60	12	M	59

Basic Operators in Relational Algebra

Basics of Relational model: Relational Model

Relational Algebra is a procedural query language that takes relations as an input and returns relations as an output. There are some basic operators which can be applied in relation to producing the required results which we will discuss one by one. We will use STUDENT_SPORTS, EMPLOYEE, and STUDENT relations as given in Table 1, Table 2, and Table 3 respectively to understand the various operators.

Table 1: STUDENT_SPORTS

ROLL_NO	SPORTS
1	Badminton
2	Cricket
2	Badminton
4	Badminton

Table 2: EMPLOYEE

EMP_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
5	NARESH	HISAR	9782918192	22
6	SWETA	RANCHI	9852617621	21
4	SURESH	DELHI	9156768971	18

Table 3: STUDENT

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20

4	SURESH	DELHI	9156768971	18

Selection operator (?): Selection operator is used to selecting tuples from a relation based on some condition. Syntax:

? **(Cond)(Relation Name)**

Extract students whose age is greater than 18 from STUDENT relation given in Table 3

? **(AGE>18)(STUDENT)**

[Note: SELECT operator does not show any result, the projection operator must be called before the selection operator to generate or project the result. So, the correct syntax to generate the result is**:** ?(? (AGE>18)(STUDENT))]

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE
3	SUJIT	ROHTAK	9156253131	20

Projection Operator (?): Projection operator is used to project particular columns from a relation. Syntax:

? **(Column 1,Column 2….Column n)(Relation Name)**

Extract ROLL_NO and NAME from STUDENT relation given in Table 3

? **(ROLL_NO,NAME)(STUDENT)**

RESULT:

ROLL_NO	NAME
1	RAM
2	RAMESH
3	SUJIT
4	SURESH

Note: If the resultant relation after projection has duplicate rows, it will be removed. For Example $?_{(ADDRESS)}$(STUDENT) will remove one duplicate row with the value DELHI and return three rows.

Cross Product(X): Cross product is used to join two relations. For every row of Relation1, each row of Relation2 is concatenated. If Relation1 has m tuples and and Relation2 has n tuples, cross product of Relation1 and Relation2 will have m X n tuples. Syntax:

Relation1 X Relation2

To apply Cross Product on STUDENT relation given in Table 1 and STUDENT_SPORTS relation given in Table 2,

STUDENT X STUDENT_SPORTS

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE	ROLL_NO	SPORTS
1	RAM	DELHI	9455123451	18	1	Badminton
1	RAM	DELHI	9455123451	18	2	Cricket
1	RAM	DELHI	9455123451	18	2	Badminton
1	RAM	DELHI	9455123451	18	4	Badminton
2	RAMESH	GURGAON	9652431543	18	1	Badminton
2	RAMESH	GURGAON	9652431543	18	2	Cricket
2	RAMESH	GURGAON	9652431543	18	2	Badminton
2	RAMESH	GURGAON	9652431543	18	4	Badminton
3	SUJIT	ROHTAK	9156253131	20	1	Badminton
3	SUJIT	ROHTAK	9156253131	20	2	Cricket
3	SUJIT	ROHTAK	9156253131	20	2	Badminton
3	SUJIT	ROHTAK	9156253131	20	4	Badminton
4	SURESH	DELHI	9156768971	18	1	Badminton
4	SURESH	DELHI	9156768971	18	2	Cricket
4	SURESH	DELHI	9156768971	18	2	Badminton

4	SURESH	DELHI	9156768971	18	4	Badminton

Union (U): Union on two relations R1 and R2 can only be computed if R1 and R2 are **union compatible** (These two relations should have the same number of attributes and corresponding attributes in two relations have the same domain). Union operator when applied on two relations R1 and R2 will give a relation with tuples that are either in R1 or in R2. The tuples which are in both R1 and R2 will appear only once in the result relation. Syntax:

Relation1 U Relation2

Find the person who is either student or employees, we can use Union operators like:

STUDENT U EMPLOYEE

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI	9156768971	18
5	NARESH	HISAR	9782918192	22
6	SWETA	RANCHI	9852617621	21

Minus (-): Minus on two relations R1 and R2 can only be computed if R1 and R2 are **union compatible**. Minus operator when applied on two relations as R1-R2 will give a relation with tuples that are in R1 but not in R2. Syntax:

Relation1 - Relation2

Find the person who is a student but not an employee, we can use minus operator like:

STUDENT - EMPLOYEE

RESULT:

ROLL_NO	NAME	ADDRESS	PHONE	AGE
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20

Rename(?): Rename operator is used to giving another name to a relation. Syntax:

?(Relation2, Relation1)

To rename STUDENT relation to STUDENT1, we can use rename operator like:

?(STUDENT1, STUDENT)

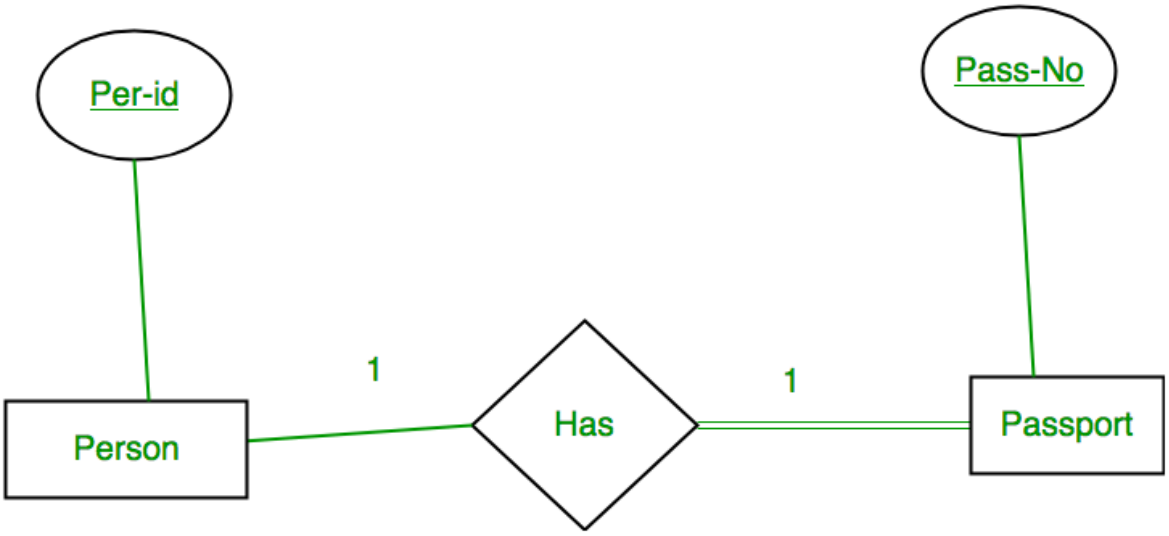
If you want to create a relation STUDENT_NAMES with ROLL_NO and NAME from STUDENT, it can be done using rename operator as:

?(STUDENT_NAMES, ?$_{(ROLL_NO, NAME)(STUDENT))}$

Mapping from ER Model to Relational Model

After designing the ER diagram of system, we need to convert it to Relational models which can directly be implemented by any RDBMS like Oracle, MySQL etc. In this article we will discuss how to convert ER diagram to Relational Model for different scenarios.

Case 1: Binary Relationship with 1:1 cardinality with total participation of an entity



A person has 0 or 1 passport number and Passport is always owned by 1 person. So it is 1:1 cardinality with full participation constraint from Passport.

First Convert each entity and relationship to tables. Person table corresponds to Person Entity with key as Per-Id. Similarly Passport table corresponds to Passport Entity with key as Pass-No. Has Table represents relationship between Person and Passport (Which person has which passport). So it will take attribute Per-Id from Person and Pass-No from Passport.

Person			**Has**			**Passport**	
Per-Id	**Other Person Attribute**		**Per-Id**	**Pass-No**		**Pass-No**	**Other PassportAttribute**
PR1	–		PR1	PS1		PS1	–

PR2	–		PR2	PS2		PS2	–
PR3	–						

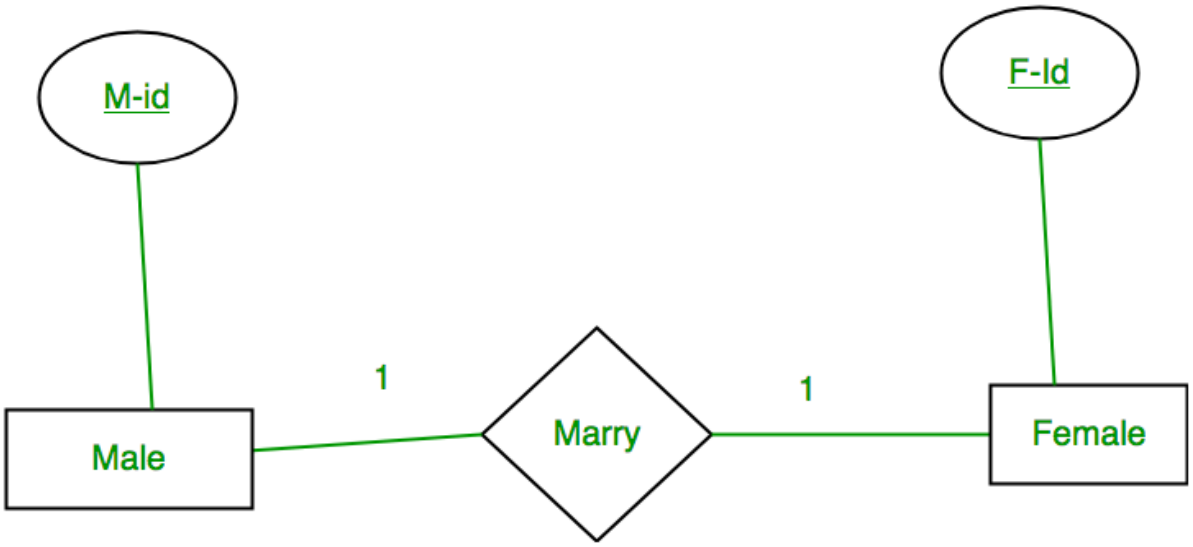
Table 1

As we can see from Table 1, each Per-Id and Pass-No has only one entry in Has Table. So we can merge all three tables into 1 with attributes shown in Table 2. Each Per-Id will be unique and not null. So it will be the key. Pass-No can't be key because for some person, it can be NULL.

Per-Id	Other Person Attribute	Pass-No	Other PassportAttribute

Table 2

Case 2: Binary Relationship with 1:1 cardinality and partial participation of both entities



A male marries 0 or 1 female and vice versa as well. So it is 1:1 cardinality with partial participation constraint from both. First Convert each entity and relationship to tables. Male table corresponds to Male Entity with key as M-Id. Similarly Female table corresponds to Female Entity with key as F-Id. Marry Table represents relationship between Male and Female

(Which Male marries which female). So it will take attribute M-Id from Male and F-Id from Female.

Male			**Marry**			**Female**	
M-Id	Other Male Attribute		M-Id	F-Id		F-Id	Other FemaleAttribute
M1	–		M1	F2		F1	–
M2	–		M2	F1		F2	–
M3	–					F3	–

Table 3

As we can see from Table 3, some males and some females do not marry. If we merge 3 tables into 1, for some M-Id, F-Id will be NULL. So there is no attribute which is always not NULL. So we can't merge all three tables into 1. We can convert into 2 tables. In table 4, M-Id who are married will have F-Id associated. For others, it will be NULL. Table 5 will have information of all females. Primary Keys have been underlined.

M-Id	Other Male Attribute	F-Id

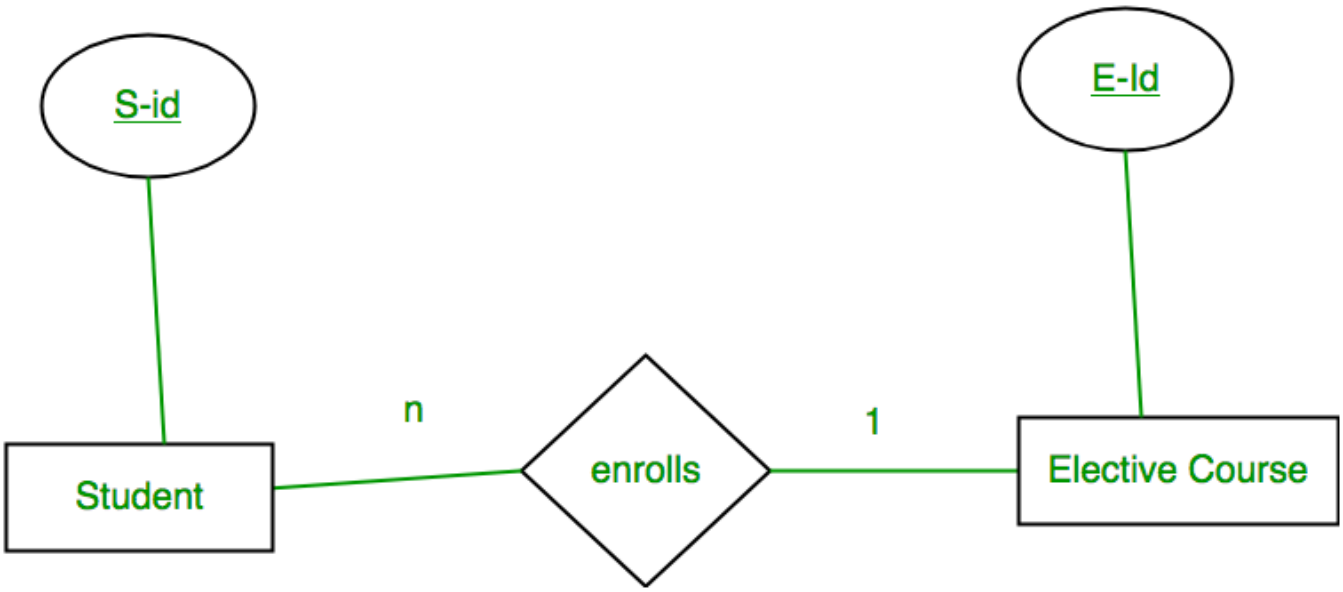
Table 4

F-Id	Other FemaleAttribute

Table 5

Note: Binary relationship with 1:1 cardinality will have 2 table if partial participation of both entities in the relationship. If atleast 1 entity has total participation, number of tables required will be 1.

Case 3: Binary Relationship with n: 1 cardinality



In this scenario, every student can enroll only in one elective course but for an elective course there can be more than one student. First Convert each entity and relationship to tables. Student table corresponds to Student Entity with key as S-Id. Similarly Elective_Course table corresponds to Elective_Course Entity with key as E-Id. Enrolls Table represents relationship between Student and Elective_Course (Which student enrolls in which course). So it will take attribute S-Id from Student and E-Id from Elective_Course.

Student		**Enrolls**		**Elective_Course**	
S-Id	Other Student Attribute	S-Id	E-Id	E-Id	Other Elective CourseAttribute
S1	–	S1	E1	E1	–
S2	–	S2	E2	E2	–
S3	–	S3	E1	E3	–
S4	–	S4	E1		

Table 6

As we can see from Table 6, S-Id is not repeating in Enrolls Table. So it can be considered as a key of Enrolls table. Both Student and Enrolls Table's key is same; we can merge it as a single table. The resultant tables are shown in Table 7 and Table 8. Primary Keys have been

underlined.

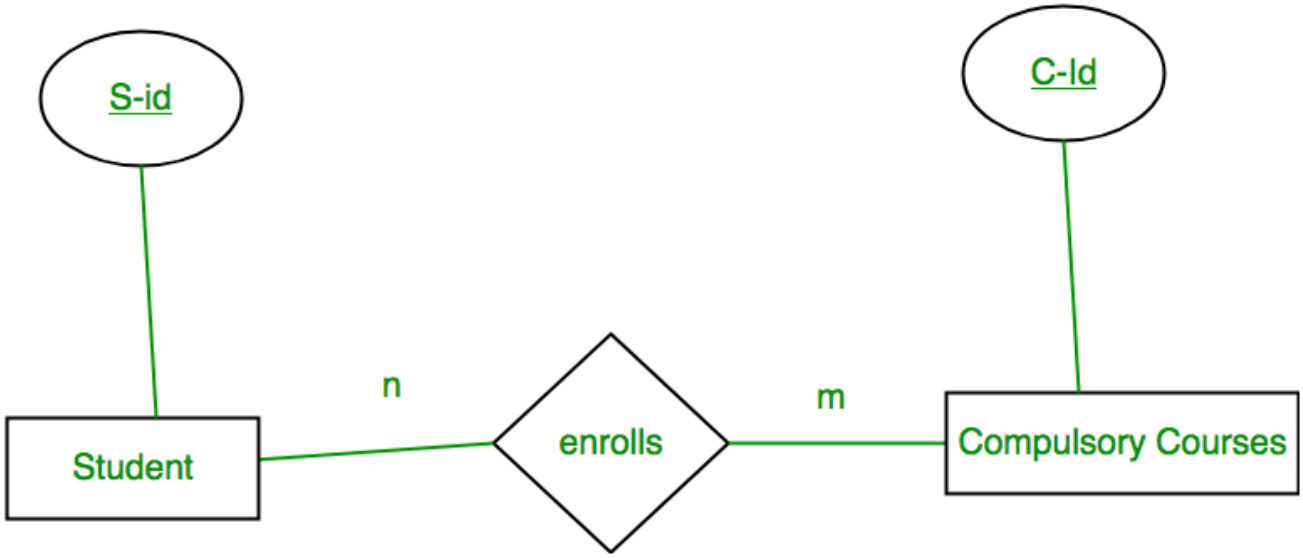
S-Id	Other Student Attribute	E-Id

Table 7

E-Id	Other Elective CourseAttribute

Table 8

Case 4: Binary Relationship with m: n cardinality



In this scenario, every student can enroll in more than 1 compulsory course and for a compulsory course there can be more than 1 student. First Convert each entity and relationship to tables. Student table corresponds to Student Entity with key as S-Id. Similarly Compulsory_Courses table corresponds to Compulsory Courses Entity with key as C-Id. Enrolls Table represents relationship between Student and Compulsory_Courses (Which student enrolls in which course). So it will take attribute S-Id from Person and C-Id from Compulsory_Courses.

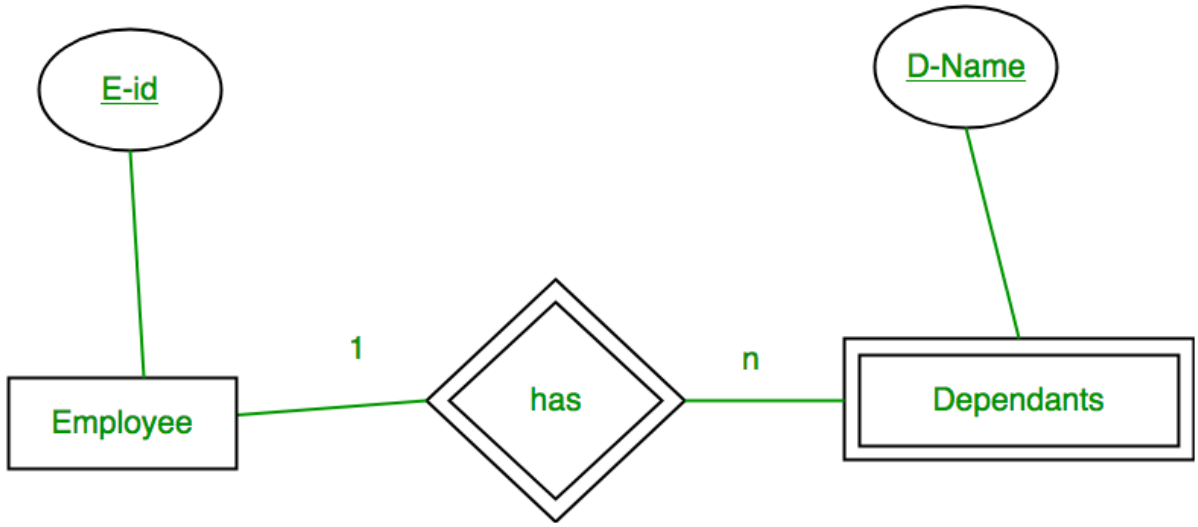
Student		**Enrolls**		**Compulsory_Courses**

S-Id	Other Student Attribute		S-Id	C-Id		C-Id	Other Compulsory CourseAttribute
S1	–		S1	C1		C1	–
S2	–		S1	C2		C2	–
S3	–		S3	C1		C3	–
S4	–		S4	C3		C4	–
			S4	C2			
			S3	C3			

Table 9

As we can see from Table 9, S-Id and C-Id both are repeating in Enrolls Table. But its combination is unique; so it can be considered as a key of Enrolls table. All tables' keys are different, these can't be merged. Primary Keys of all tables have been underlined.

Case 5: Binary Relationship with weak entity



In this scenario, an employee can have many dependents and one dependent can depend on one employee. A dependent does not have any existence without an employee (e.g; you as a child can be dependent of your father in his company). So it will be a weak entity and its participation will always be total. Weak Entity does not have key of its own. So its key will be combination of key of its identifying entity (E-Id of Employee in this case) and its partial key (D-Name).

First Convert each entity and relationship to tables. Employee table corresponds to Employee Entity with key as E-Id. Similarly Dependents table corresponds to Dependent Entity with key as

D-Name and E-Id. Has Table represents relationship between Employee and Dependents (Which employee has which dependents). So it will take attribute E-Id from Employee and D-Name from Dependents.

Employee			**Has**			**Dependents**		
E-Id	Other Employee Attribute		E-Id	D-Name		D-Name	E-Id	Other DependentsAttribute
E1	–		E1	RAM		RAM	E1	–
E2	–		E1	SRINI		SRINI	E1	–
E3	–		E2	RAM		RAM	E2	–
			E3	ASHISH		ASHISH	E3	–

Table 10

As we can see from Table 10, E-Id, D-Name is key for **Has** as well as Dependents Table. So we can merge these two into 1. So the resultant tables are shown in Tables 11 and 12. Primary Keys of all tables have been underlined.

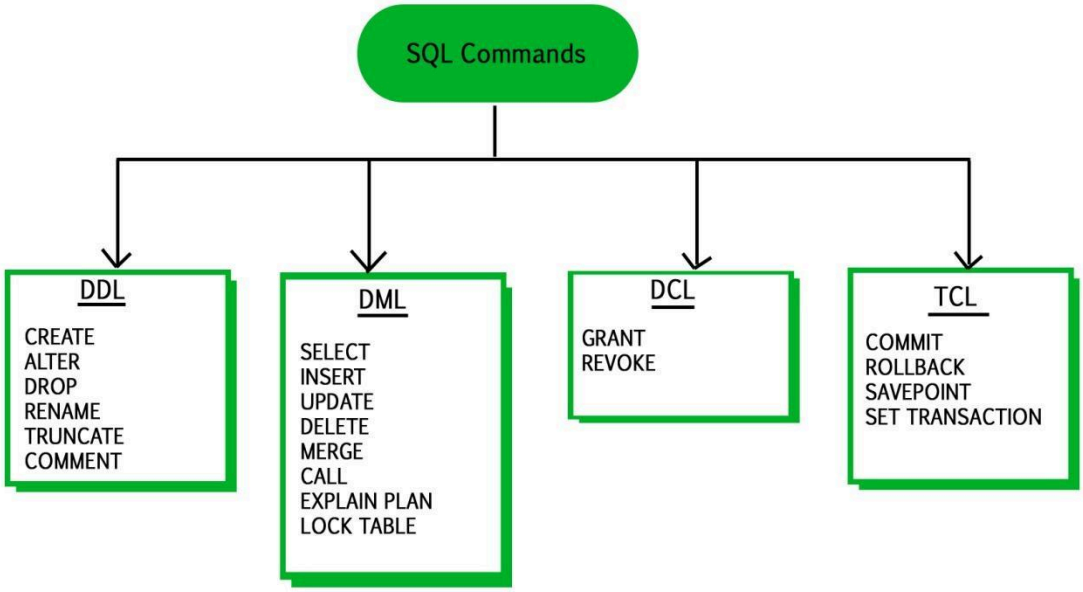
E-Id	Other Employee Attribute

Table 11

D-Name	E-Id	Other DependentsAttribute

SQL | DDL, DML, TCL and DCL

n this article, we'll be discussing Data Definition Language, Data Manipulation Language, Transaction Control Language, and Data Control Language.



DDL (Data Definition Language) :

Data Definition Language is used to define the database structure or schema. DDL is also used to specify additional properties of the data. The storage structure and access methods used by the database system by a set of statements in a special type of DDL called a data storage and definition language. These statements define the implementation details of the database schema, which are usually hidden from the users. The data values stored in the database must satisfy certain consistency constraints.

For example, suppose the university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to the test. Thus, the database system implements integrity constraints that can be tested with minimal overhead.

1. **Domain Constraints :** A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as the constraints on the values that it can take.
2. **Referential Integrity :** There are cases where we wish to ensure that a value appears in one relation for a given set of attributes also appear in a certain set of attributes in another relation i.e. Referential Integrity. For example, the department listed for each course must be one that actually exists.
3. **Assertions :** An assertion is any condition that the database must always satisfy. Domain constraints and Integrity constraints are special form of assertions.

4. **Authorization :** We may want to differentiate among the users as far as the type of access they are permitted on various data values in database. These differentiation are expressed in terms of Authorization. The most common being :
   read authorization – which allows reading but not modification of data ;
   insert authorization – which allow insertion of new data but not modification of existing data
   update authorization – which allows modification, but not deletion.

Some Commands:

```
CREATE : to create objects in database
ALTER : alters the structure of database
DROP : delete objects from database
RENAME : rename an objects
```

Following SQL DDL-statement defines the department table :

```
create table department
(dept_name  char(20),
building   char(15),
budget     numeric(12,2));
```

Execution of the above DDL statement creates the department table with three columns – dept_name, building, and budget; each of which has a specific datatype associated with it.

DML (Data Manipulation Language) :

DML statements are used for managing data with in schema objects.
DML are of two types –

1. **Procedural DMLs** : require a user to specify what data are needed and how to get those data.
2. **Declarative DMLs** (also referred as **Non-procedural DMLs**) : require a user to specify what data are needed without specifying how to get those data.
   Declarative DMLs are usually easier to learn and use than procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data.

Some Commands :

```
SELECT: retrieve data from the database
INSERT: insert data into a table
UPDATE: update existing data within a table
DELETE: deletes all records from a table, space for the records remain
```

Example of SQL query that finds the names of all instructors in the History department :

```
select instructor.name
```

```
from instructor
where instructor.dept_name = 'History';
```

The query specifies that those rows from the table instructor where the dept_name is History must be retrieved and the name attributes of these rows must be displayed.

TCL (Transaction Control Language) :

Transaction Control Language commands are used to manage transactions in the database. These are used to manage the changes made by DML-statements. It also allows statements to be grouped together into logical transactions.

Examples of TCL commands –

```
COMMIT: Commit command is used to permanently save any transaction
into the database.
ROLLBACK: This command restores the database to last committed state.
It is also used with savepoint command to jump to a savepoint
in a transaction.
SAVEPOINT: Savepoint command is used to temporarily save a transaction so
that you can rollback to that point whenever necessary.
```

DCL (Data Control Language) :

A Data Control Language is a syntax similar to a computer programming language used to control access to data stored in a database (Authorization). In particular, it is a component of Structured Query Language (SQL).

Examples of DCL commands :

```
GRANT: allow specified users to perform specified tasks.
REVOKE: cancel previously granted or denied permissions.
```

The operations for which privileges may be granted to or revoked from a user or role apply to both the Data definition language (DDL) and the Data manipulation language (DML), and may include CONNECT, SELECT, INSERT, UPDATE, DELETE, EXECUTE and USAGE.

In the Oracle database, executing a DCL command issues an implicit commit.

SQL SELECT Query

The **SQL SELECT Statement** retrieves data from a database.

SELECT Statement in SQL

The SELECT statement in SQL is used to fetch or retrieve data from a database. It allows users to access the data and retrieve specific data based on specific conditions.
We can fetch either the entire table or according to some specified rules. The data returned is stored in a result table. This result table is also called the **result set.** With the SELECT clause of a SELECT command statement, we specify the columns that we want to be displayed in the query result and, optionally, which column headings we prefer to see above the result table.
The SELECT clause is the first clause and is one of the last clauses of the select statement that the database server evaluates. The reason for this is that before we can determine what to include in the final result set, we need to know all of the possible columns that could be included in the final result set.

Syntax
The syntax for the SELECT statement is:
SELECT column1,column2…. FROM table_name ;

SELECT Statement Example
Let's look at some examples of the SQL SELECT statement, to understand it better.
Let's create a table which will be used in examples:
CREATE TABLE:

```
CREATE TABLE Customer(
   CustomerID INT PRIMARY KEY,
   CustomerName VARCHAR(50),
   LastName VARCHAR(50),
   Country VARCHAR(50),
   Age int(2),
  Phone int(10)
);
-- Insert some sample data into the Customers table
INSERT INTO Customer (CustomerID, CustomerName, LastName, Country, Age, Phone)
VALUES (1, 'Shubham', 'Thakur', 'India','23','xxxxxxxxxx'),
      (2, 'Aman ', 'Chopra', 'Australia','21','xxxxxxxxxx'),
      (3, 'Naveen', 'Tulasi', 'Sri lanka','24','xxxxxxxxxx'),
      (4, 'Aditya', 'Arpan', 'Austria','21','xxxxxxxxxx'),
      (5, 'Nishant. Salchichas S.A.', 'Jain', 'Spain','22','xxxxxxxxxx');
```

Output:

CustomerID	CustomerName	LastName	Country	Age	Phone
1	Shubham	Thakur	India	23	xxxxxxxxxx
2	Aman	Chopra	Australia	21	xxxxxxxxxx
3	Naveen	Tulasi	Sri lanka	24	xxxxxxxxxx
4	Aditya	Arpan	Austria	21	xxxxxxxxxx
5	Nishant. Salchichas S.A.	Jain	Spain	22	xxxxxxxxxx

Retrieve Data using SELECT Query

In this example, we will fetch CustomerName, LastName from the table Customer:

Query:

```
SELECT CustomerName, LastName FROM Customer;
```

Output:

CustomerName	LastName
Shubham	Thakur
Aman	Chopra
Naveen	Tulasi
Aditya	Arpan
Nishant. Salchichas S.A.	Jain

Fetch All Table using SELECT Statement

In this example, we will fetch all the fields from the table Customer:

Query:

```
SELECT * FROM Customer;
```

Output:

CustomerID	CustomerName	LastName	Country	Age	Phone
1	Shubham	Thakur	India	23	xxxxxxxxxx
2	Aman	Chopra	Australia	21	xxxxxxxxxx
3	Naveen	Tulasi	Sri lanka	24	xxxxxxxxxx
4	Aditya	Arpan	Austria	21	xxxxxxxxxx
5	Nishant. Salchichas S.A.	Jain	Spain	22	xxxxxxxxxx

SELECT Statement with WHERE Clause

Suppose we want to see table values with specific conditions then **WHERE Clause** is used with select statement.

Query:

```
SELECT CustomerName FROM Customer where Age = '21';
```

Output:

CustomerName
Aman
Aditya

SQL SELECT Statement with GROUP BY Clause

In this example, we will use SELECT statement with **GROUP BY** Clause

Query:

```
SELECT COUNT (item), Customer_id FROM Orders GROUP BY order_id;
```

Output:

COUNT (item)	customer_id
1	4
1	4
1	3
1	1
1	2

SELECT Statement with HAVING Clause
Consider the following database for **HAVING Clause**:

Results Messages

	EmployeeId	Name	Gender	Salary	Department	Experience
1	1	Rachit	M	50000	Engineering	6 year
2	2	Shobit	M	37000	HR	3 year
3	3	Isha	F	56000	Sales	7 year
4	4	Devi	F	43000	Management	4 year
5	5	Akhil	M	90000	Engineering	15 year

Query:

```
SELECT Department, sum(Salary) as Salary
FROM employee
GROUP BY department
HAVING SUM(Salary) >= 50000;
```

Output:

Results Messages

	Department	Salary
1	Engineering	140000
2	Sales	56000

SELECT Statement with ORDER BY clause in SQL

In this example, we will use SELECT Statement with **ORDER BY** clause

Query:

```
SELECT * FROM Customer ORDER BY Age DESC;
```

Output:

CustomerID	CustomerName	LastName	Country	Age	Phone
3	Naveen	Tulasi	Sri lanka	24	xxxxxxxxxx
1	Shubham	Thakur	India	23	xxxxxxxxxx
5	Nishant. Salchichas S.A.	Jain	Spain	22	xxxxxxxxxx
2	Aman	Chopra	Australia	21	xxxxxxxxxx
4	Aditya	Arpan	Austria	21	xxxxxxxxxx

Important Points with SQL SELECT Statement

- *It is used to access records from one or more database tables and views.*
- *The SELECT statement retrieves selected data based on specified conditions.*
- *The result of a SELECT statement is stored in a result set or result table.*
- *The SELECT statement can be used to access specific columns or all columns from a table.*
- *It can be combined with clauses like WHERE, GROUP BY, HAVING, and ORDER BY for more refined data retrieval.*
- *The SELECT statement is versatile and allows users to fetch data based on various criteria efficiently.*

SQL - Constraints

SQL Constraints

SQL Constraints are the rules applied to a data columns or the complete table to limit the type of data that can go into a table. When you try to perform any INSERT, UPDATE, or DELETE operation on the table, RDBMS will check whether that data violates any existing constraints and if there is any violation between the defined constraint and the data action, it aborts the operation and returns an error.

We can define a column level or a table level constraints. The column level constraints are applied only to one column, whereas the table level constraints are applied to the whole table.

SQL Create Constraints

We can create constraints on a table at the time of a table creation using the CREATE TABLE statement, or after the table is created, we can use the ALTER TABLE statement to create or delete table constraints.

```
CREATE TABLE table_name (
column1 datatype constraint,
column2 datatype constraint,
....
columnN datatype constraint
);
```

Different RDBMS allows to define different constraints. This tutorial will discuss about 7 most important constraints available in MySQL.

NOT NULL Constraint

When applied to a column, NOT NULL constraint ensure that a column cannot have a NULL value. Following is the example to create a NOT NULL constraint:

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

Check further detail on **NOT NULL Constraint**

UNIQUE Key Constraint

When applied to a column, UNIQUE Key constraint ensure that a column accepts only UNIQUE values. Following is the example to create a UNIQUE Key constraint on column ID. Once this constraint is created, column ID can't be null and it will accept only UNIQUE values.

```
CREATE TABLE CUSTOMERS (
ID INT NOT NULL UNIQUE,
```

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

Check further detail on **Unique Key Constraint**

DEFAULT Value Constraint

When applied to a column, DEFAULT Value constraint provides a default value for a column when none is specified. Following is the example to create a DEFAULT constraint on column NAME. Once this constraint is created, column NAME will set to "Not Available" value if NAME is not set to a value.

```
CREATE TABLE CUSTOMERS (
ID INT NOT NULL UNIQUE,
NAME VARCHAR (20) DEFAULT 'Not Available',
AGE INT NOT NULL,
ADDRESS CHAR (25),
SALARY DECIMAL (18, 2)
);
```

Check further detail on **DEFAULT Value Constraint**

PRIMARY Key Constraint

When applied to a column, PRIMARY Key constraint ensure that a column accepts only UNIQUE value and there can be a single PRIMARY Key on a table but multiple columns can constituet a PRIMARY Key. Following is the example to create a PRIMARY Key constraint on column ID. Once this constraint is created, column ID can't be null and it will accept only unique values.

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

Check further detail on **PRIMARY Key Constraint**

FOREIGN Key Constraint

FOREIGN Key constraint maps with a column in another table and uniquely identifies a row/record in that table. Following is an example to create a foreign key constraint on column ID available in CUSTOMERS table as shown in the statement below −

```
CREATE TABLE ORDERS (
ID INT NOT NULL,
DATE DATETIME,
CUSTOMER_ID INT FOREIGN KEY REFERENCES CUSTOMERS(ID),
AMOUNT DECIMAL,
PRIMARY KEY (ID)
);
```

Check further detail on **FOREIGN Key Constraint**

CHECK Value Constraint

When applied to a column, CHECK Value constraint works like a validation and it is used to check the validity of the data entered into the particular column of the table. table and uniquely identifies a row/record in that table. Following is an example to create a CHECK validation on AGE column which will not accept if its value is below to 18.

```
CREATE TABLE CUSTOMERS(
ID         INT NOT NULL,
NAME       VARCHAR (20) NOT NULL,
AGE        INT NOT NULL CHECK(AGE>=18),
ADDRESS    CHAR (25),
SALARY     DECIMAL (18, 2),
PRIMARY KEY (ID)
);
```

Check further detail on **CHECK Value Constraint**

INDEX Constraint

The INDEX constraints are created to speed up the data retrieval from the database. An Index can be created by using a single or group of columns in a table. A table can have a single PRIMARY Key but can have multiple INDEXES. An Index can be Unique or Non Unique based on requirements. Following is an example to create an Index on Age Column of the CUSTOMERS table.

```
CREATE INDEX idx_age ON CUSTOMERS ( AGE );
```

Check further detail on **INDEX Constraint**

Dropping SQL Constraints

Any constraint that you have defined can be dropped using the **ALTER TABLE** command with the DROP CONSTRAINT option. For example, to drop the primary key constraint from the CUSTOMERS table, you can use the following command.

```
ALTER TABLE CUSTOMERS DROP CONSTRAINT PRIMARY KEY;
```

Some RDBMS allow you to disable constraints instead of permanently dropping them from the database, which you may want to temporarily disable the constraints and then enable them later.

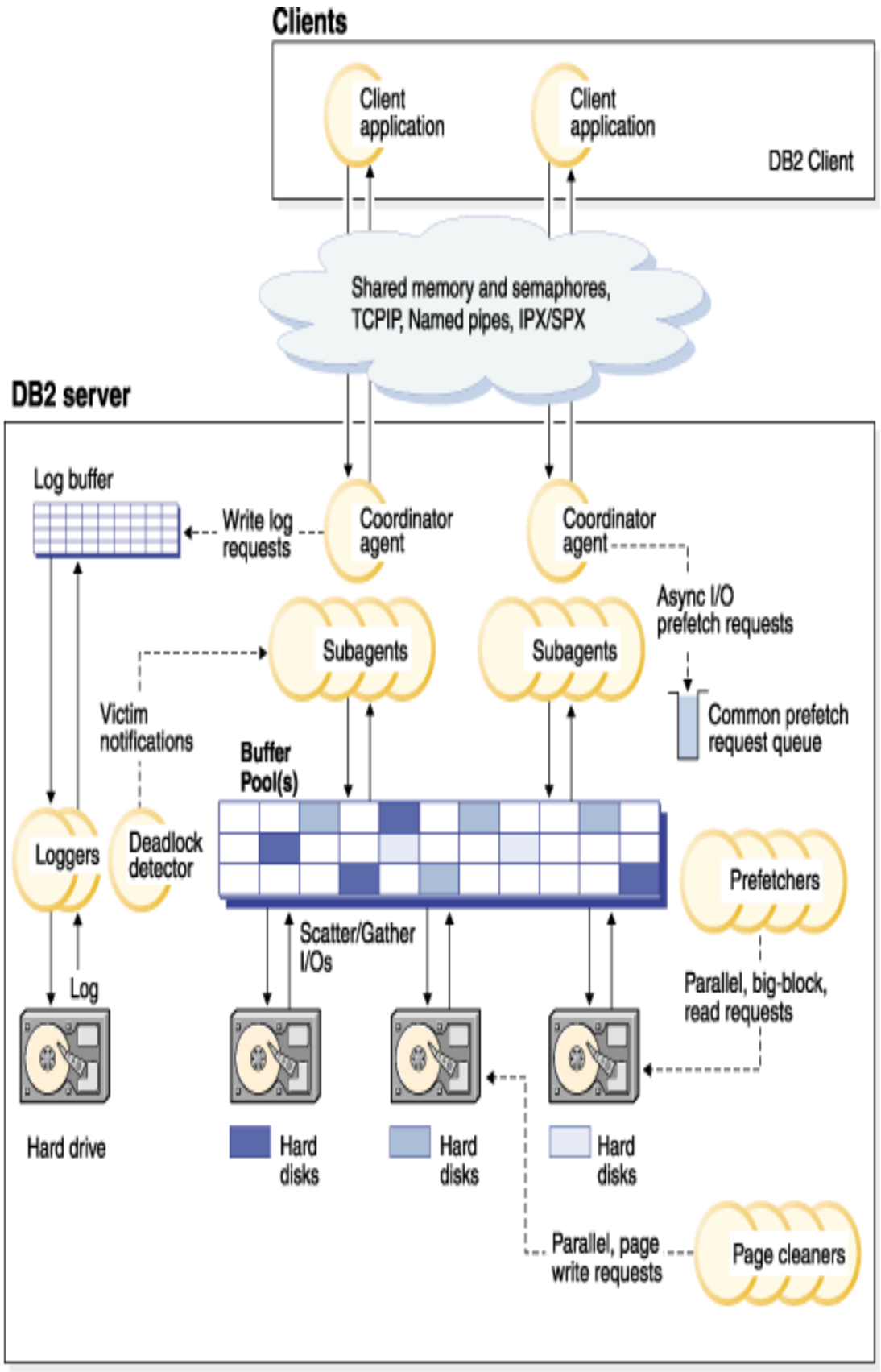
Data Integrity Constraints

Data integrity constraints are used to ensure the overall accuracy, completeness, and consistency of data. Now a days data integrity also refers to the data safety in regard to regulatory compliance, such as GDPR compliance etc.

Data integrity is handled in a relational database through the concept of referential integrity. There are many types of integrity constraints that play a role in **Referential Integrity (RI)**. These constraints include Primary Key, Foreign Key, Unique Constraints and other constraints which are mentioned above.

Db2 architecture and process overview

On the client side, local or remote applications are linked with the Db2 client library. Local clients communicate using shared memory and semaphores; remote clients use a protocol, such as named pipes (NPIPE) or TCP/IP. On the server side, activity is controlled by engine dispatchable units (EDUs).


Clients
Client application
Client application
DB2 Client
Shared memory and semaphores, TCPIP, Named pipes, IPX/SPX
DB2 server
Log buffer
Write log requests
Coordinator agent
Coordinator agent
Async I/O prefetch requests
Subagents
Subagents
Victim notifications
Common prefetch request queue
Buffer Pool(s)
Loggers
Deadlock detector
Prefetchers
Scatter/Gather I/Os
Log
Parallel, big-block, read requests
Hard drive
Hard disks
Hard disks
Hard disks
Parallel, page write requests
Page cleaners

EDUs are shown as circles or groups of circles.

EDUs are implemented as threads on all platforms. Db2 agents are the most common type of EDU. These agents perform most of the SQL and XQuery processing on behalf of applications. Prefetchers and page cleaners are other common EDUs.

A set of subagents might be assigned to process client application requests. Multiple subagents can be assigned if the machine on which the server resides has multiple processors or is part of a partitioned database environment. For example, in a symmetric multiprocessing (SMP) environment, multiple SMP subagents can exploit multiple processors.

All agents and subagents are managed by a pooling algorithm that minimizes the creation and destruction of EDUs.

Buffer pools are areas of database server memory where pages of user data, index data, and catalog data are temporarily moved and can be modified. Buffer pools are a key determinant of database performance, because data can be accessed much faster from memory than from disk.

The configuration of buffer pools, as well as prefetcher and page cleaner EDUs, controls how quickly data can be accessed by applications.

- *Prefetchers* retrieve data from disk and move it into a buffer pool before applications need the data. For example, applications that need to scan through large volumes of data would have to wait for data to be moved from disk into a buffer pool if there were no data prefetchers. Agents of the application send asynchronous read-ahead requests to a common prefetch queue. As prefetchers become available, they implement those requests by using big-block or scatter-read input operations to bring the requested pages from disk into the buffer pool. If you have multiple disks for data storage, the data can be striped across those disks. Striping enables the prefetchers to use multiple disks to retrieve data simultaneously.
- *Page cleaners* move data from a buffer pool back to disk. Page cleaners are background EDUs that are independent of the application agents. They look for pages that have been modified, and write those changed pages out to disk. Page cleaners ensure that there is room in the buffer pool for pages that are being retrieved by prefetchers.

Without the independent prefetchers and page cleaner EDUs, the application agents would have to do all of the reading and writing of data between a buffer pool and disk storage.

MODULE 4

Relational Model in DBMS



E.F. Codd proposed the relational Model to model data in the form of relations or tables. After designing the conceptual model of the Database using ER diagram, we need to convert the conceptual model into a relational model which can be implemented using any RDBMS language like Oracle SQL, MySQL, etc. So we will see what the Relational Model is.
The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as relations. The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.

What is the Relational Model?
The relational model represents how data is stored in Relational Databases. A relational database consists of a collection of tables, each of which is assigned a unique name. Consider a relation STUDENT with attributes ROLL_NO, NAME, ADDRESS, PHONE, and AGE shown in the table.

ROLL_NO	NAME	ADDRESS	PHONE	AGE
1	RAM	DELHI	9455123451	18
2	RAMESH	GURGAON	9652431543	18
3	SUJIT	ROHTAK	9156253131	20
4	SURESH	DELHI		18

Important Terminologies

- **Attribute:** Attributes are the properties that define an entity. e.g.; **ROLL_NO**, **NAME, ADDRESS**
- **Relation Schema:** A relation schema defines the structure of the relation and represents the name of the relation with its attributes. e.g.; STUDENT (ROLL_NO, NAME, ADDRESS, PHONE, and AGE) is the relation schema for STUDENT. If a schema has more than 1 relation, it is called Relational Schema.

- **Tuple:** Each row in the relation is known as a tuple. The above relation contains 4 tuples, one of which is shown as:

1	RAM	DELHI	9455123451	18

- **Relation Instance:** The set of tuples of a relation at a particular instance of time is called a relation instance. Table 1 shows the relation instance of STUDENT at a particular time. It can change whenever there is an insertion, deletion, or update in the database.
- **Degree:** The number of attributes in the relation is known as the degree of the relation. The **STUDENT** relation defined above has degree 5.
- **Cardinality:** The number of tuples in a relation is known as cardinality. The **STUDENT** relation defined above has cardinality 4.
- **Column:** The column represents the set of values for a particular attribute. The column **ROLL NO** is extracted from the relation STUDENT.

ROLL_NO
1
2
3
4

- **NULL Values:** The value which is not known or unavailable is called a NULL value. It is represented by blank space. e.g.; PHONE of STUDENT having ROLL_NO 4 is NULL.
- **Relation Key:** These are basically the keys that are used to identify the rows uniquely or also help in identifying tables. These are of the following types.
    - Primary Key
    - Candidate Key
    - Super Key
    - Foreign Key
    - Alternate Key
    - Composite Key

Constraints in Relational Model

While designing the Relational Model, we define some conditions which must hold for data present in the database are called Constraints. These constraints are checked before performing any operation (insertion, deletion, and updation) in the database. If there is a violation of any of the constraints, the operation will fail.

Domain Constraints

These are attribute-level constraints. An attribute can only take values that lie inside the domain range. e.g.; If a constraint AGE>0 is applied to STUDENT relation, inserting a negative value of AGE will result in failure.

Key Integrity

Every relation in the database should have at least one set of attributes that defines a tuple uniquely. Those set of attributes is called keys. e.g.; ROLL_NO in STUDENT is key. No two students can have the same roll number. So a key has two properties:

- It should be unique for all tuples.
- It can't have NULL values.

Referential Integrity

When one attribute of a relation can only take values from another attribute of the same relation or any other relation, it is called referential integrity. Let us suppose we have 2 relations

Table Student

ROLL_NO	NAME	ADDRESS	PHONE	AGE	BRANCH_CODE
1	RAM	DELHI	9455123451	18	CS
2	RAMESH	GURGAON	9652431543	18	CS
3	SUJIT	ROHTAK	9156253131	20	ECE
4	SURESH	DELHI		18	IT

Table Branch

BRANCH_CODE	BRANCH_NAME
CS	COMPUTER SCIENCE
IT	INFORMATION TECHNOLOGY
ECE	ELECTRONICS AND COMMUNICATION ENGINEERING
CV	CIVIL ENGINEERING

BRANCH_CODE of STUDENT can only take the values which are present in BRANCH_CODE of BRANCH which is called referential integrity constraint. The relation which is referencing another relation is called REFERENCING RELATION (STUDENT in this case) and the relation to which other relations refer is called REFERENCED RELATION (BRANCH in this case).

Anomalies in the Relational Model

An anomaly is an irregularity or something which deviates from the expected or normal state. When designing databases, we identify three types of anomalies: Insert, Update, and Delete.

Insertion Anomaly in Referencing Relation

We can't insert a row in REFERENCING RELATION if referencing attribute's value is not present in the referenced attribute value. e.g.; Insertion of a student with BRANCH_CODE 'ME' in STUDENT relation will result in an error because 'ME' is not present in BRANCH_CODE of BRANCH.

Deletion/ Updation Anomaly in Referenced Relation:

We can't delete or update a row from REFERENCED RELATION if the value of REFERENCED ATTRIBUTE is used in the value of REFERENCING ATTRIBUTE. e.g; if we try to delete a tuple from BRANCH having BRANCH_CODE 'CS', it will result in an error because 'CS' is referenced by BRANCH_CODE of STUDENT, but if we try to delete the row from BRANCH with BRANCH_CODE CV, it will be deleted as the value is not been used by referencing relation. It can be handled by the following method:

On Delete Cascade

It will delete the tuples from REFERENCING RELATION if the value used by REFERENCING ATTRIBUTE is deleted from REFERENCED RELATION. e.g.; For, if we delete a row from BRANCH with BRANCH_CODE 'CS', the rows in STUDENT relation with BRANCH_CODE CS (ROLL_NO 1 and 2 in this case) will be deleted.

On Update Cascade

It will update the REFERENCING ATTRIBUTE in REFERENCING RELATION if the attribute value used by REFERENCING ATTRIBUTE is updated in REFERENCED RELATION. e.g;, if we update a row from BRANCH with BRANCH_CODE 'CS' to 'CSE', the rows in STUDENT relation with BRANCH_CODE CS (ROLL_NO 1 and 2 in this case) will be updated with BRANCH_CODE 'CSE'.

Super Keys

Any set of attributes that allows us to identify unique rows (tuples) in a given relationship is known as super keys. Out of these super keys, we can always choose a proper subset among these that can be used as a primary key. Such keys are known as Candidate keys. If there is a combination of two or more attributes that are being used as the primary key then we call it a Composite key.

Codd Rules in Relational Model

Edgar F Codd proposed the relational database model where he stated rules. Now these are known as Codd's Rules. For any database to be the perfect one, it has to follow the rules.
For more, refer to Codd Rules in Relational Model.

Advantages of the Relational Model

- **Simple model:** Relational Model is simple and easy to use in comparison to other languages.
- **Flexible:** Relational Model is more flexible than any other relational model present.
- **Secure:** Relational Model is more secure than any other relational model.
- **Data Accuracy:** Data is more accurate in the relational data model.
- **Data Integrity:** The integrity of the data is maintained in the relational model.
- **Operations can be Applied Easily:** It is better to perform operations in the relational model.

Disadvantages of the Relational Model

- Relational Database Model is not very good for large databases.
- Sometimes, it becomes difficult to find the relation between tables.
- Because of the complex structure, the response time for queries is high.

Characteristics of the Relational Model

- Data is represented in rows and columns called relations.
- Data is stored in tables having relationships between them called the Relational model.
- The relational model supports the operations like Data definition, Data manipulation, and Transaction management.
- Each column has a distinct name and they are representing attributes.
- Each row represents a single entity.

Types of Functional dependencies in DBMS



Prerequisite: Functional dependency and attribute closure
In a relational database management, functional dependency is a concept that specifies the relationship between two sets of attributes where one attribute determines the value of another attribute. It is denoted as **X → Y**, where the attribute set on the left side of the arrow, **X** is called **Determinant**, and **Y** is called the **Dependent**.
Functional dependencies are used to mathematically express relations among database entities and are very important to understand advanced concepts in Relational Database System and understanding problems in competitive exams like Gate.

Example:

roll_no	name	dept_name	dept_building
42	abc	CO	A4
43	pqr	IT	A3
44	xyz	CO	A4
45	xyz	IT	A3
46	mno	EC	B2
47	jkl	ME	B2

From the above table we can conclude some valid functional dependencies:

- roll_no → { name, dept_name, dept_building },→ Here, roll_no can determine values of fields name, dept_name and dept_building, hence a valid Functional dependency
- roll_no → dept_name , Since, roll_no can determine whole set of {name, dept_name, dept_building}, it can determine its subset dept_name also.

- dept_name → dept_building , Dept_name can identify the dept_building accurately, since departments with different dept_name will also have a different dept_building
- More valid functional dependencies: roll_no → name, {roll_no, name} → {dept_name, dept_building}, etc.

Here are some invalid functional dependencies:

- name → dept_name Students with the same name can have different dept_name, hence this is not a valid functional dependency.
- dept_building → dept_name There can be multiple departments in the same building. Example, in the above table departments ME and EC are in the same building B2, hence dept_building → dept_name is an invalid functional dependency.
- More invalid functional dependencies: name → roll_no, {name, dept_name} → roll_no, dept_building → roll_no, etc.

Armstrong's axioms/properties of functional dependencies:

1. **Reflexivity:** If Y is a subset of X, then X→Y holds by reflexivity rule
   Example, {roll_no, name} → name is valid.
2. **Augmentation:** If X → Y is a valid dependency, then XZ → YZ is also valid by the augmentation rule.
   Example, {roll_no, name} → dept_building is valid, hence {roll_no, name, dept_name} → {dept_building, dept_name} is also valid.
3. **Transitivity**: If X → Y and Y → Z are both valid dependencies, then X→Z is also valid by the Transitivity rule.
   Example, roll_no → dept_name & dept_name → dept_building, then roll_no → dept_building is also valid.

Types of Functional Dependencies in DBMS

1. Trivial functional dependency
2. Non-Trivial functional dependency
3. Multivalued functional dependency
4. Transitive functional dependency

1. Trivial Functional Dependency

In **Trivial Functional Dependency**, a dependent is always a subset of the determinant. i.e. If **X → Y** and **Y is the subset of X**, then it is called trivial functional dependency

Example:

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18

Here, **{roll_no, name} → name** is a trivial functional dependency, since the dependent **name** is a subset of determinant set **{roll_no, name}.** Similarly, **roll_no → roll_no** is also an example of trivial functional dependency.

2. Non-trivial Functional Dependency

In **Non-trivial functional dependency**, the dependent is strictly not a subset of the determinant. i.e. If **X → Y** and **Y is not a subset of X**, then it is called Non-trivial functional dependency.

Example:

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18

Here, **roll_no → name** is a non-trivial functional dependency, since the dependent **name** is **not a subset of** determinant **roll_no.** Similarly, **{roll_no, name} → age** is also a non-trivial functional dependency, since **age** is **not a subset of {roll_no, name}**

3. Multivalued Functional Dependency

In **Multivalued functional dependency**, entities of the dependent set are **not dependent on each other.** i.e. If **a → {b, c}** and there exists **no functional dependency** between **b and c**, then it is called a **multivalued functional dependency.**

For example,

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18
45	abc	19

Here, **roll_no → {name, age}** is a multivalued functional dependency, since the dependents **name** & **age** are **not dependent** on each other(i.e. **name → age** or **age → name doesn't exist !**)

4. Transitive Functional Dependency

In transitive functional dependency, dependent is indirectly dependent on determinant. i.e. If **a → b** & **b → c**, then according to axiom of transitivity, **a → c**. This is a **transitive functional dependency.**

For example,

enrol_no	**name**	**dept**	**building_no**
42	abc	CO	4
43	pqr	EC	2
44	xyz	IT	1
45	abc	EC	2

Here, **enrol_no → dept** and **dept → building_no.** Hence, according to the axiom of transitivity, **enrol_no → building_no** is a valid functional dependency. This is an indirect functional dependency, hence called Transitive functional dependency.

5. Fully Functional Dependency

In full functional dependency an attribute or a set of attributes uniquely determines another attribute or set of attributes. If a relation R has attributes X, Y, Z with the dependencies X->Y and X->Z which states that those dependencies are fully functional.

6. Partial Functional Dependency

In partial functional dependency a non key attribute depends on a part of the composite key, rather than the whole key. If a relation R has attributes X, Y, Z where X and Y are the composite key and Z is non key attribute. Then X->Z is a partial functional dependency in RBDMS.

Advantages of Functional Dependencies

Functional dependencies having numerous applications in the field of database management system. Here are some applications listed below:

1. Data Normalization

Data normalization is the process of organizing data in a database in order to minimize redundancy and increase data integrity. Functional dependencies play an important part in data normalization. With the help of functional dependencies we are able to identify the primary key, candidate key in a table which in turns helps in normalization.

2. Query Optimization

With the help of functional dependencies we are able to decide the connectivity between the tables and the necessary attributes need to be projected to retrieve the required data from the tables. This helps in query optimization and improves performance.

3. Consistency of Data

Functional dependencies ensures the consistency of the data by removing any redundancies or inconsistencies that may exist in the data. Functional dependency ensures that the changes made in one attribute does not affect inconsistency in another set of attributes thus it maintains the consistency of the data in database.

4. Data Quality Improvement

Functional dependencies ensure that the data in the database to be accurate, complete and updated. This helps to improve the overall quality of the data, as well as it eliminates errors and inaccuracies that might occur during data analysis and decision making, thus functional dependency helps in improving the quality of data in database.

Conclusion

Functional dependency is very important concept in database management system for ensuring the data consistency and accuracy. In this article we have discuss what is the concept behind functional dependencies and why they are important. The valid and invalid functional dependencies and the types of most important functional dependencies in RDBMS. We have also discussed about the advantages of FDs.

For more details you can refer Database Normalization and Difference between Fully and Partial Functional Dependency articles.

Types of Keys in Relational Model (Candidate, Super, Primary, Alternate and Foreign)



Keys are one of the basic requirements of a relational database model. It is widely used to identify the tuples(rows) uniquely in the table. We also use keys to set up relations amongst various columns and tables of a relational database.

Different Types of Database Keys

- Candidate Key
- Primary Key
- Super Key
- Alternate Key
- Foreign Key
- Composite Key

Candidate Key

The minimal set of attributes that can uniquely identify a tuple is known as a candidate key. For Example, STUD_NO in STUDENT relation.

- It is a minimal super key.
- It is a super key with no repeated data is called a candidate key.
- The minimal set of attributes that can uniquely identify a record.
- It must contain unique values.
- It can contain NULL values.
- Every table must have at least a single candidate key.
- A table can have multiple candidate keys but only one primary key.
- The value of the Candidate Key is unique and may be null for a tuple.
- There can be more than one candidate key in a relationship.

Example:

STUD_NO is the candidate key for relation STUDENT.

Table STUDENT

STUD_NO	SNAME	ADDRESS	PHONE
1	Shyam	Delhi	123456789
2	Rakesh	Kolkata	223365796
3	Suraj	Delhi	175468965

- The candidate key can be simple (having only one attribute) or composite as well.

Example:

{STUD_NO, COURSE_NO} is a composite
candidate key for relation STUDENT_COURSE.

Table STUDENT_COURSE

STUD_NO	TEACHER_NO	COURSE_NO
1	001	C001
2	056	C005

Note: In SQL Server a unique constraint that has a nullable column, **allows** the value '**null**' in that column **only once**. That's why the STUD_PHONE attribute is a candidate here, but can not be a 'null' value in the primary key attribute.

Primary Key

There can be more than one candidate key in relation out of which one can be chosen as the primary key. For Example, STUD_NO, as well as STUD_PHONE, are candidate keys for

relation STUDENT but STUD_NO can be chosen as the primary key (only one out of many candidate keys).

- It is a unique key.
- It can identify only one tuple (a record) at a time.
- It has no duplicate values, it has unique values.
- It cannot be NULL.
- Primary keys are not necessarily to be a single column; more than one column can also be a primary key for a table.

Example:

STUDENT table -> Student(STUD_NO, SNAME, ADDRESS, PHONE) , STUD_NO is a primary key

Table STUDENT

STUD_NO	SNAME	ADDRESS	PHONE
1	Shyam	Delhi	123456789
2	Rakesh	Kolkata	223365796
3	Suraj	Delhi	175468965

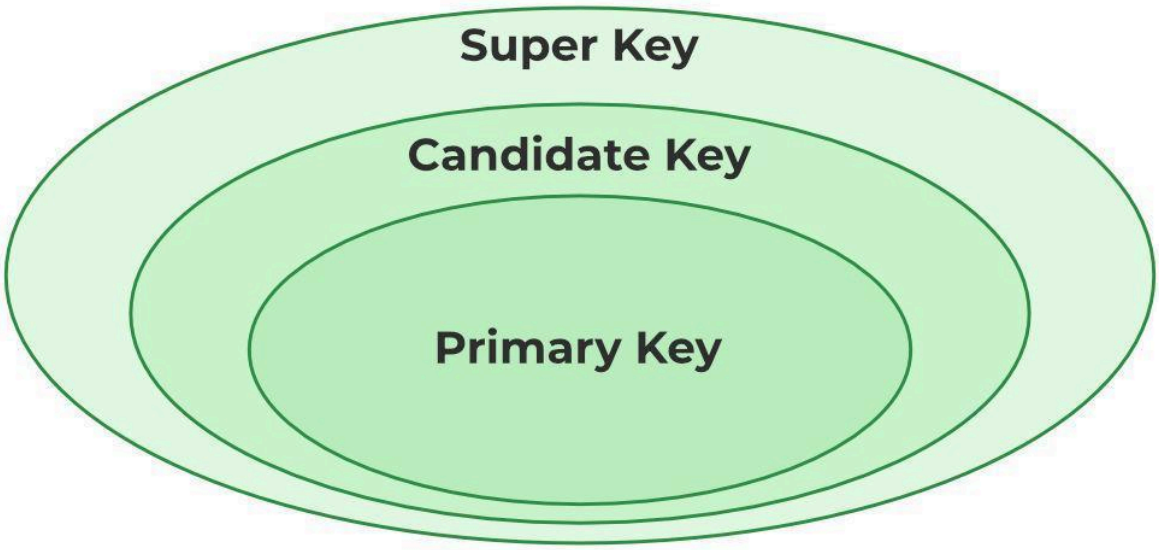
Super Key

The set of attributes that can uniquely identify a tuple is known as Super Key. For Example, STUD_NO, (STUD_NO, STUD_NAME), etc. A super key is a group of single or multiple keys that identifies rows in a table. It supports NULL values.

- Adding zero or more attributes to the candidate key generates the super key.
- A candidate key is a super key but vice versa is not true.
- Super Key values may also be NULL.

Example:

Consider the table shown above.
STUD_NO+PHONE is a super key**.**



Relation between Primary Key, Candidate Key, and Super Key

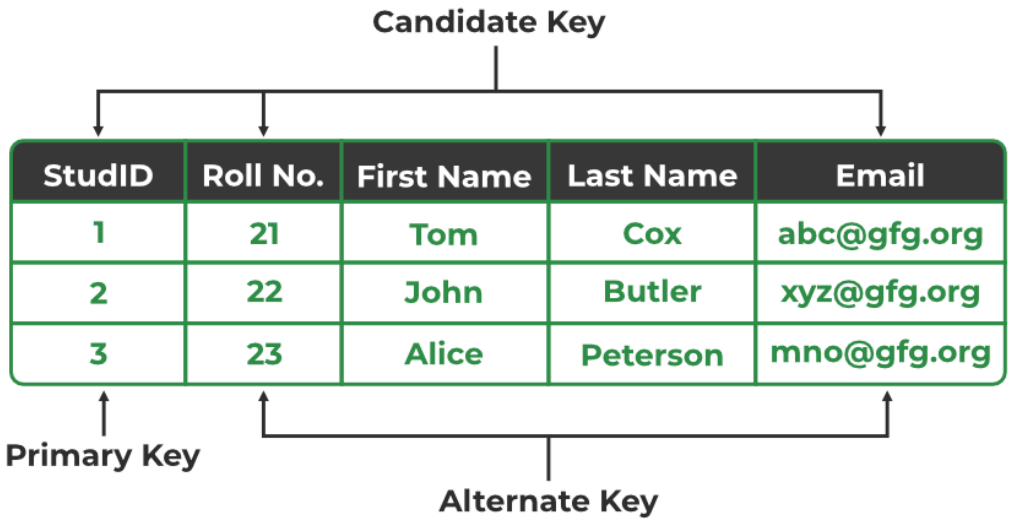
Alternate Key

The candidate key other than the primary key is called an alternate key.

- All the keys which are not primary keys are called alternate keys.
- It is a secondary key.
- It contains two or more fields to identify two or more records.
- These values are repeated.
- Eg:- SNAME, and ADDRESS is Alternate keys

Example:

Consider the table shown above.
STUD_NO, as well as PHONE both,
are candidate keys for relation STUDENT but
PHONE will be an alternate key
(only one out of many candidate keys).



Primary Key, Candidate Key, and Alternate Key

Foreign Key

If an attribute can only take the values which are present as values of some other attribute, it will be a foreign key to the attribute to which it refers. The relation which is being referenced is called referenced relation and the corresponding attribute is called referenced attribute. The referenced attribute of the referenced relation should be the primary key to it.

- It is a key it acts as a primary key in one table and it acts as secondary key in another table.
- It combines two or more relations (tables) at a time.
- They act as a cross-reference between the tables.
- For example, DNO is a primary key in the DEPT table and a non-key in EMP

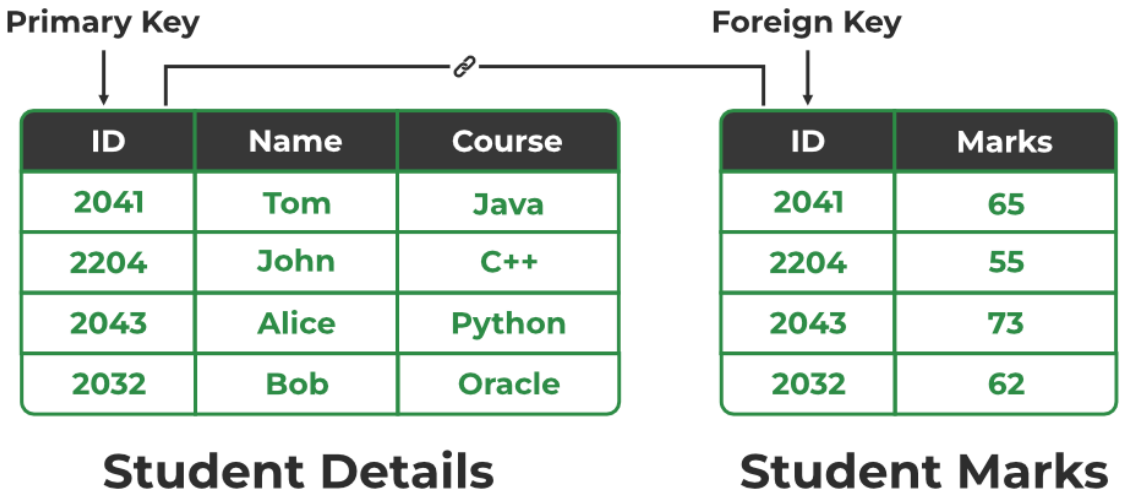
Example:

Refer Table STUDENT shown above.
STUD_NO in STUDENT_COURSE is a
foreign key to STUD_NO in STUDENT relation.

Table STUDENT_COURSE

STUD_NO	TEACHER_NO	COURSE_NO
1	005	C001
2	056	C005

It may be worth noting that, unlike the Primary Key of any given relation, Foreign Key can be NULL as well as may contain duplicate tuples i.e. it need not follow uniqueness constraint. For Example, STUD_NO in the STUDENT_COURSE relation is not unique. It has been repeated for the first and third tuples. However, the STUD_NO in STUDENT relation is a primary key and it needs to be always unique, and it cannot be null.



Relation between Primary Key and Foreign Key

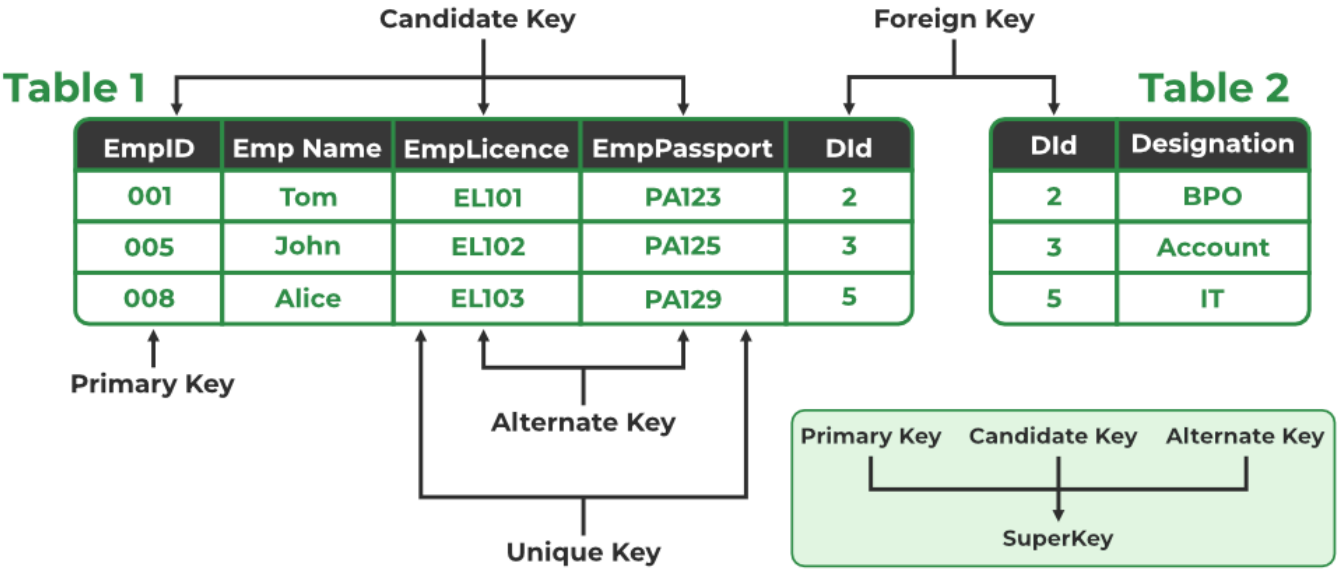
Composite Key

Sometimes, a table might not have a single column/attribute that uniquely identifies all the records of a table. To uniquely identify rows of a table, a combination of two or more columns/attributes can be used. It still can give duplicate values in rare cases. So, we need to find the optimal set of attributes that can uniquely identify rows in a table.

- It acts as a primary key if there is no primary key in a table
- Two or more attributes are used together to make a composite key.
- Different combinations of attributes may give different accuracy in terms of identifying the rows uniquely.
-

Example:

FULLNAME + DOB can be combined
together to access the details of a student.



Different Types of Keys

Conclusion

In conclusion, the relational model makes use of a number of keys: Candidate keys allow for distinct identification, the Primary key serves as the chosen identifier, Alternate keys offer other choices, and Foreign keys create vital linkages that guarantee data integrity between tables. The creation of strong and effective relational databases requires the thoughtful application of these keys.

Boyce-Codd Normal Form (BCNF)



First Normal Form, Second Normal Form, Third Normal Form

Application of the general definitions of 2NF and 3NF may identify additional redundancy caused by dependencies that violate one or more candidate keys. However, despite these additional constraints, dependencies can still exist that will cause redundancy to be present in 3NF relations. This weakness in 3NF resulted in the presentation of a stronger normal form called the **Boyce-Codd Normal Form (Codd, 1974)**.
Although, 3NF is an adequate normal form for relational databases, still, this (3NF) normal form may not remove 100% redundancy because of X−>Y functional dependency if X is not a candidate key of the given relation. This can be solved by Boyce-Codd Normal Form (BCNF).
Boyce-Codd Normal Form (BCNF)
Boyce–Codd Normal Form (BCNF) is based on functional dependencies that take into account all candidate keys in a relation; however, BCNF also has additional constraints compared with the general definition of 3NF.
Rules for BCNF
Rule 1: *The table should be in the 3rd Normal Form.*
Rule 2: *X should be a superkey for every functional dependency (FD) X−>Y in a given relation.*

Note: To test whether a relation is in BCNF, we identify all the determinants and make sure that they are candidate keys.
You came across a similar hierarchy known as the **Chomsky Normal Form** in the Theory of Computation. Now, carefully study the hierarchy above. It can be inferred that **every relation in BCNF is also in 3NF**. To put it another way, a relation in 3NF need not be in BCNF. Ponder over this statement for a while.
To determine the highest normal form of a given relation R with functional dependencies, the first step is to check whether the BCNF condition holds. If R is found to be in BCNF, it can be safely deduced that the relation is also in 3NF, 2NF, and 1NF as the hierarchy shows. The 1NF has the least restrictive constraint – it only requires a relation R to have atomic values in each tuple. The 2NF has a slightly more restrictive constraint.
The 3NF has a more restrictive constraint than the first two normal forms but is less restrictive than the BCNF. In this manner, the restriction increases as we traverse down the hierarchy.

Examples
Here, we are going to discuss some basic examples which let you understand the properties of BCNF. We will discuss multiple examples here.

Example 1

Let us consider the student database, in which data of the student are mentioned.

Stu_ID	Stu_Branch	Stu_Course	Branch_Number	Stu_Course_No
101	Computer Science & Engineering	DBMS	B_001	201
101	Computer Science & Engineering	Computer Networks	B_001	202

Stu_ID	Stu_Branch	Stu_Course	Branch_Number	Stu_Course_No
102	Electronics & Communication Engineering	VLSI Technology	B_003	401
102	Electronics & Communication Engineering	Mobile Communication	B_003	402

Functional Dependency of the above is as mentioned:

```
Stu_ID −> Stu_Branch
Stu_Course −> {Branch_Number, Stu_Course_No}
```

Candidate Keys of the above table are: **{Stu_ID, Stu_Course}**

Why this Table is Not in BCNF?

The table present above is not in BCNF, because as we can see that neither Stu_ID nor Stu_Course is a Super Key. As the rules mentioned above clearly tell that for a table to be in BCNF, it must follow the property that for functional dependency X−>Y, X must be in Super Key and here this property fails, that's why this table is not in BCNF.

How to Satisfy BCNF?

For satisfying this table in BCNF, we have to decompose it into further tables. Here is the full procedure through which we transform this table into BCNF. Let us first divide this main table into two tables **Stu_Branch** and **Stu_Course** Table.

Stu_Branch Table

Stu_ID	Stu_Branch
101	Computer Science & Engineering
102	Electronics & Communication Engineering

Candidate Key for this table: **Stu_ID**.

Stu_Course Table

Stu_Course	Branch_Number	Stu_Course_No
DBMS	B_001	201
Computer Networks	B_001	202
VLSI Technology	B_003	401
Mobile Communication	B_003	402

Candidate Key for this table: **Stu_Course**.

Stu_ID to Stu_Course_No Table

Stu_ID	Stu_Course_No
101	201
101	202
102	401
102	402

Candidate Key for this table: **{Stu_ID, Stu_Course_No}.**
After decomposing into further tables, now it is in BCNF, as it is passing the condition of Super Key, that in functional dependency X−>Y, X is a **Super Key.**

Example 2

Find the highest normal form of a relation R(A, B, C, D, E) with FD set as:

```
{ BC->D, AC->BE, B->E }
```

Explanation:

- **Step-1:** As we can see, (AC)+ ={A, C, B, E, D} but none of its subsets can determine all attributes of the relation, So AC will be the candidate key. A or C can't be derived from any other attribute of the relation, so there will be only 1 candidate key {AC}.
- **Step-2:** Prime attributes are those attributes that are part of candidate key {A, C} in this example and others will be non-prime {B, D, E} in this example.
- **Step-3:** The relation R is in 1st normal form as a relational DBMS does not allow multi-valued or composite attributes.

The relation is in 2nd normal form because BC->D is in 2nd normal form (BC is not a proper subset of candidate key AC) and AC->BE is in 2nd normal form (AC is candidate key) and B->E is in 2nd normal form (B is not a proper subset of candidate key AC).

The relation is **not** in 3rd normal form because in BC->D (neither BC is a super key nor D is a prime attribute) and in B->E (neither B is a super key nor E is a prime attribute) but to satisfy 3rd normal for, either LHS of an FD should be super key or RHS should be a prime attribute. So the highest normal form of relation will be the 2nd Normal form.
Note: A prime attribute cannot be transitively dependent on a key in BCNF relation.
Consider these functional dependencies of some relation R

```
AB ->C
C ->B
AB ->B
```

Suppose, it is known that the only candidate key of R is AB. A careful observation is required to conclude that the above dependency is a Transitive Dependency as the prime attribute B transitively depends on the key AB through C. Now, the first and the third FD are in BCNF as they both contain the candidate key (or simply KEY) on their left sides. The second dependency, however, is not in BCNF but is definitely in 3NF due to the presence of the prime attribute on the right side. So, the highest normal form of R is 3NF as all three FDs satisfy the necessary conditions to be in 3NF.

Example 3

For example consider relation R(A, B, C)

A -> BC,

B -> A

A and B both are super keys so the above relation is in BCNF.

Note: BCNF decomposition may always not be possible with dependency preserving, however, it always satisfies the lossless join condition. For example, relation R (V, W, X, Y, Z), with functional dependencies:

V, W -> X

Y, Z -> X

W -> Y

It would not satisfy dependency preserving BCNF decomposition.

Introduction of 4th and 5th Normal Form in DBMS



Two of the highest levels of database normalization are the fourth normal form (4NF) and the fifth normal form (5NF). Multivalued dependencies are handled by 4NF, whereas join dependencies are handled by 5NF.
If two or more independent relations are kept in a single relation or we can say multivalue dependency occurs when the presence of one or more rows in a table implies the presence of one or more other rows in that same table. Put another way, two attributes (or columns) in a table are independent of one another, but both depend on a third attribute. A **multivalued dependency** always requires at least three attributes because it consists of at least two attributes that are dependent on a third.
For a dependency A -> B, if for a single value of A, multiple values of B exist, then the table may have a multi-valued dependency. The table should have at least 3 attributes and B and C should be independent for A ->> B multivalued dependency.

Example:

Person	Mobile	Food_Likes
Mahesh	9893/9424	Burger/Pizza
Ramesh	9191	Pizza

Person	Mobile	Food_Likes

Person->-> mobile,
Person ->-> food_likes

This is read as “person multi determines mobile” and “person multi determines food_likes.” Note that a functional dependency is a special case of multivalued dependency. In a functional dependency X -> Y, every x determines exactly one y, never more than one.

Fourth Normal Form (4NF)

The Fourth Normal Form (4NF) is a level of database normalization where there are no non-trivial multivalued dependencies other than a candidate key. It builds on the first three normal forms (1NF, 2NF, and 3NF) and the Boyce-Codd Normal Form (BCNF). It states that, in addition to a database meeting the requirements of BCNF, it must not contain more than one multivalued dependency.

Properties

A relation R is in 4NF if and only if the following conditions are satisfied:

1. It should be in the Boyce-Codd Normal Form (BCNF).
2. The table should not have any Multi-valued Dependency.

A table with a multivalued dependency violates the normalization standard of the Fourth Normal Form (4NF) because it creates unnecessary redundancies and can contribute to inconsistent data. To bring this up to 4NF, it is necessary to break this information into two tables.

Example: Consider the database table of a class that has two relations R1 contains student ID(SID) and student name (SNAME) and R2 contains course id(CID) and course name (CNAME).

Table R1

SID	SNAME
S1	A
S2	B

Table R2

CID	CNAME
C1	C
C2	D

When their cross-product is done it resulted in multivalued dependencies.

Table R1 X R2

SID	SNAME	CID	CNAME
S1	A	C1	C
S1	A	C2	D
S2	B	C1	C
S2	B	C2	D

Multivalued dependencies (MVD) are:

SID->->CID; SID->->CNAME; SNAME->->CNAME

Join Dependency

Join decomposition is a further generalization of Multivalued dependencies. If the join of R1 and R2 over C is equal to relation R then we can say that a join dependency (JD) exists, where R1 and R2 are the decomposition R1(A, B, C) and R2(C, D) of a given relations R (A, B, C, D). Alternatively, R1 and R2 are a lossless decomposition of R. A JD ⋈ {R1, R2, …, Rn} is said to hold over a relation R if R1, R2, ….., Rn is a lossless-join decomposition. The *(A, B, C, D), (C, D) will be a JD of R if the join of joins attribute is equal to the relation R. Here, *(R1, R2, R3) is used to indicate that relation R1, R2, R3 and so on are a JD of R. Let R is a relation schema R1, R2, R3……..Rn be the decomposition of R. r(R) is said to satisfy join dependency if and only if

$$\bowtie_{i=1}^{n} \Pi_{R_i}(r) = r$$

Joint Dependency

Example:

Table R1

Company	Product
C1	Pendrive
C1	mic
C2	speaker
C2	speaker

Company->->Product

Table R2

Agent	Company
Aman	C1

Agent	Company
Aman	C2
Mohan	C1

Agent->->Company

Table R3

Agent	Product
Aman	Pendrive
Aman	Mic
Aman	speaker
Mohan	speaker

Agent->->Product

Table R1⋈R2⋈R3

Company	Product	Agent
C1	Pendrive	Aman
C1	mic	Aman
C2	speaker	speaker
C1	speaker	Aman

Agent->->Product

Fifth Normal Form/Projected Normal Form (5NF)

A relation R is in Fifth Normal Form if and only if everyone joins dependency in R is implied by the candidate keys of R. A relation decomposed into two relations must have lossless join Property, which ensures that no spurious or extra tuples are generated when relations are reunited through a natural join.

Properties

A relation R is in 5NF if and only if it satisfies the following conditions:

1. R should be already in 4NF.
2. It cannot be further non loss decomposed (join dependency).

Example – Consider the above schema, with a case as "if a company makes a product and an agent is an agent for that company, then he always sells that product for the company". Under these circumstances, the ACP table is shown as:

Table ACP

Agent	Company	Product
A1	PQR	Nut
A1	PQR	Bolt
A1	XYZ	Nut
A1	XYZ	Bolt
A2	PQR	Nut

The relation ACP is again decomposed into 3 relations. Now, the natural Join of all three relations will be shown as:

Table R1

Agent	Company
A1	PQR
A1	XYZ
A2	PQR

Table R2

Agent	Product
A1	Nut
A1	Bolt
A2	Nut

Table R3

Company	Product
PQR	Nut
PQR	Bolt
XYZ	Nut
XYZ	Bolt

The result of the Natural Join of R1 and R3 over 'Company' and then the Natural Join of R13 and R2 over 'Agent'and 'Product' will be **Table ACP**.

Hence, in this example, all the redundancies are eliminated, and the decomposition of ACP is a lossless join decomposition. Therefore, the relation is in 5NF as it does not violate the property of lossless join.

Conclusion

- Multivalued dependencies are removed by 4NF, and join dependencies are removed by 5NF.
- The greatest degrees of database normalization, 4NF and 5NF, might not be required for every application.
- Normalizing to 4NF and 5NF might result in more complicated database structures and slower query speed, but it can also increase data accuracy, dependability, and simplicity.

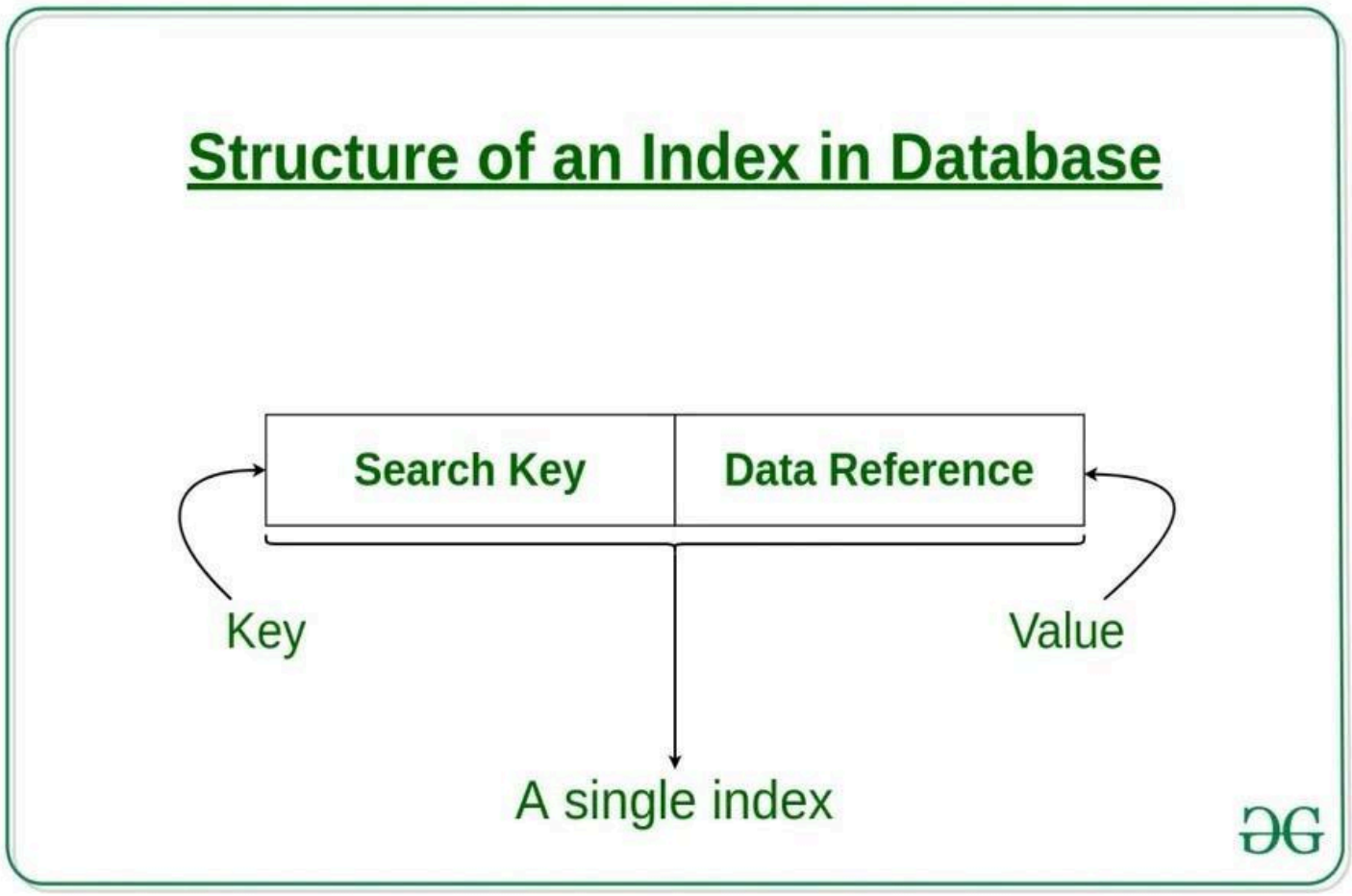
MODULE 5

Indexing in Databases



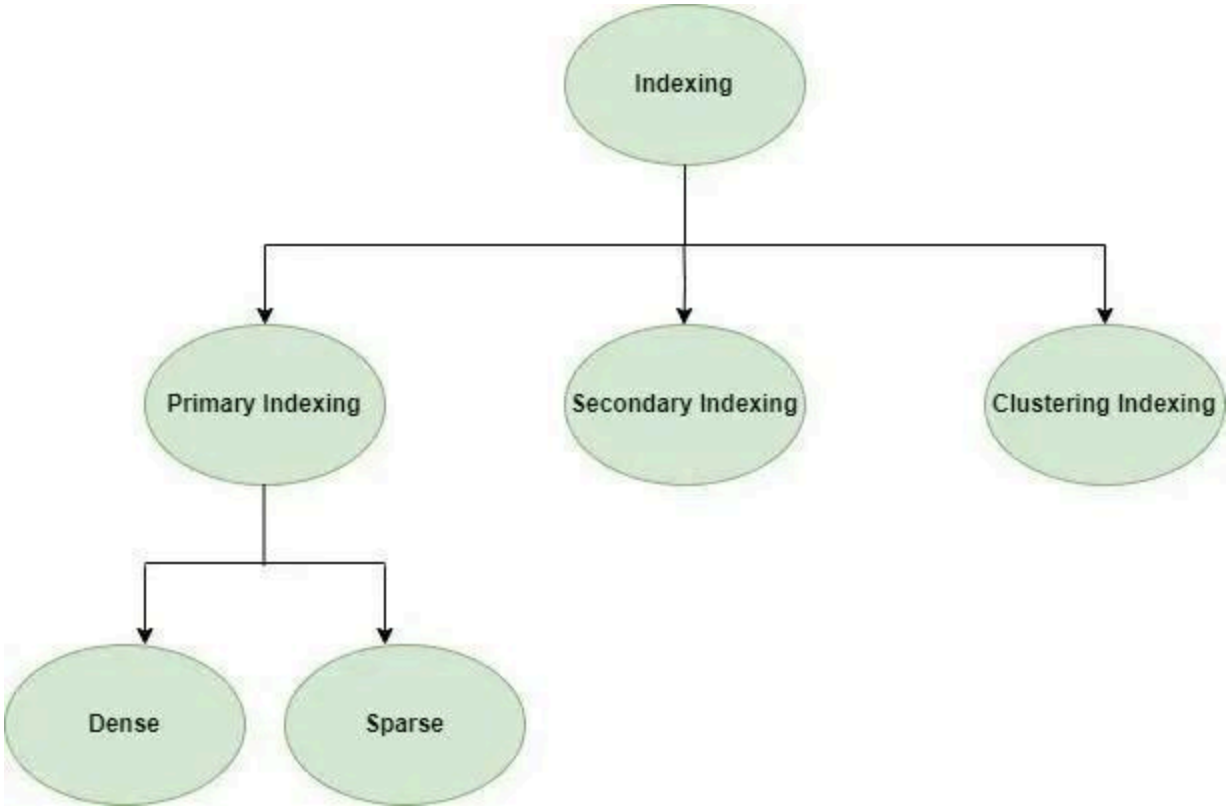
Indexing improves database performance by minimizing the number of disc visits required to fulfill a query. It is a data structure technique used to locate and quickly access data in databases. Several database fields are used to generate indexes. The main key or candidate key of the table

is duplicated in the first column, which is the Search key. To speed up data retrieval, the values are also kept in sorted order. It should be highlighted that sorting the data is not required. The second column is the Data Reference or Pointer which contains a set of pointers holding the address of the disk block where that particular key value can be found.



Attributes of Indexing

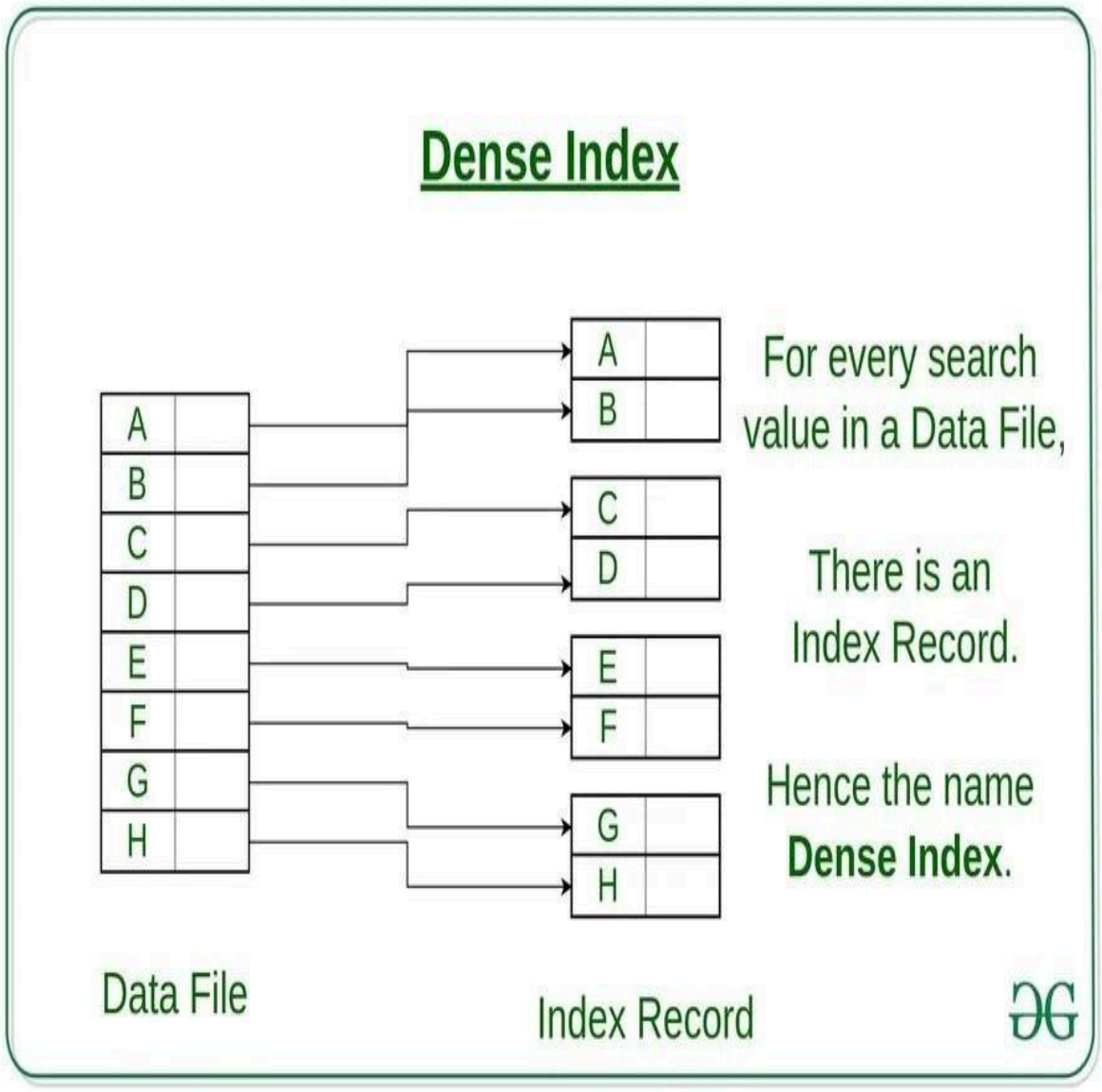
- **Access Types:** This refers to the type of access such as value-based search, range access, etc.
- **Access Time:** It refers to the time needed to find a particular data element or set of elements.
- **Insertion Time:** It refers to the time taken to find the appropriate space and insert new data.
- **Deletion Time:** Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead:** It refers to the additional space required by the index.



In general, there are two types of file organization mechanisms that are followed by the indexing methods to store the data:

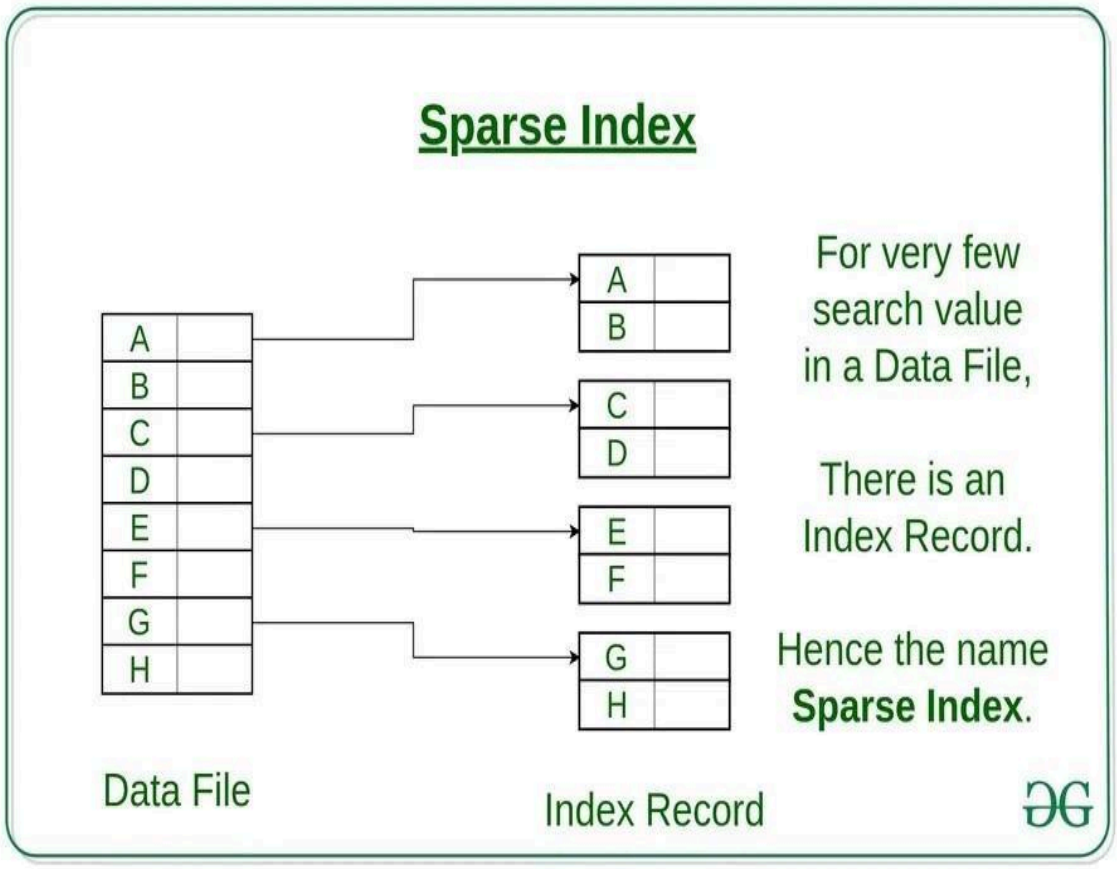
Sequential File Organization or Ordered Index File
In this, the indices are based on a sorted ordering of the values. These are generally fast and a more traditional type of storing mechanism. These Ordered or Sequential file organizations might store the data in a dense or sparse format.

- **Dense Index**
    - For every search key value in the data file, there is an index record.
    - This record contains the search key and also a reference to the first data record with that search key value.



Sparse Index

- The index record appears only for a few items in the data file. Each item points to a block as shown.
- To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
- We start at that record pointed to by the index record, and proceed along with the pointers in the file (that is, sequentially) until we find the desired record.
- Number of Accesses required=$\log_2(n)+1$, (here n=number of blocks acquired by index file)



Sparse Index

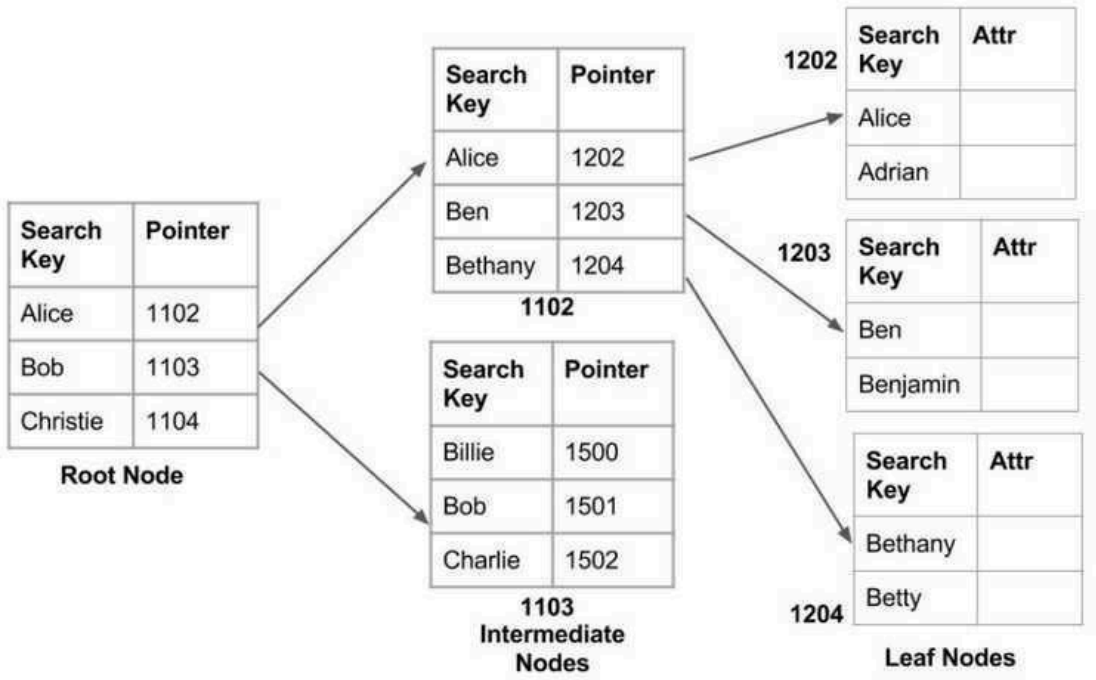
Hash File Organization

Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned are determined by a function called a hash function. There are primarily three methods of indexing:

- **Clustered Indexing**: When more than two records are stored in the same file this type of storing is known as cluster indexing. By using cluster indexing we can reduce the cost of searching reason being multiple records related to the same thing are stored in one place and it also gives the frequent joining of more than two tables (records).
  The clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create an index out of them. This method is known as the clustering index. Essentially, records with similar properties are grouped together, and indexes for these groupings are formed.
  Students studying each semester, for example, are grouped together. First-semester students, second-semester students, third-semester students, and so on are categorized.

Clustered Indexing

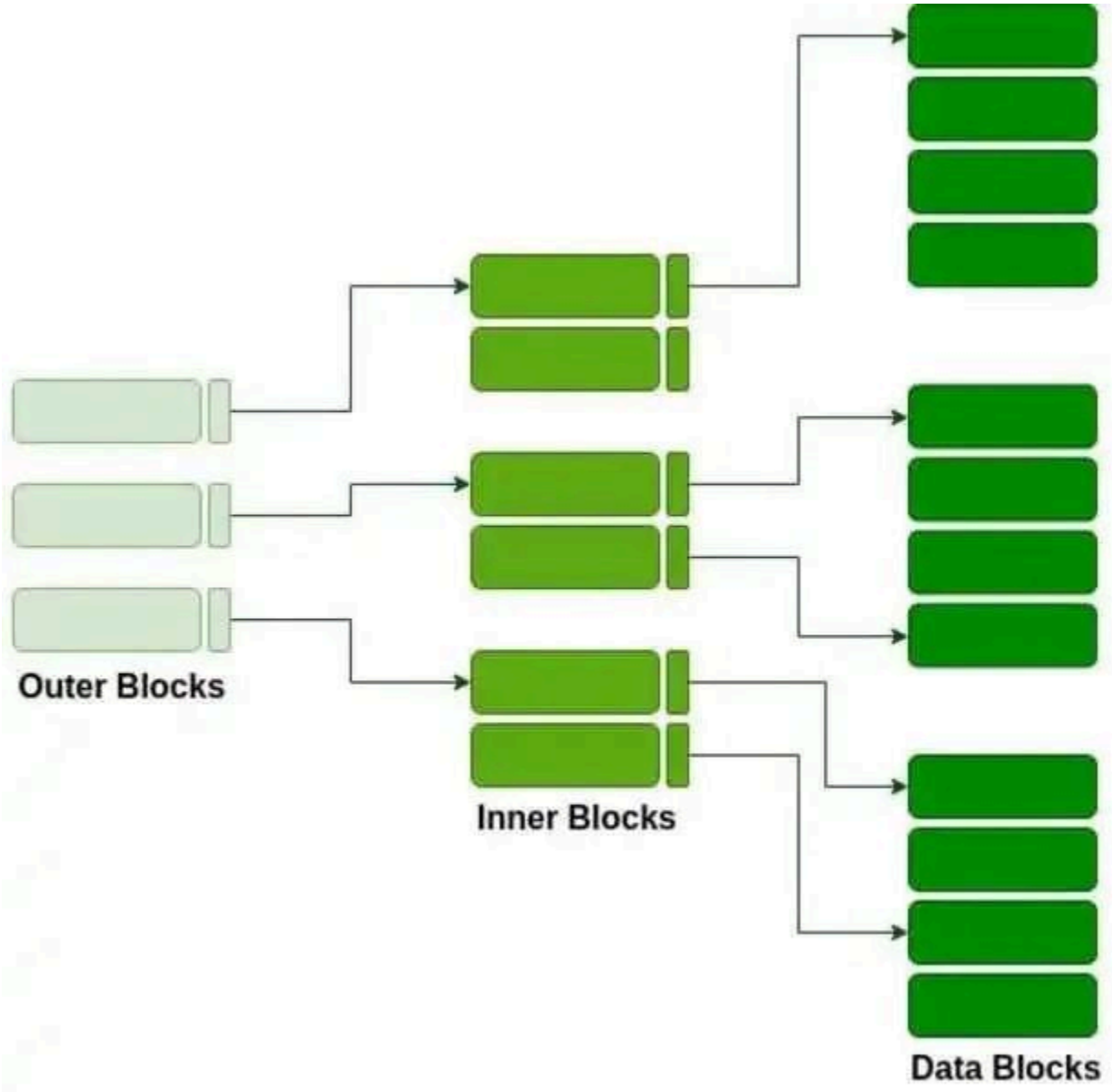
- **Primary Indexing:** This is a type of Clustered Indexing wherein the data is sorted according to the search key and the primary key of the database table is used to create the index. It is a default format of indexing where it induces sequential file organization. As primary keys are unique and are stored in a sorted manner, the performance of the searching operation is quite efficient.
- **Non-clustered or Secondary Indexing**: A non-clustered index just tells us where the data lies, i.e. it gives us a list of virtual pointers or references to the location where the data is actually stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For eg. the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here(information on each page of the book) is not organized but we have an ordered reference(contents page) to where the data points actually lie. We can have only dense ordering in the non-clustered index as sparse ordering is not possible because data is not physically organized accordingly.
  It requires more time as compared to the clustered index because some amount of extra work is done in order to extract the data by further following the pointer. In the case of a clustered index, data is directly present in front of the index.



Non clustered index

Non Clustered Indexing

- **Multilevel Indexing:** With the growth of the size of the database, indices also grow. As the index is stored in the main memory, a single-level index might become too large a size to store with multiple disk accesses. The multilevel indexing segregates the main block into various smaller blocks so that the same can be stored in a single block. The outer blocks are divided into inner blocks which in turn are pointed to the data blocks. This can be easily stored in the main memory with fewer overheads.



Multilevel Indexing

Advantages of Indexing

- **Improved Query Performance:** Indexing enables faster data retrieval from the database. The database may rapidly discover rows that match a specific value or collection of values by generating an index on a column, minimizing the amount of time it takes to perform a query.
- **Efficient Data Access:** Indexing can enhance data access efficiency by lowering the amount of disk I/O required to retrieve data. The database can maintain the data pages for frequently visited columns in memory by generating an index on those columns, decreasing the requirement to read from disk.
- **Optimized Data Sorting:** Indexing can also improve the performance of sorting operations. By creating an index on the columns used for sorting, the database can avoid sorting the entire table and instead sort only the relevant rows.
- **Consistent Data Performance:** Indexing can assist ensure that the database performs consistently even as the amount of data in the database rises. Without indexing, queries may take longer to run as the number of rows in the table grows, while indexing maintains a roughly consistent speed.
- By ensuring that only unique values are inserted into columns that have been indexed as unique, indexing can also be utilized to ensure the integrity of data. This avoids storing duplicate data in the database, which might lead to issues when performing queries or reports.

Overall, indexing in databases provides significant benefits for improving query performance, efficient data access, optimized data sorting, consistent data performance, and enforced data integrity

Disadvantages of Indexing

- Indexing necessitates more storage space to hold the index data structure, which might increase the total size of the database.
- **Increased database maintenance overhead:** Indexes must be maintained as data is added, destroyed, or modified in the table, which might raise database maintenance overhead.
- Indexing can reduce insert and update performance since the index data structure must be updated each time data is modified.
- **Choosing an index can be difficult:** It can be challenging to choose the right indexes for a specific query or application and may call for a detailed examination of the data and access patterns.

Features of Indexing

- The development of data structures, such as B-trees or hash tables, that provide quick access to certain data items is known as indexing. The data structures themselves are built on the values of the indexed columns, which are utilized to quickly find the data objects.
- The most important columns for indexing columns are selected based on how frequently they are used and the sorts of queries they are subjected to. The cardinality, selectivity, and uniqueness of the indexing columns can be taken into account.
- There are several different index types used by databases, including primary, secondary, clustered, and non-clustered indexes. Based on the particular needs of the database system, each form of index offers benefits and drawbacks.
- For the database system to function at its best, periodic index maintenance is required. According to changes in the data and usage patterns, maintenance work involves building, updating, and removing indexes.
- Database query optimization involves indexing, which is essential. The query optimizer utilizes the indexes to choose the best execution strategy for a particular query based on the cost of accessing the data and the selectivity of the indexing columns.

- Databases make use of a range of indexing strategies, including covering indexes, index-only scans, and partial indexes. These techniques maximize the utilization of indexes for particular types of queries and data access.
- When non-contiguous data blocks are stored in an index, it can result in index fragmentation, which makes the index less effective. Regular index maintenance, such as defragmentation and reorganization, can decrease fragmentation.

Conclusion

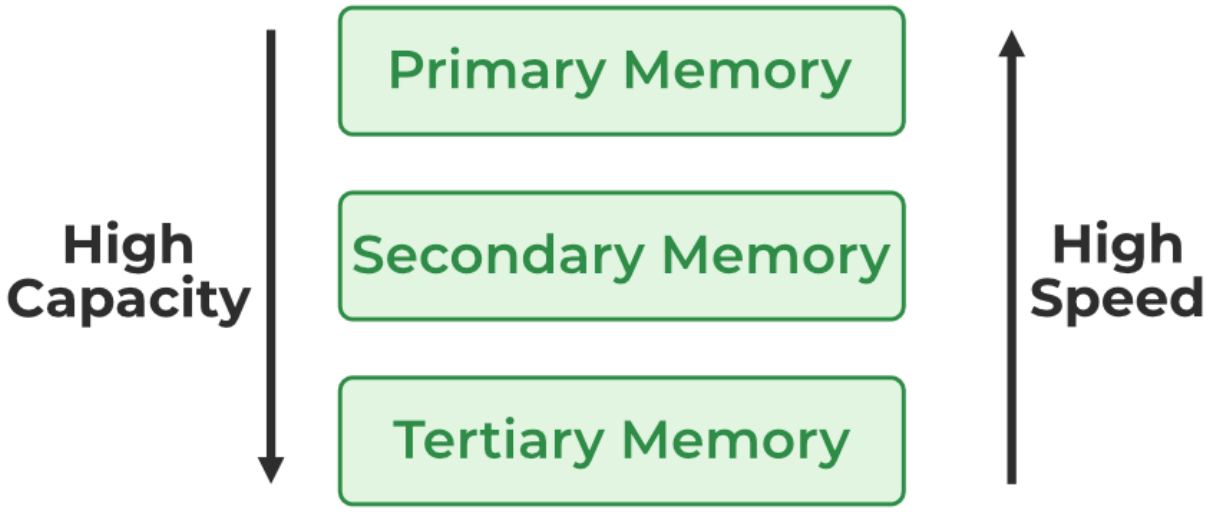
Indexing is a very useful technique that helps in optimizing the search time in database queries. The table of database indexing consists of a search key and pointer. There are four types of indexing: Primary, Secondary Clustering, and Multivalued Indexing. Primary indexing is divided into two types, dense and sparse. Dense indexing is used when the index table contains records for every search key. Sparse indexing is used when the index table does not use a search key for every record. Multilevel indexing uses B+ Tree. The main purpose of indexing is to provide better performance for data retrieval.

Storage Types in DBMS



The records in databases are stored in file formats. Physically, the data is stored in electromagnetic format on a device. The electromagnetic devices used in database systems for data storage are classified as follows:

1. Primary Memory
2. Secondary Memory
3. Tertiary Memory



Types of Memory

1. Primary Memory

The primary memory of a server is the type of data storage that is directly accessible by the central processing unit, meaning that it doesn't require any other devices to read from it. The primary memory must, in general, function flawlessly with equal contributions from the electric power supply, the hardware backup system, the supporting devices, the coolant that moderates the system temperature, etc.

- The size of these devices is considerably smaller and they are volatile.
- According to performance and speed, the primary memory devices are the fastest devices, and this feature is in direct correlation with their capacity.
- These primary memory devices are usually more expensive due to their increased speed and performance.

The cache is one of the types of Primary Memory.

- **Cache Memory:** Cache Memory is a special very high-speed memory. It is used to speed up and synchronize with a high-speed CPU. Cache memory is costlier than main memory or disk memory but more economical than CPU registers. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU.

2. Secondary Memory

Data storage devices known as secondary storage, as the name suggests, are devices that can be accessed for storing data that will be needed at a later point in time for various purposes or

database actions. Therefore, these types of storage systems are sometimes called backup units as well. Devices that are plugged or connected externally fall under this memory category, unlike primary memory, which is part of the CPU. The size of this group of devices is noticeably larger than the primary devices and smaller than the tertiary devices.

- It is also regarded as a temporary storage system since it can hold data when needed and delete it when the user is done with it. Compared to primary storage devices as well as tertiary devices, these secondary storage devices are slower with respect to actions and pace.
- It usually has a higher capacity than primary storage systems, but it changes with the technological world, which is expanding every day.

Some commonly used Secondary Memory types that are present in almost every system are:

- **Flash Memory:** Flash memory, also known as flash storage, is a type of nonvolatile memory that erases data in units called blocks and rewrites data at the byte level. Flash memory is widely used for storage and data transfer in consumer devices, enterprise systems, and industrial applications. Unlike traditional hard drives, flash memories are able to retain data even after the power has been turned off
- **Magnetic Disk Storage:** A Magnetic Disk is a type of secondary memory that is a flat disc covered with a magnetic coating to hold information. It is used to store various programs and files. The polarized information in one direction is represented by 1, and vice versa. The direction is indicated by 0.

3. Tertiary Memory

For data storage, Tertiary Memory refers to devices that can hold a large amount of data without being constantly connected to the server or the peripherals. A device of this type is connected either to a server or to a device where the database is stored from the outside.

- Due to the fact that tertiary storage provides more space than other types of device memory but is most slowly performing, the cost of tertiary storage is lower than primary and secondary. As a means to make a backup of data, this type of storage is commonly used for making copies from servers and databases.
- The ability to use secondary devices and to delete the contents of the tertiary devices is similar.

Some commonly used Tertiary Memory types that are almost present in every system are:

- **Optical Storage:** It is a type of storage where reading and writing are to be performed with the help of a laser. Typically data written on CDs and DVDs are examples of Optical Storage.
- **Tape Storage:** Tape Storage is a type of storage data where we use magnetic tape to store data. It is used to store data for a long time and also helps in the backup of data in case of data loss.

Memory Hierarchy

A computer system has a hierarchy of memory. Direct access to a CPU's main memory and inbuilt registers is available. Accessing the main memory takes less time than running a CPU. Cache memory is introduced to minimize this difference in speed. Data that is most frequently accessed by the CPU resides in cache memory, which provides the fastest access time to data. Fastest-accessing memory is the most expensive. Although large storage devices are slower and less expensive than CPU registers and cache memory, they can store a greater amount of data.

1. Magnetic Disks

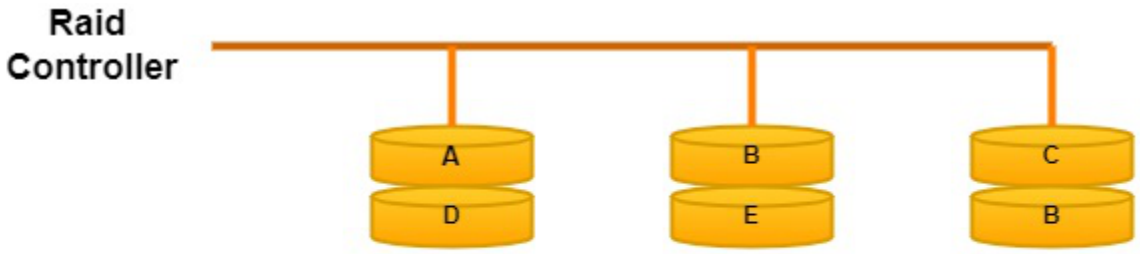
Present-day computer systems use hard disk drives as secondary storage devices. Magnetic disks store information using the concept of magnetism. Metal disks are coated with magnetizable material to create hard disks. Spindles hold these disks vertically. As the read/write head moves between the disks, it de-magnetizes or magnetizes the spots under it. There are two magnetized

spots: 0 (zero) and 1 (one). Formatted hard disks store data efficiently by storing them in a defined order. The hard disk plate is divided into many concentric circles, called tracks. Each track contains a number of sectors. Data on a hard disk is typically stored in sectors of 512 bytes.

2. Redundant Array of Independent Disks(RAID)

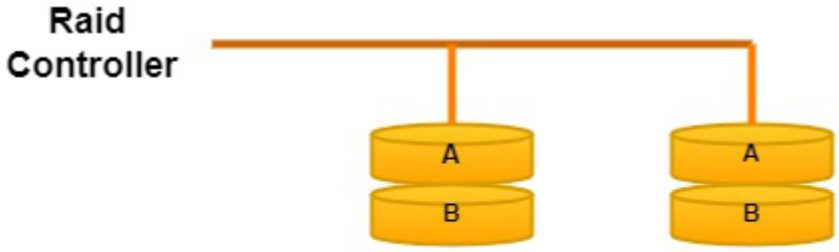
In the Redundant Array of Independent Disks technology, two or more secondary storage devices are connected so that the devices operate as one storage medium. A RAID array consists of several disks linked together for a variety of purposes. Disk arrays are categorized by their RAID levels.

- **RAID 0:** At this level, disks are organized in a striped array. Blocks of data are divided into disks and distributed over disks. Parallel writing and reading of data occur on each disk. This improves performance and speed. Level 0 does not support parity and backup.



Raid-0

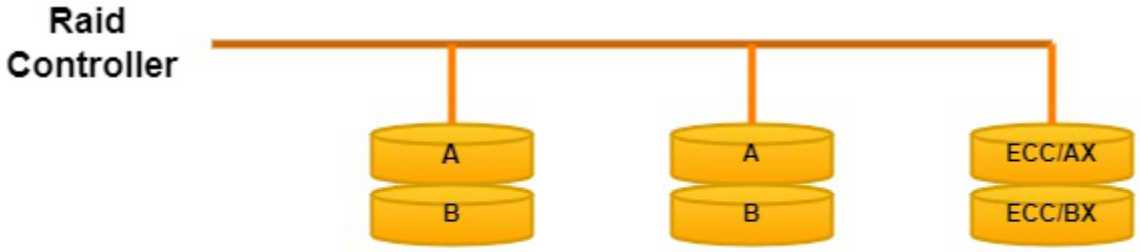
- **RAID 1:** Mirroring is used in RAID 1. A RAID controller copies data across all disks in an array when data is sent to it. In case of failure, RAID level 1 provides 100% redundancy.



Raid-1

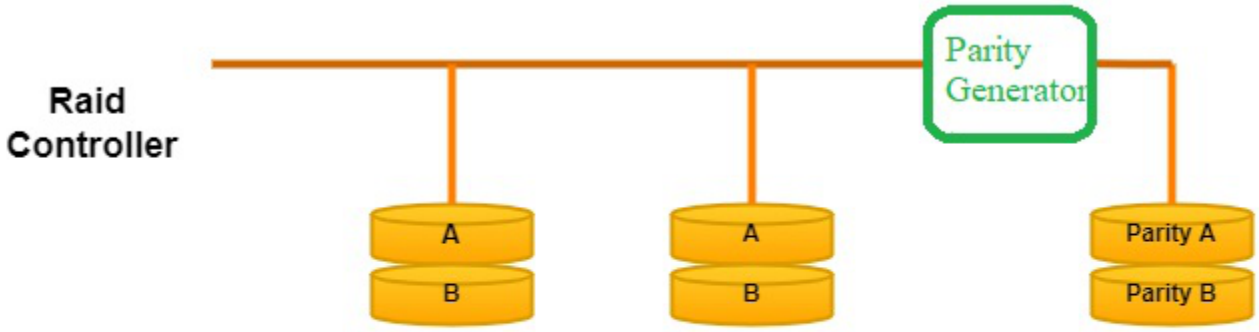
- **RAID 2:** The data in RAID 2 is striped on different disks, and the Error Correction Code is recorded using Hamming distance. Similarly to level 0, each bit within a word is stored on a

separate disk, and ECC codes for the data words are saved on a separate set of disks. As a result of its complex structure and high cost, RAID 2 cannot be commercially deployed.



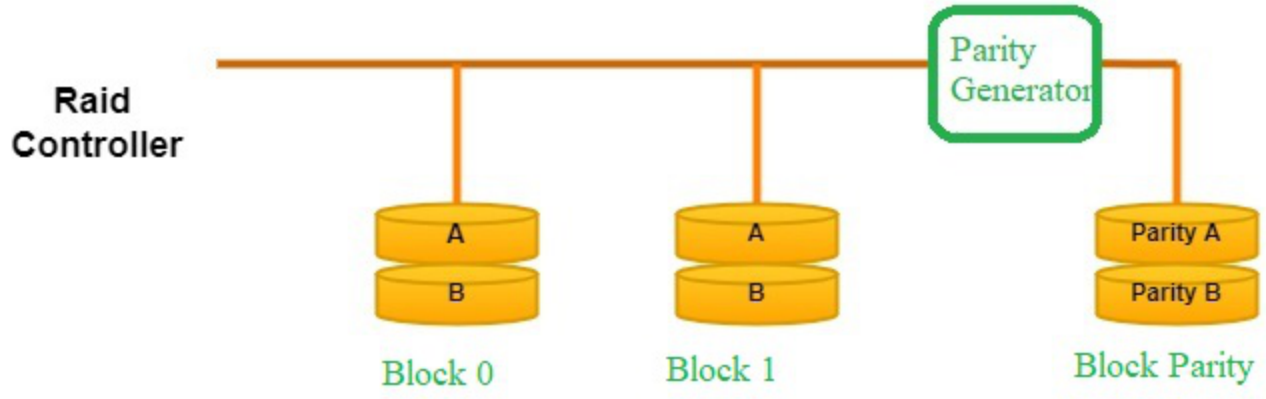
Raid-2

- **RAID 3:** Data is striped across multiple disks in RAID 3. Data words are parsed to generate a parity bit. It is stored on a different disk. Thus, single-disk failures can be avoided.



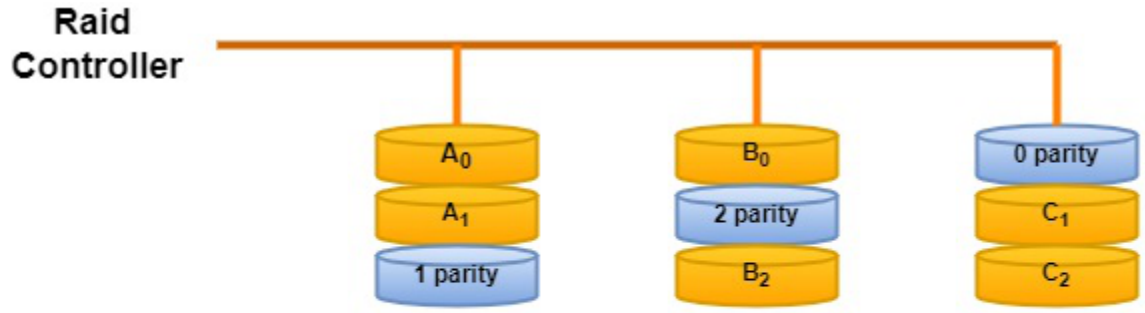
Raid-3

- **RAID 4:** This level involves writing an entire block of data onto data disks, and then generating the parity and storing it somewhere else. At level 3, bytes are striped, while at level 4, blocks are striped. Both levels 3 and 4 require a minimum of three disks.



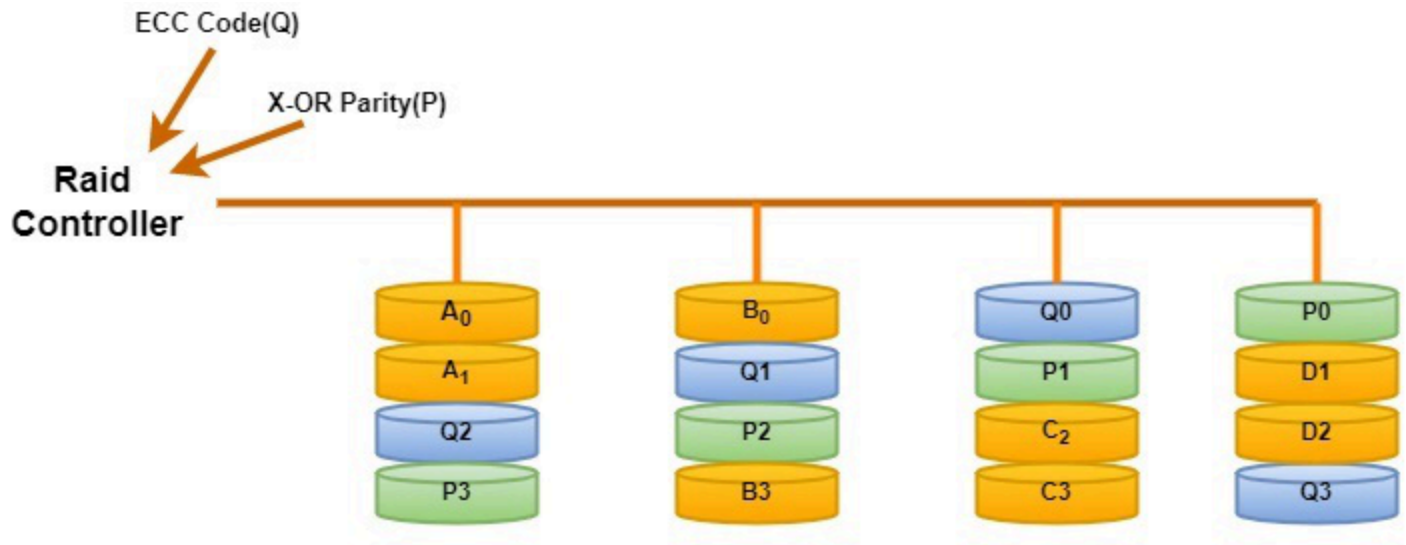
Raid-4

- **RAID 5:** The data blocks in RAID 5 are written to different disks, but the parity bits are spread out across all the data disks rather than being stored on a separate disk.



Raid-5

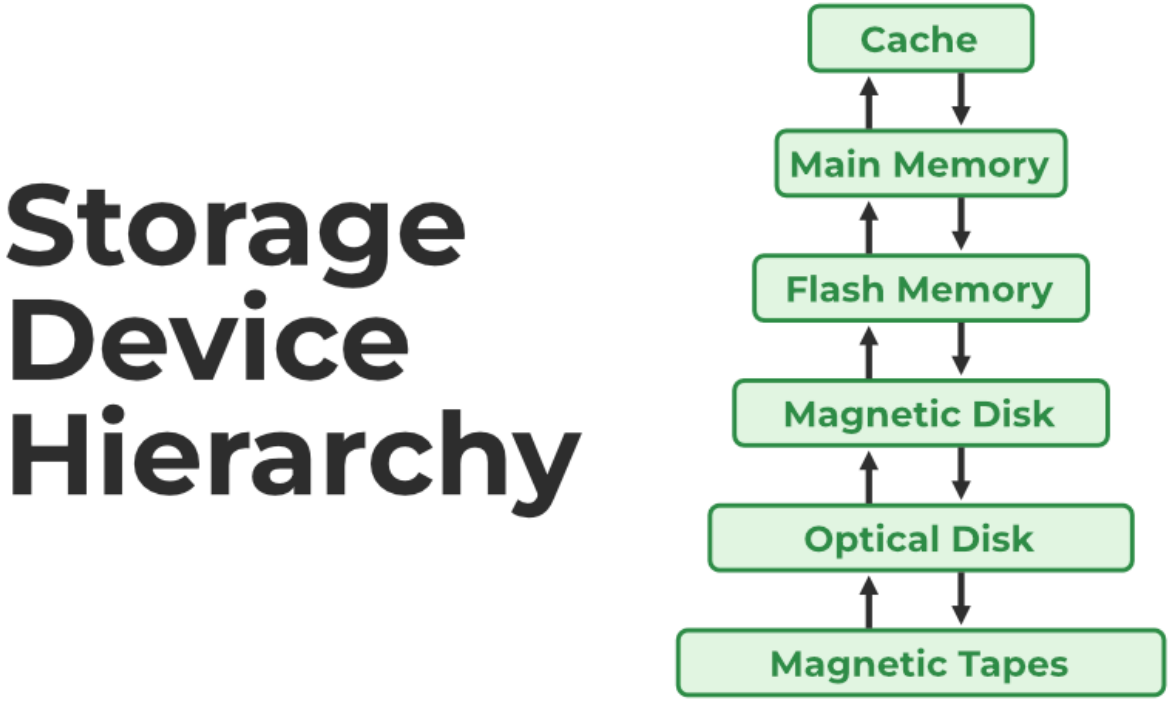
- **RAID 6:** The RAID 6 level extends the level 5 concept. A pair of independent parities are generated and stored on multiple disks at this level. A pair of independent parities are generated and stored on multiple disks at this level. Ideally, you need four disk drives for this level.



Raid-6

Storage Hierarchy

Rather than the storage devices mentioned above, there are also other devices that are also used in day-to-day life. These are mentioned below in the form of faster speed to lower speed from top to down.



Storage Hierarchy

Conclusion
A DBMS must balance the utilization of primary, secondary, and tertiary memory. Secondary memory meets long-term storage demands, tertiary memory can be used for archiving, and primary memory guarantees quick access for active data. Using various storage types strategically in accordance with needs and patterns of data access is essential for optimal database performance.

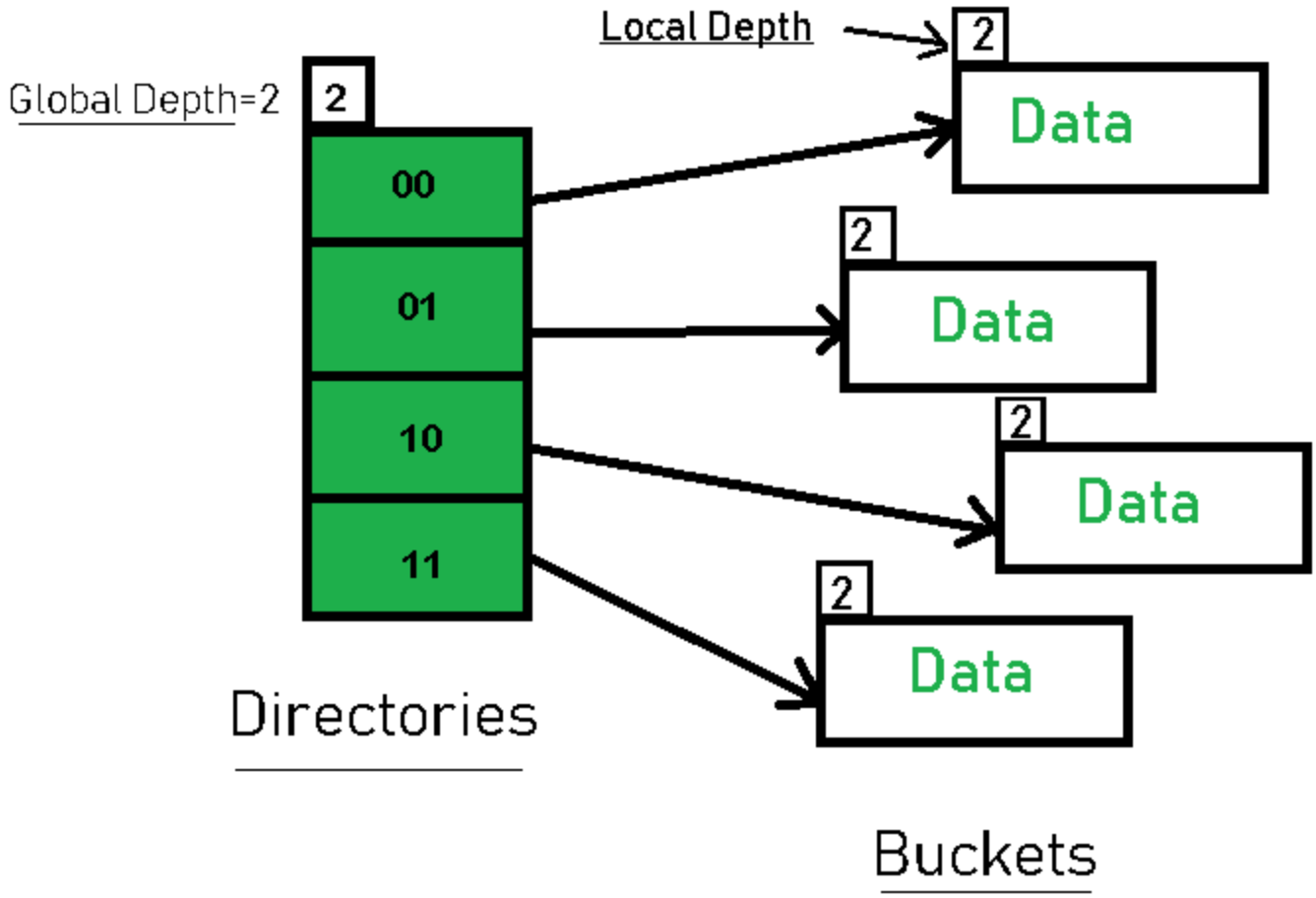
Extendible Hashing (Dynamic approach to DBMS)



Extendible Hashing is a dynamic hashing method wherein directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.
Main features of Extendible Hashing: The main features in this hashing technique are:

- **Directories:** The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.
- **Buckets:** The buckets are used to hash the actual data.

Basic Structure of Extendible Hashing:



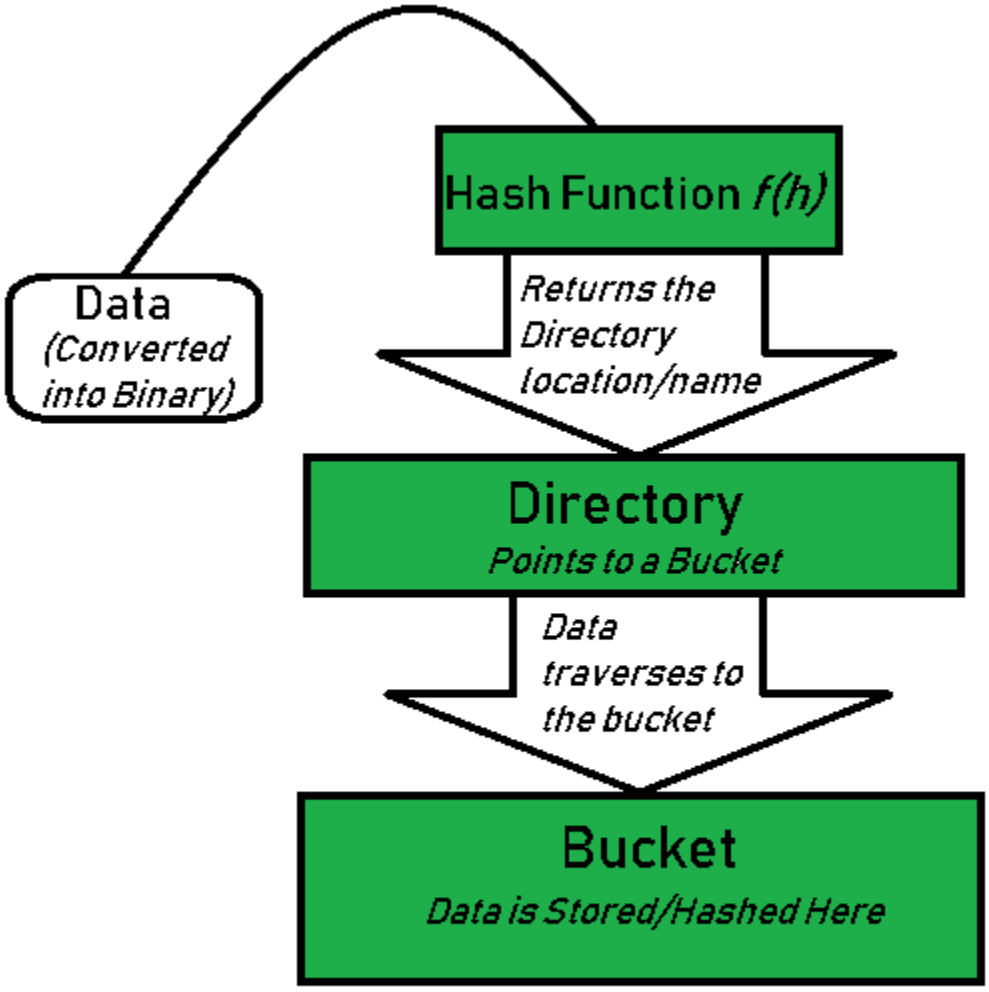
Extendible Hashing

Frequently used terms in Extendible Hashing:

- **Directories:** These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place. The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories = 2^Global Depth.
- **Buckets:** They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.
- **Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.
- **Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.

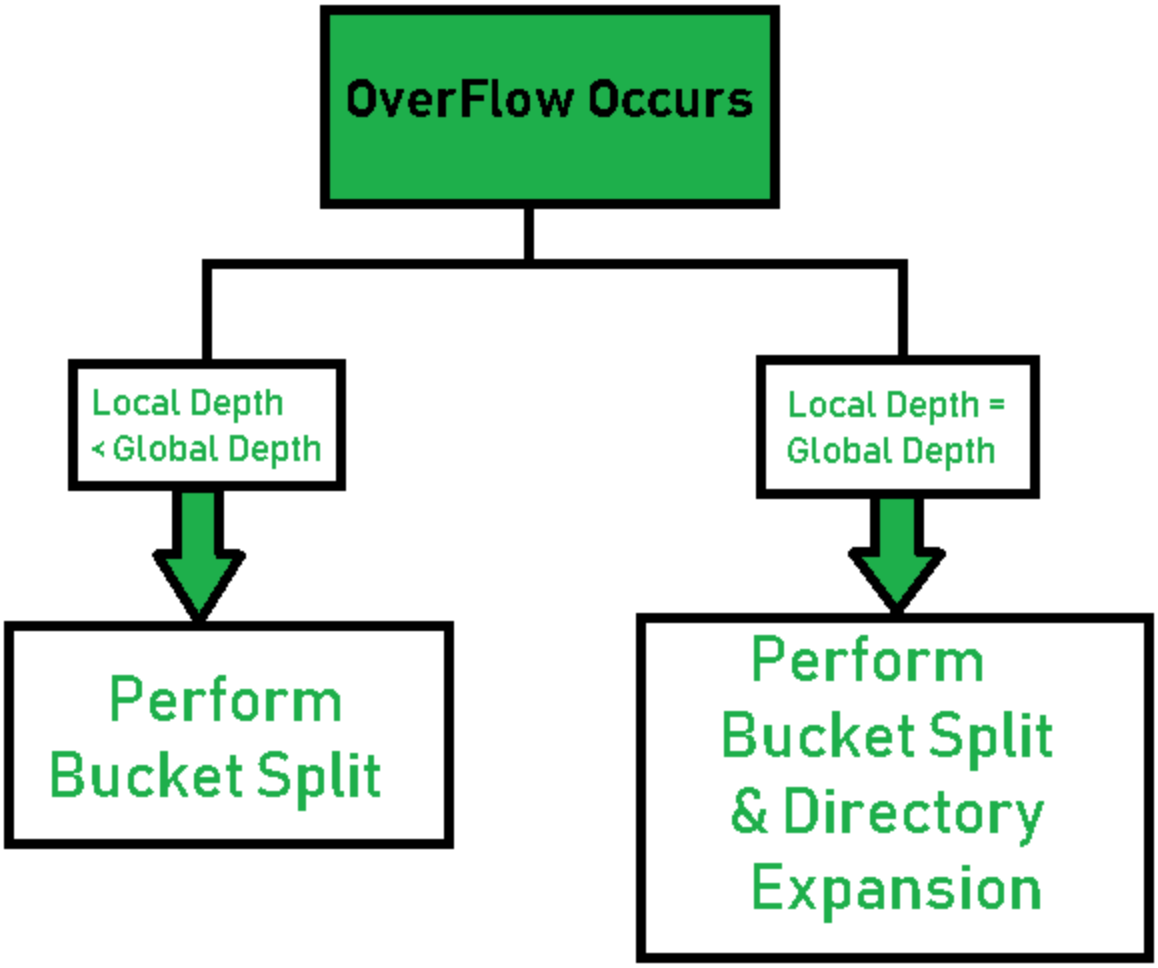
- **Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
- **Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.

Basic Working of Extendible Hashing:



- **Step 1 – Analyze Data Elements:** Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer. eg: 49.
- **Step 2 – Convert into binary format:** Convert the data element in Binary form. For string elements, consider the ASCII equivalent integer of the starting character and then convert the integer into binary form. Since we have 49 as our data element, its binary form is 110001.

- **Step 3 – Check Global Depth of the directory.** Suppose the global depth of the Hash-directory is 3.
- **Step 4 – Identify the Directory:** Consider the 'Global-Depth' number of LSBs in the binary number and match it to the directory id.
  Eg. The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110**001** viz. 001.
- **Step 5 – Navigation:** Now, navigate to the bucket pointed by the directory with directory-id 001.
- **Step 6 – Insertion and Overflow Check:** Insert the element and check if the bucket overflows. If an overflow is encountered, go to **step 7** followed by **Step 8**, otherwise, go to **step 9**.
- **Step 7 – Tackling Over Flow Condition during Data Insertion:** Many times, while inserting data in the buckets, it might happen that the Bucket overflows. In such cases, we need to follow an appropriate procedure to avoid mishandling of data.
  First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.
    - **Case1:** If the local depth of the overflowing Bucket is equal to the global depth, then Directory Expansion, as well as Bucket Split, needs to be performed. Then increment the global depth and the local depth value by 1. And, assign appropriate pointers. Directory expansion will double the number of directories present in the hash structure.
    - **Case2:** In case the local depth is less than the global depth, then only Bucket Split takes place. Then increment only the local depth value by 1. And, assign appropriate pointers.



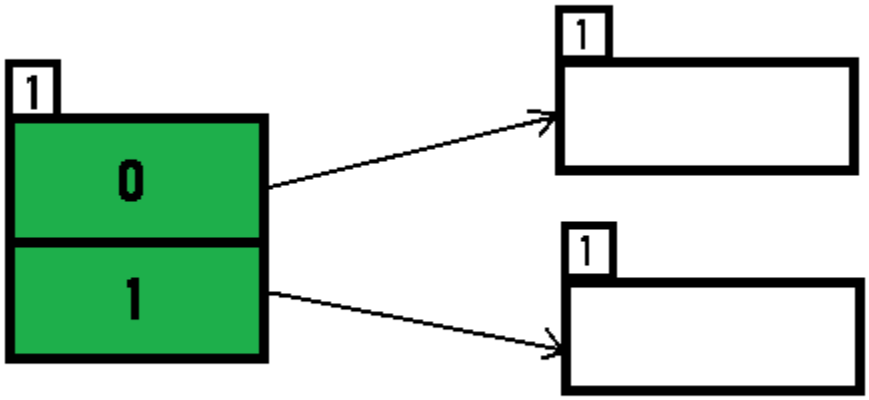
- **Step 8 – Rehashing of Split Bucket Elements:** The Elements present in the overflowing bucket that is split are rehashed w.r.t the new global depth of the directory.
- **Step 9 –** The element is successfully hashed.

Example based on Extendible Hashing: Now, let us consider a prominent example of hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**
Bucket Size: 3 (Assume)
Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

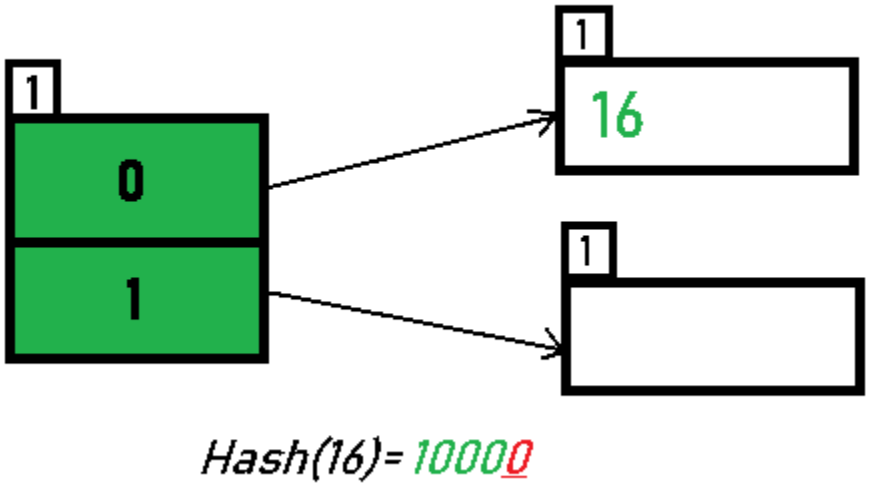
- **Solution:** First, calculate the binary forms of each of the given numbers.
  16- 10000
  4- 00100
  6- 00110
  22- 10110
  24- 11000
  10- 01010
  31- 11111
  7- 00111
  9- 01001

20- 10100
26- 11010

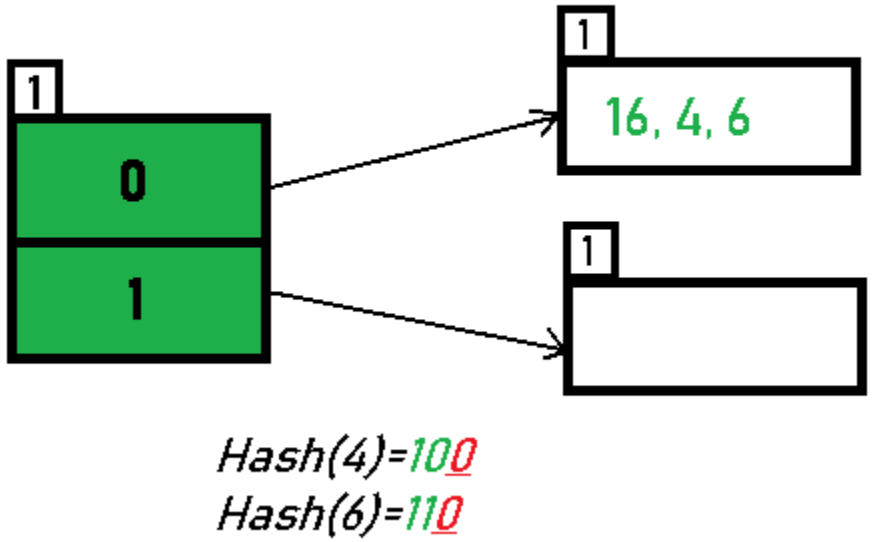
- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



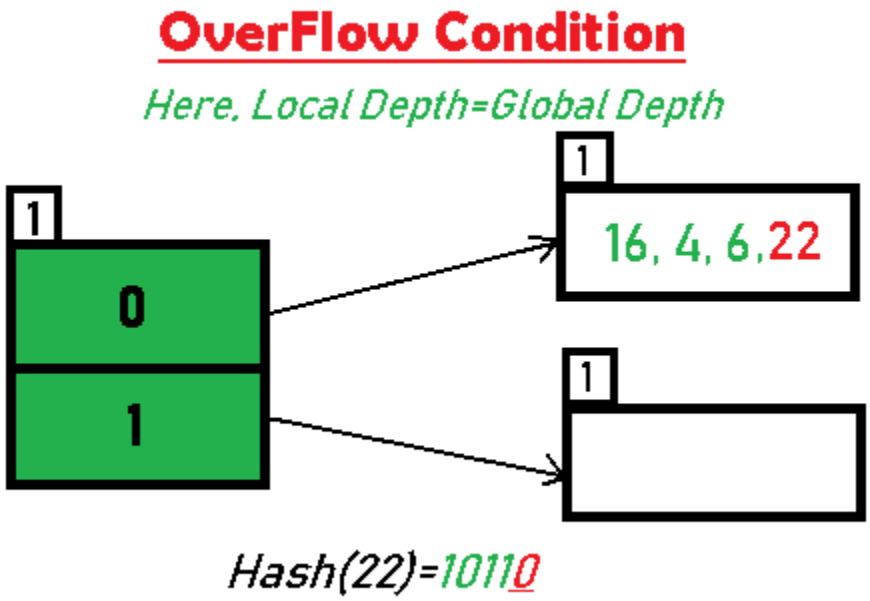
- **Inserting 16:**
  The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.



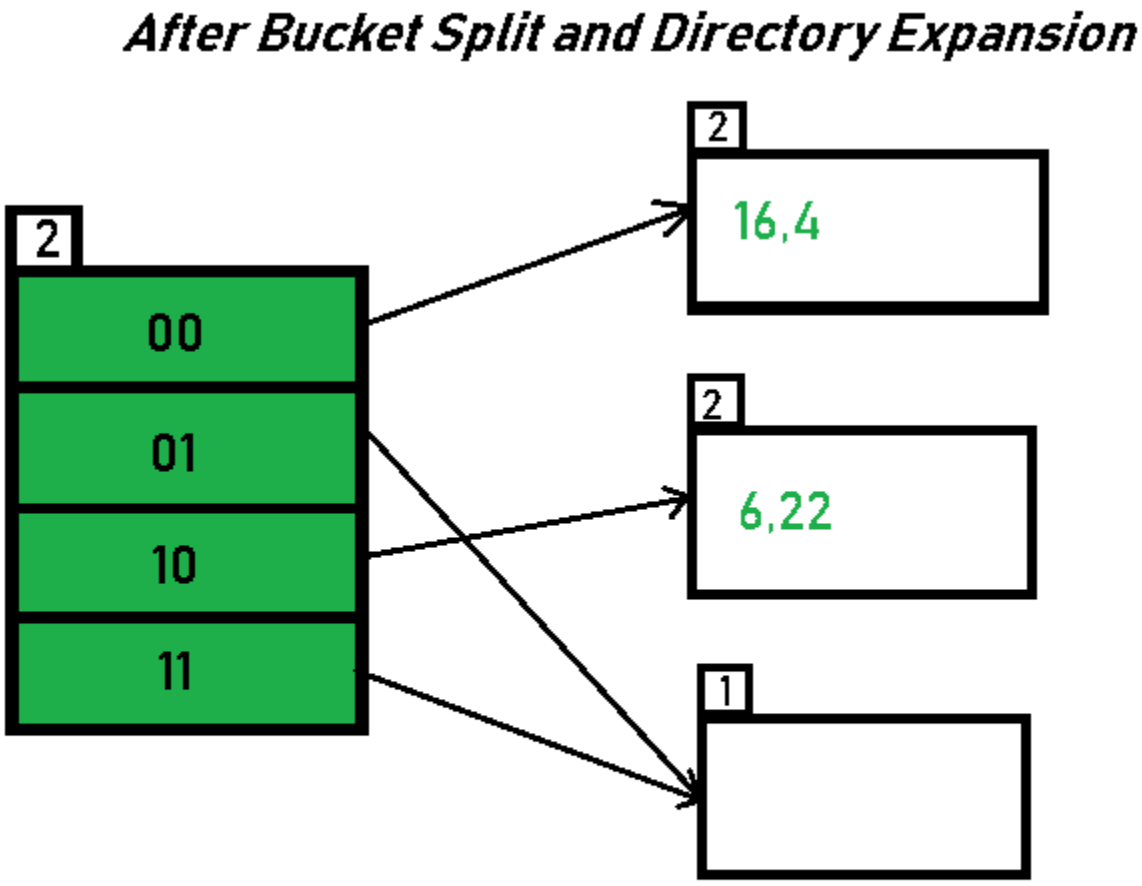
- **Inserting 4 and 6:**
  Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as follows:



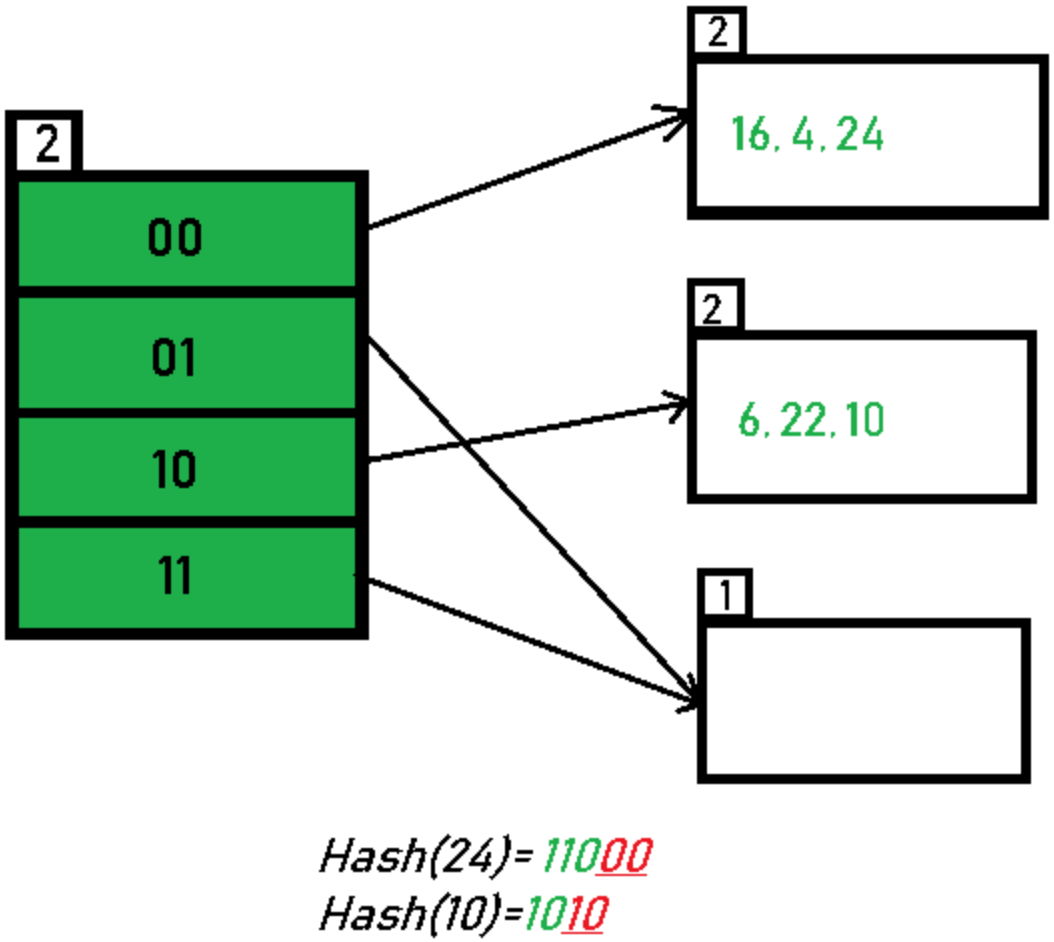
- **Inserting 22:** The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.



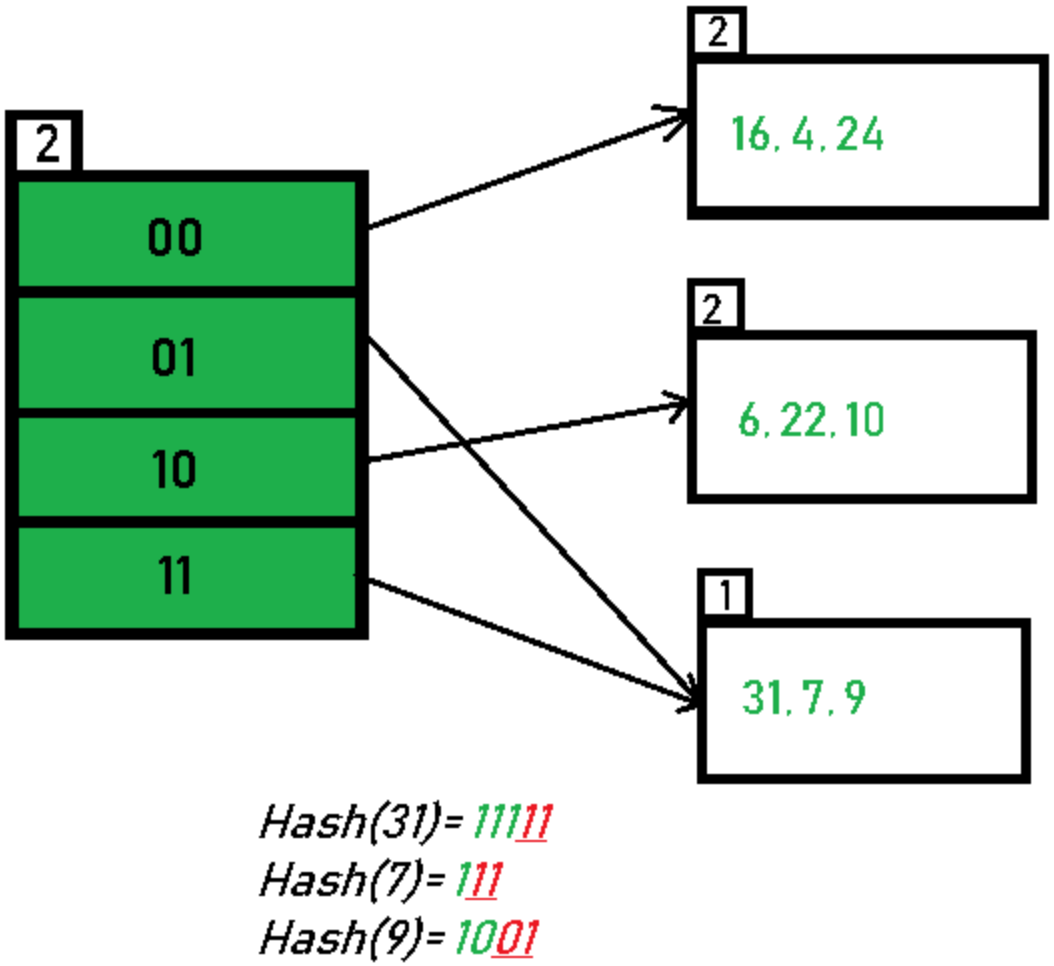
- As directed by **Step 7-Case 1**, Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now,the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(100**00**),4(1**00**),6(1**10**),22(101**10**)]



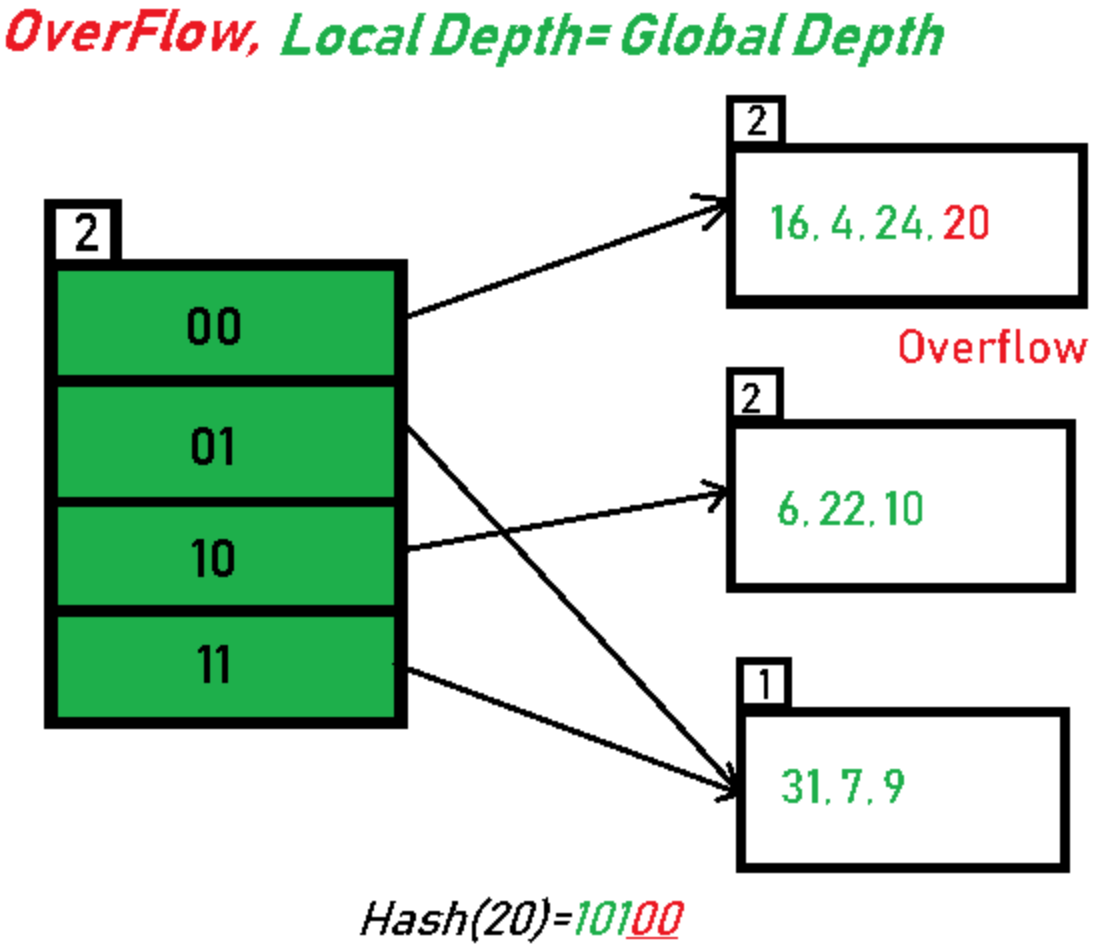
- **Notice that the bucket which was underflow has remained untouched. But, since the number of directories has doubled, we now have 2 directories 01 and 11 pointing to the same bucket. This is because the local-depth of the bucket has remained 1. And, any bucket having a local depth less than the global depth is pointed-to by more than one directories.*
- **Inserting 24 and 10:** 24(11**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



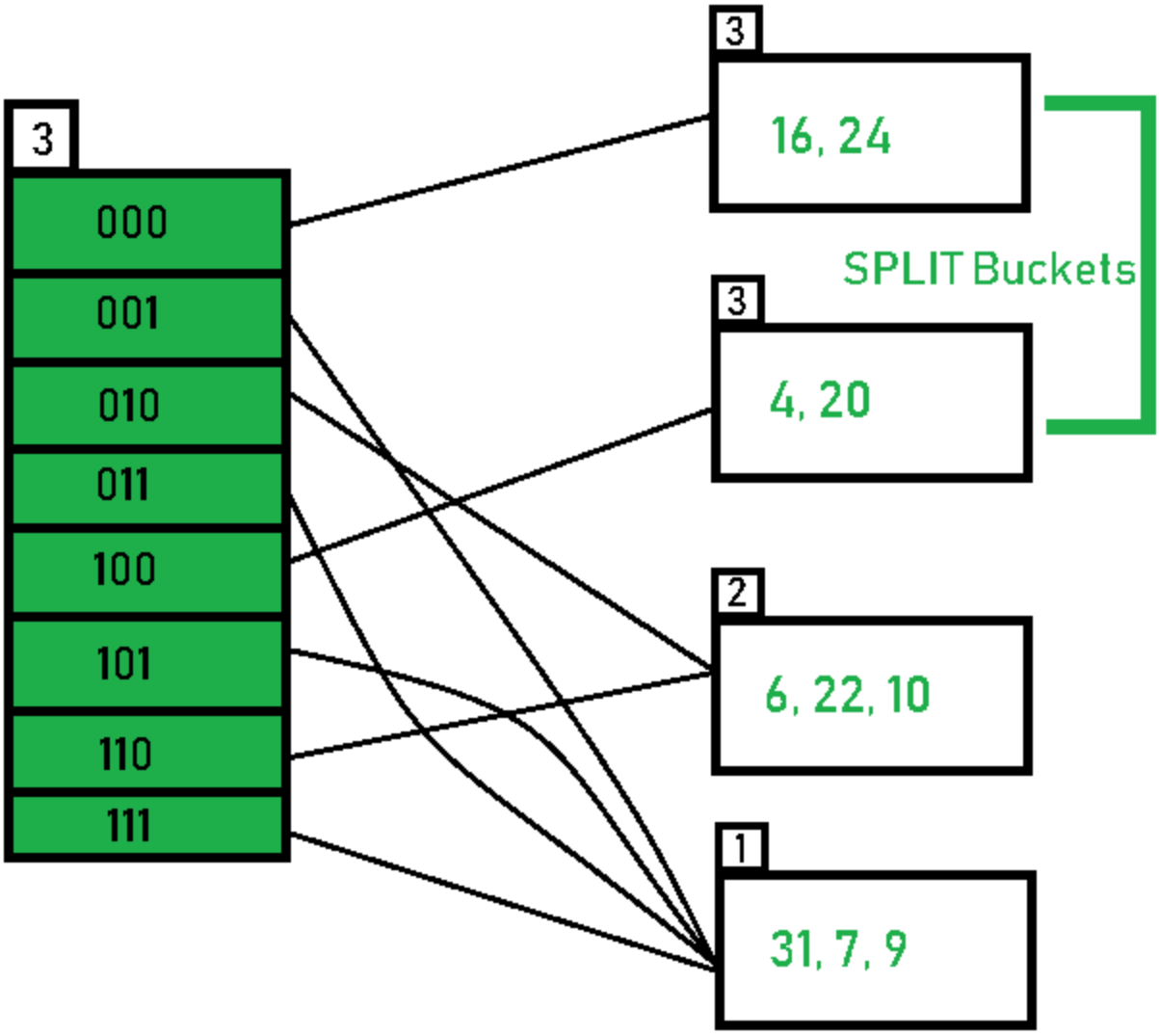
- **Inserting 31,7,9:** All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



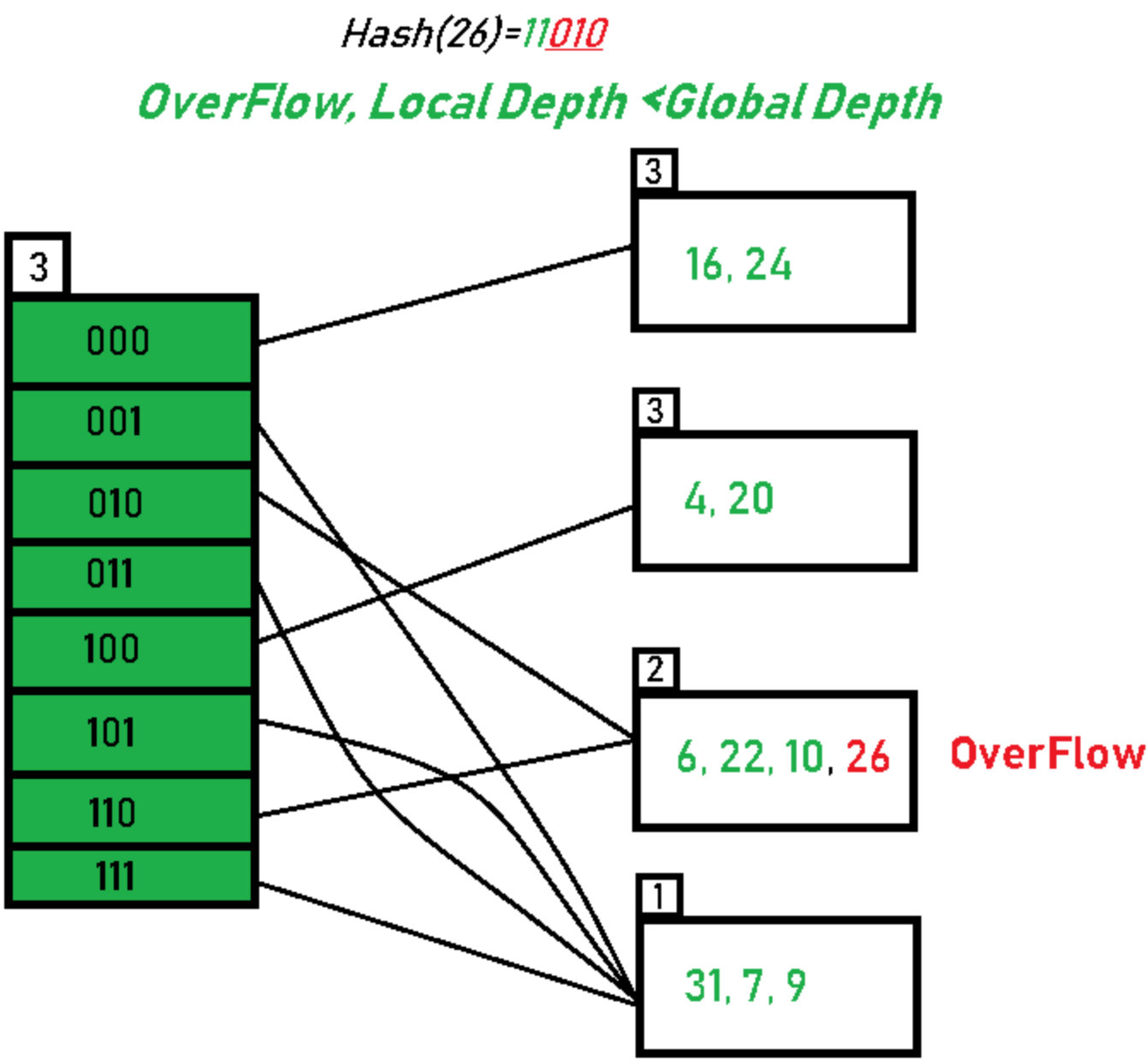
- **Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.



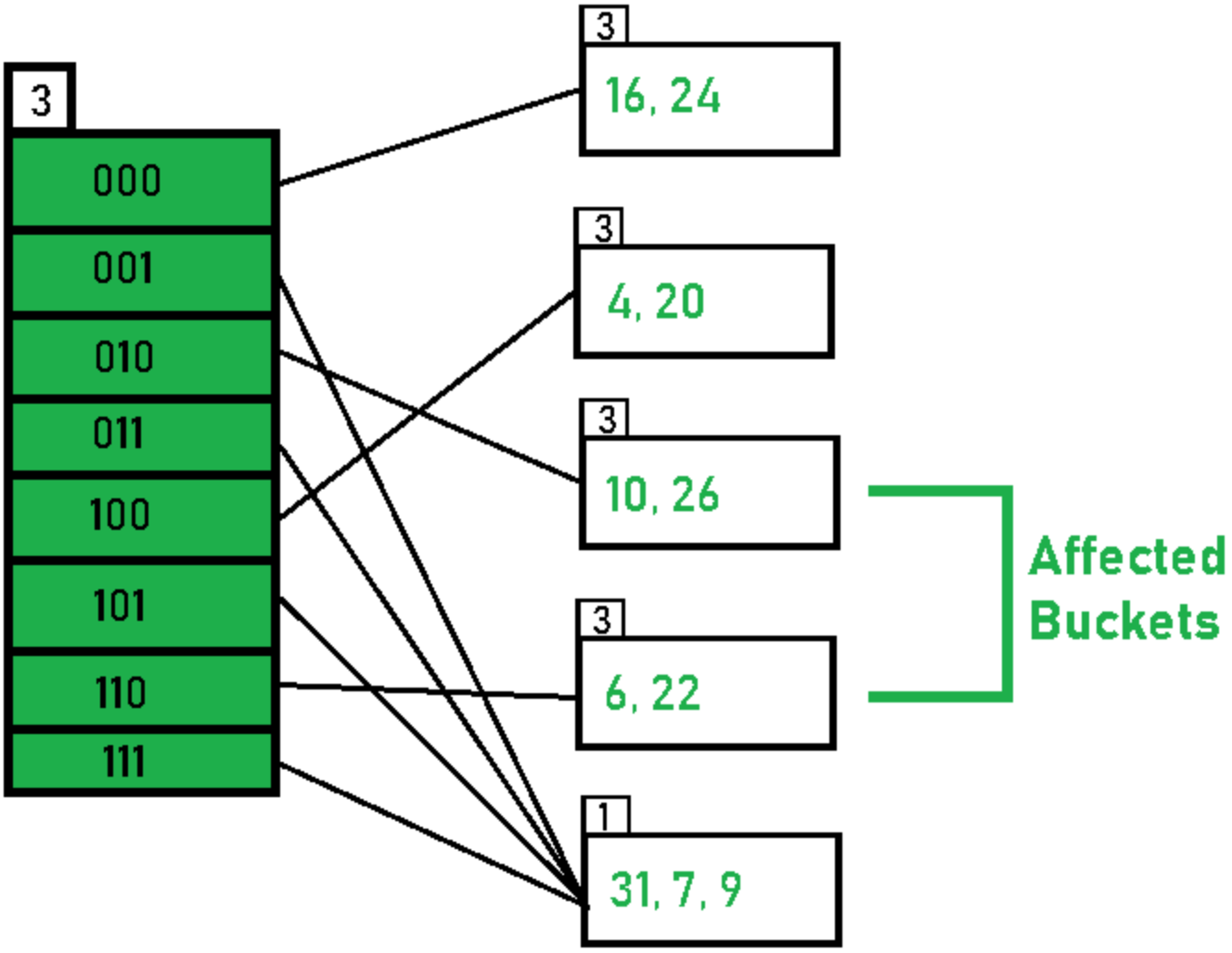
- 20 is inserted in bucket pointed out by 00. As directed by **Step 7-Case 1**, since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:



- **Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



- The bucket overflows, and, as directed by **Step 7-Case 2,** since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed.
  Finally, the output of hashing the given list of numbers is obtained.



- **Hashing of 11 Numbers is Thus Completed.**

Key Observations:

1. A Bucket will have more than one pointers pointing to it if its local depth is less than the global depth.
2. When overflow condition occurs in a bucket, all the entries in the bucket are rehashed with a new local depth.
3. If Local Depth of the overflowing bucket
4. The size of a bucket cannot be changed after the data insertion process begins.

Advantages:

1. Data retrieval is less expensive (in terms of computing).
2. No problem of Data-loss since the storage capacity increases dynamically.
3. With dynamic changes in hashing function, associated old values are rehashed w.r.t the new hash function.

Limitations Of Extendible Hashing:

1. The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
2. Size of every bucket is fixed.
3. Memory is wasted in pointers when the global depth and local depth difference becomes drastic.
4. This method is complicated to code.

Dynamic Hashing in DBMS

Last Updated : 01 Apr, 2024



In this article, we will learn about dynamic hashing in DBMS. Hashing in DBMS is used for searching the needed data on the disc. As static hashing is not efficient for large databases, dynamic hashing provides a way to work efficiently with databases that can be scaled.

What is Dynamic Hashing in DBMS?

Dynamic hashing is a technique used to dynamically add and remove data buckets when demanded. Dynamic hashing can be used to solve the problem like bucket overflow which can occur in static hashing. In this method, the data bucket size grows or shrinks as the number of records increases or decreases. This allows easy insertion or deletion into the database and reduces performance issues.

Important Terminologies Related to Dynamic Hashing

- **Hash Function:** A mathematical function that uses the primary key to generate the address of the data block.
- **Data Bucket:** These are the memory locations that contain actual data records.
- **Hash Index:** It is the address of the data block generated by hash function.
- **Bucket Overflow:** Bucket overflow occurs when memory address generated by the hash function is already filled by some data records.

How to Search a Key?

- Calculate the hash address of key.
- Calculate the number of bits used in the dictionary and denote these bits as i.
- Take the least significant i bits of hash address. This provides index of dictionary.
- This index is used to navigate to the dictionary and check for bucket address in which record may be present.

Advantages of Dynamic Hashing

- In dynamic hashing, performance will not get affected as the amount of data grows in the system. To accommodate the data, size of memory will be increased.
- Dynamic hashing improve the utilization of the memory.
- This method is efficient to handle the dynamic database where size of data changes frequently.

Disadvantages of Dynamic Hashing

- A the amount of data changes, bucket size will also get changed. Bucket address table will keep track of these addresses because data address changes as bucket size increases or

decreases. Maintenance of the bucket address table gets difficult when there is significant increase in data.

- In dynamic hashing, bucket overflow can happen.

How to Insert a New Record in Database Using Dynamic Hashing?

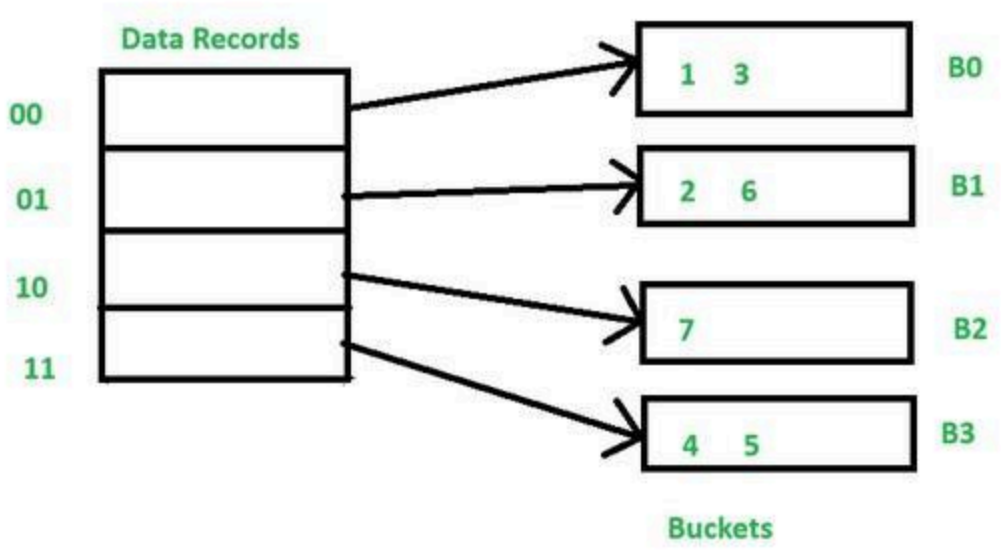
- Follow the same procedure that we used for searching which to lead to some bucket.
- If space is present in that bucket, then place record in it.
- If bucket is full, then split the bucket and redistribute the records.

Example

Consider the following table which contain key into bucket based on their hash address prefix

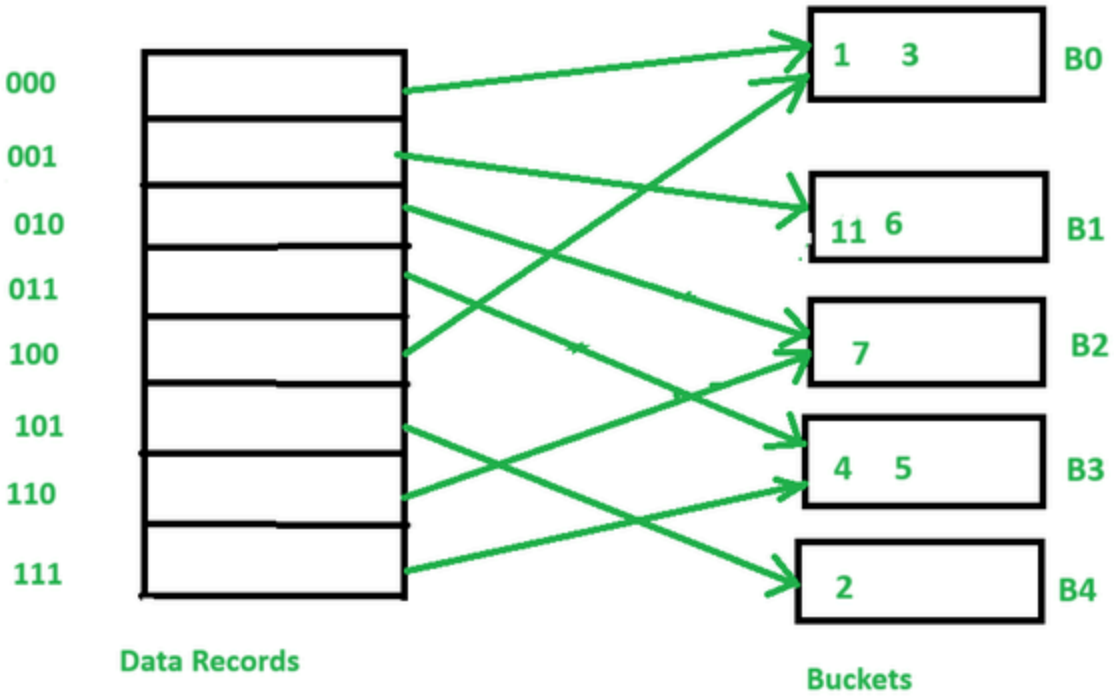
Key	Hash Address
1	10000
2	10101
3	11000
4	10011
5	11011
6	11001
7	10110

In the above table, the last two bits of 1 and 3 are 00. So, it will go into bucket B0. The last two bits of 2 and 6 are 01. So, it will go into bucket B1. The last bit of 7 is 10. So, it will go into bucket B2. The last two bits of 4 and 5 are 11. So, they will go into B3.



Now, to insert key 11 with hash address 10001 into the above structure follow the steps:-

- Since, hash address of the bucket is 10001. It will go into bucket B1. But B1 bucket is already filled, so it will get split.
- Three bits of 11 and 6 are 001. So, they will go into bucket B1. And last three bits of 2 are 101. So, it will go into B4.
- Keys 1 and 3 are still in bucket B0. The record B is pointed by 000 and 100 entry because last two bits of both the entry are 00.
- Key 7 is still in bucket B2. The record B2 is pointed by 010 and 110 entry because last two bits of both the entry are 10.
- Key 4 and 5 are still in bucket B3. The record B3 is pointed by 111 and 011 entry because last two bits of both the entry are 11.



Insert Key 11 in the Data Bucket

Frequently Asked Questions on Dynamic Hashing – FAQs

How is dynamic hashing different from static hashing?

In static hashing, the resultant data bucket address will remain same while in dynamic hashing, the data bucket size shrinks or grows while increase or decrease of records.

Which hashing method is used to access the dynamic files?

Extensible hashing approach simultaneously solves the problem of making hash tables that are extendible and of making radix search trees that are balanced. It can be used to access the dynamic files.

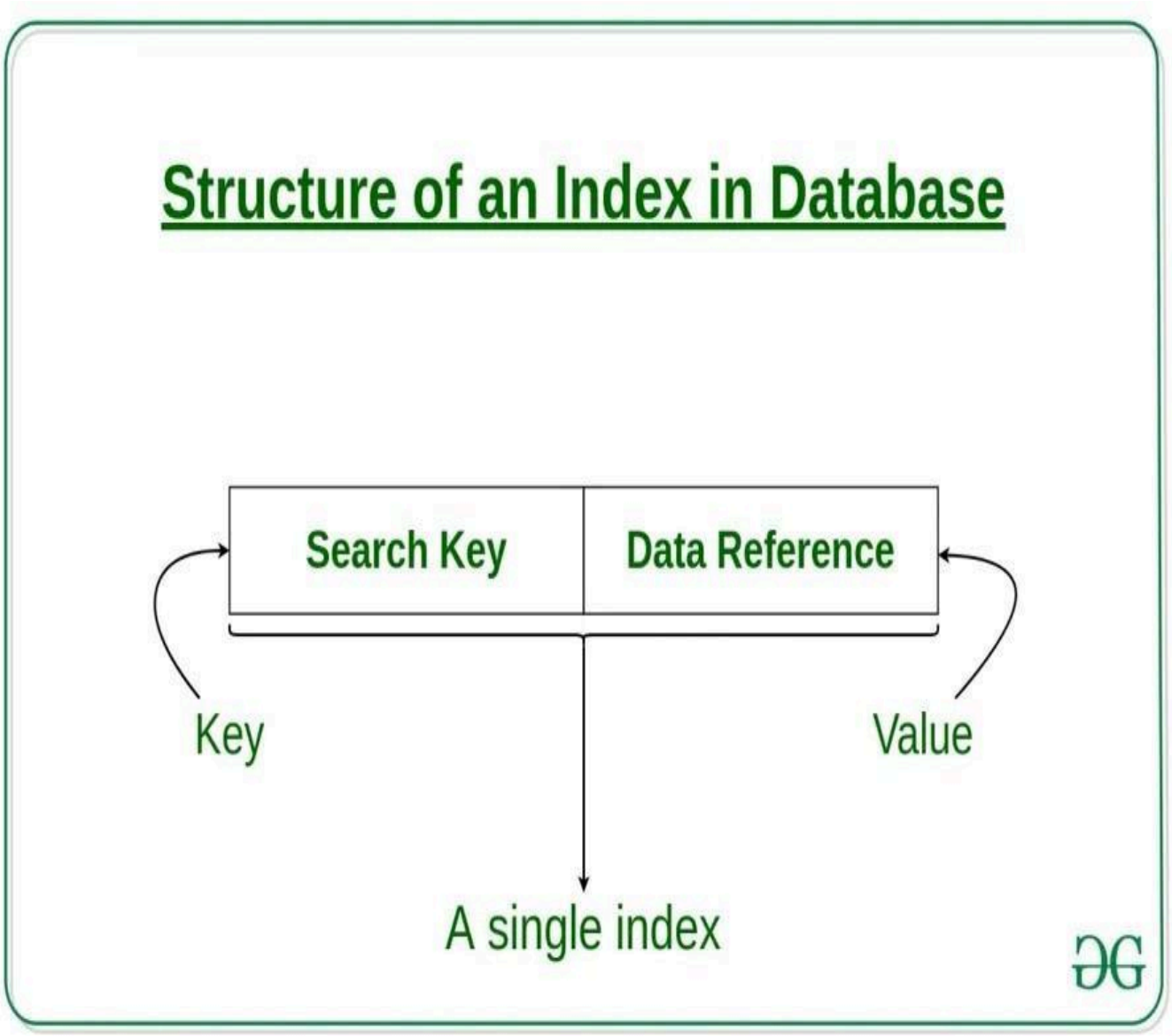
What are popular dynamic hashing techniques?

Some popular dynamic hashing techniques are linear hashing, extensible hashing and consistent hashing.

Indexing in Databases

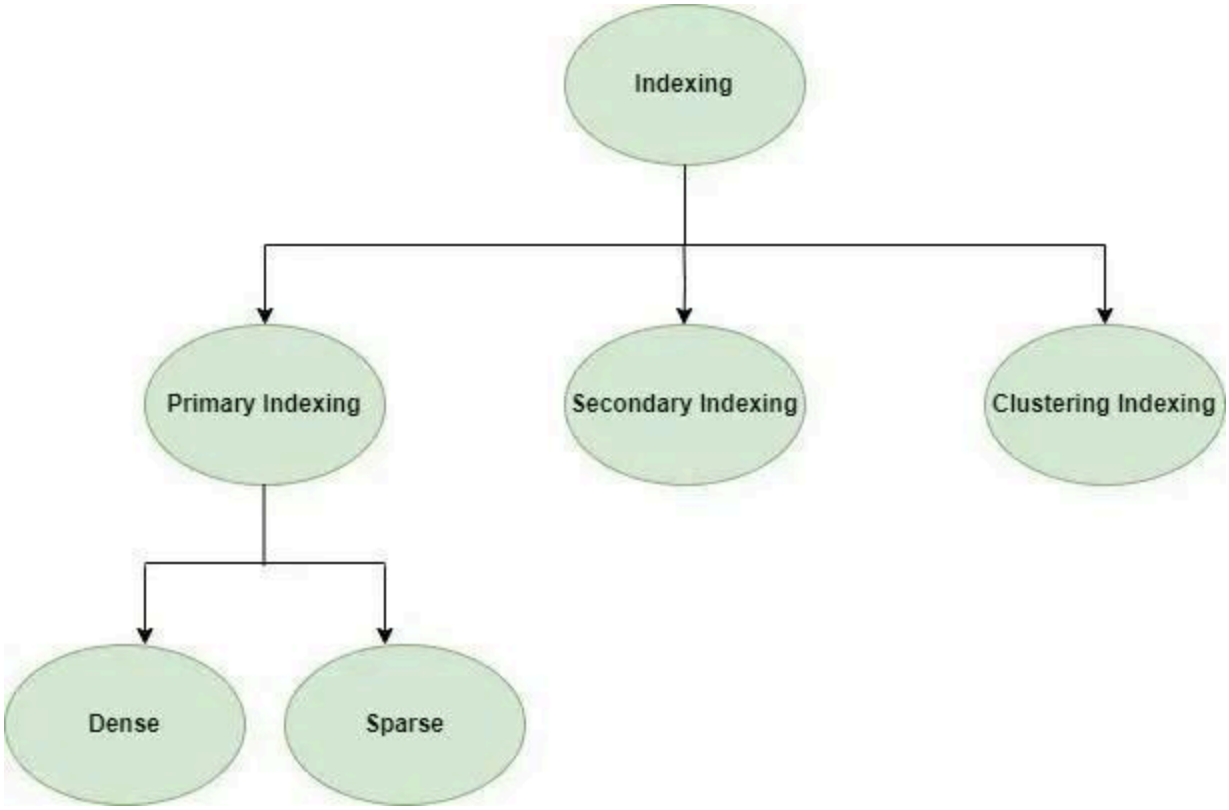
Indexing improves database performance by minimizing the number of disc visits required to fulfill a query. It is a data structure technique used to locate and quickly access data in databases. Several database fields are used to generate indexes. The main key or candidate key of the table is duplicated in the first column, which is the Search key. To speed up data retrieval, the values are also kept in sorted order. It should be highlighted that sorting the data is not required. The

second column is the Data Reference or Pointer which contains a set of pointers holding the address of the disk block where that particular key value can be found.



Attributes of Indexing

- **Access Types:** This refers to the type of access such as value-based search, range access, etc.
- **Access Time:** It refers to the time needed to find a particular data element or set of elements.
- **Insertion Time:** It refers to the time taken to find the appropriate space and insert new data.
- **Deletion Time:** Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead:** It refers to the additional space required by the index.

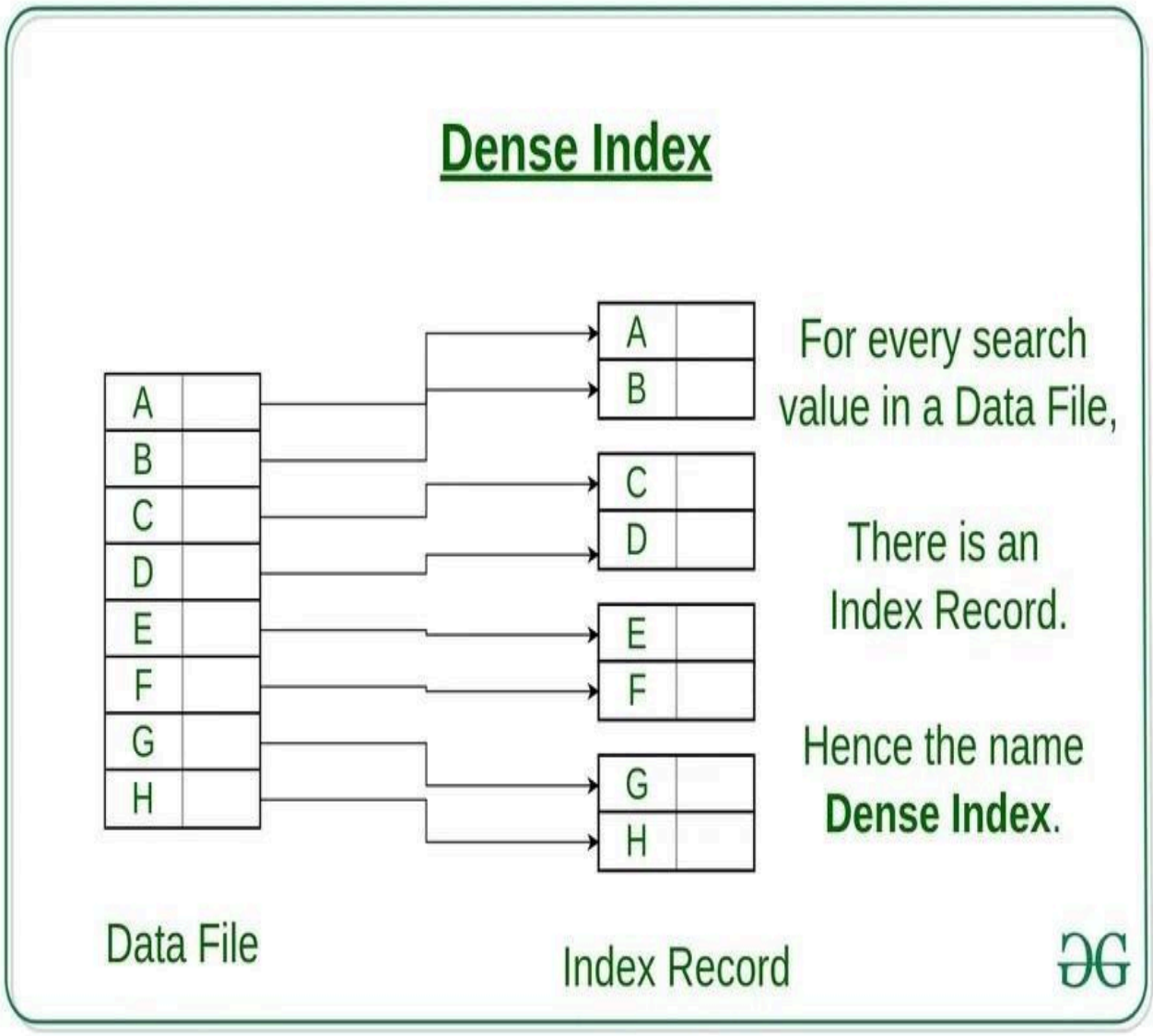


In general, there are two types of file organization mechanisms that are followed by the indexing methods to store the data:

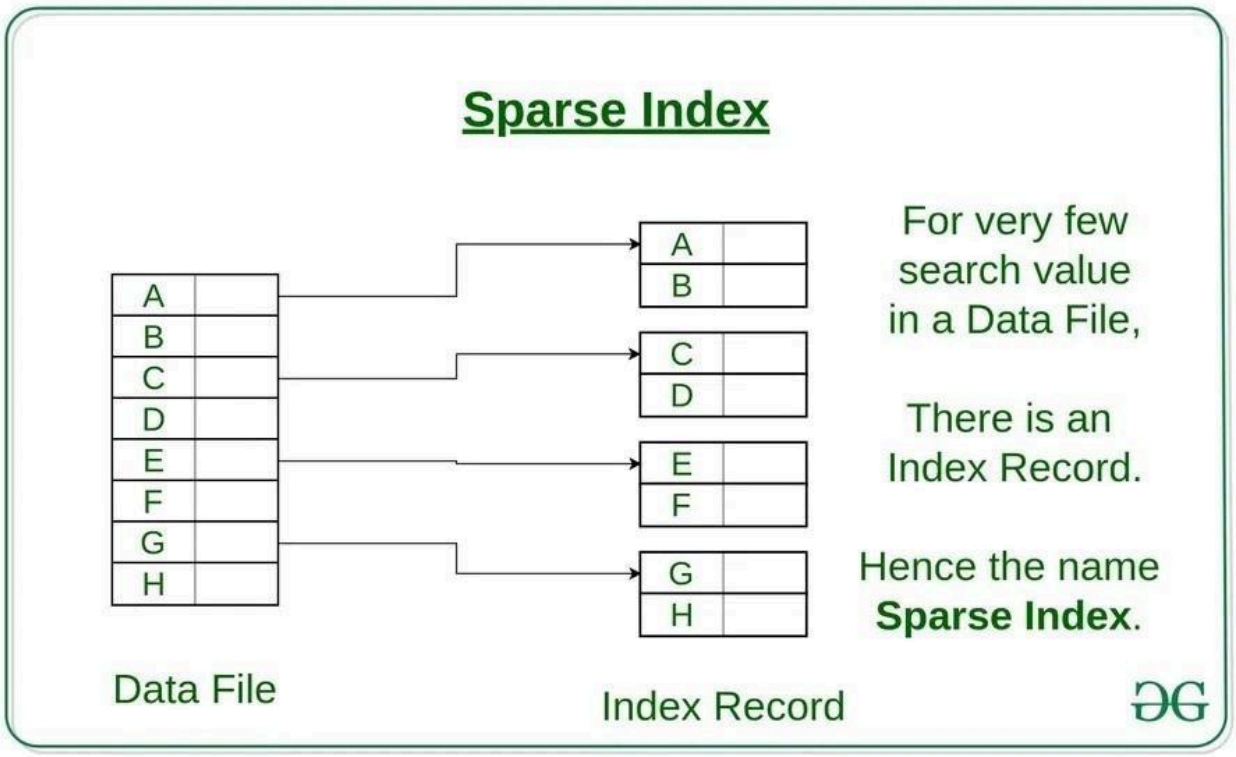
Sequential File Organization or Ordered Index File

In this, the indices are based on a sorted ordering of the values. These are generally fast and a more traditional type of storing mechanism. These Ordered or Sequential file organizations might store the data in a dense or sparse format.

- **Dense Index**
    - For every search key value in the data file, there is an index record.
    - This record contains the search key and also a reference to the first data record with that search key value.



- **Sparse Index**
  - The index record appears only for a few items in the data file. Each item points to a block as shown.
  - To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
  - We start at that record pointed to by the index record, and proceed along with the pointers in the file (that is, sequentially) until we find the desired record.
  - Number of Accesses required=$\log_2(n)+1$, (here n=number of blocks acquired by index file)

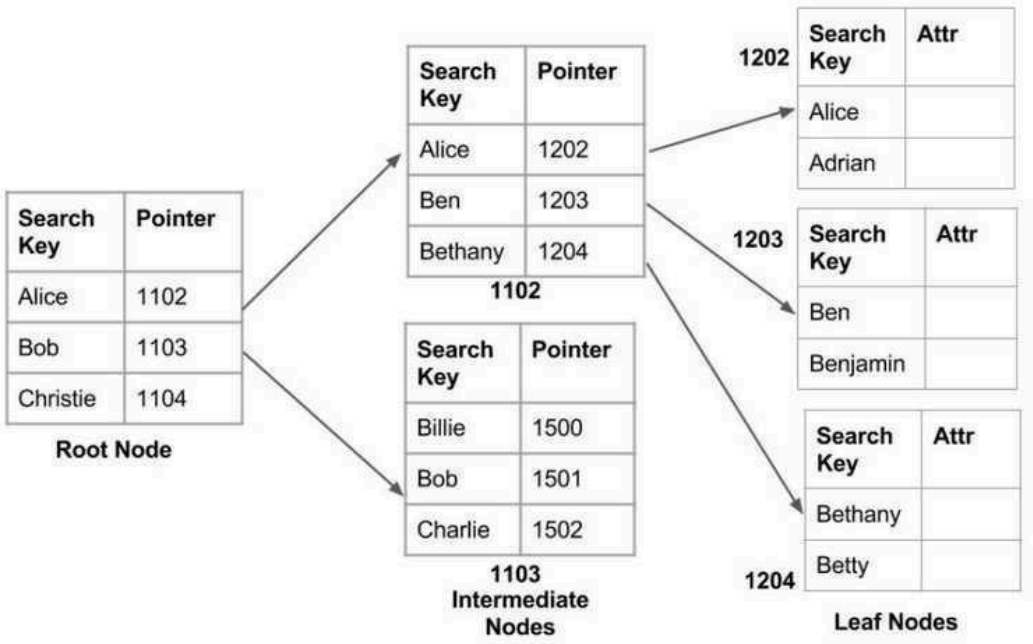


Hash File Organization

Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned are determined by a function called a hash function. There are primarily three methods of indexing:

- **Clustered Indexing**: When more than two records are stored in the same file this type of storing is known as cluster indexing. By using cluster indexing we can reduce the cost of searching reason being multiple records related to the same thing are stored in one place and it also gives the frequent joining of more than two tables (records).
  The clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create an index out of them. This method is known as the clustering index. Essentially, records with similar properties are grouped together, and indexes for these groupings are formed.
  Students studying each semester, for example, are grouped together. First-semester students, second-semester students, third-semester students, and so on are categorized.
- **Primary Indexing:** This is a type of Clustered Indexing wherein the data is sorted according to the search key and the primary key of the database table is used to create the index. It is a default format of indexing where it induces sequential file organization. As primary keys are unique and are stored in a sorted manner, the performance of the searching operation is quite efficient.

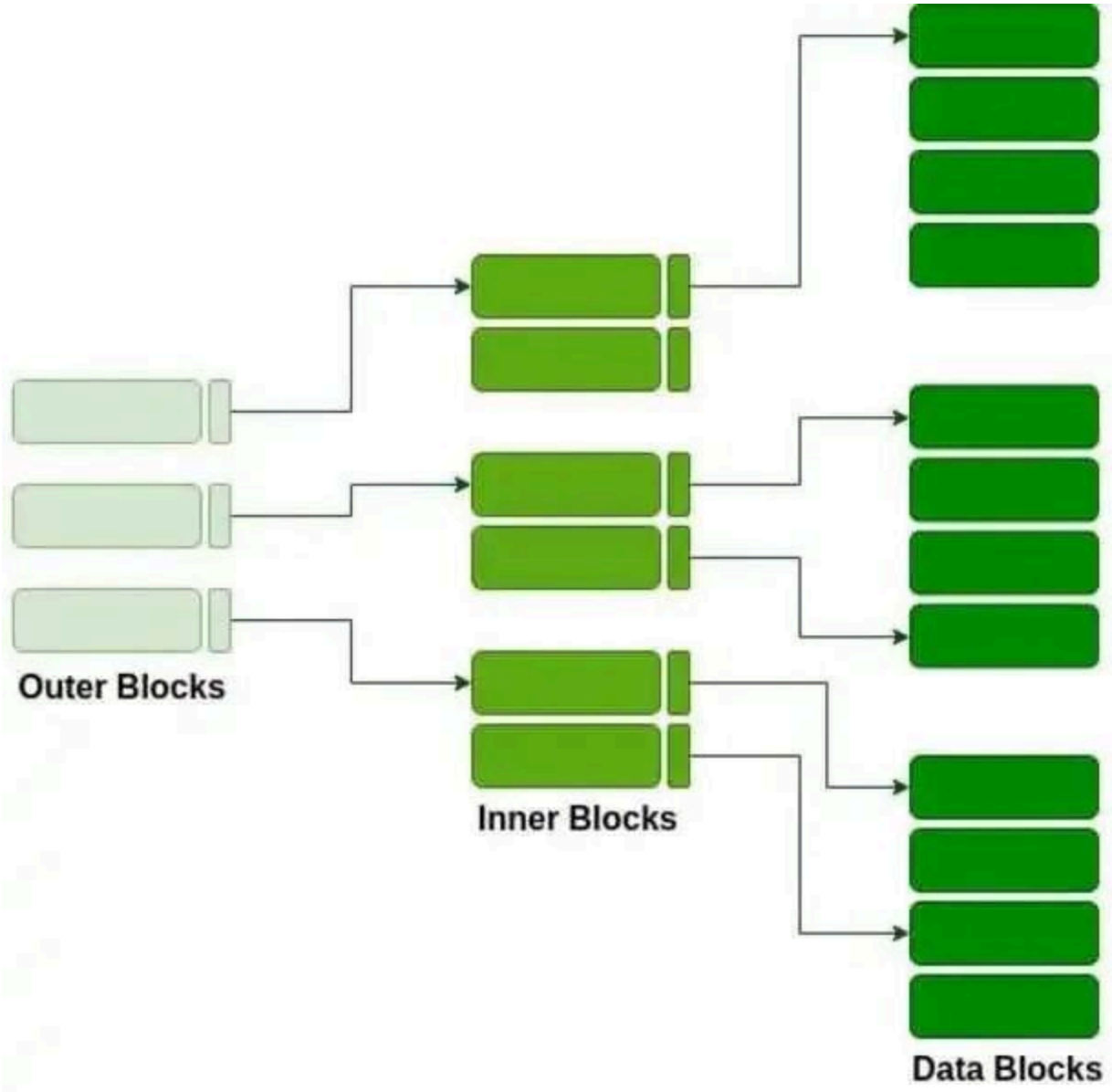
- **Non-clustered or Secondary Indexing**: A non-clustered index just tells us where the data lies, i.e. it gives us a list of virtual pointers or references to the location where the data is actually stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For eg. the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here(information on each page of the book) is not organized but we have an ordered reference(contents page) to where the data points actually lie. We can have only dense ordering in the non-clustered index as sparse ordering is not possible because data is not physically organized accordingly.
  It requires more time as compared to the clustered index because some amount of extra work is done in order to extract the data by further following the pointer. In the case of a clustered index, data is directly present in front of the index.
  -



Non clustered index

  -

- **Multilevel Indexing:** With the growth of the size of the database, indices also grow. As the index is stored in the main memory, a single-level index might become too large a size to store with multiple disk accesses. The multilevel indexing segregates the main block into various smaller blocks so that the same can be stored in a single block. The outer blocks are divided into inner blocks which in turn are pointed to the data blocks. This can be easily stored in the main memory with fewer overheads.



Advantages of Indexing

- **Improved Query Performance:** Indexing enables faster data retrieval from the database. The database may rapidly discover rows that match a specific value or collection of values by generating an index on a column, minimizing the amount of time it takes to perform a query.
- **Efficient Data Access:** Indexing can enhance data access efficiency by lowering the amount of disk I/O required to retrieve data. The database can maintain the data pages for frequently visited columns in memory by generating an index on those columns, decreasing the requirement to read from disk.
- **Optimized Data Sorting:** Indexing can also improve the performance of sorting operations. By creating an index on the columns used for sorting, the database can avoid sorting the entire table and instead sort only the relevant rows.
- **Consistent Data Performance:** Indexing can assist ensure that the database performs consistently even as the amount of data in the database rises. Without indexing, queries may

take longer to run as the number of rows in the table grows, while indexing maintains a roughly consistent speed.

- By ensuring that only unique values are inserted into columns that have been indexed as unique, indexing can also be utilized to ensure the integrity of data. This avoids storing duplicate data in the database, which might lead to issues when performing queries or reports.

Overall, indexing in databases provides significant benefits for improving query performance, efficient data access, optimized data sorting, consistent data performance, and enforced data integrity

Disadvantages of Indexing

- Indexing necessitates more storage space to hold the index data structure, which might increase the total size of the database.
- **Increased database maintenance overhead:** Indexes must be maintained as data is added, destroyed, or modified in the table, which might raise database maintenance overhead.
- Indexing can reduce insert and update performance since the index data structure must be updated each time data is modified.
- **Choosing an index can be difficult:** It can be challenging to choose the right indexes for a specific query or application and may call for a detailed examination of the data and access patterns.

Features of Indexing

- The development of data structures, such as B-trees or hash tables, that provide quick access to certain data items is known as indexing. The data structures themselves are built on the values of the indexed columns, which are utilized to quickly find the data objects.
- The most important columns for indexing columns are selected based on how frequently they are used and the sorts of queries they are subjected to. The cardinality, selectivity, and uniqueness of the indexing columns can be taken into account.
- There are several different index types used by databases, including primary, secondary, clustered, and non-clustered indexes. Based on the particular needs of the database system, each form of index offers benefits and drawbacks.
- For the database system to function at its best, periodic index maintenance is required. According to changes in the data and usage patterns, maintenance work involves building, updating, and removing indexes.
- Database query optimization involves indexing, which is essential. The query optimizer utilizes the indexes to choose the best execution strategy for a particular query based on the cost of accessing the data and the selectivity of the indexing columns.
- Databases make use of a range of indexing strategies, including covering indexes, index-only scans, and partial indexes. These techniques maximize the utilization of indexes for particular types of queries and data access.
- When non-contiguous data blocks are stored in an index, it can result in index fragmentation, which makes the index less effective. Regular index maintenance, such as defragmentation and reorganization, can decrease fragmentation.

Conclusion

Indexing is a very useful technique that helps in optimizing the search time in database queries. The table of database indexing consists of a search key and pointer. There are four types of indexing: Primary, Secondary Clustering, and Multivalued Indexing. Primary indexing is divided into two types, dense and sparse. Dense indexing is used when the index table contains records for every search key. Sparse indexing is used when the index table does not use a search key for

every record. Multilevel indexing uses B+ Tree. The main purpose of indexing is to provide better performance for data retrieval.

Introduction of B+ Tree

B + Tree is a variation of the B-tree data structure. In a B + tree, data pointers are stored only at the leaf nodes of the tree. In a **B+ tree structure** of a leaf node differs from the structure of internal nodes. The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record). The leaf nodes of the B+ tree are linked together to provide ordered access to the search field to the records. Internal nodes of a B+ tree are used to guide the search. Some search field values from the leaf nodes are repeated in the internal nodes of the B+ tree.

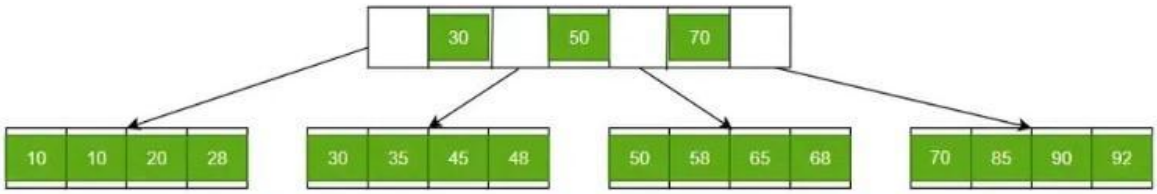
Features of B+ Trees

- **Balanced:** B+ Trees are self-balancing, which means that as data is added or removed from the tree, it automatically adjusts itself to maintain a balanced structure. This ensures that the search time remains relatively constant, regardless of the size of the tree.
- **Multi-level:** B+ Trees are multi-level data structures, with a root node at the top and one or more levels of internal nodes below it. The leaf nodes at the bottom level contain the actual data.
- **Ordered:** B+ Trees maintain the order of the keys in the tree, which makes it easy to perform range queries and other operations that require sorted data.
- **Fan-out:** B+ Trees have a high fan-out, which means that each node can have many child nodes. This reduces the height of the tree and increases the efficiency of searching and indexing operations.
- **Cache-friendly:** B+ Trees are designed to be cache-friendly, which means that they can take advantage of the caching mechanisms in modern computer architectures to improve performance.
- **Disk-oriented:** B+ Trees are often used for disk-based storage systems because they are efficient at storing and retrieving data from disk.

Why Use B+ Tree?

- B+ Trees are the best choice for storage systems with sluggish data access because they minimize I/O operations while facilitating efficient disc access.
- B+ Trees are a good choice for database systems and applications needing quick data retrieval because of their balanced structure, which guarantees predictable performance for a variety of activities and facilitates effective range-based queries.

Structure of B+ Trees

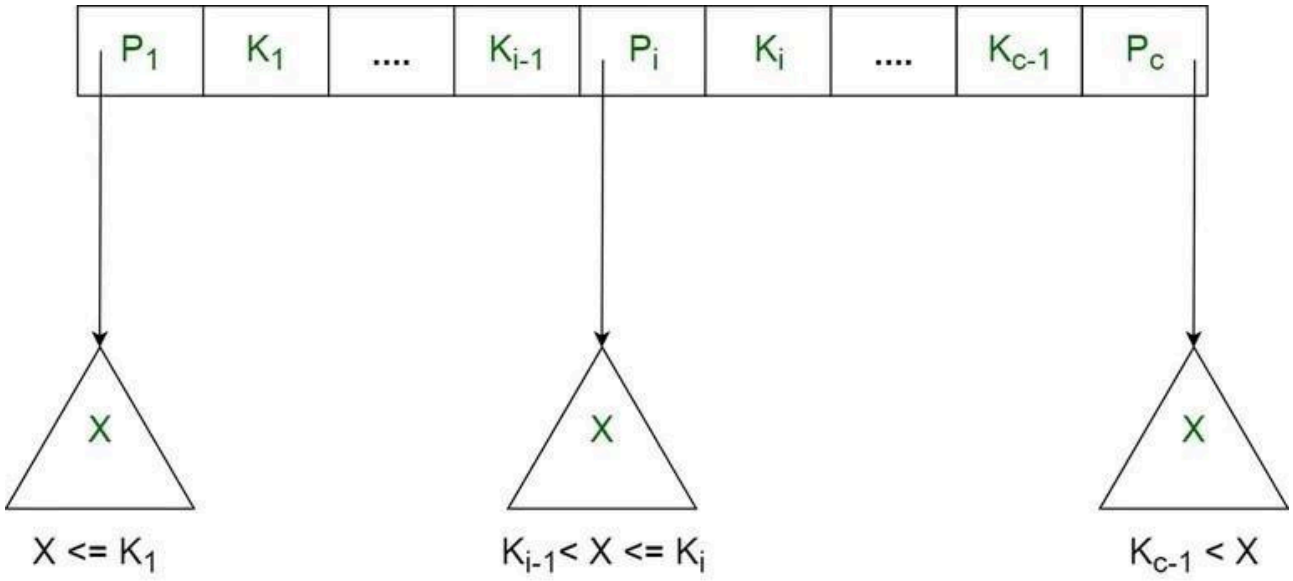


B+ Trees contain two types of nodes:

- **Internal Nodes:** Internal Nodes are the nodes that are present in at least n/2 record pointers, but not in the root node,
- **Leaf Nodes:** Leaf Nodes are the nodes that have n pointers.

The Structure of the Internal Nodes of a B+ Tree of Order 'a' is as Follows

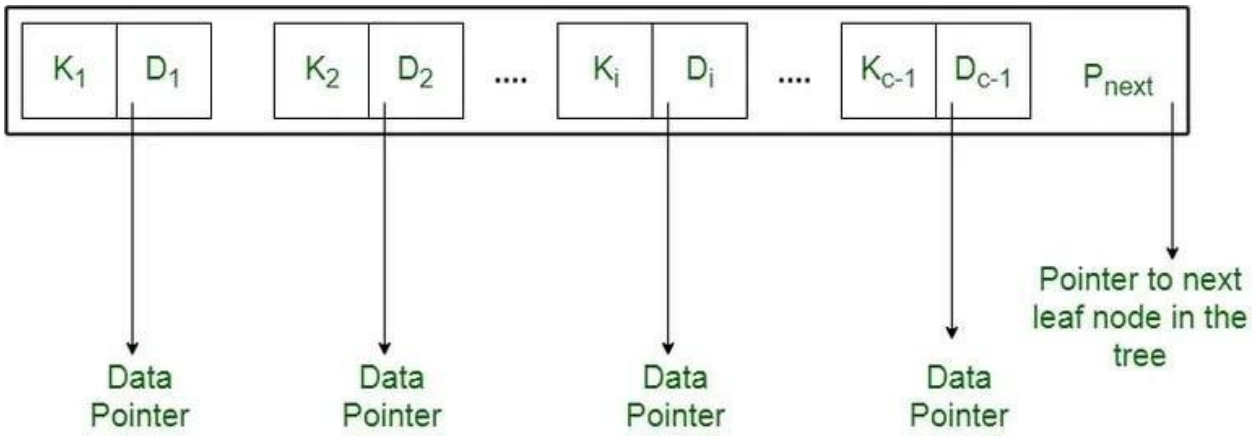
- Each internal node is of the form: <P1, K1, P2, K2, ….., Pc-1, Kc-1, Pc> where c <= a and each **Pi is a tree pointer (i.e points to another node of the tree)** and, each **Ki is a key-value** (see diagram-I for reference).
- Every internal node has : K1 < K2 < …. < Kc-1
- For each search field value 'X' in the sub-tree pointed at by Pi, the following condition holds: Ki-1 < X <= Ki, for 1 < I < c and, Ki-1 < X, for i = c (See diagram I for reference)
- Each internal node has at most 'aa tree pointers.
- The root node has, at least two tree pointers, while the other internal nodes have at least \ceil(a/2) tree pointers each.
- If an internal node has 'c' pointers, c <= a, then it has 'c – 1' key values.
-



Structure of Internal Node

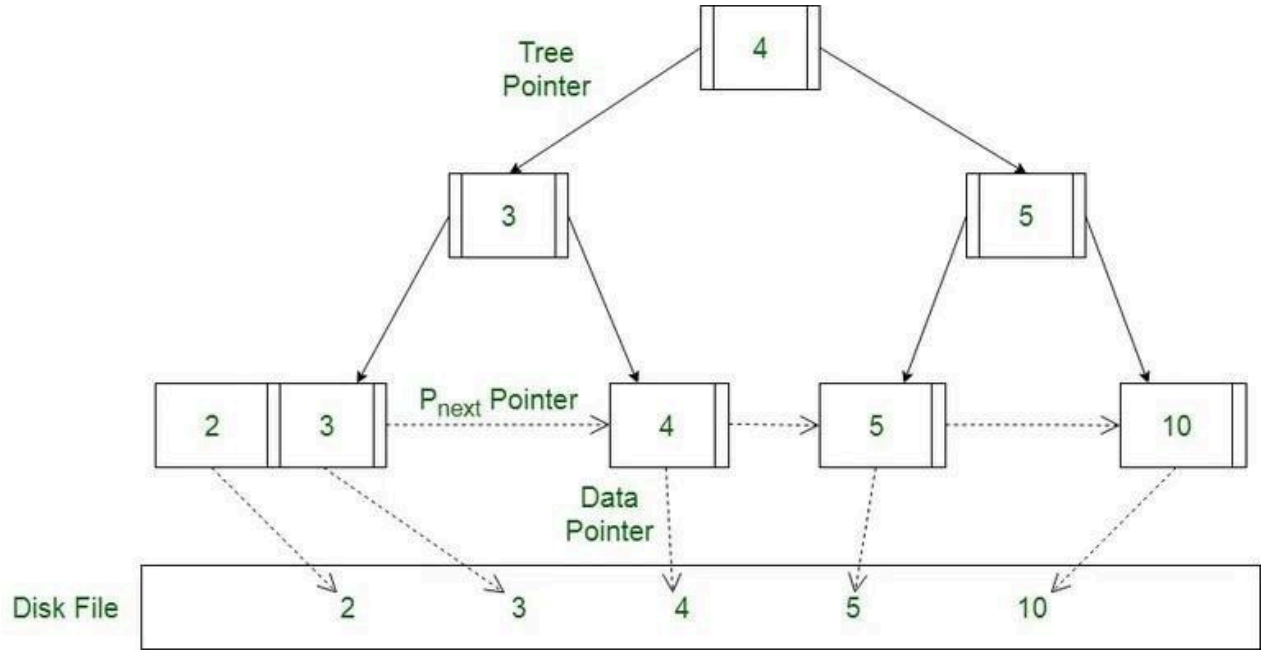
The Structure of the Leaf Nodes of a B+ Tree of Order 'b' is as Follows

- Each leaf node is of the form: <<K1, D1>, <K2, D2>, ….., <Kc-1, Dc-1>, Pnext> where c <= b and each **Di is a data pointer (i.e points to actual record in the disk whose key value is Ki or to a disk file block containing that record)** and, each **Ki is a key value** and, **Pnext points to next leaf node in the B+ tree** (see diagram II for reference).
- Every leaf node has : K1 < K2 < …. < Kc-1, c <= b
- Each leaf node has at least \ceil(b/2) values.
- All leaf nodes are at the same level.



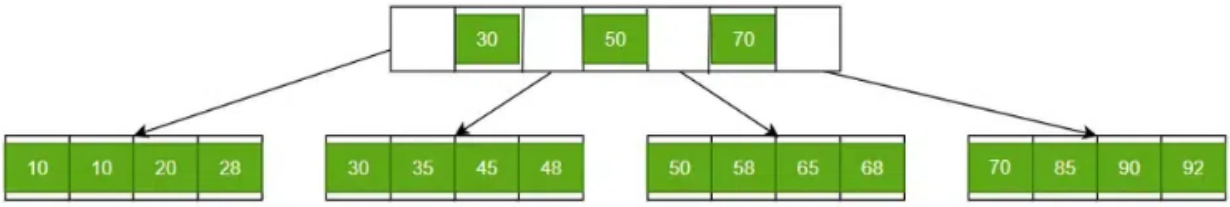
Structure of Lead Node

Diagram-II Using the Pnext pointer it is viable to traverse all the leaf nodes, just like a linked list, thereby achieving ordered access to the records stored in the disk.



Tree Pointer

Searching a Record in B+ Trees



Searching in B+ Tree

Let us suppose we have to find 58 in the B+ Tree. We will start by fetching from the root node then we will move to the leaf node, which might contain a record of 58. In the image given above, we will get 58 between 50 and 70. Therefore, we will we are getting a leaf node in the third leaf node and get 58 there. If we are unable to find that node, we will return that 'record not founded' message.

Insertion in B+ Trees

Insertion in B+ Trees is done via the following steps.

- Every element in the tree has to be inserted into a leaf node. Therefore, it is necessary to go to a proper leaf node.
- Insert the key into the leaf node in increasing order if there is no overflow.

For more, refer to Insertion in a B+ Trees.

Deletion in B+Trees

Deletion in B+ Trees is just not deletion but it is a combined process of Searching, Deletion, and Balancing. In the last step of the Deletion Process, it is mandatory to balance the B+ Trees, otherwise, it fails in the property of B+ Trees.

For more, refer to Deletion in B+ Trees.

Advantages of B+Trees

- A B+ tree with 'l' levels can store more entries in its internal nodes compared to a B-tree having the same 'l' levels. This accentuates the significant improvement made to the search time for any given key. Having lesser levels and the presence of Pnext pointers imply that the B+ trees is very quick and efficient in accessing records from disks.
- Data stored in a B+ tree can be accessed both sequentially and directly.
- It takes an equal number of disk accesses to fetch records.
- B+trees have redundant search keys, and storing search keys repeatedly is not possible.

Disadvantages of B+ Trees

- The major drawback of B-tree is the difficulty of traversing the keys sequentially. The B+ tree retains the rapid random access property of the B-tree while also allowing rapid sequential access.

Application of B+ Trees

- Multilevel Indexing
- Faster operations on the tree (insertion, deletion, search)
- Database indexing

Conclusion

In conclusion, B+ trees are an essential component of contemporary database systems since they significantly improve database performance and make efficient data management possible.

Introduction of B-Tree



The limitations of traditional binary search trees can be frustrating. Meet the B-Tree, the multi-talented data structure that can handle massive amounts of data with ease. When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage. B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.

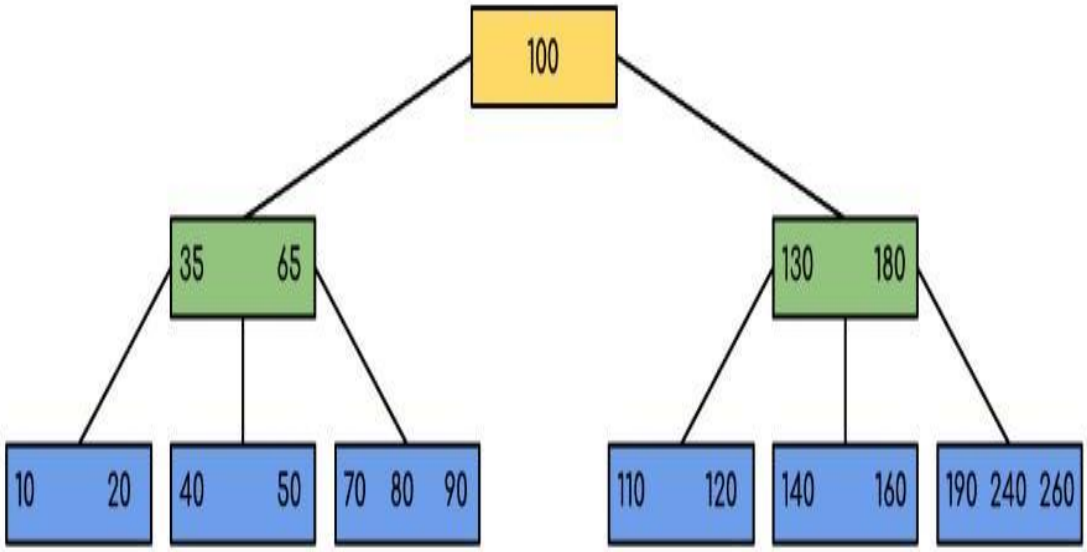
Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as "large key" trees. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height. This shallow height leads to less disk I/O, which results in faster search

and insertion operations. B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.
B-Trees maintains balance by ensuring that each node has a minimum number of keys, so the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.
Time Complexity of B-Tree:

Sr. No.	**Algorithm**	**Time Complexity**
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

Note: “n” is the total number of elements in the B-tree
Properties of B-Tree:

- All leaves are at the same level.
- B-Tree is defined by the term minimum degree ‘**t**‘. The value of ‘**t**‘ depends upon disk block size.
- Every node except the root must contain at least t-1 keys. The root may contain a minimum of **1** key.
- All nodes (including root) may contain at most (**2*t – 1**) keys.
- Number of children of a node is equal to the number of keys in it plus **1**.
- All keys of a node are sorted in increasing order. The child between two keys **k1** and **k2** contains all keys in the range from **k1** and **k2**.
- B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- Like other balanced Binary Search Trees, the time complexity to search, insert, and delete is O(log n).
- Insertion of a Node in B-Tree happens only at Leaf Node.



We can see in the above diagram that all the leaf nodes are at the same level and all non-leafs have no empty sub-tree and have keys one less than the number of their children.

Interesting Facts about B-Trees:

- *The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have is:*
- *The maximum height of the B-Tree that can exist with n number of nodes*

Traversal in B-Tree:

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for the remaining children and keys. In the end, recursively print the rightmost child.

Search Operation in B-Tree:

Search is similar to the search in Binary Search Tree. Let the key to be searched is k.

- Start from the root and recursively traverse down.
- For every visited non-leaf node,
    - If the node has the key, we simply return the node.
    - Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node.
- If we reach a leaf node and don't find k in the leaf node, then return NULL.

Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of the parent then the key is present in another branch. As these values limit the search they are also known as limiting values or separation values. If we reach a leaf node and don't find the desired key then it will display NULL.

MODULE 6

Concurrency Control in DBMS

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.

What is Transaction?

A transaction is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. However, during the execution of a transaction, it may be necessary temporarily to allow inconsistency, since either the debit of A or the credit of B must be done before the other. This temporary inconsistency, although necessary, may lead to difficulty if a failure occurs.

It is the programmer's responsibility to define properly the various transactions, so that each preserves the consistency of the database. For example, the transaction to transfer funds from the account of department A to the account of department B could be defined to be composed of two separate programs: one that debits account A, and another that credits account B. The execution of these two programs one after the other will indeed preserve consistency. However, each program by itself does not transform the database from a consistent state to a new consistent state. Thus, those programs are not transactions.

The concept of a transaction has been applied broadly in database systems and applications. While the initial use of transactions was in financial applications, the concept is now used in real-time applications in telecommunication, as well as in the management of long-duration activities such as product design or administrative workflows.

A set of logically related operations is known as a transaction. The main operations of a transaction are:

- **Read(A):** Read operations Read(A) or R(A) reads the value of A from the database and stores it in a buffer in the main memory.
- **Write (A):** Write operation Write(A) or W(A) writes the value back to the database from the buffer.

(Note: It doesn't always need to write it to a database back it just writes the changes to buffer this is the reason where dirty read comes into the picture)

Let us take a debit transaction from an account that consists of the following operations:

1. R(A);
2. A=A-1000;
3. W(A);

Assume A's value before starting the transaction is 5000.

- The first operation reads the value of A from the database and stores it in a buffer.
- the Second operation will decrease its value by 1000. So buffer will contain 4000.
- the Third operation will write the value from the buffer to the database. So A's final value will be 4000.

But it may also be possible that the transaction may fail after executing some of its operations. The failure can be because of **hardware, software or power,** etc. For example, if the debit transaction discussed above fails after executing operation 2, the value of A will remain 5000 in the database which is not acceptable by the bank. To avoid this, Database has two important operations:

- **Commit:** After all instructions of a transaction are successfully executed, the changes made by a transaction are made permanent in the database.
- **Rollback:** If a transaction is not able to execute all operations successfully, all the changes made by a transaction are undone.

For more details please refer Transaction Control in DBMS article.

Properties of a Transaction

Atomicity: As a transaction is a set of logically related operations, **either all of them should be executed or none**. A debit transaction discussed above should either execute all three operations or none. If the debit transaction fails after executing operations 1 and 2 then its new value of 4000 will not be updated in the database which leads to inconsistency.

Consistency: If operations of debit and credit transactions on the same account are executed concurrently, it may leave the database in an inconsistent state.

- For Example, with T1 (debit of Rs. 1000 from A) and T2 (credit of 500 to A) executing concurrently, the database reaches an inconsistent state.
- Let us assume the Account balance of A is Rs. 5000. T1 reads A(5000) and stores the value in its local buffer space. Then T2 reads A(5000) and also stores the value in its local buffer space.
- T1 performs A=A-1000 (5000-1000=4000) and 4000 is stored in T1 buffer space. Then T2 performs A=A+500 (5000+500=5500) and 5500 is stored in the T2 buffer space. T1 writes the value from its buffer back to the database.
- A's value is updated to 4000 in the database and then T2 writes the value from its buffer back to the database. A's value is updated to 5500 which shows that the effect of the debit transaction is lost and the database has become inconsistent.
- To maintain consistency of the database, we need **concurrency control protocols** which will be discussed in the next article. The operations of T1 and T2 with their buffers and database have been shown in Table 1.

T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
	A=5000	R(A);	A=5000	A=5000
A=A-1000;	A=4000		A=5000	A=5000
	A=4000	A=A+500;	A=5500	
W(A);			A=5500	A=4000
		W(A);		A=5500

Isolation: The result of a transaction should not be visible to others before the transaction is committed. For example, let us assume that A's balance is Rs. 5000 and T1 debits Rs. 1000 from A. A's new balance will be 4000. If T2 credits Rs. 500 to A's new balance, A will become 4500, and after this T1 fails. Then we have to roll back T2 as well because it is using the value produced by T1. So transaction results are not made visible to other transactions before it commits.

Durable: Once the database has committed a transaction, the changes made by the transaction should be permanent. e.g.; If a person has credited $500000 to his account, the bank can't say that the update has been lost. To avoid this problem, multiple copies of the database are stored at different locations.

What is a Schedule?

A schedule is a series of operations from one or more transactions. A schedule can be of two types:

Serial Schedule: When one transaction completely executes before starting another transaction, the schedule is called a serial schedule. A serial schedule is always consistent. e.g.; If a schedule S has debit transaction T1 and credit transaction T2, possible serial schedules are T1 followed by T2 (T1->T2) or T2 followed by T1 ((T2->T1). A serial schedule has low throughput and less resource utilization.

Concurrent Schedule: When operations of a transaction are interleaved with operations of other transactions of a schedule, the schedule is called a Concurrent schedule. e.g.; the Schedule of debit and credit transactions shown in Table 1 is concurrent. But concurrency can lead to inconsistency in the database. The above example of a concurrent schedule is also inconsistent.

Difference between Serial Schedule and Serializable Schedule

Serial Schedule	Serializable Schedule
In Serial schedule, transactions will be executed one after other.	In Serializable schedule transaction are executed concurrently.
Serial schedule are less efficient.	Serializable schedule are more efficient.
In serial schedule only one transaction executed at a time.	In Serializable schedule multiple transactions can be executed at a time.
Serial schedule takes more time for execution.	In Serializable schedule execution is fast.

Concurrency Control in DBMS

- Executing a single transaction at a time will increase the waiting time of the other transactions which may result in delay in the overall execution. Hence for increasing the overall throughput and efficiency of the system, several transactions are executed.
- 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 provides a procedure that is able to control concurrent execution of the operations in the database.
- The fundamental goal of database concurrency control is to ensure that concurrent execution of transactions does not result in a loss of database consistency. The concept of serializability can be used to achieve this goal, since all serializable schedules preserve consistency of the database. However, not all schedules that preserve consistency of the database are serializable.
- In general it is not possible to perform an automatic analysis of low-level operations by transactions and check their effect on database consistency constraints. However, there are simpler techniques. One is to use the database consistency constraints as the basis for a split of the database into subdatabases on which concurrency can be managed separately.
- Another is to treat some operations besides read and write as fundamental low-level operations and to extend concurrency control to deal with them.

Concurrency Control Problems

There are several problems that arise when numerous transactions are executed simultaneously in a random manner. The database transaction consist of two major operations “Read” and “Write”. It is very important to manage these operations in the concurrent execution of the transactions in order to maintain the consistency of the data.

Dirty Read Problem(Write-Read conflict)

Dirty read problem occurs when one transaction updates an item but due to some unconditional events that transaction fails but before the transaction performs rollback, some other transaction reads the updated value. Thus creates an inconsistency in the database. Dirty read problem comes under the scenario of Write-Read conflict between the transactions in the database

1. The lost update problem can be illustrated with the below scenario between two transactions T1 and T2.

2. Transaction T1 modifies a database record without committing the changes.
3. T2 reads the uncommitted data changed by T1
4. T1 performs rollback
5. T2 has already read the uncommitted data of T1 which is no longer valid, thus creating inconsistency in the database.

Lost Update Problem

Lost update problem occurs when two or more transactions modify the same data, resulting in the update being overwritten or lost by another transaction. The lost update problem can be illustrated with the below scenario between two transactions T1 and T2.

1. T1 reads the value of an item from the database.
2. T2 starts and reads the same database item.
3. T1 updates the value of that data and performs a commit.
4. T2 updates the same data item based on its initial read and performs commit.
5. This results in the modification of T1 gets lost by the T2's write which causes a lost update problem in the database.

Concurrency Control Protocols

Concurrency control protocols are the set of rules which are maintained in order to solve the concurrency control problems in the database. It ensures that the concurrent transactions can execute properly while maintaining the database consistency. The concurrent execution of a transaction is provided with atomicity, consistency, isolation, durability, and serializability via the concurrency control protocols.

- Locked based concurrency control protocol
- Timestamp based concurrency control protocol

Locked based Protocol

In locked based protocol, each transaction needs to acquire locks before they start accessing or modifying the data items. There are two types of locks used in databases.

- **Shared Lock :** Shared lock is also known as read lock which allows multiple transactions to read the data simultaneously. The transaction which is holding a shared lock can only read the data item but it can not modify the data item.
- **Exclusive Lock :** Exclusive lock is also known as the write lock. Exclusive lock allows a transaction to update a data item. Only one transaction can hold the exclusive lock on a data item at a time. While a transaction is holding an exclusive lock on a data item, no other transaction is allowed to acquire a shared/exclusive lock on the same data item.

There are two kind of lock based protocol mostly used in database:

- **Two Phase Locking Protocol :** Two phase locking is a widely used technique which ensures strict ordering of lock acquisition and release. Two phase locking protocol works in two phases.
    - **Growing Phase :** In this phase, the transaction starts acquiring locks before performing any modification on the data items. Once a transaction acquires a lock, that lock can not be released until the transaction reaches the end of the execution.
    - **Shrinking Phase :** In this phase, the transaction releases all the acquired locks once it performs all the modifications on the data item. Once the transaction starts releasing the locks, it can not acquire any locks further.
- **Strict Two Phase Locking Protocol :** It is almost similar to the two phase locking protocol the only difference is that in two phase locking the transaction can release its locks before it

commits, but in case of strict two phase locking the transactions are only allowed to release the locks only when they performs commits.

Timestamp based Protocol

- In this protocol each transaction has a timestamp attached to it. Timestamp is nothing but the time in which a transaction enters into the system.
- The conflicting pairs of operations can be resolved by the timestamp ordering protocol through the utilization of the timestamp values of the transactions. Therefore, guaranteeing that the transactions take place in the correct order.

Advantages of Concurrency

In general, concurrency means, that more than one transaction can work on a system. The advantages of a concurrent system are:

- **Waiting Time:** It means if a process is in a ready state but still the process does not get the system to get execute is called waiting time. So, concurrency leads to less waiting time.
- **Response Time:** The time wasted in getting the response from the cpu for the first time, is called response time. So, concurrency leads to less Response Time.
- **Resource Utilization:** The amount of Resource utilization in a particular system is called Resource Utilization. Multiple transactions can run parallel in a system. So, concurrency leads to more Resource Utilization.
- **Efficiency:** The amount of output produced in comparison to given input is called efficiency. So, Concurrency leads to more Efficiency.

Disadvantages of Concurrency

- **Overhead:** Implementing concurrency control requires additional overhead, such as acquiring and releasing locks on database objects. This overhead can lead to slower performance and increased resource consumption, particularly in systems with high levels of concurrency.
- **Deadlocks:** Deadlocks can occur when two or more transactions are waiting for each other to release resources, causing a circular dependency that can prevent any of the transactions from completing. Deadlocks can be difficult to detect and resolve, and can result in reduced throughput and increased latency.
- **Reduced concurrency:** Concurrency control can limit the number of users or applications that can access the database simultaneously. This can lead to reduced concurrency and slower performance in systems with high levels of concurrency.
- **Complexity:** Implementing concurrency control can be complex, particularly in distributed systems or in systems with complex transactional logic. This complexity can lead to increased development and maintenance costs.
- **Inconsistency:** In some cases, concurrency control can lead to inconsistencies in the database. For example, a transaction that is rolled back may leave the database in an inconsistent state, or a long-running transaction may cause other transactions to wait for extended periods, leading to data staleness and reduced accuracy.

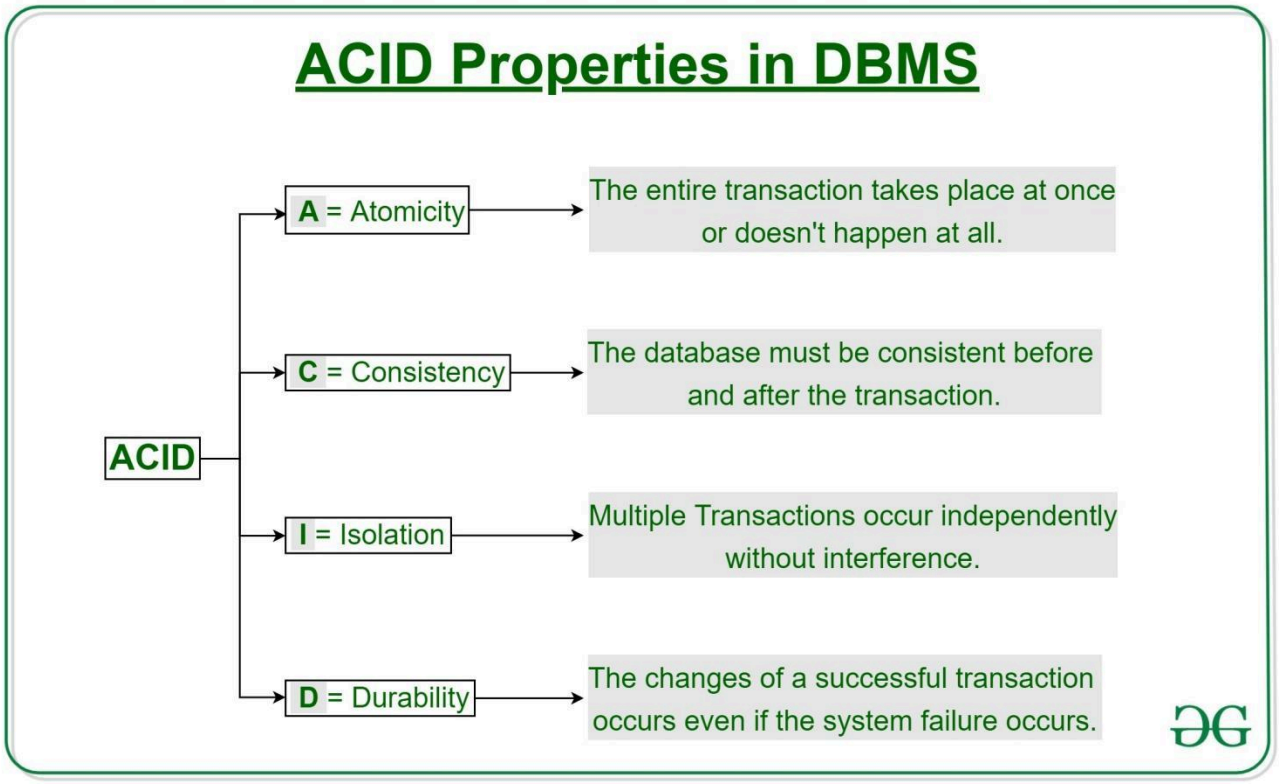
Conclusion

Concurrency control ensures transaction atomicity, isolation, consistency, and serializability. Concurrency control issues occur when many transactions execute randomly. A dirty read happens when a transaction reads data changed by an uncommitted transaction. When two transactions update data simultaneously, the Lost Update issue occurs. Lock-based protocol prevents incorrect read/write activities. Timestamp-based protocols organise transactions by timestamp.

ACID Properties in DBMS

A **transaction** is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.

In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.



Atomicity:

By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

—**Abort**: If a transaction aborts, changes made to the database are not visible.

—**Commit**: If a transaction commits, changes made are visible.

Atomicity is also known as the 'All or nothing rule'.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then the amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in its entirety in order to ensure the correctness of the database state.

Consistency:

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,
The total amount before and after the transaction must be maintained.
Total **before T** occurs = **500 + 200 = 700**.
Total **after T occurs** = **400 + 300 = 700**.
Therefore, the database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete.

Isolation:

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.
Let **X**= 500, **Y** = 500.
Consider two transactions **T** and **T".**

T	T"
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write (Y)	Read (X) Read (Y) Z: = X + Y Write (Z)

Suppose **T** has been executed till **Read (Y)** and then **T"** starts. As a result, interleaving of operations takes place due to which **T"** reads the correct value of **X** but the incorrect value of **Y** and sum computed by
T": (X+Y = 50, 000+500=50, 500)
is thus not consistent with the sum at end of the transaction:

T: (X+Y = 50, 000 + 450 = 50, 450).
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

Some important points:

Property	**Responsibility for maintaining properties**
Atomicity	Transaction Manager
Consistenc y	Application programmer
Isolation	Concurrency Control Manager
Durability	Recovery Manager

The **ACID** properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored.

ACID properties are the four key characteristics that define the reliability and consistency of a transaction in a Database Management System (DBMS). The acronym ACID stands for Atomicity, Consistency, Isolation, and Durability. Here is a brief description of each of these properties:

1. Atomicity: Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all the operations within the transaction are completed successfully, or none of them are. If any part of the transaction fails, the entire transaction is rolled back to its original state, ensuring data consistency and integrity.
2. Consistency: Consistency ensures that a transaction takes the database from one consistent state to another consistent state. The database is in a consistent state both before and after the transaction is executed. Constraints, such as unique keys and foreign keys, must be maintained to ensure data consistency.
3. Isolation: Isolation ensures that multiple transactions can execute concurrently without interfering with each other. Each transaction must be isolated from other transactions until it is completed. This isolation prevents dirty reads, non-repeatable reads, and phantom reads.
4. Durability: Durability ensures that once a transaction is committed, its changes are permanent and will survive any subsequent system failures. The transaction's changes are saved to the database permanently, and even if the system crashes, the changes remain intact and can be recovered.

Overall, ACID properties provide a framework for ensuring data consistency, integrity, and reliability in DBMS. They ensure that transactions are executed in a reliable and consistent manner, even in the presence of system failures, network issues, or other problems. These properties make DBMS a reliable and efficient tool for managing data in modern organizations.

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:

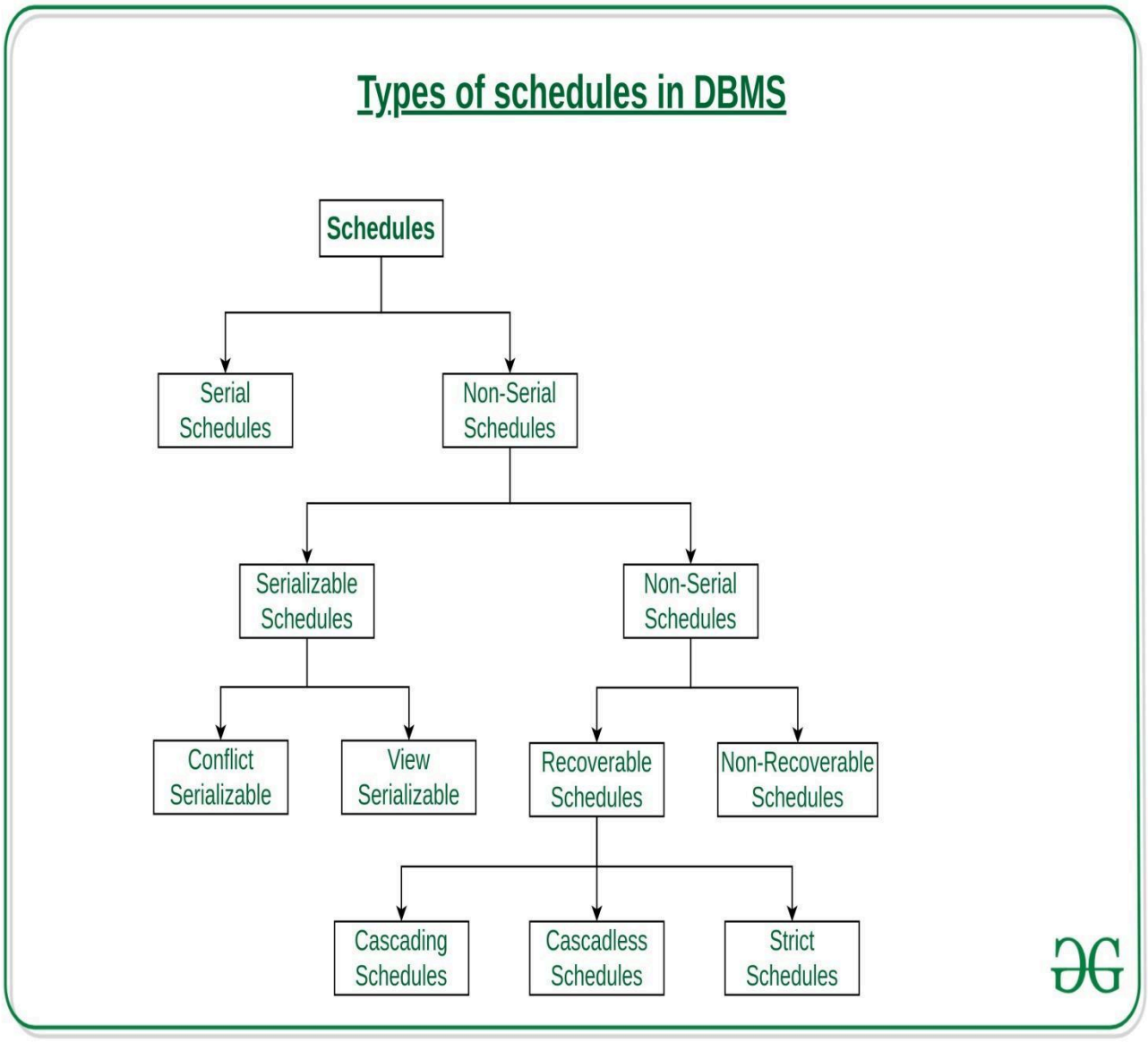
1. Performance: The ACID properties can cause a performance overhead in the system, as they require additional processing to ensure data consistency and integrity.
2. Scalability: The ACID properties may cause scalability issues in large distributed systems where multiple transactions occur concurrently.
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
4. 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

Types of Schedules in DBMS

Last Updated : 04 Feb, 2020



Schedule, as the name suggests, is a process of lining the transactions and executing them one by one. When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other, Scheduling is brought into play and the transactions are timed accordingly. The basics of Transactions and Schedules is discussed in Concurrency Control (Introduction), and Transaction Isolation Levels in DBMS articles. Here we will discuss various types of schedules.



1. **Serial Schedules:**
   Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.
   Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	

T_1	T_2
R(B)	
	W(B)
	R(A)
	R(B)

where R(A) denotes that a read operation is performed on some data item 'A'
This is a serial schedule since the transactions perform serially in the order T_1 —> T_2

2. **Non-Serial Schedule:**
   This is a type of Scheduling where the operations of multiple transactions are interleaved. This might lead to a rise in the concurrency problem. The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule. Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete. This sort of schedule does not provide any benefit of the concurrent transaction. It can be of two types namely, Serializable and Non-Serializable Schedule.
   The Non-Serial Schedule can be divided further into Serializable and Non-Serializable.

   a. **Serializable:**
      This is used to maintain the consistency of the database. It is mainly used in the Non-Serial scheduling to verify whether the scheduling will lead to any inconsistency or not. On the other hand, a serial schedule does not need the serializability because it follows a transaction only when the previous transaction is complete. The non-serial schedule is said to be in a serializable schedule only when it is equivalent to the serial schedules, for an n number of transactions. Since concurrency is allowed in this case thus, multiple transactions can execute concurrently. A serializable schedule helps in improving both resource utilization and CPU throughput. These are of two types:
      1. Conflict Serializable:
         A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations. Two operations are said to be conflicting if all conditions satisfy:
         - They belong to different transactions
         - They operate on the same data item
         - At Least one of them is a write operation
      2. View Serializable:
         A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions). A conflict schedule is a view serializable but if the serializability contains blind writes, then the view serializable does not conflict serializable.

b. **Non-Serializable:**
The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.

1. Recoverable Schedule:
Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.
Example – Consider the following schedule involving two transactions T_1 and T_2.

T_1	**T_2**
R(A)	
W(A)	
	W(A)
	R(A)
commit	
	commit

This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct.
There can be three types of recoverable schedule:

a. **Cascading Schedule:**
Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or

cascading abort. Example:

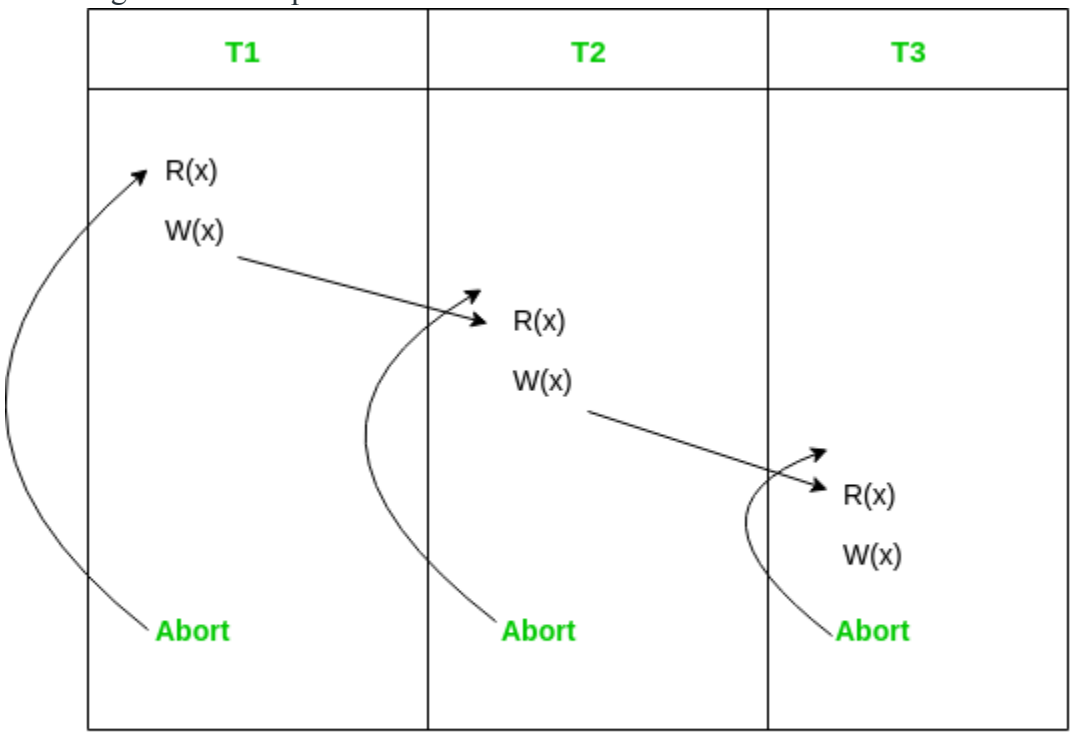


Figure - Cascading Abort

a. Cascadeless Schedule:
Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.
In other words, if some transaction T_j wants to read value updated or written by some other transaction T_i, then the commit of T_j must read it after the commit of T_i.
Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	**T_2**
R(A)	
W(A)	
	W(A)

T_1	T_2
commit	
	R(A)
	commit

This schedule is cascadeless. Since the updated value of **A** is read by T_2 only after the updating transaction i.e. T_1 commits.

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
abort	
	abort

It is a recoverable schedule but it does not avoid cascading aborts. It can be seen that if T_1 aborts, T_2 will have to be aborted too in order to maintain the correctness of the schedule as T_2 has already read the uncommitted value written by T_1.

b. **Strict Schedule:**

A schedule is strict if for any two transactions T_i, T_j, if a write operation of T_i precedes a conflicting operation of T_j (either read or write), then the commit or abort event of T_i also precedes that conflicting operation of T_j.

In other words, T_j can read or write updated or written value of T_i only after T_i commits/aborts.

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	

T_1	T_2
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

This is a strict schedule since T_2 reads and writes A which is written by T_1 only after the commit of T_1.

Non-Recoverable Schedule:

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	

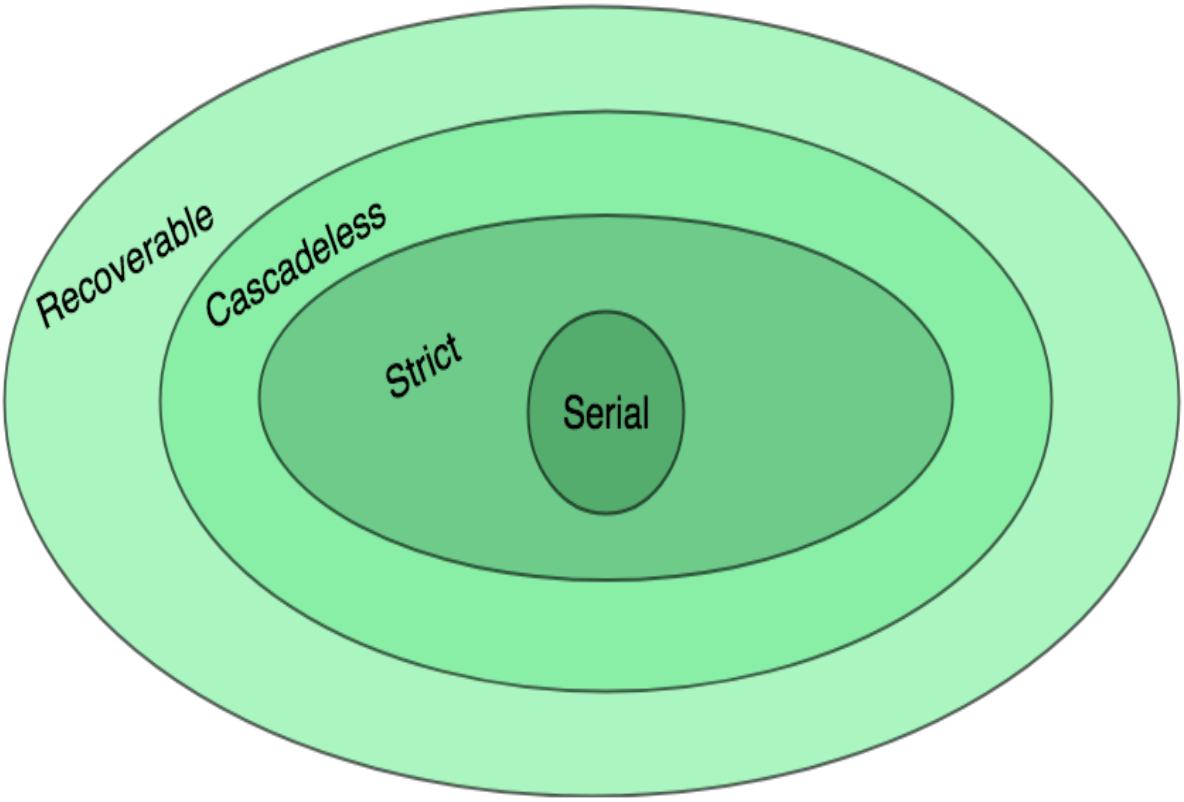
T_2 read the value of A written by T_1, and committed. T_1 later aborted, therefore the value read by T_2 is wrong, but since T_2 committed, this schedule is **non-recoverable**.

Note – It can be seen that:

1. Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
2. Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.

3. Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

The relation between various types of schedules can be depicted as:



Example: Consider the following schedule:
S:R1(A), W2(A), Commit2, W1(A), W3(A), Commit3, Commit1

Which of the following is true?
(A) The schedule is view serializable schedule and strict recoverable schedule
(B) The schedule is non-serializable schedule and strict recoverable schedule
(C) The schedule is non-serializable schedule and is not strict recoverable schedule.
(D) The Schedule is serializable schedule and is not strict recoverable schedule

Solution: The schedule can be re-written as:-

T_1	T_2	T_3
R(A)		
	W(A)	
	Commit	

T_1	T_2	T_3
W(A)		
		W(A)
		Commit
Commit		

First of all, it is a view serializable schedule as it has view equal serial schedule T_1 —> T_2 —> T_3 which satisfies the initial and updated reads and final write on variable A which is required for view serializability. Now we can see there is write – write pair done by transactions T_1 followed by T_3 which is violating the above-mentioned condition of strict schedules as T_3 is supposed to do write operation only after T_1 commits which is violated in the given schedule. Hence the given schedule is serializable but not strict recoverable.
So, option (D) is correct.

Concurrency Control in DBMS



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.

What is Transaction?

A transaction is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. However, during the execution of a transaction, it may be necessary temporarily to allow inconsistency, since either the debit of A or the credit of B must be done before the other. This temporary inconsistency, although necessary, may lead to difficulty if a failure occurs.

It is the programmer's responsibility to define properly the various transactions, so that each preserves the consistency of the database. For example, the transaction to transfer funds from the account of department A to the account of department B could be defined to be composed of two separate programs: one that debits account A, and another that credits account B. The execution of these two programs one after the other will indeed preserve consistency. However, each program by itself does not transform the database from a consistent state to a new consistent state. Thus, those programs are not transactions.

The concept of a transaction has been applied broadly in database systems and applications. While the initial use of transactions was in financial applications, the concept is now used in real-time applications in telecommunication, as well as in the management of long-duration activities such as product design or administrative workflows.

A set of logically related operations is known as a transaction. The main operations of a transaction are:

- **Read(A):** Read operations Read(A) or R(A) reads the value of A from the database and stores it in a buffer in the main memory.
- **Write (A):** Write operation Write(A) or W(A) writes the value back to the database from the buffer.

(Note: It doesn't always need to write it to a database back it just writes the changes to buffer this is the reason where dirty read comes into the picture)

Let us take a debit transaction from an account that consists of the following operations:

1. R(A);
2. A=A-1000;
3. W(A);

Assume A's value before starting the transaction is 5000.

- The first operation reads the value of A from the database and stores it in a buffer.
- the Second operation will decrease its value by 1000. So buffer will contain 4000.
- the Third operation will write the value from the buffer to the database. So A's final value will be 4000.

But it may also be possible that the transaction may fail after executing some of its operations. The failure can be because of **hardware, software or power,** etc. For example, if the debit transaction discussed above fails after executing operation 2, the value of A will remain 5000 in the database which is not acceptable by the bank. To avoid this, Database has two important operations:

- **Commit:** After all instructions of a transaction are successfully executed, the changes made by a transaction are made permanent in the database.
- **Rollback:** If a transaction is not able to execute all operations successfully, all the changes made by a transaction are undone.

For more details please refer Transaction Control in DBMS article.

Properties of a Transaction

Atomicity: As a transaction is a set of logically related operations, **either all of them should be executed or none**. A debit transaction discussed above should either execute all three operations or none. If the debit transaction fails after executing operations 1 and 2 then its new value of 4000 will not be updated in the database which leads to inconsistency.

Consistency: If operations of debit and credit transactions on the same account are executed concurrently, it may leave the database in an inconsistent state.

- For Example, with T1 (debit of Rs. 1000 from A) and T2 (credit of 500 to A) executing concurrently, the database reaches an inconsistent state.
- Let us assume the Account balance of A is Rs. 5000. T1 reads A(5000) and stores the value in its local buffer space. Then T2 reads A(5000) and also stores the value in its local buffer space.
- T1 performs A=A-1000 (5000-1000=4000) and 4000 is stored in T1 buffer space. Then T2 performs A=A+500 (5000+500=5500) and 5500 is stored in the T2 buffer space. T1 writes the value from its buffer back to the database.
- A's value is updated to 4000 in the database and then T2 writes the value from its buffer back to the database. A's value is updated to 5500 which shows that the effect of the debit transaction is lost and the database has become inconsistent.
- To maintain consistency of the database, we need **concurrency control protocols** which will be discussed in the next article. The operations of T1 and T2 with their buffers and database have been shown in Table 1.

T1	**T1's buffer space**	**T2**	**T2's Buffer Space**	**Database**
				A=5000
R(A);	A=5000			A=5000
	A=5000	R(A);	A=5000	A=5000
A=A-1000;	A=4000		A=5000	A=5000
	A=4000	A=A+500;	A=5500	
W(A);			A=5500	A=4000
		W(A);		A=5500

Isolation: The result of a transaction should not be visible to others before the transaction is committed. For example, let us assume that A's balance is Rs. 5000 and T1 debits Rs. 1000 from A. A's new balance will be 4000. If T2 credits Rs. 500 to A's new balance, A will become 4500, and after this T1 fails. Then we have to roll back T2 as well because it is using the value produced by T1. So transaction results are not made visible to other transactions before it commits.

Durable: Once the database has committed a transaction, the changes made by the transaction should be permanent. e.g.; If a person has credited $500000 to his account, the bank can't say that the update has been lost. To avoid this problem, multiple copies of the database are stored at different locations.

What is a Schedule?

A schedule is a series of operations from one or more transactions. A schedule can be of two types:

Serial Schedule: When one transaction completely executes before starting another transaction, the schedule is called a serial schedule. A serial schedule is always consistent. e.g.; If a schedule

S has debit transaction T1 and credit transaction T2, possible serial schedules are T1 followed by T2 (T1->T2) or T2 followed by T1 ((T2->T1). A serial schedule has low throughput and less resource utilization.

Concurrent Schedule: When operations of a transaction are interleaved with operations of other transactions of a schedule, the schedule is called a Concurrent schedule. e.g.; the Schedule of debit and credit transactions shown in Table 1 is concurrent. But concurrency can lead to inconsistency in the database. The above example of a concurrent schedule is also inconsistent.

Difference between Serial Schedule and Serializable Schedule

Serial Schedule	Serializable Schedule
In Serial schedule, transactions will be executed one after other.	In Serializable schedule transaction are executed concurrently.
Serial schedule are less efficient.	Serializable schedule are more efficient.
In serial schedule only one transaction executed at a time.	In Serializable schedule multiple transactions can be executed at a time.
Serial schedule takes more time for execution.	In Serializable schedule execution is fast.

Concurrency Control in DBMS

- Executing a single transaction at a time will increase the waiting time of the other transactions which may result in delay in the overall execution. Hence for increasing the overall throughput and efficiency of the system, several transactions are executed.
- 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 provides a procedure that is able to control concurrent execution of the operations in the database.
- The fundamental goal of database concurrency control is to ensure that concurrent execution of transactions does not result in a loss of database consistency. The concept of serializability can be used to achieve this goal, since all serializable schedules preserve consistency of the database. However, not all schedules that preserve consistency of the database are serializable.
- In general it is not possible to perform an automatic analysis of low-level operations by transactions and check their effect on database consistency constraints. However, there are simpler techniques. One is to use the database consistency constraints as the basis for a split of the database into subdatabases on which concurrency can be managed separately.
- Another is to treat some operations besides read and write as fundamental low-level operations and to extend concurrency control to deal with them.

Concurrency Control Problems

There are several problems that arise when numerous transactions are executed simultaneously in a random manner. The database transaction consist of two major operations “Read” and “Write”.

It is very important to manage these operations in the concurrent execution of the transactions in order to maintain the consistency of the data.

Dirty Read Problem(Write-Read conflict)

Dirty read problem occurs when one transaction updates an item but due to some unconditional events that transaction fails but before the transaction performs rollback, some other transaction reads the updated value. Thus creates an inconsistency in the database. Dirty read problem comes under the scenario of Write-Read conflict between the transactions in the database

1. The lost update problem can be illustrated with the below scenario between two transactions T1 and T2.
2. Transaction T1 modifies a database record without committing the changes.
3. T2 reads the uncommitted data changed by T1
4. T1 performs rollback
5. T2 has already read the uncommitted data of T1 which is no longer valid, thus creating inconsistency in the database.