# ISYS2120 – Data and Information Management

**Assessment**

| Assessment Details | Type | Weightage |
|---|---|---|
| Summaries | Individual | 5% |
| SQL tasks on Ed | Individual | 5% |
| Group Work | Individual | 5% |
| Assignment 1 | Group | 5% |
| Assignment 2 | Group | 5% |
| Assignment 3 | Group | 10% |
| Assignment 4 | Individual | 5% |
| Final Exam | Individual | 60% |

## Week 1

### Introduction

Data refers to raw facts, statistics, or information that are typically in a form that can be processed, stored, or transmitted by a computer. Data can take many forms, and it serves as the foundation for decision-making, analysis, and information in various contexts. A database is a structured and organized collection of data, typically stored electronically in a computer system. Data holds immense importance in various aspects of our personal, professional, and societal lives. Its significance is growing as we increasingly rely on technology and data-driven decision-making. Here are some key reasons why data is important:

- Informed Decision-Making:
  Data provides the basis for making informed decisions in various domains, from business strategies to public policy. It allows organizations and individuals to understand past performance, predict future trends, and optimize processes.
- Problem Solving:
  Data is crucial for identifying and solving problems. Whether it's diagnosing a medical condition, troubleshooting technical issues, or resolving complex challenges, data analysis plays a central role.
- Business Intelligence:
  Data is the lifeblood of business intelligence. It helps organizations gain insights into customer behaviour, market trends, and competition, enabling them to make decisions that improve efficiency and profitability.

Databases are designed to efficiently store, manage, and retrieve data, making them a fundamental component of modern information systems. They provide a systematic and reliable way to store large volumes of data, enabling data management, analysis, and retrieval. Databases play a critical role in modern information systems, and their importance can be understood from various perspectives:

- Data Centralization:
  Databases provide a centralized and structured repository for storing large volumes of data. This centralization ensures data consistency, eliminates redundancy, and allows data to be easily located and accessed.
- Data Integrity:
  Databases enforce rules and constraints to maintain data integrity. This includes validation rules, referential integrity, and data consistency checks. As a result, data quality is assured.
- Efficient Data Retrieval:

Databases are designed to efficiently retrieve data. They use indexing, caching, and query optimization techniques to reduce the time required to fetch specific information. This is crucial for applications that require rapid access to data.

- Data Security:

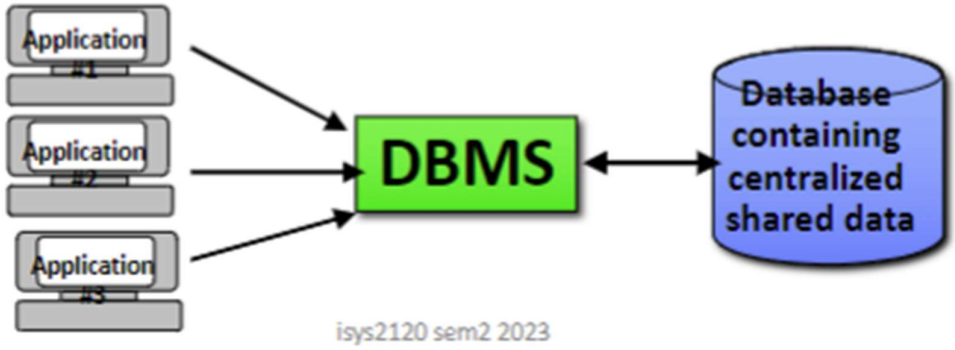
  Databases offer robust security features to control access to data. This includes user authentication, authorization, encryption, and auditing to protect sensitive information from unauthorized access or breaches.
- Scalability:

  Databases can scale to accommodate growing data volumes and increased workloads. This scalability is crucial for applications that need to handle expanding datasets or user bases.
- Transaction Management:

  Databases support transactions, which are sequences of database operations treated as a single unit. This ensures data consistency and reliability, especially in applications where financial or sensitive data is involved.
- Data Management:

  Databases simplify data management by providing tools for creating, modifying, and deleting data. They allow the implementation of data governance policies and data life cycle management.

A Database Management System (DBMS) is software that provides an interface for users and applications to interact with a database. Its primary purpose is to store, retrieve, manage, and manipulate data in a structured way efficiently and securely. A DBMS acts as an intermediary between the database itself and the users or applications that need to access the data. The usual way of storing and using data works as shown in the following diagram:




**Data Management**

Data management refers to the practice of planning, organizing, securing, and maintaining data throughout its lifecycle. It encompasses a wide range of activities and processes aimed at ensuring data is accurate, consistent, available, and secure. Effective data management is critical for organizations and individuals to make informed decisions, comply with regulations, and leverage data for various purposes. Typically, data management is divided with high-level goals (policy) set by the organisational leaders, and then settings processes, etc. (mechanisms) determined by the technical people to achieve the goals.

Data management involves various concerns and challenges that organizations and individuals must address to ensure the effectiveness, security, and compliance of their data-related activities. Here are some of the key concerns in data management:

- Data Security:
  Protecting data from unauthorized access, breaches, and cyber threats is a paramount concern. Organizations must implement robust security measures, including encryption, access controls, and security audits, to safeguard sensitive data.
- Data Privacy:
  Data privacy regulations, such as GDPR (General Data Protection Regulation) and CCPA (California Consumer Privacy Act), impose strict requirements on how personal data is collected, processed, and stored. Organizations must ensure compliance and respect individual privacy rights.
- Data Quality:
  Maintaining data quality is crucial to ensure that data is accurate, complete, consistent, and free from errors. Poor data quality can lead to incorrect decisions and operational issues.
- Data Governance:
  Effective data governance involves defining data ownership, responsibilities, policies, and processes to ensure data is managed consistently, following organizational standards.
- Data Lifecycle Management:
  Managing data throughout its lifecycle, from creation to archiving or deletion, is essential. This includes data retention policies, backup, archiving, and disposal processes.
- Data Integration:
  Integrating data from diverse sources and systems requires addressing issues related to data transformation, consistency, and compatibility. This ensures that data can be used effectively across the organization.
- Data Backup and Recovery:
  Establishing and maintaining robust backup and recovery procedures are vital for data resilience and continuity in the event of data loss or corruption.
- Data Cataloguing and Metadata:
  Metadata management concerns the documentation, indexing, and cataloguing of data assets, allowing users to discover, understand, and access data.
- Data Access and Retrieval:
  Providing secure, efficient, and controlled mechanisms for users and applications to access and retrieve data is a central concern to ensure data availability and usability.
- Data Migration:
  Data migration concerns the process of moving or transferring data between systems. Successful data migration requires planning, testing, and validation to ensure data consistency and accuracy.
- Data Ethics:
  Ethical concerns in data management relate to the responsible use of data, including protecting individuals' privacy, avoiding data bias, and ensuring that data usage aligns with ethical guidelines.

**Structure of Data**

A key idea in DBMSs is for the database itself to store descriptions of the format of the data. This is called the “System Catalogue” or “Data Dictionary”. This is what you call the meta-data.

**Relational DBMS**

A Relational Database Management System (RDBMS) is a type of database management system that stores and manages data in a structured and organized manner based on the principles of the relational model. In this, all the data is seen by users as tables of related, uniformly structured information. A few important terminologies associated with this are given below:

- Instance

  An instance is the contents of the database at a single time. It usually consists of specific values, which describe the specific situation in the world. Every update changes the instances as well.
- Schema

  The schema describes the structure of data in a particular database. It defines what tables exist and what each column are called and what is the data type of each of these columns, and so on. The instance that is created always need to fit the pattern of the schema. The schema also consists of integrity constraints which restricts the possible instances.

**Data Languages**

There are multiple different languages used in SQL. Two of the main languages related to SQL, these are:

- DDL

  DDL stands for "Data Definition Language," which is a subset of SQL (Structured Query Language) used for defining and managing the structure of a database and its objects. DDL statements are used to create, modify, and delete database objects such as tables, indexes, views, and schemas. DDL statements are essential for setting up and maintaining the database schema. DDL statements are typically executed by database administrators or users with the necessary permissions to define and manage the database schema. They have a significant impact on the structure of the database and require careful consideration to avoid unintended data loss or structural changes. Some common DDL statements include:
  - CREATE: Used to create new database objects. For example:
    - CREATE TABLE: Creates a new table.
    - CREATE INDEX: Creates an index on a table.
  - ALTER: Used to modify the structure of an existing database object. For example:
    - ALTER TABLE: Modifies an existing table, such as adding or dropping columns.
    - ALTER INDEX: Modifies an existing index.
  - DROP: Used to delete database objects. For example:
    - DROP TABLE: Deletes a table.
    - DROP INDEX: Deletes an index.
  - TRUNCATE: Removes all rows from a table but retains the table's structure.
- DML

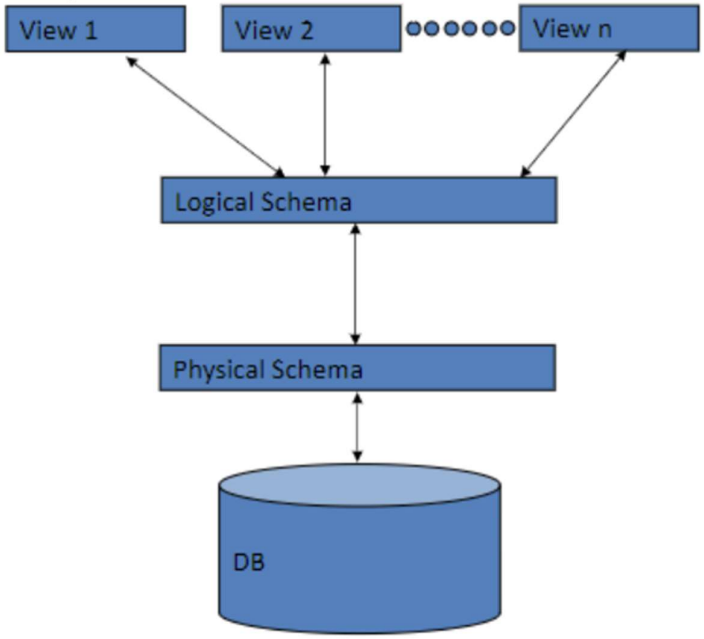
  DML stands for "Data Manipulation Language," which is a subset of SQL (Structured Query Language) used for managing and manipulating data stored in a relational database management system (RDBMS). DML statements are responsible for performing operations on the data itself, such as inserting, updating, retrieving, and deleting records within database tables. These statements are fundamental for interacting with the database's data. DML statements are typically used by applications, users, or database administrators to interact with the database's data. They allow for

data retrieval, data entry, data updating, and data removal, making them integral to a wide range of database operations. It's important to use DML statements carefully, ensuring that data remains consistent and accurate while maintaining data integrity. Some common DML statements include:

- SELECT: Used to retrieve data from one or more tables. You can specify the columns to retrieve and apply conditions to filter the data.
- INSERT: Adds new rows (records) into a table. You provide the values to be inserted into specific columns.
- UPDATE: Modifies existing data in one or more rows of a table. You can change values in one or more columns based on specified conditions.
- DELETE: Removes rows from a table based on specified conditions, effectively deleting data from the database.

**Abstraction in SQL**

In SQL (Structured Query Language), abstraction refers to the concept of hiding the complex underlying details of the database system and allowing users to interact with the data and database objects at a higher, more user-friendly level. Abstraction in SQL is achieved through various layers and components, making it easier for users to work with databases without needing to know the intricacies of the database's internal workings. There are different levels of abstraction as given below:




Here the view describes how a user will see the data, logical schema defines the structure of the data as it is shared among all users, and physical schema describes the files and indexes used for storage on disk.

**Data Independence**

Data independence is a concept in database management systems (DBMS) that refers to the separation of the way data is stored, organized, and accessed from the applications or users that interact with the data. It allows changes to be made to the database's physical structure (physical data independence) or its logical structure (logical data independence)

without affecting the applications or queries that use the data. There are two main types of data independence:

- Physical Data Independence: This type of data independence allows changes to the physical storage structure of the database without affecting the way data is accessed or used by applications. In other words, changes to how data is stored, such as moving to a different storage device, adding, or removing indexes, or reorganizing data, should not impact the application programs. This separation between the physical storage details and the logical representation of data is important for database performance optimization and maintenance.
- Logical Data Independence: Logical data independence enables changes to the logical structure of the database without affecting the application programs or queries. The logical structure includes the database schema, tables, relationships, and constraints. With logical data independence, you can modify the structure, add or remove tables, change data types, or adjust relationships between tables without requiring changes to the application code or queries that use the database.

**Roles with DBMS**

- End Users

  End users are individuals or groups who interact with the database to retrieve, update, and use data for their specific tasks and requirements. Their main responsibilities include:
  - Data Retrieval: End users use SQL queries, forms, reports, and applications to access and retrieve data from the database.
  - Data Modification: Some end users may have the privilege to add, modify, or delete data within the scope of their responsibilities.
  - Data Analysis: End users, particularly data analysts, rely on the DBMS to perform data analysis, generate reports, and extract insights from the data.
  - Data Entry: In systems where data entry is required, end users may input data into the database using forms or applications.
- DB Application Programmers

  Database application programmers are responsible for developing, maintaining, and optimizing applications that interact with the database. Their main responsibilities include:
  - Application Development: They design and develop software applications that communicate with the DBMS, including web applications, desktop software, or mobile apps.
  - Query Writing: Programmers write SQL queries and optimize them for efficient data retrieval and modification.
  - Data Validation: They implement data validation rules and business logic within the applications to ensure data quality.
  - Security: Programmers work on implementing security measures, including user authentication and authorization, to protect the database.
  - Performance Optimization: They tune the applications and database queries for performance, making sure they run efficiently.
- Database Administrators

  Database administrators are responsible for the overall management and maintenance of the database system. Their main responsibilities include:
  - Database Design: DBAs design and create the database schema, tables, relationships, and constraints.

- Data Security: They implement security measures, manage user access, and ensure data integrity through constraints, encryption, and auditing.
- Performance Optimization: DBAs monitor the database's performance, fine-tuning indexes, optimizing queries, and allocating resources to maintain efficient operation.
- Backup and Recovery: They plan and execute backup and recovery strategies to safeguard data against loss or corruption.
- Database Maintenance: Regular maintenance tasks, such as data purging, table reorganization, and software updates, are handled by DBAs.

- DBMS Vendor Staff

  DBMS vendor staff members work for the company or organization that develops and sells the DBMS software. Their main responsibilities include:
  - Software Development: They are responsible for designing, developing, testing, and maintaining the DBMS software.
  - Customer Support: Vendor staff provides technical support to organizations and users who have purchased their DBMS software.
  - Updates and Enhancements: They release updates and new versions of the DBMS to address issues, add features, and improve performance.
  - Documentation: Vendor staff creates user manuals, documentation, and knowledge resources to help users understand and utilize the DBMS effectively.
  - Training: Some DBMS vendors offer training and certification programs to help users and administrators become proficient in using their software.

**File Management System vs Database Management System**

Following is a comparative analysis of File Management Systema and DBMS:

| File Management System | Database Management System |
|---|---|
| • Data Organization:<br>  ○ Data is organized into files and folders.<br>  ○ Each file can have a specific format or structure.<br>• Data Retrieval:<br>  ○ Data retrieval relies on file paths and directories.<br>  ○ Querying data typically involves custom application code.<br>• Data Redundancy:<br>  ○ Data redundancy is common, leading to multiple copies of the same data.<br>  ○ Inconsistencies may arise due to the independence of applications.<br>• Data Integrity:<br>  ○ Data integrity relies on practices and processes implemented by individual applications.<br>  ○ There may be no mechanisms for enforcing data integrity across applications.<br>• Scalability: | • Data Organization:<br>  ○ Data is organized into structured tables, rows, and columns, following a predefined schema.<br>  ○ Constraints and relationships enforce data consistency and structure.<br>• Data Retrieval:<br>  ○ Data retrieval is simplified through SQL queries, enabling efficient data filtering, joining, and aggregation.<br>  ○ DBMSs optimize query performance with indexing and query optimization.<br>• Data Redundancy:<br>  ○ Data redundancy is minimized through normalization techniques, reducing storage, and ensuring data consistency.<br>• Data Integrity:<br>  ○ Data integrity is maintained through constraints (e.g., primary keys, foreign keys) enforced by the DBMS. |

| | |
|---|---|
| ○ Scalability can be limited, as adding new applications or data formats may require significant effort.<br>• Data Security:<br>○ Data security is typically implemented on a per-application basis.<br>○ Access controls may be limited.<br>• Data Backup and Recovery:<br>○ Backup and recovery processes are often manual and can be error prone.<br>• Concurrency and Transactions:<br>○ Handling concurrent access and ensuring data consistency can be challenging in the absence of built-in transaction management.<br>• Data Analysis:<br>○ Analysing data across different applications can be difficult and often requires complex data extraction and transformation. | ○ ACID properties ensure data consistency, even in the presence of concurrent access.<br>• Scalability:<br>○ DBMSs are designed to scale, making it easier to add new data, tables, or applications without major changes.<br>• Data Security:<br>○ DBMSs provide robust access controls, authentication, and authorization mechanisms to enforce data security.<br>• Data Backup and Recovery:<br>○ DBMSs offer built-in mechanisms for data backup, restore, and point-in-time recovery.<br>• Concurrency and Transactions:<br>○ DBMSs provide transaction management to handle concurrent access and maintain data consistency.<br>• Data Analysis:<br>○ DBMSs support efficient data analysis through SQL and often integrate with data analysis tools. |

## Week 2

**Relation Schema**

A relation schema, in the context of relational databases, is a blueprint or structure that defines the organization of data within a database table. It specifies the attributes or columns that a table will have, along with their data types and any constraints on the data. A relation schema is a fundamental part of the relational database model, which is widely used for organizing and managing structured data. Key elements of a relation schema include:

- Table Name:
  The name of the table, which is used to reference it within the database.
- Attributes/Columns:
  These are the fields or properties that define the type of data that can be stored in the table. Each attribute has a name and a data type (e.g., integer, text, date) that specifies the kind of values it can hold.
- Constraints:
  Constraints define rules and restrictions on the data that can be stored in the table. Common constraints include primary keys, foreign keys, unique constraints, check constraints, and not-null constraints. These constraints help maintain data integrity and enforce business rules.
- Primary Key:
  This is a unique identifier for each row in the table, ensuring that each row can be uniquely identified.
- Foreign Key:
  When a table has a relationship with another table, a foreign key is used to establish that relationship. It typically references the primary key of another table.

- Unique Constraints:
  These ensure that values in a specific column are unique, preventing duplicates.
- Check Constraints:
  They define conditions that data in a column must meet.
- Not-Null Constraints:
  These ensure that a column cannot have null (missing) values.

A relation schema is a crucial aspect of database design, as it provides a structured way to organize and represent data. It helps ensure data consistency and integrity within the database by defining the rules that data must adhere to. It also serves as a reference for developers and users to understand the structure and content of the database tables.

**Constraints**

Constraints are rules or conditions that are applied to the data stored in database tables to ensure data integrity and maintain the consistency and accuracy of the data. Constraints help enforce business rules and prevent the insertion of invalid or inconsistent data into the database. There are several types of constraints commonly used in relational databases:

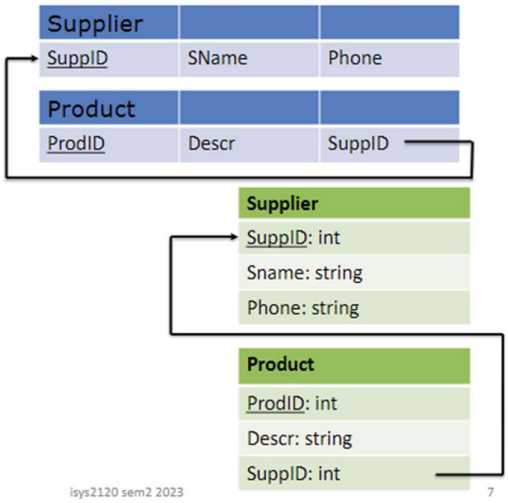
- Primary Key Constraint:
  A primary key is a column or a set of columns in a table that uniquely identifies each row in the table. It ensures that no two rows have the same values in the primary key column(s). Primary keys are used for data retrieval and for establishing relationships between tables through foreign keys.
- Foreign Key Constraint:
  A foreign key is a column or a set of columns in one table that references the primary key in another table. It enforces referential integrity by ensuring that the values in the foreign key column(s) exist in the referenced table's primary key column(s). This constraint is used to establish relationships between tables.
- Unique Constraint:
  A unique constraint ensures that the values in a specified column (or columns) are unique across all rows in the table. It allows for the presence of null values but enforces uniqueness among non-null values.
- Check Constraint:
  A check constraint defines a condition or expression that must be satisfied for data in a specific column. It is used to ensure that data meets certain criteria, such as range checks, format checks, or custom business rules.
- Not-Null Constraint:
  A not-null constraint ensures that a specific column does not contain null values. It enforces the presence of a value in that column.
- Default Constraint:
  A default constraint specifies a default value for a column. If no value is provided for that column during an insert operation, the default value is used instead.
- Unique Key Constraint:
  Like a unique constraint, a unique key constraint enforces uniqueness in a column or a set of columns, but it also allows for one null value. This means that one row can have a null value in the unique key column, and all other rows must have unique values.

Constraints are essential for maintaining data quality and consistency within a database. They help prevent data anomalies, such as duplicates, data that violates referential integrity, or

data that does not conform to defined rules. By enforcing constraints, database management systems ensure that the data remains accurate and reliable, which is critical for the integrity of the information stored in the database.

**Relation Schema Diagram**

A relational schema diagram, often referred to as an Entity-Relationship Diagram (ERD) or database schema diagram, is a visual representation of the structure and relationships of tables (relations) within a relational database. These diagrams provide a graphical way to depict the database schema, showing how tables are organized, the attributes they contain, and the relationships between them. ERDs are an essential tool in database design and documentation, helping database designers, developers, and stakeholders understand and communicate the database's structure and logic. An example of it is given below:




Key components of a relational schema diagram include:

- Entities (Tables/Relations):

  These are represented as rectangles or squares in the diagram. Each entity corresponds to a table in the database. The name of the table is written within the entity shape.
- Attributes (Columns/Fields):

  Attributes are depicted as ovals or ellipses and are connected to their respective entities. They represent the columns or fields within the table and include the attribute name and its data type. For example, for an "Employees" entity, attributes might include "EmployeeID," "FirstName," "LastName," etc.
- Primary Key:

  The primary key attribute(s) in an entity is typically underlined or marked in some way to indicate that it uniquely identifies each row in the table.
- Relationships:

  Relationships between entities are shown as lines connecting them. These lines have labels that describe the type of relationship, such as "one-to-many" or "many-to-many." The cardinality of the relationship (e.g., 1:1, 1:N, N:M) is often indicated at the ends of the relationship lines.
- Foreign Keys:

  When a table has a foreign key relationship with another table, the foreign key attribute is typically connected to the primary key attribute it references in the other table. This helps illustrate the referential integrity of the database.

- Cardinality:
  Symbols or notation at the ends of relationship lines indicate the cardinality of the relationship, specifying how many entities can be related to each other. Common notations include a crow's foot (for "many") and a straight line (for "one").
- Attributes and Constraints:
  Other notations or symbols can be added to represent constraints, such as check constraints or unique constraints, if necessary.

Relational schema diagrams are invaluable during the database design and development process. They provide a clear, visual representation of the database structure, making it easier to discuss and refine the design with stakeholders. They also aid in identifying potential issues, such as redundancy or inconsistency in data storage, and ensure that the database design adheres to best practices. Moreover, these diagrams serve as documentation that can be used for future reference and maintenance of the database.

**Keys**

Keys are attributes or combinations of attributes that play a crucial role in organizing and identifying data within tables. Keys are fundamental to maintaining data integrity, establishing relationships between tables, and ensuring the uniqueness of data. Several types of keys are commonly used in relational databases:

- Primary Key (PK):
  A primary key is a unique identifier for each row in a table. It ensures that no two rows can have the same values in the primary key column(s). The primary key is used to identify each row and is often the basis for relationships with other tables uniquely and unambiguously. A table can have only one primary key, and its values cannot be null.
- Foreign Key (FK):
  A foreign key is an attribute or set of attributes in one table that references the primary key of another table. It establishes relationships between tables, enforcing referential integrity. Foreign keys help maintain consistency in data by ensuring that values in the referencing table's foreign key column(s) correspond to values in the referenced table's primary key column(s).
- Super Key:
  A super key is a set of one or more attributes that can be used to uniquely identify rows in a table. While a super key is not necessarily minimal (it may contain more attributes than needed for uniqueness), it is a broader concept that includes the primary key.
- Candidate Key:
  A candidate key is a minimal super key, meaning it is a set of attributes that uniquely identifies rows, and removing any attribute from it would result in the loss of uniqueness. From the candidate keys, one is chosen to be the primary key.
- Composite Key:
  A composite key is a key that consists of multiple attributes rather than a single attribute. It is used when a single attribute is insufficient to ensure uniqueness.
- Composite Primary Key:
  A composite primary key, also known as a compound primary key, is a type of primary key in a relational database that consists of two or more attributes (columns) used together to uniquely identify each row in a table. Unlike a single-column primary key, which is composed of a single attribute, a composite primary key is made up of a combination of attributes. Each combination of values in the composite key must be

unique within the table. This means that no two rows can have the same combination of values in all the columns that make up the composite primary key. Composite primary keys are used in situations where a single attribute cannot effectively ensure the uniqueness of each row, but a combination of attributes can.

Keys are essential for maintaining data quality, enabling efficient data retrieval, and establishing relationships between tables in a relational database. They help ensure data integrity, prevent duplication, and maintain the consistency of data, making databases effective and reliable for storing and managing structured information.

**Database Designing**

Database design is the process of creating a structured plan for how data will be stored, organized, and accessed within a relational database management system (RDBMS). Effective database design is crucial for ensuring data integrity, efficiency, and the ability to meet the information needs of an organization. Here are the key steps and considerations in the database design process:

- **Requirements Analysis:**
  Begin by understanding the requirements of the database. This involves talking to stakeholders, end-users, and subject matter experts to identify data needs, relationships, and business rules. Determine the types of data to be stored, how it will be accessed, and what operations will be performed on it.
- **Conceptual Design:**
  Create a high-level, abstract representation of the database, often in the form of an Entity-Relationship Diagram (ERD). This conceptual design defines entities, their attributes, and the relationships between them.
- Normalization:
  Normalize the data model to eliminate data redundancy and improve data integrity. This involves breaking down data into smaller, related tables and ensuring that each piece of data is stored in only one place.
- **Schema (Logical) Design:**
  Translate the conceptual design into a logical schema. This step defines tables, columns, data types, and relationships between tables. Determine the primary keys for each table and consider the use of foreign keys to establish relationships between tables.
- Data Integrity and Constraints:
  Apply constraints like primary keys, unique constraints, foreign keys, check constraints, and not-null constraints to ensure data integrity and enforce business rules.
- Indexing:
  Identify the fields that will be frequently used for searching and sorting data and create indexes on those fields to optimize query performance.
- Data Types:
  Choose appropriate data types for each column to ensure efficient storage and retrieval. Common data types include integers, text, dates, and more specialized types like geographic data or binary data.
- Security:
  Plan for data security and access control. Define user roles and permissions to restrict data access to authorized users and protect sensitive information.
- Normalization (Again):
  Review and refine the normalization of the schema, considering factors like query patterns, data volume, and performance requirements.

- **Physical Design:**
  Determine the physical aspects of the database, including storage, file locations, and indexing strategies. Consider factors like data distribution, storage capacity, and backups.
- Data Loading:
  Develop processes to import or input data into the database. This may involve data migration, data conversion, or data entry.
- Testing:
  Thoroughly test the database to ensure that it performs as expected, data integrity is maintained, and all constraints are enforced correctly.
- Documentation:
  Create documentation that describes the database schema, constraints, relationships, and other relevant details. This documentation is essential for future maintenance and understanding the database structure.
- Optimization and Performance Tuning:
  Continuously monitor and optimize the database for performance by analysing query execution plans, indexing, and query optimization.
- Backup and Recovery Planning:
  Implement backup and recovery strategies to protect data in case of system failures, data corruption, or accidental deletions.
- Scalability:
  Consider future scalability requirements and plan for data growth by optimizing the database design to handle increasing data volumes.

Database design is an iterative process that often involves collaboration between database designers, developers, and end-users to ensure that the database meets the needs of the organization and performs efficiently. It's a critical step in database development and maintenance, as a well-designed database is essential for accurate data storage, retrieval, and analysis.

**Conceptual Data Model**

A conceptual data model is an abstract representation of the essential data entities, relationships between them, and the attributes associated with the entities. It provides a high-level view of the data that an organization or system deals with, without delving into the technical implementation details. Conceptual data models are typically created during the early stages of the database design process to help stakeholders and database designers understand the data requirements and relationships in a clear and comprehensible manner. Key characteristics of a conceptual data model include:

- Entities:
  These are the major data objects or concepts of interest in the domain being modelled. Entities represent things like customers, products, employees, orders, or any other relevant objects. Entities are depicted as rectangles in diagrams.
- Attributes:
  Attributes are the properties or characteristics of entities. They describe the data that needs to be stored for each entity. For example, attributes of a "Customer" entity might include "CustomerID," "FirstName," "LastName," and "Email." Attributes are represented as ovals or ellipses in diagrams and are associated with their respective entities.

- Relationships:
  Relationships show how entities are related or connected to each other. They describe the associations and interactions between different entities. Relationships can be one-to-one, one-to-many, or many-to-many, and they are represented as lines connecting entities. The cardinality of the relationships is often indicated to specify the number of entities involved.
- Business Rules:
  Conceptual data models often capture high-level business rules that govern how data should be structured and used in the organization. These rules help ensure data integrity and consistency.

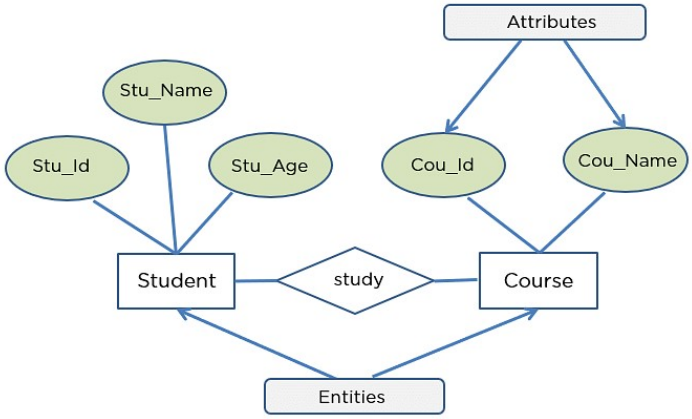
Conceptual data models are generally independent of any specific database management system or technical implementation. They serve as a communication tool between database designers and non-technical stakeholders, such as business analysts and domain experts. The primary purpose of a conceptual data model is to provide a common understanding of the data requirements and relationships within an organization, helping to align business needs with the eventual database design. There are different types of Conceptual Models as well such as:

- Concept Maps:
  Concept maps are graphical representations of concepts and their relationships. They are used in education to help students organize and understand knowledge, in problem-solving, and in brainstorming.
- Mind Maps:
  Mind maps are visual representations of ideas, concepts, or tasks organized around a central topic or idea. They are often used for brainstorming, note-taking, and project planning.
- Entity-Relationship Diagrams (ERD):
  In database design, ERDs are used to model the data structure of a database. They depict entities (tables), attributes (columns), and the relationships between entities. The conceptual ERD provides a high-level view of data requirements.
- UML Diagrams:
  The Unified Modelling Language (UML) includes various types of diagrams, such as class diagrams, use case diagrams, and activity diagrams, which are used for modelling software systems and business processes. These diagrams help visualize the structure and behaviour of a system.
- Flowcharts:
  Flowcharts use symbols and shapes to represent processes, decision points, and the flow of data or materials. They are widely used for process modelling, algorithm design, and documenting workflows.
- Data Flow Diagrams (DFD):
  DFDs are used to model the flow of data within a system or process. They represent data sources, processes, data storage, and data destinations. DFDs are commonly used in systems analysis and design.

Once the conceptual data model is created and agreed upon, it can serve as a basis for creating a logical data model, which involves defining the structure of the database more precisely, specifying data types, keys, and constraints. The logical data model, in turn, serves as a foundation for creating the physical database schema and implementing the database using a specific database management system.

**Entity Relationship Diagram**

An Entity-Relationship Diagram (ERD) is a visual representation that helps in designing and understanding the structure and relationships of data in a database. ERDs are a fundamental tool in database design, and they use a set of standardized symbols and notations to depict entities, their attributes, and the relationships between them. ERDs are essential for communicating the design of a relational database to stakeholders, including developers, business analysts, and database administrators. An example of it as follows:




Key components of an ERD include:

- Entities:

  Entities are the main objects or concepts about which data is stored in a database. In an ERD, entities are represented as rectangles, and each entity corresponds to a table in the database. For example, in a database for a library system, you might have entities like "Book," "Author," "Borrower," and "Library Branch." There are several types of entities that can be categorized based on their roles and characteristics within the database design:

  - Strong Entity:

    A strong entity is an entity that can exist independently and has its own unique identifier or primary key. It does not depend on another entity for identification. For example, in a university database, the "Student" entity is a strong entity because it can exist on its own and is uniquely identified by a StudentID.

  - Weak Entity:

    A weak entity is an entity that cannot exist independently and relies on a related strong entity for identification. It typically lacks a unique identifier of its own and is identified by a combination of attributes, including a partial key and a total key. For instance, in a database for tracking rooms and their reservations, the "Reservation" entity may be considered weak because it is identified by a combination of ReservationID (partial key) and RoomID (total key).

  - Composite Entity:

    A composite entity is an entity that is created to represent a complex relationship between other entities. It can be used to simplify and clarify the modelling of certain relationships. In a library database, a "Borrower-Book" entity may be created to handle the complex relationship between borrowers and books, which includes due dates, return status, and more.

- Attributes:

  Attributes are characteristics or properties of entities. They are depicted as ovals and are connected to the respective entity. For instance, the "Book" entity might have attributes like "Title," "ISBN," and "Publication Year." There are different types of attributes that can be used to represent various aspects of the data in a database. Here are the most common types of attributes:

  - Simple Attribute:

    A simple attribute is an attribute that cannot be divided into smaller sub-parts with meaningful values. For example, a "Name" attribute is typically a simple attribute because it is not composed of multiple parts.
  - Composite Attribute:

    A composite attribute is an attribute that can be divided into smaller sub-parts, each with its own meaning. For instance, an "Address" attribute can be divided into sub-parts like "Street," "City," "State," and "Postal Code."
  - Derived Attribute:

    A derived attribute is one whose value can be calculated from other attributes in the database. For example, the "Age" attribute can be derived from the "Date of Birth" attribute.
  - Multi-valued Attribute:

    A multi-valued attribute is an attribute that can hold multiple values for a single entity. For example, a "Phone Numbers" attribute for a contact entity may contain multiple phone numbers.
  - Key Attribute:

    A key attribute is used to uniquely identify entities within a database table. In the context of an entity, it is also known as a primary key attribute. A "StudentID" attribute in a "Student" entity could be a key attribute.
  - Single-valued Attribute:

    A single-valued attribute is an attribute that holds only one value for an entity. Most attributes are single valued by default, as they represent a single characteristic of an entity.
  - Multi-valued Composite Attribute:

    A multi-valued composite attribute is a combination of a multi-valued attribute and a composite attribute. It can hold multiple sets of composite values. For example, a "Languages Spoken" attribute may have multiple sets of language, fluency level pairs.
  - Null Attribute:

    A null attribute is an attribute that may have no value, which represents the absence of data. Null attributes are common when dealing with optional data or missing information.
- Relationships:

  Relationships show how entities are connected or associated with each other. They are represented as lines connecting entities. A relationship line may have a diamond shape on the connecting line, which indicates the type of relationship. Common types of relationships include one-to-one, one-to-many, and many-to-many. There are three main types of relationships:

  - One-to-One (1:1) Relationship:

    In a one-to-one relationship, one record in the first entity (table) is related to only one record in the second entity, and vice versa. This relationship is less common in database design but may be used to separate optional or infrequently used attributes into a separate table, or to enforce unique

constraints. For example, a "Person" entity may have a one-to-one relationship with an "Address" entity, where each person has one unique address.

- One-to-Many (1:N) Relationship:

  In a one-to-many relationship, one record in the first entity is related to one or more records in the second entity, but each record in the second entity is related to only one record in the first entity. This is the most common type of relationship and is used to represent hierarchical and parent-child relationships. For example, in a "Department" entity with a one-to-many relationship with an "Employee" entity, each department can have multiple employees, but each employee belongs to one department.

- Many-to-Many (N:M) Relationship:

  In a many-to-many relationship, multiple records in the first entity can be related to multiple records in the second entity, and vice versa. To represent a many-to-many relationship in a relational database, an associative entity (also known as a junction table or linking table) is introduced. This associative entity establishes two one-to-many relationships with the other two entities. For example, in a "Student" entity with a many-to-many relationship with a "Course" entity, an "Enrolment" entity is introduced to represent which students are enrolled in which courses.

In the context of Entity-Relationship Diagrams (ERDs) and database design, relationship attributes and roles are additional concepts used to provide more information about the relationships between entities. These concepts help clarify the nature of relationships and add details to the model.

- Relationship Attributes:

  Relationship attributes are attributes that are associated with a specific relationship between two entities. They are used to provide additional information about the relationship itself. Relationship attributes are typically represented as diamonds connected to the relationship line in an ERD. These attributes describe the relationship and can be thought of as properties of the relationship. For example, consider a database for a library. In the relationship between the "Book" entity and the "Borrower" entity, a "Due Date" attribute could be associated with the relationship, indicating when the book is due to be returned.

- Roles:

  Roles define the specific participation of an entity in a relationship. Each entity in a relationship may have a role that describes its function within that relationship. Roles are often included in the entity-relationship diagram to clarify the semantics of the relationship. For example, in a database for a hospital, consider the relationship between the "Doctor" entity and the "Patient" entity. The "Doctor" entity may have a "Primary Physician" role in the relationship, and the "Patient" entity may have a "Patient" role in the same relationship.

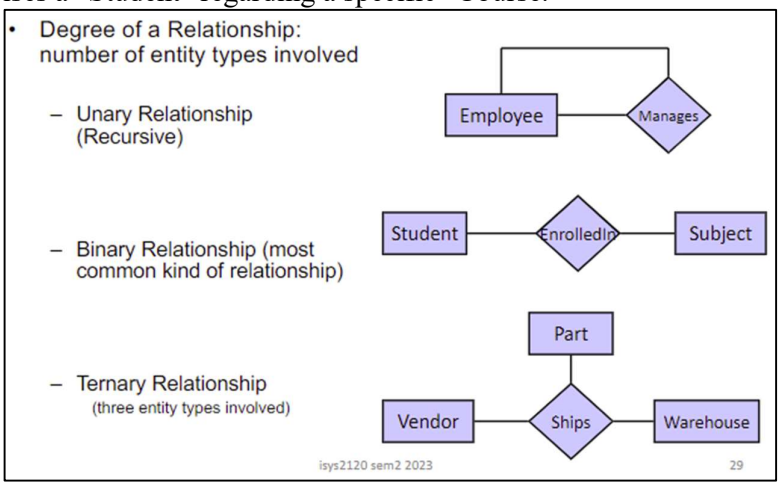
In the context of Entity-Relationship Diagrams (ERDs) and database design, "degrees in relationship" refers to the number of entities involved in a relationship and the degree to which they are connected. There are three main degrees in relationships:

- Unary Relationship (Degree 1):

  In a unary relationship, a single entity is related to itself. It is a self-referential relationship. Unary relationships are used when an entity has a relationship with other instances of the same entity. For example, in a database for an organization's hierarchy, you might have a unary relationship where an "Employee" entity relates to another "Employee" entity as a "Supervisor."
- Binary Relationship (Degree 2):

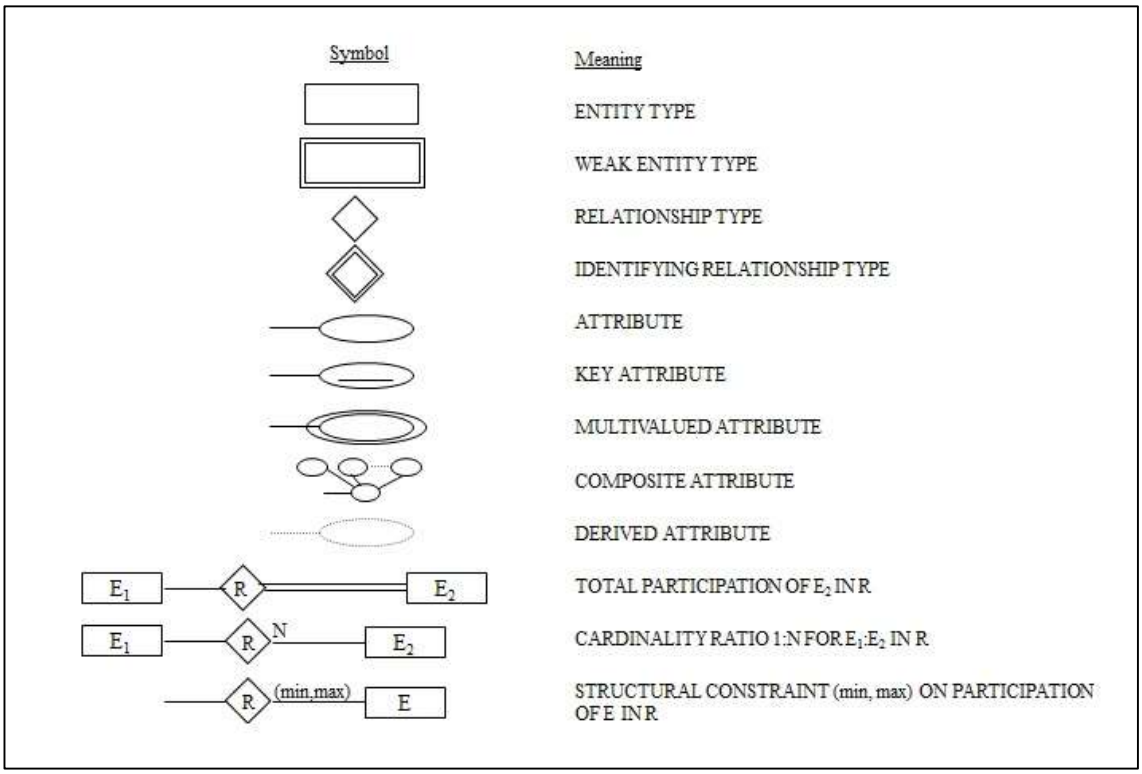
  In a binary relationship, two different entities are related to each other. It's the most common type of relationship, and it's often represented as a line connecting two entities. Binary relationships can be further categorized based on their cardinality:
  - One-to-One (1:1): Each entity in the relationship is related to only one entity in the other entity.
  - One-to-Many (1:N): Each entity in one side of the relationship is related to one or more entities in the other entity.
  - Many-to-Many (N:M): Multiple entities in one side of the relationship are related to multiple entities in the other entity. This type of relationship is implemented using an associative (junction) table.
- Ternary Relationship (Degree 3):

  In a ternary relationship, three different entities are related to each other. It's less common than binary relationships and adds complexity to the model. Ternary relationships are used when there is a complex interaction or association between three entities, and each entity's relationship with the others depends on the presence of the third entity. For example, in a database for a university, a ternary relationship might describe the "Advisership" where a "Professor" advises a "Student" regarding a specific "Course."




- Cardinality:

  Cardinality is used to define the quantity of entities participating in a relationship. It specifies how many instances of one entity are related to how many instances of another entity. Common notations include "1-1" (one-to-one), "1-M" (one-to-many), "M-N" (Many-to-Many), etc.




**Enhanced ER Model**

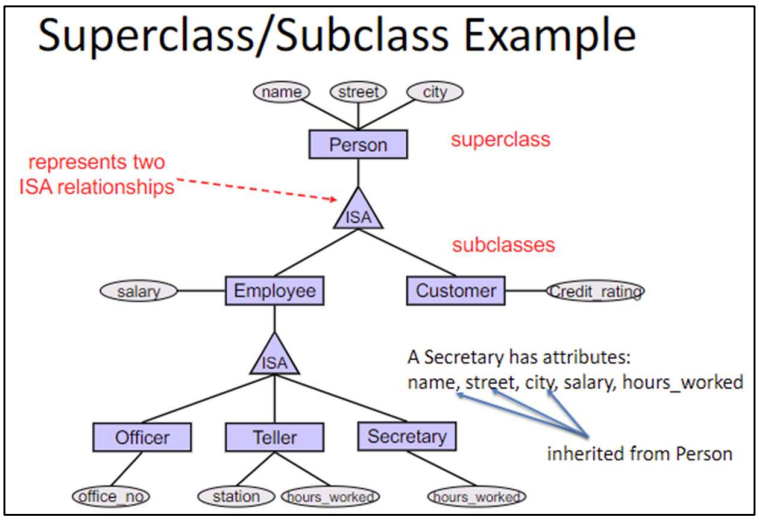
An Enhanced Entity-Relationship (ER) model, also known as Extended Entity-Relationship model (EER), is an extension of the traditional Entity-Relationship model used in database design. The Enhanced ER model includes additional constructs and features that allow for a more expressive and comprehensive representation of complex data relationships and business rules. ER Models in its original form did not support Specialization/Generalization and abstractions. This was also another reason for the development of Enhanced ER Model. The additional characteristics are:

- Generalization:

  Generalization is the process of abstracting common characteristics of multiple entities and creating a more generalized entity that represents these common attributes and relationships. The more generalized entity is referred to as a "supertype" or "parent entity." Generalization is used to capture the "is-a" relationship, where subtypes are viewed as specialized versions of a more general entity. The general entity captures attributes and relationships that are common to all subtypes. Generalization results in a hierarchical structure where the supertype is at a higher level of abstraction, and subtypes are at a lower level, inheriting the attributes and relationships from the supertype.

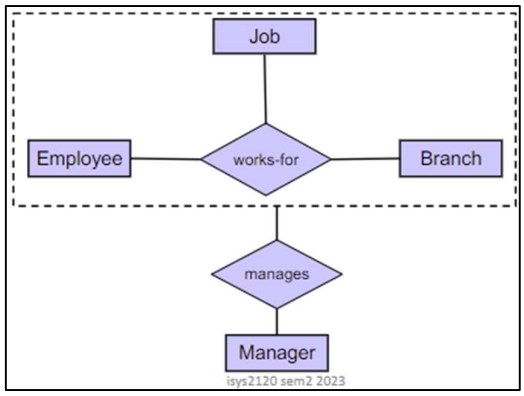
- Specialization:

  Specialization is the process of taking a more generalized entity (supertype) and creating one or more specialized entities (subtypes) that inherit the attributes and relationships of the supertype. Each subtype can have additional unique attributes and relationships. Specialization allows you to capture the differences or unique characteristics of individual subtypes while retaining the common features shared with the supertype. It is used to represent the "is-a-kind-of" relationship, where each subtype is a specific kind or type of the supertype.




- Aggregation

  Aggregation, in the context of database design and the Enhanced Entity-Relationship (EER) model, is a concept used to represent a relationship between a whole entity (the "whole" or "aggregated" entity) and its parts or components (the "part" entities). It is used to model relationships in which one entity is composed of or contains other entities. Aggregation is often used to represent a "has-a" relationship. An example is given below:




In the context of the Enhanced Entity-Relationship (EER) model, attribute and relationship inheritance refers to the way attributes and relationships are propagated from a more generalized entity (supertype) to one or more specialized entities (subtypes). Inheritance allows subtypes to inherit the attributes and relationships defined in the supertype, while also permitting the subtypes to have their unique attributes and relationships. A more detailed description is as follows:

- Attribute Inheritance:

  Attribute inheritance involves the propagation of attributes from a supertype to one or more subtypes. The supertype defines a set of attributes that are common to all entities in the hierarchy, and these attributes are automatically inherited by all subtypes. Subtypes can add their unique attributes to the inherited attributes from the supertype. Attribute inheritance is useful when there are attributes that are common to all entities in the hierarchy, and it ensures that these common attributes are defined and maintained only once at the supertype level.

- Relationship Inheritance

  Relationship inheritance involves the propagation of relationships from a supertype to subtypes. In addition to attributes, the relationships defined at the supertype level are inherited by the subtypes. Subtypes can have additional relationships that are specific to their role, and they can also refine or specialize the inherited relationships. Relationship inheritance is valuable when the relationships are common to all entities in the hierarchy and when there is a need to ensure that these relationships are defined consistently.

In both attribute and relationship inheritance, the goal is to manage and maintain the common characteristics and relationships effectively while accommodating the specific attributes and relationships unique to each subtype. This ensures that the data model accurately reflects the real-world domain and simplifies the design and maintenance of the database. Abstraction and inheritance are fundamental concepts in the EER model, allowing for a structured and hierarchical representation of data entities.

Several constraints and rules help ensure the consistency and integrity of the hierarchy. These constraints define how entities and relationships within the hierarchy should be structured. The most common constraints on ISA hierarchies are as follows:

- Overlap Constraint:

  The overlap constraint specifies whether the subtypes in a hierarchy can have overlapping sets of entities. There are two types of overlap constraints:
  - Disjoint:

    Subtypes are disjoint if their sets of entities have no overlap. An entity can belong to one and only one of the subtypes in the hierarchy.
  - Overlapping:

    Subtypes are overlapping if their sets of entities can overlap, meaning an entity can belong to more than one subtype simultaneously.
- Covering Constraint:

  The covering constraint specifies whether every entity in the supertype must be a member of at least one subtype (total specialization) or if it's possible for an entity in the supertype to not belong to any subtype (partial specialization). There are two types of completeness constraints:
  - Total Completeness:

    Every entity in the supertype must belong to at least one subtype.
  - Partial Completeness:

    Entities in the supertype are not required to belong to any subtype.

## Week 3

### Producing a Relational Schema from Conceptual Design

The process of producing a relational schema from a conceptual design, often created using techniques like Entity-Relationship Diagrams (ERDs) or the Enhanced Entity-Relationship (EER) model, involves mapping the high-level conceptual model to a set of relational database tables. This process includes several key steps:

- Identify Entities and Attributes:

  Review the conceptual model and identify the entities and their associated attributes. Each entity in the conceptual model typically corresponds to a table in the relational schema, and each attribute corresponds to a column within that table.

- Define Primary Keys:
  Determine the primary key for each table. The primary key uniquely identifies each row in the table. In many cases, the primary key may correspond to the primary identifier (e.g., StudentID, EmployeeID) specified in the conceptual model.
- Identify Relationships:
  Examine the relationships between entities in the conceptual model. For each relationship, determine how it should be represented in the relational schema. Typically, relationships are implemented using foreign keys in the related tables.
- Normalization:
  Normalize the tables to ensure that they adhere to standard forms (e.g., 1st Normal Form, 2nd Normal Form, 3rd Normal Form) to minimize data redundancy and improve data integrity.
- Data Types and Constraints:
  For each attribute in the schema, assign appropriate data types (e.g., VARCHAR, INTEGER, DATE) to ensure that data is stored in a consistent and structured manner. Apply constraints, such as NOT NULL, UNIQUE, and CHECK constraints, as needed to maintain data integrity.
- Handle Aggregation and Generalization/Specialization:
  If your conceptual model includes aggregations or inheritance hierarchies, decide how to represent these in the relational schema. For example, use tables and foreign keys to represent aggregations, and create separate tables for subtypes in specialization hierarchies.
- Normalize for Relationships:
  Normalize the schema further to address many-to-many relationships, resulting in junction (or associative) tables when necessary.
- Review Business Rules:
  Ensure that any business rules or constraints specified in the conceptual model are enforced in the relational schema. This might include rules for referential integrity, unique constraints, and default values.
- Indexing:
  Identify which columns would benefit from indexing for efficient data retrieval. Common candidates for indexing include primary key columns, foreign key columns, and columns frequently used in search criteria.
- Review and Refinement:
  Carefully review the relational schema for accuracy, completeness, and adherence to the conceptual design. Make any necessary refinements to ensure that the schema accurately represents the intended database structure.
- Documentation:
  Document the relational schema thoroughly, including table definitions, column descriptions, primary and foreign keys, indexes, and any relevant constraints. Proper documentation is essential for database maintenance and future development.
- Implement and Test:
  Create the database tables, relationships, and constraints in your database management system (DBMS) of choice. Populate the tables with data if necessary and perform testing to verify that the schema works as expected.

## Week 4

**Relational Algebra**

Relational algebra is a formal and mathematical query language used in relational database management systems (RDBMS) for performing operations on relational databases. It provides a theoretical foundation for the manipulation and retrieval of data stored in relational databases. Relational algebra operations are used to perform queries and transformations on tables (relations) to extract specific information from the database. There are several basic operations in relational algebra, which include:

- Selection (σ):
  This operation selects rows from a table that satisfy a specified condition. It is analogous to the SQL "WHERE" clause. The selection operation is represented by the Greek letter sigma (σ).
- Projection (π):
  The projection operation selects specific columns (attributes) from a table while removing the others. It is similar to the SQL "SELECT" statement but focuses on columns. The projection operation is represented by the Greek letter pi (π).
- Union (∪):
  The union operation combines two tables to produce a result that contains all distinct rows from both tables. It is analogous to the SQL "UNION" operator.
- Intersection (∩):
  The intersection operation returns rows that are common to both tables. It is like the SQL "INTERSECT" operator.
- Set Difference (-):
  The set difference operation returns rows that are in one table but not in another. It is like the SQL "EXCEPT" operator.
- Cartesian Product (×):
  The Cartesian product operation combines rows from two tables to create a new table where each row from the first table is paired with each row from the second table. It is analogous to joining all rows from one table with all rows from another table, resulting in a potentially large result set.
- Join (⋈):
  The join operation combines rows from two tables based on a common attribute (or a join condition) to create a new table. The join operation is similar to SQL "INNER JOIN," "LEFT JOIN," "RIGHT JOIN," and "FULL JOIN" operations.
- Rename (ρ):
  The rename operation is used to change the name of a relation (table) or to rename attributes within a table.

Relational algebra is used as the foundation for query optimization and execution within RDBMSs. It provides a consistent and formal way to express and manipulate database queries. The SQL query language, commonly used for interacting with relational databases, is heavily influenced by relational algebra. Relational algebra operations can be used to express SQL queries and serve as the basis for query optimization strategies within database systems.

Relational algebra plays a crucial role in Database Management Systems (DBMS) in several important ways:

- Query Language Foundation:
  Relational algebra serves as the theoretical foundation for query languages in DBMS, including Structured Query Language (SQL). It provides a formal, mathematical framework for expressing queries and operations on relational databases.

- Data Retrieval and Manipulation:

  Relational algebra operations are used to retrieve, filter, transform, and manipulate data stored in relational databases. These operations are used to express queries that extract specific information from the database.
- Query Optimization:

  Relational algebra is fundamental to query optimization in DBMS. DBMSs use query optimization techniques to determine the most efficient execution plan for a given query. This includes selecting the most appropriate join methods, access paths, and indexing strategies to minimize query processing time. The cost-based query optimizer uses algebraic rules and properties to optimize queries.
- Data Integrity and Constraints:

  Relational algebra helps ensure data integrity by providing operations for enforcing constraints, such as uniqueness constraints, referential integrity (foreign keys), and other business rules. This is essential for maintaining the consistency and correctness of data in the database.
- Data Definition and Schema Manipulation:

  While relational algebra primarily focuses on data manipulation, it can also be used for data definition and schema manipulation. This includes creating, altering, and dropping tables and constraints, which are often expressed using SQL's Data Definition Language (DDL) statements.
- Data Transformation and ETL (Extract, Transform, Load):

  Relational algebra is used for data transformation tasks in ETL processes. ETL tools and scripts use relational algebra operations to extract data from source systems, transform it into a format suitable for the target database, and load it into the destination database.
- Database Query Languages:

  SQL, the most widely used query language for relational databases, is heavily influenced by relational algebra. SQL statements for SELECT, INSERT, UPDATE, DELETE, and other operations can be seen as practical implementations of the relational algebra operations.
- Database Design and Normalization:

  Relational algebra principles and operations are used in the design and normalization of relational database schemas. They help ensure that databases are structured to minimize data redundancy, support data integrity, and maintain efficient data access.

**Relational approach to data**

The relational approach to data is a data modeling and database management methodology based on the principles of relational databases. It was introduced by Dr. Edgar F. Codd in the early 1970s, and it has become the dominant model for managing structured data in information systems. The relational approach to data is characterized by several key concepts:

- Relational Data Model:

  The core of the relational approach is the relational data model, which represents data as a collection of tables (relations) with rows (tuples) and columns (attributes). Each table corresponds to an entity, and each row represents a specific instance of that entity, while each column represents a specific attribute or property of the entity.

- Structured Data:
  The relational approach is well-suited for structured data, where the data can be organized into well-defined categories, and relationships between data entities can be explicitly modelled.
- Data Independence:
  The relational model promotes data independence, separating the physical storage of data from the logical representation. This means that applications can interact with data through high-level query languages (e.g., SQL) without needing to know the physical storage details.
- Normalization:
  The relational approach encourages the normalization of data to reduce data redundancy and improve data integrity. Normalization involves breaking down data into smaller, related tables and eliminating duplicate information.
- Consistency and Integrity:
  The relational model enforces data integrity constraints, such as primary key and foreign key constraints, to maintain data consistency and ensure that data remains accurate and reliable.
- Structured Query Language (SQL):
  The relational approach is closely associated with SQL, a powerful and widely used query language for managing relational databases. SQL allows users to retrieve, update, insert, and delete data using declarative commands.
- Joins:
  Relational databases support complex queries by allowing tables to be joined together based on related columns. This enables the retrieval of data from multiple tables in a single query.
- ACID Properties:
  Relational databases adhere to the ACID (Atomicity, Consistency, Isolation, Durability) properties, which ensure data consistency and reliability, even in the presence of system failures or concurrent access by multiple users.
- Scalability:
  While the relational approach was initially associated with on-premises monolithic databases, it has evolved to include distributed and cloud-based database systems, making it adaptable to a wide range of deployment scenarios.
- Data Security:
  Relational databases offer robust security features, including access control, authentication, and encryption, to protect sensitive data from unauthorized access.

**Set Operations**

Set operations in relational algebra are a set of operations that are used to manipulate relations (tables) and perform various set-related operations on the data within those tables. These operations are like the operations found in set theory, and they allow you to perform tasks such as filtering, combining, and comparing data sets within a relational database. The common set operations in relational algebra include:

- Union (∪):
  The union operation combines two relations to produce a result that contains all distinct rows from both input relations. Duplicate rows are eliminated. The union operation is represented by the symbol ∪.

- Intersection (∩):
  The intersection operation returns rows that are common to both input relations. It retrieves only the rows that appear in both input relations. The intersection operation is represented by the symbol ∩.
- Set Difference (-):
  The set difference operation returns rows that are in one input relation but not in the other. It retrieves rows that are unique to the first input relation. The set difference operation is represented by the symbol -.
- Cartesian Product (×):
  The Cartesian product operation combines rows from two relations to create a new relation. It pairs each row from the first relation with every row from the second relation, resulting in a potentially large result set. The Cartesian product operation is represented by the symbol ×.

**Composing Operations**

In relational algebra, you can compose multiple operations to create complex queries and manipulate data in a structured and controlled manner. Composing operations involves applying several relational algebra operations in sequence to achieve the desired result. Common operations include selection, projection, join, union, and intersection, among others.

**Projection**

Projection is one of the fundamental operations in relational algebra, which is used to extract specific columns (attributes) from a relation (table). It allows you to create a new relation that includes only the selected columns while omitting the rest. Projection is analogous to the SQL "SELECT" statement but focuses on selecting columns, not rows. The projection operation in relational algebra is represented by the Greek letter π (pi), followed by a list of attributes you want to project. In SQL, the equivalent operation is the "SELECT" statement, which allows you to specify the columns you want to retrieve from a table.

**Selection**

Selection is one of the fundamental operations in relational algebra used to extract specific rows (tuples) from a relation (table) that satisfy a specified condition or predicate. This operation is analogous to the SQL "WHERE" clause, allowing you to filter the rows in a relation based on a given criteria. The selection operation in relational algebra is represented by the Greek letter σ (sigma), followed by the condition, or predicate that specifies which rows should be selected. In SQL, the equivalent operation is the "SELECT" statement with a "WHERE" clause.

**Cross Product**

The cross product (also known as the Cartesian product) is a fundamental operation in relational algebra that combines rows from two relations (tables) to create a new relation. It pairs each row from the first relation with every row from the second relation, resulting in a result set that contains all possible combinations of rows from both relations. The result contains several rows equal to the product of the number of rows in the two input relations. The cross-product operation is represented by the symbol × in relational algebra. The cross product is often used when you need to perform operations involving all possible combinations of rows from two relations. However, it can result in many rows and is generally not used for typical query operations. In practice, the cross product is used less frequently than other relational algebra operations like selection, projection, and joins. Joins are often preferred when

you need to combine data from two relations based on related columns, as they provide more specific and meaningful results without generating all possible combinations.

**Joins**

Joins are fundamental operations in relational algebra and SQL that allow you to combine rows from two or more relations (tables) based on related columns. They are used to establish relationships between data stored in different tables, making it possible to retrieve and display information from multiple tables in a single result set. Joins are essential for querying relational databases. The most common types of joins include:

- Inner Join:

  An inner join returns only the rows for which there is a match in both input relations, based on a specified condition or predicate. It combines rows with matching values in the join columns and omits rows that don't have a match in both tables.
- Left Outer Join:

  A left join returns all rows from the left (first) table and the matching rows from the right (second) table. If there is no match for a row in the left table, the result will contain null values for the columns from the right table.
- Right Outer Join:

  A right join returns all rows from the right (second) table and the matching rows from the left (first) table. If there is no match for a row in the right table, the result will contain null values for the columns from the left table.
- Full Outer Join:

  A full outer join returns all rows from both input tables. It combines rows with matching values in the join columns and includes rows from both tables that don't have a match in the other table. If there is no match for a row in one table, the result will contain null values for the columns from the other table.
- Natural Join:

  A natural join is a join that combines rows from two tables based on columns with the same names. It automatically matches rows with the same values in columns with matching names, but it doesn't require specifying join conditions explicitly.

**Relational Division**

Relational division is a less common but powerful operation in relational algebra and SQL that allows you to query a relational database to find records in one table (relation) that are associated with a set of values in another table. It is the reverse of the Cartesian product and is used to find elements in one set that are related to all elements in another set. The operation is often expressed as R ÷ S where R is the dividend, representing the set of records you want to query and S is the divisor, representing the set of values you want to use as a filter. The result of the division operation (R ÷ S) is a new relation that contains records from R that are associated with all values in S. In other words, it finds records in R where, for each value in S, there is a matching record in R.

**Relational Expressions**

In a database management system (DBMS), relational expressions refer to logical conditions or criteria used to filter, join, project, and manipulate data in relational databases. These expressions are an integral part of querying and working with data within a DBMS. They are used to specify conditions that determine which data is included or excluded in query results, and they play a crucial role in data retrieval and manipulation. Relational expressions in a DBMS typically involve the following components:

- Comparison Operators:
  Relational expressions often employ comparison operators to compare values or attributes in the database. Common comparison operators include "=", "<", ">", "<=", ">=", "<>", and more. These operators are used to express equality, inequality, and other conditions.
- Logical Operators:
  Logical operators, such as "AND", "OR", and "NOT," allow you to combine multiple conditions in a relational expression. Logical operators are used to specify the relationships between conditions, enabling the creation of complex conditions.
- Parentheses:
  Parentheses are used to group conditions and control the order of evaluation within a relational expression. They help ensure that conditions are combined in the intended manner, especially when you have multiple conditions and logical operators.

Relational expressions are used in various aspects of a DBMS, including:

- Querying with SELECT Statements:
  In SQL, for instance, relational expressions are employed in the WHERE clause of a SELECT statement to filter rows based on specified conditions. This allows you to retrieve only the data that meets specific criteria.
- Join Operations:
  Relational expressions are used to define join conditions when combining data from multiple tables. Join conditions specify how rows in different tables are related and how they should be matched.
- Data Modification:
  In SQL, relational expressions are used in UPDATE and DELETE statements to specify the criteria for updating or deleting specific rows in a table.

**Equivalence Rules**

In database management systems (DBMS), equivalence rules refer to a set of rules and transformations that help maintain the logical consistency and integrity of a relational database. These rules ensure that operations, such as query optimization and data manipulation, produce equivalent results while preserving the original meaning of the operations. Equivalence rules are particularly important for query optimization, where the DBMS aims to find the most efficient execution plan for a given query without altering the query's semantics. Common equivalence rules in DBMS include:

- Commutative Rules:
  These rules apply to binary operations (e.g., joins and set operations) and state that the order of the operands does not affect the result. For example, the commutative rule for a union operation ($\cup$) state that $R \cup S$ is equivalent to $S \cup R$.
- Associative Rules:
  These rules apply to binary operations and state that the grouping of operations does not affect the result. For example, the associative rule for a union operation states that $(R \cup S) \cup T$ is equivalent to $R \cup (S \cup T)$.
- Distributive Rules:
  These rules specify how operations can be distributed over other operations. For example, the distributive rule for joins over unions states that $R \bowtie (S \cup T)$ is equivalent to $(R \bowtie S) \cup (R \bowtie T)$.

- Identity Rules:
  These rules involve the identity element for certain operations. For example, the identity rule for union states that R ∪ ∅ is equivalent to R, where ∅ represents an empty relation.
- Selection Equivalence Rules:
  These rules apply to selection (σ) operations. They include the commutative and distributive properties of selections, such as σ(A) ∩ σ(B) is equivalent to σ(A ∩ B).
- Projection Equivalence Rules:
  These rules apply to projection (π) operations. They include the commutative and distributive properties of projections, such as π(A) ∪ π(B) is equivalent to π(A ∪ B).
- Join Equivalence Rules:
  These rules apply to join (⋈) operations and specify how join operations can be re-arranged to achieve equivalent results, such as (R ⋈ S) ⋈ T is equivalent to R ⋈ (S ⋈ T).
- Set Operation Equivalence Rules:
  These rules apply to set operations (union, intersection, set difference) and specify how these operations can be transformed to produce equivalent results. For example, (R ∪ S) ∩ T is equivalent to (R ∩ T) ∪ (S ∩ T).

Equivalence rules are fundamental for query optimization and algebraic manipulations in a DBMS. They help the DBMS understand that different ways of expressing a query can be optimized to produce the same result. This knowledge enables the DBMS to choose efficient execution plans and improve query performance without altering the query's meaning. These rules are an essential part of the relational algebra, which is the foundation for query languages used in relational databases.

## Week 5

**Natural Language Descriptions**

Natural language descriptions in a database management system (DBMS) refer to using human-readable, plain language explanations or descriptions to document and provide information about the database's structure, data, and operations. These descriptions are intended to make the database more understandable and accessible to non-technical users, such as database administrators, business stakeholders, and end-users. Natural language descriptions in a database management system (DBMS) typically consist of various components that provide clear and understandable explanations of database elements, operations, and constraints. These descriptions use natural language, and they often include the following types of components:

- Nouns (Entities and Objects):
  Nouns in natural language descriptions refer to the entities, objects, and components within the database. These can include table names, column names, views, and other database objects. Nouns help identify the specific elements being described. Example: "The 'Employees' table contains information about company employees, including their names, salaries, and departments."
- Verbs (Actions and Operations):
  Verbs describe actions, operations, or behaviours associated with the database. They explain what the database elements do, how they are used, and the operations performed on them. Example: "The 'CalculateSalary' stored procedure computes the monthly salary for each employee based on their working hours and hourly rate."

- Adjectives (Qualities and Characteristics):

  Adjectives are used to provide additional information about the database elements, such as their characteristics, properties, or qualities. Adjectives help users understand the attributes and features of the elements. Example: "The 'Product' table contains detailed information about each product, including its name, description, price, and availability status."
- Constraints and Rules:

  Constraints and rules are essential components in natural language descriptions, as they define the limitations and requirements associated with database elements. This includes information about primary keys, foreign keys, unique constraints, check constraints, and any business rules. Example: "The 'Orders' table enforces a primary key constraint on the 'OrderID' column to ensure that each order has a unique identifier."
- Relationships and Associations:

  Descriptions often include information about the relationships between tables and how data is related. This helps users understand the data model and how various tables are connected. Example: "The 'Customers' table is linked to the 'Orders' table through a foreign key relationship on the 'CustomerID' column, indicating which customer placed each order."
- Data Types:

  Explanation of the data types used for columns is important, especially when users need to understand the kind of data stored in each column. Example: "The 'Birthdate' column in the 'Employees' table uses the 'DATE' data type to store each employee's date of birth."
- Business Rules and Logic:

  Descriptions may include details about the business rules, calculations, and logic applied to data. This helps users understand how data is processed within the database. Example: "The 'Discount' column in the 'Orders' table stores the calculated discount for each order based on the customer's loyalty level."
- Use Cases and Examples:

  Providing real-world use cases and examples can help users grasp the practical application of the database elements and operations. Example: "The 'GenerateInvoice' stored procedure generates an invoice for a customer's recent order by aggregating order details and applying appropriate taxes."
- Comments and Clarifications:

  Descriptions may also include comments or clarifications to provide additional context or explanations for complex database elements and operations. Example: "The 'Location' column in the 'Store' table indicates the physical address of each store location."

**Creating a Conceptual Model**

Creating a conceptual model for a database involves a systematic process that helps in defining the structure and relationships of the data to be stored in the database. The conceptual model serves as an abstract representation of the data, focusing on the data's semantics and high-level structure rather than technical implementation details. Here are the steps involved in creating a conceptual model:

- Identify the Purpose and Requirements:

  Begin by understanding the purpose of the database and the requirements it needs to fulfill. Meet with stakeholders, end-users, and subject matter experts to gather and document their requirements. Define the scope of the database, including the types of data it will store and the expected functionality.

- Identify Entities:

  Identify the main entities (objects or things) that need to be represented in the database. These entities should reflect the key concepts or elements in the domain you are modelling.
- Define Attributes:

  For each identified entity, specify the attributes (properties, characteristics) that describe the entity. Attributes provide details about the entities and are essential for defining the structure of the data.
- Model Relationships:

  Determine the relationships between entities. Relationships describe how entities are connected or associated with each other. Identify one-to-one, one-to-many, and many-to-many relationships.
- Create an Entity-Relationship Diagram (ERD):

  Develop an Entity-Relationship Diagram, a visual representation of the entities, attributes, and relationships. Use symbols and notations to represent entities (rectangles), attributes (ovals or ellipses), and relationships (lines and diamonds).
- Specify Constraints:

  Define constraints, including integrity constraints like primary keys, foreign keys, unique constraints, and check constraints. These constraints ensure data quality and enforce rules.
- Document the Model:

  Provide detailed descriptions for entities, attributes, relationships, and constraints. This documentation helps others understand the model and its purpose.
- Review and Validate:

  Conduct a review with stakeholders and subject matter experts to validate the conceptual model. Ensure that it accurately reflects the requirements and domain knowledge.
- Iterate and Refine:

  Be prepared for iterative refinement of the conceptual model. As you receive feedback and gather more information, make necessary adjustments to the model.

## Week 6

**Schema Design Process**

Schema design is a crucial step in developing a database that meets the needs of an organization. It involves defining the structure of the database, including tables, columns, relationships, keys, and constraints. The relational schema is best obtained by starting with a conceptual design. Evaluating a database involves assessing its performance, reliability, security, and overall efficiency. This evaluation is crucial to ensure that the database meets the organization's requirements and performs optimally. Here are key aspects to consider when evaluating a database:

- Performance:
  - Query Performance: Evaluate the speed and efficiency of database queries. Ensure that common queries, reports, and operations execute in a reasonable time frame.
  - Scalability: Assess whether the database can handle increasing data volumes and user loads. Consider the need for horizontal or vertical scaling.

- Reliability and Availability:
    - Uptime and Availability: Measure the database's availability and uptime. Ensure that it meets the organization's availability requirements and has adequate failover mechanisms in place.
    - Data Recovery: Test and evaluate the database's backup and recovery processes to ensure data can be restored in case of data loss or system failures.
- Security:
    - Access Control: Review access control mechanisms to ensure that only authorized users have access to sensitive data. Verify that role-based access and permission levels are correctly implemented.
    - Data Encryption: Evaluate data encryption methods to protect data at rest and in transit.
    - Auditing and Monitoring: Ensure that comprehensive auditing and monitoring are in place to track database activities and detect unauthorized access or suspicious activities.
- Data Integrity:

  Verify that data constraints and integrity rules are enforced. Check for data anomalies, duplicate records, or inconsistencies. Ensure that referential integrity is maintained through proper use of foreign keys and relationships.
- Scalability and Performance Optimization:

  Assess whether the database can handle increasing data volumes and user loads. Consider techniques such as indexing, query optimization, and caching to improve performance.
- Data Model and Schema Design:

  Evaluate the database's schema and data model to ensure it aligns with the organization's requirements. Assess whether normalization is appropriate for data storage. Consider how well the database accommodates changes and new data requirements.
- Backup and Recovery:

  Verify the database's backup and recovery processes. Test data restoration from backups to ensure data integrity. Review backup strategies, backup frequency, and retention policies.
- Data Archiving and Purging:

  Ensure that data archiving and purging strategies are in place to manage historical data and improve database performance.
- Compliance:

  Ensure that the database complies with relevant data protection regulations and industry standards, such as GDPR, HIPAA, or PCI DSS.
- User Feedback:

  Solicit feedback from end-users and administrators to understand their experiences and any issues they encounter when interacting with the database.
- Documentation:

  Review the database documentation, including schema diagrams, data dictionaries, and data flow diagrams. Ensure that documentation is up to date and readily available.
- Costs and Budget:

  Evaluate the total cost of ownership (TCO) of the database, including licensing, maintenance, hardware, and operational costs. Ensure that it aligns with the organization's budget and goals.

- Future Requirements:
  Consider the organization's future data management needs and assess whether the current database can accommodate future growth and new features.
- Disaster Recovery Plan:
  Ensure that a robust disaster recovery plan is in place to address potential data loss or system failures.

**Redundancy**

Redundancy in the context of databases refers to the practice of storing the same piece of data in multiple locations or tables within a database. While some degree of redundancy can be useful for certain purposes, excessive redundancy can lead to various issues and is generally discouraged in well-designed databases. There are three main types of anomalies: insertion anomalies, update anomalies, and deletion anomalies. These anomalies can impact data integrity and database functionality, making it essential to understand and address them during the database design and maintenance process.

- Insertion Anomalies:
  Insertion anomalies occur when there are difficulties or errors in adding new data to a database. These anomalies typically involve problems related to adding data to tables. Common insertion anomalies include:
  - Incomplete Information: When not all required information is available or known, inserting a new record can be challenging. For example, if you cannot insert a new order into the database without knowing the customer's address, it leads to incomplete data.
  - Redundant Data: Inserting data that already exists in other records or tables can lead to redundancy. For instance, entering a customer's address multiple times for different orders results in redundant information.
- Update Anomalies:
  Update anomalies occur when there are difficulties or errors in modifying existing data. These anomalies often involve inconsistencies that result from updates made to some, but not all, occurrences of a piece of data. Common update anomalies include:
  - Inconsistent Data: If data in a database is not consistently updated, discrepancies can arise. For example, if the price of a product is updated in some records but not in others, the database contains inconsistent data.
  - Data Integrity Issues: Updates can introduce data integrity problems, especially if related data is not updated together. For instance, updating a customer's address in one table but not updating it in another can lead to integrity issues.
- Deletion Anomalies:
  Deletion anomalies occur when there are difficulties or errors in removing data from a database. These anomalies often involve unintentional data loss or problems associated with deleting data. Common deletion anomalies include:
  - Data Loss: Deleting a single record might result in the loss of valuable data that is not stored elsewhere. For example, deleting an order could also remove customer information that is not associated with any other orders.
  - Referential Integrity Issues: Deletions can cause referential integrity constraints to be violated if data that is referenced in other parts of the database is deleted without proper handling.

**Functionality Dependency**

A functional dependency in the context of a relational database refers to a relationship between two sets of attributes (columns) within a table. It describes how the values in one set of attributes uniquely determine the values in another set. In simpler terms, if you have a functional dependency between two sets of attributes, knowing the values in one set allows you to determine the values in the other set with certainty.

Functional dependencies play a fundamental role in the process of database normalization, where tables are decomposed to reduce data redundancy and improve data integrity. The higher the level of normalization achieved, the more functional dependencies have been identified and eliminated, resulting in a more efficient and well-structured database schema. Functional Dependency are written as X → Y.

Discovering or deducing functional dependencies (FDs) is an essential step in the database design and normalization process. Here are some methods and techniques for deducing FDs:

- Observation and Understanding:

  Start by examining the data and the meaning of the attributes within the tables. Understanding the semantics of the data can provide insights into potential FDs. For example, in a database of employees, you can deduce that each employee's Social Security Number (SSN) uniquely determines their full name because two employees cannot share the same SSN.
- Closure of Attribute Sets:

  The closure of an attribute set is a systematic way to determine all FDs that can be derived from a set of attributes. To find the closure of a set X, you can repeatedly apply Armstrong's axioms, a set of logical rules, to X and its functional dependencies until you can't derive any more attributes. The closure of X is denoted as X+. Armstrong's axioms include:
  - Reflexivity: If Y is a subset of X, then X → Y.
  - Augmentation: If X → Y, then XZ → YZ for any Z.
  - Transitivity: If X → Y and Y → Z, then X → Z.
  - Union Rule: If X → Y and X → Z, then X → YZ. This rule allows you to combine FDs when they share a common set of attributes on the left-hand side.
  - Decomposition Rule: If X → YZ, then X → Y and X → Z. The decomposition rule allows you to split an FD into multiple smaller FDs.
  - Pseudo transitivity Rule: If X → Y and WY → Z, then WX → Z. This rule extends the concept of transitivity to include an additional attribute.

Attribute closure, often denoted as X+, represents the set of all attributes that can be functionally determined by a given set of attributes X, according to a set of functional dependencies (FDs) in a relational database. The attribute closure is a fundamental concept in the context of reasoning about FDs, normalization, and database design. To find the attribute closure X+ under a set of FDs, you can follow a systematic process, which is usually based on Armstrong's axioms and involves repeatedly applying these axioms to the set of attributes X. The closure X+ is the set of all attributes that can be derived by applying the given FDs to X.

**Normalisation**

Normalization is a database design technique that aims to minimize data redundancy and dependency by organizing data into separate tables and structuring relationships between

those tables. The process of normalization results in the creation of a set of rules known as normal forms, which specify the criteria for a well-structured and efficient relational database. The primary goals of normalization are to:

- Reduce Data Redundancy: By eliminating duplicate data, normalization reduces storage requirements and ensures that data is consistent.
- Improve Data Integrity: Normalization enforces referential integrity and minimizes update anomalies, ensuring that data remains accurate and reliable.
- Enhance Query Performance: A well-normalized database can improve query performance by allowing for efficient data retrieval.

There are several normal forms in database design, each building upon the rules of the previous form. The most used normal forms are:

- First Normal Form (1NF): In 1NF, a table must have a primary key, and all attributes must be atomic (indivisible). Each column should contain only one piece of data.
- Second Normal Form (2NF): To achieve 2NF, a table must already be in 1NF, and all non-key attributes must be functionally dependent on the entire primary key, not on just a portion of it.
- Third Normal Form (3NF): A table in 3NF must be in 2NF, and it should have no transitive dependencies. In other words, non-key attributes should not depend on other non-key attributes.
- Boyce-Codd Normal Form (BCNF): BCNF is a stricter form of 3NF. To achieve BCNF, a table must meet 3NF requirements, and all attributes must be functionally dependent on the primary key.

Decomposition, in the context of database design, refers to the process of breaking down a single, complex table into multiple smaller tables or relations. This process is often used as part of the normalization technique to reduce data redundancy and improve data integrity. The objective of decomposition is to achieve a higher normal form (such as Third Normal Form, Boyce-Codd Normal Form, or higher) in the relational database.

Normalization is the process of structuring a relational database in a way that reduces data redundancy and improves data integrity. It involves decomposing large tables into smaller, related tables and creating relationships between them. The goal is to organize data to minimize duplication and potential anomalies while optimizing query performance. Here are the steps to normalize a database:

- Identify the Functional Dependencies (FDs):

  Begin by identifying the functional dependencies that exist within your dataset. FDs describe the relationships between attributes in a table and specify which attributes can be functionally determined by others.
- Create an Entity-Relationship Diagram (ERD):

  Design an ERD that illustrates the entities (tables) in your database and the relationships between them. ERDs, help you visualize the structure of your database and the entities that need to be normalized.
- Choose a Normal Form:

  Determine the level of normalization you want to achieve based on the specific requirements of your application. Common normal forms include First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), and Boyce-Codd Normal Form (BCNF), among others. Each normal form has its own set of rules and requirements.

- Create New Tables:
  Based on the FDs and the chosen normal form, create new tables to represent the data in a more structured way. Each table should have a clear and single-purpose representation.
- Define Primary Keys:
  Determine the primary key for each new table. The primary key uniquely identifies each row in the table. In many cases, the primary key includes one or more attributes that represent unique identifiers.
- Define Foreign Keys:
  Establish relationships between tables by defining foreign keys in the child tables. Foreign keys create referential integrity constraints, ensuring that data remains consistent and accurate across related tables.
- Remove Partial Dependencies:
  If you're normalizing to at least 2NF or higher, eliminate partial dependencies by creating new tables that separate related data. Partial dependencies occur when non-prime attributes depend on only part of a candidate key.
- Remove Transitive Dependencies:
  In 3NF or higher normalization, eliminate transitive dependencies. Transitive dependencies occur when non-prime attributes depend on other non-prime attributes through the primary key.
- Ensure Data Integrity:
  Enforce referential integrity constraints by specifying that foreign keys in child tables must match primary keys in parent tables. This ensures data consistency and prevents orphaned records.
- Test the Normalized Schema:
  Insert sample data into the normalized tables and run test queries to ensure that the database functions as expected. Verify that data is correctly stored and retrieved.
- Optimize Query Performance:
  As you normalize, consider query performance. Create appropriate indexes and analyse query patterns to ensure that your normalized schema allows efficient data retrieval.
- Document the Schema:
  Maintain thorough documentation of your normalized schema, including the structure of tables, their attributes, and the relationships between them.

Decomposition of a relational schema involves breaking down a single, complex table into multiple smaller tables, with the goal of improving data organization, reducing data redundancy, and enhancing data integrity. The decomposition process is a key aspect of database normalization and ensures that data is structured efficiently. Lossless join decomposition is a database design technique that involves breaking down a complex, unnormalized table into multiple smaller tables in a way that preserves the ability to recombine them through a natural join operation without losing any information. In other words, lossless join decomposition ensures that you can reconstruct the original data by joining the decomposed tables without any loss of data. If in the decomposed tables, the join has extra tuples then the decomposition is not lossless as knowledge was lost.

Checking for lossless join decomposition involves verifying that you can recombine decomposed tables through a natural join operation to retrieve the original data without any loss or introduction of spurious data. There are formal methods and techniques to check for the lossless join property, including the Chase Test and the Dependency Preservation Theorem.

**Theorem:** The decomposition of *R* into X and Y is lossless-join wrt *F* if and only if the closure of F contains either

- $X \cap Y \rightarrow X$, or
- $X \cap Y \rightarrow Y$, or both

Dependency-preserving decomposition is a database design technique in which a single, unnormalized table is decomposed into multiple smaller tables in a way that preserves the original functional dependencies (FDs) of the data. The primary goal of dependency-preserving decomposition is to ensure that the FDs identified in the original table continue to hold true in the decomposed tables. This approach helps maintain data integrity and consistency while organizing data more efficiently.

## Week 7

### Introduction

Data breaches and security failures are significant challenges in the world of information technology and cybersecurity. They occur when unauthorized individuals or entities gain access to sensitive or confidential data, resulting in the exposure, theft, or compromise of that data. These incidents can have severe consequences for organizations and individuals, including financial losses, reputational damage, and legal liabilities. Here's an overview of data breaches and security failures:

- Data Breach

  A data breach is an incident where unauthorized access to data leads to the exposure, theft, or compromise of sensitive information. This can include personal data (such as names, addresses, Social Security numbers), financial information (credit card numbers, bank account details), intellectual property, or any other confidential data. Data breaches can occur through various means, such as hacking, malware, physical theft, or social engineering attacks.
- Security Failure

  A security failure is a broader term that encompasses any event in which a security control, process, or mechanism does not function as intended, leading to a vulnerability or breach. Security failures can result from technical issues, human errors, or a combination of both. They may involve vulnerabilities in software or hardware, weak passwords, misconfigured systems, lack of employee training, or other factors that can compromise security.

### Database Security

Database security is a critical aspect of information technology and data management that focuses on protecting the confidentiality, integrity, and availability of data stored within a database. Ensuring the security of databases is essential because they often contain sensitive and valuable information, making them prime targets for cyberattacks and unauthorized access. Data security encompasses several critical goals, which collectively aim to protect the confidentiality, integrity, and availability of data. These goals are fundamental in ensuring that data remains secure and trustworthy. The main data security goals are:

- Confidentiality:
  - Goal: Protect data from unauthorized access.
  - Description: Confidentiality ensures that data is only accessible to authorized individuals or entities. It involves measures such as access controls, encryption, and authentication to prevent unauthorized access to sensitive information.

- Integrity:
    - Goal: Ensure data is accurate and unaltered.
    - Description: Integrity safeguards data from unauthorized or unintended changes, ensuring that data remains reliable and consistent. Techniques like data validation, checksums, and digital signatures are used to verify data integrity.
- Availability:
    - Goal: Ensure data is available when needed.
    - Description: Availability ensures that data is accessible and usable by authorized users at any required time. This goal involves redundancy, fault tolerance, and disaster recovery measures to prevent downtime.
- Authenticity:
    - Goal: Verify the identity of users and data sources.
    - Description: Authenticity ensures that users are who they claim to be and that data sources are legitimate. Techniques like authentication, digital certificates, and public key infrastructure (PKI) help achieve authenticity.
- Accountability:
    - Goal: Attribute actions to specific users.
    - Description: Accountability tracks and logs the actions of users in the system, enabling the tracing of activities to specific individuals. Audit trails and logging mechanisms are used for accountability.
- Privacy:
    - Goal: Protect sensitive or private information.
    - Description: Privacy focuses on safeguarding personally identifiable information (PII) and other sensitive data from unauthorized access or disclosure. Techniques include data anonymization, access controls, and encryption.

Security policies are essential documents that outline an organization's approach to information security. These policies provide a framework for managing and protecting data, information systems, and network resources. Security policies help ensure that an organization's data and systems remain secure, and they guide employees, contractors, and other stakeholders in adhering to best practices for security. Security mechanisms are tools and techniques used to protect information, systems, and networks from unauthorized access, attacks, and vulnerabilities. These mechanisms help organizations implement security measures to safeguard their data and assets.

Data minimalism is a philosophy and approach that emphasizes collecting, storing, and using only the minimum amount of data necessary for a specific purpose while minimizing the collection of extraneous or unnecessary information. This approach is rooted in the principles of data privacy and protection, as well as ethical considerations regarding the handling of personal information. Data privacy, also known as information privacy, refers to the protection of an individual's personal information and how it is collected, used, shared, and managed. It involves safeguarding sensitive data from unauthorized access, disclosure, and misuse. Data privacy is a fundamental right, and it is crucial for maintaining trust in the digital age.

**Database Access Control**

Database access control is the practice of managing and restricting access to a database system, its data, and the operations that can be performed within it. It is a crucial component of database security, helping protect sensitive information from unauthorized access, modification, and misuse. Database access control encompasses various mechanisms and

strategies to ensure that only authorized users and applications can interact with the database. Database access control is essential for safeguarding the integrity, confidentiality, and availability of data within an organization's database systems. It ensures that only authorized individuals and applications can interact with data, reducing the risk of data breaches and security incidents. Properly implementing access control measures requires a deep understanding of the database system's features and capabilities and adherence to best practices in security. Here are key aspects of database access control:

- Authentication:

  Authentication is the process of verifying the identity of users or entities trying to access the database. This typically involves validating usernames and passwords or other authentication methods, such as biometrics or multi-factor authentication.
- Authorization:

  Authorization determines what actions and data a user or application is allowed to access within the database. Authorization is based on roles, privileges, and permissions assigned to users or roles. The principle of least privilege is often applied, which means users are granted the minimum access necessary to perform their tasks.

In SQL databases, access control is managed using various mechanisms and SQL statements to control who can perform specific actions on database objects. SQL databases typically have their own user management systems. You can create and manage users who are allowed to connect to the database. Common SQL statements for user management include CREATE USER, ALTER USER, DROP USER, and GRANT or REVOKE privileges from users.

The GRANT command in SQL is used to assign specific privileges to users, roles, or other database objects, allowing them to perform various operations on database objects. The specific privileges associated with the GRANT command may vary depending on the database management system (DBMS) being used. However, some common privileges and their associated operations include:

- The SELECT privilege allows a user or role to retrieve data from a table or view.
- The INSERT privilege permits the insertion of new rows into a table.
- The UPDATE privilege allows a user to modify existing data in a table.
- The DELETE privilege allows the removal of rows from a table.
- The EXECUTE privilege is used for stored procedures, functions, and other executable database objects. It enables the execution of these objects.
- The REFERENCES privilege is typically associated with foreign keys and allows a user to refer to columns in another table in a foreign key relationship.
- The USAGE privilege is often used for schemas and allows a user to access and use objects within a specific schema.

The REVOKE command in SQL is used to revoke previously granted privileges from users, roles, or other entities on database objects. It allows database administrators to remove specific permissions from individuals or roles, thereby restricting their access to certain operations on database objects. The privileges that can be revoked using the REVOKE command are the same as those granted using the GRANT command.

**Column Specific Access Control**

Column-specific access control, also known as column-level security or fine-grained access control, is a security mechanism in a database that allows you to control access to individual columns within database tables. This means that you can grant or revoke privileges

to specific columns for users or roles, giving you fine-grained control over who can view, modify, or query specific data within a table. This is particularly useful in scenarios where certain columns contain sensitive or confidential information, and you want to restrict access to that data.

Column-specific access control is particularly useful in scenarios where different user roles require varying levels of access to specific data within a table. By implementing this level of granularity, you can enhance data security and privacy, ensuring that users can only access the data they are authorized to see or modify. It's important to choose the right approach for your specific database system, as the implementation of column-specific access control may vary between different database management systems.

**Content Based Access Control**

Content-based access control (CBAC) is an access control mechanism that considers the content or context of the data being accessed when making access control decisions. Instead of relying solely on traditional access control methods, which typically consider the identity of the user or their role, CBAC considers the actual content of the data and the context in which the access request is made. This approach adds an extra layer of granularity and security to access control.

**Role based Authorization:**

Role-based authorization, also known as role-based access control (RBAC), is a widely used and effective method for managing and controlling access to resources, including databases and information systems. In RBAC, access control decisions are based on the roles that users or entities assume within an organization. This approach simplifies access management by grouping users into roles, each of which is associated with a set of permissions or privileges.

**User Identity Authentication**

User identity authentication is the process of verifying and confirming the identity of an individual or entity seeking access to a system, application, or resource. It is a fundamental component of cybersecurity and access control, ensuring that only authorized users can access specific systems and data. User identity authentication typically involves confirming the identity of users through various methods and mechanisms. In a Database Management System (DBMS), the term "accounts" typically refers to user accounts or database accounts. User accounts are used to manage and control access to the database system, and they play a crucial role in database security and administration. Here's an overview of DBMS accounts:

- Database User Accounts:

  These accounts are created to allow individuals, applications, or services to access and interact with the database. Each user account is associated with specific permissions and privileges that determine what actions the user can perform within the database.
- Types of Database User Accounts:
  - Database user accounts can be categorized into different types based on their roles and responsibilities. Some common types include:
  - Database Administrators (DBAs): DBAs have extensive privileges and are responsible for managing and maintaining the database system.
  - Application Users: These accounts are used by applications or services to access the database for specific tasks.

- End Users: End user accounts are used by individuals who interact with the database through applications or user interfaces.

**Relational Views**

Relational views, often referred to simply as "views," are a concept in relational database management systems (RDBMS) that allow you to create virtual tables based on the result of a SQL query. These virtual tables, or views, are not actual physical tables but rather dynamic subsets of data from one or more underlying tables. Views are used to simplify data access, enhance security, and provide a consistent and convenient way to interact with the data in the database. To create a view in a relational database using SQL, you can use the CREATE VIEW statement. Below is the basic syntax for creating a view:

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

Views offer several benefits in the context of relational databases, making them a valuable tool for database management and application development. Here are some of the key benefits of using views:

- Data Abstraction:
  Views allow you to abstract the underlying data by presenting a simplified and user-friendly interface. Users can work with views without needing to understand the complexity of the database schema.
- Security:
  Views can enhance data security by restricting access to sensitive information. You can define views that expose only the necessary data for specific user roles or applications while hiding other data in the underlying tables.
- Complex Query Encapsulation:
  Views are useful for encapsulating complex and frequently used SQL queries. Instead of writing the same complex query multiple times in application code, you can create a view with the query and then reference the view in your application.
- Performance Optimization:
  Views can be used to optimize query performance by precomputing aggregations, joining tables, or denormalizing data. This can reduce the computational load on application code and improve query response times.
- Data Consistency:
  Views can help ensure data consistency by providing a single point of access to data. This reduces the risk of data discrepancies that may occur if data is accessed directly from multiple tables.
- Readability:
  Views improve query readability and maintainability. They provide meaningful names for sets of data, making SQL queries more self-explanatory and easier to understand.
- Reusability:
  Views are reusable database objects. Once a view is created, it can be used in multiple queries, reports, or applications, reducing the need to rewrite the same query logic.

- Data Partitioning:
  Views can be used to partition data into smaller, more manageable subsets. This is particularly useful for very large databases, where data can be partitioned based on date ranges or other criteria.
- Role-Based Access Control:
  Views can be used in role-based access control (RBAC) to ensure that users have access to specific subsets of data that align with their roles and responsibilities.

Updating views in a relational database can be a bit more complex than querying them. Whether you can update a view depends on various factors, including the complexity of the view definition, the database system you're using, and the permissions you have.

**Semantic Integrity Constraints**

Semantic integrity constraints are a set of rules or conditions applied to the data in a relational database to ensure the accuracy, consistency, and meaningfulness of the data. These constraints go beyond the basic structural constraints, such as primary keys and foreign keys, and focus on maintaining the quality and semantic correctness of the data. Semantic integrity constraints are typically defined and enforced using business rules and application-specific logic. Here are some common types of semantic integrity constraints:

- Domain Constraints:
  These constraints define the permissible values that can be stored in a column. They ensure that data falls within a specific domain or range. For example, a domain constraint on an "age" column may specify that the age value must be a positive integer between 0 and 120.
- Entity Integrity:
  Entity integrity constraints ensure that the primary key of a table is unique and that it does not contain NULL values. This constraint is fundamental to maintaining data accuracy and ensuring that each row in a table can be uniquely identified.
- Referential Integrity:
  Referential integrity constraints ensure the consistency of data in related tables by enforcing foreign key relationships. They specify that a foreign key value in one table must match a primary key value in another table. This maintains the integrity of relationships between tables.

**Assertions**

Assertions in the context of databases refer to a type of integrity constraint used to enforce specific rules and conditions on the data within the database. Assertions are logical expressions or predicates that are defined by users or database administrators to ensure that the data in the database adheres to certain business rules or requirements. These rules can be more complex than standard integrity constraints like primary keys, foreign keys, or check constraints.

Constraints checked externally, also known as “external constraints” or “application-level constraints,” are rules and conditions that are enforced and validated outside of the database management system (DBMS) or database itself. These constraints are typically implemented and enforced by the application or software that interacts with the database rather than being defined and maintained within the DBMS.

**Transaction**

In the context of a database management system (DBMS), a transaction is a logical unit of work that represents one or more related database operations. Transactions ensure the integrity, consistency, and reliability of the data within a database by following the principles of the ACID properties:

- Atomicity:

  A transaction is atomic, meaning it is treated as a single, indivisible unit. All the operations within a transaction are executed as a whole or not at all. If any part of the transaction fails, the entire transaction is rolled back, and the database is left unchanged.
- Consistency:

  A transaction takes the database from one consistent state to another consistent state. This means that the data must satisfy all integrity constraints before and after the transaction. If a transaction violates these constraints, it is rolled back.
- Isolation:

  Transactions are typically executed in isolation from each other, meaning that the operations of one transaction are not visible to other transactions until the first transaction is committed. This ensures that concurrent transactions do not interfere with each other.
- Durability:

  Once a transaction is committed, its changes are permanent and survive any subsequent system failures. The changes are stored in a durable and persistent form, ensuring that data is not lost.

Transactions are used to maintain data integrity in multi-user, multi-operation environments. They allow multiple users to work with the database concurrently without corrupting the data or violating integrity constraints. Transactions are employed in various applications and systems where data consistency is critical, such as banking systems, airline reservation systems, e-commerce platforms, and more.

**Constraint Checking**

In the context of database transactions and constraints, "deferred constraint checking" is a concept that relates to when the database management system (DBMS) validates integrity constraints, such as referential integrity constraints (foreign key constraints) or check constraints. Deferred constraint checking allows the constraints to be checked at a specific point in a transaction, rather than immediately as each data modification operation (e.g., insert, update, delete) is performed. Deferred constraint checking can be particularly useful in scenarios where you need to perform a sequence of related data modifications that temporarily violate integrity constraints but where you can guarantee that the final state of the data will be consistent. It allows you to avoid the need for explicit constraint manipulation within the transaction and gives you more control over the order in which data is updated.

**Triggers**

Triggers are database objects or stored procedures in a database management system (DBMS) that are automatically executed in response to specified events or conditions. Triggers are designed to enhance the functionality and data integrity of a database by allowing you to automate certain actions, enforce business rules, and maintain data consistency. Triggers are commonly used in relational database systems, such as MySQL, PostgreSQL, Oracle, SQL Server, and more. Triggers are a powerful feature in relational databases, providing a way to automate actions and enforce rules that ensure data integrity and consistency. However, they should be used judiciously and with careful consideration of their impact on database

performance and behaviour. Triggers offer several advantages in a database management system (DBMS), making them a valuable tool for automating tasks, enforcing business rules, and maintaining data integrity. Here are some of the key advantages of using triggers:

- Automation:
  Triggers automate actions in response to specific events, reducing the need for manual intervention. This is especially valuable for routine or repetitive tasks, such as data validation, logging changes, or updating related records.
- Data Integrity:
  Triggers help enforce data integrity by ensuring that data modifications meet specific criteria or business rules. They can prevent the insertion, updating, or deletion of data that would violate constraints, thereby maintaining the quality and consistency of the data.
- Error Prevention:
  Triggers can help prevent errors by validating data before it is written to the database. If data doesn't meet the required criteria, the trigger can halt the operation and provide feedback to the user.
- Data Validation:
  Triggers can validate data consistency and accuracy, ensuring that data adheres to predefined standards. This is essential in applications where data quality is critical, such as financial systems.
- Business Rule Enforcement:
  Triggers enable the enforcement of complex business rules that go beyond standard database constraints. They allow you to implement application-specific logic and workflow requirements within the database.

In the context of database triggers, "trigger events" refer to the specific actions or events that can cause a trigger to be executed. The "granularity" of a trigger refers to the level at which the trigger operates, which may be at the row-level or statement-level. Trigger Events are the events that can trigger the execution of a database trigger depend on the DBMS and the capabilities it supports. Common trigger events include:

- Data Modification Events:
  - INSERT: Triggered when a new record is inserted into a table.
  - UPDATE: Triggered when existing records are updated.
  - DELETE: Triggered when records are deleted from a table.
  - MERGE: Triggered when a MERGE (or equivalent) statement is used to perform both INSERT and UPDATE operations.
- Database Events:
  - CREATE: Triggered when a new database object, such as a table or view, is created.
  - ALTER: Triggered when an existing database object is modified.
  - DROP: Triggered when a database object is deleted.
- Time-Based Events:
  - AFTER: Triggered after the associated event has occurred. For example, an "AFTER INSERT" trigger runs after a new record is inserted.
  - BEFORE: Triggered before the associated event occurs. For example, a "BEFORE UPDATE" trigger runs before an update operation.
- Login and Logout Events:
  - LOGON: Triggered when a user logs into the database.
  - LOGOFF: Triggered when a user logs out of the database.

The granularity of a trigger refers to whether it operates at the row-level or statement-level. This determines how many times a trigger is executed in response to an event:

- Row-Level Trigger:
  A row-level trigger is executed once for each row affected by the triggering event. For example, if an "AFTER INSERT" trigger is defined as a row-level trigger, it will be executed separately for each row inserted in a multi-row INSERT statement.
- Statement-Level Trigger:
  A statement-level trigger is executed once for each triggering event, regardless of the number of rows affected by that event. For example, an "AFTER INSERT" trigger defined as a statement-level trigger will be executed once for an entire multi-row INSERT statement.

Triggers are a powerful database feature, but there are situations where they should be used judiciously or avoided altogether. Here are some scenarios in which it may not be advisable to use triggers:

- Performance Impact:
  Triggers can have a significant impact on database performance, especially if they involve complex logic or operations. For high-transaction systems, using triggers extensively can lead to performance bottlenecks. Before implementing triggers, consider the potential performance implications and conduct thorough testing.
- Overly Complex Logic:
  Triggers with overly complex logic can be challenging to develop, maintain, and troubleshoot. If the trigger logic is convoluted and difficult to understand, it may lead to errors and make the database harder to maintain.
- Unnecessary Complexity:
  Sometimes, triggers are used to perform tasks that could be handled more simply and efficiently through application code. If a task can be accomplished without introducing triggers, it may be preferable to do so to keep the database schema less complex.
- Inconsistent Behaviour:
  Triggers can lead to inconsistent behaviour, especially when multiple triggers are defined for the same event or when triggers interact in unexpected ways. It's crucial to thoroughly test and document the behaviour of triggers to ensure consistent results.

**Stored Procedures**

Stored procedures are precompiled database programs that are stored and executed in a database management system (DBMS). They are a type of database object that contains one or more SQL statements or procedural code, and they are typically used to perform specific tasks or operations within a database. Stored procedures offer several advantages, including improved performance, security, and maintainability. Here are key features and benefits of stored procedures:

- Performance:
  Stored procedures are precompiled and stored in the database, which can lead to improved query performance. This is because the DBMS does not need to recompile the same SQL statements each time they are executed, reducing the overhead of query optimization.
- Security:
  Stored procedures can provide an additional layer of security by allowing controlled access to database operations. Users or applications can be granted

permission to execute specific stored procedures while being restricted from direct access to tables or views.

- Modularity:
  Stored procedures encapsulate SQL logic and procedural code in a modular way. This modularity enhances code organization, readability, and maintainability. Changes to the logic can be made in one place, and the changes will affect all invocations of the procedure.
- Abstraction:
  Stored procedures abstract the underlying data structure, allowing applications to work with the database using high-level procedures. This can simplify application code and reduce the need for complex SQL statements embedded in application code.
- Reusability:
  Stored procedures can be reused across multiple parts of an application or even across different applications. This reduces code duplication and promotes a consistent approach to database operations.
- Transaction Control:
  Stored procedures can manage transactions effectively. They allow for the grouping of multiple SQL statements within a single transaction, ensuring that a series of operations are either completed successfully or rolled back as a whole.
- Parameterization:
  Stored procedures can accept parameters, making them adaptable to different scenarios and allowing for dynamic queries. Parameters enable the customization of stored procedure behaviour.
- Encapsulation:
  Business logic and data access logic can be encapsulated within stored procedures, making it easier to maintain and update these logic components independently.
- Reduced Network Traffic:
  When applications execute stored procedures, only a small amount of data, such as parameter values, is transmitted over the network. This can reduce network traffic and improve application performance.
- Data Validation:
  Stored procedures can include validation checks and business rules, ensuring data integrity at the database level.
- Version Control:
  Storing procedures in the database allows for version control, which can be valuable for tracking and managing changes to database logic over time.
- Consistency:
  By using stored procedures, you can ensure that common tasks are performed consistently and that best practices are followed.

## Week 8

**Database Backed Software**

Database-backed software, also known as database-driven software or database-driven applications, refers to software applications that rely on a database system to store, manage, and retrieve data. Databases are an essential component of many software applications, from simple web applications to complex enterprise systems. Here are some key aspects of database-backed software:

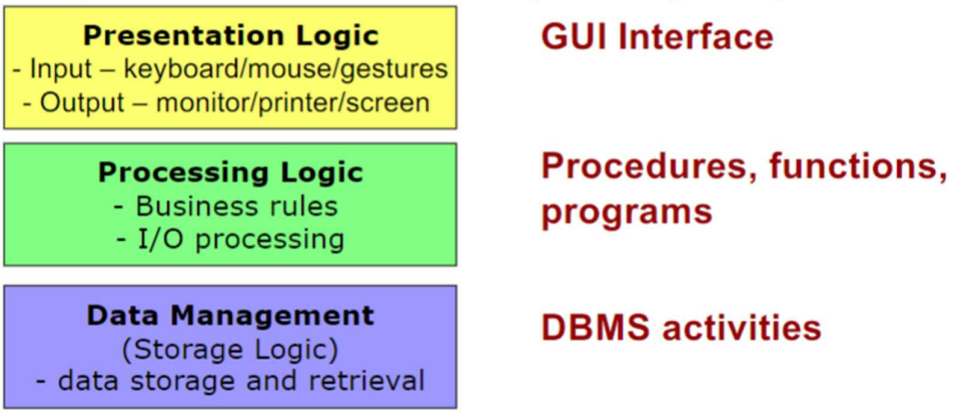
- Data Storage:
  In database-backed software, the application's data, such as user information, product details, transactions, and more, is stored in a structured format within a database. This allows for efficient and organized data management.
- Relational Databases:
  Most database-backed software applications use relational database management systems (RDBMS), such as MySQL, PostgreSQL, Microsoft SQL Server, or Oracle. These databases organize data into tables with rows and columns, which are related to each other through keys.
- Data Manipulation:
  The software interacts with the database using database query languages like SQL (Structured Query Language) to insert, update, retrieve, and delete data. This enables users and applications to interact with the data stored in the database.
- Dynamic Content:
  Database-backed software often generates dynamic content, such as web pages, reports, or application responses, by querying the database to retrieve the relevant data. This dynamic content can be personalized for users and can change in real-time based on user actions and system events.
- Data Integrity and Security:
  Databases provide mechanisms to ensure data integrity and security. Features like data validation, constraints, and access controls help maintain the accuracy and confidentiality of the data.

In software development, queries refer to requests made to a database to retrieve, manipulate, or modify data. These queries are typically written in a query language like SQL (Structured Query Language). Database queries are essential for interacting with and retrieving data from a database in a structured and efficient manner. SQL is the most common query language used to interact with relational databases. It allows you to perform various operations on the data, such as SELECT (retrieve data), INSERT (add data), UPDATE (modify data), and DELETE (remove data). Other types of databases may use different query languages. To work with queries and databases, you need database management system (DBMS) software. Popular DBMS software includes MySQL, PostgreSQL, Microsoft SQL Server, Oracle Database, and SQLite. These systems provide the infrastructure for storing and managing data and offer tools for executing queries. Software applications, such as web applications or desktop applications, connect to databases using specific libraries or drivers. These libraries allow the software to send queries to the database, retrieve results, and process data within the application.

**Data Intensive Systems**

Data-intensive systems, also known as data-intensive applications or data-driven systems, are software systems and architectures designed to handle and process large volumes of data. These systems are characterized by their ability to collect, store, process, and analyse significant amounts of data efficiently and reliably. Data-intensive systems are crucial in various domains, including e-commerce, social media, finance, healthcare, and more. The software architecture of data-intensive systems is a critical aspect of designing and building these complex applications. It plays a fundamental role in determining how data is collected, stored, processed, and presented to users.

There are 3 types of functionalities which are often placed in separate layers of code:




When considering the functionalities of a software application in terms of presentation logic, processing logic, and data management, you can break them down as follows:

- Presentation Logic Functionalities:

  The presentation logic layer is responsible for the user interface (UI) and the overall user experience (UX) of the software. It focuses on how the application presents information to users and how users interact with the system.

  - User Interface (UI):

    This refers to the presentation layer of the application, which includes the visual elements and interactions that users see and use.

    - Designing user-friendly interfaces.
    - Implementing responsive design for various devices.
    - Creating navigation and user flow.
    - Incorporating UI elements like forms, buttons, menus, and multimedia.
  - User Experience (UX):

    UX functionalities focus on ensuring a positive and efficient user experience.

    - Conducting usability testing and user feedback analysis.
    - Optimizing user interactions and feedback.
    - Implementing accessibility features for a diverse user base.
  - Front-End Development:

    Front-end functionalities involve translating design and user experience requirements into actual code and functionality.

    - Writing HTML, CSS, and JavaScript for the user interface.
    - Integrating with back-end services and APIs.
    - Handling user input validation and error handling.
  - Localization:

    If the application is used in multiple regions or languages, localization functionalities are needed.

    - Adapting the UI for different languages and cultures.
    - Managing content and UI translations.
    - Handling date, time, and currency formats specific to regions.
- Processing Logic Functionalities:

  The processing logic layer contains the core application logic, business rules, and server-side operations. It's responsible for handling data processing and user interactions.

  - Business Logic:

Business logic functionalities are responsible for the core operations and rules of the application.

- Implementing algorithms and calculations specific to the application's domain.
- Enforcing business rules and validations.
- Handling user transactions and data processing.

- Server-Side Logic:

  These functionalities deal with the back-end operations of the application.
  - Handling HTTP requests and responses.
  - Implementing server-side scripting (e.g., using PHP, Python, or Node.js).
  - Managing user authentication and authorization.
  - Integrating with databases and external services.
- Workflow Management:

  Workflow functionalities are about managing the sequence of tasks and processes within the application.
  - Defining and controlling task dependencies.
  - Managing state transitions.
  - Handling asynchronous and parallel processing.
- Error Handling and Logging:

  These functionalities are essential for managing and tracking issues in the application.
  - Implementing error handling routines.
  - Generating logs for debugging and auditing.
  - Alerting and notifications for critical errors.

- Data Management Functionalities:

  The data management layer deals with the storage, retrieval, and processing of data within the application. It ensures data is stored securely, efficiently, and is readily accessible.
  - Data Storage:

    Data storage functionalities involve managing the persistence of data within the application.
    - Choosing the appropriate database system (e.g., relational, NoSQL, in-memory).
    - Designing database schemas and data models.
    - Implementing data access and storage mechanisms.
  - Data Retrieval and Processing:

    These functionalities are about retrieving, manipulating, and presenting data.
    - Writing database queries and data retrieval methods.
    - Performing data validation and transformation.
    - Aggregating and analysing data for reporting or analytics.
  - Data Security and Privacy:

    Data security and privacy functionalities are crucial for protecting sensitive information.
    - Implementing access controls and permissions.
    - Encrypting data at rest and in transit.
    - Ensuring compliance with data protection regulations (e.g., GDPR).

- Data Backup and Recovery:

  Data backup and recovery functionalities are essential for data reliability and disaster recovery.
  - Setting up automated data backups.
  - Implementing data versioning or snapshotting.
  - Defining recovery procedures in case of data loss or system failures.

**SQL in Application Code**

SQL (Structured Query Language) is commonly used within application code to interact with databases. Applications often need to retrieve, insert, update, or delete data from a database, and SQL provides a standardized way to do this. SQL commands can be called from within a host language. These programs must include a statement to connect to the right database so that the SQL statements can refer to host variables. Statement-level interface and Call-level interface are two different approaches for interacting with databases, each serving distinct purposes in the context of database management systems. There are 2 main integration approaches:

- Statement Level Interface (SLI)

  The statement-level interface, often associated with database management systems, allows developers to work with SQL statements on an individual basis. It primarily deals with executing SQL statements one at a time. This method embeds SQL in the host language making the application program a mixture of host language statements and SQL. A special compiler is required to deal with both aspects.
- Call Level Interface (CLI)

  The call-level interface is often associated with more comprehensive and automated database interactions. It allows developers to call stored procedures or functions on the database server, encapsulating multiple SQL statements within a single call. This method calls for the creation of a special API to call SQL commands which are then passed as arguments to host language. This requires a standard programming language compiler, and the program is combined with a library that supports the API.

When code is executed within the context of a database management system (DBMS), such as SQL code in the case of SQL databases, the privileges and permissions under which the code runs depend on the user or role executing the code and the database system's security model. Many databases are accessed indirectly, and the program can do lots of checking to ascertain whether the access is appropriate or not before sending the SQL statements to the DBMS.

**Python DB-API2**

The Python DB API 2.0 (Python Database API Specification 2.0) is a standard interface for interacting with relational databases from Python programs. It provides a consistent way to connect to, query, and manage various database systems using Python code. Python DB API 2.0 is commonly used to work with databases such as MySQL, PostgreSQL, SQLite, Oracle, and others.

The API allows you to establish a connection to a database using a database-specific driver or module. Connections are typically created using connection objects provided by the DB API-compliant driver. Static and dynamic SQL are two approaches to constructing and executing SQL statements in the context of database operations. Each approach has its own advantages and use cases, and the choice between them depends on the specific requirements of your application. Static SQL refers to SQL statements that are hard coded within the

application source code and do not change during runtime. Dynamic SQL involves constructing SQL statements at runtime based on variables, user input, or changing criteria.

# Python Database API Interfaces

Connection Management

- **pg8000.connect()** connects to a database
- **conn.cursor()** creates a cursor object for query execution

Start SQL statements

- **curs.execute()** for static SQL, and also parameterized SQL queries
- **curs.callproc()** for executing a stored procedure including parameters

Result retrieval

- **curs.fetchone()** retrieves next row of a result or **None** when no more data
- **curs.fetchall()** retrieves the whole (remaining) result set, and returns it as a list of tuples

Transaction control

- **conn.commit()** successfully finishes (commits) current transaction
- **conn.rollback()** aborts current transaction

Error Handling

- Via standard exception handling of Python

# DB-API: Executing SQL Statements

Three different ways of executing SQL statements:

- ***cursor*.execute**(*sql*) semi-static SQL statements
- ***cursor*.execute**(*sql,params*) parameterized SQL statements
- ***cursor*.callproc**(*call,args*) invoke a stored procedure in DBMS
- ***cursor*.executemany**(*sql,seq_of_params*) repeatedly executes parameterized SQL statements

In DB-API 2.0,

- Need to create new cursor and re-issue SQL statement each time when parameters change – or if possible use **executemany()**
- Some other APIs offer "prepared statements" – parsed and optimized once in the dbms, then re-executed over and over with different parameters

Parameterized queries, also known as prepared statements or parameter binding, are a best practice in database programming for executing SQL statements in a way that helps prevent SQL injection and enhance security. Parameterized queries separate the SQL code from the data, allowing you to pass data as parameters, which are automatically sanitized by the database driver. This ensures that user input or data from external sources cannot be directly inserted into the SQL statement, reducing the risk of SQL injection attacks. Anonymous and named parameterized queries are two approaches to using parameterized queries in database

programming. Both types provide security benefits by separating SQL code from data, preventing SQL injection, and allowing for safe parameter binding. Here's an explanation of both types:

- Anonymous Parameterized Queries

  In anonymous parameterized queries, placeholders for parameters are indicated using positional markers, such as question marks (?), in the SQL statement. The order in which you bind parameters to these placeholders must correspond to the order of the placeholders in the SQL statement.
- Named Parameterized

  In named parameterized queries, placeholders for parameters are named using a specific format (e.g., :name, :email, or $param_name). The order of parameter binding does not matter because parameters are referenced by name.

Host variables, also known as bind variables, are placeholders within SQL statements used to bind values from the application code to the SQL query. These variables are typically used when writing dynamic SQL. Host variables are a way to pass data from the application to the database system and can be used in combination with parameterized SQL to create flexible queries.

A "buffer mismatch problem" typically refers to a situation in software development where there is a discrepancy or inconsistency between data that is being read from or written to a buffer or memory region. This problem can manifest in various ways and can lead to issues such as data corruption, crashes, or security vulnerabilities. Error handling is an important aspect of working with Python's Database API (DB API) when interacting with databases. Handling errors gracefully can help you detect and respond to issues that may arise during database operations. The DB API specifies a set of exceptions that you can catch and handle when working with databases.

**Stored Procedures**

Stored procedures are database objects that contain one or more SQL statements or commands. They are precompiled database objects that encapsulate a series of SQL statements, allowing you to execute those statements as a single unit. Stored procedures are typically stored and managed within a database and can be invoked by applications, triggers, and other stored procedures. Stored procedures are a powerful tool for encapsulating server-side application logic within a database. They allow you to centralize data-related operations and business rules in the database, reducing the need for complex application code and promoting a modular, organized approach. The advantages are given below:

Advantages:

- Central code-base for all applications
- Improved maintainability
- Additional abstraction layer
  (programmers do not need to know the schema)
- Reduced data transfer
- Less long-held locks
- DBMS-centric security and consistent logging/auditing (important!)

Stored procedures are supported by various relational database management systems (RDBMS), but the specific language or dialect used to create stored procedures can vary

between database systems. Each RDBMS may have its own language or procedural extension for defining and implementing stored procedures.

**Security Threats with DB backed applications.**

Database-backed applications are vulnerable to various security threats, and it's crucial to address these threats to protect sensitive data and ensure the integrity and availability of your application. Here are some common security threats associated with DB-backed applications:

- SQL Injection (SQLi):
  SQL injection is one of the most prevalent threats. It occurs when an attacker injects malicious SQL code into input fields or URL parameters, manipulating the database query. This can lead to unauthorized data access, data manipulation, or even database compromise.
- Cross-Site Scripting (XSS):
  Cross-Site Scripting attacks involve injecting malicious scripts into web application output, which are then executed by users' browsers. These scripts can steal data, hijack sessions, or perform other malicious actions.
- Inadequate Authentication and Authorization:
  Weak or improperly configured authentication and authorization mechanisms can allow unauthorized users to access sensitive data or perform actions they should not be allowed to do.
- Insecure Direct Object References (IDOR):
  IDOR occurs when an attacker can manipulate object references in URLs or forms to access data they are not authorized to access. Proper authorization checks are crucial to mitigate this threat.
- Security Misconfigurations:
  Improperly configured databases, web servers, and application servers can lead to security vulnerabilities. Ensuring that these components are configured securely and patched is essential.
- Sensitive Data Exposure:
  Exposing sensitive data, such as passwords, credit card information, or personal identifiers, can have severe consequences. Proper data encryption, access controls, and data masking are necessary to protect this information.
- Persistent XSS
  Persistent Cross-Site Scripting (XSS), also known as stored XSS, is a type of web security vulnerability where an attacker injects malicious scripts (usually JavaScript) into a web application. These injected scripts are then permanently stored on the target server and served to other users when they access a specific page or resource. Persistent XSS attacks can have serious consequences, as they allow attackers to execute arbitrary code within the context of a victim's web browser.

To mitigate these security threats, it's essential to follow best practices in secure coding, regularly update and patch software components, and conduct security assessments, including penetration testing and vulnerability scanning. Additionally, using a Web Application Firewall (WAF) and employing strong access controls can help protect against many of these threats. Security is an ongoing process, and it's critical to stay vigilant and up to date with the latest security practices.

Protecting a web database is critical to ensuring the confidentiality, integrity, and availability of your data. Databases often store sensitive information, and a breach can have serious consequences. Here are key steps and best practices to protect a web database:

- Authentication and Authorization:
  Implement strong authentication mechanisms to ensure that only authorized users can access the database. Enforce role-based access control (RBAC) to grant specific privileges to users and limit what actions they can perform.
- Secure Connection (HTTPS):
  Ensure that all data transmission between the web application and the database is encrypted using HTTPS. This prevents eavesdropping on data in transit.
- Strong Password Policies:
  Enforce strong password policies, including password complexity and rotation. Encourage users to choose secure passwords.
- SQL Injection Prevention:
  Use parameterized queries or prepared statements to prevent SQL injection attacks. Avoid dynamic SQL construction with user inputs.
- Firewall Rules:
  Configure firewall rules to restrict access to the database server. Whitelist allowed traffic and block everything else.
- Security Patching:
  Keep the database software, web application framework, and server operating system up to date with security patches. Vulnerabilities can be exploited if not patched.
- Data Encryption:
  Encrypt sensitive data at rest using database encryption features or third-party encryption solutions. Protect database backups as well.
- Web Application Firewall (WAF):
  Deploy a WAF to protect against various application-level attacks, including SQL injection, XSS, and other vulnerabilities.

## Week 10

**Query Processing**

Query processing is a fundamental operation in database management systems (DBMS) and refers to the steps involved in executing a database query. The process encompasses various stages that transform a user's query into results from the database. Here are the key components and steps involved in query processing:

- Query Parsing:
  The first step is to parse the user's query to understand its structure and intent. This involves breaking down the query into its constituent parts, such as keywords, table names, conditions, and expressions.
- Query Optimization:
  After parsing, the DBMS's query optimizer analyses different execution plans for the query and selects the most efficient one. This step aims to minimize the use of system resources and deliver query results quickly.
- Query Rewriting:
  In some cases, the query optimizer may rewrite the original query to improve its performance. This can involve reordering operations or selecting different access paths.
- Access Path Selection:
  The DBMS must determine how to access the data needed to fulfill the query. It selects access methods, such as full table scans, index scans, or join methods, based on the query and the available database structures.

- Data Retrieval:
  The selected access paths are executed to retrieve the necessary data from the database. This may involve reading from disk or memory, depending on the location of the data.
- Join Processing:
  If the query involves joining multiple tables, the DBMS performs the join operation. Different algorithms, like nested loop joins or hash joins, may be used to combine the data.
- Filtering and Sorting:
  Data retrieved from the tables is filtered based on the query conditions and sorted if the query specifies an order for the results.
- Aggregation and Grouping:
  If the query includes aggregation functions (e.g., SUM, AVG) or grouping, the DBMS performs these operations to produce summary data.
- Result Generation:
  The final query result set is generated based on the filtering, sorting, and aggregation operations performed.
- Result Presentation:
  The query results are presented in a format suitable for consumption by the user or application. This can include tabular results, reports, or structured data.
- Error Handling:
  The DBMS handles any errors or exceptions that may occur during query processing. This includes issues like data not found, constraint violations, and database connectivity problems.
- Query Execution Plan:
  The execution plan, which represents the specific steps taken during query processing, is generated, and often cached for future use. This plan provides a roadmap for how the query is executed.
- Resource Management:
  Throughout query processing, the DBMS must manage resources like CPU, memory, and disk access efficiently to ensure that the query doesn't overwhelm the system.

**DBMS Computations**

In a Database Management System (DBMS), computations often refer to the ability to perform various operations or calculations on the data stored in a database. These computations can range from basic arithmetic calculations to more complex data manipulations and transformations. Here are some common types of computations performed within a DBMS:

- Arithmetic Computations:
  Basic arithmetic computations, such as addition, subtraction, multiplication, and division, can be performed on numeric data stored in the database. For example, you can calculate the total sales, average scores, or tax amounts.
- Aggregation:
  Aggregation functions like SUM, AVG, COUNT, MAX, and MIN are used to compute summary statistics or aggregate data across a set of records. These are often used in SQL queries to summarize data.
- String Manipulation:
  DBMSs provide functions for string manipulation, allowing you to concatenate strings, extract substrings, change case, and perform other text-related operations.

- Date and Time Computations:
  You can perform calculations involving date and time data, such as determining the time difference between two dates, extracting components (e.g., year, month, day), and formatting dates.
- Conditional Computations:
  Conditional expressions and computations allow you to perform calculations based on specific conditions. For example, you can calculate discounts based on purchase quantities or apply different tax rates based on the location of the transaction.
- Joins and Set Operations:
  DBMSs support join operations to combine data from multiple tables based on related columns. You can perform inner joins, outer joins, and other types of set operations to compute new datasets.
- Window Functions:
  Window functions allow you to perform calculations over a specific "window" of data, typically within a result set. Common window functions include RANK, DENSE_RANK, and LAG/LEAD.
- User-Defined Functions:
  Many DBMSs allow users to define custom functions (stored procedures or user-defined functions) to perform complex computations or transformations on data.
- Mathematical Computations:
  Mathematical functions and expressions can be used for more advanced calculations, including trigonometric functions, logarithms, and exponentiation.
- Statistical Computations:
  DBMSs often provide statistical functions for performing various statistical calculations, such as standard deviation, variance, and regression analysis.
- Data Transformation:
  You can transform data using various operations like pivoting, unpivoting, and transposing to reshape the data for reporting or analysis.
- Complex Queries:
  SQL queries can be used to perform computations by combining multiple operations and functions. For example, you can calculate sales totals for specific products, categories, or time periods using SQL queries.
- Data Mining and Analysis:
  DBMSs may include data mining and analytical capabilities, allowing you to perform more advanced computations like clustering, classification, and predictive modelling.
- GIS Computations:
  Geographic Information Systems (GIS) databases allow for computations related to spatial data, such as distance calculations, area measurements, and geospatial analysis.

**Data Storage**

CPU operations use data in a few registers, which are themselves loaded/stored from the program memory which is held in DRAM which is volatile by nature. Meaning, that if the computer restarts, any data stored on the DRAM will be lost. Data storage devices are hardware components or media used for storing and retaining digital data. These devices come in various forms, each with different characteristics, capacities, and use cases. Here are some of the most common types of data storage devices:

- Hard Disk Drives (HDD):
  Hard disk drives use spinning platters to store data magnetically. They are known for their relatively large storage capacities and are commonly used in desktop and laptop computers, servers, and data centres.
- Solid-State Drives (SSD):
  Solid-state drives store data on flash memory chips, offering faster data access and improved durability compared to HDDs. SSDs are commonly found in laptops, desktops, and high-performance computing devices.
- USB Flash Drives:
  USB flash drives, also known as thumb drives or pen drives, use NAND flash memory to store data. They are portable, plug-and-play devices used for data transfer and backup.
- Memory Cards:
  Memory cards are used primarily in digital cameras, smartphones, and other portable devices. They come in various formats, such as SD, microSD, and CompactFlash, with different capacities.
- External Hard Drives:
  External hard drives are portable HDD or SSD storage devices connected to a computer or other devices via USB, Thunderbolt, or eSATA. They are used for data backup and additional storage.
- Cloud Storage:
  Cloud storage refers to data storage services provided by cloud providers like Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure. Users store and access data over the internet, often using web-based interfaces or APIs.
- RAM (Random Access Memory):
  RAM is volatile memory used by computers to store data temporarily while the system is running. It is used for active processes, and data is lost when the system is powered off.
- In-Memory Databases:
  In-memory databases store and manage data entirely in RAM, providing exceptionally fast data access. They are used in high-performance database applications.

**How to store a Database**

A database is a collection of relations. Each relation is a set of records (or tuples). A record is a sequence of fields (or attributes). Alternatively, there are different file organizations as well. A few of these alternatives are:

- Heap Files
  Heap files are a type of file organization used in database management systems (DBMS) to store records or data in an unordered or unsorted manner. In a heap file, records are inserted sequentially without regard for any specific order or structure. This means that new records are simply appended to the end of the file without any sorting or organization based on the values of the records.

## (Unordered) Heap Files

Simplest file structure contains records in no particular order.

- No way to find a record of interest, except to look through all the records one after another, until you find the one you want

Rows appended to end of file as they are inserted

- Hence the file stays unordered
- Insertion is quick

Deleted rows create gaps in file

- File must be periodically compacted to recover space

## Access to Heap Files

How to find records in a heap file, to answer a query?

Eg SELECT <some columns> FROM table WHERE <condition>

Access method is a ***linear scan with filter*** (also called: table scan)

- Examine each block in turn (from buffer, or fill buffer from disk if necessary)
- Within the block, look at each record in turn, and check whether <condition> is true
  - If so, output the contents of the appropriate columns

This is expensive!

- Usually, the whole file must be processed
- If you can stop once you have found what you are looking for (eg you know there is only one matching record), then on average half of the pages in a file must be read,
- This is as efficient as possible if all rows are returned (SELECT * FROM *table*)
- Very inefficient (relative to what is really needed) if a *few* rows are wanted

- Sorted Files

Sorted files, also known as ordered files, are a type of file organization used in database management systems (DBMS) and data storage systems to store records or data in a specific order based on the values of one or more fields within the records. The primary characteristic of sorted files is that records are arranged in a specific order to facilitate efficient retrieval and searching.

Rows are sorted based on some attribute

- Successive rows are stored in same (or successive) pages
- This makes it fairly fast to get to records with a particular value for the sorting attribute
  - Or for records where sorting attribute lies in some range
- The values of the other attributes are not arranged in any particular way

One problem: Maintaining sorted order

- After the correct position for an insert has been determined, shifting of subsequent tuples necessary to make space (very expensive)

## Access to Sorted File

How to find records in a heap file, to answer a query?

Eg SELECT <some columns> FROM table WHERE <condition>

First, suppose that <condition> determines the sorting attribute (either exactly, or within a range)

- Eg suppose sorting attribute is StudentID, condition is WHERE semester = 'S1998' AND StudentID = 123456

Access method could be a ***binary search*** **(details in comp2123)**

- In logarithmic time (much faster than linear), get to first row that has appropriate value for the sorting attribute
- Then check each successive record until sorting attribute is no longer suitable, to see if the rest of <condition> also is true
  - If so, output the contents of the appropriate columns

## Access to Sorted File

How to find records in a heap file, to answer a query?

Eg SELECT <some columns> FROM table WHERE <condition>

Now, suppose that <condition> does NOT determine the sorting attribute (either exactly, or within a range)

- Eg suppose sorting attribute is StudentID, condition is WHERE semester = 'S1998' AND gpa >= 3.0

Access method needs to be linear scan-and-filter

- Look at all the rows, one after another
- (just as if data is stored as Heap, as far as condition is concerned!)
- Very inefficient (relative to what is really needed) if a *few* rows are wanted

- Indexes

Indexes are data structures used in database management systems (DBMS) to improve the speed and efficiency of data retrieval operations from a database table. Indexes are like the indexes found at the end of a book—they provide a way to quickly locate specific information (i.e., rows or records) within the database. In SQL, an index is a database object used to improve the speed and efficiency of data retrieval operations, such as SELECT queries. Indexes are created on one or more columns of a database table to allow the database management system (DBMS) to quickly locate specific rows or records in the table without having to perform a full table scan.

You can create an index on one or more columns of a table using SQL statements like CREATE INDEX. Indexes are usually created after the table is created, but some DBMSs allow for the creation of indexes when defining a table. Given below are the advantages and disadvantages of Indices:

- Advantages
  - Improved Query Performance:
    Indexes significantly speed up data retrieval operations, especially SELECT statements, by reducing the amount of data that needs to be scanned. This results in faster query execution.
  - Faster Joins:
    Indexes are particularly valuable when joining multiple tables. They allow the database to quickly identify and match rows from different tables, improving query performance.
  - Efficient WHERE Clause Filtering:
    Queries that filter rows based on specific conditions (e.g., WHERE clauses) benefit from indexes. Indexes help narrow down the search to only the relevant rows.
  - Optimized ORDER BY:
    Indexes make it faster to retrieve data in a specific order, improving the performance of queries that use the ORDER BY clause.
  - Facilitate Range Queries:
    Indexes are useful for retrieving data within a specified range of values, such as date ranges, price ranges, or alphabetical ranges.
  - Consistency:
    Indexes help maintain the order and consistency of data, ensuring that queries produce reliable results.
  - Support for Unique Constraints:
    Indexes can enforce uniqueness constraints on columns, ensuring that no two rows in the indexed column(s) have the same value.
  - Query Optimization:
    The query optimizer uses indexes to generate efficient query execution plans, saving time and resources.
- Disadvantages
  - Storage Overhead:
    Indexes consume additional storage space. The more indexes you create, the more storage is required, which can be a concern for large databases.
  - Write Performance:
    Indexes can slow down write operations, such as INSERT, UPDATE, and DELETE, as the indexes need to be updated to reflect changes in the data.
  - Maintenance Overhead:
    Indexes must be maintained as data changes. Frequent data updates or insertions may increase the overhead of index maintenance.
  - Complexity:
    Managing a large number of indexes can become complex. Careful planning and monitoring are required to avoid excessive indexing.
  - Choice of Wrong Columns:
    Indexing the wrong columns can lead to suboptimal query performance. Poorly chosen indexes may not benefit query execution or may even hinder it.

- Performance Trade-offs:

  While indexes improve read performance, they may slow down write operations. There is a trade-off between read and write performance that database administrators must carefully consider.
- Space Considerations:

  Some DBMSs have limitations on the number and size of indexes that can be created. Creating too many indexes can reach these limits.
- Outdated or Unused Indexes:

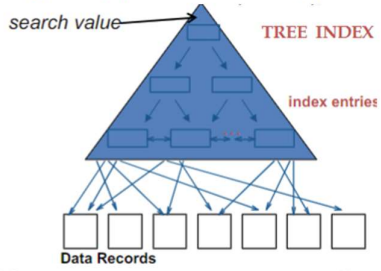
  Over time, indexes that were once useful may become outdated or no longer used by queries. It's essential to periodically review and potentially remove unused or unnecessary indexes.

**Indexes**

In a database management system (DBMS), various types of indexes can be used to optimize data retrieval and query performance. The choice of index type depends on the specific use case, database system, and query patterns. Here are some common types of Indexes:

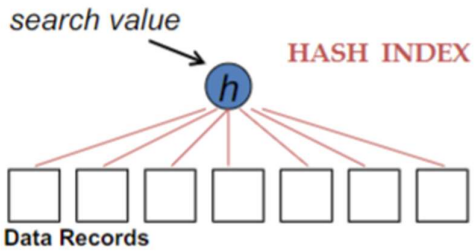
- B-Tree Index:

  B-Tree (Balanced Tree) indexes are the most common type of index in relational database systems. They organize data in a balanced tree structure that allows for efficient range queries, equality searches, and sorting. B-Tree indexes are well-suited for columns with high cardinality (many distinct values).
  - *Very flexible, supports point queries (equality to a specific value), range queries and prefix searches*
  - *Index entries are stored in sorted order by search key (with a complex arrangement for access that looks at very few blocks)*
  - *Found in every DBMS platform*
  - This is default index in PostgreSQL (what is done by plain CREATE INDEX statement)



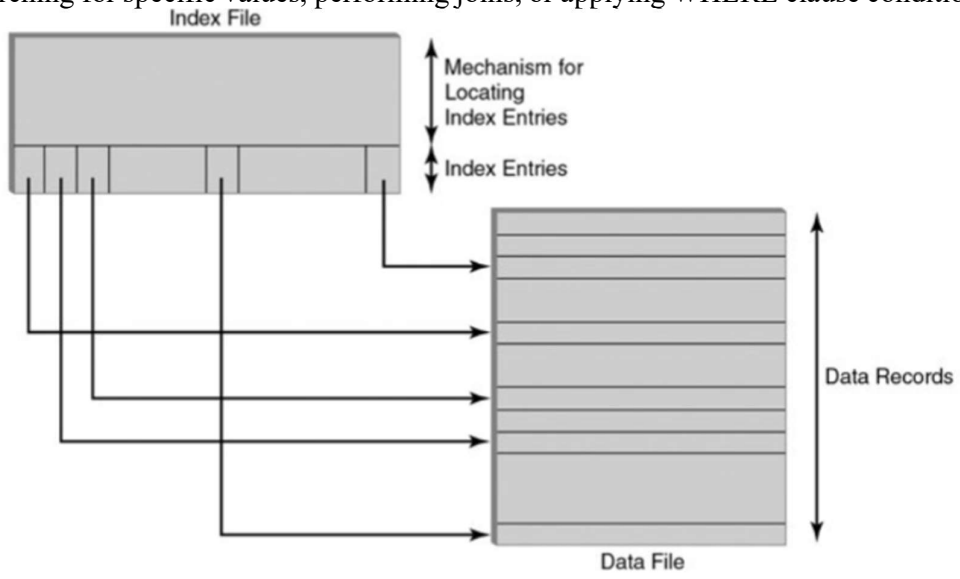

- Hash Index:

  Hash indexes are designed for fast equality searches, making them suitable for columns where exact matches are the primary use case. Hash indexes use a hash function to map keys to specific locations, providing constant-time access.
  - *Fast for point (equality) searches – but not fast for other calculations*
  - PostgreSQL syntax: CREATE INDEX ***indexname*** ON ***table*** USING HASH (***column***);



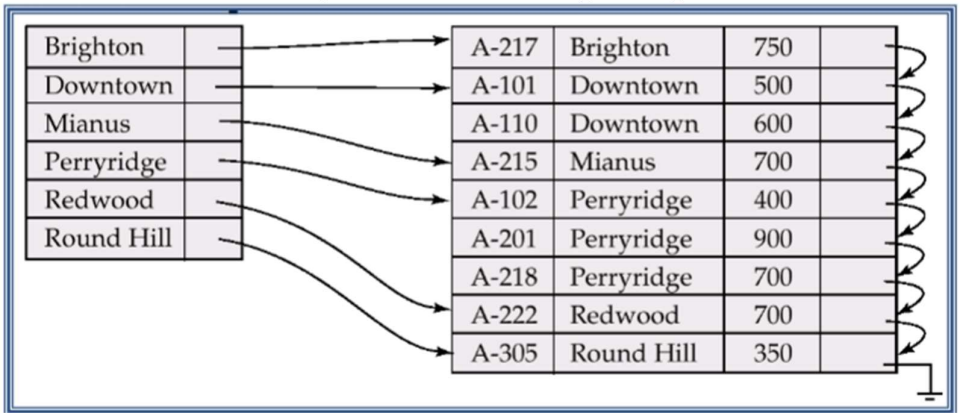
- Unclustered Index:

  A non-clustered index is a separate data structure that contains a copy of the indexed columns' values and a pointer to the actual data rows in the table. Unlike clustered indexes, you can create multiple non-clustered indexes on a single table. Non-clustered indexes are useful for optimizing the performance of SELECT queries that involve columns other than the primary key. They are particularly beneficial when searching for specific values, performing joins, or applying WHERE clause conditions.




- Clustered Index

  A clustered index determines the physical order of the data rows in a table. In other words, the data rows in the table are stored on disk in the same order as the index key. Each table can have only one clustered index. This index type is typically used for the primary key of a table. When you create a clustered index, you are essentially reorganizing the entire table's data to match the order of the index key. Queries that involve the clustered index key can benefit from improved performance.



# Index Classifications

## Unique vs. Non-Unique

– an index over a candidate key is called a **unique index** (no duplicates)

## Single-Attribute vs. Multi-Attribute

– whether the search key has one or multiple fields

## Clustering vs. Unclustered

– If data records and index entries are ordered the same way, then called **clustering index**.

Choosing the right indexes for your database is a critical aspect of database performance optimization. Indexes can significantly speed up query execution, but they come with trade-offs in terms of storage and maintenance. Index selection is a critical aspect of database optimization, and the right choice of indexes can significantly improve query performance. Here are some guidelines for selecting indexes in a relational database:

- Understand Query Patterns:
  Start by understanding the types of queries your application regularly performs. Identify the columns used in WHERE clauses, JOIN operations, and ORDER BY clauses.
- Prioritize High-Selectivity Columns:
  Columns with high selectivity, which have a large number of distinct values, are strong candidates for indexing. These indexes can narrow down the search quickly.
- Primary Key Index:
  By default, most databases create a clustered index on the primary key column. If your application frequently queries by the primary key, this index is crucial for performance.
- Foreign Key Indexes:
  Columns used as foreign keys in JOIN operations should typically be indexed. This improves query performance when joining tables.
- Columns in WHERE Clauses:
  Index columns used in WHERE clauses, especially those with equality conditions (e.g., =, IN), to accelerate data retrieval.
- Avoid Over-Indexing:
  Be mindful of creating too many indexes. Over-indexing can increase storage requirements and degrade performance for write operations.
- Composite Indexes:
  For queries that filter or join based on multiple columns, consider creating composite indexes (indexes on multiple columns). Composite indexes can be highly effective in optimizing queries.

- Covering Indexes:
  Consider creating covering indexes that include all columns required for a query. This can eliminate the need to access the table data, resulting in faster query performance.
- Regularly Review Indexes:
  Periodically review the effectiveness of existing indexes. Remove unused or redundant indexes that do not contribute to query performance.
- Monitor Performance:
  Use database performance monitoring tools to track query execution and index usage. Identify slow-performing queries and examine their execution plans.
- Test and Profiling:
  Test and profile query performance with and without indexes to identify areas for improvement. Use query execution plans to analyze index usage.
- Consider Index Types:
  Choose the appropriate index type (e.g., B-Tree, hash, bitmap) based on the specific use case and query requirements. Different index types have varying strengths and weaknesses.
- Index Maintenance:
  Be aware that indexes require maintenance as data changes. Frequent data updates or insertions can impact index performance. Monitor and optimize index maintenance.
- Database Engine Capabilities:
  Familiarize yourself with the indexing capabilities and limitations of your specific database system. Different databases may have unique features and behaviours related to indexing.
- Think About the Future:
  Consider how your data and query patterns may change over time. Choose indexes that are flexible enough to accommodate potential future requirements.

## Week 11

**ACID Properties**

The ACID properties are a set of characteristics that guarantee the reliability and consistency of transactions in a database management system (DBMS). These properties ensure that database transactions are processed in a way that maintains data integrity, even in the presence of system failures. ACID is an acronym for the following four properties:

- Atomicity:
  Atomicity guarantees that a transaction is treated as a single, indivisible unit of work. It means that either all the changes made by a transaction are committed to the database or none of them are. If a transaction is interrupted (e.g., due to a system crash), any changes made by that transaction are rolled back to their previous state.
- Consistency:
  Consistency ensures that a transaction brings the database from one consistent state to another. It enforces data integrity rules, constraints, and relationships defined in the database schema. If a transaction violates any of these rules, the entire transaction is rolled back, and the database remains in a consistent state.
- Isolation:
  Isolation guarantees that the operations of one transaction are isolated from the operations of other concurrent transactions. Each transaction should operate as if it's the only one being executed, preventing interference or conflicts with other

transactions. Isolation levels, such as Read Uncommitted, Read Committed, Repeatable Read, and Serializable, define the degree of isolation in a DBMS.

- Durability:

  Durability ensures that once a transaction is committed, its changes are permanent and will survive system failures, including power outages or crashes. These changes are stored in non-volatile storage (e.g., disk) and are not lost even if the system restarts. Durability is typically achieved through techniques like write-ahead logging and data synchronization.

These ACID properties collectively provide a strong guarantee of data integrity and reliability in a DBMS, making it suitable for applications where the accuracy and consistency of data are critical, such as financial systems, inventory management, and e-commerce platforms.

**Transactions**

In a database management system (DBMS), a transaction is a fundamental concept that represents a single unit of work or a sequence of operations that need to be executed atomically, ensuring the database remains in a consistent state before and after the transaction. Transactions play a crucial role in maintaining data integrity and consistency in the database. To code a transaction in a database using SQL, you'll typically use a programming language like Python, Java, or a similar language that interacts with the database through a library or driver. In transactions, a few control statements are:

- BEGIN (or START TRANSACTION): Initiates a new transaction.
- COMMIT: Commits the current transaction, making the changes permanent.
- ROLLBACK: Rolls back (undoes) the changes made during the current transaction.
- SAVEPOINT: Establishes a save point within a transaction, allowing you to roll back to that point if needed.
- ROLLBACK TO: Rolls back to a specified save point within the transaction.
- RELEASE SAVEPOINT: Releases a save point, making it permanent within the transaction.

The specific API for managing transactions depends on the database system and the programming language you are using. Different databases and programming languages provide their own libraries and functions for working with transactions. Ensuring the integrity of transactions is a fundamental goal of database management systems (DBMS). Database transactions must meet the ACID properties (Atomicity, Consistency, Isolation, Durability) to guarantee their integrity. Transaction consistency, as one of the ACID (Atomicity, Consistency, Isolation, Durability) properties, ensures that a database transaction brings the database from one consistent state to another. It enforces data integrity rules, constraints, and relationships defined in the database schema, preserving the correctness and reliability of the data. Consistent transactions:

- Enforcing Data Integrity Rules:

  Transaction consistency ensures that the changes made by a transaction adhere to the data integrity rules defined in the database schema. These rules can include constraints (e.g., primary key, unique key, foreign key constraints), data types, and validation rules.

- Maintaining Referential Integrity:

  In databases with multiple tables, consistency requires maintaining referential integrity. For example, if a foreign key constraint specifies that each order must have a

valid customer, a transaction should not be able to insert an order without a corresponding customer record.

- Preserving Business Rules:
  Consistency also involves preserving business-specific rules and logic. For instance, in an e-commerce application, a transaction to process an order should ensure that the product's quantity is reduced from the inventory, and the customer's payment is processed correctly.
- No Partial Updates:
  A consistent transaction should not result in partial updates. In other words, it's an all-or-nothing operation. If any part of the transaction fails, the entire transaction is rolled back, and the database returns to its previous state to maintain consistency.
- Conflict Resolution:
  In a multi-user environment, consistency addresses potential conflicts between transactions. If two transactions attempt to update the same data simultaneously, the DBMS must ensure that these transactions do not interfere with each other, thus maintaining consistency.
- Constraint Violations:
  If a transaction violates any data integrity constraints or rules during its execution, the DBMS should automatically roll back the transaction to prevent inconsistent data from being stored in the database.
- Constraints and Triggers:
  Database designers often use constraints (e.g., CHECK constraints) and triggers to enforce consistency. Constraints prevent invalid data from being inserted or updated, and triggers can automatically enforce additional business logic when data changes occur.
- Consistency Checks:
  Database administrators can periodically perform consistency checks or use automated tools to ensure that data remains consistent and conforms to the defined rules and constraints.

Transaction concurrency is a crucial aspect of database management that deals with how multiple transactions can execute simultaneously while maintaining data consistency and integrity. Managing concurrency is essential in multi-user database systems to prevent data anomalies and conflicts. Anomalies in transactions refer to unexpected or undesirable behaviours that can occur when multiple transactions access and modify the same data concurrently. These anomalies can lead to data inconsistencies and integrity issues. Some famous anomalies in transactions include:

- Dirty Read:
  A dirty read occurs when one transaction reads data that has been modified by another transaction but has not been committed yet. If the second transaction is rolled back, the data read by the first transaction becomes invalid.
- Non-Repeatable Read:
  Non-repeatable reads happen when a transaction reads the same data multiple times, but the data changes between reads due to other transactions. This inconsistency can lead to incorrect results or decisions based on the changing data.
- Phantom Read:
  Phantom reads occur when a transaction reads a set of rows that satisfy a condition, but another transaction inserts, updates, or deletes rows that would have met the same condition. This can lead to unexpected or inconsistent query results.

- Lost Update:

  A lost update anomaly happens when two transactions attempt to update the same data concurrently. The second transaction may overwrite the changes made by the first transaction, resulting in data loss and an incomplete update.
- Inconsistent Retrieval:

  This occurs when a transaction retrieves data based on some criteria, and while it processes the data, another transaction modifies it, causing the retrieved data to be inconsistent with the current state of the database.

**Serializability**

Serializability is a concept in database management that ensures that the execution of concurrent transactions produces results that are equivalent to some serial execution of those transactions. In other words, it guarantees that the database will behave as if all the transactions were executed one after the other in some order, even though they may run concurrently. Serializability is a key property of transactions and helps maintain data consistency and integrity in a multi-user database system. While serializability is a desirable property in database systems for ensuring data consistency and integrity, it comes with certain challenges and potential problems, particularly in terms of performance and scalability. Some of the issues and problems associated with serializability include:

- Reduced Concurrency:

  Achieving full serializability often results in reduced concurrency. In a highly concurrent environment, transactions may be forced to wait for locks or resources, leading to performance bottlenecks and slower response times.
- Deadlocks:

  To ensure serializability, database systems may use locking mechanisms, which can lead to the occurrence of deadlocks. Deadlocks happen when two or more transactions are waiting for each other to release locks, causing a standstill in transaction processing.
- Locking Overhead:

  Lock-based concurrency control mechanisms can introduce overhead due to the management of locks. Lock contention and lock acquisition can result in additional processing time and resource consumption.
- Limited Scalability:

  Full serializability can limit the scalability of a database system, as it may not effectively handle many concurrent users or transactions.
- Complexity and Maintenance:

  Implementing and maintaining serializability in a database system can be complex, especially in scenarios where multiple transactions interact with a variety of data objects and dependencies.
- Performance Trade-Off:

  Balancing the need for serializability with performance requirements can be challenging. In some cases, the choice of isolation levels other than Serializable, such as Read Committed or Repeatable Read, may be preferred to improve performance.
- High Resource Utilization:

  Achieving full serializability may lead to high resource utilization, including memory and CPU usage, due to the need to track and manage transactions and their interactions.

- Isolation Levels:

  Not all transactions require the highest level of serializability. In some cases, a lower isolation level may be sufficient to meet application requirements without incurring the performance costs associated with full serializability.

## Week 12

**Data Management in Enterprises**

Data management in enterprises refers to the systematic and strategic handling of data throughout its lifecycle, from creation and acquisition to storage, processing, analysis, and eventual disposal. Effective data management is crucial for organizations to derive valuable insights, maintain data integrity, ensure regulatory compliance, and support decision-making. Here are key aspects and best practices of data management in enterprises:

- Data Governance:

  Establish data governance policies, roles, and responsibilities to ensure data quality, security, and compliance with regulatory requirements. A data governance framework defines how data is managed, accessed, and protected within the organization.
- Data Strategy:

  Develop a comprehensive data strategy aligned with business objectives. This strategy outlines how data will be collected, stored, processed, and utilized to drive business value.
- Data Architecture:

  Design a data architecture that defines the structure of data within the organization, including databases, data warehouses, data lakes, and integration points. Consider both relational and NoSQL databases based on the data's nature.
- Data Security:

  Protect sensitive data through encryption, access controls, authentication, and authorization mechanisms. Implement security measures to safeguard data against unauthorized access, breaches, and cyber threats.
- Data Integration:

  Establish data integration processes to combine and unify data from various sources across the organization. Data integration tools and ETL (Extract, Transform, Load) processes are commonly used to facilitate data movement.
- Data Analytics and Business Intelligence:

  Leverage data for analytics, reporting, and business intelligence purposes. Implement data analytics tools, dashboards, and reporting systems to gain insights and make data-driven decisions.
- Data Privacy and Compliance:

  Comply with data privacy regulations (e.g., GDPR, HIPAA) by implementing policies and processes for data protection, consent management, and data subject rights.

Data analysis is a critical component of enterprise operations, enabling organizations to extract valuable insights from their data, make informed decisions, and gain a competitive edge. Data analysis in enterprises involves the systematic process of examining, cleaning, transforming, and interpreting data to draw meaningful conclusions. Here are key aspects and best practices for data analysis in enterprises:

- Data Collection and Integration:
  Gather data from various sources, including internal databases, external data providers, IoT devices, sensors, social media, and more. Implement data integration processes to consolidate and harmonize data for analysis.
- Data Preparation and Cleaning:
  Clean and preprocess data to address issues like missing values, duplicates, outliers, and inconsistencies. Data cleansing is essential to ensure the quality and reliability of the data.
- Exploratory Data Analysis (EDA):
  Conduct EDA to gain an initial understanding of the data's characteristics. Visualizations, summary statistics, and data profiling help identify patterns and relationships.
- Data Modelling and Analysis:
  Apply statistical, machine learning, and analytical techniques to explore data relationships, make predictions, and discover insights. Regression analysis, clustering, classification, and time series analysis are examples of modeling techniques.
- Hypothesis Testing:
  Formulate hypotheses and perform statistical tests to validate or reject them based on data evidence. Hypothesis testing helps draw conclusions and make data-driven decisions.
- Data Visualization:
  Create data visualizations, such as charts, graphs, and dashboards, to convey insights to stakeholders effectively. Visualization makes complex data more understandable and accessible.
- Big Data and Advanced Analytics:
  For enterprises dealing with large datasets, consider using big data technologies (e.g., Hadoop, Spark) and advanced analytics tools (e.g., data mining, natural language processing) to handle the volume, velocity, and variety of data.
- Predictive Analytics:
  Apply predictive analytics to forecast future trends, customer behavior, and potential business opportunities. Techniques include regression, time series analysis, and machine learning algorithms.
- Prescriptive Analytics:
  Go beyond descriptive and predictive analytics to prescriptive analytics, which provides actionable recommendations and suggests optimal actions to achieve business goals.
- Data Governance and Security:
  Ensure data governance and security measures are in place to protect sensitive and confidential data. Compliance with data privacy regulations (e.g., GDPR, HIPAA) is essential.
- Data Storage and Access:
  Implement efficient data storage solutions and ensure data accessibility for analysts and decision-makers. Consider data warehousing, data lakes, and cloud-based storage options.
- Data Quality and Data Quality Assurance:
  Continuously monitor and improve data quality to maintain the accuracy and reliability of data used for analysis. Implement data quality assurance processes and tools.

**OLTP vs OLTP**

OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing) are two distinct categories of database systems and workloads, each serving a different purpose within an organization. They are designed to handle different types of data processing tasks and have specific characteristics:

- OLTP (Online Transaction Processing):
  - Type of Workload:

    OLTP systems are designed for transactional workloads, which involve many short, simple, and frequent read and write operations on a database. These transactions are typically associated with day-to-day operations, such as processing orders, updating inventory, and managing customer information.
  - Data Characteristics:

    OLTP systems store and manage current, up-to-date data. The data tends to be normalized, with minimal redundancy, to maintain data integrity and support data consistency.
  - Database Schema:

    OLTP databases often use a normalized schema to minimize data redundancy and ensure data consistency. Tables are typically related using foreign key relationships.
  - Query Complexity:

    OLTP queries are simple and typically involve selecting, updating, inserting, or deleting a small number of records. The primary focus is on data retrieval and modification.
  - Performance Requirements:

    OLTP systems prioritize low-latency and high-concurrency capabilities to handle numerous concurrent users and ensure quick response times for individual transactions.
  - Indexes:

    OLTP databases typically include primary key and foreign key indexes to support fast data retrieval and maintain data integrity.
  - Examples:

    E-commerce websites, banking systems, reservation systems, and point-of-sale (POS) systems are examples of applications that use OLTP databases.
- OLAP (Online Analytical Processing):
  - Type of Workload:

    OLAP systems are designed for analytical workloads that involve complex, read-intensive operations for data analysis, reporting, and decision support. These queries are often more complex and involve aggregations, summarizations, and multidimensional data.
  - Data Characteristics:

    OLAP systems store historical, summarized, and aggregated data to support analytical queries. Data is denormalized to improve query performance.
  - Database Schema:

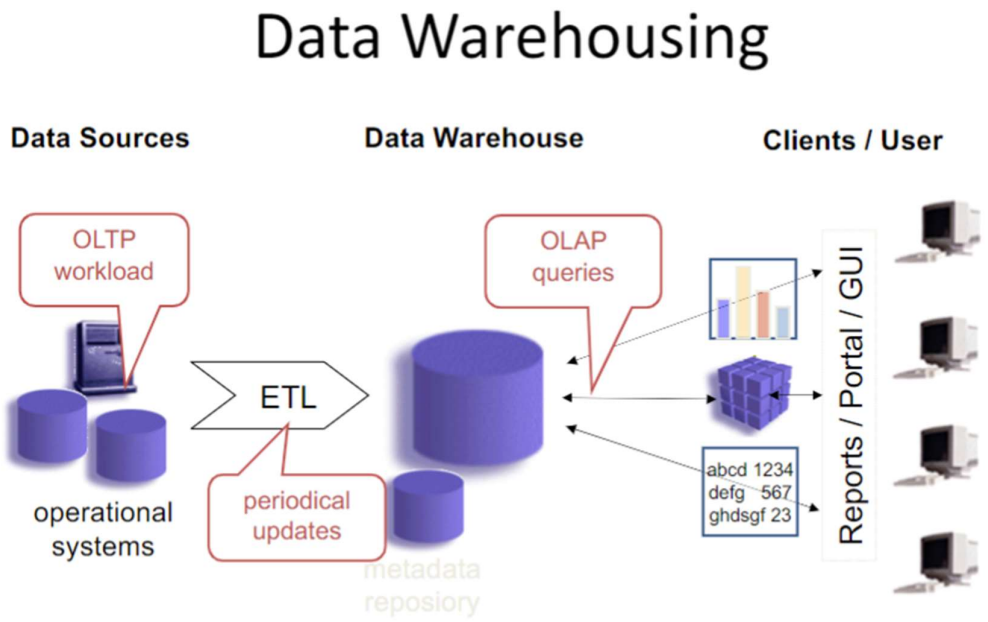
    OLAP databases often use a star or snowflake schema, which involves fact tables and dimension tables to facilitate complex queries. These schemas enable data analysis and reporting.
  - Query Complexity:

    OLAP queries are complex and typically involve aggregation, grouping, and calculations. They are used to generate reports, make business decisions, and uncover insights from historical data.

- Performance Requirements:
  OLAP systems prioritize query performance and throughput over low-latency transaction processing. These systems are optimized for large-scale data analysis.
- Indexes:
  OLAP databases include indexes optimized for query performance, such as bitmap indexes and materialized views, which speed up analytical queries.
- Examples:
  Business intelligence tools, data warehouses, and reporting systems use OLAP databases for analysing historical data, generating reports, and supporting decision-making.

**Data Warehousing**

Data warehousing is a specialized data management and analysis solution that involves the consolidation, storage, and retrieval of large volumes of data from various sources for the purpose of business intelligence, reporting, and data analysis. Data warehouses are designed to support complex query and reporting functions, making them a valuable resource for decision-making within organizations. Data warehousing is essential for organizations that rely on data-driven decision-making, as it enables them to consolidate and analyse data efficiently. It plays a key role in supporting various functions, including financial analysis, market research, performance monitoring, and strategic planning.




Data warehousing, while a valuable tool for organizations, can encounter various challenges and issues that need to be addressed to ensure the effectiveness and reliability of the data warehousing environment. Here are some common issues associated with data warehousing:

- Data Quality:
  Inaccurate, incomplete, or inconsistent data from source systems can affect the quality of data in the data warehouse.
- Data Transformation:
  Extracting and transforming data from heterogeneous sources can be complex, requiring careful mapping and transformation rules.

- Scalability:
  As the volume of data and the number of users grow, data warehouses may face scalability challenges. Performance may degrade if the infrastructure is not scaled appropriately.
- Data Modelling Complexity:
  Designing and maintaining complex data models, such as star schemas and snowflake schemas, can be challenging, particularly for large and intricate datasets.
- Query Performance:
  Complex queries and reporting requests can strain the system's performance. Indexing, query optimization, and hardware resources may be necessary to maintain acceptable performance.
- Data Governance:
  Ensuring data governance, including data lineage, metadata management, and data quality, is often a significant challenge. Inadequate data governance can lead to issues with data accuracy, compliance, and security.
- Data Security:
  Data security is crucial in data warehousing, especially when dealing with sensitive and confidential information. Access control, encryption, and data masking are important considerations.
- Data Volume and Storage Costs:
  Storing large volumes of historical data can result in significant storage costs. Organizations must plan for efficient data archiving and storage strategies.
- Data Latency:
  Delay in data updates and ETL processes can lead to data latency, affecting the timeliness of reports and analysis.
- Data Privacy Compliance:
  Ensuring compliance with data privacy regulations, such as GDPR or HIPAA, requires careful handling of personal and sensitive data.
- ETL Complexity:
  Extract, Transform, Load (ETL) processes can be complex, involving multiple steps and dependencies. Managing and monitoring these processes is essential for maintaining data freshness.

**ETL Process**

The ETL process, which stands for Extract, Transform, Load, is a crucial component of data integration and data warehousing. It involves the movement and manipulation of data from source systems to a target destination, such as a data warehouse, where the data can be stored, analysed, and used for reporting and business intelligence. Here's an overview of each stage of the ETL process:

- Extract:
  The "Extract" phase involves gathering data from various source systems, which can include databases, flat files, external systems, APIs, and more. Data extraction can be full (all data from a source) or incremental (only changes since the last extraction). Extracted data may be in various formats, such as structured data (relational databases), semi-structured data (XML, JSON), or unstructured data (text files, documents).
- Transform:
  The "Transform" phase focuses on cleaning, processing, and reshaping the extracted data to meet the requirements of the target system. Data transformation includes tasks like data cleansing (removing duplicates, handling missing values), data

enrichment (adding calculated or derived fields), and data normalization (ensuring data consistency and integrity). Business rules, validation, and data quality checks are applied during this phase. Data is often converted into a standardized format suitable for analysis, which can include converting currencies, dates, and units of measurement.

- Load:

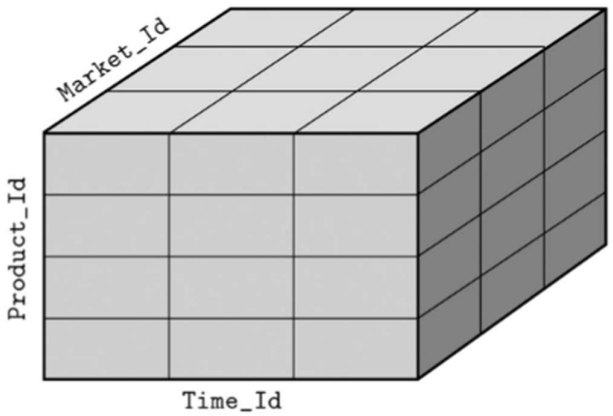
  The "Load" phase is where the transformed data is loaded into the target destination, which is typically a data warehouse, data mart, or another storage system optimized for analytics and reporting. Data loading can be batch-based (scheduled at specific intervals) or near-real-time (streaming data directly into the target system). Various loading strategies can be used, such as insert-only (appending new data), update (modifying existing records), or merge (combining new and existing data).

Metadata refers to data about data. It provides information that describes various aspects of data, helping users and systems understand, manage, and utilize the data effectively. Metadata is crucial in a wide range of fields, including data management, libraries, archives, and information systems. As with other databases, a warehouse must include a metadata repository.

**Relational OLAP**

Relational Online Analytical Processing (ROLAP) is a type of online analytical processing (OLAP) that stores and manages data in a relational database management system (RDBMS). ROLAP systems are designed to facilitate complex data analysis and reporting by leveraging the power of relational databases. In data warehousing and online analytical processing (OLAP), a fact table is a central component of a star schema or snowflake schema. It stores quantitative data (facts) related to a business process, such as sales, inventory, or customer transactions. Fact tables contain numeric or additive data and serve as the foundation for data analysis and reporting.

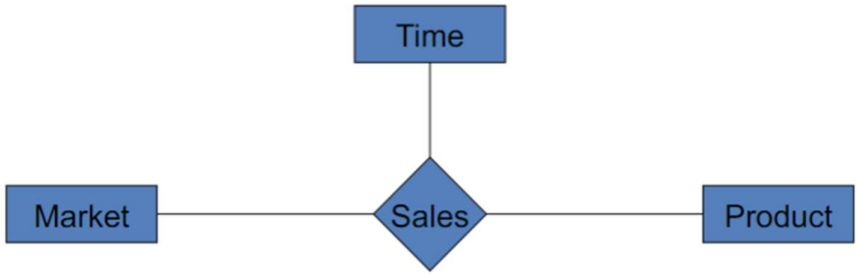
A data cube is a multidimensional representation of data that allows for efficient data analysis, exploration, and reporting in online analytical processing (OLAP) systems. Data cubes provide a structured way to organize and store data, enabling users to perform complex queries and aggregations along multiple dimensions. An example of a data cube is given below:




Dimension tables are a fundamental component of data warehousing and online analytical processing (OLAP) systems. They store descriptive information about various attributes or dimensions of the data, providing context and categorization for the measures (quantitative data) stored in fact tables. Dimension tables are crucial for organizing, analyzing, and reporting on data.

**Star Schema**

A star schema is a type of data modelling schema commonly used in data warehousing and online analytical processing (OLAP) systems. It organizes data into a structured and easily accessible format, making it efficient for complex queries and data analysis. In a star schema, data is organized around a central fact table and dimension tables, creating a star-like structure. An example of its architecture is given below:




**OLAP Queries**

OLAP queries are used to retrieve and analyse data from multidimensional data sources, such as data warehouses and data cubes, to gain insights and support decision-making. OLAP queries are typically more complex than standard SQL queries because they involve operations like aggregation, pivoting, slicing, and dicing. Here are some common OLAP query operations:

- Aggregation over time:

  It refers to the process of summarizing or consolidating data over a specific time frame or period. This operation is valuable for understanding trends, patterns, and changes in data over time.
- Slice:

  Slicing involves selecting a single dimension from a multidimensional dataset to view a specific cross-section of the data. For example, you can slice data to view all sales data for a particular time, regardless of other dimensions.
- Dice:

  Dicing involves selecting specific values or combinations of values from two or more dimensions to view a subset of data. For instance, you can dice data to view sales data for a specific product category in a particular region.
- Pivot (Rotate):

  Pivoting is the process of changing the orientation of data to view it from a different perspective. You can pivot data to change the rows and columns or to rotate dimensions.
- Roll-Up:

  Rolling up data means aggregating data from a lower level of granularity to a higher level. For example, you can roll up monthly sales data to view quarterly or yearly totals.
- Drill-Down:

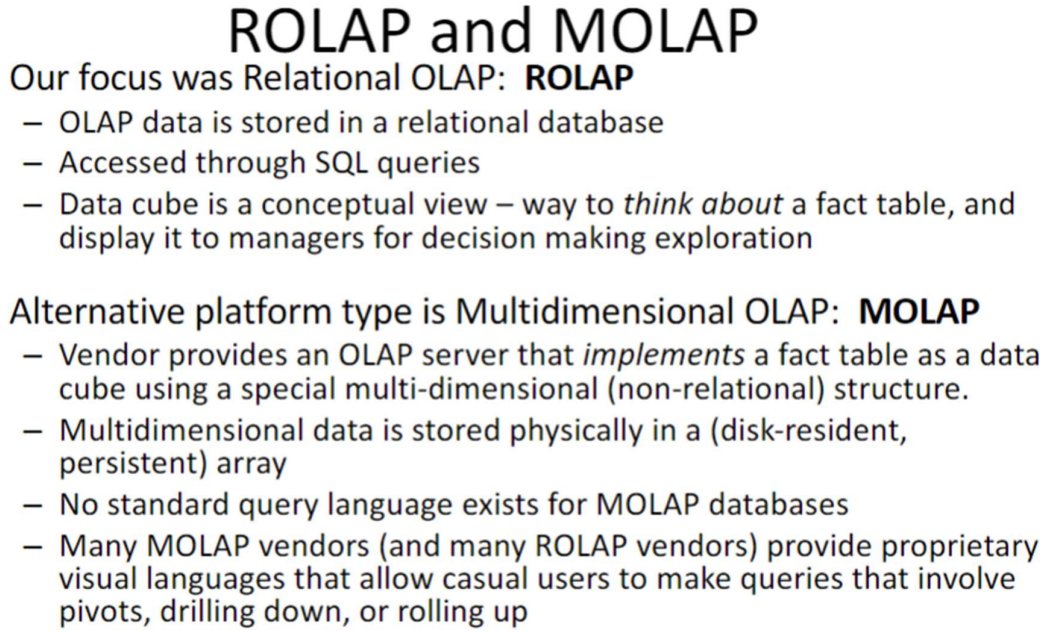
  Drilling down involves viewing data at a more detailed level of granularity. For instance, you can drill down from yearly sales data to monthly or daily sales.
- Cross tab with marginals

  A cross-tabulation (crosstab) with marginals, also known as a contingency table with row and column totals, is a tabular representation of the relationship between two categorical variables. This table provides a summary of the frequencies or counts of observations that fall into various combinations of categories for the two variables.

Marginals in this context refer to the total counts or frequencies for each category of one or both variables.

**ROLAP vs MOLAP**

ROLAP (Relational Online Analytical Processing) and MOLAP (Multidimensional Online Analytical Processing) are two different approaches to organizing and querying data in the context of online analytical processing (OLAP) systems.




**Modern Trends in Analytics**

Data lakes and data Lakehouses are both data storage and management architectures used in the field of big data and analytics. They have similarities, but they also have distinct characteristics and purposes.

- Data Lake

  A data lake is a central repository for storing vast amounts of structured and unstructured data, including raw data, logs, sensor data, and more. It follows a schema-on-read approach, meaning data is ingested and stored without a predefined schema. Data lakes can store data in its native format, which makes them suitable for handling a wide variety of data types, from text and images to structured data in databases. Data lakes are highly scalable and can accommodate large volumes of data. They can scale out horizontally to handle growing datasets. Data lakes are often combined with data processing engines (e.g., Apache Spark, Apache Hadoop) to transform and analyse data. Processing is typically done on an ad-hoc basis, depending on the analysis requirements.

- Data Lakehouse

  A data Lakehouses is an architecture that combines the storage capabilities of a data lake with the processing and query capabilities of a data warehouse. It follows a schema-on-read approach like a data lake but incorporates some elements of schema-on-write, allowing for more structured data storage. Like data lakes, data Lakehouses can store a wide variety of data types. However, they offer more options for organizing and structuring data. Data Lakehouses maintain the scalability of data lakes for storage while also providing the benefits of a data warehouse for querying and processing. Data Lakehouses often integrate with modern data processing engines, but they also allow

for SQL-based querying and analytics, making it easier to analyse data directly within the storage system.

**Data Governance**

Data governance is a comprehensive framework and set of practices that organizations use to manage their data effectively, ensuring data is accurate, secure, compliant, and available for decision-making. Data governance encompasses policies, procedures, roles, responsibilities, and technologies that support data management and ensure the quality and integrity of data throughout its lifecycle.