Section 1- World before Database

Before databases, data was managed manually or with rudimentary tools that lacked the structure, flexibility, and efficiency of modern systems. Here's a look at the approaches and systems used to manage data before the advent of databases:

---

1. Manual Record Keeping

- **Paper-Based Systems:**
  - Data was recorded on paper in ledgers, journals, and logbooks.
  - Organizing and retrieving information was time-consuming and error-prone.
- **Indexing Methods:**
  - File cabinets and card catalogues were used to index and locate records.
  - Examples include library catalogue systems for organizing books.

2. Early Mechanical Systems

- **Punch Cards :**
  - Invented by Herman Hollerith for the 1890 U.S. Census.
  - Data was encoded on physical cards using punched holes.
  - Mechanical tabulators processed the cards to perform basic calculations and sorting.
- **Keypunch Machines:** Allowed operators to punch data into cards manually.

3. Magnetic Tape Storage:

- Magnetic tapes replaced punch cards as a storage medium.
- Data was stored sequentially, which made reading large datasets efficient but random access difficult.
- Common in early computing systems, such as the UNIVAC I.

4. Flat File Systems:

- **Flat Files:**
  - Data was stored in plain text or binary files with no internal relationships.
  - Each file was standalone, and accessing data required custom programs.
  - Files were often organized in a hierarchical directory structure.
- **Challenges:**
  - Redundancy: The same data was repeated across files.
  - Inflexibility: Difficult to modify or scale.
  - Lack of data integrity and security.

5. Proprietary Data Management Systems:

- Organizations developed **custom file systems** for specific applications.
- Early computing systems like IBM's 1401 and 7090 used these for tasks like payroll or inventory management.
- Data access relied heavily on low-level programming, such as COBOL or assembly language.

Key Limitations before Databases

1. **Lack of Standardization:** Each system had its own way of organizing and accessing data.
2. **No Relationships:** Data could not be easily linked or queried.
3. **Redundancy:** Data duplication led to inconsistencies.
4. **Poor Scalability:** Systems were not designed for large-scale data management.

The limitations of these early systems led to the development of **databases**, which introduced structured data storage, relationships, and query capabilities, revolutionizing how information was managed and accessed.

Need of Database

The need for databases arises from the growing complexity, volume, and interconnectivity of data in modern applications. Databases address several key challenges and provide solutions that are essential for efficient and reliable data management.

1. Structured Data Organization

- Databases provide a structured way to store data using tables, rows, and columns (in relational databases) or other formats like key-value pairs, documents, or graphs.
- This structure makes data more accessible, understandable, and easier to manage.

2. Efficient Data Storage and Retrieval

- Manual or file-based systems struggle with large amounts of data and complex queries.
- Databases enable quick and efficient data retrieval through indexing, querying languages (like SQL), and optimization techniques.

3. Eliminating Redundancy

- **Normalization:** Databases reduce redundancy by organizing data into smaller, related tables. This improves consistency and saves storage space.
- **Data Integrity:** Ensures that data remains accurate and consistent across the system.

4. Supporting Multiple Users

- Databases are designed to handle multiple users accessing and modifying data simultaneously (concurrency).
- Features like **locking mechanisms** and **transaction management** prevent conflicts and ensure data consistency.

5. Scalability

- Databases can scale to handle growing amounts of data and user demands, whether vertically (adding more resources to a server) or horizontally (adding more servers).

6. Data Security

- Databases provide robust security features:
    - Access controls to restrict who can view or modify data.
    - Encryption to protect data from unauthorized access.
    - Audit trails to track changes and identify potential breaches.

7. Data Relationships

- Modern applications require data to be interconnected (e.g., linking customers to orders or products to categories).

- Databases allow the creation and management of these relationships, enabling complex queries and insights.

8. Transaction Management

- **Atomicity, Consistency, Isolation, Durability (ACID):**
  - Ensures that database transactions are processed reliably.
  - Critical for applications like banking, where accuracy is essential.

9. Backup and Recovery

- Databases provide tools for automated backups and recovery, ensuring that data can be restored in case of hardware failure, software bugs, or human errors.

10. Integration with Applications

- Databases serve as the backbone for modern applications, enabling them to:
  - Store user data, configuration settings, and logs.
  - Perform analytics and reporting.
  - Integrate with other systems via APIs or data pipelines.

11. Big Data and Analytics

- Databases are essential for storing and analyzing large datasets to extract valuable insights.
- Specialized databases, like NoSQL and data warehouses, support big data and advanced analytics.

Without databases, modern applications would face significant challenges in managing data, leading to inefficiency, errors, and scalability issues. Databases provide the foundation for robust, reliable, and efficient data handling.

Relational Databases (RDBMS)

- **Definition:** Store data in tables with rows and columns, and relationships between data are defined through keys (primary and foreign).
- **Features:**
  - Use SQL (Structured Query Language) for data manipulation.
  - Ensure data integrity through constraints and normalization.
  - ACID compliance for transaction reliability.
- **Examples:**
  - MySQL
  - PostgreSQL
  - Oracle Database
  - Microsoft SQL Server
  - SQLite

A **relational database** is like a digital filing system that organizes data into **tables**, similar to how you might organize information in a spreadsheet. Each table contains rows and columns:

1. **Rows:** Each row represents a single record or entry (like one student's information in a student database).
2. **Columns:** Each column represents a specific type of data or attribute (like Name, Age, or Grade).

Key Features of a Relational Database

1. **Tables (Relations):** Data is stored in tables, which are connected to each other by relationships.
2. **Primary Key:** A unique identifier for each row in a table (like a student ID number).
3. **Foreign Key:** A column in one table that links to the primary key in another table, creating relationships between tables (e.g., linking students to their courses).
4. **SQL (Structured Query Language):** A language used to manage and query the data in the database.

Example

Imagine a school database:

- **Student Table:**

Student_ID	Name	Age
1	Alice	15
2	Bob	16

- **Courses Table:**

Course_ID	Course_Name
101	Math
102	Science

- **Enrollment Table:**

Enrollment_ID	Student_ID	Course_ID
1	1	101
2	1	102
3	2	101

Here:

- The `Student_ID` in the Student table is a **primary key**.
- The `Student_ID` in the Enrollment table is a **foreign key**, linking it to the Student table.

Why Use Relational Databases?

- **Organized:** Data is structured, making it easy to retrieve and update.
- **Efficient:** Relationships between tables reduce duplication and save space.
- **Reliable:** They ensure data accuracy and consistency with rules like **primary keys** and **foreign keys**.

Relational databases are widely used because they are straightforward, scalable, and work well for many applications, like e-commerce websites, banking systems, and inventory management.

SQL is relational database?

No, **SQL** itself is not a relational database. Instead, **SQL (Structured Query Language)** is a **programming language** used to manage and interact with **relational databases**.

SQL is the **tool** or **language** that you use to work with a relational database. While SQL is closely tied to relational databases, the database itself is the system that organizes, stores, and retrieves the data.

SQL History

SQL's journey began with the theoretical groundwork laid by Edgar F. Codd and matured through IBM's development of System R and SEQUEL. Over decades, it has evolved into a powerful, standardized language that underpins modern data management and remains a critical tool in technology and business.

Is SQL syntax differs in oracle, MYSQL, MS SQL server?

Yes, **SQL differs slightly in Oracle, MySQL, SQL Server, and other relational database systems**, even though they all use SQL as the standard query language. These differences arise because each database management system (DBMS) extends the SQL standard with its own proprietary features and optimizations.

SQL fundamentals

1. SQL Basics Overview

SQL is used to interact with a relational database. A database consists of tables that are organized into rows (records) and columns (fields).

Diagram: Basic Table Structure

ID	Name	Age	Email
1	Alice	20	alice@example.com
2	Bob	25	bob@example.com
3	Charlie	22	charlie@example.com

2. SQL Categories with Visual Examples

a. Data Definition Language (DDL)

DDL commands define the structure of the database.

Diagram: Creating a Table

```
CREATE TABLE Students (
    ID INT PRIMARY KEY,
    Name VARCHAR(50),
    Age INT
);
```

Operation	Description
`CREATE TABLE`	Creates a new table.
`ALTER TABLE`	Modifies a table structure.
`DROP TABLE`	Deletes a table.

b. Data Manipulation Language (DML)

DML commands manipulate the data within tables.

Diagram: Adding and Querying Data

```
INSERT INTO Students (ID, Name, Age) VALUES (1, 'Alice', 20);
INSERT INTO Students (ID, Name, Age) VALUES (2, 'Bob', 25);

SELECT * FROM Students;
```

ID	Name	Age
1	Alice	20
2	Bob	25

c. Data Query Language (DQL)

DQL is used for querying data.

```
SELECT Name, Age FROM Students WHERE Age > 20;
```

Output:

Output:

Name	Age
Bob	25

d. Data Control Language (DCL)

DCL commands manage access and permissions.

```
GRANT SELECT ON Students TO User1;
REVOKE SELECT ON Students FROM User1;
```

Command	Effect
`GRANT`	Gives permissions to a user.
`REVOKE`	Removes permissions from a user.

e. Transaction Control Language (TCL)

TCL manages groups of SQL commands.

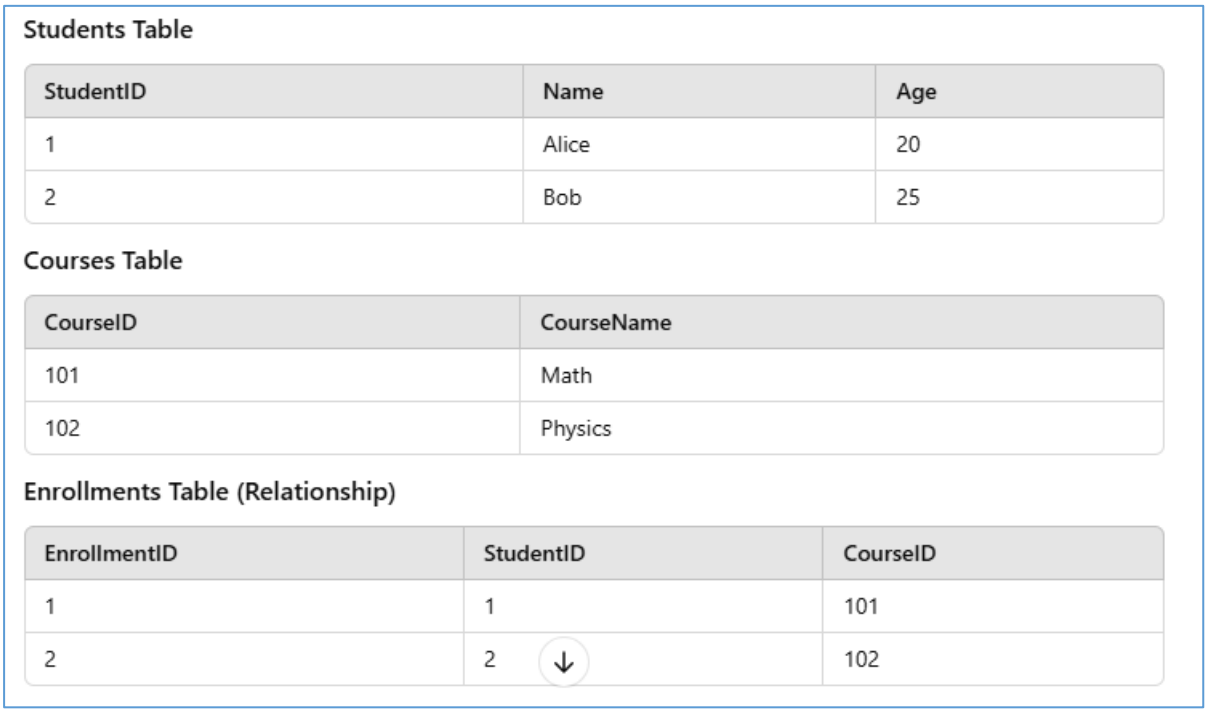
```
BEGIN TRANSACTION;
UPDATE Students SET Age = 21 WHERE ID = 1;
ROLLBACK; -- Undo changes
```

Operation	Effect
`COMMIT`	Saves changes permanently.
`ROLLBACK`	Undoes changes made during a transaction.

3. SQL Key Concepts

a. Primary Key and Foreign Key

Diagram: Relationships between Tables



Students Table

StudentID	Name	Age
1	Alice	20
2	Bob	25

Courses Table

CourseID	CourseName
101	Math
102	Physics

Enrollments Table (Relationship)

EnrollmentID	StudentID	CourseID
1	1	101
2	2	102

- **Primary Key:** A unique identifier (e.g., `StudentID` in `Students`).
- **Foreign Key:** Links one table to another (e.g., `StudentID` in `Enrollments` refers to `Students`).

---

b. Data Types

Data Type	Description	Example
INT	Integer numbers	`25`
VARCHAR(size)	Text with specified max length	`'Alice'`
DATE	Date values	`2025-01-07`
FLOAT	Decimal numbers	`25.5`

4. Querying Data with SELECT

Diagram: Filtering and Sorting

Query Example:

```
SELECT Name, Age FROM Students WHERE Age > 20 ORDER BY Age DESC;
```

Output:

Name	Age
Bob	25

Name	Age
Bob	25

---

5. JOINs

a. INNER JOIN

Combines rows from two tables where a match exists.

Query Example:

```
SELECT Students.Name, Courses.CourseName
FROM Students
INNER JOIN Enrollments ON Students.ID = Enrollments.StudentID
INNER JOIN Courses ON Enrollments.CourseID = Courses.CourseID;
```

Output:

Name CourseName

Alice Math

Name	CourseName
Bob	Physics

b. LEFT JOIN

Returns all rows from the left table and matching rows from the right table.

Diagram for JOINs

---

6. Aggregate Functions

Aggregate functions perform calculations on data.

Function	Description	Example
COUNT()	Counts rows	`COUNT(*)`
SUM()	Adds numeric values	`SUM(Age)`
AVG()	Calculates average	`AVG(Age)`
MAX()	Finds maximum value	`MAX(Age)`
MIN()	Finds minimum value	`MIN(Age)`

Query Example:

```
SELECT COUNT(*) AS TotalStudents FROM Students;
```

Output:

TotalStudents
2

---

7. Constraints

Constraints ensure data integrity in the database.

Constraint	Description
NOT NULL	Prevents null values.
UNIQUE	Ensures all values are unique.
PRIMARY KEY	Unique identifier for a row.
FOREIGN KEY	Links tables to enforce rules.
DEFAULT	Assigns default values.

Example:

```
CREATE TABLE Students (
    ID INT PRIMARY KEY,
    Name VARCHAR(50) NOT NULL,
    Age INT DEFAULT 18
);
```

---

8. Normalization

Normalization organizes data to reduce redundancy.

Diagram: Example of Normalization

1. **Unnormalized Table**
   | **StudentID** | **Name** | **Course1** | **Course2** | |---------------|----------|-------------|-----------| | 1 | Alice | Math | Physics |
2. **Normalized Tables**
   Students Table
   | **StudentID** | **Name** | |---------------|----------| | 1 | Alice |

Enrollments Table

StudentID	CourseName
1	Math
1	Physics

---

9. Transactions

Transactions group multiple SQL operations into a single logical unit.

Diagram: Transaction States

1. Begin Transaction

2. Perform Operations
3. Commit (Save) or Rollback (Undo)

Example:

```
BEGIN TRANSACTION;
UPDATE Students SET Age = 21 WHERE ID = 1;
COMMIT;
```

---

10. Indexes

Indexes speed up query performance by creating a structure for fast lookups.

Diagram: Index on a Table

Without Index:
Search through all rows.

With Index:
Use a binary tree or similar structure to locate data faster.

Example:

```
CREATE INDEX idx_name ON Students(Name);
```

Conclusion

SQL fundamentals involve understanding how to create, manipulate, and query databases effectively. Using diagrams like table structures, relationships, and join operations makes it easier to visualize and grasp the concepts. If you'd like custom diagrams or further examples, let me know!