SQL
Study Material

Hello and welcome to our SQL learning journey!

I'm S.Q. the Squirrel, your friendly guide...

to mastering the world of databases and SQL.

INDEX

Hello and welcome to our SQL learning journey! I'm S.Q. the Squirrel, your friendly guide to mastering the world of databases and Structured Query Language (SQL). Whether you're a complete beginner or looking to brush up on your SQL skills, I'm here to help you every step of the way.

So, whether you're ready to embark on your SQL adventure or simply curious about databases and data management, you've come to the right place! Let's squirrel away some SQL knowledge and unlock the full potential of databases together. Get ready to dive in, ask questions, and explore the fascinating world of SQL with me, S.Q. the Squirrel!

Welcome aboard, and let's get started on our SQL learning journey!

Chapter 1:Introduction to Databases

1.1 What is a Database?

A database is a structured collection of data organized for easy retrieval, manipulation, and storage. It's like a digital warehouse where you can store various types of information, such as customer details, product inventory, or employee records.

Imagine a database as a digital filing cabinet where you can store and organize information. It helps us keep track of data efficiently, making it easier to retrieve and manage.

Example:

Let's consider a simple database for a library. Each book in the library has its own entry in the database, containing information such as the book title, author, publication year, and genre. This information is organized into tables within the database, with each table representing a different type of data (e.g., books, authors, genres).

To create a basic database, we'll use SQL (Structured Query Language). Here's an example of SQL code to create a simple library database:

```
CREATE DATABASE LibraryDatabase;
```

This is an SQL command used to create a new database.

This is the name of our database. You can replace it with any name you prefer.

This SQL command creates a new database named "LibraryDatabase". Think of it as creating a blank canvas where we'll later add tables to store our library information.

1.2 Types of Databases

1.2.1 Relational Databases (SQL Databases):

- Think of these databases like spreadsheets. Data is organized neatly into tables with rows and columns, just like in Excel.
- They are good for structured data where relationships between different pieces of information are important.
- Examples: MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server.

1.2.2 NoSQL Databases:

- NoSQL databases are like a collection of different tools rather than just one tool like a spreadsheet. They can handle a variety of data types and structures.
- They're flexible and can handle large amounts of data that might not fit neatly into tables.
- NoSQL databases can be categorized into several types:
    - Document Stores: Store data in flexible, JSON-like documents. Examples: MongoDB, Couchbase.
    - Key-Value Stores: Store data as key-value pairs. Examples: Redis, Amazon DynamoDB.
    - Column-Family Stores: Organize data into columns rather than rows. Examples: Apache Cassandra, HBase.
    - Graph Databases: Represent data as nodes, edges, and properties, making them suitable for complex relationships. Examples: Neo4j, Amazon Neptune.

1.2.3 NewSQL Databases:

- These databases are like the cool new version of the old spreadsheet. They can handle lots of data and work well in big teams, just like Google Sheets.
- They combine the reliability of traditional databases with the flexibility of newer ones.
- Examples: Google Spanner, CockroachDB.

1.2.4 In-Memory Databases:

- In-memory databases are like a super-fast version of a regular database. They keep all the data in the computer's memory (RAM) instead of on the hard drive, making them really quick to access.
- They're great for applications where speed is important, like video games or financial trading systems.
- Examples: Redis (can also be categorized as a NoSQL key-value store), Oracle TimesTen, SAP HANA.

1.2.5 Wide Column Stores:

- These databases are like storing data in a series of columns rather than rows. It's like having a bunch of columns where each column has its own set of data.
- They're good for handling a lot of data and can be really fast, especially for certain types of tasks.
- Examples: Apache Cassandra, HBase.

Each type of database has its own strengths and use cases, and the choice of database depends on factors such as data structure, scalability requirements, consistency needs, and performance considerations.

Chapter 2:Introduction to SQL

2.1 What is SQL?

SQL is a standard language used for managing and manipulating relational databases. It provides a set of commands that allow users to interact with databases in a structured and efficient manner.

in simple way, SQL is a language used to communicate with databases.

Whether you want to retrieve specific information from a database, make changes to existing data, or define the structure of a database, SQL has you covered.

It allows us to perform various operations like retrieving data, adding new data, updating existing data, and deleting data.

2.2 Basic SQL Syntax

SQL syntax refers to the rules and conventions used to write SQL statements. Here are some basic components of SQL syntax:

- **Keywords:** SQL statements are made up of keywords that indicate the action to be performed. Examples of keywords include SELECT, INSERT, UPDATE, DELETE, and CREATE.
- **Clauses:** SQL statements typically consist of one or more clauses, which are keywords that specify additional instructions or conditions for the statement. Common clauses include WHERE, ORDER BY, GROUP BY, and HAVING.

- **Expressions:** SQL statements often include expressions, which are combinations of values, operators, and functions that produce a result. Expressions can be used to calculate values, filter data, and perform other operations.

- **Comments:** SQL allows you to include comments in your code to provide explanations or annotations. Comments are preceded by two consecutive hyphens (--).

2.3 SQL Data Types

SQL data types define the type of data that can be stored in a column of a database table. Each column in a table is assigned a specific data type, which determines the kind of values that can be stored in that column.

Here are some common SQL data types:

2.3.1 Integer:

- Used to store whole numbers (e.g., 1, 10, -5).
- Example: INT, INTEGER, SMALLINT, BIGINT.

2.3.2 Decimal/Floating Point:

- Used to store numbers with decimal points (e.g., 3.14, 10.5, -0.75).
- Example: DECIMAL, NUMERIC, FLOAT, REAL, DOUBLE.

2.3.3 Character/String

- Used to store text or character data (e.g., "hello", 'world', '123').
- Example: CHAR, VARCHAR, TEXT.

2.3.4 Date/Time

- Used to store date and time values (e.g., '2024-03-08', '12:30:00').
- Example: DATE, TIME, DATETIME, TIMESTAMP.

2.3.5 Boolean:

- Used to store true/false or binary data (e.g., TRUE, FALSE, 1, 0).
- Example: BOOLEAN, BIT.

2.3.6 Binary:

- Used to store binary data, such as images or files.
- Example: BLOB, BYTEA.

Chapter 3: Creating a Database and a Table

Databases are organized collections of data, typically stored and managed using specialized software called Database Management Systems (DBMS).

In this chapter, we'll explore how to create a database and add data to it using SQL (Structured Query Language).

3.1 Setting Up SQL Environment:

Before we can create a database, we need to set up our SQL environment. This may involve installing a DBMS like MySQL, PostgreSQL, or SQLite, and connecting to it using a command-line interface or a graphical user interface (GUI) tool.

1. Install and configure a DBMS (e.g., MySQL, PostgreSQL, SQLite).
2. Connect to the DBMS using a command-line interface (CLI) or a graphical user interface (GUI) tool like MySQL Workbench or pgAdmin.

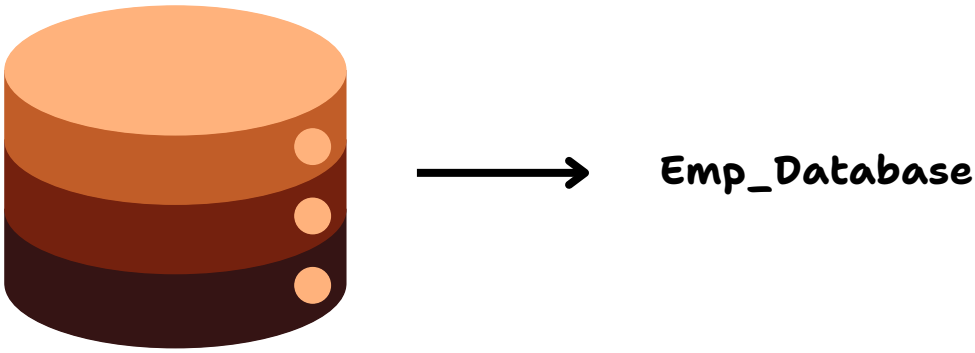
3.2 Creating a Database:

We'll learn how to create a new database using SQL commands. A database provides a structured environment for storing and organizing data, making it easier to manage and retrieve information.

- Use the **CREATE DATABASE** statement to create a new database.

```
CREATE DATABASE Emp_Database;
```

now we have created a new database called Emp_Database



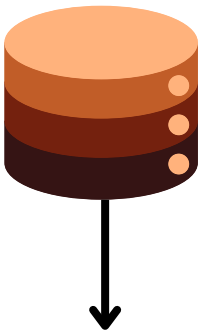
3.3 Defining Database Tables:

Once the database is created, we'll define the structure of the data by creating tables. Tables consist of rows and columns, with each column representing a specific attribute of the data and each row representing a single record.

- Use the **CREATE TABLE** statement to define the structure of a table.

```
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT,
    department VARCHAR(50)
);
```

now we have created a new table called Employees



Id	name	age	department

3.4 Adding Data to Tables:

With the database and tables in place, we'll explore various methods for adding data to the tables. This may involve inserting individual records or bulk-inserting data from external sources.

- Use the INSERT INTO statement to add new records to a table.

a) Inserting a single record.

```
INSERT INTO employees (id, name, age, department)
VALUES (1, 'Bella', 25, 'HR');
```

Id	name	age	department
1	Bella	25	HR

b) Inserting multiple records in a single statement.

```
INSERT INTO employees (id, name, age, department)
VALUES
      (2, 'Cece', 30, 'IT'),
      (3, 'Cody', 35, 'Finance');
```

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
3	Cody	35	Finance

c) Inserting data without specifying column names

```
INSERT INTO employees
VALUES
        (4, 'Nico', 40, 'HR'),
        (5, 'Emma', 28, 'Marketing');
```

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
3	Cody	35	Finance
4	Nico	40	HR
5	Emma	28	Marketing

Chapter 4: Accessing Data in a Table

Accessing data in a table means getting the information stored in it. It's like looking up information in a big book.

In this chapter, we'll learn different ways to do this using SQL, the language for working with databases.

4.1 SELECT Statement:

- The **SELECT** statement is like asking for specific information from the table.
- It's like saying, "Show me the names and ages of all employees."
- Syntax:

  SELECT column1, column2, ... FROM table_name;

```
#Use the SELECT statement to retrieve all columns from a table:

SELECT * FROM employees;
```

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
3	Cody	35	Finance
4	Nico	40	HR
5	Emma	28	Marketing

If you want to select specific columns instead of using the wildcard *, you can list the column names separated by commas after the SELECT keyword.

```
SELECT name, department FROM employees;
```

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
3	Cody	35	Finance
4	Nico	40	HR
5	Emma	28	Marketing

name	department
Bella	HR
Cece	IT
Cody	Finance
Nico	HR
Emma	Marketing

4.2 SELECT DISTINCT:

- **SELECT DISTINCT** is used when you want to see only unique values in a column.
- It's like asking, "Tell me all the different departments we have."
- Syntax:

  SELECT DISTINCT column_name FROM table_name;

```
SELECT DISTINCT department FROM employees;
```

department
HR
IT
Finance
HR
Marketing

department
HR
IT
Finance
Marketing

4.3 WHERE Clause:

- The WHERE clause is like applying a filter to your search.
- It's like saying, "Show me only the employees who work in the HR department."
- Syntax:

SELECT * FROM table_name WHERE condition;

```sql
SELECT * FROM employees WHERE department = 'HR';
```

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
3	Cody	35	Finance
4	Nico	40	HR
5	Emma	28	Marketing

Id	name	age	department
1	Bella	25	HR
4	Nico	40	HR

4.4 ORDER BY Clause:

- ORDER BY is used when you want to sort the results in a specific order(ascending or descending order).
- It's like asking, "List the employees in order of their ages, from oldest to youngest."
- Syntax:

SELECT * FROM table_name ORDER BY column_name [ASC|DESC];

```
SELECT * FROM employees ORDER BY age DESC;
```

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
3	Cody	35	Finance
4	Nico	40	HR
5	Emma	28	Marketing

Id	name	age	department
4	Nico	40	HR
3	Cody	35	Finance
2	Cece	30	IT
5	Emma	28	Marketing
1	Bella	25	HR

4.5 LIMIT Clause:

- LIMIT is used when you want to see only a certain number of results.
- It's like saying, "Just show me the first 5 employees."
- Syntax:

  SELECT * FROM table_name LIMIT n;

```
SELECT * FROM employees Limit 3;
```

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
3	Cody	35	Finance
4	Nico	40	HR
5	Emma	28	Marketing

↓

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
3	Cody	35	Finance

Chapter 5: Filtering Data

Filtering data is a fundamental concept in working with databases. It involves selecting specific records from a dataset based on certain criteria or conditions.

By filtering data, we can focus on the information that is relevant to our analysis or application, making it easier to extract insights or perform tasks efficiently.

5.1 Comparison Operators

Comparison operators are used in SQL to compare values in expressions and conditions. They allow us to filter data based on specific criteria.

5.1.1 Equal to (=):

- Checks if two values are equal.

```sql
SELECT * FROM employees WHERE department = 'HR';
```

Id	name	age	department
1	Bella	25	HR
4	Nico	40	HR

5.1.2 Not Equal to (!= or <>):

- Checks if two values are not equal.

```
SELECT * FROM employees WHERE age <> 30;
```

Id	name	age	department
1	Bella	25	HR
3	Cody	35	Finance
4	Nico	40	HR
5	Emma	28	Marketing

5.1.3 Greater Than (>):

- Checks if one value is greater than another

```
SELECT * FROM employees WHERE age > 30;
```

Id	name	age	department
3	Cody	35	Finance
4	Nico	40	HR

5.1.4 Greater Than or Equal To (>=):

- Checks if one value is greater than or equal to another

```
SELECT * FROM employees WHERE age >= 30;
```

Id	name	age	department
2	Cece	30	IT
3	Cody	35	Finance
4	Nico	40	HR

5.1.5 Less Than (<) :

- Checks if one value is less than another.

```
SELECT * FROM employees WHERE age < 35;
```

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
5	Emma	28	Marketing

5.1.6 Less Than or Equal To (<=):

- Checks if one value is less than or equal to another.

```
SELECT * FROM employees WHERE age <= 35;
```

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
3	Cody	35	Finance
5	Emma	28	Marketing

5.2 Logical Operators

Logical operators are used to combine multiple conditions in SQL queries. They allow us to create more complex filtering criteria.

5.2.1 AND Operator:

Combines two or more conditions, and all conditions must be true for the row to be included in the result.

```
SELECT * FROM employees WHERE department = 'HR' AND age > 25;
```

Id	name	age	department
1	Bella	25 ✗	HR ✓
2	Cece	30 ✓	IT ✗
3	Cody	35 ✓	Finance ✗
4	Nico	40 ✓	HR ✓
5	Emma	28 ✓	Marketing ✗

→ ✗
→ ✗
→ ✗
→ ✓
→ ✗

↑ age > 25 ?

↑ department = HR

↓

Id	name	age	department
4	Nico	40	HR

5.2.2 OR Operator:

Combines two or more conditions, and at least one condition must be true for the row to be included in the result.

```
SELECT * FROM employees WHERE department = 'HR' OR department = 'IT';
```

Id	name	age	department
1	Bella	25	HR ✓
2	Cece	30	IT ✓
3	Cody	35	Finance
4	Nico	40	HR ✓
5	Emma	28	Marketing

↑ department = HR/IT

↓

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
4	Nico	40	HR

5.2.2 NOT Operator:

Combines two or more conditions, and at least one condition must be true for the row to be included in the result.

```
SELECT * FROM employees WHERE NOT department = 'HR';
```

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
3	Cody	35	Finance
4	Nico	40	HR
5	Emma	28	Marketing

department NOT= HR

Id	name	age	department
2	Cece	30	IT
3	Cody	35	Finance
5	Emma	28	Marketing

Chapter 6: Aggregate Functions and GROUP BY Clause

Aggregate functions in SQL are used to perform calculations on a set of values and return a single result. They are often used to summarize data or generate statistical information from a dataset.

6.1 Aggregate Functions

Lets's add one more column in our employee table to perform aggregation functions.

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

6.1.1 COUNT Function:

- Counts the number of rows in a result set.
- Syntax:

  COUNT(expression)

```
# Counts the number of rows in the "employees" table
SELECT  COUNT(*) FROM employees;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

Count(*)
5

This query returns the total number of rows in the "employees" table, which is 5.

6.1.2 SUM Function:

- Calculates the sum of values in a column.
- Syntax:

SUM(expression)

```
-- Calculates the total salary of all employees
SELECT SUM(salary) FROM employees;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

SUM(SALARY)
245000

This query calculates the sum of the "salary" column for all employees in the "employees" table, which is 245000.

6.1.3 AVG Function:

- Calculates the average of values in a column.
- Syntax:

  AVG(expression)

```
-- Calculates the average age of all employees
SELECT AVG(age) FROM employees:
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

AVG(age)
31.6

This query computes the average age of all employees in the "employees" table, which is approximately 31.6.

6.1.4 MIN Function:

- Returns the minimum value in a column.
- Syntax:

  MIN(expression)

```
-- Retrieves the minimum age among all employees
SELECT MIN(age) FROM employees;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

MIN(age)
25

This query finds the smallest value in the "age" column of the "employees" table, which is 25.

6.1.5 MAX Function:

- Returns the maximum value in a column.
- Syntax:

  MAX(expression)

```sql
-- Retrieves the maximum age among all employees
SELECT MAX(age) FROM employees;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

MAX(age)
40

This query finds the largest value in the "age" column of the "employees" table, which is 40

6.2 GROUP BY Clause:

- The GROUP BY clause is used to group rows that have the same values into summary rows.
- Syntax:

  GROUP BY column_name

6.2.1 Grouping Data with GROUP BY:

```
-- Counts the number of employees in each department
SELECT department, COUNT(*) FROM employees GROUP BY department;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

department	COUNT(*)
HR	2
IT	1
Finance	1
Marketing	1

This query groups employees by their department and counts the number of employees in each department.

6.2.2 Combining Aggregate Functions with GROUP BY:

```
-- Calculates the average salary for each department
SELECT department, AVG(salary) FROM employees GROUP BY department;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

department	AVG(salary)
HR	60000
IT	45000
Finance	55000
Marketing	25000

This query groups employees by their department and computes the average salary for each department.

6.2.3 HAVING Clause with GROUP BY:

```
SELECT department, AVG(salary) AS avg_salary
FROM employees
GROUP BY department
HAVING COUNT(id) > 1;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

department	AVG(salary)
HR	60000

- The GROUP BY department groups the rows by the "department" column.
- The HAVING COUNT(id) > 1 filters out groups where the count of employees in each department is not greater than 1.
- So, only the "HR" department is returned because it has more than one employee, and the average salary for the "HR" department is calculated.

Chapter 7: Joins in SQL

When you have data spread across multiple tables in a database, you often need to combine them to get meaningful insights. Joins allow you to do just that. They bring together related data from different tables based on a common column.

7.1 Types of Joins:

There are several types of joins in SQL, each serving a different purpose:

1. Inner Join
2. Left Join (or Left Outer Join)
3. Right Join (or Right Outer Join)
4. Full Join (or Full Outer Join)

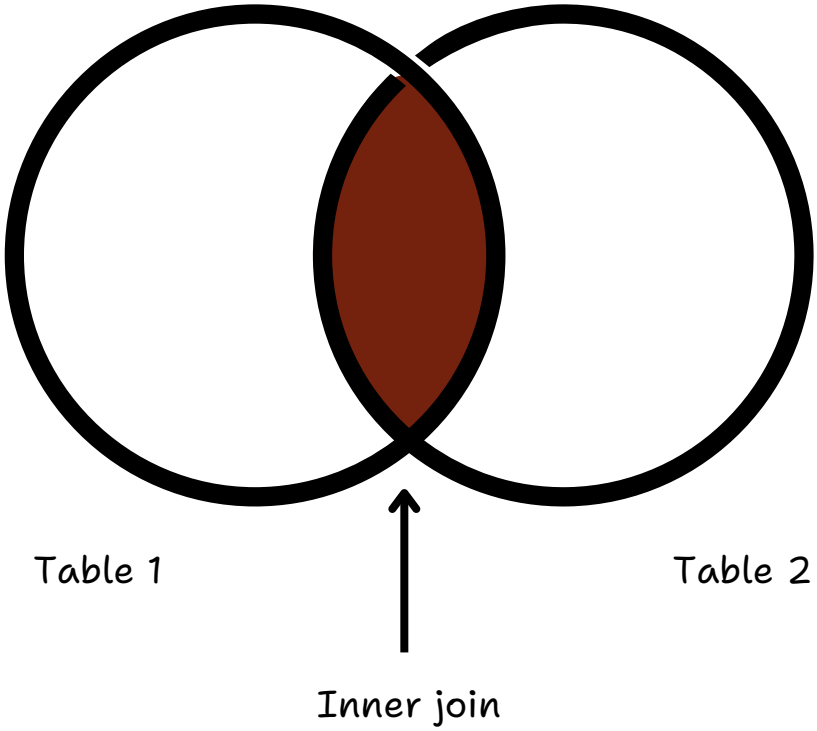
Let's create a new table to perform joins in SQL. we already have table Employee, we will now create table Department

Id	Name	department	Location
1	Bella	HR	New York
2	Cece	IT	London
3	Cody	Finance	Tokyo
4	Nico	HR	Paris
5	Emma	Marketing	New York

7.1.1 Inner Join:

- An inner join returns only the rows that have matching values in both tables based on the specified column.
- It filters out unmatched rows from both tables.
- It's like a Venn diagram intersection, where only the overlapping area is returned.
- Syntax:

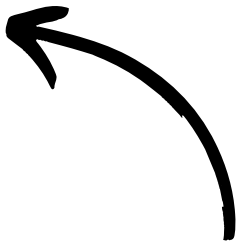
SELECT columns FROM table1 INNER JOIN table2 ON table1.column = table2.column;

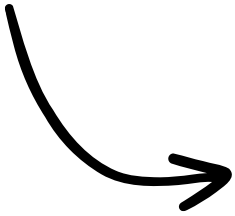



```
-- Joining the employee table with a department table to get employees along with their department

SELECT employees.name, employees.department, department.location
FROM employees
INNER JOIN department ON employees.department = department.name;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000





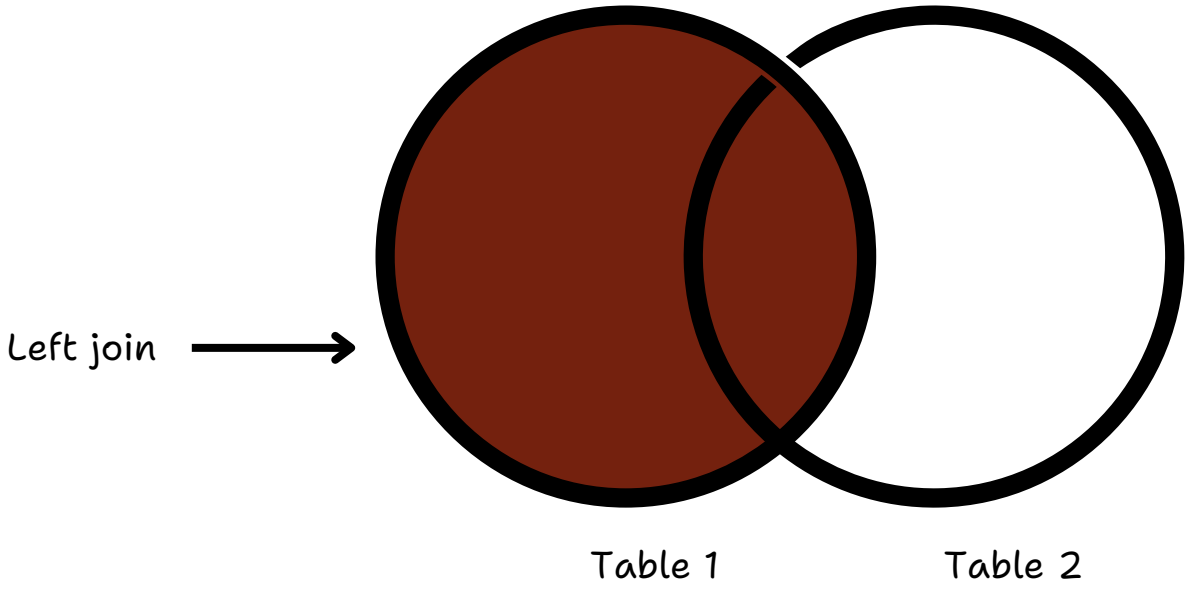
Id	Name	department	Location
1	Bella	HR	New York
2	Cece	IT	London
3	Cody	Finance	Tokyo
4	Nico	HR	Paris
5	Emma	Marketing	New York

Id	name	department	Location
1	Bella	HR	New York
2	Cece	IT	London
3	Cody	Finance	Tokyo
4	Nico	HR	Paris
5	Emma	Marketing	New York

7.1.2 Left Join:

- left join returns all rows from the left table and the matched rows from the right table.
- If there's no match in the right table, NULL values are returned for the columns of the right table.
- It ensures that all rows from the left table are included in the result set.
- Syntax:

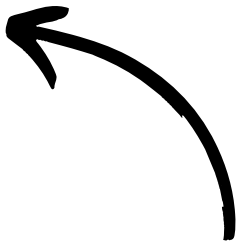
SELECT columns FROM table1 LEFT JOIN table2 ON table1.column = table2.column;

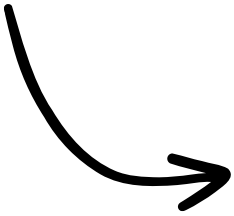



```
-- Retrieving all employees along with their department information, including those without a department

SELECT employees.name, employees.department, department.location
FROM employees
LEFT JOIN department ON employees.department = department.name;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000





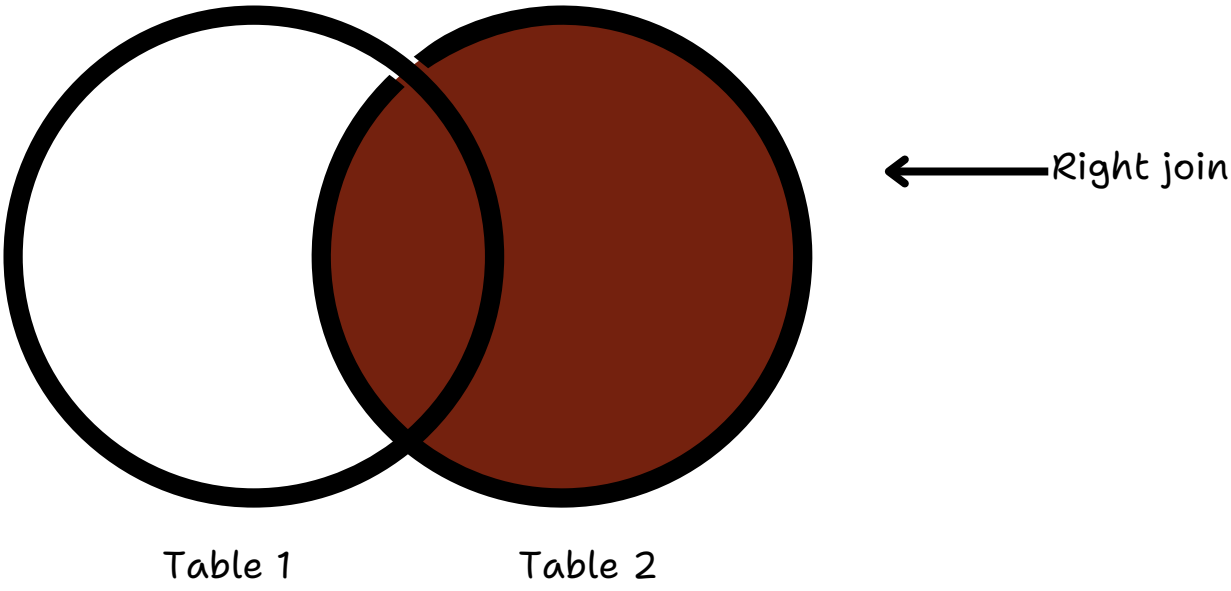
Id	Name	department	Location
1	Bella	HR	New York
2	Cece	IT	London
3	Cody	Finance	Tokyo
4	Nico	HR	Paris
5	Emma	Marketing	New York

Id	name	age	department	Salary	Location
1	Bella	25	HR	60000	New York
2	Cece	30	IT	45000	London
3	Cody	35	Finance	55000	Tokyo
4	Nico	40	HR	60000	Paris
5	Emma	28	Marketing	25000	New York

7.1.3 Right Join:

- A right join returns all rows from the right table and the matched rows from the left table.
- If there's no match in the left table, NULL values are returned for the columns of the left table.
- It ensures that all rows from the right table are included in the result set.
- Syntax:

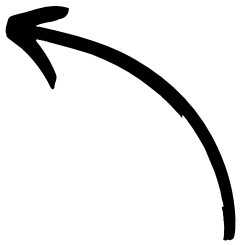
SELECT columns FROM table1 RIGHT JOIN table2 ON table1.column = table2.column;

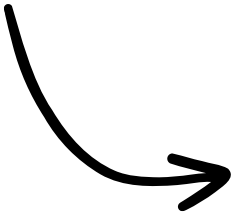



```sql
-- Retrieving all departments along with their employee information, including departments without employees

SELECT employees.name, employees.department, department.location
FROM employees
RIGHT JOIN department ON employees.department = department.name;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000



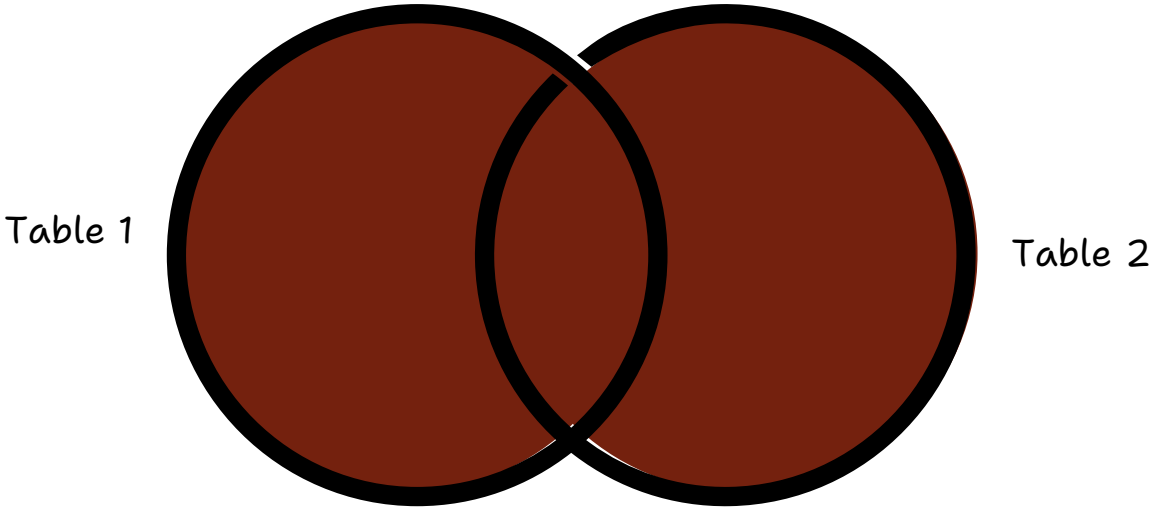


Id	Name	department	Location
1	Bella	HR	New York
2	Cece	IT	London
3	Cody	Finance	Tokyo
4	Nico	HR	Paris
5	Emma	Marketing	New York

Id	name	department	Location	age	Salary
1	Bella	HR	New York	25	60000
2	Cece	IT	London	30	45000
3	Cody	Finance	Tokyo	35	55000
4	Nico	HR	Paris	40	60000
5	Emma	Marketing	New York	28	25000

7.1.4 Full Join (or Full Outer Join):

- A full join, also known as a full outer join, combines the results of both left and right outer joins. It includes all rows from both tables and fills in NULL values for missing matches on either side.
- Think of it as a combination of left join and right join. It brings in all rows from both tables, matching them where possible, and adding NULL values where there are no matches.

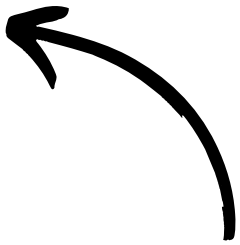


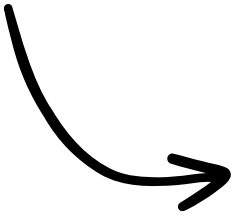
Full Outer join

```
-- Retrieving all departments along with their employee information, including departments without employees

SELECT employees.name, employees.department, department.location
FROM employees
RIGHT JOIN department ON employees.department = department.name;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000





Id	Name	department	Location
1	Bella	HR	New York
2	Cece	IT	London
3	Cody	Finance	Tokyo
4	Nico	HR	Paris
5	Emma	Marketing	New York

Id	name	age	department	Salary	Location
1	Bella	25	HR	60000	New York
2	Cece	30	IT	45000	London
3	Cody	35	Finance	55000	Tokyo
4	Nico	40	HR	60000	Paris
5	Emma	28	Marketing	25000	New York

Chapter 8: Data Modification

In SQL, data modification statements are used to add, update, or remove records from a database table. These statements allow you to make changes to the data stored in the tables, enabling you to manage and maintain the integrity of your database.

Data modification statements are essential for managing the contents of database tables. Whether you need to add new records, update existing ones, or remove unwanted data, these statements provide the means to keep your database accurate and up-to-date. It's important to use them carefully to ensure the integrity and consistency of your data.

8.1 INSERT Statement:

- Adds new records to a table.
- Syntax:
  INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);

```
-- Inserting a new employee record into the employees table

INSERT INTO employees (name, age, department, salary) VALUES ('John', 27,IT,37000);
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000
6	John	27	IT	37000

8.2 UPDATE Statement:

- Modifies existing records in a table.
- Syntax:

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

```
-- Updating the age of an employee with id 2

UPDATE employees SET age = 31 WHERE id = 2;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	31	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

8.3 DELETE Statement:

- Removes one or more records from a table.
- Syntax:

DELETE FROM table_name WHERE condition;

```
-- Deleting an employee record with id 3

DELETE FROM employees WHERE id = 3;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

8.4 TRUNCATE Statement:

- The **TRUNCATE** statement in SQL is used to delete all rows from a table, while still maintaining the table structure. Unlike the **DELETE** statement, which removes rows one by one and generates individual **DELETE** operations for each row, **TRUNCATE** operates more efficiently by deallocating the data pages of the table, resulting in faster performance especially for large tables.
- Syntax:

  TRUNCATE TABLE table_name;

```
-- Deleting the table employee but keeping the structure

TRUNCATE TABLE employees;
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

Id	name	age	department	Salary
1				
2				
4				
5				

After executing the TRUNCATE statement, the "employees" table will have no rows, but its structure will remain intact.

Chapter 9: Set Operations

Set operations in SQL allow you to combine the results of multiple queries into a single result set. These operations include union, union all, intersect, and except (or minus), each providing a different way to manipulate and combine data from multiple queries.

Consider two tables: "employees" and "managers."

employees

Id	name	department
1	Bella	HR
2	Cece	IT
3	Cody	Finance

managers

Id	name	department
2	Cece	IT
4	Emma	Marketing

9.1 UNION:

- Combines the results of two or more SELECT statements into a single result set, removing duplicates.
- Syntax:

SELECT column1, column2, ... FROM table1 UNION SELECT column1, column2, ... FROM table2;

```
-- Combining the names of employees and managers, removing duplicates

SELECT name FROM employees
UNION
SELECT name FROM managers;
```

Id	name	department
1	Bella	HR
2	Cece	IT
3	Cody	Finance

Id	name	department
2	Cece	IT
4	Emma	Marketing

name
Bella
Cece
Cody
Emma

9.2 UNION ALL:

- Combines the results of two or more SELECT statements into a single result set, including duplicates.
- Syntax:

SELECT column1, column2, ... FROM table1 UNION ALL SELECT column1, column2, ... FROM table2;

```
-- Combining the names of employees and managers, removing duplicates

SELECT name FROM employees
UNION ALL
SELECT name FROM managers;
```

Id	name	department
1	Bella	HR
2	Cece	IT
3	Cody	Finance

Id	name	department
2	Cece	IT
4	Emma	Marketing

name
Bella
Cece
Cody
Cece
Emma

9.3 INTERSECT:

- Returns the common rows between two result sets, removing duplicates.
- Syntax:

SELECT column1, column2, ... FROM table1 INTERSECT SELECT column1, column2, ... FROM table2;

```
-- Combining the names of employees and managers, removing duplicates

SELECT name FROM employees
INTERSECT
SELECT name FROM managers;
```

Id	name	department
1	Bella	HR
2	Cece	IT
3	Cody	Finance

Id	name	department
2	Cece	IT
4	Emma	Marketing

name
Cece

9.4 EXCEPT:

- Returns the rows from the first result set that are not present in the second result set, removing duplicates.
- Syntax:

SELECT column1, column2, ... FROM table1 EXCEPT SELECT column1, column2, ... FROM table2;

```
-- Combining the names of employees and managers, removing duplicates

SELECT name FROM employees
EXCEPT
SELECT name FROM managers;
```

Id	name	department
1	Bella	HR
2	Cece	IT
3	Cody	Finance

Id	name	department
2	Cece	IT
4	Emma	Marketing

name
Bella
Cody

Chapter 10: Constraints

Constraints in SQL are rules that are enforced on the data stored in tables. They help maintain the integrity, accuracy, and consistency of the data by imposing certain conditions or restrictions on the values that can be inserted, updated, or deleted in a table.

10.1 NOT NULL Constraint:

- Ensures that a column cannot contain NULL values.
- It enforces the requirement that every row must have a value for that column.
- Syntax:

  column_name data_type NOT NULL

```
-- Creating the employees table with a NOT NULL constraint on the name column
CREATE TABLE employees (
    id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    age INT,
    department_id INT
);
```

10.2 UNIQUE Constraint:

- Ensures that all values in a column (or a set of columns) are unique.
- It prevents duplicate values from being inserted into the column(s).
- Syntax:

  UNIQUE (column_name)

```
-- Adding a UNIQUE constraint on the department_id column

ALTER TABLE employees
ADD CONSTRAINT unique_department_id UNIQUE (department_id);
```

10.3 PRIMARY KEY Constraint:

- Uniquely identifies each record in a table.
- It combines the NOT NULL and UNIQUE constraints.
- Only one primary key is allowed per table.
- Syntax:

  PRIMARY KEY (column_name)

```
-- Adding a PRIMARY KEY constraint on the id column

ALTER TABLE employees
ADD CONSTRAINT pk_id PRIMARY KEY (id);
```

10.4 FOREIGN KEY Constraint:

- Establishes a relationship between two tables.
- It ensures referential integrity by enforcing a link between the data in the referencing table and the referenced table.
- Syntax:

  FOREIGN KEY (column_name) REFERENCES other_table(column_name)

```
-- Adding a FOREIGN KEY constraint on the department_id column referencing the departments table

ALTER TABLE employees
ADD CONSTRAINT fk_department_id FOREIGN KEY (department_id) REFERENCES departments(id);
```

Chapter 11: Subqueries

Subqueries, also known as nested queries or inner queries, are queries nested within another SQL query.
They allow you to use the result of one query as a condition or filter in another query. Subqueries can be used to retrieve data from one or more tables and can be classified into different types based on their behavior and result set.

1. Single Row Subquery
2. Multiple Row Subquery
3. Correlated Subquery

Consider the below table for example:

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

11.1 Single Row Subquery:

- A single row subquery returns only one row of result.
- It is used to compare a single value with the result of a subquery.
- Syntax:

SELECT column FROM table WHERE column = (SELECT column FROM table);

```
-- Retrieving employees with salary greater than the average salary

SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

name	Salary
Bella	60000
Nico	60000

11.2 Multiple Row Subquery:

- A multiple row subquery returns multiple rows of result.
- It is used with operators like IN, ANY, ALL to compare with the result of a subquery.
- Syntax:

  SELECT column FROM table WHERE column IN (SELECT column FROM table);

```sql
-- Retrieving employees from the IT department

SELECT name
FROM employees
WHERE department_id IN (SELECT id FROM departments WHERE name = 'IT');
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

name
Cece

11.3 Correlated Subquery:

- A correlated subquery is a subquery that references one or more columns from the outer query.
- It executes once for each row processed by the outer query.
- Syntax:

SELECT column FROM table WHERE column = (SELECT column FROM table WHERE table.column = outer_table.column);

```
-- Retrieving employees with salaries greater than their department's average salary

SELECT name, salary
FROM employees e
WHERE salary > (SELECT AVG(salary) FROM employees WHERE department_id = e.department_id);
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

name	Salary
Bella	60000
Nico	60000

Chapter 12: Views

In SQL, a view is a virtual table based on the result set of a SELECT statement. Views allow you to simplify complex queries, encapsulate logic, and provide a layer of abstraction over the underlying tables. They are useful for presenting data in a specific format, restricting access to certain columns, or simplifying data manipulation tasks.

12.1 Creating Views:

- Views are created using the CREATE VIEW statement.
- They can be based on one or more tables or other views.
- Syntax:

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

```
-- Creating a view to display employee details

CREATE VIEW employee_details AS
SELECT name, age, department
FROM employees;
```

Id	name	age	department
1	Bella	25	HR
2	Cece	30	IT
3	Cody	35	Finance
4	Nico	40	HR
5	Emma	28	Marketing

12.2 Updating Views:

- Views can be updated using the CREATE OR REPLACE VIEW statement.
- You can modify the SELECT query of an existing view.
- Syntax:

**CREATE OR REPLACE VIEW view_name AS SELECT new_column1, new_column2, ...
FROM table_name WHERE new_condition;**

```
-- Modifying the view to include salary information

CREATE OR REPLACE VIEW employee_details AS
SELECT name, age, department, salary
FROM employees:
```

Id	name	age	department	Salary
1	Bella	25	HR	60000
2	Cece	30	IT	45000
3	Cody	35	Finance	55000
4	Nico	40	HR	60000
5	Emma	28	Marketing	25000

12.3 Dropping Views:

- Views can be dropped (deleted) from the database using the DROP VIEW statement.
- Once dropped, the view and its definition are removed from the database.
- Syntax:

  DROP VIEW view_name;

```
-- Dropping the employee_details view

DROP VIEW employee_details;
```

View dropped successfully.

Chapter 13: Transactions

In database management systems, transactions are sets of operations that are executed as a single unit of work. Transactions ensure data integrity and consistency by following the ACID properties (Atomicity, Consistency, Isolation, Durability)

13.1 Dropping Views:

ACID is an acronym for Atomicity, Consistency, Isolation, and Durability. These properties ensure that database transactions are reliable, consistent, and maintain data integrity.

13.1.1 Atomicity:

- Atomicity ensures that a transaction is treated as a single unit of work, meaning either all of its operations are successfully completed or none of them are.
- If any part of the transaction fails, the entire transaction is rolled back to its original state.
- This property helps maintain data integrity and prevents the database from being left in an inconsistent state.

13.1.2 Consistency:

- Consistency ensures that the database remains in a consistent state before and after the transaction.
- All data modifications must adhere to the rules and constraints defined by the database schema.
- If a transaction violates any constraints, the database remains unchanged, and the transaction is rolled back.

13.1.3 Isolation:

- Isolation ensures that the intermediate states of a transaction are invisible to other transactions until the transaction is committed.
- Transactions should not interfere with each other, even if they are executed concurrently.
- Isolation levels determine the degree to which transactions are isolated from each other.

13.1.4 Durability:

- Durability guarantees that the effects of a committed transaction persist even in the event of system failures.
- Once a transaction is committed, its changes are permanently stored in the database and are not lost, even if the system crashes or restarts.

Example:

Consider a banking system where a user transfers funds from one account to another.

```
-- Begin transaction
BEGIN TRANSACTION;

-- Deduct amount from sender's account
UPDATE accounts SET balance = balance - 100 WHERE account_id =
'sender_account_id';

-- Add amount to receiver's account
UPDATE accounts SET balance = balance + 100 WHERE account_id =
'receiver_account_id';

-- Commit transaction
COMMIT;
```

- If both UPDATE statements execute successfully, the changes are committed to the database, and the transaction is complete.
- If any error occurs during the transaction (e.g., insufficient funds, network failure), the changes are rolled back, and the database remains unchanged.
- This ensures that either the entire transfer operation is completed successfully, or no changes are made to the accounts, maintaining the atomicity and consistency of the database.

Chapter 14: Normalization

Normalization is the process of organizing data in a database efficiently. It involves splitting up large tables into smaller, related tables and defining relationships between them. The goal of normalization is to reduce data redundancy and dependency, ensuring data integrity and minimizing the risk of anomalies.

Normalization is like organizing your clothes in a wardrobe. Instead of tossing everything in one big pile, you separate them into smaller categories. For example, shirts go in one drawer, pants in another, and socks in another. This way, it's easier to find what you need, and you avoid having duplicates scattered around.

Similarly, in a database, normalization involves breaking down large tables into smaller ones based on related information. Each table holds specific types of data, and relationships between tables are defined to link them together. By organizing data this way, redundancy (having the same information repeated unnecessarily) is reduced, and the chances of errors or inconsistencies are minimized. It's like tidying up your database to make it more efficient and reliable.

Consider below table named "orders" for normalization:

order_id	customer_id	customer_name	product_id	product_name	quantity	price
1	101	John	1	Laptop	2	1500
2	102	Alice	2	Smartphone	1	1000
3	101	John	3	Tablet	3	800

14.1 1NF (First Normal Form):

- 1NF requires that each column in a table contains atomic values, meaning values are indivisible or cannot be further subdivided.
- It eliminates repeating groups within rows and ensures each cell contains a single value.
- Example: Breaking up a table of employees into separate tables for employee details and employee addresses.

To convert this table into 1NF, we need to ensure that each column contains atomic values and eliminate repeating groups within rows.

order_id	customer_id	product_id	quantity	price
1	101	1	2	1500
2	102	2	1	1000
3	101	3	3	800

14.2 2NF (Second Normal Form):

- 2NF builds on 1NF and requires that a table be in 1NF and that all non-key attributes are fully functional dependent on the entire primary key.
- It eliminates partial dependencies by moving subsets of data into separate tables.
- Example: Splitting a table of orders into separate tables for order details and customer information.

To convert this table into 2NF, we need to ensure that all non-key attributes are fully functional dependent on the entire primary key.

Orders Table:

order_id	customer_id	quantity	price
1	101	2	1500
2	102	1	1000
3	101	3	800

Products Table:

product_id	product_name
1	Laptop
2	Smartphone
3	Tablet

14.3. 3NF (Third Normal Form):

- 3NF builds on 2NF and requires that a table be in 2NF and that all non-key attributes are transitively dependent on the primary key.
- It eliminates transitive dependencies by removing fields that are not dependent on the primary key.
- Example: Splitting a table of courses into separate tables for course information and instructor details.

To convert this table into 3NF, we need to ensure that all non-key attributes are transitively dependent on the primary key.

Orders Table:

order_id	customer_id	product_id	quantity	price
1	101	1	2	1500
2	102	2	1	1000
3	101	3	3	800

Customers Table:

customer_id	customer_name
101	John
102	Alice
101	John

Products Table:

product_id	product_name
1	Laptop
2	Smartphone
3	Tablet