Relational database Concepts and limitations

Benmounah Zakaria
Constantine 2 University
SDIA Master 1
2023-2024

PART 01

A CORE CONCEPTS OF RELATIONAL DATABASES

TUPLE AND ATTRIBUTE

- A tuple is a single row in a table representing a record, and an attribute is a column holding specific pieces of information.
- Example: In a table named Students, each row is a tuple, and each column is an attribute.

student_id	first_name	last_name	age
1	John	Doe	20
2	Jane	Smith	22
3	Mike	Johnson	21

- Here, each row (e.g., | 1 | John | Doe | 20 |) is considered a tuple.
- Each column (e.g., student_id, first_name, last_name, age) is considered an attribute.

SQL Query Examples

- Here’s a SQL query to create the above Students table:

```
CREATE TABLE Students (
 student_id INT PRIMARY KEY,
 first_name VARCHAR(50),
 last_name VARCHAR(50),
 age INT
);
```

- And here’s a SQL query to insert a tuple into the Students table:

```
INSERT INTO Students (student_id, first_name, last_name, age)
VALUES (1, 'John', 'Doe', 20);
```

- This SQL query will select all tuples from the Students table where the age attribute is 21:

```
SELECT * FROM Students WHERE age = 21;
```

DOMAIN

- The domain is a set of permissible values for a given attribute, defining the allowable data type and format, and ensuring data integrity and consistency within the attribute.
  (For example, if you have a column for storing ages, the domain of that column would be non-negative integers.)

```
CREATE TABLE Students (
 student_id INT PRIMARY KEY,
 first_name VARCHAR(50),
 last_name VARCHAR(50),
 age INT CHECK (age >= 0 AND age <= 150)
);
```

- The age column in a Students table has a domain defined by a CHECK constraint allowing only non-negative integers less than or equal to 150.
- This domain ensures that any value for age inserted into the Students table conforms to logical rules, maintaining the integrity of the data stored within the table.

DOMAIN Violating Error

In SQL Server

```
Msg 547, Level 16, State 0, Line 1 The INSERT statement conflicted with the CHECK constraint
"constraint name". The conflict occurredin database "database_name", table
"schema_name.table_name", column 'column_name'.
```

In PostgreSQL Server

```
 ERROR: new row for relation "table name" violates check constraint
"constraint_name" DETAIL: Failing row contains (value1, value2, ...).
```

ATOMIC AND NONATOMIC

- Atomic values are indivisible and stored in a single field, while nonatomic values can be divided into simpler pieces.
- Better Practice: A more normalized approach is to use a separate table **to represent multi-valued attributes**, avoiding nonatomic values in fields.
- An atomic value is typically represented by a simple data type like integer, character, or string, and it is stored in a single field of a database table.

ATOMIC Value EXAMPLE

- An atomic value is typically represented by a simple data type like integer, character, or string, and it is stored in a single field of a database table.

```
CREATE TABLE Employees (
 employee_id INT PRIMARY KEY,
 first_name VARCHAR(50),
 last_name VARCHAR(50),
 salary INT
);

INSERT INTO Employees (employee_id, first_name, last_name, salary)
VALUES (1, 'John', 'Doe', 50000);
```

- In this example, values like 1, 'John', 'Doe', and 50000 are all atomic values, as they cannot be further decomposed.

NONATOMIC Value EXAMPLE

- Nonatomic values are often represented by data structures like arrays, lists, or other complex types.

```
CREATE TABLE Projects (
 project_id INT PRIMARY KEY,
 project_name VARCHAR(50),
 team_members VARCHAR(255)
);

INSERT INTO Projects (project_id, project_name, team_members)
VALUES (1, 'ProjectA', '101,102,103');
```

- The team_members field holds a nonatomic value '101,102,103', which is a comma-separated string representing a list of team member IDs. **Ideally, a separate relational table should be used to model many-to-many relationships like this, avoiding nonatomic values in fields.**

NULL

- NULL represents missing, undefined, or unknown data.
- Special attention is needed when querying data with NULL values, and SQL provides functions like COALESCE() and NULLIF() to deal with them.
- Consider a Students table with the following columns: student_id, first_name, last_name, and age:

```
CREATE TABLE Students (
 student_id INT PRIMARY KEY,
 first_name VARCHAR(50) NOT NULL,
 last_name VARCHAR(50) NOT NULL,
 age INT
);
```

- In this example, first_name and last_name are marked as NOT NULL, meaning they cannot have NULL values, while the age column can contain NULL values.

NULL

- If we insert two rows:

```sql
INSERT INTO Students (student_id, first_name, last_name, age)
VALUES (1, 'John', 'Doe', NULL),
       (2, 'Jane', 'Smith', 25);
```

- In this case, John Doe’s age is unknown, so it is inserted as NULL, whereas Jane Smith’s age is 25.

Querying with NULL values

- As NULL is not equal to any value, even another NULL.
- So, if we want to filter rows where the age is NULL, we need to use IS NULL:
  ```
  SELECT * FROM Students WHERE age IS NULL;
  ```
- This will return all rows where the age is NULL.
- Often, we may need to handle NULL values in your queries. SQL provides the COALESCE() function and the NULLIF() function to deal with NULL values:
  ```
  SELECT first_name, last_name, COALESCE(age, 'Unknown') as age FROM Students;
  ```
- This will return the age for students where it is not NULL and ‘Unknown’ where it is NULL.

RELATION SCHEMA

- A relation schema defines the structure of a relation, and a relation instance is a set of tuples adhering to the defined schema:

```
Students(student_id: INT, first_name: VARCHAR, last_name: VARCHAR, age: INT)
```

- The relation schema for a Students table specifies the attributes and their data types, and a relation instance represents the actual data in the table.

RELATION INSTANCE

- Given the above schema, a relation instance (or table instance) of Students might look like this:

```
| student_id | first_name | last_name | age |
| 1          | John       | Doe       | 20  |
| 2          | Jane       | Smith     | 22  |
| 3          | Mike       | Johnson   | 21  |
```

- In this instance, the Students table has three tuples (or rows) each containing data that conforms to the relation schema.

KEYS: SUPERKEY

- Keys are used to uniquely identify rows and establish relationships between tables.
- Superkey is a set of one or more columns that uniquely identify all rows,
- In a Students table with columns: `student_id`, `first_name`, `last_name`, any combination of columns that includes **`student_id`** would be a superkey, like **`{student_id}`**, **`{student_id, first_name}`**, **`{student_id, last_name}`**, **`{student_id, first_name, last_name}`**, etc.

KEYS: CANDIDATE KEY

- A candidate key is a minimal superkey, meaning it has the least number of columns required to uniquely identify each row in a table. It is a subset of a superkey.
- In the same `Students` table, `{student_id}` would be a candidate key as it is the minimal set of columns that can uniquely identify each row.

KEYS: PRIMARY KEY

- Primary Key is one of the candidate keys chosen to uniquely identify each row.
- In the `Students` table, `student_id` can be chosen as the primary key.

EXAMPLE

```sql
CREATE TABLE Students (
 student_id INT,
 first_name VARCHAR(50),
 last_name VARCHAR(50),
 age INT,
 PRIMARY KEY (student_id)
);
```

- **Superkey:** `{student_id}`, `{student_id, first_name}`, `{student_id, last_name}`, `{student_id, first_name, last_name}` are all examples of superkeys.
- **Candidate Key:** `{student_id}` is a candidate key as it is the minimal superkey.
- **Primary Key:** `student_id` is chosen as the primary key for the Students table.

CONCLUSION

- We covered the core concepts of relational databases, including tuple, attribute, domain, atomic and nonatomic values, null, relation schema and instance, and keys.
- Understanding these concepts is crucial for effective database design and management.

PART 02

ACID Properties

Introduction

- In SQL, the ACID properties (**Atomicity**, **Consistency**, **Isolation**, **Durability**) ensure reliability in every transaction performed by the database management system.

1- Atomicity

- Ensures that a transaction (s) is treated as a single, indivisible unit, either completing in its entirety or having no effect at all.

- Atomicity ensures that a database transaction is treated as a single, indivisible, logical unit of work, which either completes entirely or has no effect at all. If any part of the transaction fails, the entire transaction is rolled back, and the database state remains unchanged.

Example

- Suppose we have two bank accounts in a simple BankAccount table:

```sql
CREATE TABLE BankAccount (
 account_id INT PRIMARY KEY,
 balance DECIMAL(10, 2)
);

-- Insert two rows representing two accounts.
INSERT INTO BankAccount (account_id, balance) VALUES (1, 1000.00);
INSERT INTO BankAccount (account_id, balance) VALUES (2, 1000.00);
```

Problem

- Now, let's create a transaction to transfer 500.00 from account 1 to account 2:

```
START TRANSACTION;

-- Step 1: Withdraw 500.00 from account 1
UPDATE BankAccount SET balance = balance - 500.00 WHERE account_id = 1;

-- Assume here that a failure occurs, like a power outage or crash.

-- Step 2: Deposit 500.00 to account 2
UPDATE BankAccount SET balance = balance + 500.00 WHERE account_id = 2;

COMMIT;
```

- In this example, **if a failure occurs between the two UPDATE statements**, the transaction would be incomplete, leading to an inconsistent state where money has been withdrawn from account 1 but not deposited to account 2.

Solution: Applying Atomicity

- To enforce atomicity, typically we use transaction control statements like START TRANSACTION, COMMIT, and ROLLBACK.

- If any error occurs before the COMMIT, we have to issue a ROLLBACK statement to undo any changes made to the database during the transaction:

```
START TRANSACTION;
BEGIN
….
END;
```

Example: Transfer and balance test

- If any step of the transaction fails (e.g., due to an insufficient balance in account 1), a ROLLBACK would be issued, and the entire transaction would be undone, ensuring the atomicity of the transaction.

```
START TRANSACTION;

BEGIN
 -- Attempt to Withdraw 500.00 from account 1
 UPDATE BankAccount SET balance = balance - 500.00 WHERE account_id = 1;

 -- Simulate a condition that causes a failure (e.g. insufficient funds)
 IF (SELECT balance FROM BankAccount WHERE account_id = 1) < 0 THEN
 ROLLBACK;
 RETURN;
 END IF;

 -- Deposit 500.00 to account 2
 UPDATE BankAccount SET balance = balance + 500.00 WHERE account_id = 2;
END;
COMMIT;
```

2- Consistency

- Consistency refers to ensuring that a transaction brings the database from one consistent state to another,

- Maintaining the integrity constraints and business rules defined in the database.

- If a transaction violates any integrity constraints or rules, it is rolled back to avoid corrupting the database state.

Enforcing Consistency

- Consider a BankAccount table, and let's say the business rule is that the balance of a bank account should never be negative (This is a consistency requirement).
- Here's the SQL to create a BankAccount table with a check constraint to enforce the consistency requirement:

Enforcing Consistency

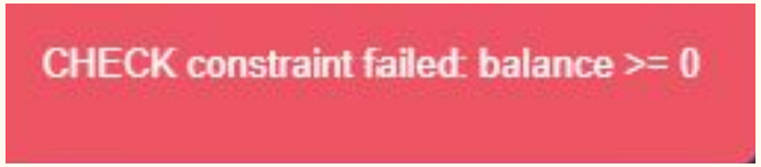
```
CREATE TABLE BankAccount (
 account_id INT PRIMARY KEY,
 balance DECIMAL(10, 2) CHECK (balance >= 0) -- This ensures consistency by preventing
negative balances.
);

-- Insert initial data
INSERT INTO BankAccount (account_id, balance) VALUES (1, 1000.00);
```

- let’s attempt a transaction that tries to debit an amount from the bank account that would violate the check constraint and lead to an inconsistent state:

```
START TRANSACTION;
-- Attempt to withdraw 1500.00 from account 1
UPDATE BankAccount SET balance = balance - 1500.00 WHERE account_id = 1;

COMMIT;
```


CHECK constraint failed: balance >= 0

Enforcing Consistency

- In the previous example since the balance would become -500.00, which violates the CHECK constraint, the database system will roll back the transaction automatically, maintaining the consistency of the database.

- By utilizing constraints like CHECK, UNIQUE, and FOREIGN KEY, we can enforce consistency in our database, ensuring that the business rules and integrity constraints are not violated by any transaction.

3 Isolation

- Provides a mechanism where multiple transactions occurring simultaneously do not impact each other’s execution.
- It ensures that the concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.
- This means the intermediate state of a transaction is invisible to other transactions, and transactions appear to be executed in isolation.

Example: transfer Money vs Check Balance

- Consider two transactions: one is transferring money between two accounts, and another is calculating the total balance in the bank.
- Let's assume we have a BankAccount table with the following initial data:

account_id	balance
1	1000.00
2	1000.00

```sql
-- Transfer money --
START TRANSACTION;
UPDATE BankAccount SET balance = balance - 100 WHERE account_id = 1; -- Deduct from account 1
UPDATE BankAccount SET balance = balance + 100 WHERE account_id = 2; -- Add to account 2
COMMIT;

-- Check Balance --
SELECT SUM(balance) AS total_balance FROM BankAccount;
```

Scenarios

- **Scenario without Isolation:** If Transaction 1 and Transaction 2 are executed concurrently without isolation, Transaction 2 might calculate the total_balance when the money has been deducted from account 1 but not yet added to account 2, leading to incorrect results.
- **Scenario with Isolation:** With proper isolation, Transaction 2 will either see the balance before the start of Transaction 1 or after the completion of Transaction 1 but not *in the intermediate state*. Therefore, it will always calculate the correct `total_balance`.

Implementing Isolation

- READ UNCOMMITTED: This level allows dirty reads, which means a transaction may see uncommitted changes from another transaction.
- READ COMMITTED: This level doesn't allow dirty reads, which means a transaction may only see committed changes from other transactions.
- REPEATABLE READ: This level ensures that if a transaction reads a row, no other transaction can modify or delete that row until the first transaction completes.
- SERIALIZABLE: This is the highest isolation level, where transactions are executed in such a way that the result is equivalent to executing them serially, one after the other.

- In SQL, we can set the isolation level using the SET TRANSACTION statement like this:
-
```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

4 Durability

- Assures that once a transaction is committed, the changes are permanent and can withstand any subsequent failures or system crashes.

- It guarantees that once a transaction is committed, it will survive permanently and will not be undone, even in the event of system failures, crashes, or restarts. This is typically achieved through the use of persistent storage mechanisms and transaction logs.

Example

- Let's assume you have a BankAccount table, and you are performing a transaction to transfer money between two accounts.

Initial Data:

```
CREATE TABLE BankAccount (
 account_id INT PRIMARY KEY,
 balance DECIMAL(10,2)
);

INSERT INTO BankAccount VALUES (1, 1000.00), (2, 500.00);
```

Transaction:

```
START TRANSACTION;
UPDATE BankAccount SET balance = balance - 200.00 WHERE account_id = 1; -- Deduct 200 from
account 1
UPDATE BankAccount SET balance = balance + 200.00 WHERE account_id = 2; -- Add 200 to account 2
COMMIT;
```

Durability in Action

- Now, after executing the previous Example, let's say the system suddenly crashes or there is a power failure.
- When the system recovers, the changes made by the transaction (i.e., transferring 200 from account 1 to account 2) must still be in place,
- The balances should reflect the post-transaction state, demonstrating the durability property.
- The database management system typically uses write-ahead logging (WAL) and other mechanisms to ensure that committed transactions are saved to persistent storage, ensuring durability.

PART 03

Normalization and Denormalization

Introduction

- The concepts of normalization and denormalization are pivotal,
- Each serving its unique purpose in optimizing database performance and ensuring data integrity.

Normalization

- Normalization is a systematic approach used in relational database design to **reduce data redundancy** and **avoid undesirable characteristics** like insertion, update, and deletion anomalies.
- It involves **decomposing** a table into **less redundant** (and smaller) tables but without losing information; defining foreign keys in the old tables referencing the primary keys of the new ones.
- The main goal of normalization is to add, delete, and modify data without causing data anomalies.

Techniques of Normalization

Normalization is performed in several stages, each referred to as a “normal form.” The most commonly used normal forms are

First Normal Form (1NF):
- Ensures each column contains atomic (indivisible) values and values in each column are of the same data type.

Second Normal Form (2NF):
- All requirements of 1NF are met, and all non-key attributes are fully functionally dependent on the primary key.

Third Normal Form (3NF):
- Meets all requirements of 2NF, and all the attributes are functionally dependent only on the primary key.

Example: Unnormalized Table

```
| StudentID | StudentName | Courses              | Professor          |
| 1         | John Doe    | Math, English        | Mr. Smith, Ms. Lee |
| 2         | Jane Smith  | Science, Math        | Dr. Brown, Mr. Smith|
```

1NF: First Normal Form

To satisfy 1NF:

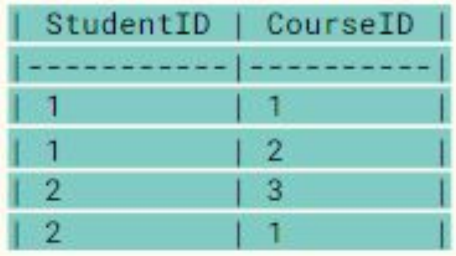
- The table must have a primary key.
- All columns should contain atomic (indivisible) values.
- We will split the courses and professors into separate rows:

StudentID	StudentName	Course	Professor
1	John Doe	Math	Mr. Smith
1	John Doe	English	Ms. Lee
2	Jane Smith	Science	Dr. Brown
2	Jane Smith	Math	Mr. Smith

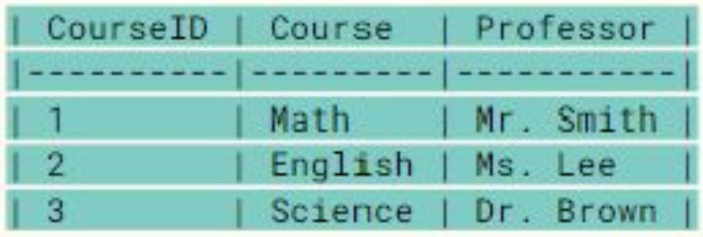
2NF: Second Normal Form

To satisfy 2NF:

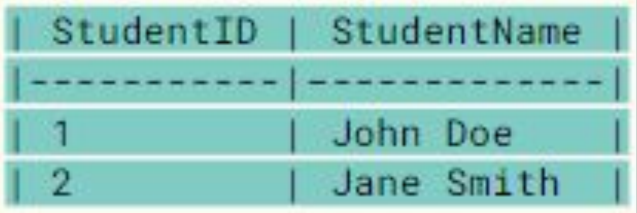
- Table should be in 1NF.
- Remove subsets of data that apply to multiple rows and place them in separate tables.
- We will move the courses and professors to a new table and reference them using a foreign key:



StudentID	CourseID
1	1
1	2
2	3
2	1



CourseID	Course	Professor
1	Math	Mr. Smith
2	English	Ms. Lee
3	Science	Dr. Brown

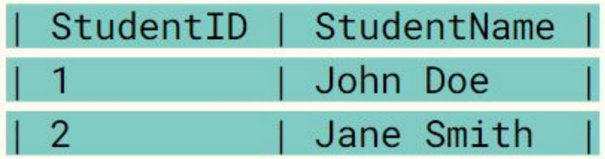


StudentID	StudentName
1	John Doe
2	Jane Smith

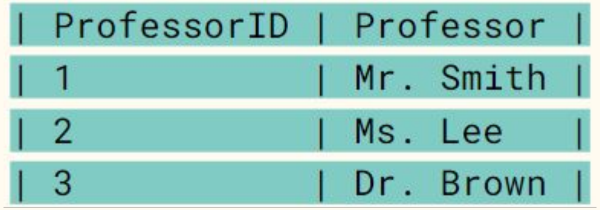
3NF: Third Normal Form

To satisfy 3NF:

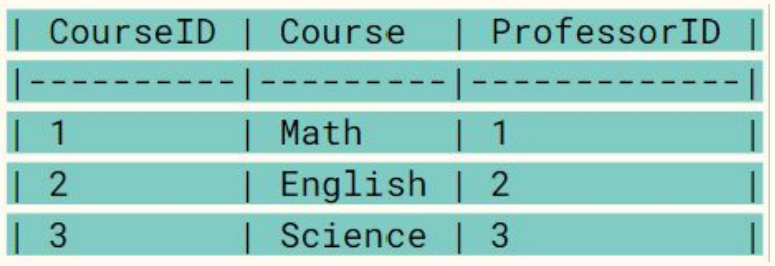
- Table should be in 2NF.
- There should be no transitive functional dependency.
- The Professor is dependent on Course, not on the student, so we create a new table for Professor and reference it in the Courses table.



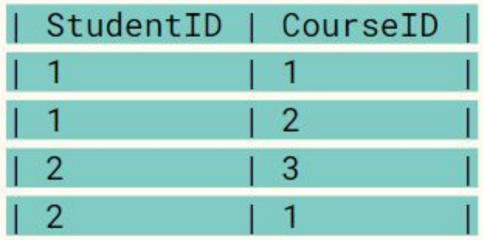
StudentID	StudentName
1	John Doe
2	Jane Smith



ProfessorID	Professor
1	Mr. Smith
2	Ms. Lee
3	Dr. Brown



CourseID	Course	ProfessorID
1	Math	1
2	English	2
3	Science	3



StudentID	CourseID
1	1
1	2
2	3
2	1

Denormalization

- Denormalization is the process of adding redundancy to a database by incorporating redundant data where one table references another.
- It is essentially the opposite of normalization and is used to improve the read performance of a database in certain situations.
- Denormalization is considered when complex joins and queries are affecting the performance of the read-intensive database.

When to Consider Denormalization

Denormalization is not a one-size-fits-all solution and should be considered carefully. It is typically used when:

1. Read Performance:
   - The system is read-intensive, and complex joins are impacting the performance.
2. Aggregation:
   - Pre-aggregated data is needed, and the overhead of computing aggregations on the fly is high.
3. Scalability:
   - The database needs to scale, and normalization is causing bottlenecks

Reasons for Denormalization

- Improve Performance: Denormalization can improve query performance, as joining multiple tables can be expensive, especially if the tables are large.
- Simplify Queries: It can make querying simpler, as users or developers don't need to write complex join statements to retrieve data.

Denormalization Downside

- **Redundancy:** There will be more redundant data.
- **Maintenance Overhead:** Updates, Inserts, and Deletes become more complex and error-prone as the same piece of information might need to be updated in multiple places.
- **Storage Cost:** Denormalized tables will likely require more storage.

Balancing Normalization and Denormalization

- Striking the right balance between normalization and denormalization is crucial.
- While normalization eliminates redundancy and improves data integrity, it can lead to complex joins, which might impact performance.
- On the other hand, denormalization can optimize read performance but at the expense of increased storage and decreased data integrity.
- The decision to normalize or denormalize should be made based on specific use cases, system requirements, and performance needs.

PART 04

SQL Advanced Queries

SUM and GROUP BY

- This query calculates the total amount spent by each customer in the Orders table and groups the results by CustomerID.

```sql
SELECT CustomerID, SUM(TotalAmount) as TotalSpent
FROM Orders
GROUP BY CustomerID;
```

AVG and GROUP BY

- This query calculates the average quantity of each product in the OrderDetails table and groups the results by ProductID.

```
SELECT ProductID, AVG(Quantity) as AverageQuantity
FROM OrderDetails
GROUP BY ProductID;
```

COUNT, GROUP BY, and HAVING

- This query retrieves the customers who have placed more than 5 orders, groups the results by CustomerID, and filters the groups using the HAVING clause to include only those with more than 5 orders.

```
SELECT CustomerID, COUNT(OrderID) as NumberOfOrders
FROM Orders
GROUP BY CustomerID
HAVING COUNT(OrderID) > 5;
```

Conditional Queries using CASE

- This query calculates the number of orders where the quantity of a product is more than 5, for each product in the OrderDetails table, and groups the results by ProductID.

```sql
SELECT ProductID,
SUM(CASE WHEN Quantity > 5 THEN 1 ELSE 0 END) as OrdersWithMoreThanFiveItems
FROM OrderDetails
GROUP BY ProductID;
```

Using AVG, COUNT, and Conditional with CASE

- This query calculates the average price and count of both new and used products in the Products table.

```sql
SELECT
 AVG(CASE WHEN Condition = 'New' THEN Price ELSE NULL END) as AvgNewPrice,
 AVG(CASE WHEN Condition = 'Used' THEN Price ELSE NULL END) as AvgUsedPrice,
 COUNT(CASE WHEN Condition = 'New' THEN 1 END) as CountNew,
 COUNT(CASE WHEN Condition = 'Used' THEN 1 END) as CountUsed
FROM Products;
```

SUM, GROUP BY, and HAVING with Subquery

- This query retrieves the customers whose total spending is above the average total spending of all customers, grouping the results by CustomerID, and filtering the groups using a subquery in the HAVING clause.

```sql
SELECT CustomerID, SUM(TotalAmount) as TotalSpent
FROM Orders
GROUP BY CustomerID
HAVING SUM(TotalAmount) > (SELECT AVG(TotalAmount) FROM Orders);
```

Nested GROUP BY with COUNT and HAVING

- This query finds the products that appear in more than three distinct orders in the OrderDetails table.

```
SELECT ProductID
FROM OrderDetails
GROUP BY ProductID
HAVING COUNT(DISTINCT OrderID) > 3;
```

Using GROUP BY with Multiple Columns and Aggregations

- This query calculates the average and total quantity for each product for each customer in the OrderDetails table and groups the results by CustomerID and ProductID.

```sql
SELECT CustomerID, ProductID, AVG(Quantity) as AverageQuantity, SUM(Quantity) as
TotalQuantity
FROM OrderDetails
GROUP BY CustomerID, ProductID;
```

Subquery in a SELECT Statement

- Here, for each customer in the Customers table, a subquery counts the number of orders in the Orders table.

```
SELECT CustomerName,
 (SELECT COUNT(*)
 FROM Orders
 WHERE Customers.CustomerID = Orders.CustomerID) as OrderCount
FROM Customers;
```

Subquery in a WHERE Clause

- This query retrieves all products from the Products table that have never been ordered, i.e., their IDs do not appear in the OrderDetails table.

```sql
SELECT *
FROM Products
WHERE ProductID NOT IN (SELECT ProductID FROM OrderDetails);
```

Multiple Joins with Aggregation

- This query counts the total number of orders for each customer by joining Customers, Orders, and OrderDetails tables and then grouping the results by the customer's name.

```sql
SELECT Customers.CustomerName, COUNT(Orders.OrderID) as TotalOrders
FROM ((Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN OrderDetails ON Orders.OrderID = OrderDetails.OrderID)
GROUP BY Customers.CustomerName;
```

Window Function

- This query assigns a rank to each product based on its sales using a window function.

```
SELECT ProductID,
 ProductName,
 Sales,
 RANK() OVER (ORDER BY Sales DESC) as SalesRank
FROM ProductSales;
```

Common Table Expressions (CTE)

- This query, using a CTE, finds the month with the highest sales from the Orders table.

```sql
WITH MonthlySales AS (
 SELECT MONTH(OrderDate) as Month, SUM(TotalAmount) as Sales
 FROM Orders
 GROUP BY MONTH(OrderDate)
)
SELECT Month, Sales
FROM MonthlySales
WHERE Sales = (SELECT MAX(Sales) FROM MonthlySales);
```

Complex Nested Query

- This query retrieves the customer name and the highest sales amount from the Customers and Orders tables, where the customer's total sales are equal to the highest total sales among all customers.

```sql
SELECT CustomerName,
 (SELECT MAX(TotalSales)
 FROM (SELECT CustomerID, SUM(TotalAmount) as TotalSales
 FROM Orders
 GROUP BY CustomerID) as CustomerSales) as HighestSales
FROM Customers
WHERE (SELECT SUM(TotalAmount)
 FROM Orders
 WHERE Customers.CustomerID = Orders.CustomerID) = (SELECT MAX(TotalSales)
 FROM (SELECT CustomerID, SUM(TotalAmount) as TotalSales
 FROM Orders
 GROUP BY CustomerID) as CustomerSales);
```

Recursive CTE

- This recursive CTE finds all the managers in the hierarchy above a given employee in the Employees table.

```
WITH RECURSIVE Ancestry AS (
 SELECT EmployeeID, ManagerID
 FROM Employees
 WHERE EmployeeID = 5
 UNION ALL
 SELECT e.EmployeeID, e.ManagerID
 FROM Employees e
 INNER JOIN Ancestry a ON e.EmployeeID = a.ManagerID
)
SELECT * FROM Ancestry;
```

Recursive CTE

- This recursive CTE finds all the managers in the hierarchy above a given employee in the Employees table.

```
WITH RECURSIVE Ancestry AS (
 SELECT EmployeeID, ManagerID
 FROM Employees
 WHERE EmployeeID = 5
 UNION ALL
 SELECT e.EmployeeID, e.ManagerID
 FROM Employees e
 INNER JOIN Ancestry a ON e.EmployeeID = a.ManagerID
)
SELECT * FROM Ancestry;
```

PART 05

Limitations of Relational Databases

Scalability

Limitation: Traditional relational databases can struggle with horizontal scaling (scaling out), which is essential to handle very large amounts of data and high load.

Example: If a small e-commerce application using a relational database grows into a large platform with millions of users, it might face significant challenges in handling the load and may require sharding or partitioning, which can be complex to manage.

Complexity and Overhead

Limitation: The structure of relational databases can sometimes lead to unnecessary complexity and overhead, especially with heavy normalization.

Example: A highly normalized database might require joining multiple tables to answer a single query, affecting performance. Denormalization to optimize read performance might lead to redundancy and maintenance challenges.

Object-Relational Impedance Mismatch

Limitation: Relational databases are table-oriented, and this can lead to a mismatch with object-oriented programming languages.

Example: When using an object-oriented programming language like Java or Python, developers often need to use Object-Relational Mapping (ORM) frameworks to interact with relational databases, which can add additional layers of complexity and may lead to inefficiencies.

Rigid Schema

Limitation: Relational databases require a predefined schema, and altering the schema at a later stage can be complex and risky, especially with large datasets.

Example: If a business requirement changes and a new attribute needs to be added to a table, it could require altering the table schema, which could be time-consuming and could impact the availability of the application.

Handling of Hierarchical/Graph Data

Limitation: Relational databases are not well-suited to handle hierarchical or graph data models efficiently.

Example: Modeling and querying hierarchical data, like organizational structures or family trees, can be cumbersome and inefficient in relational databases compared to graph databases like Neo4j.

Large-Scale Write-Heavy Workloads

Limitation: For large-scale, write-heavy workloads, the ACID properties of relational databases can become a bottleneck.

Example: In applications like social media platforms where billions of new data points (e.g., posts, likes, comments) are generated every day, relational databases might struggle to keep up with the write load compared to NoSQL databases like Cassandra.

Handling of Unstructured or Semi-Structured Data

Limitation: Relational databases are best suited for structured data and can struggle with unstructured or semi-structured data.

Example: Storing and querying JSON, XML, or other semi-structured data formats can be challenging and inefficient in relational databases compared to document stores like MongoDB.

Real-Time Processing

Limitation: Relational databases are not ideally suited for real-time processing and analytics on streaming data.

Example: Real-time analytics on streaming data, like clickstream analytics, can be challenging to implement efficiently in a relational database compared to specialized solutions like Apache Kafka.

Cost

Limitation: Enterprise relational database management systems can be expensive to license and maintain.

Example: Licensing costs for enterprise-grade RDBMS like Oracle can be prohibitive for small to medium-sized enterprises, leading them to explore open-source or NoSQL alternatives.

PART 06

Challenges in Scaling ACID Databases

ACID databases are consistent and reliable, but they face scaling challenges.

Cost

- High-end servers with a large amount of RAM, multiple CPUs, and fast storage can be very expensive. The cost rises exponentially as you approach the higher end of server specifications.

Downtime

- Upgrading hardware usually requires downtime.
- Even if it's brief, this can be problematic for applications that require high availability.

Diminishing Returns

At a certain point, adding more resources doesn't yield a proportional increase in performance. For example, doubling the RAM might not double the performance of the database.

Single Point of Failure

Even if a server is powerful, it can still fail. If the database resides on a single server, it becomes a single point of failure, which can lead to significant downtime until the server is repaired or replaced.

Heat and Power Consumption

High-performance servers generate more heat, which requires more cooling. This increases the infrastructure costs and power consumption.

Vendor Lock-in

Once you invest in specific high-end hardware, you might be locked into a particular vendor for parts, upgrades, and maintenance, which can limit flexibility and increase costs.

Maintenance Complexity

High-end hardware can be more complex to maintain and may require specialized knowledge or expertise

Lack of Horizontal Scalability

While vertical scaling focuses on adding resources to a single server, it doesn't leverage the advantages of distributing the load across multiple servers (horizontal scaling). There are benefits to horizontal scaling, like load distribution and redundancy, which vertical scaling doesn't provide.

Backup and Recovery

Backing up very large databases can be challenging. Recovery times can also be longer since there's more data to restore.

Hardware Limitations

There's an upper limit to how much you can scale up a single server. Once the maximum capacity is reached, you can't add more resources to further improve performance.

Licensing Costs

Some software licenses are based on hardware specifications, such as the number of CPU cores. Scaling up might inadvertently increase software licensing costs.