Technical Demonstration: Query Languages and Database Constraints

Anass Inani

October 27, 2024

Question 1

Using a Query Language to Formulate Queries

Query languages are specialized languages used to make queries in databases and information systems. The most common query language is **Structured Query Language (SQL)**, which is used for managing and manipulating relational databases.

Let us consider a relational database that contains the following tables:

- **Customers**(CustomerID, Name, City)
- **Orders**(OrderID, OrderDate, CustomerID, Amount)

Here, CustomerID and OrderID denote the primary keys of their respective tables.

An example of an SQL query to find the names of customers who have placed orders exceeding $1000 is:

```sql
SELECT Customers.Name
FROM Customers
JOIN Orders ON Customers.CustomerID = Orders.CustomerID
WHERE Orders.Amount > 1000;
```

This query performs the following steps:

1. Joins the **Customers** and **Orders** tables on the common **CustomerID** field.
2. Filters the records where **Orders.Amount** is greater than 1000.
3. Selects the **Name** field from the resulting set.

Mathematical Foundation for Query Languages

The mathematical foundation of query languages, particularly SQL, is based on **Relational Algebra** and **Relational Calculus**.

Relational Algebra

Relational algebra is a procedural query language that consists of a set of operations that take one or two relations as input and produce a new relation as output. The fundamental operations are:

- **Selection** (σ): Filters rows based on a predicate.
- **Projection** (π): Selects specific columns.
- **Union** ($\cup$): Combines the tuples of two relations.
- **Set Difference** ($-$): Finds tuples in one relation but not in another.
- **Cartesian Product** ($\times$): Combines tuples from two relations.
- **Rename** (ρ): Renames the relation or its attributes.

For example, the SQL query above can be expressed in relational algebra as:

$$\pi_{\text{Name}}\left(\sigma_{\text{Amount}>1000}\left(\text{Customers} \bowtie \text{Orders}\right)\right)$$

Where $\bowtie$ denotes the natural join operation.

Relational Calculus

Relational calculus is a non-procedural query language that uses mathematical predicate calculus to describe the queries. It comes in two flavors:

- **Tuple Relational Calculus (TRC)**: Queries are expressed as $\{t \mid P(t)\}$, where t is a tuple and $P(t)$ is a predicate.
- **Domain Relational Calculus (DRC)**: Queries are expressed as $\{\langle x_1, x_2, \ldots, x_n\rangle \mid P(x_1, x_2, \ldots, x_n)\}$, where x_i are domain variables.

The previous query in TRC can be written as:

$$\{c.\text{Name} \mid \exists o\left(\text{Customers}(c) \wedge \text{Orders}(o) \wedge c.\text{CustomerID} = o.\text{CustomerID} \wedge o.\text{Amount} > 1000\right)\}$$

Reflection

Understanding the mathematical foundations of query languages allows for a deeper comprehension of how queries are processed and optimized. Relational algebra provides a clear procedural method for constructing queries, while relational calculus offers a declarative approach. Both are essential for database theory and practice, and they underpin the functionality of SQL and other query languages.

Question 2

Embedded SQL Statements in a Third-Generation Programming Language

Embedded SQL allows SQL statements to be included within code written in a third-generation programming language (3GL) such as C, C++, Java, or Python. This integration enables applications to interact with databases directly, executing queries and processing results within the program's context.

Example: Embedded SQL in Python

Using Python with a library such as **sqlite3** or **PyMySQL**, we can embed SQL statements within Python code.

Connecting to the Database

First, we establish a connection to the database:

```python
import sqlite3

# Connect to the SQLite database
conn = sqlite3.connect('example.db')
cursor = conn.cursor()
```

Executing an SQL Query

Suppose we want to retrieve the names of customers from a specific city:

```python
city = 'New York'

# Parameterized query to prevent SQL injection
cursor.execute('SELECT Name FROM Customers WHERE City = ?', (city
    ,))
```

Processing the Results

We can fetch and process the results:

```python
results = cursor.fetchall()

for row in results:
    print(row[0])
```

Example: Embedded SQL in Java

In Java, we can use **JDBC** (Java Database Connectivity) to embed SQL statements.

Connecting to the Database

```java
import java.sql.*;

Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/mydb", "user", "password");
Statement stmt = conn.createStatement();
```

Executing an SQL Query

```java
String city = "New York";
String query = "SELECT Name FROM Customers WHERE City = ?";
PreparedStatement pstmt = conn.prepareStatement(query);
pstmt.setString(1, city);
ResultSet rs = pstmt.executeQuery();
```

Processing the Results

```java
while(rs.next()) {
    String name = rs.getString("Name");
    System.out.println(name);
}
```

Reflection

Embedding SQL in a 3GL allows developers to create dynamic and powerful applications that interact with databases seamlessly. It is crucial to use parameterized queries or prepared statements to prevent SQL injection attacks. Understanding how to integrate SQL within programming languages enhances the ability to build robust data-driven applications.

Question 3

Implications of Different Constraints on the Database Structure

Constraints in a database define rules that the data must comply with, ensuring data integrity and consistency. Different types of constraints have various implications on the database structure and how data is managed.

Primary Key Constraint

A primary key uniquely identifies each record in a table.
Implications:

- Enforces uniqueness of the key column(s).
- Automatically creates an index, improving query performance.
- Ensures that no duplicate or `NULL` values are entered in the primary key columns.

Foreign Key Constraint

A foreign key in one table refers to a primary key in another table, establishing a relationship between the two tables.
Implications:

- Enforces referential integrity between related tables.
- Prevents actions that would destroy links between tables.
- May restrict deletion or updating of referenced data.

Unique Constraint

Ensures that all values in a column are unique.
Implications:

- Prevents duplicate entries in the constrained column(s).
- Allows one `NULL` value (in most SQL implementations).
- May create an index to enforce uniqueness.

Not Null Constraint

Specifies that a column cannot have `NULL` values.

Implications:

- Ensures that the column must have a value upon record insertion.
- Prevents accidental omission of critical data.
- May impact the order of data insertion when dealing with dependencies.

Check Constraint

Enforces domain integrity by limiting the values that can be placed in a column.

Implications:

- Validates data against a logical expression.
- Prevents invalid data entry that does not meet the specified condition.
- May impact performance if complex expressions are used.

Default Constraint

Specifies a default value for a column when no value is provided.

Implications:

- Ensures that a column has a valid value even if none is provided.
- Simplifies data entry by auto-populating fields.
- May mask omissions if defaults are not carefully chosen.

Overall Implications on Database Structure

Implementing constraints affects the database structure in several ways:

- **Data Integrity**: Constraints enforce rules that maintain the accuracy and reliability of the data.
- **Performance**: Indexes created by constraints like primary keys and unique constraints can improve query performance but may slow down data modification operations due to the overhead of maintaining indexes.
- **Complexity**: Constraints add complexity to the database schema, requiring careful design to ensure that they do not conflict or cause unintended consequences.
- **Maintenance**: Changes to constraints may require significant effort, especially in large databases with extensive dependencies.

Reflection

Understanding the implications of constraints is crucial for designing robust databases. Constraints help enforce business rules and maintain data integrity, but they must be carefully planned to balance data integrity with performance and flexibility. Proper use of constraints leads to a well-structured database that supports reliable and efficient data operations.