DBMS

Week 4

Nested Subqueries

Subqueries in SQL

- Subqueries are queries that are embedded within another query.
- They allow you to break down complex problems into smaller, more manageable parts.
- Subqueries are enclosed in parentheses and can be used in various parts of a query, including the SELECT, FROM, WHERE, HAVING, and JOIN clauses.

Set Membership

- Subqueries can be used to check if a value or tuple is present in a set of values returned by another query.
- The IN operator checks for set membership, and the NOT IN operator checks for absence of set membership.
- Example: Finding courses taught in both Fall 2017 and Spring 2018 semesters using the IN operator to compare sets.

Set Comparison

- Subqueries can compare sets using comparison operators (<, >, <=, >=, =, <>) with keywords like SOME, ALL, ANY.
- SOME compares the left side with at least one value on the right side; ALL compares the left side with all values on the right side.
- Example: Comparing instructors' salaries to find those greater than at least one Biology department instructor.

Test for Empty Relations

- The EXISTS operator checks if a subquery returns any result tuples.
- It is often used to test for the existence of related records before performing actions.
- The NOT EXISTS operator checks if a subquery returns an empty result set.
- Example: Using EXISTS to find courses taught in both Fall 2017 and Spring 2018.

Test for Duplicate Tuples

- The UNIQUE operator checks if a subquery's result contains duplicate tuples.
- It evaluates to true if all tuples in the result set are distinct.
- Example: Finding courses offered at most once in a specific year.

Subqueries in the FROM Clause

- Subquery expressions can be used in the FROM clause to create temporary relations.
- These temporary relations can be treated as regular relations within the outer query.
- Useful for calculations involving aggregates or complex conditions.

The WITH Clause

- The WITH clause allows you to define named temporary relations that can be referenced multiple times within a query.
- Enhances readability and reduces repetition by defining subqueries once and reusing them.

Scalar Subqueries

- Scalar subqueries return a single value and can be used within expressions where a single value is expected.
- Commonly used in the SELECT clause to add computed values to the result set.

Scalar Without a FROM Clause

- Some databases require a relation in the FROM clause even when no actual relation is needed.
- Special relations like DUAL (in Oracle) or similar structures (in other databases) are used for such cases.

Modification of the Database

Deletion

- Some databases require a relation in the FROM clause even when no actual relation is needed.
- Special relations like DUAL (in Oracle) or similar structures (in other databases) are used for such cases.

Insertion

- To insert data into a relation, use the INSERT INTO statement.
- Insert a single tuple: INSERT INTO course VALUES ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
- Attribute values must match the schema's domain and tuple length.
- Attribute names can be explicitly provided in the statement.
- Inserting tuples based on a query result: INSERT INTO instructor SELECT ID, name, dept_name, 18000 FROM student WHERE dept_name = 'Music' AND tot_cred > 144;

Updates

- The UPDATE statement modifies existing tuples in a relation.
- Syntax: UPDATE r SET attr = value WHERE P;
- Change values of attributes in specified tuples based on a condition.
- Example: UPDATE instructor SET salary = salary * 1.05;
- Example: UPDATE instructor SET salary = salary * 1.05 WHERE salary < 70000;
- Complex conditions can be used in the WHERE clause.
- Use a nested SELECT in the SET clause to update based on subquery results.
- Example: UPDATE instructor SET salary = salary * 1.05 WHERE salary < (SELECT AVG(salary) FROM instructor);
- Use the CASE statement to update based on conditions.
- Example:
  - UPDATE instructor
  - SET salary = CASE
  - WHEN salary <= 100000 THEN salary * 1.05
  - ELSE salary * 1.03
  - END;

Class activity

- Data-definition language
- Data-manipulation language
- Database schema
- Database instance
- Relation schema
- Relation instance
- Primary key
- Foreign key
  - Referencing relation
  - Referenced relation
- Null value
- Query language
- SQL query structure
  - select clause
  - from clause
  - where clause
- Multiset relational algebra
- as clause
- order by clause
- Table alias
- Correlation name (correlation variable, tuple variable)
- Set operations
  - union
  - intersect
  - except
- Aggregate functions
  - avg, min, max, sum, count
  - group by
  - having
- Nested subqueries
- Set comparisons
  - {<,<=,>,>=} { some, all }
  - exists
  - unique
- lateral clause
- with clause
- Scalar subquery
- Database modification
  - Delete
  - Insert
  - Update

Chapter 4 Intermediate SQL

Intermediate SQL builds on the foundation of basic SQL knowledge and delves into more complex querying, data manipulation, and optimization techniques. It covers advanced features and techniques that allow you to perform more sophisticated operations on databases.

Join Expressions

Natural Join

- A type of join operation that considers only pairs of tuples with the same value on attributes that appear in the schemas of both relations.

Join using On Condition

- A general join operation that allows you to specify an arbitrary join condition using the "ON" keyword.

Outer Joins

- These are join operations that preserve tuples even if they don't have matching values in the other relation being joined. There are three types of outer joins:
  - Left Outer Join: Preserves tuples only in the relation before the left outer join operation.
  - Right Outer Join: Preserves tuples only in the relation after the right outer join operation.
  - Full Outer Join: Preserves tuples in both relations being joined.

Join Conditions

- These conditions specify how the relations are to be joined. There are three types of join conditions:
  - Natural: Matches tuples based on attributes with the same name in both relations.
  - Using: Matches tuples based on specified attributes.
  - On: Allows you to specify an arbitrary predicate for matching tuples.

Join Types and Conditions

- You can combine different join types with different join conditions to achieve specific results. For example, you can have a "left outer join using" or a "right outer join on".

Views

Introduction to SQL Views

- SQL Views: Creating Virtual Relations
  - Definition: Views are query-based virtual tables in SQL.
  - Provide an abstracted and secure way to access data.
- Purpose of SQL Views:
  - Security: Control which data users can access.
  - Data Abstraction: Simplify complex queries for users.
  - Query Simplification: Provide a predefined structure for commonly used queries.

Creating Views

- Syntax: CREATE VIEW view_name AS query_expression;
- Defining a Simple View:
  - CREATE VIEW faculty AS
  - SELECT ID, name, dept_name
  - FROM instructor;
- Benefits of Views:
  - Customized Data Access: Choose specific attributes for users.
  - Data Hiding: Prevent users from accessing sensitive information.
  - Query Abstraction: Simplify querying by encapsulating complex logic.

Using Views in Queries

- Views are treated like tables in queries.
- Example Query:
  - SELECT course_id
  - FROM physics_fall_2017
  - WHERE building = 'Watson';
- Flexibility: Views can be combined with other tables and views.
- Abstraction: Users interact with high-level views instead of complex relations.

Attribute Names in Views

- Attribute names can be explicitly specified in the view definition.
- Example:
  - CREATE VIEW departments_total_salary(dept_name, total_salary) AS
  - SELECT dept_name, SUM(salary)
  - FROM instructor
  - GROUP BY dept_name;
- Clarity: Explicit naming improves understanding of view attributes.

Materialized Views

- Materialized Views:
  - Some databases support materialized views.
  - Precomputed results stored in the database.
  - Benefit: Faster query performance.
- View Maintenance:
  - Immediate: Update view when underlying data changes.
  - Lazy: Update view when accessed.
  - Periodic: Update view at specified intervals.

Update of Views

- Challenges of Updating Views:
  - Complex translation from view to underlying relations.
  - Default SQL approach restricts modifications on views.
- Updatable Views:
  - Conditions for Updatability:
    - Single relation in FROM clause.
    - Simple SELECT clause without aggregates.
    - Unlisted attributes can be set to null.
    - No GROUP BY or HAVING clause.

With Check Option

- WITH CHECK OPTION:
  - View Definition: CREATE VIEW view_name AS query_expression WITH CHECK OPTION;
  - Prevents Insertions that Violate View Conditions.
  - Ensures data consistency between view and underlying relations.
- Control: Offers control over what data can be inserted or updated.

Alternative: Triggers

- Triggers for View Modifications:
  - INSTEAD OF Triggers: Replace default actions for inserts, updates, and deletes.
  - Provide fine-grained control over modification operations.
- Use Cases: Handling complex modifications through views.

Transactions

Introduction to Transactions

- Transactions in SQL group multiple database actions as a single unit.
- Ensures either all actions are fully completed or fully undone.
- Vital for scenarios like bank account transfers and database updates.

Starting and Ending Transactions

- Transaction begins implicitly with an SQL statement.
- Finalize a transaction with:
  - COMMIT WORK: Makes updates permanent, starts a new transaction.
  - ROLLBACK WORK: Undoes updates, restores database state.

Managing Transactions with BEGIN and END

- BEGIN and END group SQL statements into a single transaction.
- Ensures related actions are treated as a single unit.
- Useful for applying changes together or none at all.

Automatic Commit Behavior

- Some databases treat each SQL statement as a separate transaction.
- Immediate commit after each statement, akin to saving after each step.
- Can lead to issues when you want a series of steps as one transaction.

Turning Off Automatic Commit

- To treat multiple steps as a single transaction, disable automatic commit.
- Various methods based on the database used.
- Examples: SET AUTOCOMMIT OFF or BEGIN TRANSACTION.

Best Practices and Considerations

- Transactions ensure data consistency and error handling.
- Acts as a safety net: roll back changes if something goes wrong.
- Choose when changes become permanent: after each step or series.

Integrity Constraints

Integrity Constraints Overview

- Integrity constraints ensure data consistency and prevent accidental data loss.
- They guard against unauthorized changes that could disrupt database consistency.
- Distinguished from security constraints, which protect against unauthorized access.
- Examples of Integrity Constraints:
  - Instructor name must not be null.
  - Each instructor must have a unique instructor ID.
  - Department names in the course relation must match those in the department relation.
  - Department budget must be greater than $0.00.

Defining and Applying Integrity Constraints

- Integrity constraints are part of database schema design.
- They can be declared using the CREATE TABLE command or added later using ALTER TABLE.
- Constraints can also have names for easier identification and management.

Types of Integrity Constraints

- Not Null Constraint:
  - Forbids null values in specific attributes.
  - Used to ensure attributes like names or budget values are never null.
- Unique Constraint:
  - Specifies a set of attributes as a superkey.
  - Ensures no two tuples have identical values on the specified attributes.
  - Allows null values unless explicitly declared as non-null.
- Check Clause Constraint:
  - Specifies a predicate that must be true for all tuples in a relation.
  - Useful for ensuring specific attribute conditions or simulating enumerated types.
  - Can include subqueries but not widely supported in practice.

Referential Integrity

- Ensures relationships between relations are maintained.
- Foreign keys reference primary keys in other relations.
- Prevents orphaned references or actions that break relationships.
- Foreign key constraints can cascade actions like updates and deletions.

Complex Check Conditions and Assertions

- Complex check conditions enforce more intricate constraints.
- Not well-supported in SQL implementations.
- Assertions express conditions to be maintained at all times.
- Useful for specifying complex relationships and business rules.

Use with Caution

- Integrity constraints should be used carefully due to potential overhead.
- Testing and maintaining complex constraints can impact performance.
- Triggers can be used as an alternative to implement constraints in some systems.