四川大學
SICHUAN UNIVERSITY

Database System Concepts

Advanced SQL

伍元凯

College of Computer Science (Software), Sichuan University

wuyk0@scu.edu.cn

2023/03/22

海纳百川
有容乃大

SQL is a **domain-specific** language used for **managing** and **manipulating** relational databases. While SQL is a powerful tool for interacting with databases, it is not a **general-purpose** programming language.

- There exist queries that can be expressed in C, Jave, or Python that cannot be expressed in SQL (regression analysis, hypothesis testing, time-series analysis, NLP, clustering, classification, and prediction)
- **Nondeclarative** action (printing, interacting, graphical user interface)

- **Dynamic SQL** refers to a programming technique that allows SQL statements to be constructed and executed at runtime, rather than being hard-coded into an application or stored procedure. Dynamic SQL is typically implemented using **string concatenation or substitution to build the SQL statement**, and then executing the resulting string as a single command.
- **Embedded SQL**, on the other hand, refers to the practice of embedding SQL statements directly into a host programming language, such as Java, C++, or COBOL. This approach allows SQL to be executed directly from within the host language, rather than having to call out to a separate SQL interpreter.

目录

1 Accessing SQL from a Programming Language
   - JDBC
   - Database Access from Python
   - OUBC
   - Embedded SQL and Database

2 Functions and Procedures

3 Triggers

4 Recursive Quries

5 Advanced Aggregation Features

6 Ending

JDBC example

```java
import java.sql.*;
public class ExampleJDBC {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            // Open a connection to the database
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydatabase", "root", "mypassword");
            // Create a statement object to execute SQL queries
            Statement stmt = conn.createStatement();
            // Execute a SELECT query and get the result set
            ResultSet rs = stmt.executeQuery("SELECT * FROM mytable");
            // Iterate over the result set and print each row to the console
            while (rs.next()) {
                int id = rs.getInt("id");
                String name = rs.getString("name");
                double salary = rs.getDouble("salary");
                System.out.println("id=" + id + ", name=" + name +
                    ", salary=" + salary);
            }
            // Close the result set, statement, and connection
            rs.close();
            stmt.close();
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

1. register the JDBC driver ”Class.forName(”com.mysql.jdbc.Driver”)”
2. open a connection ”DriverManager.getConnection(url, username, password)”
3. create and execute statement ”stmt.executeQuery(”SELECT * FROM mytable”)”
4. iterate over the Result Set using while ”(rs.next())”.
5. close the ResultSet, Statement, and Connection objects.

Prepared Statement

```java
PreparedStatement stmt = conn.prepareStatement("INSERT INTO table_name (column1, column2, ...) VALUES (?, ?, ...)");
stmt.setXXX(parameterIndex, value);
stmt.setYYY(parameterIndex, value);
...
stmt.executeUpdate();
```

- XXX, YYY, data type (String)
- parameterIndex, ”?” in PreparedStatement
- value, the corresponding value of the parameterIndex

Real example

```java
String sql = "INSERT INTO employees (name, salary, hire_date) VALUES (?, ?, ?)";
PreparedStatement stmt = conn.prepareStatement(sql);
// Set the input parameters for the prepared statement
stmt.setString(1, "John");
stmt.setDouble(2, 50000.00);
stmt.setDate(3, new java.sql.Date(System.currentTimeMillis()));
```

Dangerous query

```
String sourceAccount = request.getParameter("sourceAccount");
String destinationAccount = request.getParameter("destinationAccount");
double amount = Double.parseDouble(request.getParameter("amount"));

String updateQuery = "UPDATE accounts SET balance = balance - " + amount + " WHERE account_number = " + sourceAccount;
Statement stmt = conn.createStatement();
int numRowsAffected = stmt.executeUpdate(updateQuery);

updateQuery = "UPDATE accounts SET balance = balance + " + amount + " WHERE account_number = " + destinationAccount;
numRowsAffected = stmt.executeUpdate(updateQuery);
```

Inject ”12345; DROP TABLE accounts; –” to ”sourceAccount”

Injection Attack

```
UPDATE accounts SET balance = balance - 1000.0 WHERE account_number = 12345; DROP TABLE accounts; --;
```

Prevention by Prepared Statements

```
String sourceAccount = request.getParameter("sourceAccount");
String destinationAccount = request.getParameter("destinationAccount");
double amount = Double.parseDouble(request.getParameter("amount"));
String updateQuery = "UPDATE accounts SET balance = balance - ? WHERE account_number = ?";
PreparedStatement stmt = conn.prepareStatement(updateQuery);
stmt.setDouble(1, amount);
stmt.setString(2, sourceAccount);
int numRowsAffected = stmt.executeUpdate();
updateQuery = "UPDATE accounts SET balance = balance + ? WHERE account_number = ?";
stmt = conn.prepareStatement(updateQuery);
stmt.setDouble(1, amount);
stmt.setString(2, destinationAccount);
numRowsAffected = stmt.executeUpdate();
```

If an attacker tries to enter the same input as before (12345; DROP TABLE accounts; –) as the source account number, the prepared statement will treat it as a string value and escape any special characters

- Prepared statements allow the DBMS to **cache the execution plan** and reuse it across multiple executions of the same statement.
- In contrast, when a query is executed without using a prepared statement, the DBMS must parse and optimize the query every time it is executed.

Callable Statements

```
String procedure = "{CALL my_stored_procedure(?, ?, ?)}";
CallableStatement stmt = conn.prepareCall(procedure);

// set input parameters
stmt.setString(1, "John");
stmt.setString(2, "Doe");
stmt.setInt(3, 30);

// register output parameter
stmt.registerOutParameter(3, Types.INTEGER);

// execute the stored procedure
stmt.execute();

// retrieve the output parameter
int result = stmt.getInt(3);
```

Invocation of SQL stored procedures and functions.

database product name, version, and driver version:

```java
DatabaseMetaData metadata = connection.getMetaData();
String productName = metadata.getDatabaseProductName();
String productVersion = metadata.getDatabaseProductVersion();
String driverVersion = metadata.getDriverVersion();
```

Get a list of tables in the database:

```java
DatabaseMetaData metadata = connection.getMetaData();
ResultSet tables = metadata.getTables(null, null, null, new String[] {"TABLE"});
while (tables.next()) {
    String tableName = tables.getString("TABLE_NAME");
    // process the table name
}
```

Get information about the columns of a table:

```java
DatabaseMetaData metadata = connection.getMetaData();
ResultSet columns = metadata.getColumns(null, null, "my_table", null);
while (columns.next()) {
    String columnName = columns.getString("COLUMN_NAME");
    String dataType = columns.getString("DATA_TYPE");
    int columnSize = columns.getInt("COLUMN_SIZE");
    // process the column information
}
```

Get information about the primary keys of a table:

```java
DatabaseMetaData metadata = connection.getMetaData();
ResultSet primaryKeys = metadata.getPrimaryKeys(null, null, "my_table");
while (primaryKeys.next()) {
    String columnName = primaryKeys.getString("COLUMN_NAME");
    int keySeq = primaryKeys.getInt("KEY_SEQ");
    // process the primary key information
}
```

Dealing Transactions:

```
// assume conn is a valid Connection object
try {
    // set auto-commit to false to start a transaction
    conn.setAutoCommit(false);
    // execute some SQL statements within the transaction
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("UPDATE my_table SET my_column = 'value' WHERE id = 1");
    stmt.executeUpdate("DELETE FROM my_table WHERE id = 2");
    // commit the transaction if all statements executed successfully
    conn.commit();
} catch (SQLException e) {
    // rollback the transaction if any statement fails
    conn.rollback();
    e.printStackTrace();
} finally {
// restore auto-commit to true after the transaction completes
    conn.setAutoCommit(true);
}
```

We first set the **autoCommit** property of the Connection object to false to start a transaction. If all statements execute successfully, we call the **commit** method to commit the transaction. If any statement fails, we catch the **SQLException** and call the **rollback** method to roll back the transaction.

目录

1. Accessing SQL from a Programming Language
   - JDBC
   - Database Access from Python
   - OUBC
   - Embedded SQL and Database
2. Functions and Procedures
3. Triggers
4. Recursive Quries
5. Advanced Aggregation Features
6. Ending

```python
import psycopg2
# establish a database connection
conn = psycopg2.connect(
    host="localhost",
    database="my_database",
    user="my_username",
    password="my_password"
)
# create a cursor object
cur = conn.cursor()
# prepare a SQL statement
query = "SELECT name, age FROM my_table WHERE id = %s"
cur.execute("PREPARE prepared_query AS {}".format(query))
# execute the prepared statement with different parameters
id1 = 1
cur.execute("EXECUTE prepared_query (%s)", (id1,))
print(cur.fetchone())

cur.execute("DEALLOCATE prepared_query")
# commit the transaction and close the cursor and connection
conn.commit()
cur.close()
```

- psycopg2.connect: Establish a database connection
- cursor: Execute SQL queries and fetch the results.
- execute: passing in the query string and any necessary parameters.
- commit: commit the transaction.

OUBC | 目录

1. Accessing SQL from a Programming Language
   - JDBC
   - Database Access from Python
   - OUBC
   - Embedded SQL and Database
2. Functions and Procedures
3. Triggers
4. Recursive Quries
5. Advanced Aggregation Features
6. Ending

```c
#include <stdio.h>
#include <stdlib.h>
#include <sql.h>
#include <sqlext.h>
#define BUFFER_SIZE 256
int main() {
    SQLHENV env;
    SQLHDBC dbc;
    SQLHSTMT stmt;
    SQLCHAR query[BUFFER_SIZE];
    SQLRETURN ret;
    SQLINTEGER id;
    SQLCHAR name[BUFFER_SIZE];
    SQLINTEGER age;
    // Allocate environment handle
    SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env);
    // Set ODBC version to 3
    SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION, (SQLPOINTER)
        SQL_OV_ODBC3, 0);
    // Allocate connection handle
    SQLAllocHandle(SQL_HANDLE_DBC, env, &dbc);
    // Connect to database
    SQLConnect(dbc, (SQLCHAR*) "DSN=mydatabase", SQL_NTS, (SQLCHAR
        *) "", SQL_NTS, (SQLCHAR*) "", SQL_NTS);
    // Allocate statement handle
    SQLAllocHandle(SQL_HANDLE_STMT, dbc, &stmt);
    // Prepare SQL query
    SQLPrepare(stmt, (SQLCHAR*) "SELECT * FROM mytable WHERE id = ?
        ", SQL_NTS);
    // Bind input parameter to statement
    SQLBindParameter(stmt, 1, SQL_PARAM_INPUT, SQL_C_LONG,
        SQL_INTEGER, 0, 0, &id, 0, NULL);
    id = 1;
```

```c
    // Execute statement
    SQLExecute(stmt);
    // Bind result columns to variables
    SQLBindCol(stmt, 1, SQL_C_CHAR, name, BUFFER_SIZE, NULL);
    SQLBindCol(stmt, 2, SQL_C_LONG, &age, 0, NULL);
    // Fetch results
    while (SQLFetch(stmt) == SQL_SUCCESS) {
        printf("Name: %s\n", name);
        printf("Age: %d\n", age);
    }
    // Clean up resources
    SQLFreeHandle(SQL_HANDLE_STMT, stmt);
    SQLDisconnect(dbc);
    SQLFreeHandle(SQL_HANDLE_DBC, dbc);
    SQLFreeHandle(SQL_HANDLE_ENV, env);
    return 0;
}
```

- The parameter value is bound to the statement using the SQLBindParameter function.
- The statement is then executed using SQLExecute.
- the result columns are bound to variables using SQLBindCol.
- the results are fetched using SQLFetch.

目录

1. **Accessing SQL from a Programming Language**
   - JDBC
   - Database Access from Python
   - OUBC
   - Embedded SQL and Database
2. **Functions and Procedures**
3. **Triggers**
4. **Recursive Quries**
5. **Advanced Aggregation Features**
6. **Ending**

- An embedded database is a database management system that is **integrated into an application, rather than running as a separate server or service.** This means that the database engine and the application are packaged together, and the application communicates directly with the database engine to perform database operations.
- Embedded SQL is a technique for integrating SQL statements into host language code. With embedded SQL, developers can **write SQL statements directly in their host language code**, and use the language's native interface to execute those statements against a database.

目录

1 Accessing SQL from a Programming Language

2 Functions and Procedures

- Declaring and Invoking SQL Functions and Procedures
- Language Constructs for Procedures and Functions
- External Language Routines

3 Triggers

4 Recursive Quries

5 Advanced Aggregation Features

6 Ending

A complex function

```
CREATE FUNCTION GetTopSellingProductsByCategory(category VARCHAR(50),
limit INTEGER)
    RETURNS TABLE(ProductName VARCHAR(50), SalesAmount DECIMAL(10,2))
    RETURNS TABLE
        (SELECT p.ProductName, SUM(od.Quantity * od.UnitPrice) AS SalesAmount
        FROM Products p
        INNER JOIN Categories c ON p.CategoryID = c.CategoryID
        INNER JOIN OrderDetails od ON p.ProductID = od.ProductID
        INNER JOIN Orders o ON od.OrderID = o.OrderID
        WHERE c.CategoryName = $1
        GROUP BY p.ProductName
        ORDER BY SalesAmount DESC);

SELECT * FROM GetTopSellingProductsByCategory('Beverages', 3);
```

A complex procedure

```sql
CREATE PROCEDURE calculate_sales_revenue(
    IN category_name VARCHAR(50),
    IN start_date DATE,
    IN end_date DATE,
    OUT total_revenue DECIMAL(10,2)
)
BEGIN
    SELECT SUM(p.price * oi.quantity) INTO total_revenue
    FROM products p
    JOIN order_items oi ON p.product_id = oi.product_id
    JOIN orders o ON oi.order_id = o.order_id
    JOIN categories c ON p.category_id = c.category_id
    WHERE c.category_name = category_name
    AND o.order_date BETWEEN start_date AND end_date;
END

CALL calculate_sales_revenue('Electronics', '2022-01-01', '
    2022-12-31', @total_revenue);
```

目录

1. Accessing SQL from a Programming Language
2. **Functions and Procedures**
   - Declaring and Invoking SQL Functions and Procedures
   - Language Constructs for Procedures and Functions
   - External Language Routines
3. Triggers
4. Recursive Quries
5. Advanced Aggregation Features
6. Ending

```sql
CREATE PROCEDURE calculate_factorial(
    IN number INT,
    OUT factorial INT
)
BEGIN
    SET factorial = 1;
    SET @counter = 1;
    WHILE (@counter <= number) DO
        SET factorial = factorial * @counter;
        SET @counter = @counter + 1;
    END WHILE;
END

SET @result = 0;
CALL calculate_factorial(5, @result);
SELECT @result;
```

```
CREATE PROCEDURE find_min_max_salary(
    OUT min_salary DECIMAL(10, 2),
    OUT max_salary DECIMAL(10, 2)
)
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE cur_salary DECIMAL(10, 2);
    DECLARE cur CURSOR FOR
        SELECT salary FROM employees;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
    OPEN cur;
    SET min_salary = 99999999.99;
    SET max_salary = 0.00;
    REPEAT
        FETCH cur INTO cur_salary;
        IF NOT done THEN
            IF cur_salary < min_salary THEN
```

```
                SET min_salary = cur_salary;
            END IF;
            IF cur_salary > max_salary THEN
                SET max_salary = cur_salary;
            END IF;
        END IF;
    UNTIL done END REPEAT;
    CLOSE cur;
END

CALL find_min_max_salary(@min_salary, @max_salary);
SELECT @min_salary, @max_salary;
```

- DECLARE: This keyword is used to declare local variables and cursors within the procedure.
- CURSOR: A cursor is a mechanism that allows you to fetch rows from a result set one at a time.
- HANDLER: A handler is a type of construct that allows you to define an action to take when a certain condition occurs.
- OPEN: This keyword is used to open the cursor and prepare it for fetching rows.
- FETCH: This keyword is used to fetch the next row from the cursor and assign its value to a local variable.
- REPEAT: This keyword is used to define the start of a loop that will repeat until a certain condition is met.
- UNTIL: This keyword is used to specify the condition that must be met in order for the loop to exit.
- END REPEAT: This keyword is used to define the end of the loop.
- CLOSE: This keyword is used to close the cursor once we're done fetching rows from it.

```sql
CREATE PROCEDURE update_product_prices(
    IN price_change DECIMAL(10,2)
)
BEGIN
    DECLARE product_id INT;

    FOR product_id IN (SELECT id FROM products) DO
        UPDATE products SET price = price + price_change WHERE id = product_id;
    END FOR;
END

CALL update_product_prices(10.00);
```

The procedure will then update the prices of all products in the database by adding 10.00 to each product's price.

目录

❶ Accessing SQL from a Programming Language

❷ Functions and Procedures

Declaring and Invoking SQL Functions and Procedures

Language Constructs for Procedures and Functions

External Language Routines

❸ Triggers

❹ Recursive Quries

❺ Advanced Aggregation Features

❻ Ending

```
CREATE [OR REPLACE] FUNCTION function_name ([argument_list])
    RETURNS return_type
    LANGUAGE language_name
    [EXTERNAL NAME 'external_name']
CREATE [OR REPLACE] PROCEDURE procedure_name (in [argument_list],
    out [argument_list])
    LANGUAGE language_name
    [EXTERNAL NAME 'external_name']
```

- CREATE [OR REPLACE] FUNCTION|PROCEDURE is used to create a new function or procedure or replace an existing one.
- function_name, procedure_name is the name of the function or procedure.
- LANGUAGE language_name specifies the programming language used for the function or procedure.
- EXTERNAL NAME ’external_name’ is used to specify the name of the external function or procedure that is being called by the SQL function or procedure.

```
CREATE FUNCTION compute_average_age (IN table_name TEXT, OUT average_age INTEGER)
    RETURNS RECORD
    LANGUAGE 'plpythonu'
    AS $$
        import plpy
        conn = plpy.connect()
        query = f"SELECT age FROM {table_name}"
        age_result = conn.execute(query)
        # Calculate the average age
        total_age = 0
        num_rows = 0
        for row in age_result:
            total_age += row['age']
            num_rows += 1
        average_age = total_age // num_rows
        # Return the result as a record

        return {'average_age': average_age}
    $$;
```

Triggers are used in SQL to automatically execute a set of actions when a specified event (insertion, update and delete) occurs on a table or view. Triggers are useful because they allow you to automate complex data management tasks that would be **difficult or time-consuming to perform manually**. They can also help you **maintain data integrity**.

```sql
CREATE TRIGGER update_customer_stats
AFTER INSERT ON orders
FOR EACH ROW
EXECUTE PROCEDURE update_customer_total();

CREATE PROCEDURE update_customer_total() RETURNS trigger AS $$
BEGIN
    UPDATE customer_stats
    SET total_order_amount = total_order_amount + NEW.order_amount
    WHERE customer_id = NEW.customer_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Triggers in SQL ●○○○○○ ▌目录

1. Accessing SQL from a Programming Language

2. Functions and Procedures

3. **Triggers**
   - Triggers in SQL

When Not to Use Triggers

4. Recursive Quries

5. Advanced Aggregation Features

6. Ending

Standard Trigger

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
ON table_name
[REFERENCING OLD AS old NEW AS new]
[FOR EACH ROW]
[WHEN (condition)]
trigger_body;
```

- CREATE [OR REPLACE] TRIGGER - This creates a new trigger or replaces an existing one with the same name.
- BEFORE | AFTER - This specifies whether the trigger should fire before or after the specified event occurs on the table.
- [REFERENCING] - This clause is used to define aliases for the OLD and NEW variables that are used in the trigger action.
- [WHEN (condition)] - This clause is used to specify a condition that must be true for the trigger to fire.

Table 1: Employees table

id	name	salary	departments
1	John Doe	50000	Sales
2	Jane Doe	60000	HR
3	Bob Smith	70000	HR

```sql
CREATE TRIGGER prevent_salary_updates
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
  -- Check if the employee is in the Sales department
  IF NEW.department = 'Sales' THEN
    -- If they are, prevent the update to the salary column
    SET NEW.salary = OLD.salary;
  END IF;
END;

UPDATE employees SET salary = 55000 WHERE id = 1;
```

```sql
SELECT * FROM employees;
```

Table 2: Employees table

id	name	salary	departments
1	John Doe	50000	Sales
2	Jane Doe	60000	HR
3	Bob Smith	70000	HR

Using before to prevent errors

```
CREATE TRIGGER enforce_positive_total_price_and_discount
BEFORE INSERT ON orders
FOR EACH ROW
BEGIN
  IF NEW.total_price <= 0 THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Total price must be a positive number';
  END IF;
  IF NEW.discount < 0 THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Discount must be a non-negative number';
  END IF;
END;

INSERT INTO orders (id, total_price, discount) VALUES (1, -100, 20);


ERROR 1644 (45000): Total price must be a positive number
```

- alter trigger ... disable/enable: disable and enable a specific trigger.
- drop trigger ...: removes a trigger permanently.

目录

When Not to Use Triggers

① Accessing SQL from a Programming Language

② Functions and Procedures

③ Triggers

Triggers in SQL

④ Recursive Quries

⑤ Advanced Aggregation Features

⑥ Ending

Cascade Deletion

```
CREATE TABLE departments (
  id INT PRIMARY KEY,
  name VARCHAR(50)
);

CREATE TABLE employees (
  id INT PRIMARY KEY,
  name VARCHAR(50),
  department_id INT,
  FOREIGN KEY (department_id) REFERENCES departments(id)
);

CREATE TRIGGER delete_employees
ON departments
FOR DELETE
AS
BEGIN
  DELETE FROM employees
  WHERE department_id IN (SELECT id FROM DELETED);
END;
```

Maintain materialized views

```sql
CREATE TABLE sales (
  id INT PRIMARY KEY,
  customer_id INT,
  amount DECIMAL(10, 2),
  date DATE
);

CREATE MATERIALIZED VIEW customer_sales AS
SELECT customer_id, SUM(amount) AS total_sales
FROM sales
GROUP BY customer_id;

CREATE TRIGGER update_customer_sales
AFTER INSERT OR UPDATE OR DELETE ON sales
FOR EACH STATEMENT
AS
BEGIN
  REFRESH MATERIALIZED VIEW customer_sales;
END;
```

Triggers should be used judiciously and with caution.

- **Performance issues.**
- **Complex logic.**
- **Scalability issues.**
- **Security concerns.**
- **Data integrity.**

A poor designed trigger with performance issue

```
CREATE TABLE customers (
  id INTEGER PRIMARY KEY,
  name VARCHAR(255),
  email VARCHAR(255),
  total_spent DECIMAL(10,2)
);

CREATE TABLE orders (
  id INTEGER PRIMARY KEY,
  customer_id INTEGER,
  amount DECIMAL(10,2),
  order_date DATE
);

CREATE TRIGGER update_customer_totals
AFTER INSERT ON orders
FOR EACH ROW
```

```
BEGIN
  UPDATE customers
  SET total_spent = (SELECT SUM(amount) FROM orders WHERE
      customer_id = NEW.customer_id)
  WHERE id = NEW.customer_id;
END;
```

A modified one

```
CREATE TRIGGER update_customer_totals
AFTER INSERT ON orders
FOR EACH ROW
BEGIN
  UPDATE customers
  SET total_spent = total_spent + NEW.amount
  WHERE id = NEW.customer_id;
END;
```

This trigger simply **adds the new order's amount** to the customer's existing total spent, rather than **recalculating the total spent** for every customer every time a new order is inserted.

Employees table

employee_id	employee_name	supervisor_id
1	Alice	NULL
2	Bob	1
3	Charlie	2
4	Dave	2
5	Eve	1
6	Frank	5

In this example, Alice is the **CEO of the organization and has no supervisor**. Bob and Eve report directly to Alice, while Charlie and Dave report to Bob, and Frank reports to Eve.

Resulted table

level	name
1	Alice
2	-Bob
3	–Charlie
3	–Dave
2	-Eve
3	–Frank

目录

1. Accessing SQL from a Programming Language
2. Functions and Procedures
3. Triggers
4. **Recursive Quries**
   - Transitive Closure Using Iteration
   - Recursion in SQL
5. Advanced Aggregation Features
6. Ending

Without recursive query

```sql
WITH
  level_one AS (
    SELECT employee_name, 1 AS level
    FROM employee
    WHERE supervisor_id IS NULL
  ),
  level_two AS (
    SELECT CONCAT('-', employee_name) AS employee_name, 2 AS level
    FROM employee
    WHERE supervisor_id IN (SELECT employee_id FROM level_one)
  ),
  level_three AS (
    SELECT CONCAT('--', employee_name) AS employee_name, 3 AS level
    FROM employee
    WHERE supervisor_id IN (SELECT employee_id FROM level_two)
  )
SELECT employee_name, level
```

```
FROM level_one
UNION ALL
SELECT employee_name, level
FROM level_two
UNION ALL
SELECT employee_name, level
FROM level_three;
```

1. The first CTE, level_one, extracts the employees who do not have a supervisor.
2. The second and third CTE, level_two and level_three, extracts the employees who have a supervisor in level_one and level_two.
3. Finally, we combine the results of all three CTEs using **UNION ALL** to produce the desired output, which includes the employee names and their respective levels.

目录

1. Accessing SQL from a Programming Language
2. Functions and Procedures
3. Triggers
4. **Recursive Quries**
   - Transitive Closure Using Iteration
   - Recursion in SQL
5. Advanced Aggregation Features
6. Ending

Recursive query

```sql
WITH RECURSIVE org_chart AS (
  SELECT employee_id, employee_name, supervisor_id, 1 AS level
  FROM employees
  WHERE supervisor_id IS NULL
  UNION ALL
  SELECT e.employee_id, e.employee_name, e.supervisor_id, oc.level + 1
  FROM employees e
  JOIN org_chart oc ON e.supervisor_id = oc.employee_id
)
SELECT CONCAT(REPEAT('-', level - 1), employee_name) AS name, level
FROM org_chart
ORDER BY level, name;
```

General Form

```sql
WITH RECURSIVE cte_name AS (
    -- Anchor member
    SELECT ...
    UNION ALL
    -- Recursive member
    SELECT ...
    FROM cte_name
    WHERE ...
)
SELECT ...
FROM cte_name
WHERE ...
```

In summary, a recursive SQL query uses a CTE to define a set of rules for generating a result set that includes **multiple levels** of data. The query includes an **anchor member that defines the starting point for the recursion**, and **a recursive member** that generates new rows based on the results of previous iterations.

- **WITH RECURSIVE**: This clause starts the CTE and tells the database that it should perform a recursive query.
- **cte_name**: This is the name you give to the CTE. You'll use this name later in the query to reference the results of the CTE.
- **UNION ALL**: This clause combines the anchor member and the recursive member of the CTE.

- Aggregate functions: Recursive queries cannot use aggregate functions such as SUM, AVG, COUNT, MAX, MIN, etc.
- GROUP BY: Similarly, recursive queries cannot use the GROUP BY clause in the recursive part of the query.
- Non-deterministic functions: Recursive queries cannot use non-deterministic functions such as RAND(), NEWID(), or GETDATE() in the recursive part of the query.
- Set difference whose right-hand side uses the recursive view.

目录

1 Accessing SQL from a Programming Language

2 Functions and Procedures

3 Triggers

4 Recursive Quries

5 Advanced Aggregation Features
- Ranking
- Windowing
- Pivoting
- Rollup and Cube

6 Ending

Ranking is a common technique used in SQL to **assign a rank** or order to rows within a result set, based on one or more criteria.

General Form

```
SELECT col1, col2, ..., RANK() OVER (ORDER BY col3 DESC) AS rank
FROM table_name;
```

- col1, col2, etc. represent the columns you want to select from the table.
- RANK() OVER is the ranking function. You can replace RANK() with other ranking functions such as DENSE_RANK() or ROW_NUMBER() if needed.
- ORDER BY col3 DESC specifies the column(s) to use for ranking. In this case, we're using col3 in descending order .
- AS rank is an optional alias for the ranking column.

Sales

ID	Sales	Region	Date
1	1000	West	2022-01-01
2	750	East	2022-01-02
3	1200	South	2022-01-03
4	NULL	West	2022-01-04
5	1500	North	2022-01-05
6	800	East	2022-01-06
7	1100	South	2022-01-07
8	NULL	North	2022-01-08
9	1300	West	2022-01-09
10	900	East	2022-01-10

```sql
SELECT ID, Sales, RANK() OVER (ORDER BY Sales DESC NULLS LAST) AS
     SalesRank
FROM Sales;
```

Table 3: Sales Rank Table

ID	Sales	SalesRank
5	1500	1
3	1200	2
9	1300	3
7	1100	4
1	1000	5
6	800	6
10	900	7
2	750	8
4	NULL	8
8	NULL	8

Without ranking

```
SELECT
  ID,
  Sales,
  (
    SELECT COUNT(*)
    FROM
      (SELECT DISTINCT Sales FROM SalesTable WHERE Sales IS NOT NULL) AS temp
    WHERE
      SalesTable.Sales < temp.Sales OR
      (SalesTable.Sales IS NULL AND temp.Sales IS NOT NULL)
  ) + 1 AS SalesRank
FROM
  SalesTable
ORDER BY
  SalesRank ASC,
  ID ASC;
```

```sql
SELECT ID, Sales, Region, Date,
       RANK() OVER (PARTITION BY Region ORDER BY Sales DESC NULLs LAST) AS Region_Rank
FROM sales;
```

Equal query

```sql
SELECT s1.ID, s1.Sales, s1.Region, s1.Date,
       COUNT(s2.ID) + 1 AS Region_Rank
FROM sales s1
LEFT JOIN sales s2
  ON s1.Region = s2.Region AND s1.Sales < s2.Sales
GROUP BY s1.ID, s1.Sales, s1.Region, s1.Date
ORDER BY s1.Region, s1.Sales DESC;
```

Sales Table with Region Ranks

ID	Sales	Region	Date	Region_Rank
9	1300	West	2022-01-09	1
1	1000	West	2022-01-01	2
4	NULL	West	2022-01-04	2
10	900	East	2022-01-10	1
6	800	East	2022-01-06	2
2	750	East	2022-01-02	3
5	1500	North	2022-01-05	1
8	NULL	North	2022-01-08	1
3	1200	South	2022-01-03	1
7	1100	South	2022-01-07	2

RANK() in subquery

```
SELECT ID, Sales, Region, Date
FROM (
  SELECT ID, Sales, Region, Date, RANK() OVER (ORDER BY Sales DESC)
         AS Sales_Rank
  FROM Sales
)
WHERE Sales_Rank = 1;
```

Sales Table with Region Ranks

ID	Sales	Region	Date	Sales_Rank
5	1500	North	2022-01-05	1

Some other functions that can be used in place of RANK() in SQL:

- **DENSE_RANK()**: This function is similar to RANK(), but it does not skip any ranks if there are ties. Instead, it assigns the same rank to all tied rows and then continues assigning ranks in sequential order.
- **ROW_NUMBER()**: This function simply assigns a unique number to each row in the result set, starting from 1.
- **NTILE()**: This function divides the result set into a specified number of ”tiles” or groups.
- **PERCENT_RANK()**: This function returns the relative rank of each row within the result set as a percentage.
- **CUME_DIST()**: This function returns the cumulative distribution of each row within the result set.
- ...

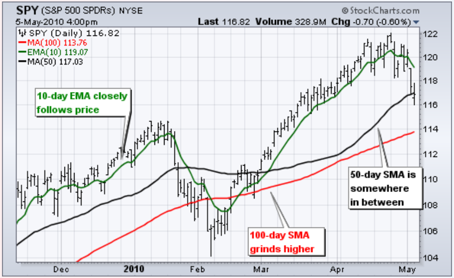
```sql
SELECT *, NTILE(4) OVER(ORDER BY Sales) as Quartile
FROM Sales;
```

ID	Sales	Region	Date	Quartile
2	750	East	2022-01-02	1
6	800	East	2022-01-06	1
10	900	East	2022-01-10	1
1	1000	West	2022-01-01	2
7	1100	South	2022-01-07	2
3	1200	South	2022-01-03	2
9	1300	West	2022-01-09	3
5	1500	North	2022-01-05	3
4	NULL	West	2022-01-04	4
8	NULL	North	2022-01-08	4

Sales by Region, Date, and Quartile

目录

1. Accessing SQL from a Programming Language
2. Functions and Procedures
3. Triggers
4. Recursive Quries
5. Advanced Aggregation Features
   - Ranking
   - Windowing
   - Pivoting
   - Rollup and Cube
6. Ending



suppose you want to calculate the 30-day moving average of a stock's closing price. You could use the following SQL query:

```
SELECT date, close_price, AVG(close_price) OVER (
    ORDER BY date
    ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
) AS moving_avg
FROM stock_prices
```

```
<window function>([expression]) OVER (
    [PARTITION BY partition_expression, ... ]
    [ORDER BY sort_expression [ASC | DESC], ... ]
    [ROWS <frame specification>]
)
```

- window function: SUM, AVG, MAX, MIN, etc.
- expression: attribute/column
- frame specification: defines the subset of rows to include in the window function calculation. There are three types of frame specification:

date	region	product	sales _ amount
2022-01-01	North	Product A	1000.00
2022-01-01	North	Product B	1500.00
2022-01-02	North	Product A	2000.00
2022-01-02	North	Product B	2500.00
2022-01-01	South	Product A	3000.00
2022-01-01	South	Product B	3500.00
2022-01-02	South	Product A	4000.00
2022-01-02	South	Product B	4500.00

Rolling Average

```
SELECT date, region, product, sales_amount, AVG(sales_amount) OVER(
    PARTITION BY region, product ORDER BY date ROWS BETWEEN 1
    PRECEDING AND CURRENT ROW) as rolling_avg
FROM table_name;
)
```

date	region	product	sales_amount	rolling_avg
2022-01-01	North	Product A	1000.00	1000.00
2022-01-01	North	Product B	1500.00	1500.00
2022-01-02	North	Product A	2000.00	1500.00
2022-01-02	North	Product B	2500.00	2000.00
2022-01-01	South	Product A	3000.00	3000.00
2022-01-01	South	Product B	3500.00	3500.00
2022-01-02	South	Product A	4000.00	3500.00
2022-01-02	South	Product B	4500.00	4000.00

Rank the sales_amount

```
SELECT date, region, product, sales_amount, RANK() OVER(PARTITION BY region, product ORDER BY sales_amount DESC) as rank
FROM table_name;
```

date	region	product	sales_amount	rank
2022-01-01	North	Product A	1000.00	2
2022-01-01	North	Product B	1500.00	2
2022-01-02	North	Product A	2000.00	1
2022-01-02	North	Product B	2500.00	1
2022-01-01	South	Product A	3000.00	2
2022-01-01	South	Product B	3500.00	2
2022-01-02	South	Product A	4000.00	1
2022-01-02	South	Product B	4500.00	1

Frame specification:
ROWS BETWEEN: Specifies a range of rows relative to the current row to include in the window frame. The possible values are:

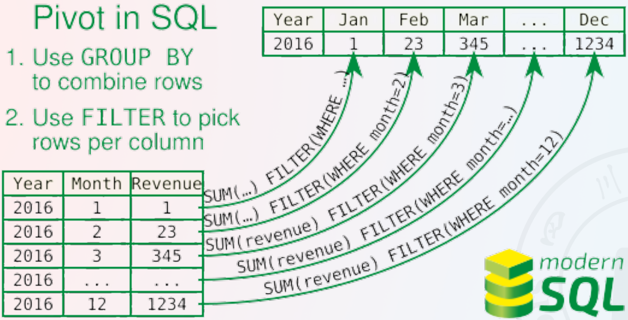
- **UNBOUNDED PRECEDING**: Includes all rows before the current row.
- **n PRECEDING**: Includes the n rows before the current row.
- **CURRENT ROW**: Includes only the current row.
- **n FOLLOWING**: Includes the n rows after the current row.
- **UNBOUNDED FOLLOWING**: Includes all rows after the current row.

- **RANGE BETWEEN**: Specifies a range of values relative to the current row to include in the window frame. **This is only applicable for window functions that use an ordering clause. The possible values are the same as ROWS BETWEEN.**
- **GROUPS BETWEEN**: Specifies a range of peer groups relative to the current row to include in the window frame.
- **UNBOUNDED PRECEDING**: Includes all rows before the current row.
- **UNBOUNDED FOLLOWING**: Includes all rows after the current row.

目录

1. Accessing SQL from a Programming Language
2. Functions and Procedures
3. Triggers
4. Recursive Quries
5. Advanced Aggregation Features
    - Ranking
    - Windowing
    - Pivoting
    - Rollup and Cube
6. Ending

Pivoting in SQL can be very useful in situations where you need to transform data from **rows (values) into columns (attributes)** to make it easier to analyze.



Pivot in SQL

1. Use GROUP BY to combine rows
2. Use FILTER to pick rows per column

Year	Month	Revenue
2016	1	1
2016	2	23
2016	3	345
2016	...	...
2016	12	1234

Year	Jan	Feb	Mar	...	Dec
2016	1	23	345	...	1234

SUM(...) FILTER(WHERE ...)
SUM(...) FILTER(WHERE month=2)
SUM(revenue) FILTER(WHERE month=3)
SUM(revenue) FILTER(WHERE month=...)
SUM(revenue) FILTER(WHERE month=12)
modern SQL

Month	Product	Sales	Expenses
January	Product A	1000	500
January	Product B	1500	750
February	Product A	1200	600
February	Product B	1800	900
March	Product A	800	400
March	Product B	1200	600

Sales and expenses by product and month

Without Pivoting

```sql
SELECT
    Month,
    SUM(CASE WHEN Product = 'Product A' THEN Sales ELSE 0 END) AS 'Product A Sales',
    SUM(CASE WHEN Product = 'Product A' THEN Expenses ELSE 0 END) AS 'Product A Expenses',
    SUM(CASE WHEN Product = 'Product B' THEN Sales ELSE 0 END) AS 'Product B Sales',
    SUM(CASE WHEN Product = 'Product B' THEN Expenses ELSE 0 END) AS 'Product B Expenses'
FROM
    SalesExpenses
GROUP BY
    Month;
```

Pivoting

```sql
SELECT Month,
    [Product A Sales], [Product A Expenses],
    [Product B Sales], [Product B Expenses]
FROM (
    SELECT Month, Product, Sales, Expenses
    FROM table1
) AS t
PIVOT (
    SUM(Sales) FOR Product IN ([Product A], [Product B])
) AS SalesPivot
PIVOT (
    SUM(Expenses) FOR Product IN ([Product A], [Product B])
) AS ExpensesPivot
```

Month	Product A Sales	Product A Expenses	Product B Sales	Product B Expenses
January	1000	500	1500	750
February	1200	600	1800	900
March	800	400	1200	600

General Form

```
SELECT <non-pivoted column(s)>
    , [first pivoted column] AS <column name>
    , [second pivoted column] AS <column name>
    , ...
FROM
    (<source table>)
PIVOT
(
    <aggregation function>(<value column>)
    FOR
    [<column to pivot on>]
    IN ( [first pivoted column], [second pivoted column], ... )
) AS <alias for the pivot table>
```

目录

1 Accessing SQL from a Programming Language

2 Functions and Procedures

3 Triggers

4 Recursive Quries

5 Advanced Aggregation Features
- Ranking
- Windowing
- Pivoting
- Rollup and Cube

6 Ending

ROLLUP is a SQL keyword used to generate **subtotals** and grand totals of data. It generates **multiple grouping sets** by specifying columns on which data should be aggregated.

Region	Country	Product	SalesAmount
North	USA	Laptop	2000
North	USA	Phone	1500
North	Canada	Laptop	1800
North	Canada	Phone	1200
South	Brazil	Laptop	2500
South	Brazil	Phone	3000
South	Argentina	Laptop	1800
South	Argentina	Phone	2200

Sales

```sql
SELECT Region, Country, Product, SUM(SalesAmount) as TotalSales
FROM Sales
GROUP BY ROLLUP(Region, Country, Product)
```

Region	Country	Product	TotalSales
North	Canada	Laptop	1,800
North	Canada	Phone	1,200
North	USA	Laptop	2,000
North	USA	Phone	1,500
North	NULL	NULL	6,500
South	Argentina	Laptop	1,800
South	Argentina	Phone	2,200
South	Brazil	Laptop	2,500
South	Brazil	Phone	3,000
South	NULL	NULL	9,500
NULL	NULL	NULL	16,000

Sales by Region, Country, and Product

Without Rollup

```sql
SELECT Region, Country, Product, SUM(SalesAmount) as TotalSales
FROM Sales
GROUP BY Region, Country, Product

UNION ALL

SELECT Region, Country, NULL as Product, SUM(SalesAmount) as TotalSales
FROM Sales
GROUP BY Region, Country

UNION ALL

SELECT Region, NULL as Country, NULL as Product, SUM(SalesAmount) as TotalSales
FROM Sales
GROUP BY Region
```

Multiple Rollup

```
SELECT Region, Country, SUM(SalesAmount) AS TotalSales
FROM Sales
GROUP BY ROLLUP (Region, Country), ROLLUP(Country)
```

Region	Country	TotalSales
North	Canada	1800
North	USA	3500
North	**Total**	5300
Total	Canada	3000
Total	USA	3500
Total	**Total**	6500
South	Argentina	2200
South	Brazil	5500
South	**Total**	7700
Total	Argentina	2200
Total	Brazil	5500
Total	**Total**	7700

Rollup by Region and Country, with multiple rollups on Country(Total As Null)

```sql
SELECT Region, Country, SUM(SalesAmount) AS TotalSales
FROM Sales
GROUP BY CUBE (Region, Country)
```

$$\{(Region, Country), (Region), (Country), ()\}$$

```sql
SELECT Region, Country, SUM(SalesAmount) AS TotalSales
FROM Sales
GROUP BY GROUPING SETS ((Region, Country))
```

$$\{(Region, Country)\}$$

Thanks
End of Chapter 5