Structured Query Language (SQL)

Structured Query Language (SQL) is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS). It is particularly useful in handling structured data, i.e., data incorporating relations among entities and variables.

SQL is used for the following:

- modifying database table and index structures;
- adding, updating and deleting rows of data; and
- retrieving subsets of information from within relational database management systems (RDBMSes) -- this information can be used for transaction processing, analytics applications and other applications that require communicating with a relational database.

SQL queries and other operations take the form of commands written as statements and are aggregated into programs that enable users to add, modify or retrieve data from database tables.

A table is the most basic unit of a database and consists of rows and columns of data. A single table holds records, and each record is stored in a row of the table. Tables are the most used type of database objects, or structures that hold or reference data in a relational database.

SQL Data Types

Introduction

SQL data types define the type of value that can be stored in a table column. For example, if you want a column to store only integer values, you can define its data type as `INT`.

SQL data types can be broadly divided into the following categories.

1. Numeric data types such as: `INT`, `TINYINT`, `BIGINT`, `FLOAT`, `REAL`, etc.
2. Date and Time data types such as: `DATE`, `TIME`, `DATETIME`, etc.
3. Character and String data types such as: `CHAR`, `VARCHAR`, `TEXT`, etc.
4. Unicode character string data types such as: `NCHAR`, `NVARCHAR`, `NTEXT`, etc.
5. Binary data types such as: `BINARY`, `VARBINARY`, etc.
6. Miscellaneous data types - `CLOB`, `BLOB`, `XML`, `CURSOR`, `TABLE`, etc.

SQL Data Definition

Data definition language (DDL) describes the portion of SQL that creates, alters, and deletes database objects. These database objects include schemas, tables, views, sequences, catalogs, indexes, variables, masks, permissions, and aliases.

In the context of SQL, **data definition** or **data description language** (**DDL**) is a syntax for creating and modifying database objects such as tables, indices, and users. DDL statements are similar to a computer programming language for defining data structures, especially database schemas. Common examples of DDL statements include `CREATE`, `ALTER`, and `DROP`.

CREATE TABLE statement

A commonly used *CREATE* command is the *CREATE TABLE* command. The typical usage is:

```
CREATE TABLE [table name] ( [column definitions] ) [table parameters]
```

An example statement to create a table named *employees* with a few columns is:

```
CREATE TABLE employees (
   id          INTEGER       PRIMARY KEY,
   first_name    VARCHAR(50)   not null,
```

```
    last_name     VARCHAR(75)   not null,
    mid_name      VARCHAR(50)   not null,
    dateofbirth   DATE          not null
);
```

DROP statement

The *DROP* statement destroys an existing database, table, index, or view.

```
DROP objecttype objectname.
```

For example, the command to drop a table named **employees** is:

```
DROP TABLE employees;
```

ALTER statement

The *ALTER* statement modifies an existing database object.

```
ALTER objecttype objectname parameters.
```

For example, the command to add (then remove) a column named **bubbles** for an existing table named **sink** is:

```
ALTER TABLE sink ADD bubbles INTEGER;
ALTER TABLE sink DROP COLUMN bubbles;
```

TRUNCATE statement

The *TRUNCATE* statement is used to delete all data from a table. It's much faster than *DELETE*.

```
TRUNCATE TABLE table_name;
```

Constraints, Create, Insert, Delete, and Update Statements

SQL Constraints

SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Prevents actions that would destroy links between tables
- CHECK - Ensures that the values in a column satisfies a specific condition
- DEFAULT - Sets a default value for a column if no value is specified
- CREATE INDEX - Used to create and retrieve data from the database very quickly

Create Table

The CREATE TABLE statement is used to create a new table in a database. In that table, if you want to add multiple columns, use the below syntax.

Syntax

```
1. CREATE TABLE table_name (
2.     column1 datatype,
3.     column2 datatype,
4.     column3 datatype,
5.    ....
6. );
```

The column parameters specify the names of the columns of the table.

The data type parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

Create Table Example

```
1. CREATE TABLE Employee(
2.     EmpId int,
3.     LastName varchar(255),
4.     FirstName varchar(255),
5.     Address varchar(255),
6.     City varchar(255)
7. );
```

The EmpId column is of type int and will hold an integer.

The LastName, FirstName, Address, and City columns are of type varchar and will hold characters and the maximum length for these fields is 255 characters.

Insert Value in this Table

The INSERT INTO statement is used to insert new records in a table.

It is possible to write the INSERT INTO statement in two ways.

Syntax

The first way specifies both the column names and the values to be inserted.

If you are adding values for all the columns of the table, then no need to specify the column names in the SQL query. However, make sure that the order of the values is in the same order as the columns in the table.

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);

  '2nd way
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

Example

Insert value in a 1st way. The column names are used here

```
INSERT INTO Employee    (EmpId,LastName,FirstName,ADDRESS,City)
VALUES (1, 'XYZ', 'ABC', 'India', 'Mumbai' );
  INSERT INTO Employee (EmpId,LastName,FirstName,ADDRESS,City)
VALUES (2, 'X', 'A', 'India', 'Pune' );
```

Insert value in a 2nd way.

```
INSERT INTO Employee
VALUES (3, 'XYZ', 'ABC', 'India', 'Mumbai' );
```

Select Statment in SQL

The SELECT statement is used to select data from a database.

The data returned is stored in a result table, called the result-set.

```
SELECT column1, column2, ...
FROM table_name;
```

Here, column1, column2, ... are the field names of the table you want to select from the data. If you want to select all the fields available in the table, use the following syntax:

```
SELECT * FROM table_name;
```

If the above query is executed, then all record is displayed.

Example

```
1. Select EmpId, LastName from Employee;
2.
3. Select * from Employee;
```

Update Table

The UPDATE statement is used to modify the existing records in a table.

Syntax

```
1. UPDATE table_name
2. SET column1 = value1, column2 = value2, ...
3. WHERE condition;
```

Example

```
1. UPDATE Employee
2. SET FirstName= 'KS', City= 'Pune'
3. WHERE EmpId= 1;
```

If the above query is executed then for EmpId= 1, "Firstname" and "City" column data will be updated.

Update Multiple Rows

It is the WHERE clause that determines how many records will be updated.

```
1. UPDATE Employee
2. SET City='Pune'
```

Delete Statment in SQL

The DELETE statement is used to delete existing records in a table for a particular Record.

Syntax

```
1. DELETE FROM table_name WHERE condition;
```

Example

```
1. DELETE FROM Employee WHERE EmpId=1;
```

In Employee table EmpId = 1 record gets deleted.

Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact,

```
1. DELETE FROM table_name;
2.
3. DELETE From Employee  ;
```

When the above query is executed, only table Data gets deleted.

Views, Stored Procedures and Functions

SQL Stored Procedures

SQL stored procedures are implemented as a set of T-SQL queries with a specific name (i.e. procedure name). They are stored in the RDBMS to be used and reused by various users and programs.

Stored procedures can be used to maintain the security of your data by only giving people specific access. End users may view or edit the data but not write the stored procedures themselves, so you can control how data is viewed and modified.

A stored procedure can help maintain data integrity by preventing irregularities from entering the system. It can also improve productivity by requiring less time to enter common statements.

Let's create some stored procedures on the Companies table to be used in an application with the following syntax:

Stored Procedure to Return a List of All the Companies

```
USE HRDatabase;
GO

CREATE OR ALTER PROCEDURE dbo.GetCompanies -- CREATE PROCEDURE
AS
BEGIN
    SELECT [ID]
        ,[CompanyName]
        ,[CompAddress]
        ,[CompContactNo]
        ,[CreateDate]
    FROM [dbo].[Companies]
END;

--To run the Stored Procedure you would run the following SQL code:
EXEC dbo.GetCompanies;
```

Stored Procedure to Return a Single Company Based on an ID

```
CREATE OR ALTER PROCEDURE dbo.GetCompany
  @ID int  -- input parameter
AS
BEGIN
    SELECT [ID]
```

```
          ,[CompanyName]
          ,[CompAddress]
          ,[CompContactNo]
          ,[CreateDate]
     FROM [dbo].[Companies]
     WHERE ID = @ID
END;

--To Execute Stored Procedure you would run the following SQL code:
EXEC dbo.GetCompany @ID = 3;
```

Stored Procedure to Insert a Single Company

```
CREATE OR ALTER PROCEDURE dbo.InsCompany
     @CompanyName   varchar(80), -- input parameters
     @CompAddress   varchar(80),
     @CompContactNo varchar(20)
AS
BEGIN
     INSERT INTO [dbo].[Companies]
          ([CompanyName]
          ,[CompAddress]
          ,[CompContactNo]
          ,[CreateDate])
        VALUES
          (@CompanyName
          ,@CompAddress
          ,@CompContactNo
          ,getdate())
END;

--To Execute Stored Procedure you would run the following SQL code:
EXEC dbo.InsCompany
     @CompanyName   = 'Zulu-Yankee Company',
     @CompAddress   = '123 Some street, Somewhere Far away, Europe ext 10',
     @CompContactNo= '(999) 852 7401';

SELECT * FROM dbo.Companies;
```

Stored Procedure to Update a Single Company

```
CREATE OR ALTER PROCEDURE dbo.UpdCompany
     @ID            int         = null,
     @CompanyName   varchar (80) = null,
     @CompAddress   varchar (80) = null,
     @CompContactNo varchar (20) = null
AS
BEGIN
```

```
    UPDATE dbo.Companies
    SET CompanyName   = ISNULL(@CompanyName, CompanyName)
      ,CompAddress   = ISNULL(@CompAddress, CompAddress)
      ,CompContactNo = ISNULL(@CompContactNo, CompContactNo)
    WHERE ID = @ID
END

--To Execute Stored Procedure you would run the following SQL code:
EXEC dbo.UpdCompany
    @ID           = 6,
    @CompanyName   = 'Zulu-Yanke Company',
    @CompAddress   = null,
    @CompContactNo = '(777) 852 7401'

SELECT * FROM dbo.Companies;
```

Stored Procedure to Delete a Single Company based on its ID

```
CREATE OR ALTER PROCEDURE dbo.DelCompany
    @ID int
AS
BEGIN
    DELETE FROM dbo.Companies
    WHERE ID = @ID
END;

--To Execute Stored Procedure you would run the following SQL code:
EXEC dbo.DelCompany
    @ID = 3;

SELECT * FROM dbo.Companies;
```

In the five subsections above, we have created the stored procedures for the Companies table. You can now create the same stored procedures for the Employees table and any other tables in the database.

Stored procedures should only do the following operations on data:

- Insert (Create)
- Select (Read)
- Update (Update)
- Delete (Delete)

There are two approaches regarding stored procedures:

1. Do the business logic in the stored procedure, or
2. Let the stored procedure just do the CRUD operations (in brackets above).

You decide what works best for your situation.

There may be environments where nearly all the business logic gets handled in the stored procedures, which can lead to large and bulky stored procedures that are very difficult to handle. Alternatively, there could also be an environment where the business logic is handled in the application and the stored procedures perform the CRUD operations. In your environment, it is necessary to determine how the business logic and CRUD operations will be supported.

SQL Views

SQL views are virtual tables that can be a great way to optimize the database experience. Not only are views good for defining a table without using extra storage, but they also accelerate data analysis and can provide data with extra security.

Benefits of using views:

- **Security** - Views prevent someone from seeing the underlying tables and gives the DBA the option to expose only specific data to users while protecting other data simultaneously.
- **Simplicity** - You can hide complicated queries behind views. They can be hidden from the person browsing the site, but you can reuse the queries easily.
- **Column Name Simplification or Clarification** - Using views, columns can be given aliases to make them more recognizable.
- **Additional Options** - Views can help you process more complex queries. They work in a "multi-level" query and often have more options than standard table queries.

Let us create a simple SELECT query that joins two tables together:

```
SELECT  c.CompanyName
     , e.EmployeeName
     , e.ContactNo
     , e.Email
FROM dbo.Companies c
JOIN dbo.Employees e on e.CompID = c.ID;
```

Instead of running the whole SQL query (which can become very big), create a view to return the data.

```
CREATE VIEW dbo.EmployeeDetailsView
AS

  SELECT  c.CompanyName
       , e.EmployeeName
       , e.ContactNo
```

```
      , e.Email
  FROM dbo.Companies c
  JOIN dbo.Employees e on e.CompID = c.ID;
```

After the view is created, you can do a SELECT from the view:

```
SELECT *
FROM dbo.EmployeeDetailsView
```

And you can also filter data from the view:

```
SELECT *
FROM dbo.EmployeeDetailsView
WHERE CompanyName = 'Alpha Company';
```

As seen in the examples above, it can be a good idea to have no filter in the view definition but rather be able to filter data when selecting from the view.

However, it can also be a good idea to hide data from a user that you do not want the user to see.

User-Defined SQL Functions

SQL Server functions are sets of SQL statements that execute a specific task. Their primary use is to allow common tasks to be easily replicated. The use of SQL Server functions is similar to that of functions in mathematics in that they correlate an input variable with output variables.

In SQL Server, you find four different types of functions:

- Scalar-valued Functions
- Table-valued Functions
- Aggregate Functions
- System Functions

The idea behind functions is to store them in the database and avoid writing the same code repeatedly.

Let us concentrate on the first two functions.

Create a Scalar-valued Function to Add Two Integers

```
CREATE or ALTER FUNCTION dbo.udfGetSum(@NumA int, @NumB int)
RETURNS int
AS
BEGIN
```

```
    DECLARE @SumOfNumbers int
    SELECT @SumOfNumbers = @NumA + @NumB
    RETURN @SumOfNumbers
END;

-- Check the function
SELECT dbo.udfGetSum(5,4);
```

Create a Table-valued Function to Return a Result Set

```
CREATE or ALTER FUNCTION dbo.udfGetEmployees(@CompID int)
RETURNS TABLE
AS
RETURN
    SELECT *
    FROM dbo.Companies
    WHERE ID = @CompID;

-- Check the function
SELECT * FROM dbo.udfGetEmployees(1);
```

Above are the most straightforward implementations of functions in SQL Server. You can also use functions to return some values where you do not want to re-type the code repeatedly.

Database trigger

A **database trigger** is procedural code that is automatically executed in response to certain events on a particular table or view in a database. The trigger is mostly used for maintaining the integrity of the information on the database. For example, when a new record (representing a new worker) is added to the employees table, new records should also be created in the tables of the taxes, vacations and salaries. Triggers can also be used to log historical data, for example to keep track of employees' previous salaries.

The four main types of triggers are:

1. Row-level trigger: This gets executed before or after *any column value of a row* changes
2. Column-level trigger: This gets executed before or after the *specified column* changes
3. For each row type: This trigger gets executed once for each row of the result set affected by an insert/update/delete
4. For each statement type: This trigger gets executed only once for the entire result set, but also fires each time the statement is executed.

Microsoft SQL Server[edit]

A list of all available firing events in Microsoft SQL Server for DDL triggers is available on Microsoft Docs.[2]

Performing conditional actions in triggers (or testing data following modification) is done through accessing the temporary *Inserted* and *Deleted* tables.

PostgreSQL[edit]

Introduced support for triggers in 1997. The following functionality in SQL:2003 was previously not implemented in PostgreSQL:

- SQL allows triggers to fire on updates to specific columns; As of version 9.0 of PostgreSQL this feature is also implemented in PostgreSQL.
- The standard allows the execution of a number of SQL statements other than SELECT, INSERT, UPDATE, such as CREATE TABLE as the triggered action. This can be done through creating a stored procedure or function to call CREATE TABLE.[3]

Synopsis:

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
   ON TABLE [ FOR [ EACH ] { ROW | STATEMENT } ]
   EXECUTE PROCEDURE funcname ( arguments )
```

SQL Injection

The SQL Injection is a code penetration technique that might cause loss to our database. It is one of the most practiced web hacking techniques to place malicious code in SQL statements, via webpage input. SQL injection can be used to manipulate the application's web server by malicious users.

SQL injection generally occurs when we ask a user to input their username/userID. Instead of a name or ID, the user gives us an SQL statement that we will unknowingly run on our database. For Example - we create a SELECT statement by adding a variable "demoUserID" to select a string. The variable will be fetched from user input (getRequestString).

1. demoUserI = getrequestString("UserId");
2. demoSQL = "SELECT * FROM users WHERE UserId =" +demoUserId;

Types of SQL injection attacks

SQL injections can do more harm other than passing the login algorithms. Some of the SQL injection attacks include:

4M

439

Exception Handling in Java - Javatpoint

- Updating, deleting, and inserting the data: An attack can modify the cookies to poison a web application's database query.
- It is executing commands on the server that can download and install malicious programs such as Trojans.
- We are exporting valuable data such as credit card details, email, and passwords to the attacker's remote server.
- Getting user login details: It is the simplest form of SQL injection. Web application typically accepts user input through a form, and the front end passes the user input to the back end database for processing.

Example of SQL Injection

We have an application based on employee records. Any employee can view only their own records by entering a unique and private employee ID. We have a field like an Employee ID. And the employee enters the following in the input field:

22224487 or 1=1

It will translate to:

1. **SELECT** * **from** EMPLOYEE **where** EMPLOYEE_ID == 22224487 or 1=1

The SQL code above is valid and will return EMPLOYEE_ID row from the EMPLOYEE table. The 1=1 will return all records for which this holds true. All the employee data is compromised; now, the malicious user can also similarly delete the employee records.

Example:

1. **SELECT** * **from** Employee **where** (Username == "" or 1=1) AND (**Password**="" or 1=1).

Now the malicious user can use the '=' operator sensibly to retrieve private and secure user information. So instead of the query mentioned above, the following query, when exhausted, retrieve protected data, not intended to be shown to users.

1. **SELECT** * **from** EMPLOYEE **where** (Employee_name =" " or 1=1) AND (**Password**=" " or 1=1)

How to detect SQL Injection attacks

Creating a SQL Injection attack is not difficult, but even the best and good-intentioned developers make mistakes. The detection of SQL Injection is, therefore, an essential component of creating the risk of an SQL injection attack. Web Application Firewall can detect and block basic SQL injection attacks, but we should depend on it as the sole preventive measure.

Intrusion Detection System (IDS) is both network-based and host-based. It can be tuned to detect SQL injection attacks. Network-based IDSec can monitor all connections to our database server, and flags suspicious activities. The host-based IDS can monitor web server logs and alert when something strange happens.