**Functions in SQL**

Functions in SQL are predefined or user-defined operations that can be used to perform calculations, manipulate data, or return a result based on input arguments. SQL functions are an important aspect of SQL as they provide a convenient way to apply logic within SQL queries without having to write complex code.

SQL functions are typically categorized into two types:

1. Predefined (Built-in) Functions: These are functions that are provided by the SQL database system. Examples include mathematical, string, date, and aggregate functions.
2. **User-Defined Functions (UDFs):** These are custom functions created by the user to perform specific tasks that are not available through the built-in functions.

**2. User-Defined Functions (UDFs) in SQL**

User-Defined Functions allow you to create custom functions that perform specific tasks. There are two types of UDFs:

1. Scalar Functions: These return a single value, such as a calculation or transformation on data.
2. Table-Valued Functions (TVFs): These return a table, which can be used in a SELECT query like a regular table.

1. **Scalar User-Defined Function (UDF)**

**Ex-1 This UDF calculates the total price including a sales tax.**

CREATE FUNCTION CalculateTax (@price DECIMAL(10, 2), @tax\_rate DECIMAL(4, 2))

RETURNS DECIMAL(10, 2)

AS

BEGIN

RETURN @price + (@price \* @tax\_rate / 100)

END

Usage:

SELECT dbo.CalculateTax(100, 8); -- Result: 108.00

**Ex. 2 This simple function will calculate the area of a rectangle given its length and width.**

CREATE FUNCTION dbo.fn\_area\_of\_rectangle

(

@length FLOAT,

@width FLOAT

)

RETURNS FLOAT

AS

BEGIN

RETURN @length \* @width

END

* **Usage**: You can now use this function in your queries to calculate the area of a rectangle.

SELECT dbo.fn\_area\_of\_rectangle(10, 5) AS area; -- Result: 50

**2.Table-Valued Function (TVF)**

This TVF returns a table of employees based on the department passed as a parameter.

CREATE FUNCTION GetEmployeesByDepartment (@department\_name NVARCHAR(100))

RETURNS TABLE

AS

RETURN

(

SELECT employee\_id, name

FROM employees

WHERE department = @department\_name

)

Usage:

SELECT \* FROM dbo.GetEmployeesByDepartment('HR');

Assuming the employees table contains the following data:

|  |  |  |
| --- | --- | --- |
| **employee\_id** | **name** | **department** |
| 1 | John | HR |
| 2 | Alice | IT |
| 3 | Bob | HR |
| 4 | Charlie | Sales |
| 5 | David | HR |

**The result would be:**

|  |  |
| --- | --- |
| **employee\_id** | **name** |
| 1 | John |
| 3 | Bob |
| 5 | David |

**Real-Life Example of User-Defined Functions (UDFs) in SQL:**

**User-Defined Functions (UDFs)** allow you to create custom functions in SQL that encapsulate complex logic, making it reusable and modular. These functions can be scalar (returning a single value) or table-valued (returning a table). Here’s a look at real-life examples for both types of UDFs:

**1. Scalar UDF Example: Calculating Tax on a Purchase**

Imagine you're running an e-commerce platform where you need to calculate the tax on customer purchases based on their location. You can create a **scalar UDF** to calculate tax rates.

**Example:**

CREATE FUNCTION dbo.CalculateTax(@purchase\_amount DECIMAL, @state NVARCHAR(50))

RETURNS DECIMAL

AS

BEGIN

DECLARE @tax\_rate DECIMAL;

-- Tax rates based on state

IF @state = 'California'

SET @tax\_rate = 0.075;

ELSE IF @state = 'Texas'

SET @tax\_rate = 0.065;

ELSE

SET @tax\_rate = 0.05; -- Default tax rate

-- Return the calculated tax amount

RETURN @purchase\_amount \* @tax\_rate;

END;

**Usage:**

SELECT product\_name, price,

dbo.CalculateTax(price, 'California') AS tax\_amount

FROM products

WHERE category = 'Electronics';

**Sample Output:**

|  |  |  |
| --- | --- | --- |
| **product\_name** | **price** | **tax\_amount** |
| Laptop | 1000.00 | 75.00 |
| Smartphone | 500.00 | 37.50 |
| Headphones | 150.00 | 11.25 |

This is **not** a table-valued function; instead, the query generates a result set with multiple rows.

**Real-Life Application:**

* In e-commerce platforms or retail systems, UDFs can be used to calculate dynamic tax rates based on various factors (location, product type, customer type). This simplifies queries and ensures consistency in tax calculations across the system.

**2. Table-Valued UDF Example: Returning Customer Orders**

Imagine you are a manager and need to retrieve all the orders placed by a specific customer, but only those that are "shipped" and above a certain amount. This logic can be encapsulated in a **table-valued UDF**.

**Example:**

CREATE FUNCTION dbo.GetCustomerOrders(@customer\_id INT, @min\_amount DECIMAL)

RETURNS TABLE

AS

RETURN

(

SELECT order\_id, order\_date, total\_amount

FROM orders

WHERE customer\_id = @customer\_id

AND total\_amount >= @min\_amount

AND status = 'Shipped'

)

**Usage:**

SELECT \*

FROM dbo.GetCustomerOrders(12345, 100)

WHERE total\_amount > 200;

Let’s assume the orders table contains the following data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **order\_id** | **order\_date** | **total\_amount** | **customer\_id** | **status** |
| 1 | 2024-01-01 | 150 | 12345 | Shipped |
| 2 | 2024-01-05 | 250 | 12345 | Shipped |
| 3 | 2024-01-10 | 80 | 12345 | Shipped |
| 4 | 2024-02-01 | 300 | 12345 | Shipped |
| 5 | 2024-02-05 | 120 | 67890 | Shipped |

**Output for the Query:**

SELECT \*

FROM dbo.GetCustomerOrders(12345, 100)

WHERE total\_amount > 200;

**Steps:**

* The function dbo.GetCustomerOrders(12345, 100) will return orders for customer 12345 where the total\_amount >= 100 and the status is 'Shipped':
  + Order 1: Total = 150 (Meets condition)
  + Order 2: Total = 250 (Meets condition)
  + Order 3: Total = 80 (Does not meet condition, as total\_amount >= 100 is required)
* The WHERE total\_amount > 200 filter is applied on the result of the function, so only orders with a total\_amount greater than 200 will be shown.

**Final Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **order\_id** | **order\_date** | **total\_amount** | **customer\_id** | **status** |
| 2 | 2024-01-05 | 250 | 12345 | Shipped |
| 4 | 2024-02-01 | 300 | 12345 | Shipped |

This is the result set after applying both the function and the additional WHERE clause filtering.

**Real-Life Application:**

* In a customer relationship management (CRM) system, a table-valued UDF like this can help easily retrieve customer-specific data, such as order history or outstanding payments, with minimal code repetition and cleaner queries.

**3. Scalar UDF Example: Determining Customer Loyalty Tier**

In a retail loyalty program, customers can be assigned a tier based on their total spending over the past year. You can create a **scalar UDF** to determine the loyalty tier based on the amount spent.

**Example:**

CREATE FUNCTION dbo.DetermineLoyaltyTier(@total\_spent DECIMAL)

RETURNS NVARCHAR(50)

AS

BEGIN

DECLARE @loyalty\_tier NVARCHAR(50);

-- Determine loyalty tier based on spending

IF @total\_spent >= 1000

SET @loyalty\_tier = 'Gold';

ELSE IF @total\_spent >= 500

SET @loyalty\_tier = 'Silver';

ELSE

SET @loyalty\_tier = 'Bronze';

RETURN @loyalty\_tier;

END;

**Usage:**

SELECT customer\_id, dbo.DetermineLoyaltyTier(total\_spent) AS loyalty\_tier

FROM customers;

Let’s assume the customers table contains the following data:

|  |  |
| --- | --- |
| **customer\_id** | **total\_spent** |
| 101 | 1200 |
| 102 | 700 |
| 103 | 350 |
| 104 | 1500 |
| 105 | 450 |

**Output for the Query:**

SELECT customer\_id, dbo.DetermineLoyaltyTier(total\_spent) AS loyalty\_tier

FROM customers;

**Execution:**

* For customer\_id = 101 with total\_spent = 1200, the tier will be 'Gold' because the spending is greater than or equal to 1000.
* For customer\_id = 102 with total\_spent = 700, the tier will be 'Silver' because the spending is between 500 and 999.
* For customer\_id = 103 with total\_spent = 350, the tier will be 'Bronze' because the spending is less than 500.
* For customer\_id = 104 with total\_spent = 1500, the tier will be 'Gold' because the spending is greater than or equal to 1000.
* For customer\_id = 105 with total\_spent = 450, the tier will be 'Bronze' because the spending is less than 500.

**Final Output:**

|  |  |
| --- | --- |
| **customer\_id** | **loyalty\_tier** |
| 101 | Gold |
| 102 | Silver |
| 103 | Bronze |
| 104 | Gold |
| 105 | Bronze |

This is the result set with the loyalty tier for each customer based on their total spending.

**Real-Life Application:**

* In a loyalty program, a scalar UDF like this could be used to determine and assign customer tiers based on their spending. This helps to easily categorize customers and offer personalized discounts or rewards.

**4. Real-Life Benefits of UDFs:**

* **Encapsulation of Business Logic:** You can encapsulate business rules, like calculating taxes or loyalty tiers, within UDFs, making them easier to maintain.
* **Reusability:** Once created, UDFs can be used in multiple queries, reducing code repetition and improving consistency.
* **Simplified Queries:** UDFs abstract away complex logic, allowing queries to remain simple and readable.
* **Modular Design:** UDFs promote a modular approach to code, making it easier to update business logic in one place (i.e., the function definition) rather than throughout all queries.

**Simple TVF Example (Inline Table-Valued Function)**

CREATE FUNCTION dbo.fn\_get\_employee\_salary\_by\_id

(

@employee\_id INT

)

RETURNS TABLE

AS

RETURN

(

SELECT employee\_id, name, salary

FROM employees

WHERE employee\_id = @employee\_id

)

* **Usage**:

SELECT \*

FROM dbo.fn\_get\_employee\_salary\_by\_id(1001);

**Complex TVF Example (Multi-Statement Table-Valued Function)**

A function that calculates and returns a table of employees' names and their cumulative salary over a period.

CREATE FUNCTION dbo.fn\_get\_salary\_over\_period

(

@start\_date DATE,

@end\_date DATE

)

RETURNS @SalaryTable TABLE

(

employee\_id INT,

name NVARCHAR(100),

total\_salary FLOAT

)

AS

BEGIN

INSERT INTO @SalaryTable

SELECT e.employee\_id, e.name, SUM(s.salary)

FROM employees e

JOIN salaries s ON e.employee\_id = s.employee\_id

WHERE s.payment\_date BETWEEN @start\_date AND @end\_date

GROUP BY e.employee\_id, e.name

RETURN

END

* **Usage**:

SELECT \*

FROM dbo.fn\_get\_salary\_over\_period('2024-01-01', '2024-12-31');

**Sample Data**

**employees Table**

|  |  |
| --- | --- |
| **employee\_id** | **name** |
| 1 | Alice |
| 2 | Bob |
| 3 | Charlie |

**salaries Table**

|  |  |  |
| --- | --- | --- |
| **employee\_id** | **salary** | **payment\_date** |
| 1 | 5000 | 2024-01-15 |
| 1 | 5000 | 2024-02-15 |
| 2 | 4500 | 2024-03-15 |
| 3 | 4000 | 2024-04-15 |
| 3 | 4000 | 2024-05-15 |

**Query Execution:**

SELECT \*

FROM dbo.fn\_get\_salary\_over\_period('2024-01-01', '2024-12-31');

**Output**:

|  |  |  |
| --- | --- | --- |
| **employee\_id** | **name** | **total\_salary** |
| 1 | Alice | 10000 |
| 2 | Bob | 4500 |
| 3 | Charlie | 8000 |

**Real-Life Applications**

1. **Payroll System**:
   * Calculate cumulative employee earnings over a specific period for reporting or auditing purposes.
2. **Project Budgeting**:
   * Determine total salary expenses for employees working on a specific project or department during a given period.
3. **Performance Incentives**:
   * Identify employees with the highest cumulative earnings for bonuses or performance reviews.

**Real-Life Usage of UDFs and TVFs**

1. **Simplifying Repetitive Logic**:
   * In daily business scenarios, certain operations (like calculating taxes, discounts, or bonuses) are repeated across multiple reports and queries. **UDFs** can be created for these repetitive tasks, making the code easier to maintain.
   * Example: A **UDF** to calculate the price after a discount for an e-commerce store.
2. **Business Intelligence**:
   * Complex calculations involving aggregates, filtering, and joins are often required in **reporting**. Instead of manually repeating complex SQL queries, **TVFs** can provide reusable, flexible data sets.
   * Example: A **TVF** to get employee salary data for a specific year.
3. **Encapsulation of Logic**:
   * UDFs and TVFs allow you to encapsulate business logic inside the database, improving **security** (by restricting access to the actual query) and **performance** (by running operations closer to the data).
4. **Dynamic Query Generation**:
   * TVFs can return tables that can be dynamically used in other queries, making them powerful for applications requiring dynamic data processing and reporting.

**Comparison Table: Simple vs. Complex (UDF vs. TVF)**

|  |  |  |
| --- | --- | --- |
| Feature | Simple UDF (Scalar) | Complex UDF (TVF) |
| Returns | Single value (scalar) | Table |
| Use | Simple calculations (e.g., taxes, discounts) | Complex data retrieval with joins or aggregates |
| Performance | Faster for simple operations | May have performance overhead due to multiple operations |
| Example | Calculate area, tax, or discount | Get employees by department or salary summary |
| Flexibility | Limited to scalar values | Very flexible for complex queries |

In summary, UDFs and TVFs allow us to encapsulate logic for repeated use, improving both performance and maintainability. They are essential for any application that requires frequent, reusable queries or complex data transformations.