HiveQL 1: CTE/Views/Temporary table/UDFs

Writing queries in Hive is pretty straightforward, most of the syntax is the same as in SQL. But as SQL is mainly meant for transactional queries and Hive is meant for analytical queries, so tehre are few additional features that are added by Hive community to ensure that we can write Hive queries in efficient manner. In this and subsequent few lectures, we are going to discuss few of such awesome features.

CTE

When you write queries for actual production use cases, you are going to have multiple joins in datasets, in some cases, these joins might come in the range of 12-15 datasets in a single query. Some of these joins results will be reused again and then you can imagine how complex a query can become. To solve this issue, Hive introduced something called CTE. So Common Table Expressions (CTEs) are basically a temporary result sets defined within the execution scope of a single SQL statement. They are used to simplify complex queries by breaking them down into smaller, more manageable parts.

Structure of a CTE

Basic Syntax:

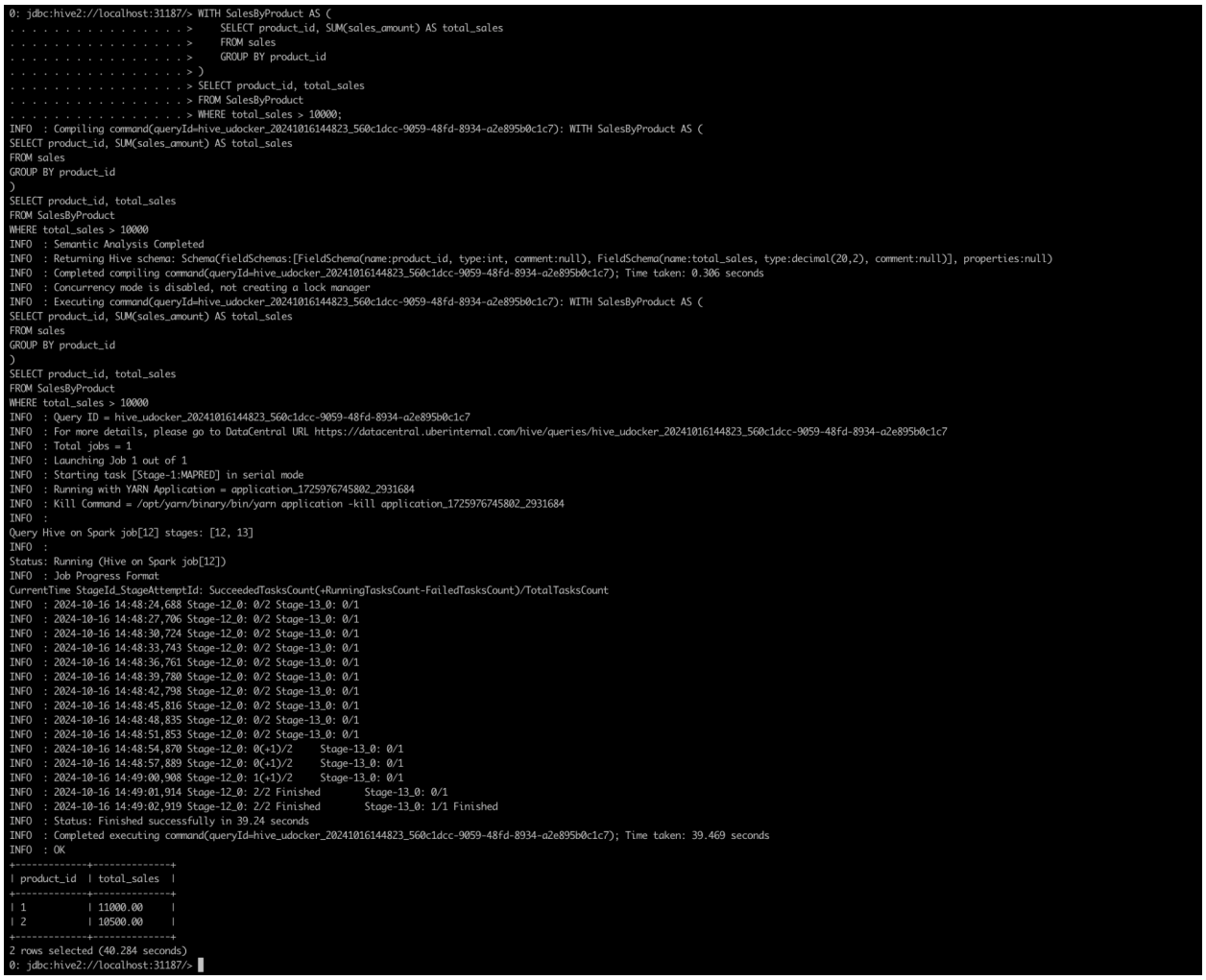
```
WITH cte_name AS (
   SELECT column1, column2, ...
   FROM table_name
   WHERE condition
)
SELECT * FROM cte_name WHERE another_condition;
```

Explanation:

- WITH: Keyword that introduces the CTE.
- cte_name: A temporary name given to the result set produced by the subquery.
- The subquery inside the CTE can reference any table or other CTE defined earlier.
- The main query that follows can reference the CTE by its name.

Example:

```
WITH SalesByProduct AS (
    SELECT product_id, SUM(sales_amount) AS total_sales
    FROM sales
    GROUP BY product_id
)
SELECT product_id, total_sales
FROM SalesByProduct
WHERE total_sales > 10000;
```



Explanation:

- The CTE SalesByProduct calculates the total sales for each product.
- The main query then filters out products with total sales exceeding 10,000.

Now this is a simple query and you might be wondering what does it actually solve. Now lets use multiple CTE and see how its useful.

Example: Using multiple CTEs to process sales data.

Original query:

```
SELECT customer_id, total_spent, avg_order_value, rank
FROM (
    SELECT customer_id, total_spent,
           total_spent / order_count AS avg_order_value,
           RANK() OVER (ORDER BY total_spent DESC) AS rank
    FROM (
        SELECT customer_id, SUM(order_amount) AS total_spent,
               COUNT(order_id) AS order_count
        FROM orders
        GROUP BY customer_id
        HAVING COUNT(order_id) > 5
    ) AS customer_orders
    WHERE total_spent > 1000
) AS ranked_customers
ORDER BY rank;
```

CTE query:

```
WITH CustomerOrderSummary AS (
    SELECT customer_id, SUM(order_amount) AS total_spent,
           COUNT(order_id) AS order_count
    FROM orders
```

```
    GROUP BY customer_id
    HAVING COUNT(order_id) > 5
),
FilteredCustomers AS (
    SELECT customer_id, total_spent,
           total_spent / order_count AS avg_order_value
    FROM CustomerOrderSummary
    WHERE total_spent > 1000
),
RankedCustomers AS (
    SELECT customer_id, total_spent, avg_order_value,
           RANK() OVER (ORDER BY total_spent DESC) AS rank
    FROM FilteredCustomers
)
SELECT customer_id, total_spent, avg_order_value, rank
FROM RankedCustomers
ORDER BY rank;
```

```
INFO  : Executing command(queryId=hive_udocker_20241016145008_858cfefa-8c4c-40b5-9a2c-b75683643a84): WITH CustomerOrderSummary AS (
SELECT customer_id, SUM(order_amount) AS total_spent,
COUNT(order_id) AS order_count
FROM orders
GROUP BY customer_id
HAVING COUNT(order_id) > 5
),
FilteredCustomers AS (
SELECT customer_id, total_spent,
total_spent / order_count AS avg_order_value
FROM CustomerOrderSummary
WHERE total_spent > 1000
),
RankedCustomers AS (
SELECT customer_id, total_spent, avg_order_value,
RANK() OVER (ORDER BY total_spent DESC) AS rank
FROM FilteredCustomers
)
SELECT customer_id, total_spent, avg_order_value, rank
FROM RankedCustomers
ORDER BY rank
INFO  : Query ID = hive_udocker_20241016145008_858cfefa-8c4c-40b5-9a2c-b75683643a84
INFO  : For more details, please go to DataCentral URL https://datacentral.uberinternal.com/hive/queries/hive_udocker_20241016145008_858cfefa-8c4c-40b5-9a2c-b75683643a84
INFO  : Total jobs = 1
INFO  : Launching Job 1 out of 1
INFO  : Starting task [Stage-1:MAPRED] in serial mode
INFO  : Running with YARN Application = application_1725976745802_2931684
INFO  : Kill Command = /opt/yarn/binary/bin/yarn application -kill application_1725976745802_2931684
INFO  :
Query Hive on Spark job[13] stages: [15, 16, 17, 14]
INFO  :
Status: Running (Hive on Spark job[13])
INFO  : Job Progress Format
CurrentTime StageId_StageAttemptId: SucceededTasksCount(+RunningTasksCount-FailedTasksCount)/TotalTasksCount
INFO  : 2024-10-16 14:50:13,260 Stage-14_0: 0/2 Stage-15_0: 0/1 Stage-16_0: 0/1 Stage-17_0: 0/1
INFO  : 2024-10-16 14:50:16,286 Stage-14_0: 0/2 Stage-15_0: 0/1 Stage-16_0: 0/1 Stage-17_0: 0/1
INFO  : 2024-10-16 14:50:19,315 Stage-14_0: 0/2 Stage-15_0: 0/1 Stage-16_0: 0/1 Stage-17_0: 0/1
INFO  : 2024-10-16 14:50:22,340 Stage-14_0: 0/2 Stage-15_0: 0/1 Stage-16_0: 0/1 Stage-17_0: 0/1
INFO  : 2024-10-16 14:50:25,366 Stage-14_0: 0/2 Stage-15_0: 0/1 Stage-16_0: 0/1 Stage-17_0: 0/1
INFO  : 2024-10-16 14:50:28,394 Stage-14_0: 0/2 Stage-15_0: 0/1 Stage-16_0: 0/1 Stage-17_0: 0/1
INFO  : 2024-10-16 14:50:31,419 Stage-14_0: 0/2 Stage-15_0: 0/1 Stage-16_0: 0/1 Stage-17_0: 0/1
INFO  : 2024-10-16 14:50:33,437 Stage-14_0: 0(+1)/2     Stage-15_0: 0/1 Stage-16_0: 0/1 Stage-17_0: 0/1
INFO  : 2024-10-16 14:50:36,464 Stage-14_0: 0(+1)/2     Stage-15_0: 0/1 Stage-16_0: 0/1 Stage-17_0: 0/1
INFO  : 2024-10-16 14:50:39,490 Stage-14_0: 2/2 Finished        Stage-15_0: 0/1 Stage-16_0: 0/1 Stage-17_0: 0/1
INFO  : 2024-10-16 14:50:40,500 Stage-14_0: 2/2 Finished        Stage-15_0: 1/1 Finished        Stage-16_0: 1/1 Finished        Stage-17_0: 0/1
INFO  : 2024-10-16 14:50:41,511 Stage-14_0: 2/2 Finished        Stage-15_0: 1/1 Finished        Stage-16_0: 1/1 Finished        Stage-17_0: 1/1 Finished
INFO  : Status: Finished successfully in 32.27 seconds
INFO  : Completed executing command(queryId=hive_udocker_20241016145008_858cfefa-8c4c-40b5-9a2c-b75683643a84); Time taken: 32.45 seconds
INFO  : OK
+--------------+--------------+--------------------------+------+
| customer_id  | total_spent  |      avg_order_value     | rank |
+--------------+--------------+--------------------------+------+
| 103          | 3600.00      | 514.2857142857142857429  | 1    |
| 101          | 1300.00      | 216.66666666666666666667 | 2    |
+--------------+--------------+--------------------------+------+
```

CTE vs. Subqueries

- **Comparison**:
    - **Subqueries**: Embedded directly within the main query; often less readable.
    - **CTEs**: Defined separately, improving clarity and organization.
- **Advantages of CTEs over Subqueries**:
    - **Readability**: CTEs provide a clear, named structure that can be referenced later in the query.
    - **Reuse**: A CTE can be referenced multiple times within the same query, unlike a subquery.
    - **Maintenance**: Updating a CTE is simpler since it's defined once and used throughout the query.

Performance Considerations

- **Execution Plan**: CTEs do not inherently improve performance but can help optimize query execution by simplifying the query structure.
- **Materialization**: In some database systems, CTEs might be materialized (i.e the result of the underlying query will be computed and stored, and retrieved later when the same price of query ix executed)before the main query

executes, which can either help or hinder performance depending on the scenario.

Views

Syntax

CREATE VIEW view_name AS SELECT column1, column2, ... FROM table_name WHERE condition;

As we saw in CTE, how CTE reduces the complexity of the query, but CTE is something which can only be reused in the same query. But there might be some cases in which a certain piece of query be reused across multiple queries and these queries in which it will be reused need not be part of same application.I know its tough, so lets understand this problem with a usecase.

Scenario:

Imagine you are working for an e-commerce company that wants to build a reporting dashboard to monitor sales performance. The dashboard needs to display various metrics, including total sales, average order value, customer retention rates, and top-selling products across different regions and time periods. These metrics are derived from multiple tables like ***orders, customers, products, and regions***. Now we might need to create multiple dashboards like these .

Now one solution, is to create separate complex queries for each of the metrics in dashboard, in which even if we want to reduce complexity, we will write CTE, but again, we would be writing same CTE across multiple queries, so instead of that, we would create views. You can treat view as a special CTE which is persisted in HMS and can be used across queries. You can also treat it as a table whose select query is predefined and results are not stored anywhere unlike table and need to be computed everytime.

So here is how we would use views

View 1: Total Sales by Region

CREATE VIEW total_sales_by_region AS SELECT r.region_name, SUM(o.order_amount) AS total_sales FROM orders o' JOIN customers c ON o.customer_id = c.customer_id JOIN regions r ON c.region_id = r.region_id GROUP BY r.region_name;

View 2: Average Order Value by Product

```
CREATE VIEW avg_order_value_by_product_and_region AS SELECT r.region_name, p.product_name, AVG(o.order_amount) AS avg_order_value FROM orders o JOIN products p ON o.product_id = p.product_id JOIN customers c ON o.customer_id = c.customer_id JOIN regions r ON c.region_id = r.region_id GROUP BY r.region_name, p.product_name;
```

View 3: Customer Retention Rate

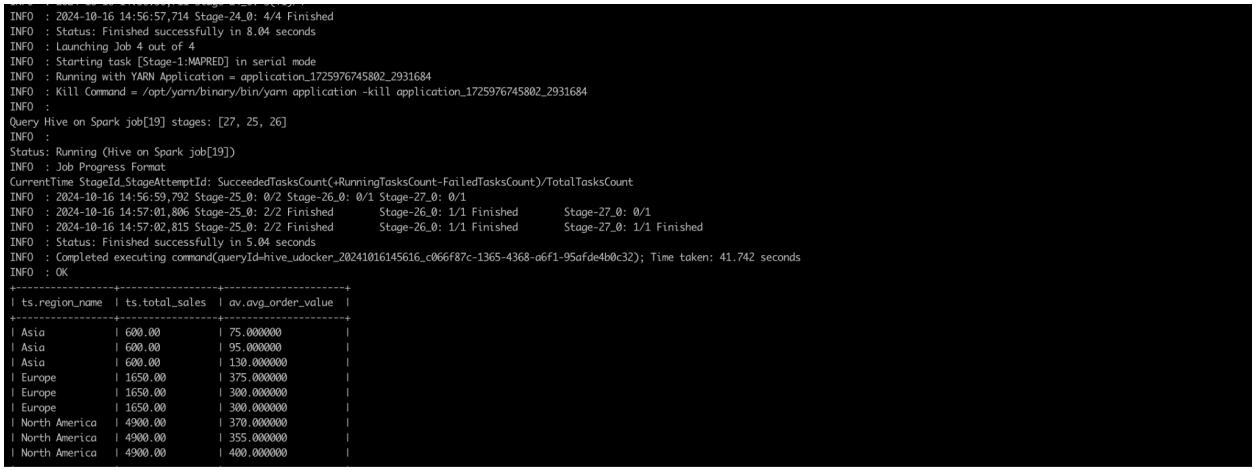
```
CREATE VIEW customer_retention_rate AS SELECT c.customer_id, COUNT(o.order_id) AS number_of_orders, CASE WHEN COUNT(o.order_id) > 1 THEN 'Retained' ELSE 'New' END AS retention_status FROM orders o JOIN customers c ON o.customer_id = c.customer_id GROUP BY c.customer_id;
```

Use Views in Dashboard Queries

Now that the views are created, you can use them in your dashboard queries to fetch the required data easily.

Example: Fetching Data for the Sales Dashboard

```
SELECT ts.region_name, ts.total_sales, av.avg_order_value FROM total_sales_by_region ts JOIN avg_order_value_by_product_and_region av ON ts.region_name = av.region_name ORDER BY ts.region_name;
```



```
INFO  : 2024-10-16 14:56:57,714 Stage-24_0: 4/4 Finished
INFO  : Status: Finished successfully in 8.04 seconds
INFO  : Launching Job 4 out of 4
INFO  : Starting task [Stage-1:MAPRED] in serial mode
INFO  : Running with YARN Application = application_1725976745802_2931684
INFO  : Kill Command = /opt/yarn/binary/bin/yarn application -kill application_1725976745802_2931684
INFO  :
Query Hive on Spark job[19] stages: [27, 25, 26]
INFO  :
Status: Running (Hive on Spark job[19])
INFO  : Job Progress Format
CurrentTime StageId_StageAttemptId: SucceededTasksCount(+RunningTasksCount-FailedTasksCount)/TotalTasksCount
INFO  : 2024-10-16 14:56:59,792 Stage-25_0: 0/2 Stage-26_0: 0/1 Stage-27_0: 0/1
INFO  : 2024-10-16 14:57:01,806 Stage-25_0: 2/2 Finished       Stage-26_0: 1/1 Finished       Stage-27_0: 0/1
INFO  : 2024-10-16 14:57:02,815 Stage-25_0: 2/2 Finished       Stage-26_0: 1/1 Finished       Stage-27_0: 1/1 Finished
INFO  : Status: Finished successfully in 5.04 seconds
INFO  : Completed executing command(queryId=hive_udocker_20241016145616_c066f87c-1365-4368-a6f1-95afde4b0c32); Time taken: 41.742 seconds
INFO  : OK
+-----------------+-----------------+---------------------+
| ts.region_name  | ts.total_sales  | av.avg_order_value  |
+-----------------+-----------------+---------------------+
| Asia            | 600.00          | 75.000000           |
| Asia            | 600.00          | 95.000000           |
| Asia            | 600.00          | 130.000000          |
| Europe          | 1650.00         | 375.000000          |
| Europe          | 1650.00         | 300.000000          |
| Europe          | 1650.00         | 300.000000          |
| North America   | 4900.00         | 370.000000          |
| North America   | 4900.00         | 355.000000          |
| North America   | 4900.00         | 400.000000          |
```

Limitations of Views in Hive

- **No Materialization**: Hive views are not materialized, meaning they do not store data themselves. Every time a view is queried, the underlying SELECT statement is executed. This can lead to performance issues if the underlying query is complex or runs over a large dataset.

- **Limited Use with Indexes**: Hive doesn't support indexing on views, which can limit the performance optimization options available when querying large datasets.
- **Dependency Management**: When you drop a table that is referenced by a view, the view becomes invalid. You need to manually manage dependencies between tables and views

In a nutshell,

Use CTEs When:

- You need to break down a complex query into more manageable parts within a single execution.
- The transformation or aggregation logic is only needed for a single query or is too specific to be reused.

Use Views When:

- You want to create a reusable, persistent abstraction over complex SQL logic.
- You need to share a simplified, consistent data interface across multiple queries or users.

Temporary Table

The temporary table can be treated as materialized view for a single Hive session, and as soon as HiveSession is deleted, the table gets dropped and can't be used.

Temporary tables are used to hold data temporarily for the duration of a session or transaction. They are particularly useful when dealing with complex queries, large datasets, or when you need to perform multiple operations on a subset of data without altering the base tables.

Syntax for Creating Temporary Tables

Basic Syntax:

```
CREATE TEMPORARY TABLE temp_table_name (
   column1 datatype,
   column2 datatype,
   ...
);
```

Example:

```
CREATE TEMPORARY TABLE temp_sales_summary (
   product_id INT,
   total_sales DECIMAL(10, 2)
);
```

Syntax for Inserting Data into a Temporary Table:

```
INSERT INTO temp_sales_summary (product_id, total_sales)
SELECT product_id, SUM(sales_amount)
FROM sales
GROUP BY product_id;
```

Using a Temporary Table:

```
SELECT * FROM temp_sales_summary WHERE total_sales > 10000;
```

Characteristics of Temporary Tables

- **Session or Transaction Scoped**:
    - **Session Scoped**: A temporary table exists for the duration of the user session. Once the session ends, the temporary table is automatically dropped.
    - **Transaction Scoped**: A temporary table can also be created to exist only within a specific transaction. Once the transaction ends, the temporary table is dropped.
- **Isolation from Other Sessions**: Temporary tables are unique to each session. Multiple users can create temporary tables with the same name, and they will not conflict with each other because each session maintains its own set of temporary tables.
- **Automatic Cleanup**: Temporary tables are automatically dropped when the session or transaction ends, eliminating the need for manual cleanup.
- **Storage**: While temporary tables are stored in the database's temporary storage, the data stored in them can still be indexed and manipulated like data in regular tables.

Common Use Cases for Temporary Tables

- **Staging Data**: Temporary tables are often used as staging areas for data that is being cleaned, transformed, or aggregated before it is inserted into a final destination table.
- **Simplifying Complex Queries**: When dealing with complex queries that involve multiple subqueries, joins, or aggregations, temporary tables can break down the query into more manageable steps.

- **Performance Optimization**: In some cases, using a temporary table can improve query performance by storing intermediate results, thereby reducing the need for repetitive calculations.
- **Testing and Debugging**: Temporary tables are useful for testing and debugging SQL queries. You can store intermediate results in a temporary table and examine the data before proceeding with further operations.

Temporary Tables vs. Other SQL Concepts

- **Temporary Tables vs. CTEs (Common Table Expressions)**:
    - **Scope**: Temporary tables persist for the duration of a session or transaction, while CTEs exist only within the execution of a single query.
    - **Usage**: Temporary tables are more suitable for scenarios where you need to perform multiple operations on the intermediate data, whereas CTEs are ideal for breaking down a single complex query.
- **Temporary Tables vs. Views**:
    - **Data Storage**: Temporary tables store data, whereas views are virtual tables that do not store data.
    - **Persistence**: Temporary tables are temporary and session-bound, while views are permanent objects in the database that can be reused across sessions.

Best Practices for Using Temporary Tables

- **Name Your Temporary Tables Clearly**: Use clear, descriptive names for your temporary tables to avoid confusion, especially in complex queries with multiple temp tables.
- **Monitor Performance**: While temporary tables can improve performance, they also consume memory and temporary storage. Monitor their usage to ensure they are not negatively impacting the database performance.
- **Drop Temporary Tables Explicitly When Necessary**: Although temporary tables are automatically dropped at the end of a session, you can drop them explicitly if they are no longer needed before the session ends.
- **Use Indexes Sparingly**: You can create indexes on temporary tables to improve query performance, but be mindful that indexing can also add overhead.

Case Study for better understanding

Scenario Overview: You are working on a data pipeline that processes daily sales data from multiple sources. The raw data is often messy, containing duplicates, missing values, and inconsistent records. Before loading this data into the main data warehouse, you need to perform a series of data quality checks and transformations.

Temporary tables are ideal for this use case, allowing you to isolate each step of the data cleansing and transformation process without affecting the main database schema.

Objective: The goal is to clean and transform the raw sales data, ensuring it meets the required quality standards, and then load the cleaned data into a permanent table in the data warehouse.

Solution

Extract and Load Raw Data into a Staging Table

Raw data from multiple sources is loaded into a staging table, raw_sales_data.

```
CREATE TABLE raw_sales_data (
    transaction_id INT,
    product_id INT,
    sale_date DATE,
    sale_amount DECIMAL(10, 2),
    customer_id INT,
    source_system VARCHAR(50)
);
```

Create a Temporary Table for Data Deduplication

The raw data might contain duplicate records. Create a temporary table to store the deduplicated data.

```
CREATE TEMPORARY TABLE deduped_sales_data AS
SELECT DISTINCT transaction_id, product_id, sale_date, sale_amount, customer_id,
source_system
FROM raw_sales_data;
```

Create a Temporary Table for Missing Value Handling

Identify records with missing values (e.g., NULL values) and create a temporary table to hold cleaned records. For example, if sale_amount is missing, assume a default value of 0.

```
CREATE TEMPORARY TABLE cleaned_sales_data AS
SELECT transaction_id, product_id, sale_date,
    COALESCE(sale_amount, 0) AS sale_amount,
    customer_id, source_system
```

```
FROM deduped_sales_data;
```

Create a Temporary Table for Data Enrichment

Suppose you need to enrich the sales data by adding product details from another table, product_details. Use a temporary table to join and enrich the data.

```
CREATE TEMPORARY TABLE enriched_sales_data AS
SELECT c.transaction_id, c.product_id, p.product_name, c.sale_date, c.sale_amount,
c.customer_id, c.source_system
FROM cleaned_sales_data c
JOIN product_details p ON c.product_id = p.product_id;
```

Create a Temporary Table for Data Quality Checks

Perform additional data quality checks, such as ensuring that sale_amount is positive and that all product_ids exist in the product_details table.

```
CREATE TEMPORARY TABLE quality_checked_sales_data AS
SELECT *
FROM enriched_sales_data
WHERE sale_amount > 0
AND product_id IN (SELECT product_id FROM product_details);
```

Load the Final Cleaned Data into the Data Warehouse

Finally, load the cleaned, deduplicated, and enriched data into the permanent final_sales_data table in the data warehouse.

```
INSERT INTO final_sales_data
SELECT * FROM quality_checked_sales_data;
```

Drop Temporary Tables (Optional)

While temporary tables are automatically dropped at the end of the session, you can explicitly drop them if they are no longer needed before the session ends.

```
DROP TEMPORARY TABLE IF EXISTS deduped_sales_data;
DROP TEMPORARY TABLE IF EXISTS cleaned_sales_data;
DROP TEMPORARY TABLE IF EXISTS enriched_sales_data;
DROP TEMPORARY TABLE IF EXISTS quality_checked_sales_data;
```

UDF (User defined functions)

You would have learnt about various functions when writing SQL like sum, avg, min, lower, etc, which enables you to transform/aggregate the data. Hive have some more of these functions which allows you to do many other transformations on data, but there can be a custom usecase in which you would want to add your custom logic to transform the data, there you would require a custom function which executes your custom code, this custom code nad function is called UDF. So basically User-Defined Function (UDF) in Hive is a custom function created by the user to extend the capabilities of Hive's SQL-like query language. UDFs allow users to implement specific logic that isn't available in Hive's built-in functions. UDFs are used to perform complex transformations, calculations, or data manipulations that go beyond the standard functions provided by Hive.

Why Use UDFs?

- **Customization**: When the built-in functions in Hive do not meet your specific needs, UDFs provide a way to create tailored solutions.
- **Reusability**: UDFs can be reused across different queries and datasets, making your code more modular and maintainable.
- **Extending Functionality**: UDFs extend Hive's capabilities, allowing you to process data in ways that would otherwise be impossible or cumbersome with just the built-in functions.

Creating a UDF in Hive

- **Language Support**:
    - **Java**: The most common language for writing UDFs in Hive is Java.
    - **Python (Using PyHive)**: UDFs can also be written in Python, though this is less common and requires additional configuration.

Basic Structure of a Hive UDF in Java:

```
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public class MyUpperCase extends UDF {
    public Text evaluate(Text input) {
        if (input == null) {
            return null;
        }
        return new Text(input.toString().toUpperCase());
```

```
    }
}
```

- **Explanation**:
    - **UDF Class**: The custom UDF class extends the `UDF` class from Hive's library.
    - **Evaluate Method**: This method contains the logic of the function. The method name must be `evaluate`, and it can be overloaded to accept different numbers or types of arguments.
    - **Return Type**: The return type should be compatible with Hive data types (e.g., `Text`, `IntWritable`, `DoubleWritable`).
- **Compiling the UDF**:
    - Compile the Java code into a JAR file using a build tool like Maven or directly using `javac`.

Example:
```
javac -classpath $(hive --config) MyUpperCase.java
jar cf my_udfs.jar MyUpperCase.class
```

    -
- **Registering the UDF in Hive**:

Add the JAR File: Before using the UDF, you need to add the JAR file to Hive.
```
ADD JAR /path/to/my_udfs.jar;
```
Create the Function:
```
CREATE FUNCTION my_uppercase AS 'com.example.MyUpperCase';
```

4. Using UDFs in Hive Queries

Example Query:
```
SELECT my_uppercase(customer_name)
FROM customers;
```

- **Explanation**: The custom `my_uppercase` UDF is applied to the `customer_name` column, converting all names to uppercase.
- **Chaining UDFs**:
    - UDFs can be used in combination with built-in functions or other UDFs for more complex operations.

Example:
```
SELECT CONCAT(my_uppercase(first_name), ' ', my_uppercase(last_name))
FROM customers;
```

---

5. Best Practices for Writing UDFs

- **Keep It Simple**: UDFs should be designed to perform a single, well-defined task. Complex logic should be broken down into multiple UDFs if necessary.
- **Efficient Processing**: Since UDFs are executed for each row in a query, ensure that the code is optimized for performance. Avoid unnecessary computations or memory allocations.
- **Handle Nulls**: Always handle `NULL` inputs gracefully in your UDFs to prevent runtime errors.
- **Testing**: Thoroughly test your UDFs with different inputs to ensure they behave as expected.

---

6. Example: A Complex UDF

Scenario: You need a UDF to mask sensitive information, such as email addresses, by replacing part of the email with asterisks.

Java UDF Example:

```
public class MaskEmail extends UDF {
    public Text evaluate(Text input) {
        if (input == null || !input.toString().contains("@")) {
            return input;
        }
        String[] parts = input.toString().split("@");
        String maskedEmail = parts[0].replaceAll("(?<=.{2}).", "*") +
"@" + parts[1];
        return new Text(maskedEmail);
    }
}
```

Usage in Hive:

```
ADD JAR /path/to/masked_email_udf.jar;
CREATE FUNCTION mask_email AS 'com.example.MaskEmail';

SELECT mask_email(email_address)
FROM users;
```

- **Output**: If the email address is `john.doe@example.com`, the UDF would transform it to `jo******@example.com`.

---

7. Performance Considerations

- **Execution Overhead**: UDFs can introduce overhead, especially if they are complex or if they process large volumes of data. It's important to profile and optimize UDFs to minimize their impact on query performance.
- **Scalability**: Ensure that UDFs can handle large datasets efficiently, especially when used in distributed environments like Hadoop.