05: Aggregations

CONFLUENT
Global Education

Module Overview

This module contains six lessons:

- Overview
- GROUP BY vs OVER
- Aggregate Functions
- Special Aggregations Queries
- Additional Options
- Important Considerations

05a: Overview

Description

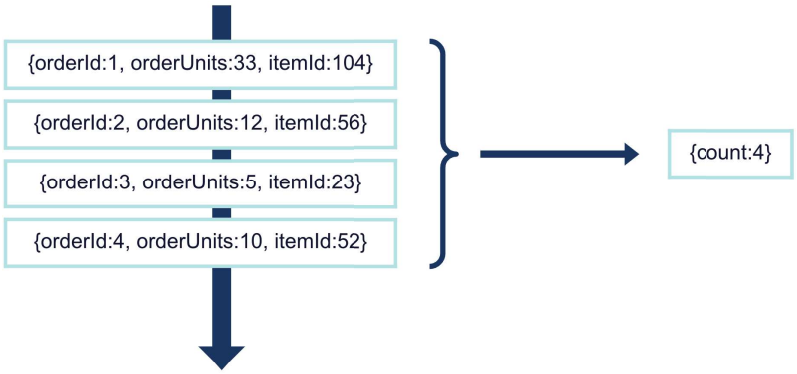
Overview of Aggregations.

Aggregation Overview

Computing a single result from multiple input rows

You need to define:

- Set of rows to apply the aggregate function (GROUP BY, OVER)
- Aggregate Function (SUM, COUNT, MAX, ...)



05b: GROUP BY vs OVER

Description

Defining GROUP BY and OVER clauses. Comparing both approaches.

Defining the Set of Rows

Must define the set of rows to apply the aggregation:

- `GROUP BY`
- `OVER`

```
SELECT
  itemid,
  COUNT(*) AS orders_count,
  SUM(orderunits) AS
units_accum_per_item
 FROM orders
 GROUP BY itemid;
```

```
SELECT
  ordertime,
  orderid,
  itemid,
  orderunits,
  SUM(orderunits) OVER ( PARTITION BY itemid
                         ORDER BY $rowtime
                       ) AS
units_accum_per_item
 FROM orders;
```

GROUP BY: Overview

- Group rows that have the same value/s for the selected column/s
- Reduce all input rows to a **single output row** for every group
- Partition the dataset based on **one or more columns**

Syntax:

```
SELECT
    column_1,
    AGGR_FUNC_1(),
    AGGR_FUNC_2()
FROM
    table
GROUP BY
    column_1;
```

`GROUP BY` enables you to categorize data into logical groups and apply aggregate functions to each group independently. This is particularly useful in various industry applications where insights need to be derived from continuously incoming data streams.

Two use cases for `GROUP BY`:

- In the financial sector, especially within stock trading platforms or financial news aggregators, calculating real-time metrics such as average transaction value, total volume of trades per stock symbol, or count of transactions in a given time window is crucial.
- E-commerce platforms often need to track user activities in real-time to analyze behaviors, such as the number of products viewed, added to cart, and purchased by users over specific time intervals. This data helps in personalizing user experience and optimizing inventory management.

Activity: GROUP BY - Fill The Gaps (I)

Scenario 1:
You have a continuous stream of stock market data (symbol, price, volume and timestamp). You want to analyze this streaming data to calculate the **total traded volume** and the **average price** for each stock symbol, updated in real-time.

```
SELECT
    symbol,
    __________ AS total_volume,
    AVG(price) AS avg_price
FROM
    stock_data
____________________;
```

Scenario 1 - Solution:

```
SELECT
    symbol,
    SUM(volume) AS total_volume,
    AVG(price) AS avg_price
FROM
    stock_data
GROUP BY
    symbol;
```


Activity: GROUP BY - Fill The Gaps (II)

Scenario 2:
You have a continuous stream of sales data from all your stores across the country (store_id, sale_amount, sale_time). You want to calculate the total number of sales and total revenue generated per store each hour.

```
SELECT
    store_id,
    _________________ AS total_sales,
    SUM(sale_amount) AS total_revenue,
    window_start,
    window_end
FROM
    TABLE(TUMBLE(TABLE sales, DESCRIPTOR(sale_time),
INTERVAL '1' HOUR))
GROUP BY
_____________________________;
```

Scenario 2 - Solution:


```
SELECT
    store_id,
    COUNT(*) AS total_sales,
    SUM(sale_amount) AS total_revenue,
    window_start,
    window_end
FROM
    TABLE(TUMBLE(TABLE sales, DESCRIPTOR(sale_time), INTERVAL '1' HOUR))
GROUP BY
    store_id,
    window_start,
    window_end;
```

GROUP BY: Question

What is wrong with this query?

Syntax:

```
SELECT
    symbol,
    price,
    volume,
    SUM(volume) AS total_volume,
    AVG(price) AS avg_price
FROM
    stock_data
GROUP BY
    symbol;
```

The issue in this query is the inclusion of the `price` and `volume` columns in the `SELECT` list, which are not part of an aggregate function nor included in the `GROUP BY` clause. SQL standards require that any column in the `SELECT` clause that isn't an aggregated value must be included in the `GROUP BY` clause. This ensures that the query results are logically consistent, as each row generated by the query has a clear grouping basis.

Here, the `price` and `volume` fields can have multiple values per `symbol`, so the SQL engine doesn't inherently know which specific `price` or `volume` value to return for each `symbol`.

OVER: Overview

- Produce an aggregated value **for every input row**
- Aggregate functions operate on a **WINDOW** of rows. Defined by:
  - Partition By
  - Order By
  - Range Definitions
- Partition the dataset based on **one or more columns**

Syntax:

```
SELECT
    column_1,
    column_2,
    AGGR_FUNC() OVER <window> AS
column_name
FROM
    table;
```

Compute an aggregated value for every row over a range of ordered rows.

OVER clause computes an aggregated value for every input row over a range of ordered rows. In contrast to a GROUP BY aggregate, an OVER aggregate doesn't reduce the number of result rows to a single row for every group. Instead, OVER produces an aggregated value for every input row.

You can define multiple OVER window aggregates in a SELECT clause. However, for streaming queries, the OVER windows for all aggregates must be identical due to current limitation. Meaning that you can use several aggregated functions in a single select, but they all must use the same OVER window definition.

OVER: Simple Example

Using the OVER clause to calculate a running total:

```
SELECT
  SalesRepID,
  Amount,
  SUM(Amount) OVER (PARTITION BY SalesRepID ORDER BY $rowtime) AS RunningTotal
FROM
  Sales;
```

Result:

SalesRepID	Amount	RunningTotal
101	10	10
101	30	40
101	5	45
102	20	20
102	2	22

OVER: Use Case (I)

Calculating the daily cumulative sales amount for each product

```
SELECT
    order_id,
    product_id,
    sales_amount,
    order_date,
    SUM(sales_amount) OVER ( PARTITION BY order_date,
product_id
                             ORDER BY $rowtime
                           ) AS daily_product_sales
FROM
    sales_table;
```

In this example:

- We're calculating the daily sales for each product, partitioned by both `order_date` and `product_id`.
- The `OVER` clause partitions the data based on both `order_date` and `product_id`, forming distinct groups for each combination of date and product.
- Within each partition, the rows are ordered by `$rowtime`.
- The `SUM(sales_amount)` function then computes the cumulative sum of `sales_amount` within each partition, resulting in the daily sales for each product.
- The `daily_product_sales` column contains the cumulative sales for each product on each day.

This query is especially useful in a real-time dashboard or a monitoring system where businesses need to track the sales performance of each product every day as data streams in. It allows businesses to see how sales accumulate throughout the day for each product and can help in making quick decisions related to inventory management, pricing strategies, or promotional offers based on real-time sales trends.

OVER: Use Case (II)

Calculating Moving Average of Stock Prices

```
SELECT
    stock_symbol,
    price,
    volume,
    AVG(price) OVER ( PARTITION BY stock_symbol
                      ORDER BY $rowtime
                      RANGE BETWEEN INTERVAL '5' MINUTE PRECEDING AND
CURRENT ROW
                    ) AS moving_avg_price
FROM stock_prices;
```

In this example, calculating the moving average price of stocks every time a new price is received using a 5-minute sliding window:

- OVER Clause with Time Interval: The query calculates the moving average of the stock price over a 5-minute window.
- The PARTITION BY clause ensures that the aggregation is performed separately for each stock symbol.
- The ORDER BY $rowtime specifies that the window is based on the ordering of the $rowtime.
- The RANGE BETWEEN INTERVAL '5' MINUTE PRECEDING AND CURRENT ROW specifies the size of the sliding window.

In real trading platforms, such moving averages are crucial for triggering buy or sell signals. Traders often use these indicators to identify potential reversals, momentum, and entry or exit points based on historical prices. The real-time aspect of Flink allows this information to be updated and available instantly with each new piece of data, enabling rapid decision-making that is essential in high-frequency trading environments.

OVER: Use Case (III)

Real-Time Monitoring of Production Line Errors in Manufacturing

```
SELECT
     sensor_id,
     error_type,
     severity,
     MAX(severity) OVER w AS max_severity_last_10_events,
     AVG(severity) OVER w AS avg_severity_last_10_events
FROM production_errors
WINDOW w AS ( PARTITION BY sensor_id
              ORDER BY $rowtime
              ROWS BETWEEN 9 PRECEDING AND CURRENT ROW );
```

`OVER` windows for all aggregates in the statement must be identical

---

In this example, calculating a moving average and maximum severity of errors reported by each sensor to detect abnormal spikes that exceed typical variations in error rates:

- `WINDOW` clause with `ROWS`: The window is specified with the last 10 events (`ROWS BETWEEN 9 PRECEDING AND CURRENT ROW`). This fixed-size window allows the system to calculate metrics based on the most recent 10 events regardless of the actual time they occurred.
- Calculations: The query calculates the maximum severity (`max_severity_last_10_events`) and the average severity (`avg_severity_last_10_events`) for the last 10 error events for each sensor. These metrics help in quickly identifying unusual patterns, such as a sudden increase in error severity.

This real-time data processing setup can help the manufacturing company in several ways:

- Immediate Error Detection: Quick identification of spikes in error rates enables the maintenance team to react promptly, possibly even automating some responses such as shutting down equipment for safety checks.
- Quality Control: Maintaining a low error rate is crucial for product quality. Early detection of anomalies allows for immediate corrective actions, reducing the risk of defective products.
- Maintenance Scheduling: By analyzing trends over time, the system can also suggest optimal times for regular maintenance, avoiding unnecessary shutdowns or delays.

Alternatively, we could change the query from a `ROW` range to a `TIME INTERVAL` range to analyze/detect errors from the last hour and count how many errors we have.

GROUP BY vs OVER

	GROUP BY	OVER
Purpose	Compute a summary for each group	Perform calculations over a window for every input row
Performance	It is highly optimized and maintains smaller state	It is more CPU-intensive (more complex queries) and usually maintains larger state
Use Cases	Real-Time Analytics & Dashboards Event Pattern Detection Alerting Based on Thresholds	Running Totals & Moving Averages Cumulative Summaries Sequential Data Access using `LAG()`

`LAG()` function will be covered in the next lesson.

Lesson 05c

05c: Aggregate Functions

Description

Reviewing the different Aggregate Functions available in Flink SQL.

Aggregate Functions: Overview (I)

Compute a single result from multiple input rows

FUNCTION	DEFINITION
COUNT	Counts the number of rows in a group
SUM	Calculates the sum of values in a group
AVG	Computes the average of values in a group
MAX	Finds the maximum value in a group
MIN	Finds the minimum value in a group
FIRST_VALUE	Returns the first value in a group
LAST_VALUE	Returns the last value in a group
STDDEV_POP	Calculates the population standard deviation
STDDEV_SAMP	Calculates the sample standard deviation

Just for clarification:

Population: The entire group from which data can be collected, representing all individuals or items of interest.

Sample: A subset of the population selected for analysis, used to estimate the characteristics of the entire population.

In standard deviation terms, the population standard deviation uses `N` (the total number of individuals), while the sample standard deviation uses `n-1` (the sample size minus one) to account for bias.

Aggregate Functions: Overview (II)

FUNCTION	DEFINITION
VAR_POP	Computes the population variance
VAR_SAMP or VARIANCE	Computes the sample variance
COLLECT	Aggregates values into a multiset
LAG	Accesses the value of a previous row in the result set
LISTAGG	Concatenates values into a single string
ROW_NUMBER	Assigns a unique sequential integer to each row

Lesson 05d

05d: Special Aggregation Queries

Description

Understanding how to use Top-N and Deduplication.

Special Aggregation Queries: Overview

Two special types of queries using `ROW_NUMBER()` function:

- Top-N queries
- Deduplication queries

Top-N: Overview

Find the N-smallest or N-largest values, ordered by column/s in a table

Syntax:

```
SELECT [column_list]
FROM (
  SELECT [column_list],
         ROW_NUMBER() OVER ([PARTITION BY column1[, column2...]]
                            ORDER BY column1 [asc|desc][, column2
[asc|desc]...]
                           ) AS rownum
  FROM table_name)
WHERE rownum <= N [AND conditions]
```

For Top-N queries, the `ORDER BY` in the OVER window can be defined on any column (not only the time attribute)

`ASC` to get the N smallest values. `DESC` to get the N largest values

---

Top-N queries return the N smallest or largest values in a table, ordered by columns. Both smallest and largest values sets are considered Top-N queries. Top-N queries are useful in cases where the need is to display only the N bottom-most or the N top-most records from batch/streaming table on a condition. This result set can be used for further analysis.

Flink uses the combination of a OVER window clause and a filter condition to express a Top-N query. With the power of OVER window `PARTITION BY` clause, Flink also supports per group Top-N.

The Top-N query is Result Updating, which means that Flink sorts the input stream according to the order key. If the top N rows have changed, the changed rows are sent downstream as retraction/update records.

Syntax explanation:

- `ROW_NUMBER()`: Assigns an unique, sequential number to each row, starting with one, according to the ordering of rows within the partition.
- `PARTITION BY column1[, column2…]`: Specifies the partition columns. Each partition has a Top-N result.

05e: Additional Aggregation Option

Description

Describe different aggregation options, such as:

- DISTINCT
- HAVING
- GROUPING SETS

Additional Aggregation Options: Overview

You can implement additional functions to your aggregation queries:

- DISTINCT
- HAVING
- GROUPING SETS
  - ROLLUP
  - CUBE

DISTINCT: Overview

Removes duplicate values before applying an aggregation function

Example:

```
SELECT COUNT(DISTINCT order_id) FROM
orders
```

It counts the number of distinct `order_ids` instead of the total number of rows in an orders table

---

DISTINCT: Real Use Case

Unique Web Visitor Count per Day

```
SELECT
    CAST(window_start AS DATE) AS
day,
    COUNT(DISTINCT user_id) AS
unique_visitors
FROM
    TABLE(
        CUMULATE(
            TABLE web_traffic,
            DESCRIPTOR(event_time),
            INTERVAL '1' MINUTE,
            INTERVAL '1' DAY)
        )
GROUP BY
    window_start;
```

Input

user_id	event_time
...	...
user39	2024-06-25 17:40:51
user4	2024-06-25 17:42:05
user63	2024-06-25 17:42:41
user78	2024-06-25 17:44:26
...	...

Result

day	unique_visitors
2024-06-23	96
2024-06-24	73
2024-06-25	4

Analyze web traffic data and derive insights into the number of unique visitors per day over time.

HAVING: Overview

Eliminates group rows that don't satisfy the specified condition

Example:

```
SELECT users, SUM(amount)
FROM orders
GROUP BY users
HAVING SUM(amount) > 50
```

Filtering: HAVING vs. WHERE

WHERE	HAVING
acts on fields	acts on results of aggregation functions

GROUPING SETS: Overview

Allows you to specify multiple `GROUP BY` sets within a single query

Syntax:

```
GROUP BY GROUPING SETS (
    (col1, col2),
    (col1),
    (col3),
    ()
)
```

Grouping sets enable more complex grouping operations than those you can describe with a standard `GROUP BY` clause. Rows are grouped separately by each specified grouping set, and aggregates are computed for each group just as for simple `GROUP BY` clauses.

Each sublist of `GROUPING SETS` specifies zero or more columns or expressions and is interpreted as if it were used directly in the `GROUP BY` clause. An empty grouping set means that all rows are aggregated down to a single group, which is output even if no input rows were present.

References to the grouping columns or expressions are replaced by null values in result rows for grouping sets in which those columns don't appear.

GROUPING SETS: Real Use Case

Sales Report for Multiple Dimensions

```
SELECT
    store_id,
    prod_id,
    COUNT(*) AS num_trans,
    SUM(sale_price) AS total_sales,
    AVG(sale_price) AS avg_price
FROM
    sales_data
GROUP BY GROUPING SETS (
    (store_id, prod_id),
    (store_id),
    (prod_id),
    ()
);
```

Result

store_id	prod_id	num_trans	total_sales	avg_price
1	101	10	1000.00	100.00
1	102	8	720.00	90.00
2	101	5	550.00	110.00
1	NULL	18	1720.00	95.56
2	NULL	5	550.00	110.00
NULL	101	15	1550.00	103.33
NULL	102	8	720.00	90.00
NULL	NULL	23	2270.00	98.70

Create a sales report that provides total sales, count of sales transactions, and average sales price for products by different dimensions: by product, by store, and overall.

Lesson 05f

05f: Aggregations: Important Considerations

Description

Understand Window Aggregation syntax, aggregation state nad cascading time attributes.

Window Aggregation: Example (I)

Unique Web Visitor Count per Day

Option 1

```
SELECT
    CAST(window_start AS DATE) AS `day`,
    COUNT(DISTINCT user_id) AS
unique_visits
FROM
    TABLE(
        CUMULATE(
            TABLE web_traffic,
            DESCRIPTOR(event_time),
            INTERVAL '1' MINUTE,
            INTERVAL '1' DAY)
        )
GROUP BY
    window_start;
```

Option 2

```
SELECT
    CAST(window_start AS DATE) AS `day`,
    COUNT(DISTINCT user_id) AS
unique_visits
FROM
    TABLE(
        CUMULATE(
            TABLE web_traffic,
            DESCRIPTOR(event_time),
            INTERVAL '1' MINUTE,
            INTERVAL '1' DAY)
        )
GROUP BY
    window_start,
    window_end;
```

Will there be any difference in the result?

The table `web_traffic` was created with this query:


```
CREATE TABLE web_traffic (
    user_id STRING,
    event_time TIMESTAMP(3),
    WATERMARK FOR event_time AS event_time - INTERVAL '5' SECONDS
);
```

Lab Module 05: Using Aggregations in a Practical Use Case

Please work on **Lab Module 05: Using Aggregations in a Practical Use Case**.

Refer to the Exercise Guide.