PUBLIC
2025-02-04

SAP Signavio Analytics Language Guide

THE BEST RUN SAP

Content

1 SAP Signavio Analytics Language

Query language of SAP Signavio Process Intelligence optimized for performing process mining tasks on large amounts of event data

SAP Signavio Analytics Language, SIGNAL is a specialized query language for process analysis.

The language is based on SQL. Like SQL, you use queries to retrieve data and perform calculations on the data. However, it is not possible to change or delete process data.

The difference to SQL is the data model. While you usually query data from multiple tables with SQL, SIGNAL queries the data from only one table, which contains nested events. In addition, SIGNAL provides numerous custom functions to work more effectively with this data structure.

SIGNAL is optimized for process mining, for example to determine conformance, cycle times, and rework, and it supports exploration at scale by all kind of SAP Signavio Process Intelligence users.

With SIGNAL, you can only retrieve data from processes to which you have access.

Data model

When mining the data of a process, you retrieve the data of a single table. This table contains the case attributes and their nested events and event attributes. The following table shows this nested structure.

case_ID	Customer ID	Status	City	events			
				event_name	**end_time**	**Payment method**	**Cancellation reason**
1001	2001	delivered	Berlin	Receive customer order	2020-07-01T 09:00:00		
				Receive payment	2020-07-02T 10:00:00	Bank transfer	
				Ship goods	2020-07-03T 11:00:00		
				event_name	**end_time**	**Payment method**	**Cancellation reason**
1002	2002	canceled		Receive customer order	2020-07-04T 13:00:00		
				Cancel order	2020-07-04T 14:00:00		Wrong size

The columns `case_id`, `event_name` and `end_time` are always present. Case attributes like `Customer ID` , `Status`, and `City` have the same value throughout the case. Additional event attributes, in this example `Payment method` and `Cancellation reason` can have different values for each event.

There are two ways to iterate over this data:

- per case
  Each case is treated as one row. The nested events and event attributes are represented as a nested table.
- per event
  Each event is treated as one row. The case ID and case attributes are repeated for each event.

Data types

The data type of a column defines which value the column can hold. All data types can occur on case level as well as on event level (nested).

SIGNAL supports the following data types:

- Strings
- Numbers stored as double precision floating point
- Timestamps stored with millisecond precision, without time zone information.
- Durations stored with millisecond precision
- Booleans

All of these data types can appear in the source file and in the query result.

Both case and event attributes can be Null, indicating the absence of a value or an unknown value.

Process mining

SIGNAL queries are used in the widgets of an investigation.

When configuring widgets, you have the following options:

- Create you own queries
- Use the default queries in the widgets and customize them if necessary
- Use the predefined queries from the metrics library and customize them if necessary
- Add your own queries to the metrics library for reuse

Read more on the widgets in section Widgets.

1.1 Syntax

SIGNAL follows a specific syntax that is described throughout this documentation using a special notation.

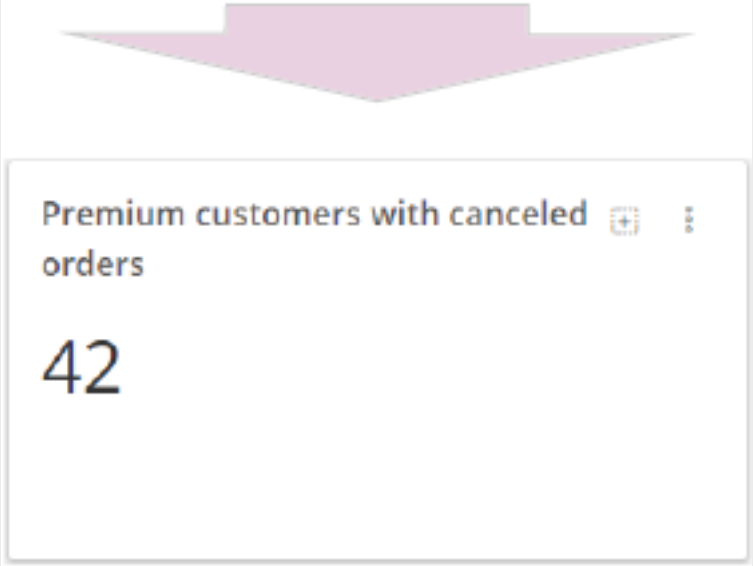
The syntax of SIGNAL is based on SQL but enhanced with functions to run in-depth process analysis queries. All queries always follow this basic structure:

```
SELECT expressions
FROM table/process
WHERE conditions
```

Example

Example

```
SELECT count(case_id)
FROM THIS_PROCESS
WHERE "Customer Type" = 'Premium'
and  "Order Status" = 'Canceled'
```



This query counts the cases in the declared table for which the condition is true.

Syntax Notation

The following notation is used to describe the syntax of SIGNAL. Note that this notation isn't part of the actual query:

- Angle brackets indicate a required element. Don't include the angle brackets as part of a query.

Example: `ORDER BY <expression>`

- Square brackets indicate optional elements. Don't include the square brackets as part of a query.
  Example: `[ WHERE <condition> ]`
- The pipe symbol indicates a choice between two or more options. They're listed within square brackets (meaning inclusion of a choice is optional) or curly brackets (meaning one of the options must be chosen). Don't include the pipe or brackets as part of a query.
  Example: `[ ASC | DESC ]`
  Example: `{ AND | OR }`
- An ellipsis indicates that the preceding element can be repeated an arbitrary number of times.
  Example: `GROUP BY columnIndex [, columnIndex ...]`

Keywords

SIGNAL keywords are case-insensitive, but are by convention written upper-case to distinguish them from expressions. Read more on keywords in SIGNAL keywords [page 288].

Attribute Names

Attribute names are case-insensitive. In the following cases, the attribute name must be enclosed in double quotes:

- The attribute shares the same name as a reserved SIGNAL keyword.
- The attribute doesn't start with a letter or an underscore.
- The attribute name contains characters other than letters, numbers, or underscores.
- If the attribute name is enclosed in double quotes, the entire expression must be enclosed in double quotes.
  Example: "Premium User" -> ""Premium User""
- If the attribute name contains one or more double quotes, these double quotes must be followed by a double quote.
  Example: Width in " (inches) -> "Width in "" (inches)"
- If the attribute name contains one or more single quotes, these single quotes must be followed by a double quote.
  Example: O'Reilly -> "O'"Reilly"

Attribute Values

Attribute values are, where applicable, case-sensitive. Attribute values of type String, Timestamp, and Duration must always be enclosed in single quotes.

Semantic Attributes

For SIGNAL, the following pre-defined semantic attributes are always present:

- `case_id`: Unique identifier of the case
- `event_name`: Event names of the case ordered by their time
- `end_time`: End timestamps of the events of a case

The syntax rules for semantic attributes are the same as for the other attributes.

1.2 Aliases

Learn about aliases in SIGNAL, the process mining query language of SAP Signavio Process Intelligence.

Aliases are used to give the result set a temporary name to make the column headings in your result set easier to read. It's common to alias a column when using an aggregate function in a query. Without an alias, a name will be generated based on the column and operations in the expression. Read more in section SIGNAL functions [page 91].

An alias is only valid within the scope of the SIGNAL statement.

Syntax:

```
expression AS alias_name
```

Parameter	Description	Required
`expression`	The expression that you want to give a better name.	Required
`alias_name`	The temporary name to assign. Enclose the name in double quotes if it contains a character that isn't a letter or digit.	Required

Example:

In this example, we have the following data in a table:

case_id	event_name	end_time
1001	Receive customer order	2020-07-01T09:00:00
1002	Receive customer order	2020-07-04T13:00:00
1003	Receive customer order	2020-07-05T11:00:00

Enter the following SIGNAL statement:

```
SELECT COUNT(case_id) AS "No. of cases"
FROM THIS_PROCESS
```

This query returns the following result set:

No. of cases
3

1.3 SELECT Statement and Clauses

Use the SELECT statement to select data from a process and return it in a result set.

Syntax

```
SELECT { <columnExpression> [FILTER (WHERE <condition>)] [, ...] | * }
FROM <tableExpression>
[TABLESAMPLE EXACT( { <probability> PERCENT | <num> ROWS } ) [ REPEATABLE(
<seed> ) ] ]
[FILTER EVENTS WHERE <booleanExpression>]
[WHERE <whereExpression>]
[UNION ALL <selectStatement>]
[GROUP BY <columnIndex> [, <columnIndex> ...] ]
[ORDER BY <columnIndex> [ { ASC | DESC } ] [ NULLS { FIRST | LAST } ] [, ... ] ]
[FILL <fillSpecification> [, <fillSpecification>, ...] ]
[LIMIT <count>]
[OFFSET <start>]
```

Clause	Description
SELECT	Specifies columns or values to be included in a result set.
FROM	Specifies the table in your process from which you want to retrieve the data. You can reference the process by using the explicit Process ID, which can be found on the API tab in the process settings page. Alternatively you can use the alias THIS_PROCESS to refer to the default view.
TABLESAMPLE	Specifies the absolute or percentage table fraction to be considered for the query.
FILTER EVENTS	Filters rows at event-level.
WHERE	Specifies the condition that must be met for cases to be selected. If this clause isn't provided, then all records are selected.
UNION ALL	Concatenates the result sets of two or more SELECT statements.
GROUP BY	Collects data across multiple records and groups the results by one or more columns. The GROUP BY clause requires an index similar to the ORDER BY clause. You can use one or multiple indices.

Clause	Description
`ORDER BY`	Sorts the records in the result set. If more than one index is provided, separate them with a comma. `ASC` sorts the result set in ascending order by `expression`, `DESC` sorts it in descending order. `NULLS FIRST` sorts the result set with null values first, `NULLS LAST` with null values last.
`FILL`	Fills any gaps inside a time series column by inserting new rows into the result set, each containing missing timestamps.
`LIMIT`	Specifies the number of records in the result set.
`OFFSET`	Specifies the starting point to return rows from a result set.

Column Expressions

The `SELECT` statement is used to structure the rows of your result set.

Syntax:

```
SELECT <columnExpression> [, <columnExpression> ...]
FROM <tableExpression>
```

Each column expression specifies how to populate the values in that particular column.

For example, you can select a column from the table referred to by `tableExpression`. Doing so populates your result set with the values from the specified column in the table.

Example

Let's assume we have the follow data (`THIS_PROCESS`):

case_id	Customer ID	Status	Order Quantity
1	C_1023	Received	1
2	C_1198	Received	3
3	C_1212	Dispatched	2

```
SELECT "Customer ID"
FROM THIS_PROCESS
```

This query returns a result set with one column containing all the values from the process data's customer ID column:

Customer ID
C_1023
C_1198

Customer ID
C_1212

A column expression can also be a literal value. Such a value is repeated in that column for each row in the result set.

Example

```
SELECT case_id, 1
FROM THIS_PROCESS
```

This query creates a result set with two columns. The first column is populated with the values from `case_id` in `THIS_PROCESS`. The second column is populated with the literal value 1.

case_id	1
1	1
2	1
3	1

You can also create column expressions by combining values, column names, operators or function calls.

Example

```
SELECT "Customer ID", "Order Quantity" + 1
FROM THIS_PROCESS
```

In this query, the result set's second column is populated with the values from the `Order Quantity` column in `THIS_PROCESS`, but in each case that value is increased by 1.

Customer ID	"Order Quantity" + 1
C_1023	2
C_1198	4
C_1212	3

Selecting All Columns From a Process

You can use the * operator in a `SELECT` statement to select all columns of a table under the following conditions:

- The query contains multiple `SELECT` statements.
- At least one of those `SELECT` statements references an explicit list of columns.

Example

```
SELECT case_id,
    "Customer ID",
```

```
    "Order Quantity"
FROM (
    SELECT *
    FROM THIS_PROCESS
) AS sub
```

This query uses a subquery to select all columns from `THIS_PROCESS`. From them, the outer query then selects an explicit subset of columns:

case_id	Customer ID	Order Quantity
1	C_1023	1
2	C_1198	3
3	C_1212	2

Including Clauses

When building a SIGNAL query, add any of the optional clauses in the order they appear in the syntax.

Example

```
SELECT case_id, "Customer ID", "Order Quantity"
FROM table
ORDER BY 3 DESC
LIMIT 2
```

This query returns the case ID, customer ID and order quantity of the first 2 cases in the table. The clause `ORDER BY 3 DESC` orders the result set by the third column in descending order.

case_id	Customer ID	Order Quantity
2	C_1198	3
3	C_1212	2

Related Information

Subqueries [page 49]

1.3.1 SELECT DISTINCT Clause

Modifies the `SELECT` statement so that it returns only unique values of a specified column.

If `NULL` values are present in a column, then `NULL` appears in the result set.

> **Note**
>
> This function isn't supported in subqueries.

Syntax

```
SELECT DISTINCT <expression>
```

Parameter	Description
`expression`	The column, expression, or event-attribute that is returned. If more than one expression is provided, separate the values with a comma.
`table`	The process table or view from which you want to retrieve data. It's referenced by explicit Process ID or the alias `THIS_PROCESS`.

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	city	event_name
00007	San Francisco	Receive Customer Order
		Receive Payment
		Send items to Printing
		Items Printed
		Receive items from Printing
		Ship Goods Standard
00008	Houston	Receive Customer Order
		Change Order Quantity
		Receive Payment
		Ship Goods Express
		Receive Delivery Confirmation

case_id	city	event_name
00009	Miami	Receive Customer Order
		Receive Payment
		Receive Payment
		Ship Goods Express
		Receive Delivery Confirmation
00010	Houston	Receive Customer Order
		Change Order Quantity
		Receive Payment
		Ship Goods Express
		Receive Delivery Confirmation

The following query selects all cities in the process data but excludes duplicate values, meaning every city name appears only once. The resulting cities are then sorted alphabetically.

```
SELECT DISTINCT city
FROM THIS_PROCESS
ORDER BY 1 ASC
```

Output:

city
Houston
Miami
San Francisco

Example 2

Assuming the same process data as the previous example, the goal is to find all unique event name sequences (variants) in your process.

```
SELECT DISTINCT event_name
FROM THIS_PROCESS
```

The query is executed at the case level, meaning that each event name sequence is treated as a list of strings. Two sequences are considered equal if they share the same event names in the same order. Duplicate values are removed from the result set, so every variant appears only once.

Running the query produces the following output (notice how the events of cases 00008 and 00010, being equal, appear only once):

event_name
Receive Customer Order
Receive Payment
Receive Payment
Ship Goods Express
Receive Delivery Confirmation
Receive Customer Order
Change Order Quantity
Receive Payment
Ship Goods Express
Receive Delivery Confirmation
Receive Customer Order
Receive Payment
Send items to Printing
Items Printed
Receive items from Printing
Ship Goods Standard
Receive Delivery Confirmation

Example 3

In this example, the goal is to retrieve only unique event names in your process. Unlike the previous example, we need to consider individual event names rather than event name sequences. To do this, we FLATTEN the process data, representing each nested event as a single row containing case and event attributes.

```
SELECT DISTINCT event_name
FROM FLATTEN (THIS_PROCESS)
ORDER BY 1 ASC
```

Duplicate values are then removed from the result set as a whole, meaning every event name appears only once.

Output:

event_name
Receive Customer Order
Receive Payment
Items Printed
Receive Delivery Confirmation
Ship Goods Express
Change Order Quantity
Ship Goods Standard
Receive items from Printing
Send items to Printing

Related Information

FLATTEN Operator [page 44]

1.3.2 FROM Clause

Specifies the table in your process from which you want to retrieve data.

`THIS_PROCESS` is the alias assigned to the table set as the process view inside the process analysis, such as in a dashboard. The process view selected therefore serves as the underlying data source for all references to `THIS_PROCESS` inside a query.

Syntax

```
FROM <table>
```

Parameter	Description
`table`	The process table or view from which you want to retrieve records. You can reference the process by using the explicit Process ID, which can be found on the API tab in the process settings page. Alternatively, you can use the alias `THIS_PROCESS` to refer to the default view.

Example

Assume the following process data:

case_id	City	Payment Type
00001	Houston	Bank Transfer
00002	San Francisco	Credit Card
00005	Houston	Credit Card
00009	Miami	Credit Card
00012	San Francisco	Credit Card
00017	New York	Bank Transfer
00018	Washington	Credit Card
00019	San Francisco	Credit Card
00022	San Francisco	Credit Card
00026	Boston	Bank Transfer

The following query retrieves a list of the unique cities in your process data. By referring to the source table as `THIS_PROCESS`, it fetches data from the process view selected in the process analysis.

```
SELECT DISTINCT city
FROM THIS_PROCESS
```

Output:

City
Boston
New York
San Francisco
Houston
Washington
Miami

Related Information

Changing the Process View
Creating Process Views

1.3.3 WHERE Clause

Filters the data by applying conditions to the `SELECT` statement. Only rows matching the condition are included in the result set.

Syntax

```
WHERE <condition>
```

Parameter	Description
`condition`	The Boolean expression that a row must match is included in the result set.

The following is a non-exhaustive list of common operators used in a `WHERE` condition:

Symbols
> (greater than)
< (less than)
>= (greater than or equal to)
<= (less than or equal to)
= (equal to)
!= (not equal to)

Example 1

Assume the following process data (THIS_PROCESS):

case_id	City	Order Amount	Order Status
00001	Houston	944.42	Delivered
00002	San Francisco	270.04	Canceled
00003	Houston	469.9	Delivered
00004	San Francisco	268.34	Delivered
00005	Houston	327.94	Delivered
00006	San Francisco	599.07	Delivered
00007	San Francisco	521.17	Delivered
00008	Houston	162.58	Delivered

case_id	City	Order Amount	Order Status
00009	Miami	165.44	Delivered
00010	Houston	319.18	Delivered

The following query retrieves the total order amount from purchases made in San Francisco. Only order amounts from rows matching the `WHERE` condition are counted.

```
SELECT SUM("Order Amount")
FROM THIS_PROCESS
WHERE("City" = 'San Francisco')
```

Output:

SUM(Order Amount)
1658.62

Example 2

Assume the same process data as the previous example. The following query finds all rows where both city and order status match the condition.

```
SELECT case_id,
    "City",
    "Order Amount",
    "Order Status"
FROM THIS_PROCESS
WHERE "City" = 'San Francisco' AND "Order Status" = 'Delivered'
```

The query returns the following result.

case_id	City	Order Amount	Order Status
00004	San Francisco	268.34	Delivered
00006	San Francisco	599.07	Delivered
00007	San Francisco	521.17	Delivered

Related Information

FILTER EVENTS Clause [page 22]
SUM [page 129]

1.3.4 FILTER Clause

The FILTER clause is used to filter data inside aggregations. This lets you include or exclude specific cases or events from aggregations in your query.

Syntax

```
FILTER (WHERE <condition>)
```

Parameter	Description
`condition`	A Boolean expression. Only cases or events for which this expression evaluates to true are included in the aggregation.

Example 1

Assuming the following process data (`THIS_PROCESS`):

case_id	City	event_name	end_time
00001	Houston	Receive Customer Order	01/08/2020, 12:32
		Change Order Quantity	02/08/2020, 03:19
		Receive Payment	04/08/2020, 16:47
		Ship Goods Express	05/08/2020, 09:57
		Receive Delivery Confirmation	09/08/2020, 09:19
00002	San Francisco	Receive Customer Order	06/04/2020, 06:38
		Receive Payment	07/04/2020, 22:36
		Send items to Printing	08/04/2020, 17:26
		Order Canceled	09/04/2020, 23:43
00003	Houston	Receive Customer Order	21/02/2020, 09:14
		Change Order Quantity	22/02/2020, 16:12
		Receive Payment	26/02/2020, 06:51
		Ship Goods Standard	28/02/2020, 18:00
		Receive Delivery Confirmation	02/03/2020, 20:15

case_id	City	event_name	end_time
00004	San Francisco	Receive Customer Order	09/10/2020, 19:26
		Change Order Quantity	10/10/2020, 15:16
		Receive Payment	14/10/2020, 04:27
		Ship Goods Express	14/10/2020, 12:40
		Receive Delivery Confirmation	16/10/2020, 02:45
00005	Miami	Receive Customer Order	25/12/2020, 18:21
		Change Order Quantity	27/12/2020, 02:13
		Change Order Quantity	27/12/2020, 19:04
		Receive Payment	29/12/2020, 23:55
		Send items to Printing	30/12/2020, 23:46
		Items Printed	03/01/2021, 06:18
		Receive items from Printing	05/01/2021, 19:07
		Ship Goods Express	06/01/2021, 01:14
		Receive Delivery Confirmation	07/01/2021, 01:24

The following query counts the number of orders to the cities of Houston and Miami:

```
SELECT COUNT(case_id) FILTER (WHERE city in ('Houston', 'Miami')) as "Count"
FROM THIS_PROCESS
```

Result:

Count
3

Example 2

Assuming the same process data as the previous example, the following query finds cases where the receipt of an order was followed directly by the order quantity being changed:

```
SELECT COUNT(case_id) FILTER (WHERE event_name MATCHES ('Receive Customer Order'
-> 'Change Order Quantity')) as "Updated Quantity"
FROM THIS_PROCESS
```

Updated Quantity
4

Example 3

The following query finds the duration between two events: first receipt of an order and completion of payment for that order. It does this by executing an event-level subquery, querying event-level data. In obtaining the timestamp of the final payment, only events named '`Receive Payment`' are included by the filter. Because an order may include multiple payments, `LAST` is applied to the resulting aggregation to obtain the latest timestamp.

Similarly, the timestamp of the first event named '`Receive Customer Order`' is queried. The difference between the two timestamps gives the cycle time from order receipt to payment. (See Cycle Time Between Two Specific Events [page 317] for more information.)

```
SELECT
    case_id,
    (SELECT LAST(end_time) FILTER (WHERE event_name = 'Receive Payment') -
        FIRST(end_time) FILTER (WHERE event_name = 'Receive Customer Order')
    ) AS PaymentTurnaround
FROM THIS_PROCESS
```

Result when applied to the example process data:

case_id	PaymentTurnaround
00001	3d 4h
00002	1d 15h
00003	4d 21h
00004	4d 9h
00005	4d 5h

Related Information

Aggregate Functions [page 92]
Cycle Time Between Two Specific Events [page 317]
Event-Level Subqueries [page 52]

1.3.5 FILTER EVENTS Clause

Filter data from a result set at event-level.

This clause evaluates a given boolean expression on the event attributes and removes from the result set those which evaluate to false. This stands in contrast to the `WHERE` statement, which removes entire cases from a result set.

> **Note**
>
> Because this clause removes events rather than entire cases, it's possible for a result set to include cases with empty event lists. Refer to Example 3 [page 26] for a demonstration of how to remove such cases from a result set.

Syntax

```
FILTER EVENTS WHERE <booleanExpression>
```

Parameter	Description
`booleanExpression`	The condition that an event must match to achieve inclusion in the filtered result set.

> **Restriction**
>
> `MATCHES` and `BEHAVIOR MATCHES` expressions can't be nested. Consequently, these expressions can't be used in the `booleanExpression` of a `FILTER EVENTS` clause.

Clause Evaluation Order

If a query contains both a `FILTER EVENTS` clause and a `WHERE` clause, the `FILTER EVENTS` clause is evaluated before the `WHERE` clause. Having this evaluation order enables the elimination of particular events before cases as a whole are filtered further. Example 4 demonstrates this behavior.

The expressions defined in the `FILTER EVENTS` and `WHERE` clauses are evaluated after the `FROM` clause. Consequently, using event-level filtering in combination with the `FLATTEN` operator isn't possible because flattening transforms all event-level columns to case level.

> **→ Tip**
>
> As an alternative, you can use event-level filtering in a subquery and then flatten the result.

Referencing Case and Event-Level Attributes

The `booleanExpression` can reference both case and event-level attributes. Refer to Example 2 for a demonstration.

Usage of the `FILTER EVENTS` clause requires that an event-level attribute appears in at least one of the following (otherwise, an error is returned):

- The set of columns selected as part of the query or subquery

- A follow-up operator, such as a `WHERE` clause

Example 1

Given the following data (`THIS_PROCESS`):

case_id	Order Amount	Event Name	Payment Amount
00007	521.17	Receive Customer Order	null
		Receive Payment	521.17
		Send items to Printing	null
		Items Printed	null
		Receive items from Printing	null
		Ship Goods Standard	null
		Receive Delivery Confirmation	null
00008	162.58	Receive Customer Order	null
		Change Order Quantity	null
		Receive Payment	162.58
		Ship Goods Express	null
		Receive Delivery Confirmation	null
00009	165.44	Receive Customer Order	null
		Receive Payment	126.44
		Receive Payment	39
		Ship Goods Express	null
		Receive Delivery Confirmation	null
00010	319.18	Receive Customer Order	null
		Change Order Quantity	null
		Receive Payment	319.18
		Ship Goods Express	null
		Receive Delivery Confirmation	null

The following query filters out all events where the payment amount is greater than 300.

```
SELECT
```

```
    case_id,
    "Order Amount",
    event_name,
    "Payment Amount"
FROM "THIS_PROCESS"
FILTER EVENTS WHERE "Payment Amount" > 300
```

Output:

case_id	Order Amount	event_name	Payment Amount
00007	521.17	Receive Payment	521.17
00008	162.58		
00009	165.44		
00010	319.18	Receive Payment	319.18

Notice how cases 00008 and 00009 have empty event lists since they have no events fulfilling the `FILTER EVENTS` condition.

Example 2

Using the same process data as the previous example, the following query demonstrates how both case and event-level attributes can be included in a `FILTER EVENTS` clause.

```
SELECT
    case_id,
    "Order Amount",
    event_name,
    "Payment Amount"
FROM "THIS_PROCESS"
FILTER EVENTS WHERE "Payment Amount" < "Order Amount"
```

As the example data shows, all orders were paid for in one complete payment with the exception of case '00009', which was paid for in two smaller installments. Therefore, this query filters out all events except the payment-related events of case '00009'.

case_id	Order Amount	event_name	Payment Amount
00007	521.17		
00008	162.58		
00009	165.44	Receive Payment	126.44
		Receive Payment	39
00010	319.18		

Example 3

Event-level filtering can leave behind cases with empty event lists. To filter out such cases, combine the `FILTER EVENTS` clause with a `WHERE` clause as in the following query (which adapts the query from Example 2):

```
SELECT
    case_id,
    "Order Amount",
    event_name,
    "Payment Amount"
FROM "THIS_PROCESS"
FILTER EVENTS WHERE "Payment Amount" < "Order Amount"
-- Eliminate empty event lists
WHERE (SELECT COUNT(event_name) > 0)
```

This method takes advantage of every event automatically having an `event_name` attribute. Event lists featuring no `event_name` can therefore be considered as empty.

case_id	Order Amount	event_name	Payment Amount
00009	165.44	Receive Payment	126.44
		Receive Payment	39

Example 4

Again using the process data from Example 1, this query further demonstrates combining the `FILTER EVENTS` and `WHERE` clauses.

```
SELECT
    case_id,
    "Order Amount",
    event_name,
    "Payment Amount"
FROM "THIS_PROCESS"
FILTER EVENTS WHERE event_name IN ('Receive Customer Order', 'Receive Payment',
'Receive Delivery Confirmation')
WHERE event_name MATCHES (^'Receive Customer Order' -> 'Receive Payment' ->
'Receive Delivery Confirmation')
```

The `FILTER EVENTS` clause keeps only those events named '`Receive Customer Order`', '`Receive Payment`', or '`Receive Delivery Confirmation`'. The containing cases are filtered further by the `WHERE` clause, which uses a matching expression. Only cases whose event lists follow an expected pattern are kept, that pattern being:

1. Receiving an order
2. Receiving a single payment
3. Receiving delivery confirmation

All cases follow this pattern apart from case '00009', which, as earlier examples showed, involves multiple payments.

case_id	Order Amount	event_name	Payment Amount
00007	521.17	Receive Customer Order	null
		Receive Payment	521.17
		Receive Delivery Confirmation	null
00008	162.58	Receive Customer Order	null
		Receive Payment	162.58
		Receive Delivery Confirmation	null
00010	319.18	Receive Customer Order	null
		Receive Payment	319.18
		Receive Delivery Confirmation	null

Related Information

FLATTEN Operator [page 44]
Matching Expressions [page 80]
WHERE Clause [page 18]

1.3.6 GROUP BY Clause

Groups rows by one or more columns and computes aggregate functions for each group.

Columns are referred to by a number, which reflects their position in the `SELECT` clause. For example, `GROUP BY 2` groups the result by the second expression in the `SELECT` clause. All rows that share the same values for the grouped expression are condensed into a single row.

Syntax

```
GROUP BY <columnIndex> [, ...]
```

Parameter	Description
`columnIndex`	The position in `SELECT` clause of the column being grouped by. The indexing is 1-based, so the first column has position 1. → Remember In a `GROUP BY` clause, you can't refer to columns by their name.

Grouping and Aggregations

The `GROUP BY` clause is often used with aggregate functions, such as `COUNT`, `MAX`, or `AVG`. The aggregate function is computed across all rows of each group and returns a separate value for each group. To specify the rows to be considered for the aggregation, you can apply the `FILTER` clause to the aggregate function.

The `GROUP BY` clause is optional. If the `GROUP BY` clause isn't present, then the following applies:

- If there are aggregate and non-aggregate expressions in the `SELECT` statement, then the result is automatically grouped by any non-aggregate expressions.
- If there are only aggregate expressions in the `SELECT` statement, then the result is a single group comprising all the selected rows.

If the `GROUP BY` clause is present, then the following applies:

- You must group by all expressions in the `SELECT` statement that aren't encapsulated by an aggregate function. **Exception**: The ungrouped expression is functionally dependent on a grouped expression (see Example 2).
- You can't group by an expression that contains an aggregate function.

Grouping by Events

If a case-level attribute is chosen for the grouping, the result set is grouped according to the values present in that attribute. If an event-level attribute is chosen, the result set is grouped by the list of identical sequences of events. This approach can be used to identify process variants.

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	City	Order Amount	Order Status	Order Quantity
00001	Houston	944.42	Delivered	17

case_id	City	Order Amount	Order Status	Order Quantity
00002	San Francisco	270.04	Canceled	10
00003	Houston	469.9	Delivered	15
00004	San Francisco	268.34	Delivered	17
00005	Houston	327.94	Delivered	18
00006	San Francisco	599.07	Delivered	10
00007	San Francisco	521.17	Delivered	8
00008	Houston	162.58	Delivered	19
00009	Miami	165.44	Delivered	18
00010	Houston	319.18	Delivered	10

```
SELECT
    "City",
    "Order Status",
    COUNT(case_id) AS "No. of Cases"
FROM THIS_PROCESS
GROUP BY 1, 2
```

The `GROUP BY` clause in this query groups by the two expressions in the `SELECT` statement that aren't encapsulated by an aggregation function. Since this `SELECT` statement contains two such expressions ('`City`' and '`Order Status`'), the `GROUP BY` index must refer to both expressions. `GROUP BY 1` or `GROUP BY 2` wouldn't be valid in this case.

The query produces the following output:

City	Order Status	No. of Cases
Miami	Delivered	1
San Francisco	Delivered	3
	Canceled	1
Houston	Delivered	5

Example 2

Assume the same process data as the previous example. The following query groups orders by order size and counts how many of each size there are. It also includes in each group that order quantity incremented.

```
SELECT
    "Order Quantity",
    ("Order Quantity" + 1) AS "Orders + 1",
    COUNT(case_id) AS "No. of Cases"
FROM THIS_PROCESS
GROUP BY 1
```

Since the second expression, `("Order Quantity" + 1)`, is functionally dependent on the first column, `"Order Quantity"`, `GROUP BY 1` is valid in this case. There is no need to include the second column in the grouping.

The query returns the following result.

Order Quantity	Orders + 1	No. of Cases
8	9	1
10	11	3
15	16	1
17	18	2
18	19	2
19	20	1

Related Information

Aggregate Functions [page 92]
FILTER Clause [page 20]

1.3.7 ORDER BY Clause

Sorts the result set in ascending or descending order.

> ⚠ Caution
>
> For very large data sets, this operator may require excessive CPU activity, causing long query execution times.
>
> For more information, refer to Performance [page 294].

Syntax

```
ORDER BY <columnIndex> [ { ASC | DESC } ] [ NULLS { FIRST | LAST } ] [, ... ]
```

Parameter	Description
`columnIndex`	The index of a column used to sort the records in the result set. If more than one expression is provided, separate the values with a comma. The index is 1-based, meaning that the first column is referred to as 1.

Additional keywords can be added to influence the sorting:

- The direction of sorting: `ASC` sorts the column in ascending order. `DESC` sorts it in descending order. If neither keyword is specified, the default sorting is ascending.
- The position of `NULL` values: `NULLS FIRST` sorts a column so that `NULL` values are presented first. `NULLS LAST` sorts a column so that `NULL` values are presented last. If neither is specified, the default position is `NULLS FIRST`.

Example

Assume the following process data (`THIS_PROCESS`):

case_id	City	Order Amount	Order Status	Order Quantity
00001	Houston	944.42	Delivered	17
00002	San Francisco	270.04	Canceled	10
00003	Houston	469.9	Delivered	15
00004	San Francisco	268.34	Delivered	17
00005	Houston	327.94	Delivered	18
00006	San Francisco	599.07	Delivered	10
00007	San Francisco	521.17	Delivered	8
00008	Houston	162.58	Delivered	19
00009	Miami	165.44	Delivered	18
00010	Houston	319.18	Delivered	10

```
SELECT
    case_id,
    "City",
    "Order Amount",
    "Order Status",
    "Order Quantity"
FROM THIS_PROCESS
ORDER BY 3 DESC
```

This query returns cases in the process sorted by the order amount (column number 3) in descending order.

case_id	City	Order Amount	Order Status	Order Quantity
00001	Houston	944.42	Delivered	17
00006	San Francisco	599.07	Delivered	10
00007	San Francisco	521.17	Delivered	8
00003	Houston	469.9	Delivered	15
00005	Houston	327.94	Delivered	18
00010	Houston	319.18	Delivered	10

case_id	City	Order Amount	Order Status	Order Quantity
00002	San Francisco	270.04	Canceled	10
00004	San Francisco	268.34	Delivered	17
00009	Miami	165.44	Delivered	18
00008	Houston	162.58	Delivered	19

1.3.8 FILL Clause

The `FILL` clause fills any gaps inside a time series column. It does so by inserting new rows into the result set, each containing missing timestamps.

As part of the fill, the precision must be specified. Assume, for example, you specify a precision level of '`day`'. If the time series column contains the values 'Mon, 4th September 2023' and 'Thu, 7th September 2023' with no intervening values, then two rows would be inserted containing the missing dates ('Tue, 5th September 2023' and 'Wed, 6th September 2023') in the time series column.

This precision level is part of a larger fill specification you need to provide, which instructs SIGNAL how to populate the columns in all inserted rows.

Syntax

```
SELECT <columnExpression> [, <columnExpression> ...]
FROM <tableExpression>
FILL <fillSpecification> [, <fillSpecification >... ]
```

Parameter	Description
`columnExpression`	A column or value to be selected.
`tableExpression`	The process table or view from which you want to retrieve records, referenced by explicit Process ID or the alias `THIS_PROCESS`.
`fillSpecification`	A fill specification for the corresponding column expression. **Note** The number of fill specifications provided must equal the number of columns selected. Under certain circumstances, missing specifications are added automatically. See Writing Fill Specifications [page 33] for more information.

Writing Fill Specifications

A fill specification takes one of the following values:

- `TIMESERIES(<precision>)`: Defines the precision of the filling, determining with what regularity missing entries are filled in the time series. Exactly one fill specification must be a `TIMESERIES`.
  The precision parameter accepts a string literal specifying the precision level. The following levels are supported:
  - 'year'
  - 'quarter'
  - 'month'
  - 'week' (ISO 8601-week numbering is applied)
  - 'day'
  - 'hour'
- `GROUP`: In the corresponding column, newly added rows are filled by repeating the same value for the whole group. A fill specification can include at most one `GROUP`. Learn more in the following section, Using GROUP [page 34].
- `NULL`: Enters a `NULL` value into the corresponding column for all new rows.

Each fill specification matches with the column at the same index. In other words, the first specification in the `FILL` clause corresponds to the first column in the `SELECT` clause, the second specification corresponds to the second column, and so on.

```
SELECT
    col1,
    col2,
    col3
FROM THIS_PROCESS
FILL
    TIMESERIES('day'), -- corresponds to col1
    GROUP,             -- corresponds to col2
    NULL               -- corresponds to col3
```

Each column must have a fill specification. Any specifications you fail to provide are assumed to be of type `NULL` and implicitly added to the end of the `FILL` clause. Therefore, the following query is equivalent to the previous query:

```
SELECT
    col1,
    col2,
    col3
FROM THIS_PROCESS
FILL
    TIMESERIES('day'),
    GROUP
    -- NULL added here implicitly
```

However, if a `NULL` specification falls between `TIMESERIES` and `GROUP` specifications, then the `NULL` specification must be added explicitly.

```
SELECT
    col1,
    col2,
    col3
FROM THIS_PROCESS
```

```
FILL
    TIMESERIES('day'),
    NULL,
    GROUP
```

Using GROUP

The `FILL` clause isn't only useful for filling missing timestamps in the overall result set. It can also group the values in a specified column and then add rows containing all missing timestamps to each group.

→ Remember

At most one column can be specified as a `GROUP` in a `FILL` clause.

✤ Example

Assume we have the following process data (`THIS_PROCESS`):

City	Customer Type	Received
Boston	Standard	2020-01-01
New York	Premium	2020-01-02
San Francisco	Standard	2020-01-04

If we fill the time series with missing dates without specifying a `GROUP` column, then the missing dates are added independently of the other columns. The newly added rows would therefore be populated with `NULL` values in all other columns.

```
SELECT
    Received,
    City,
    "Customer Type"
FROM THIS_PROCESS
FILL TIMESERIES('day')
```

This query returns the following output:

Received	City	Customer Type
01/01/2020, 00:00	Boston	Standard
02/01/2020, 00:00	New York	Premium
03/01/2020, 00:00	null	null
04/01/2020, 00:00	San Francisco	Standard

By specifying '`City`' as a group, all missing dates are added for each group of cities. Within each group, the value of '`City`' is repeated for each inserted row.

```
SELECT
```

```
    Received,
    City,
    "Customer Type"
FROM THIS_PROCESS
FILL TIMESERIES('day'), GROUP
```

This query returns the following output:

Received	City	Customer Type
01/01/2020, 00:00	Boston	Standard
02/01/2020, 00:00	Boston	null
03/01/2020, 00:00	Boston	null
04/01/2020, 00:00	Boston	null
01/01/2020, 00:00	New York	null
02/01/2020, 00:00	New York	Premium
03/01/2020, 00:00	New York	null
04/01/2020, 00:00	New York	null
01/01/2020, 00:00	San Francisco	null
02/01/2020, 00:00	San Francisco	null
03/01/2020, 00:00	San Francisco	null
04/01/2020, 00:00	San Francisco	Standard

Using GROUP BY and ORDER BY

The `GROUP BY` clause is applied after the filling. It's not related to the `GROUP` fill specification.

The `FILL` clause influences the definition and output of the `ORDER BY` clause. The `FILL` clause returns the filled time series for each group (as defined by the `GROUP` specification) in ascending order. Consequently, in the `ORDER BY` clause, the group column must be followed by the ascending time series column (as defined by `TIMESERIES(<precision>)`).

> **Note**
>
> If no `ORDER BY` is provided, the output is first grouped by the `GROUP` specified column and then by the `TIMESERIES` specified column.

Example

This example builds a query for showing the types of orders received on a daily basis. An overview of every day should be shown but not every type of order is sold every day, so we use `FILL` to fill any gaps. There are two types of customer: '`Standard`' and '`Premium`'.

```
SELECT
    DATE_TRUNC('day', (SELECT FIRST(end_time))) AS "Date",
    "Customer Type"
FROM THIS_PROCESS
GROUP BY 1, 2
```

Result:

Date	Customer Type
01/01/2020	Standard
02/01/2020	Premium
02/01/2020	Standard
03/01/2020	Premium
03/01/2020	Standard
04/01/2020	Premium
04/01/2020	Standard

As the output shows, no premium customers made orders on 1 January.

The next step is to add extra rows for each missing order type on each day. Specifically, we group the values in the '`Customer Type`' column into two groups and then add new rows for the missing days within each group. The value for the '`Customer Type`' column in the new rows takes the value of the current group.

To do that, add a `FILL` clause. `TIMESERIES('day')` identifies the first column as the time series. `GROUP` identifies '`Customer Type`' as the group column.

```
SELECT
    DATE_TRUNC('day', (SELECT FIRST(end_time))) AS "Date",
    "Customer Type"
FROM THIS_PROCESS
GROUP BY 1, 2
FILL
    TIMESERIES('day'),
    GROUP
```

Result:

Date	Customer Type
01/01/2020	Premium
02/01/2020	Premium
03/01/2020	Premium
04/01/2020	Premium

Date	Customer Type
05/01/2020	Premium
06/01/2020	Premium
07/01/2020	Premium
01/01/2020	Standard
02/01/2020	Standard
03/01/2020	Standard
04/01/2020	Standard
05/01/2020	Standard
06/01/2020	Standard
07/01/2020	Standard

Finally, we select the number of orders for each group on each date. The corresponding fill specification is `NULL`, meaning that column is filled with null values in the new rows.

```
SELECT
    DATE_TRUNC('day', (SELECT FIRST(end_time))) AS "Date",
    "Customer Type",
    COUNT(case_id)
FROM THIS_PROCESS
GROUP BY 1, 2
FILL
    TIMESERIES('day'),
    GROUP,
    NULL
```

The result shows the number of each order type on a daily basis:

Date	Customer Type	COUNT(case_id)
01/01/2020	Standard	4
	Premium	
02/01/2020	Standard	6
	Premium	4
03/01/2020	Standard	7
	Premium	1
04/01/2020	Standard	5
	Premium	4
05/01/2020	Standard	8
	Premium	7
06/01/2020	Standard	11
	Premium	4
07/01/2020	Standard	3

Date	Customer Type	COUNT(case_id)
	Premium	4

Related Information

COUNT [page 104]
DATE_TRUNC [page 174]
FIRST [page 109]

1.3.9 LIMIT Clause

Specifies the number of rows to return in a result set.

Without a `LIMIT` clause, the result set is limited to 500 rows by default.

> → Tip
>
> This clause is often used together with the `ORDER BY` clause, as demonstrated in the example.

Syntax

```
LIMIT <count>
```

Parameter	Description
`count`	The number of rows to include in the result set.

Example

Assume the following process data (`THIS_PROCESS`):

case_id	City	Order Amount	Order Status	Order Quantity
00001	Houston	944.42	Delivered	17
00002	San Francisco	270.04	Canceled	10
00003	Houston	469.9	Delivered	15
00004	San Francisco	268.34	Delivered	17

case_id	City	Order Amount	Order Status	Order Quantity
00005	Houston	327.94	Delivered	18
00006	San Francisco	599.07	Delivered	10
00007	San Francisco	521.17	Delivered	8
00008	Houston	162.58	Delivered	19
00009	Miami	165.44	Delivered	18
00010	Houston	319.18	Delivered	10

The following query returns the top five largest orders by quantity. It does so by sorting by '`Order Quantity`' (the fifth column) in descending order and limiting the result set to five rows.

```
SELECT
    case_id,
    "City",
    "Order Amount",
    "Order Status",
    "Order Quantity"
FROM THIS_PROCESS
ORDER BY 5 DESC
LIMIT 5
```

Output:

case_id	City	Order Amount	Order Status	Order Quantity
00008	Houston	162.58	Delivered	19
00005	Houston	327.94	Delivered	18
00009	Miami	165.44	Delivered	18
00001	Houston	944.42	Delivered	17
00004	San Francisco	268.34	Delivered	17

Related Information

ORDER BY Clause [page 30]

1.3.10 TABLESAMPLE Clause

Learn about the `TABLESAMPLE` clause in SIGNAL, the process mining query language of SAP Signavio Process Intelligence.

With the `TABLESAMPLE` clause, you can specify either the absolute or percentage table fraction to be considered for the query. The table fraction is sampled randomly after the `FROM` clause is evaluated and before all other clauses.

Syntax

Syntax:

```
TABLESAMPLE EXACT( { <probability> PERCENT | <num> ROWS } ) [ REPEATABLE( <seed>
) ]
```

Parameter	Description
`probability`	Percentage probability for selecting the sample. Any integer between 0 (no rows selected) and 100 (all rows selected) inclusive.
`num`	Number of rows to sample from the table
`seed`	Used for generating random numbers for the sampling method.

Example

```
SELECT "Order Amount in EUR", "Duration Order To Payment"
FROM (
    SELECT
        "Order Amount in EUR",
        (SELECT LAST (end_time) FILTER (WHERE event_name = 'Receive Payment')) -
        (SELECT FIRST (end_time) FILTER (WHERE event_name = 'Receive Customer
Order')) AS "Duration Order To Payment"
    FROM THIS_PROCESS
    WHERE
        ("Order Amount in EUR" IS NOT NULL)
    AND
        ((SELECT LAST (end_time) FILTER (WHERE event_name = 'Receive Payment')) -
        (SELECT FIRST (end_time) FILTER (WHERE event_name = 'Receive Customer
Order'))
        IS NOT NULL)
) AS sub_query
TABLESAMPLE EXACT(10 ROWS) REPEATABLE(123)
```

This query returns 10 rows sampled randomly from the specified table.

For `TABLESAMPLE EXACT(10 PERCENT) REPEATABLE(123)`, the query returns 10 percent of the rows sampled randomly from the specified table.

1.3.11 OFFSET Clause

Specifies the starting point to return rows from a result set.

The starting point is determined by providing the number of rows to skip from the top of the table. An offset of *N* rows means that a result set begins from the $(N+1)^{th}$ row.

> → Remember
>
> For `OFFSET` to provide consistent results, you must add an `ORDER BY` clause to the query.

Syntax

```
OFFSET <number>
```

Parameter	Description
`number`	The number of rows to skip from the top of the table.

Example

Assume the following process data (`THIS_PROCESS`):

case_id	City	Order Amount	Order Status	Order Quantity
00001	Houston	944.42	Delivered	17
00002	San Francisco	270.04	Canceled	10
00003	Houston	469.9	Delivered	15
00004	San Francisco	268.34	Delivered	17
00005	Houston	327.94	Delivered	18
00006	San Francisco	599.07	Delivered	10
00007	San Francisco	521.17	Delivered	8
00008	Houston	162.58	Delivered	19
00009	Miami	165.44	Delivered	18
00010	Houston	319.18	Delivered	10

The following query sorts the result set by order amount and skips the first four rows.

```
SELECT
    case_id,
    "City",
    "Order Amount",
    "Order Status",
    "Order Quantity"
FROM THIS_PROCESS
ORDER BY 3
OFFSET 4
```

Consequently, the four lowest value orders are excluded from the result set.

case_id	City	Order Amount	Order Status	Order Quantity
00010	Houston	319.18	Delivered	10
00005	Houston	327.94	Delivered	18
00003	Houston	469.9	Delivered	15
00007	San Francisco	521.17	Delivered	8
00006	San Francisco	599.07	Delivered	10
00001	Houston	944.42	Delivered	17

Related Information

ORDER BY Clause [page 30]

1.3.12 FLATTEN Operator

Flattens a table so that each event attribute becomes a top-level row. Case attributes are repeated accordingly.

> ⚠ Caution
>
> For very large data sets, this operator may require excessive CPU activity, causing long query execution times.
>
> For more information, refer to Performance [page 294].

The following example shows a typical table before being flattened:

case_id	Customer ID	Status	City	Events
1001	2001	delivered	New York	**event_name** / **end_time** Receive customer order / 2024-05-01T09:30:00 Receive payment / 2024-05-02T09:45:00 Ship goods / 2024-05-07T11:20:00
1002	2002	canceled	Boston	**event_name** / **end_time** Receive customer order / 2024-05-02T14:00:00 Cancel order / 2024-05-03T09:00:00

The same table after being flattened:

case_id	Customer ID	Status	City	event_name	end_time
1001	2001	delivered	New York	Receive customer order	2024-05-01T09:30:00
1001	2001	delivered	New York	Receive payment	2024-05-02T09:45:00
1001	2001	delivered	New York	Ship goods	2024-05-07T11:20:00

case_id	Customer ID	Status	City	event_name	end_time
1002	2002	canceled	Boston	Receive customer order	2024-05-02T14:00:00
1002	2002	canceled	Boston	Cancel order	2024-05-03T09:00:00

The flattened table allows aggregations based on event names or other event attributes.

> → Remember
>
> Don't combine the `FLATTEN` operator with the `MATCHES` operators. The `MATCHES` operators work only with nested tables and not with flattened tables.

Syntax

```
FLATTEN(<tableName>)
```

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	event_name
00002	Receive Customer Order
	Receive Payment
	Send items to Printing
	Order Canceled
00013	Receive Customer Order
	Ship Goods Express
	Receive Payment
	Receive Delivery Confirmation
00016	Receive Customer Order
	Receive Payment
	Ship Goods Express
	Receive Delivery Confirmation

case_id	event_name
00019	Receive Customer Order
	Receive Payment
	Ship Goods Express
	Receive Delivery Confirmation
00020	Receive Customer Order
	Receive Payment
	Ship Goods Standard
	Receive Delivery Confirmation

This query flattens the result set.

```
SELECT case_id, event_name
FROM FLATTEN (THIS_PROCESS)
```

Partial output (rows including case IDs 00019 and 00020 not shown):

case_id	event_name
00002	Receive Customer Order
00002	Receive Payment
00002	Send items to Printing
00002	Order Canceled
00013	Receive Customer Order
00013	Ship Goods Express
00013	Receive Payment
00013	Receive Delivery Confirmation
00016	Receive Customer Order
00016	Receive Payment
00016	Ship Goods Express
00016	Receive Delivery Confirmation

Example 2

Assuming the same process data as the previous example, the following query counts the number of cases in which each event occurs. By counting **distinct** case IDs, repeated events within a single case aren't counted.

```
SELECT COUNT (DISTINCT case_id), "event_name"
FROM FLATTEN (THIS_PROCESS)
GROUP BY 2
```

Output:

event_name	COUNT_DISTINCT(case_id)
Ship Goods Express	3
Order Canceled	1
Receive Customer Order	5
Send items to Printing	1
Ship Goods Standard	1
Receive Delivery Confirmation	4
Receive Payment	5

Related Information

FILTER EVENTS Clause [page 22]
Matching Expressions [page 80]

1.3.13 UNION ALL Clause

Combines the result sets of two or more `SELECT` statements.

The number of table columns and column data types must match for each `SELECT` statement.

Syntax

```
<selectStatement>
UNION ALL
<selectStatement>
```

`SELECT` statements before and after `UNION ALL` can only have the following clauses:

- `SELECT`
- `FROM`
- `TABLESAMPLE`
- `WHERE`
- `GROUP BY`

Other clauses, for example the `ORDER BY`, `LIMIT`, and `OFFSET` can only be applied to the result of `UNION ALL`.

Example 1

```
SELECT case_id AS "Case ID"
FROM
(
    SELECT case_id
    FROM THIS_PROCESS
    UNION ALL
    SELECT case_id
    FROM THIS_PROCESS
) as sub
ORDER BY 1
LIMIT 20
```

This query returns a combined result set of the `case_id` values from tables in a column.

Case ID
00001
00001
00002
00002
00003
00003
00004
00004

Example 2

```
SELECT case_id, ts, is_first
FROM
(
    SELECT
        case_id,
        DATE_TRUNC('day', (SELECT FIRST(end_time))) AS ts,
        1.0 AS is_first
    FROM THIS_PROCESS
    UNION ALL
    SELECT
        case_id,
        DATE_TRUNC('day', (SELECT LAST(end_time)) + DURATION '1day') AS ts,
        0.0 as is_first
    FROM THIS_PROCESS
) AS sub
ORDER BY 1, 2, 3
LIMIT 20
```

This query returns a combined result set of dates where an event was either open or closed on the given date. A timestamp (`ts`) is used, where 1 means the event is open and 0 means the event is closed on the respective dates.

case_id	ts	is_first
00001	08/01/2020	1
00001	08/10/2020	0
00002	04/06/2020	1
00002	04/10/2020	0
00003	21/02/2020	1
00003	03/03/2020	0
00004	10/09/2020	1
00004	17/10/2020	0

1.4 Subqueries

A subquery is a query that is nested inside another query.

There are two types of subquery: general and event-level. They differ based on the level at which they operate, and what types of data they return.

- General subqueries operate at case level and return tables.
- Event-level subqueries operate at event level and return scalar values.

They follow this general syntax:

```
SELECT (<eventLevelSubquery>) [, <columns> ...]
FROM (<generalSubquery>) AS <alias>
WHERE <expression> (<eventLevelSubquery>)
```

Related Information

Event-Level Subqueries [page 52]
General Subqueries [page 50]

1.4.1 General Subqueries

General subqueries operate at case level and return tables. They're used in combination with the `FROM` clause and are useful for adapting result sets before selecting from them.

Syntax

The syntax of a general subquery follows that of a standard query. The difference comes from its placement.

This is the partial syntax of a query:

```
SELECT <columnExpression> [, <columnExpression> ...]
FROM { table | (<generalSubquery>) AS <alias>}
```

Parameter	Description
`columnExpression`	The value or expression selected as a column in the result.
`generalSubquery`	A query that returns a table.
`alias`	An identifying label for the subquery. → Remember A general subquery always needs an alias, otherwise the query is invalid.

A general subquery follows the same syntax. However, the subquery is part of a `FROM` statement, providing to its containing query a table from which to select data. As it builds the table, the subquery can adapt the contents, for example by filtering or applying functions, providing the outer query with a customized data set.

Because a general subquery follows this same syntax, it's possible for a subquery to also select from a further nested subquery. This recursive behavior allows you to nest subqueries to an arbitrary extent.

Examples

The following example demonstrates using a general subquery.

Example

This query uses a general subquery to group orders by city and calculate their number. From the resulting table, only those rows having a number of cases greater than 500 are selected.

```
SELECT "City", "Cases"
FROM (
    SELECT "City", COUNT(1) AS "Cases"
    FROM THIS_PROCESS
    GROUP BY 1
) AS grouped_cities
```

```
WHERE "Cases" > 500
```

City	Cases
Washington	587
New York	566
Houston	1219

A general subquery can select from either a table, such as `THIS_PROCESS`, or from another nested subquery. This nested subquery might in turn select from a further nested subquery and so on. Let's look at an example.

Example

The following query shows the average payment amounts per city:

```
SELECT
    "City",
    AVG("Order Amount") AS "avg_payment"
FROM THIS_PROCESS
GROUP BY 1
```

Output:

City	avg_payment
San Francisco	374.311
Washington	378.419
New York	411.996
Miami	365.917
Boston	347.096
Houston	393.040

Developing this further, the next query shows only the average payment amounts per city above a certain amount.

```
SELECT
    "City",
    avg_payment
FROM (
    -- sub1 begins here
    SELECT
        "City",
        avg_payment
    FROM (
        -- sub2 begins here
        SELECT
            "City",
            AVG("Order Amount") AS "avg_payment"
        FROM THIS_PROCESS
        GROUP BY 1
    ) as sub2
    WHERE avg_payment > 375
) as sub1
```

The initial query has now become the innermost subquery (`sub2`). Its containing query, which is itself a subquery (`sub1`), adapts the data further by filtering out the cases where average payment is equal to or below 375. This provides the table for the outermost query to select from.

Output:

City	avg_payment
Washington	378.419
New York	411.996
Houston	393.040

1.4.2 Event-Level Subqueries

Event-level subqueries operate on event data and return scalar values. They're usually used with `SELECT` or `WHERE` statements.

Syntax

```
(SELECT <columnExpression> [, <columnExpression> ...] [ FROM
<eventLevelSubquery> ] ) AS <alias>
```

Parameter	Description
`columnExpression`	Value or expression selected as a column in the result.
`eventLevelSubquery`	An optional nested event-level subquery that returns a value to the current subquery. For further information, refer to section Nesting.
`alias`	An identifying label for the subquery. For further information, refer to section Nesting.

Event-level subqueries are actually expressions, meaning they can be used wherever expressions can be used. For example, they can be used as column expressions in a standard query:

```
SELECT case_id,
    (SELECT LAST(end_date))
FROM THIS_PROCESS
```

In this example, the second column is chosen using an event-level subquery: `(SELECT LAST(end_date))`. Note the parentheses surrounding the `SELECT` clause, which denote this as an event-level subquery. For each case in the process, this subquery drops to event-level and selects from the case's event data. In this example, the column `end_date` is selected. Because an event-level subquery must return a scalar value, the event data needs aggregating somehow up to a single row at the case level. The aggregation function `LAST` is used to return only the timestamp of the case's latest event, rather than a whole column of timestamps.

Nesting

An event-level subquery may optionally provide its own `FROM` clause.

- If included, this clause defines a nested event-level subquery for the containing subquery to select from.
- If a subquery doesn't include a `FROM` clause, it selects data from the table defined in the `FROM` clause of the outermost query.

Nesting of event-level subqueries can be done to an arbitrary extent:

```
(SELECT <cols> FROM
(
    ...
        (SELECT <cols> FROM
            (SELECT <cols>) AS <alias>
        ) AS <alias>
    ...
) [AS <alias>]
```

When nesting event-level subqueries, note the following:

- The innermost subquery doesn't require a `FROM` clause. It selects data from the table defined in the outermost query's `FROM` clause.
- All subqueries require an alias, the exception being the outermost subquery (where an alias is optional).
- A nested subquery doesn't need to use another event-level subquery to access event-level attributes. It already operates at event-level by virtue of being nested. Example 3 demonstrates this.

Example 1

The following query selects the ID of each case and uses an event-level subquery to access each case's event data. The subquery must return a scalar value, which in this example is the difference between the latest and earliest timestamps. Since it's already the innermost subquery, no `FROM` clause is necessary.

```
SELECT
    case_id,
    (SELECT LAST("end_time") - FIRST("end_time")) AS "Turnaround Time"
FROM THIS_PROCESS
```

Output:

case_id	Turnaround Time
00001	7d 20h
00002	3d 17h
00003	10d 11h
00004	6d 7h
00005	12d 7h

Example 2

Event-level subqueries can also be used in combination with the `WHERE` statement to filter out cases based on event attributes. In this example, payments are event-level data since several payments can be made per order. When an order is canceled, the last event in that case is '`Order Canceled`'.

```
SELECT case_id, (SELECT SUM("Payment Amount")) AS "Total Payment"
FROM THIS_PROCESS
WHERE (SELECT LAST(event_name)) <> 'Order Canceled'
```

This example uses two event-level subqueries.

- The first (in the `SELECT` clause) selects all payment amounts per case and sums them, yielding the total of that order.
- The second (in the `WHERE` clause) selects the last event in each case and filters out those orders that were canceled.

Output:

case_id	Total Payment
00006	599.07
00007	521.17
00008	162.58
00009	165.44
00010	319.18

Example 3

An event-level subquery can contain a nested subquery, as demonstrated in this example for finding the lowest payment received per order.

```
-- Outermost query
SELECT
    case_id,
    (
        -- Subquery
        SELECT MIN("Payment Amount")
        FROM (
            -- Nested subquery
            SELECT "Payment Amount"
            WHERE event_name = 'Receive Payment'
        ) as sub
    ) AS "Lowest Payment"
FROM THIS_PROCESS
```

The nested subquery, being the innermost, selects from the table defined by the outermost query (`THIS_PROCESS`) and provides a list of payment amounts for the current order. The containing subquery selects the minimum value from this list, thus finding the lowest payment and meeting its requirement to return a scalar value.

Output:

case_id	Lowest Payment
00006	599.07
00007	521.17
00008	162.58
00009	39.00
00010	319.18

1.5 Expressions

In SIGNAL, an expression is a combination of values, operators, and functions which is evaluated to return a single value.

SIGNAL provides the following types of expressions:

Arithmetic Expressions [page 55]
: Use arithmetic expressions to execute mathematical operations on numeric values.

Comparison Expressions [page 58]
: Comparison expressions accept two values and evaluate to a Boolean depending on the result of the comparison. They're typically used to establish the equality or inequality of two values.

Conditional Expressions [page 68]
: Conditional expressions return values dependent on the evaluation of Boolean conditions.

Literal Expressions [page 73]
: A literal is a fixed value of a certain type. SIGNAL supports literals of several different types.

Logical Expressions [page 78]
: A logical expression connects Boolean expressions using logical operators and can be evaluated to return a true or false value.

Matching Expressions [page 80]
: Matching expressions recognize patterns in sequences of event-level values within a case, returning true if a given pattern matches, or false otherwise.

1.5.1 Arithmetic Expressions

Use arithmetic expressions to execute mathematical operations on numeric values.

The following arithmetic operators are available:

Operator	Description
+	Add

Operator	Description
–	Subtract
*	Multiply
/	Divide
%	Modulo

Syntax

	Type Information				
`expression1 + expression2`					
	Number	+	Number	=	Number
	Time-stamp	+	Duration	=	Time-stamp
	Duration	+	Time-stamp	=	Time-stamp
	Duration	+	Duration	=	Duration
`expression1 - expression2`					
	Number	–	Number	=	Number
	Time-stamp	–	Time-stamp	=	Duration
	Time-stamp	–	Duration	=	Time-stamp
	Duration	–	Duration	=	Duration
`expression1 * expression2`					
	Number	*	Number	=	Number
	Number	*	Duration	=	Duration
	Duration	*	Number	=	Duration
`expression1 / expression2`					
	Number	/	Number	=	Number
	Duration	/	Number	=	Duration
	Duration	/	Duration	=	Number
`expression1 % expression2`					
	Number	%	Number	=	Number

Order of Operations

These operators follow the standard mathematical order of operations. This means:

- Evaluation proceeds from left to right. For example, 9 - 5 + 2 is interpreted as (9 - 5) + 2 = 6 rather than 9 - (5 + 2) = 2.
- Evaluation is performed in order of precedence (from higher to lower). For example, 5 + 2 * 2 is interpreted as 5 + (2 * 2) = 9 rather than (5 + 2) * 2 = 14.

Example 1

This query subtracts the order quantity from the shipment quantity, calculating any difference between an order's size and the total number of goods actually dispatched.

```
SELECT case_id, (SELECT SUM("Shipment Quantity")) - "Order Quantity" AS Diff
FROM THIS_PROCESS
```

Result (canceled orders, where no delivery took place, have no number displayed):

case_id	Diff
1	0
2	
3	0
4	0
5	0
6	10
7	0
8	0

Example 2

This query divides each payment received by the total amount of the order and multiplies by 100. The resulting figure is the percentage of the full amount each individual payment represents.

```
SELECT
    case_id,
    "Payment Amount" / "Order Amount" * 100 AS "Proportion %"
FROM FLATTEN(THIS_PROCESS)
WHERE event_name = 'Receive Payment'
```

Result:

case_id	Proportion %
6	100.000
7	100.000
8	100.000
9	76.426
9	23.574
10	100.000

Example 3

This query returns the remainder when dividing the shipment quantity by 15. Assuming goods from an order are shipped in containers with a capacity of 15, this query calculates how many units were shipped in a non-full container.

```
SELECT "Shipment Quantity" % 15 AS "Units shipped in non-full container"
FROM FLATTEN(THIS_PROCESS)
WHERE ("Shipment Quantity" IS NOT NULL)
```

Result:

Units shipped in non-full container
2
0
2
3
0
5
8
4

1.5.2 Comparison Expressions

Comparison expressions accept two values and evaluate to a Boolean depending on the result of the comparison. They're typically used to establish the equality or inequality of two values.

Value Comparison [page 59]

These operators allow you to compare scalar values, which can be tested for equality. Depending on the data type, such values can also be tested for their ordering, that is whether one value is greater than or less than the other.

IN [page 61]
: Tests whether a value is equal to any value in a specified list. If so, then the whole expression evaluates to true, otherwise it evaluates to false.

IS NULL [page 63]
: Evaluate whether or not an expression is `NULL`.

LIKE and ILIKE [page 64]
: Search for a specified string pattern in a column of string data type.

1.5.2.1 Value Comparison

These operators allow you to compare scalar values, which can be tested for equality. Depending on the data type, such values can also be tested for their ordering, that is whether one value is greater than or less than the other.

The following operators are available:

Operator	Description
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal to
<>	Not equal to

Syntax

```
<expression1> { = | <> | < | <= | > | >= } <expression2>
```

Parameters	Description
`expression1`	The first expression to be compared.
`expression2`	The second expression to be compared.

Type Information

This table summarizes which combinations of data types are valid with the various comparison operators and what type of result each combination yields.

Type of `expression1`	Type of `expression2`	Valid Operators	Result Type
Number	Number	=, <>, <, <=, =>, >	Boolean
Timestamp	Timestamp	=, <>, <, <=, =>, >	Boolean
Duration	Duration	=, <>, <, <=, =>, >	Boolean
Text	Text	=, <>	Boolean
Boolean	Boolean	=, <>, <, <=, =>, >	Boolean

> **Note**
>
> When comparing Boolean types, `false < true`.

Example 1

The following example query uses the comparison operations, '>', '<', and '=' for defining conditions in the `WHERE` clause. It returns a list of delivered orders whose order amount is between 100 and 900. If the order status is 'Delivered' and the order amount is greater than 100 and less than 900, then the query returns the case IDs of these orders along with other details.

```
SELECT case_id, "order amount", "Order Status"
FROM THIS_PROCESS
WHERE ("ORDER Status" = 'Delivered' AND "order amount" > 100 AND "order amount"
< 900)
LIMIT 10
```

The query returns the following result.

case_id	order amount
00003	469.9
00004	268.34
00005	327.94
00006	599.07
00007	521.17
00008	162.58
00009	165.44
00010	319.18

Example 2

The following example query uses the comparison operations, '>=', '<=', and '<>' for defining conditions in the `WHERE` clause. It returns a list of orders that are not delivered and with the order amount between 300 and

2000. If the order status is not 'Delivered' and the order amount is greater than or equal to 300 and less than or equal to 2000, then the query returns the case IDs of these orders along with other details.

```
SELECT "case_id", "ORDER AMOUNT" AS "Orders between 300to2000", "order status"
FROM THIS_PROCESS
WHERE ("order amount" >= 300 AND "order amount" <= 2000 AND "Order status" <>
'Delivered')
LIMIT 10
```

The query returns the following result.

case_id	Orders between 300to2000	order status
00056	354.47	Canceled
00061	924.96	Canceled
00075	1,281.46	Canceled
00114	517.91	Canceled
00168	602.85	Canceled
00171	588.06	Canceled
00208	679.66	Canceled
00211	768.78	Canceled

1.5.2.2 IN

Tests whether a value is equal to any value in a specified list. If so, then the whole expression evaluates to true, otherwise it evaluates to false.

Syntax

```
<valueExpression> [NOT] IN (<value> [, <value> ...])
```

Parameter	Description	Type
`valueExpression`	A column or expression that evaluates to a single value.	Any
`value`	A literal value.	Any **Note** The types of the values in the list must match the type of `valueExpression`.

Example 1

Our example process data records sales of several types of items.

```
SELECT "Type of Goods"
FROM THIS_PROCESS
GROUP BY 1
```

Type of Goods
Cappy
T-shirt with Print
Cappy with Print
Hoody
Hoody with Print
T-shirt

You can use `IN` to select only a subset of those entries, as the following example does.

```
SELECT "Type of Goods"
FROM THIS_PROCESS
WHERE "Type of Goods" IN ('Cappy', 'Cappy with Print')
GROUP BY 1
```

Output:

Type of Goods
Cappy
Cappy with Print

Example 2

Let's take the previous example further. The following query uses `IN` to find the total number of items of headgear sold – in other words, items whose type is '`Cappy`' or '`Cappy with Print`' – and compares it with the number of all other types sold.

```
SELECT
    SUM("Order Quantity") AS "Sold",
    IF ("Type of Goods" IN ('Cappy', 'Cappy with Print'), 'Yes', 'No') AS
"Headgear?"
FROM THIS_PROCESS
```

The query gives each order a label called "`Headgear?`". If the value of the order's `Type of Goods` field matches with either '`Cappy`' or '`Cappy with Print`', then the label becomes '`Yes`'. Otherwise, the label becomes '`No`'. The `SUM` function totals the items in both categories.

Headgear?	Sold
Yes	19,622
No	22,383

1.5.2.3 IS NULL

Evaluate whether or not an expression is `NULL`.

Syntax

```
<expression> IS [NOT] NULL
```

Parameter	Description
`expression`	The expression to test for a `NULL` value.

`IS NULL` returns `true` if an expression evaluates to `NULL`. Otherwise, it returns `false`.

Alternatively, `IS NOT NULL` returns `true` if an expression doesn't evaluate to `NULL`. Otherwise, it returns `false`.

Example - IS NULL

Assume the following process data:

Case ID	Event name	Shipment Number	Shipment Carrier
00001	Receive Customer Order	null	null
00001	Change Order Quantity	null	null
00001	Receive Payment	null	null
00001	Ship Goods Express	U8787	UPS
00001	Receive Delivery Confirmation	U8787	UPS

The following query returns all events where the shipment number is `NULL`.

```
SELECT
    case_id AS "Case ID",
    event_name AS "Event name",
    "Shipment Number",
    "Shipment Carrier"
FROM FLATTEN(THIS_PROCESS)
```

```
WHERE "Shipment Number" IS NULL
```

Result:

Case ID	Event name	Shipment Number	Shipment Carrier
00001	Receive Customer Order	null	null
00001	Change Order Quantity	null	null
00001	Receive Payment	null	null

Example - IS NOT NULL

Assuming the same process data as the previous example, the following query returns all events where the shipment number isn't `NULL`.

```
SELECT
    case_id AS "Case ID",
    event_name AS "Event name",
    "Shipment Number",
    "Shipment Carrier"
FROM FLATTEN(THIS_PROCESS)
WHERE "Shipment Number" IS NOT NULL
```

Result:

Case ID	Event name	Shipment Number	Shipment Carrier
00001	Ship Goods Express	U8787	UPS
00001	Receive Delivery Confirmation	U8787	UPS

1.5.2.4 LIKE and ILIKE

Search for a specified string pattern in a column of string data type.

The `LIKE` operator matches case-sensitive string patterns, whereas the `ILIKE` operator matches case-insensitive string patterns.

These operators provide:

- flexible string comparison using special characters called wildcards
- improved string filtering and bucketing
- flexible process variant filtering using `MATCHES` and `BEHAVIOR MATCHES`

By adding the `NOT` keyword, you can search for non-matching string patterns.

Syntax

```
<expression> [NOT] LIKE <pattern>
```

```
<expression> [NOT] ILIKE <pattern>
```

Parameter	Description
`expression`	The string expression that is compared with the defined pattern.
`pattern`	The defined string pattern the expression is compared with. The string pattern includes special characters percent (%) and underscore (_) to conduct a character-by-character comparison.

These operators return `true` or `false` depending on whether a pattern matches.

For example:

Expression	Output
`'Amsterdam' LIKE 'A%'`	`true`
`'Ibiza' ILIKE '%A'`	`true`
`'Boston' NOT LIKE 'B%'`	`false`
`'Miami' NOT ILIKE '%I'`	`false`

Wildcards

The `LIKE` and `ILIKE` operators use special characters called wildcards in the string pattern. They have specific meaning when used in a pattern:

- The percent (%) wildcard represents zero, one, or multiple characters.
- The underscore (_) wildcard represents exactly one character.

If you wish to search for a literal percent or underscore character in a string, use the backslash symbol (\) to escape that wildcard's special meaning. For example: `LIKE '100\%'`

You can also combine these special characters in a pattern:

Example Expression	Description
`ILIKE '%to%'`	Returns `true` if the matching string pattern contains "to" in any case and in any position. Example matches: 'Washington', 'Boston', 'Palo Alto', 'Stockton'
`LIKE '_o%'`	Returns `true` if the matching string pattern contains "o" in the second position. Example matches: 'Boston', 'Rostock'
`ILIKE 'm__%'`	Returns `true` if the matching string pattern starts with "m" or "M" and is at least 3 characters in length. Example matches: 'Miami', 'Munich'
`ILIKE 'b%n'`	Returns `true` if the matching string pattern starts with "b" or "B", and ends with "n" or "N" Example matches: 'Berlin', 'bern', 'BOSTON'

LIKE and ILIKE with MATCHES

You can also use `LIKE` and `ILIKE` with `MATCHES` and `BEHAVIOR MATCHES` for filtering nested data. Refer to Examples 2 and 3.

Example 1

The following example displays the use of wildcards in the string pattern.

```
SELECT "Type of Goods", COUNT("Type of Goods")
FROM FLATTEN(THIS_PROCESS)
WHERE "Type of Goods" ILIKE '%with print'
ORDER BY 1
```

Result:

The query collects '`Type of Goods`' entries ending with the value '`with print`'.

Type of Goods	COUNT(Type of Goods)
Cappy with Print	3,266
Hoody with Print	2,872
T-shirt with Print	4,060

Example 2

The following example displays the use of `ILIKE` expression in a `MATCHES` expression.

```
SELECT count(case_id)
FROM THIS_PROCESS
WHERE event_name MATCHES ('Receive Customer Order' ~> ILIKE '%print%' ~> ILIKE
'%ship%')
```

Result:

The query returns the count of `case_id`s from `THIS_PROCESS` where a print event occurred, and where the order was shipped.

COUNT(case_id)
1105

Example 3

The following example displays the use of `ILIKE` in `BEHAVIOR MATCHES`.

```
SELECT count(case_id)
FROM THIS_PROCESS
WHERE
BEHAVIOR
    (event_name == 'Receive Customer Order') as order_received,
    (event_name ILIKE '%print%') as order_printed,
    (event_name ILIKE '%ship%') as order_shipped
MATCHES (order_received ~> order_printed ~> order_shipped)
```

Result:

The query returns the count of `case_id`s from `THIS_PROCESS` where a print event occurred, and where the order was shipped.

COUNT(case_id)
1105

Example 4

The following example displays the use of `NOT LIKE` .

```
SELECT DISTINCT(event_name)
FROM FLATTEN(THIS_PROCESS)
WHERE event_name NOT LIKE '%Purchase%'
```

Result:

The query selects the `event_name` from `THIS_PROCESS` where the `event_name` doesn't contain the string '`Purchase`'.

event_name
Receive items from Printing
Order Canceled
Ship Goods Express
Receive Customer Order
Ship Goods Standard
Receive Delivery Confirmation
Change Order Quantity
Items Printed
Send items to Printing
Receive Payment

Related Information

Matching Expressions [page 80]

1.5.3 Conditional Expressions

Conditional expressions return values dependent on the evaluation of Boolean conditions.

The following conditional expressions are available:

CASE WHEN [page 69]
: Evaluates a list of conditions and returns a value associated with the first condition that is met.

IF [page 71]
: Evaluates a single condition and returns one of two values accordingly.

1.5.3.1 CASE WHEN

Evaluates a list of conditions and returns a value associated with the first condition that is met.

Syntax

```
CASE
    WHEN <condition> THEN <result>
    [WHEN <condition> THEN <result>] ...
    [ELSE <defaultResult>]
END
```

Each condition is an expression that evaluates to a Boolean value. Conditions are evaluated one at a time from top to bottom. Once a condition is met, its associated result is returned and no further conditions are evaluated.

If no conditions evaluate to true, one of two things happens:

- If an `ELSE` clause is present, its associated value is returned. This optional clause provides a means to supply a default value.
- If no `ELSE` clause is present, a `NULL` value is returned.

Usage

The `CASE WHEN` expression can be used in the following clauses:

- `SELECT`
- `WHERE`

The following operators can be used in the `WHEN` clause:

Operator / Key words	Example
<	`WHEN "Order Amount" < '1000' THEN 'Approved'`
>	`WHEN "Order Amount" > '1000' THEN 'Approved'`
<=	`WHEN "Order Amount" <= '1000' THEN 'Approved'`
>=	`WHEN "Order Amount" >= '1000' THEN 'Approved'`

Operator / Key words	Example
=	`WHEN "Order Amount" = '1000' THEN 'Approved'`
<>	`WHEN "Order Amount" <> '1000' THEN 'Rejected'`
IN	`WHEN "Country" IN ('Germany', 'France') THEN 'Europe'`
NOT IN	`WHEN "Country" NOT IN ('Germany', 'France') THEN 'Rest of the World'`

Example 1

The following example selects countries based on regions. When a country is in a specific region (condition is true), then it returns the associated region (result). If a country doesn't match any of the conditions, the default value is returned ("Rest of World").

```
SELECT DISTINCT
     "Country",
    CASE WHEN "Country" = 'USA' THEN 'North America'
         WHEN "Country" = 'Germany' THEN 'EU'
         WHEN "Country" = 'France' THEN 'EU'
         WHEN "Country" = 'South Africa' THEN 'Africa'
         ELSE 'Rest of World'
         END AS "Region"
FROM THIS_PROCESS
```

Output:

Country	Region
USA	North America
France	EU
Greenland	Rest of World

Example 2

The following example compares the minimum and maximum values of customer satisfaction ratings (CSAT) at an event level in a `CASE WHEN` expression.

```
SELECT "Country",
    CASE
        WHEN (SELECT(MAX("CSAT"))) = (SELECT(MIN("CSAT"))) THEN 'None'
        WHEN (SELECT(MAX("CSAT"))) - (SELECT(MIN("CSAT"))) <= 1 THEN 'Low'
```

```
        WHEN (SELECT(MAX("CSAT"))) - (SELECT(MIN("CSAT"))) <= 3 THEN 'Medium'
        ELSE 'High'
    END AS "CSAT Delta",
    count(case_id)
FROM THIS_PROCESS
GROUP BY 1,2
```

The query returns the total number of cases in each CSAT Delta category (None, Low, Medium, and High) and Country. The CSAT Delta category is determined by the delta between the maximum CSAT and minimum CSAT value within the case.

Country	CSAT Delta	COUNT(case_id)
USA	Low	82
	None	36
	High	913
	Medium	1,372
Germany	None	11
	Medium	442
	Low	27
	High	310
France	High	301
	Low	28
	Medium	461
	None	17

1.5.3.2 IF

Evaluates a single condition and returns one of two values accordingly.

Syntax

```
IF(<condition>, <valueIfTrue>, <valueIfFalse>)
```

Parameter	Description
`condition`	The condition to be evaluated.
`valueIfTrue`	The value returned if the condition is met. This value can be a literal, column attribute, or expression.
`valueIfFalse`	The value returned if the condition isn't met. This value can be a literal, column attribute, or expression.

- If `condition` evaluates to true, `valueIfTrue` is returned.
- If `condition` evaluates to false, `valueIfFalse` is returned.

Nested IF

`IF` is an expression, meaning that an `IF` expression can be used as the `valueIfTrue` or `valueIfFalse`. An `IF` expression included in this manner is called a nested `IF` expression. The number of nested `IF` expressions is limited to 256.

Example 1

The following example returns a distinct list of cities along with an associated state. The `IF` expression is used to compare `City` to a literal value 'San Francisco'. If the name matches the defined condition, the state value is returned as 'California'. If `City` doesn't match the defined condition, then the state value is returned as 'Other'.

```
SELECT
    DISTINCT "City",
    IF("City" = 'San Francisco', 'California', 'Other') AS "State"
FROM THIS_PROCESS
```

Result:

City	State
New York	Other
Boston	Other
Miami	Other
Washington	Other
Houston	Other
San Francisco	California

Example 2

The following example demonstrates nested `IF` expressions. The expression returns a distinct list of cities along with corresponding states. The first `IF` expression is used to compare `City` to a literal value 'San Francisco'. If the name matches the defined condition, the state value is returned as 'California'. If `City` doesn't match the defined condition, the next nested `IF` expression is evaluated. If the state doesn't match the condition in the last `IF` expression, then the state value is returned as 'Other'.

```
SELECT
    DISTINCT "City",
    IF(
        "City" = 'San Francisco',
```

```
        'California',
        IF(
            "City" = 'Miami',
            'Florida',
            IF("City" = 'Houston', 'Texas', 'Other')
        )
    ) AS "State"
FROM THIS_PROCESS
```

Result:

City	State
Washington	Other
Houston	Texas
Boston	Other
Miami	Florida
New York	Other
San Francisco	California

1.5.4 Literal Expressions

A literal is a fixed value of a certain type. SIGNAL supports literals of several different types.

Boolean Literal [page 73]
A representation of a Boolean value.

Date and Timestamp Literals [page 74]
Create values representing calendar dates and times.

Duration Literal [page 76]
A representation of a time interval.

Number Literal [page 77]
Represents literal numeric values.

String Literal [page 77]
Represents a fixed sequence of characters.

1.5.4.1 Boolean Literal

A representation of a Boolean value.

A Boolean literal has a value of either:

- TRUE
- FALSE

Example

Assume the following input data:

case_id	Order Amount
02302	404.54
01553	482.85
00681	1030.71
01153	700.68
01524	706.96

Let's say that orders worth over 500 qualify for a discount. This query shows which orders qualify.

```
SELECT
    case_id,
    "Order Amount",
    IF("Order Amount" > 500, TRUE, FALSE) AS "Qualifies for Discount"
FROM THIS_PROCESS
```

case_id	Qualifies for Discount
02302	False
01553	False
00681	True
01153	True
01524	True

1.5.4.2 Date and Timestamp Literals

Create values representing calendar dates and times.

DATE

Create a literal value representing a calendar date by using the `DATE` operator as follows:

```
DATE <value>
```

The `value` argument must be a string literal containing the date value in the format 'YYYY-MM-DD'. Refer to the section Format Symbols for further explanation.

> **Note**
>
> The `DATE` operator actually returns a value of the type Timestamp with the associated time set to 00:00:00.

TIMESTAMP

Create a literal value representing a timestamp by using the `TIMESTAMP` operator as follows:

```
TIMESTAMP <value>
```

The `value` argument must be a string literal containing the timestamp value in the format 'YYYY-MM-DDTHH:mm:ss'. Refer to the section Format Symbols for further explanation.

Format Symbols

The symbols describing the format of the value accepted by the `DATE` and `TIMESTAMP` operators have the following meanings:

Character Pattern	Description
YYYY	Four-digit year
MM	Two-digit month
DD	Two-digit day of the month
HH	Two-digit hour
mm	Two-digit minute
ss	Two-digit second

All other symbols are included using their literal representation.

> → Remember
>
> The value isn't automatically padded, so you must include zeroes where necessary. For example, January 2, 2024 must be represented using '2024-01-02'. The string '2024-1-2' is considered invalid.

Examples

- `DATE '2024-09-25'` creates a timestamp representing midnight (00:00:00) on September 25, 2024.
- `TIMESTAMP '2024-10-05T14:43:00'` creates a timestamp representing 14:43 exactly on October 5, 2024.

1.5.4.3 Duration Literal

A representation of a time interval.

Create a literal value of the type Duration by using the `DURATION` operator.

```
DURATION <value>
```

The `value` argument must be a string literal representing a duration. This value includes the time unit and a number indicating how many of those units the duration covers, for example:

- `'3weeks'`
- `'4days'`
- `'3hours'`
- `'5minutes'`
- `'6seconds'`
- `'7milliseconds'`

→ Tip

- Combined usage is possible, for example `'1day1hour'`.
- For all duration strings, the singular and plural forms are supported.
- You can also follow the standard ISO-8601, for example `'P1D6H30M'`.

⚠ Restriction

Because years and months can be of varying length, it's not possible to specify a duration literal using the units `year` or `month`. Attempting to do this results in an error.

As an alternative, consider specifying the number of days or weeks.

Example

Assume the following data, where the columns `Started` and `Ended` are the timestamps of the earliest and latest events in each case respectively:

case_id	Started	Ended
00001	01/08/2020, 12:32	09/08/2020, 09:19
00002	06/04/2020, 06:38	09/04/2020, 23:43
00003	21/02/2020, 09:14	02/03/2020, 20:15
00004	09/10/2020, 19:26	16/10/2020, 02:45
00005	25/12/2020, 18:21	07/01/2021, 01:24

The following query filters cases, including only cases which took longer than 10 days to process:

```
SELECT case_id
```

```
FROM THIS_PROCESS
WHERE (SELECT LAST(end_time) - FIRST(end_time)) > DURATION '10days'
```

Output:

case_id
00003
00005

1.5.4.4 Number Literal

Represents literal numeric values.

You can create numeric values using the following conventions:

- Integer digits: 0-9
- Decimal values: 1.5 | .5
- Scientific notation: 1e-10 | .5e+1000

1.5.4.5 String Literal

Represents a fixed sequence of characters.

A string literal is represented by a sequence of characters enclosed in single quotes, for example: `'This is a string'`. The quote characters denote the start and end of the string.

> → Tip
>
> To include a quote character inside a string literal, follow the quote character immediately with another quote character. For example, the name O'Reilly can be put into a string literal like this: `'O''Reilly'`.

> → Remember
>
> Strings are case-sensitive. This means, for example, that the literals 'Smith' and 'smith' are considered different.

1.5.5 Logical Expressions

A logical expression connects Boolean expressions using logical operators and can be evaluated to return a true or false value.

The following logical operators are available:

Operator	Description
AND	True if all the expressions separated by AND are true.
NOT	Negates the expression.
OR	True if any of the expressions separated by OR is true.

AND, OR

Syntax:

```
<expression1> AND <expression2>
```

```
<expression1> OR <expression2>
```

Parameters	Description
expression1	The first expression to be evaluated.
expression2	The second expression to be evaluated.

Operator Precedence

These two operators have different operator precedence. Specifically, AND takes precedence over OR. In compound expressions including both operators, this means that AND operations are evaluated before OR operations. For example, `A OR B AND C` is interpreted as `A OR (B AND C)`.

To override this and force OR operations to take precedence, enclose the relevant part of the expression in parentheses, for example, `(A OR B) AND C`.

NOT

Syntax:

```
NOT(<expression>)
```

Parameters	Description
expression	The expression to be negated.

Type Information

This table summarizes which combinations of data types are valid with the various logical operators and what type of result each combination yields.

Type of `expression1`	Type of `expression2`	Valid Operators	Result Type
Boolean	Boolean	AND, OR	Boolean
Boolean		NOT	Boolean

Example 1: AND

The following example returns a list of delivered orders in the city of Boston. If the order status is delivered and the city is Boston, then the query returns the case IDs of these orders along with other details.

```
SELECT case_id, "City", "Order Status", "order amount"
FROM THIS_PROCESS
WHERE ("ORDER Status" = 'Delivered' AND "City" = 'Boston')
LIMIT 10
```

The query returns the following result.

case_id	City	Order Status	order amount
26	Boston	Delivered	101.23
38	Boston	Delivered	39.42
42	Boston	Delivered	313.88
43	Boston	Delivered	502.89
52	Boston	Delivered	112.01
62	Boston	Delivered	102.37
76	Boston	Delivered	556.53
93	Boston	Delivered	509.92

Example 2: OR

The following example returns orders where the value is less than 50 or the value is greater than 2000 and the city is Houston. Keep in mind the preceding remark noting how AND operations take precedence over OR operations.

```
SELECT case_id, "order amount", "Order Status", "city"
FROM THIS_PROCESS
WHERE ("Order amount" <= 50 OR "Order amount" >= 2000 AND "City" = 'Houston')
ORDER BY 2 DESC
LIMIT 15
```

The query returns the following result.

case_id	order amount	Order Status	city
382	2,166.14	Canceled	Houston
2491	2,059.67	Canceled	Houston
2806	2,039.52	Canceled	Houston
1601	2,037.38	Canceled	Houston
2531	2,028.77	Delivered	Houston
1802	2,021.11	Delivered	Houston
1925	2,003.98	Delivered	Houston
2171	49.92	Delivered	San Francisco
2312	49.75	Delivered	Washington
1273	49.75	Canceled	Washington
762	49.75	Delivered	Houston

Example 3: `NOT`

The following example returns the total order amount of all purchases excluding orders from the cities Boston and Miami.

```
SELECT SUM("Order Amount")
FROM THIS_PROCESS
WHERE NOT("City" = 'Boston' AND "City" = 'Miami')
```

The query returns the following result.

SUM(Order Amount)
1266532.97

1.5.6 Matching Expressions

Matching expressions recognize patterns in sequences of event-level values within a case, returning true if a given pattern matches, or false otherwise.

> ⚠ Restriction
>
> Matching expressions work only with event-level columns. They don't work with flattened data. Don't combine a matching expression with the `FLATTEN` operator.

Matching Patterns

Matching patterns are used with the `MATCHES` and `BEHAVIOR MATCHES` operators to form matching expressions.

Matching patterns are constructed from the following matching operators:

Name	Operator	Example	Description
Starts with	`^`	`^ A`	Sequence starts with A in any specific case.
Ends with	`$`	`A $`	Sequence ends with A in any specific case.
Directly follows	`->`	`A -> B`	A directly followed by B. Equivalent to `A B`.
Follows	`~>`	`A ~> B`	A directly or indirectly followed by B. Equivalent to `A ANY* B`.
Matches any	`ANY`	`A ANY B`	A is followed by any single value, which is then followed by B.
Matches NULL	`NULL`	`A NULL B`	A is followed by NULL, which is then followed by B.
Matches one from a set	`|`	`(A | B)`	All values that are A or B.
Doesn't match	`NOT`	`NOT A`	All values that are not A (even if the sequence contains A values).

You can further tailor your matching expression by using quantifiers. Apply quantifiers to any part of your expression that matches events:

Name	Quantifier	Example	Description
Zero or more	`*`	`A ANY* B`	A is followed by any number of values, which are then followed by B. Equivalent to `A ~> B`.
One or more	`+`	`A+ -> B`	At least one A is followed directly by B.
Zero or one	`?`	`B? $`	The sequence ends with at most one B.
Range	`{min, max}`	`A ANY{2, 3} B`	A is followed by a minimum of two and a maximum of three events, which are then followed by B. **Note** The maximum value is optional.

Examples

See the `MATCHES` and `BEHAVIOR MATCHES` documentation for examples of using these patterns.

Related Information

MATCHES [page 82]
BEHAVIOR MATCHES [page 87]

1.5.6.1 MATCHES

Filters based on a matching pattern that checks for a certain sequence of event-level values, such as `event_name`.

Syntax

```
<expression> MATCHES (<pattern>)
```

Parameter	Description
`expression`	The column or expression that is compared with the matching pattern.
`pattern`	The matching pattern with which the expression is compared. Refer to the section Matching Patterns [page 81].

Example 1

Match cases where the first event records receipt of a customer order and the delivery confirmation follows at any point later on.

```
SELECT case_id, event_name
FROM THIS_PROCESS
WHERE event_name MATCHES(^'Receive Customer Order' ~> 'Receive Delivery
Confirmation')
```

→ Tip

The matching pattern in this example could alternatively be expressed as: `^'Receive Customer Order' ANY* 'Receive Delivery Confirmation'`.

Output:

case_id	event_name
00001	Receive Customer Order
	Change Order Quantity
	Receive Payment
	Ship Goods Express
	Receive Delivery Confirmation
00003	Receive Customer Order
	Change Order Quantity
	Receive Payment
	Ship Goods Standard
	Receive Delivery Confirmation
00004	Receive Customer Order
	Change Order Quantity
	Receive Payment
	Ship Goods Express
	Receive Delivery Confirmation

Example 2

Match cases where exactly one event of any kind occurs between a payment being received and the order being canceled.

```
SELECT case_id, event_name
FROM THIS_PROCESS
WHERE "event_name" MATCHES ('Receive Payment' ANY 'Order Canceled')
```

Output:

case_id	event_name
00114	Receive Customer Order
	Receive Payment
	Send items to Printing
	Order Canceled

case_id	event_name
00142	Receive Customer Order
	Receive Payment
	Receive Payment
	Order Canceled

Example 3

Match cases where one or more events of any kind occur between receiving a customer order and shipping it as express.

```
SELECT case_id, event_name
FROM THIS_PROCESS
WHERE event_name MATCHES ('Receive Customer Order' ANY+ 'Ship Goods Express')
```

Output:

case_id	event_name
00001	Receive Customer Order
	Change Order Quantity
	Receive Payment
	Ship Goods Express
	Receive Delivery Confirmation
00004	Receive Customer Order
	Change Order Quantity
	Receive Payment
	Ship Goods Express
	Receive Delivery Confirmation

case_id	event_name
00005	Receive Customer Order
	Change Order Quantity
	Change Order Quantity
	Receive Payment
	Send items to Printing
	Items Printed
	Receive items from Printing
	Ship Goods Express
	Receive Delivery Confirmation

Example 4

Match cases where either zero or one event of any kind occurs between receiving a customer order and shipping it as express.

```
SELECT case_id, event_name
FROM THIS_PROCESS
WHERE event_name MATCHES ('Receive Customer Order' ANY? 'Ship Goods Express')
```

Output:

case_id	event_name
00013	Receive Customer Order
	Ship Goods Express
	Receive Payment
	Receive Delivery Confirmation
00016	Receive Customer Order
	Receive Payment
	Ship Goods Express
	Receive Delivery Confirmation
00019	Receive Customer Order
	Receive Payment
	Ship Goods Express
	Receive Delivery Confirmation

Example 5

Match cases where at least three but not more than four events of any kind occur between receiving a customer order and shipping it as express.

```
SELECT case_id, event_name
FROM THIS_PROCESS
WHERE event_name MATCHES ('Receive Customer Order' ANY{3,4} 'Ship Goods Express')
```

Output:

case_id	event_name
00006	Receive Customer Order
	Receive Payment
	Send items to Printing
	Items Printed
	Receive items from Printing
	Ship Goods Express
	Ship Goods Express
	Receive Delivery Confirmation
	Receive Delivery Confirmation
00022	Receive Customer Order
	Receive Payment
	Send items to Printing
	Items Printed
	Receive items from Printing
	Ship Goods Express
	Receive Delivery Confirmation
00035	Receive Customer Order
	Receive Payment
	Send items to Printing
	Items Printed
	Receive items from Printing
	Ship Goods Express
	Ship Goods Express
	Receive Delivery Confirmation
	Receive Delivery Confirmation

Related Information

Matching Expressions [page 80]

1.5.6.2 BEHAVIOR MATCHES

An extension of the `MATCHES` operator allowing filtering that is more complex and able to incorporate case- and event-level data.

> → Tip
>
> Refer to the section Comparison with MATCHES [page 88] for an introductory discussion of this feature.

Syntax

```
BEHAVIOR
    (<expression>) AS <alias_name>
    [, (<expression>) AS <alias_name> ...]
MATCHES (<pattern>)
```

> ⓘ Note
>
> It is also acceptable to use the British English spelling of the this operator, `BEHAVIOUR`.

Parameter	Description
`expression`	A Boolean expression that filters cases for inclusion in the following `MATCHES` clause. Attributes in this expression can be case- or event-level.
`alias_name`	The name to assign to this behavior. You can then use this name in the matching pattern. > ⓘ Note > Enclose the name in double quotes if it contains a character that isn't a letter or digit.
`pattern`	The pattern used to match against the cases exhibiting the defined behaviors. See the section Matching Patterns [page 81].

> → Remember
>
> Up to eight behaviors can be defined, each of which require an alias.

Comparison with MATCHES

The MATCHES operator works by evaluating an event-level attribute to determine whether a specified pattern is present or not. When the pattern is not present in a case, that entire case is excluded from the result set.

For example, if you want to match only cases where a specified standard sales process is followed, then using MATCHES suffices. For demonstration purposes, let's say a standard sales process includes the events "Receive Customer Order", "Receive Payment", and "Ship Goods Standard" strictly in that order and with no other events preceding or following:

```
SELECT
    case_id,
    event_name
FROM THIS_PROCESS
WHERE
    event_name MATCHES (^ 'Receive Customer Order'
        -> 'Receive Payment'
        -> 'Ship Goods Standard'
        $
    )
```

BEHAVIOR MATCHES extends this concept by allowing the inclusion of extra filtering criteria. For example, in addition to matching cases that follow the standard process, we might also want to include only cases where the order was shipped after a specific time, even if the earlier ones followed the standard process. BEHAVIOR MATCHES allows us to do so.

```
SELECT
    case_id,
    event_name
FROM THIS_PROCESS
WHERE
    BEHAVIOR
        (event_name = 'Receive Customer Order') AS process_event_1,
        (event_name = 'Receive Payment') AS process_event_2,
        (event_name = 'Ship Goods Standard'
            AND end_time > DATE '2024-08-01')
            AS process_event_3
    MATCHES (^ process_event_1 -> process_event_2 -> process_event_3 $)
```

In this query, the BEHAVIOR clause defines three behaviors. Each behavior acts as a filter which removes cases or events failing to match the criteria. Each behavior must be assigned an alias using the AS keyword. In this example:

- The behavior process_event_1 matches events where event_name is 'Receive Customer Order'.
- The behavior process_event_2 matches events where event_name is 'Receive Payment'.
- The behavior process_event_3 matches events where event_name is 'Ship Goods Standard' and the associated timestamp is later than 01/08/2024, 00:00.

In the case of BEHAVIOR MATCHES, the MATCHES clause excludes cases failing to satisfy the behavior criteria **as well as** cases failing to match the matching pattern.

The behaviors process_event_1 and process_event_2 from the second query correspond with the functionality of the first query. Specifically, the first event must have an event_name of 'Receive Customer Order' and must be immediately followed by an event with an event_name of 'Receive Payment'. However, process_event_3 differs. In matching cases, the third event in the chain must have not only an event_name of 'Ship Goods Standard' but also an end_time later than 01/08/2024, 00:00.

> → Tip
>
> As Example 2 demonstrates, it's also possible to include case-level attributes in a behavior.

Example 1

Assume the following data (`THIS_PROCESS`):

case_id	City	event_name	Payment Amount	end_time
00001	Boston	Receive Customer Order	null	28/07/2024, 13:14
		Receive Payment	75.66	28/07/2024, 15:19
		Receive Payment	135	29/07/2024, 13:18
		Ship Goods Standard	null	30/07/2024, 10:44
00002	New York	Receive Customer Order	null	29/07/2024, 09:01
		Receive Payment	68.32	31/07/2024, 15:19
		Ship Goods Standard	null	02/08/2024, 15:00
00003	New York	Receive Customer Order	null	30/07/2024, 10:43
		Receive Payment	101	31/07/2024, 09:55
		Receive Payment	39.60	01/08/2024, 11:04
		Ship Goods Standard	null	02/08/2024, 14:10
		Ship Goods Standard	null	02/08/2024, 18:02
00004	San Francisco	Receive Customer Order	null	31/07/2024, 14:15
		Receive Payment	45.67	01/08/2024, 08:35
		Ship Goods Standard	null	02/08/2024, 10:10
00005	Miami	Receive Customer Order	null	31/07/2024, 16:54
		Receive Payment	74.29	31/07/2024, 17:08
		Ship Goods Express	null	01/08/2024, 08:55
		Receive Payment	159	01/08/2024, 11:07

The following query matches cases where a small payment was followed at some point by a larger payment.

```
SELECT case_id, event_name, "Payment Amount"
```

```
FROM THIS_PROCESS
WHERE BEHAVIOR
    (event_name = 'Receive Payment' AND "Payment Amount" < 100) AS
changed_small_amount,
    (event_name = 'Receive Payment' AND "Payment Amount" >= 100) AS
changed_large_amount
MATCHES (changed_small_amount ~> changed_large_amount)
```

Output:

case_id	event_name	Payment Amount
00001	Receive Customer Order	null
	Receive Payment	75.66
	Receive Payment	135
	Ship Goods Standard	null
00005	Receive Customer Order	null
	Receive Payment	74.29
	Ship Goods Express	null
	Receive Payment	159

Example 2

This example extends the example begun in Comparison with MATCHES [page 88]. In that example, we defined a query that finds cases following the standard sales process, excluding cases that shipped before a specific time (even if they followed the standard process).

```
SELECT
    case_id,
    event_name
FROM THIS_PROCESS
WHERE
    BEHAVIOR
        (event_name = 'Receive Customer Order') AS process_event_1,
        (event_name = 'Receive Payment') AS process_event_2,
        (event_name = 'Ship Goods Standard'
            AND end_time > DATE '2024-08-01')
            AS process_event_3
    MATCHES (^ process_event_1 -> process_event_2 -> process_event_3 $)
```

Let's extend this example, limiting our search to orders from a specific city, which is a case-level attribute.

```
SELECT
    case_id,
    City,
    event_name,
    end_time
FROM THIS_PROCESS
WHERE
    BEHAVIOR
        (event_name = 'Receive Customer Order'
            AND "City" = 'New York') AS process_event_1,
```

```
        (event_name = 'Receive Payment') AS process_event_2,
        (event_name = 'Ship Goods Standard'
            AND end_time > DATE '2024-08-01')
            AS process_event_3
    MATCHES (^ process_event_1 -> process_event_2 -> process_event_3 $)
```

The behavior still finds orders following the standard process that were shipped after a certain date. However, `process_event_1`. In that has been amended so that the result set is further restricted. The first event in the chain must now match orders that were dispatched to New York. This criterion implicitly applies to all other events in the same chain, since "City" is a case-level attribute. As such, the condition requires adding to only one behavior.

Output:

case_id	City	event_name	end_time
00002	New York	Receive Customer Order	29/07/2024, 09:01
		Receive Payment	31/07/2024, 15:19
		Ship Goods Standard	02/08/2024, 15:00

Related Information

Matching Expressions [page 80]

1.6 Functions

Use functions in SIGNAL to carry out specific tasks and calculate values.

The following categories of function are available:

Aggregate Functions [page 92]
: Use aggregate functions to reduce collections of values down to a single value.

Conditional Functions [page 146]
: Use conditional functions to define which operations to execute or which value to return based on a condition.

Conversion Functions [page 154]
: Use conversion functions to convert values from one data type to another.

Date Functions [page 165]
: Use date functions to execute operations on date and time values.

Mathematical Functions [page 182]
: Use these functions to perform a range of mathematical operations.

String Functions [page 195]
: Use string functions to manipulate strings in queries.

Window Functions [page 220]

Window functions perform calculations on a window, which is a subset of rows that are defined as being related.

BUCKET Function [page 285]
Learn about the BUCKET() function in SIGNAL, the process mining query language of SAP Signavio Process Intelligence.

1.6.1 Aggregate Functions

Use aggregate functions to reduce collections of values down to a single value.

Linear Regression Functions [page 93]
With the linear regression functions, you can calculate the relationship between two variables using the least squares regression method, which is a standard approach for calculating a linear relationship. The least squares regression line, also called the line of best fit, can be visualized as a straight line drawn through a set of data points that represents the relationship between them.

AVG [page 96]
Calculates the average of a collection of numeric values. `NULL` values are ignored

BOOL_AND [page 98]
Returns true if the supplied expression evaluates to true for all input rows, otherwise it returns false.

BOOL_OR [page 101]
Returns true if the supplied expression evaluates to true for any input row, otherwise it returns false.

COUNT [page 104]
Counts the number of values in a specified column. `NULL` values aren't counted.

COUNT (DISTINCT) [page 106]
Counts the number of distinct values in a specified column. If `NULL` values are present, they are excluded.

FIRST [page 109]
Returns the first element from a collection of values.

LAST [page 113]
Returns the last element from a collection of values.

MAX [page 116]
Finds the maximum value in a collection of values.

MEDIAN [page 119]
A shortcut for the `PERCENTILE_CONT` function with a fixed percentile of 0.5, that is the middle number.

MIN [page 120]
Finds the minimum value in a collection of values.

PERCENTILE_CONT [page 123]
Returns a percentile value based on the input column's continuous distribution. If no value lies at the percentile, the linear interpolation between the closest two values is returned.

PERCENTILE_DISC [page 125]
Returns a percentile value based on the discrete distribution of values in a column. The returned value has the smallest distance to the given percentile.

STDDEV [page 127]

Calculates the standard deviation for a collection of values.

SUM [page 129]
Calculates the sum of all values in a collection of numeric values. `NULL` values are ignored.

TRIMMED_AVG [page 132]
Calculates the average of a group of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and calculating the average.

TRIMMED_STDDEV [page 136]
Calculates the population standard deviation of a group of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and calculating the average.

TRIMMED_VARIANCE [page 139]
Calculates the population variance for a collection of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and performing the calculation.

VARIANCE [page 143]
Calculates the population variance for a collection of values. `NULL` values are ignored.

1.6.1.1 Linear Regression Functions

With the linear regression functions, you can calculate the relationship between two variables using the least squares regression method, which is a standard approach for calculating a linear relationship. The least squares regression line, also called the line of best fit, can be visualized as a straight line drawn through a set of data points that represents the relationship between them.

For the regression line calculation, you need two parameters, slope and intercept. The slope and intercept show how two variables are related according to an average rate of change, which works well with scatter plots because scatter plots show two variables. The regression line is plotted on a scatter plot of the same data to show the general data trend.

The purpose of the linear regression function is to find the relationship between the explanatory variable, X, and the dependent variable, Y. It predicts the value of Y when the value of X is known.

Where to Use Linear Regression

- The primary purpose of the linear regression function is to facilitate the plotting of regression lines in *Correlation* widget.
- You can also use the linear regression function within custom SQL statements to calculate trend or regression values. This is useful for making predictions that extend beyond the provided data. For example, suppose that order value is plotted against quantity and you want to know the value of an order size that is 10 times the maximum order size within your dataset.

Linear Regression Formula and Functions

A linear regression line has an equation of the form: `Y = a + bX`

`X` is the explanatory variable and `Y` is the dependent variable.

`a` = intercept (the value of Y when X = 0) and `b` = slope.

Following are the two functions implemented to support the linear regression:

- regr_slope(Y, X) function: slope of the least-squares-fit linear equation determined by the (X, Y) pairs
- regr_intercept(Y, X) function: Y-intercept of the least-squares-fit linear equation determined by the (X, Y) pairs

The slope indicates the steepness of the regression line, whereas the intercept indicates its intersection with the Y axis. The Y-intercept is the point at which the graph intersects the Y-axis. Based on the slope of each X-axis unit, the subsequent point on the Y-axis is determined.

Syntax

```
REGR_INTERCEPT(<yAttribute>, <xAttribute>)
REGR_SLOPE(<yAttribute>, <xAttribute>)
```

Parameter	Description	Valid Types
yAttribute	Y is the dependent variable and is plotted along the Y axis of a scatterplot.	Number
xAttribute	X is the explanatory variable and is plotted along the X axis of a scatterplot.	Number

Returns: A Number, the intercept of the univariate linear regression line determined by the (X, Y) pairs.

Example 1

The following query correlates the order amount with order quantity and return values for intercept and slope.

```
SELECT
    REGR_INTERCEPT("Order Amount" , "Order Quantity") as "Intercept",
    REGR_SLOPE("Order Amount" , "Order Quantity") as "Slope"
FROM THIS_PROCESS
```

The query returns the following result.

Intercept	Slope
-2.82	30.372

Following is the scatter plot, generated using *Correlation* widget in a process. It shows the scatter plot and regression line for order amount plotted against order quantity.

Example 2

The following query calculates the order value along the regression line relative to an order size that isn't included in your data set.

```
SELECT 300 as "Quantity","intercept" + 300 * "slope" as "Amount"
FROM
(
  SELECT
    REGR_INTERCEPT("Order Amount" , "Order Quantity") as "intercept",
    REGR_SLOPE("Order Amount" , "Order Quantity") as "slope"
  FROM THIS_PROCESS
) as sub
```

The query returns the following result.

Quantity	Amount
300	9108.917

Example 3

The following query calculates the order value along the regression line relative to an order size that isn't included in your data set and grouped by City.

```
SELECT "City",
  300 as "Quantity",
  "intercept" + 300 * "slope" as "Amount"
FROM
(
  SELECT "City",
    REGR_INTERCEPT("Order Amount" , "Order Quantity") as "intercept",
    REGR_SLOPE("Order Amount" , "Order Quantity") as "slope"
  FROM THIS_PROCESS
  GROUP BY 1
```

```
) as sub
```

The query returns the following result.

City	Quantity	Amount
Miami	300	9291.066
Washington	300	8122.38
Boston	300	8422.01
New York	300	10248.269
San Francisco	300	8154.724
Houston	300	9562.717

1.6.1.2 AVG

Calculates the average of a collection of numeric values. `NULL` values are ignored

Syntax

```
AVG(<expression>)
```

Parameter	Description	Valid Types
`expression`	The column of values to be averaged.	Number, Timestamp, Duration

Returns: The average as a Number, Timestamp, or Duration. The return type matches the type of the `expression` parameter.

Use as a Window Function

`AVG` can also be used as a window function:

```
AVG(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
AVG(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount
02302	01/01/2020, 00:48	404.54
01553	01/01/2020, 03:00	482.85
00681	01/01/2020, 04:36	1030.71
01153	01/01/2020, 10:31	700.68
01524	02/01/2020, 00:54	706.96

This query returns the average order amount of all cases.

```
SELECT AVG("Order Amount") AS "Average Order Amount"
FROM THIS_PROCESS
```

Output:

Average Order Amount
665.148

Example 2

This example demonstrates a windowed average. Assuming the same data set as the previous example, the following query calculates a running average at the point of each sale. It does so by defining for each row a window between the first row and the current row, calculating the average order amount within that window.

```
SELECT case_id,
    "Order Date",
    "Order Amount",
    AVG("Order Amount")
        OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS "AVG All Orders to Date"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Order Amount	AVG All Orders to Date
02302	01/01/2020, 00:48	404.54	404.54
01553	01/01/2020, 03:00	482.85	443.695
00681	01/01/2020, 04:36	1030.71	639.367
01153	01/01/2020, 10:31	700.68	654.695
01524	02/01/2020, 00:54	706.96	665.148

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT AVG("Order Amount") FILTER (WHERE "Order Amount" < 500) AS "Average"
FROM THIS_PROCESS
```

Average
443.70

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.1.3 BOOL_AND

Returns true if the supplied expression evaluates to true for all input rows, otherwise it returns false.

Syntax

```
BOOL_AND(<expression>)
```

Parameter	Description	Valid Types
`expression`	An expression applied to a collection of rows.	Boolean

Returns: A Boolean. Result is true if all input rows evaluate to true for the supplied expression, otherwise false.

Use as a Window Function

`BOOL_AND` can also be used as a window function:

```
BOOL_AND(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
BOOL_AND(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount	Customer Type
03015	02/01/2020, 04:09	269.4	Standard
00854	02/01/2020, 04:29	374.01	Premium
02060	02/01/2020, 06:48	84.33	Standard
00010	02/01/2020, 09:02	319.18	Standard
03040	02/01/2020, 12:27	20	Premium
01665	02/01/2020, 16:44	492.27	Premium
01951	02/01/2020, 21:32	521.2	Standard

The following query determines whether all cases have order amounts over 30.

```
SELECT BOOL_AND("Order Amount" > 30) AS "All Over 30?"
FROM THIS_PROCESS
```

The answer is false because one of the amounts is lower than 30.

All Over 30?
FALSE

Example 2

Assuming the same data set as the previous example, the following query demonstrates the windowed version of `BOOL_AND`. It considers two cases at a time: the current case and its immediate predecessor. If "`Customer Type`" is listed as "`Premium`" in both, the function returns true.

```
SELECT case_id,
    "Order Date",
    "Customer Type",
    BOOL_AND("Customer Type" = 'Premium')
        OVER (ROWS BETWEEN 1 PRECEDING AND CURRENT ROW)
        AS "Both Premium?"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Customer Type	Both Premium?
03015	02/01/2020, 04:09	Standard	FALSE
00854	02/01/2020, 04:29	Premium	FALSE
02060	02/01/2020, 06:48	Standard	FALSE
00010	02/01/2020, 09:02	Standard	FALSE
03040	02/01/2020, 12:27	Premium	FALSE
01665	02/01/2020, 16:44	Premium	TRUE
01951	02/01/2020, 21:32	Standard	FALSE

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT BOOL_AND("Order Amount" > 50) FILTER (WHERE "Customer Type" = 'Standard')
AS "All Standard Over 50?"
FROM THIS_PROCESS
```

All Standard Over 50?
TRUE

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.1.4 BOOL_OR

Returns true if the supplied expression evaluates to true for any input row, otherwise it returns false.

Syntax

```
BOOL_OR(<expression>)
```

Parameter	Description	Valid Types
`expression`	An expression applied to a collection of rows.	Boolean

Returns: A Boolean. Result is true if any input row evaluates to true for the supplied expression, otherwise false.

Use as a Window Function

`BOOL_OR` can also be used as a window function:

```
BOOL_OR(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
BOOL_OR(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount	Customer Type
03015	02/01/2020, 04:09	269.4	Standard
00854	02/01/2020, 04:29	374.01	Premium
02060	02/01/2020, 06:48	84.33	Standard
00010	02/01/2020, 09:02	319.18	Standard
03040	02/01/2020, 12:27	20	Premium
01665	02/01/2020, 16:44	492.27	Premium
01951	02/01/2020, 21:32	521.2	Standard

The following query determines whether any cases have order amounts over 30.

```
SELECT BOOL_OR("Order Amount" > 30) AS "Any Over 30?"
FROM THIS_PROCESS
```

The answer is true since at least one of the amounts is greater than 30:

Any Over 30?
TRUE

Example 2

Assuming the same data set as the previous example, the following query demonstrates the windowed version of `BOOL_OR`. It considers two cases at a time: the current case and its immediate predecessor. If "`Customer Type`" is listed as "`Premium`" in either, the function returns true.

```
SELECT case_id,
    "Order Date",
    "Customer Type",
    BOOL_OR("Customer Type" = 'Premium')
        OVER (ROWS BETWEEN 1 PRECEDING AND CURRENT ROW)
        AS "Either Premium?"
```

```
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Either Premium?	Customer Type
03015	02/01/2020, 04:09	FALSE	Standard
00854	02/01/2020, 04:29	TRUE	Premium
02060	02/01/2020, 06:48	TRUE	Standard
00010	02/01/2020, 09:02	FALSE	Standard
03040	02/01/2020, 12:27	TRUE	Premium
01665	02/01/2020, 16:44	TRUE	Premium
01951	02/01/2020, 21:32	TRUE	Standard

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT BOOL_OR("Order Amount" < 20) FILTER (WHERE "Customer Type" = 'Premium')
AS "Any Premium Under 20?"
FROM THIS_PROCESS
```

Any Premium Under 20?
FALSE

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.1.5 COUNT

Counts the number of values in a specified column. `NULL` values aren't counted.

Syntax

```
COUNT(<expression>)
```

Parameter	Description	Valid Types
`expression`	The column whose values are to be counted.	Any

Returns: The number of values in the provided collection of values.

Use as a Window Function

`COUNT` can also be used as a window function:

```
COUNT(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
COUNT(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Customer Type
00681	01/01/2020, 04:36	Standard
01153	01/01/2020, 10:31	Standard
01524	02/01/2020, 00:54	Standard
03055	02/01/2020, 03:09	Premium
03015	02/01/2020, 04:09	Standard
00854	02/01/2020, 04:29	Premium
02060	02/01/2020, 06:48	Standard
00010	02/01/2020, 09:02	Standard
03040	02/01/2020, 12:27	Premium
01665	02/01/2020, 16:44	Premium

The following query counts the number of rows.

```
SELECT COUNT(case_id) AS "Num. of Orders"
FROM THIS_PROCESS
```

Output:

Num. of Orders
10

Example 2

This example demonstrates a windowed count. Assuming the same data set as the previous example, the following query calculates how many premium customers placed orders between the time of each order and the previous twelve hours.

```
SELECT case_id,
    "Order Date",
    "Customer Type",
    COUNT(case_id)
        OVER (ORDER BY "Order Date" RANGE BETWEEN DURATION '12hours' PRECEDING
AND CURRENT ROW)
        AS "Premium Orders Over 12 Hrs"
FROM THIS_PROCESS
WHERE "Customer Type" = 'Premium'
```

The first three premium orders arrive within less than twelve hours, hence the count climbs from 1 to 3. By the time of the fourth premium order, the first two orders have dropped out of the twelve-hour window and so the count drops to 2.

case_id	Order Date	Customer Type	Premium Orders Over 12 Hrs
03055	02/01/2020, 03:09	Premium	1
00854	02/01/2020, 04:29	Premium	2
03040	02/01/2020, 12:27	Premium	3
01665	02/01/2020, 16:44	Premium	2

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT COUNT(case_id) FILTER (WHERE "Customer Type" = 'Premium') AS "Count"
FROM THIS_PROCESS
```

Count
4

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.1.6 COUNT (DISTINCT)

Counts the number of distinct values in a specified column. If `NULL` values are present, they are excluded.

In the case of counting event-level columns, since event-level columns are lists, this function counts the number of distinct lists.

Syntax

```
COUNT(DISTINCT <expression>)
```

Parameter	Description	Valid Types
`expression`	The column whose distinct values are to be counted.	Number, Text, Timestamp, List (i.e. event-level column)

Returns: The number of distinct values in the provided collection of values.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
COUNT(DISTINCT <expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1 (Case-level)

Assuming the following process data (`THIS_PROCESS`):

case_id	City	event_name
00001	Houston	Receive Customer Order
		Change Order Quantity
		Receive Payment
		Ship Goods Express
		Receive Delivery Confirmation
00002	San Francisco	Receive Customer Order
		Receive Payment
		Send items to Printing
		Order Canceled
00003	Houston	Receive Customer Order
		Change Order Quantity
		Receive Payment
		Ship Goods Standard
		Receive Delivery Confirmation

case_id	City	event_name
00004	San Francisco	Receive Customer Order
		Change Order Quantity
		Receive Payment
		Ship Goods Express
		Receive Delivery Confirmation
00005	Miami	Receive Customer Order
		Change Order Quantity
		Change Order Quantity
		Receive Payment
		Send items to Printing
		Items Printed
		Receive items from Printing
		Ship Goods Express
		Receive Delivery Confirmation

This query returns the number of distinct cities.

```
SELECT COUNT(DISTINCT city) AS Count
FROM THIS_PROCESS
```

Result:

Count
3

Example 2 (Event-level)

This query counts the number of distinct event sequences per case. Each distinct sequence represents a process variant, therefore this query returns the number of process variants.

```
SELECT COUNT(DISTINCT event_name) AS Variants
FROM THIS_PROCESS
```

Result:

Variants
4

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT COUNT(DISTINCT city) FILTER (WHERE city <> 'Miami') AS Cities
FROM THIS_PROCESS
```

Result:

Cities
2

Related Information

FILTER Clause [page 20]

1.6.1.7 FIRST

Returns the first element from a collection of values.

Syntax

```
FIRST(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection of values from which the first is chosen.	Number, Timestamp, Duration, Text, Boolean

Returns: The first value as a Number, Timestamp, Duration, Text or Boolean. The return type matches the type of the `expression` parameter.

Use as a Window Function

`FIRST` can also be used as a window function:

```
FIRST(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
```

```
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
FIRST(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	event_name
00606	Receive Customer Order
	Receive Payment
	Receive Payment
	Send items to Printing
	Order Canceled
02178	Receive Customer Order
	Ship Goods Standard
	Receive Payment
	Receive Delivery Confirmation
01200	Receive Customer Order
	Receive Payment
	Ship Goods Express
	Ship Goods Express
	Receive Delivery Confirmation
	Receive Delivery Confirmation

case_id	event_name
01750	Receive Customer Order
	Change Order Quantity
	Receive Payment
	Ship Goods Standard
	Receive Delivery Confirmation

The following query returns the first event that occurs in each case after receipt of an order (which is always the first event). It does so by filtering out those receipt events and then taking the first event from what remains.

```
SELECT
    case_id,
    (SELECT FIRST(event_name)) AS "First Action After Receipt"
FROM THIS_PROCESS
FILTER EVENTS WHERE event_name <> 'Receive Customer Order'
```

Output:

case_id	First Action After Receipt
00606	Receive Payment
02178	Ship Goods Standard
01200	Receive Payment
01750	Change Order Quantity

Example 2

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. The following query achieves the same goal as the previous example by using the`FILTER` clause in a subquery instead of using `FILTER EVENTS`:

```
SELECT case_id,
    (SELECT FIRST(event_name) FILTER (WHERE event_name <> 'Receive Customer
Order')) AS "First Action After Receipt"
FROM THIS_PROCESS
```

Result:

case_id	First Action After Receipt
00606	Receive Payment
01200	Receive Payment
01750	Change Order Quantity
02178	Ship Goods Standard

Example 3

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Customer Type
00681	01/01/2020, 04:36	Standard
01153	01/01/2020, 10:31	Standard
01524	02/01/2020, 00:54	Standard
03055	02/01/2020, 03:09	Premium
03015	02/01/2020, 04:09	Standard
00854	02/01/2020, 04:29	Premium
02060	02/01/2020, 06:48	Standard
00010	02/01/2020, 09:02	Standard
03040	02/01/2020, 12:27	Premium
01665	02/01/2020, 16:44	Premium

The following query finds the date of the second order before the current one. It does so by using a window comprising the two previous rows and taking the order date of the first row in the window.

```
SELECT case_id,
    "Order Date",
    FIRST("Order Date")
        OVER (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
        AS "Date Two Orders Prior"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Date Two Orders Prior
1553	01/01/2020, 03:00	01/01/2020, 03:00
681	01/01/2020, 04:36	01/01/2020, 03:00
1153	01/01/2020, 10:31	01/01/2020, 03:00
1524	02/01/2020, 00:54	01/01/2020, 04:36
3055	02/01/2020, 03:09	01/01/2020, 10:31
3015	02/01/2020, 04:09	02/01/2020, 00:54
854	02/01/2020, 04:29	02/01/2020, 03:09

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.1.8 LAST

Returns the last element from a collection of values.

Syntax

```
LAST(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection of values from which the last is chosen.	Number, Timestamp, Duration, Text, Boolean

Returns: The last value as a Number, Timestamp, Duration, Text or Boolean. The return type matches the type of the `expression` parameter.

Use as a Window Function

LAST can also be used as a window function:

```
LAST(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
LAST(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Status	event_name
00606	Canceled	Receive Customer Order
		Receive Payment
		Receive Payment
		Send items to Printing
		Order Canceled
02178	Delivered	Receive Customer Order
		Ship Goods Standard
		Receive Payment
		Receive Delivery Confirmation
01200	Delivered	Receive Customer Order
		Receive Payment
		Ship Goods Express
		Ship Goods Express
		Receive Delivery Confirmation
01750	Delivered	Receive Customer Order
		Change Order Quantity
		Receive Payment
		Ship Goods Standard
		Receive Delivery Confirmation

The following query returns the last event that occurs in each case.

```
SELECT
    case_id,
    (SELECT LAST(event_name)) AS "Last Action"
FROM THIS_PROCESS
```

Output:

case_id	Last Action
00606	Order Canceled
02178	Receive Delivery Confirmation
01200	Receive Delivery Confirmation

case_id	Last Action
01750	Receive Delivery Confirmation

Example 2

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. The following query finds the last event in all delivered orders but filters the final delivery confirmation event, resulting in it returning the penultimate event:

```
SELECT case_id,
    (SELECT LAST(event_name) FILTER (WHERE event_name <> 'Receive Delivery
Confirmation')) AS "Last Action Before Delivery"
FROM THIS_PROCESS
WHERE "Order Status" = 'Delivered'
```

Result:

case_id	Last Action Before Delivery
01200	Ship Goods Express
01750	Ship Goods Standard
02178	Receive Payment

Example 3

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Customer Type
00681	01/01/2020, 04:36	Standard
01153	01/01/2020, 10:31	Standard
01524	02/01/2020, 00:54	Standard
03055	02/01/2020, 03:09	Premium
03015	02/01/2020, 04:09	Standard
00854	02/01/2020, 04:29	Premium
02060	02/01/2020, 06:48	Standard
00010	02/01/2020, 09:02	Standard
03040	02/01/2020, 12:27	Premium
01665	02/01/2020, 16:44	Premium

The following query finds the date of the second order after the current one. It does so by using a window comprising the two following rows and taking the order date of the last row in the window.

```
SELECT case_id,
    "Order Date",
    LAST("Order Date")
        OVER (ROWS BETWEEN CURRENT ROW AND 2 FOLLOWING)
        AS "Date Two Orders Later"
ORDER BY 2
```

Output:

case_id	Order Date	Date Two Orders Later
01553	01/01/2020, 03:00	01/01/2020, 10:31
00681	01/01/2020, 04:36	02/01/2020, 00:54
01153	01/01/2020, 10:31	02/01/2020, 03:09
01524	02/01/2020, 00:54	02/01/2020, 04:09
03055	02/01/2020, 03:09	02/01/2020, 04:29
03015	02/01/2020, 04:09	02/01/2020, 04:29
00854	02/01/2020, 04:29	02/01/2020, 04:29

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.1.9 MAX

Finds the maximum value in a collection of values.

Syntax

```
MAX(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection from which the maximum value is chosen.	Number, Timestamp, Duration

Returns: The maximum value as a Number, Timestamp or Duration. The return type matches the type of the `expression` parameter.

Use as a Window Function

MAX can also be used as a window function:

```
MAX(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
MAX(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (THIS_PROCESS):

case_id	Order Date	Order Amount
02302	01/01/2020, 00:48	404.54
01553	01/01/2020, 03:00	482.85
00681	01/01/2020, 04:36	1030.71
01153	01/01/2020, 10:31	700.68
01524	02/01/2020, 00:54	706.96

This query returns the maximum order amount found in all cases.

```
SELECT MAX("Order Amount") AS "Maximum Order Amount"
FROM THIS_PROCESS
```

Output:

Maximum Order Amount
1030.71

Example 2

This example demonstrates a windowed maximum. Assuming the same data set as the previous example, the following query calculates the maximum order amount to date when each order was placed. It does so by defining for each row a window between the first row and the current row, finding the maximum order amount within that window.

```
SELECT case_id,
    "Order Date",
    "Order Amount",
    MAX("Order Amount")
        OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS "MAX All Orders to Date"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Order Amount	MAX All Orders to Date
02302	01/01/2020, 00:48	404.54	404.54
01553	01/01/2020, 03:00	482.85	482.85
00681	01/01/2020, 04:36	1030.71	1030.71
01153	01/01/2020, 10:31	700.68	1030.71
01524	02/01/2020, 00:54	706.96	1030.71

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT MAX("Order Amount") FILTER (WHERE "Order Amount" < 1000) AS "Maximum
Under 1000"
FROM THIS_PROCESS
```

Maximum Under 1000
706.96

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.1.10 MEDIAN

A shortcut for the `PERCENTILE_CONT` function with a fixed percentile of 0.5, that is the middle number.

Syntax

```
MEDIAN(<expression>)
```

Parameter	Description	Valid Types
`expression`	The column of which you want to determine the value that separates the lower from the upper half.	Number, Timestamp, Duration

Returns: The median value as a Number, Timestamp or Duration. The return type matches the type of the `column_name` parameter.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
MEDIAN(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

This query calculates the median of a set of numerical values.

	A
1	65
2	72
3	81
4	95
5	112
6	128

```
SELECT MEDIAN(A)
FROM THIS_PROCESS
```

The query determines in a first step the value position = 3. Because the sorted list has an even number of items, this query calculates the arithmetic mean, that is (81 + 95)/2 = 88.

Example 2

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount
02302	01/01/2020, 00:48	404.54
01553	01/01/2020, 03:00	482.85
00681	01/01/2020, 04:36	1030.71
01153	01/01/2020, 10:31	700.68
01524	02/01/2020, 00:54	706.96

This query calculates the median order amount of all orders under 1000. It does so by using the `FILTER` clause to remove orders totaling 1000 or over.

```
SELECT MEDIAN("Order Amount") FILTER (WHERE "Order Amount" < 1000) AS "Median
Under 1000"
FROM THIS_PROCESS
```

Median Under 1000
591.77

Related Information

FILTER Clause [page 20]

1.6.1.11 MIN

Finds the minimum value in a collection of values.

Syntax

```
MIN(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection from which the minimum value is chosen.	Number, Timestamp, Duration

Returns: The minimum value as a Number, Timestamp or Duration. The return type matches the type of the `expression` parameter.

Use as a Window Function

`MIN` can also be used as a window function:

```
MIN(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
MIN(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount
02302	01/01/2020, 00:48	404.54
01553	01/01/2020, 03:00	482.85
00681	01/01/2020, 04:36	1030.71
01153	01/01/2020, 10:31	700.68
01524	02/01/2020, 00:54	706.96

This query returns the minimum order amount found in all cases.

```
SELECT MIN("Order Amount") AS "Minimum Order Amount"
```

```
FROM THIS_PROCESS
```

Output:

Minimum Order Amount
404.54

Example 2

This example demonstrates a windowed minimum. Assuming the same data set as the previous example, the following query calculates the minimum order amount to date at the point when each order was placed. It does so by defining for each row a window between the first row and the current row, finding the minimum order amount within that window.

```
SELECT case_id,
    "Order Date",
    "Order Amount",
    MIN("Order Amount")
        OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS "MIN All Orders to Date"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Order Amount	MIN All Orders to Date
02302	01/01/2020, 00:48	404.54	404.54
01553	01/01/2020, 03:00	482.85	404.54
00681	01/01/2020, 04:36	1030.71	404.54
01153	01/01/2020, 10:31	700.68	404.54
01524	02/01/2020, 00:54	706.96	404.54

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT MIN("Order Amount") FILTER (WHERE "Order Amount" > 500) AS "Minimum over
500"
FROM THIS_PROCESS
```

Minimum over 500
700.68

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.1.12 PERCENTILE_CONT

Returns a percentile value based on the input column's continuous distribution. If no value lies at the percentile, the linear interpolation between the closest two values is returned.

`NULL` values are ignored. If all values are `NULL`, this function returns `NULL`.

Syntax

```
PERCENTILE_CONT(<p>) WITHIN GROUP(ORDER BY <expression>)
```

Parameter	Description	Valid Types
p	The percentile of the value that you want to find. For example, specify 0.8 if you want to find the 80th percentile. Must be between 0.0 and 1.0	Number (literal)
`expression`	The column for which you want to determine the value that separates the lower from the upper percentile for the given percentile rank.	Number, Timestamp, Duration

Returns: The calculated continuous percentile as a Number, Timestamp or Duration. The return type matches the type of the `expression` parameter.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
PERCENTILE_CONT(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

	A
1	65
2	72
3	81
4	95
5	112
6	128

```
SELECT
PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY A)
FROM THIS_PROCESS
```

This query determines in a first step the value position = 1.5. Because it's a fraction, the result value is the average of position 1 and 2 = 68.5. Values below 68.5 are in the lower percentile, values above 68.5 are in the upper percentile.

```
SELECT
PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY A)
FROM THIS_PROCESS
```

For p = 0.75, the value position = 4.5. The result is the average of position 4 and 5, that is 103.5.

Example 2

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Amount
00008	162.58
00009	165.44
00004	268.34
00002	270.04
00010	319.18
00005	327.94
00003	469.90
00007	521.17
00006	599.07
00001	944.42

This query calculates the 80th percentile of the order amount, but only on orders totaling less than 500.

```
SELECT PERCENTILE_CONT(0.8)
```

```
    WITHIN GROUP (ORDER BY "Order Amount")
    FILTER (WHERE "Order Amount" < 500) AS Percentile
FROM THIS_PROCESS
```

Percentile
342.44

Related Information

FILTER Clause [page 20]
PERCENTILE_DISC [page 125]

1.6.1.13 PERCENTILE_DISC

Returns a percentile value based on the discrete distribution of values in a column. The returned value has the smallest distance to the given percentile.

`NULL` values are ignored. If all values are `NULL`, this function returns `NULL`.

Syntax

```
PERCENTILE_DISC(<p>) WITHIN GROUP(ORDER BY <expression>)
```

Parameter	Description	Valid Types
p	The percentile of the value that you want to find. For example, specify 0.8 if you want to find the 80th percentile. Must be between 0.0 and 1.0	Number (literal)
`expression`	The column for which you want to determine the value that separates the lower from the upper percentile for the given percentile rank.	Number, Timestamp, Duration

Returns: The calculated discrete percentile. The return type matches the type of the `expression` parameter.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
PERCENTILE_DISC(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

	A
1	65
2	72
3	81
4	95
5	112
6	128

```
SELECT
PERCENTILE_DISC(0.25) WITHIN GROUP (ORDER BY A)
FROM THIS_PROCESS
```

This query determines the value position = 1.5. Because it's a fraction, the value at the next higher position is returned, that is 72.

Example 2

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Amount
00008	162.58
00009	165.44
00004	268.34
00002	270.04
00010	319.18
00005	327.94
00003	469.90

case_id	Order Amount
00007	521.17
00006	599.07
00001	944.42

This query calculates the 80th percentile of the order amount, but only on orders totaling less than 500.

```
SELECT PERCENTILE_DISC(0.8)
    WITHIN GROUP (ORDER BY "Order Amount")
    FILTER (WHERE "Order Amount" < 500) AS Percentile
FROM THIS_PROCESS
```

Percentile
342.52

Related Information

FILTER Clause [page 20]
PERCENTILE_CONT [page 123]

1.6.1.14 STDDEV

Calculates the standard deviation for a collection of values.

The standard deviation describes the average deviation of all measured values from the mean value. A low standard deviation indicates that the values tend to be close to the mean value. A high standard deviation indicates that the values are spread out over a wide range.

Syntax

```
STDDEV(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection of values for which you want to determine the standard deviation.	Number, Duration

Returns: The standard deviation as a Number, Timestamp, or Duration. The return type matches the type of the `expression` parameter.

Use as a Window Function

STDDEV can also be used as a window function:

```
STDDEV(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
STDDEV(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

This query returns the standard deviation of all order amounts.

```
SELECT
    MIN("Order Amount") AS "Min",
    MAX("Order Amount") AS "Max",
    AVG("Order Amount") AS "Avg",
    STDDEV("Order Amount") AS "StdDev"
FROM THIS_PROCESS
```

Output:

Min	Max	Avg	StdDev
20	2,166.14	385.668	357.452

Example 2

This query shows how the cycle time of a process changes week-to-week. Specifically, it calculates both the average and the standard deviation of the cycle time on a weekly basis.

To smooth out short-term fluctuations in the data, it uses windowed versions of the average (AVG) and standard deviation (STDDEV) functions. These versions calculate a moving average and moving standard deviation for the weekly mean cycle time respectively.

In the nested query, `sq1`, all cases are clustered together by week. The mean cycle time within each cluster is calculated.

The outer query applies the windowed `AVG` and `STDDEV` functions to the weekly mean cycle time. In both cases, the window includes the three weeks preceding and the three weeks following the current week.

```
SELECT
    "Week",
    "Avg Cycle Time",
    AVG("Avg Cycle Time") OVER (ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING) as
mean,
    STDDEV("Avg Cycle Time") OVER (ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING) as
std
FROM
(
    SELECT
        DATE_TRUNC('WEEK', (SELECT LAST (END_TIME))) AS "Week",
        AVG((SELECT LAST(end_time) - FIRST(end_time))) AS "Avg Cycle Time"
    FROM THIS_PROCESS
    ORDER BY 1 ASC NULLS FIRST
    FILL timeseries('WEEK')
) as sq1
```

Week	Avg Cycle Time	mean	std
06/01/2020, 00:00	6d 14h	9d 20h	1d 23h
13/01/2020, 00:00	9d 20h	10d	1d 19h
20/01/2020, 00:00	11d 5h	10d	1d 15h
27/01/2020, 00:00	11d 16h	10d 4h	1d 13h
03/02/2020, 00:00	10d 16h	10d 18h	13h 40m
10/02/2020, 00:00	10d 3h	10d 20h	11h 12m
17/02/2020, 00:00	10d 23h	10d 18h	10h 45m
24/02/2020, 00:00	10d 18h	10d 17h	9h 18m

Related Information

Window Functions [page 220]

1.6.1.15 SUM

Calculates the sum of all values in a collection of numeric values. `NULL` values are ignored.

Syntax

```
SUM(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection of values to be summed.	Number, Duration

Returns: The sum as a Number or Duration. The return type matches the type of the `expression` parameter.

Use as a Window Function

`SUM` can also be used as a window function:

```
SUM(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
SUM(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount
02302	01/01/2020, 00:48	404.54
01553	01/01/2020, 03:00	482.85
00681	01/01/2020, 04:36	1030.71
01153	01/01/2020, 10:31	700.68
01524	02/01/2020, 00:54	706.96

This query returns the total order amount in all cases.

```
SELECT SUM("Order Amount") AS "Total Order Amount"
FROM THIS_PROCESS
```

Output:

Total Order Amount
3325.74

Example 2

This example demonstrates a running total. Assuming the same data set as the previous example, the following query calculates the running total at the point when each order was placed. It does so by defining for each row a window between the first row and the current row, finding the sum of all order amounts within that window.

```
SELECT case_id,
    "Order Date",
    "Order Amount",
    SUM("Order Amount")
        OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS "SUM All Orders to Date"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Order Amount	SUM All Orders to Date
02302	01/01/2020, 00:48	404.54	404.54
01553	01/01/2020, 03:00	482.85	887.39
00681	01/01/2020, 04:36	1030.71	1918.1
01153	01/01/2020, 10:31	700.68	2618.78
01524	02/01/2020, 00:54	706.96	3325.74

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT SUM("Order Amount") FILTER (WHERE "Order Amount" < 1000) AS "Sum"
FROM THIS_PROCESS
```

Sum
1407.64

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.1.16 TRIMMED_AVG

Calculates the average of a group of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and calculating the average.

A cutoff ranges from 0 to 1 and expresses the percentage of values to be trimmed before the calculation is performed. For example, if a group contains 10 values, setting the lower and upper cutoffs to 0.2 means that up to 20% of the lowest values and up to 20% of the highest values are trimmed. 20% of 10 is 2, so the 2 lowest and 2 highest values are trimmed, excluding 4 values in total from the calculation.

If a cutoff results in a non-integer number of values to be trimmed, the resulting number is rounded down to the next lower whole number. For a group of 10 values, a cutoff of 0.15 results in 1.5, in which case the result is rounded down to 1.

Syntax

```
TRIMMED_AVG(<expression>, <lowerCutoff>, <upperCutoff>)
```

Parameter	Description	Valid Types
`expression`	The group of values to be trimmed and averaged. `NULL` values are ignored.	Number, Timestamp, Duration
`lowerCutoff`	A number between 0 and 1. Specifies what percentage of the smallest values are trimmed before calculating the average. **Note** • 0 means that no values are trimmed. • 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number

Parameter	Description	Valid Types
`upperCutoff`	A number between 0 and 1. Specifies what percentage of the largest values are trimmed before calculating the average. **Note** - 0 means that no values are trimmed. - 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number

Returns: The trimmed average as a Number, Timestamp, or Duration. The return type matches the type of the `expression` parameter. If all values of `expression` are `NULL` or no values remain after trimming, then `NULL` is returned.

Use as a Window Function

This function can also be used as a window function:

```
TRIMMED_AVG(<expression>, <lowerCutoff>, <upperCutoff>) OVER (
    [ PARTITION BY <partitionExpression> [, <partitionExpression> ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression> ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
TRIMMED_AVG(<expression>, <lowerCutoff>, <upperCutoff>) FILTER (WHERE
<condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	20
00002	40
00003	40
00004	20
00005	5
00006	115
00007	null

The following query demonstrates the difference between calculating an average and a trimmed average.

```
SELECT
    AVG("Order Amount") AS "Average",
    TRIMMED_AVG("Order Amount", 0.2, 0.2) AS "Trimmed Average"
FROM THIS_PROCESS
```

- When calculating the **average**, all (non-`NULL`) values are included in the calculation.
- In this example, the **trimmed average** has upper and lower cutoffs of 0.2, meaning that up to 20% of the largest and smallest values are excluded from the calculation. There are 6 non-`NULL` values and 20% of 6 is 1.2. This result is therefore rounded down to the next lowest whole number, 1, and so the single highest and lowest values (115 and 5) are excluded.

Output:

Average	Trimmed Average
40	30

Example 2

Because this function is also available as a window function, it's possible to calculate the trimmed average of a window frame.

In the following query, the lower cutoff of 0.4 removes up to 40% of the smallest values in the specified window frame. That window frame includes the current row and the two preceding rows.

```
SELECT
    case_id,
    "Order Amount",
    TRIMMED_AVG("Order Amount", 0.4, 0.0)
        OVER (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
        AS "Trimmed Average - Window"
FROM THIS_PROCESS
ORDER BY 1
```

Output:

case_id	Order Amount	Trimmed Average - Window
00001	20	20
00002	40	30
00003	40	40
00004	20	40
00005	5	30
00006	115	67.5
00007	null	60

The trimmed average values are calculated as follows:

- For the first row (case_id = 00001), the window frame includes only one value (20). Based on the defined cutoffs, no value can be removed from the window frame, so the average is 20.
- For the second row (case_id = 00002), the window frame includes the values 20 and 40. Removing the lowest value would exceed the defined 40 percent cutoff limit, so no values are trimmed. Therefore, the average is 30 (40 + 20 / 2).
- For the third row (case_id = 00003), the window frame includes the values 20, 40, and 40. It's now possible to trim values because 40% of 3 is 1.2, which is rounded down to 1. That means the lowest value (20) is excluded from the calculation. The average of the remaining values (40 and 40) is 40.
- Calculations continue like this for all remaining rows in the data set. In the final row (case_id = 00007) `NULL` is ignored, which means the window contains only two values, neither of which are trimmed.

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT TRIMMED_AVG("Order Amount", 0.2, 1) FILTER (WHERE "Order Amount" < 100)
AS "Average"
FROM THIS_PROCESS
```

Average
443.70

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.1.17 TRIMMED_STDDEV

Calculates the population standard deviation of a group of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and calculating the average.

A cutoff ranges from 0 to 1 and expresses the percentage of values to be trimmed before the calculation is performed. For example, if a group contains 10 values, setting the lower and upper cutoffs to 0.2 means that up to 20% of the lowest values and up to 20% of the highest values are trimmed. 20% of 10 is 2, so the 2 lowest and 2 highest values are trimmed, excluding 4 values in total from the calculation.

If a cutoff results in a non-integer number of values to be trimmed, the resulting number is rounded down to the next lower whole number. For a group of 10 values, a cutoff of 0.15 results in 1.5, in which case the result is rounded down to 1.

Syntax

```
TRIMMED_STDDEV(<expression>, <lowerCutoff>, <upperCutoff>)
```

Parameter	Description	Valid Types
`expression`	The group of values to be trimmed and have their standard deviation calculated. `NULL` values are ignored.	Number, Duration
`lowerCutoff`	A number between 0 and 1. Specifies what percentage of the smallest values are trimmed before calculating the standard deviation. **Note** • 0 means that no values are trimmed. • 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number

Parameter	Description	Valid Types
`upperCutoff`	A number between 0 and 1. Specifies what percentage of the largest values are trimmed before calculating the standard deviation. **Note** • 0 means that no values are trimmed. • 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number

Returns: The standard deviation as a Number or Duration. The return type matches the type of the `expression` parameter. If all values of `expression` are `NULL` or no values remain after trimming, then `NULL` is returned.

Use as a Window Function

This function can also be used as a window function:

```
TRIMMED_STDDEV(<expression>, <lowerCutoff>, <upperCutoff>) OVER (
    [ PARTITION BY <partitionExpression> [, <partitionExpression> ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression> ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
TRIMMED_STDDEV(<expression>, <lowerCutoff>, <upperCutoff>) FILTER (WHERE
<condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	20
00002	40
00003	40
00004	20
00005	5
00006	115
00007	null

The following query demonstrates the difference between calculating a standard deviation and a trimmed standard deviation.

```
SELECT
    STDDEV("Order Amount") AS "Std. Dev",
    TRIMMED_STDDEV("Order Amount", 0.1, 0.2) AS "Trimmed Std. Dev"
FROM THIS_PROCESS
```

- When calculating the **standard deviation**, all (non-`NULL`) values are included in the calculation.
- In this example, the **trimmed standard deviation** has lower and upper cutoffs of 0.1 and 0.2 respectively.
  - Up to 10% of the smallest values are excluded from the calculation. There are 6 values and 10% of 6 is 0.6. This result is therefore rounded down to the next lowest whole number, 0, so none of the lowest values are excluded.
  - Up to 20% of the largest values are excluded from the calculation. 20% of 6 values is 1.2. This result is therefore rounded down to the next lowest whole number, 1, and so the single highest value (115) is excluded.

Output:

Std. Dev.	Trimmed Std. Dev.
35.707	13.416

Example 2

Because this function is also available as a window function, it's possible to calculate the trimmed standard deviation of a window frame.

In the following query, the lower cutoff of 0.4 removes up to 40% of the smallest values in the specified window frame. That window frame includes the current row and the two preceding rows.

```
SELECT
    case_id,
    "Order Amount",
```

```
    TRIMMED_STDDEV("Order Amount", 0.4, 0.0)
        OVER (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
        AS "Trimmed Std. Dev. - Window"
FROM THIS_PROCESS
ORDER BY 1
```

Output:

case_id	Order Amount	Trimmed Std. Dev. - Window
00001	20	0
00002	40	10
00003	40	0
00004	20	0
00005	5	10
00006	115	47.5
00007	null	55

The trimmed standard deviation values are calculated as follows:

- For the first row (case_id = 00001), the window frame includes only one value (20). Based on the defined cutoffs, no value can be removed from the window frame, so the standard deviation is 0.
- For the second row (case_id = 00002), the window frame includes the values 20 and 40. Removing the lowest value would exceed the defined 40 percent cutoff limit, so no values are trimmed. Therefore, the standard deviation of values 20 and 40 is 10.
- For the third row (case_id = 00003), the window frame includes the values 20, 40, and 40. It's now possible to trim values because 40% of 3 is 1.2, which is rounded down to 1. That means the lowest value (20) is excluded from the calculation. The standard deviation of the remaining values (40 and 40) is 0.
- Calculations continue like this for all remaining rows in the data set. In the final row (case_id = 00007) `NULL` is ignored, which means the window contains only two values, neither of which are trimmed.

Related Information

Window Functions [page 220]

1.6.1.18 TRIMMED_VARIANCE

Calculates the population variance for a collection of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and performing the calculation.

A cutoff ranges from 0 to 1 and expresses the percentage of values to be trimmed before the calculation is performed. For example, if a group contains 10 values, setting the lower and upper cutoffs to 0.2 means that up to 20% of the lowest values and up to 20% of the highest values are trimmed. 20% of 10 is 2, so the 2 lowest and 2 highest values are trimmed, excluding 4 values in total from the calculation.

If a cutoff results in a non-integer number of values to be trimmed, the resulting number is rounded down to the next lower whole number. For a group of 10 values, a cutoff of 0.15 results in 1.5, in which case the result is rounded down to 1.

> **Note**
>
> This function calculates **population** variance, which measures dispersion using all values in a data set. This shouldn't be confused with **sample** variance, which measures dispersion using a subset of those values and yields a different result.

Syntax

```
TRIMMED_VARIANCE(<expression>, <lowerCutoff>, <upperCutoff>)
```

Parameter	Description	Valid Types
`expression`	The group of values for which you want to determine the variance. `NULL` values are ignored.	Number > **→ Tip** > To calculate the variance of duration values, you can first convert them into numeric values using the functions `DATE_DIFF` or `DURATION_TO_DAYS`.
`lowerCutoff`	A number between 0 and 1. Specifies what percentage of the smallest values are trimmed before calculating the variance. > **Note** > • 0 means that no values are trimmed. > • 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number

Parameter	Description	Valid Types
`upperCutoff`	A number between 0 and 1. Specifies what percentage of the largest values are trimmed before calculating the variance. **Note** • 0 means that no values are trimmed. • 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number

Returns: The trimmed variance as a Number. If all values of `expression` are `NULL` or no values remain after trimming, then `NULL` is returned.

Use as a Window Function

This function can also be used as a window function:

```
TRIMMED_VARIANCE(<expression>, <lowerCutoff>, <upperCutoff>) OVER (
    [ PARTITION BY <partitionExpression> [, <partitionExpression> ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression> ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
TRIMMED_VARIANCE(<expression>, <lowerCutoff>, <upperCutoff>) FILTER (WHERE
<condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	20
00002	40
00003	40
00004	20
00005	5
00006	115
00007	null

The following query demonstrates the difference between calculating variance and trimmed variance.

```
SELECT
    VARIANCE("Order Amount") AS "Variance",
    TRIMMED_VARIANCE("Order Amount", 0.2, 0.2) AS "Trimmed Variance"
FROM THIS_PROCESS
```

- When calculating the **variance**, all (non-`NULL`) values are included in the calculation.
- In this example, the **trimmed variance** has upper and lower cutoffs of 0.2, meaning that up to 20% of the largest and smallest values are excluded from the calculation. There are 6 (non-`NULL`) values and 20% of 6 is 1.2. This result is therefore rounded down to the next lowest whole number, 1, and so the single highest and lowest values (115 and 5) are excluded.

Output:

Variance	Trimmed Variance
1275	100

Example 2

Because this function is also available as a window function, it's possible to calculate the trimmed variance of a window frame.

In the following query, the lower cutoff of 0.4 removes up to 40% of the smallest values in the specified window frame. That window frame includes the current row and the two preceding rows.

```
SELECT
    case_id,
    "Order Amount",
    TRIMMED_VARIANCE("Order Amount", 0.4, 0.0)
        OVER (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
        AS "Trimmed Variance - Window"
FROM THIS_PROCESS
ORDER BY 1
```

Output:

case_id	Order Amount	Trimmed Average - Window
00001	20	0
00002	40	100
00003	40	0
00004	20	0
00005	5	100
00006	115	2256.25
00007	null	3.025

The trimmed variance values are calculated as follows:

- For the first row (case_id = 00001), the window frame includes only one value (20). Based on the defined cutoffs, no value can be removed from the window frame, so the variance is 0.
- For the second row (case_id = 00002), the window frame includes the values 20 and 40. Removing the lowest value would exceed the defined 40 percent cutoff limit, so no values are trimmed. Therefore, the variance of values 20 and 40 is 100.
- For the third row (case_id = 00003), the window frame includes the values 20, 40, and 40. It's now possible to trim values because 40% of 3 is 1.2, which is rounded down to 1. That means the lowest value (20) is excluded from the calculation. The variance of the remaining values (40 and 40) is 0.
- Calculations continue like this for all remaining rows in the data set. In the final row (case_id = 00007) `NULL` is ignored, which means the window contains only two values, neither of which are trimmed.

Related Information

Conversion Functions [page 154]
FILTER Clause [page 20]
Window Functions [page 220]

1.6.1.19 VARIANCE

Calculates the population variance for a collection of values. `NULL` values are ignored.

Variance measures the dispersion of values within a data set, determining how far those values are spread out from their average value.

> **Note**
>
> This function calculates **population** variance, which measures dispersion using all values in a data set. This shouldn't be confused with **sample** variance, which measures dispersion using a subset of those values and yields a different result.

Syntax

```
VARIANCE(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection of values for which you want to determine the variance.	Number → Tip To calculate the variance of duration values, you can first convert them into numeric values using the functions `DATE_DIFF` or `DURATION_TO_DAYS`.

Returns: The variance as a Number. If all values in `expression` are `NULL`, this function returns `NULL`.

Use as a Window Function

This function can also be used as a window function:

```
VARIANCE(<expression>) OVER (
    [ PARTITION BY <partitionExpression> [, <partitionExpression> ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression> ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
VARIANCE(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	20
00002	40
00003	40
00004	20
00005	5
00006	115
00007	null

The following query calculates the variance of all non-`NULL` order amounts.

```
SELECT VARIANCE("Order Amount")
FROM THIS_PROCESS
```

Result:

VARIANCE(Order Amount)
1275

Example 2

Because this function is also available as a window function, it's possible to calculate the variance of a window frame.

The following query calculates the variance of three order values, namely the one in the current row and those in the individual rows immediately preceding and following.

```
SELECT
    case_id,
    "Order Amount",
    VARIANCE("Order Amount")
        OVER (ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
        AS "Windowed Variance"
FROM THIS_PROCESS
```

Result:

case_id	Order Amount	Windowed Variance
00001	20	100
00002	40	88,889

case_id	Order Amount	Windowed Variance
00003	40	88,889
00004	20	205,556
00005	5	2,372,222
00006	115	3,025
00007	null	0

Related Information

Conversion Functions [page 154]
FILTER Clause [page 20]
Window Functions [page 220]

1.6.2 Conditional Functions

Use conditional functions to define which operations to execute or which value to return based on a condition.

COALESCE [page 146]
Accepts an arbitrary number of arguments and returns the first argument that isn't `NULL`.

GREATEST [page 148]
Returns the largest non-`NULL` value from a collection of values.

LEAST [page 150]
Returns the smallest non-`NULL` value from a collection of values.

NULLIF [page 152]
Compares two arguments and returns `NULL` if those arguments are equal. Otherwise, the first argument is returned.

1.6.2.1 COALESCE

Accepts an arbitrary number of arguments and returns the first argument that isn't `NULL`.

Syntax

```
COALESCE(<expression> [, <expression> ...] )
```

Parameter	Description	Valid Types
`expression`	Any combination of values, functions, or operators that evaluate to a value. A minimum of one expression must be provided.	Any **Note** While this function accepts arguments of any type, all expressions must evaluate to the same type within the same invocation of the function.

Returns: The value of the first non-`NULL` expression. If all arguments are `NULL`, this function returns `NULL`.

Example 1

Let's assume a process containing the following data (THIS_PROCESS):

case_id	City	Country	Continent
1	Berlin	Germany	Europe
2	Singapore	null	Asia
3	null	null	Antarctica
4	null	null	null

The following query returns the most precise location available in each case:

```
SELECT COALESCE(City, Country, Continent) AS "Location"
FROM THIS_PROCESS
```

Output:

Location
Berlin
Singapore
Antarctica
null

Example 2

A useful application of `COALESCE` is providing a fallback in case an expression evaluates to `NULL`. The fallback should be a literal value supplied as the last argument to the function.

Assuming the same process data as in Example 1, the following query displays the location in each case, but has a fallback value of "Unknown" in cases where the city is `NULL`.

```
SELECT COALESCE(City, 'Unknown') AS "Location" FROM THIS_PROCESS
```

Output:

Location
Berlin
Singapore
Unknown
Unknown

1.6.2.2 GREATEST

Returns the largest non-`NULL` value from a collection of values.

Syntax

```
GREATEST(<expression> [, <expression> ...] )
```

Parameter	Description	Valid Types
`expression`	Any combination of values, functions, or operators that evaluate to a value. A minimum of one expression must be provided.	Number, Timestamp, Duration **Note** While this function accepts arguments of several types, all expressions must evaluate to the same type within the same invocation of the function.

Returns: The largest non-`NULL` value of all provided expressions. If all arguments are `NULL`, this function returns `NULL`.

Example 1

Assume the following test data (`THIS_PROCESS`):

City	T-shirts Sold	Caps Sold
New York	2,934	3,329
Boston	1,273	984
San Francisco	342	345
Tokyo	0	null
Atlantis	null	null

Cities which are not currently served by the company record sales as either 0 or `NULL`. The following query returns the number sold of the best-selling clothing type per city:

```
SELECT
    City,
    GREATEST("T-shirts Sold", "Caps Sold") as "Number of Best-Selling Type"
FROM THIS_PROCESS
```

Output:

City	Number of Best-Selling Type
New York	3,329
Boston	1,273
San Francisco	345
Tokyo	0
Atlantis	null

Example 2

By providing an additional literal value, you can set a minimum value to be returned or provide a default value in case all other arguments are `NULL`.

Assuming the same input data as the previous example, consider the following query. It demonstrates the behavior of `GREATEST` when the values of its arguments are adjusted:

```
SELECT
    City,
    GREATEST("T-shirts Sold" - 1, "Caps Sold" - 1, 0) as "Number of Best-Selling
Type"
FROM THIS_PROCESS
```

Output:

City	Number of Best-Selling Type
New York	3,328
Boston	1,272
San Francisco	344
Tokyo	0
Atlantis	0

1.6.2.3 LEAST

Returns the smallest non-`NULL` value from a collection of values.

Syntax

```
LEAST(<expression> [, <expression> ...] )
```

Parameter	Description	Valid Types
`expression`	Any combination of values, functions, or operators that evaluate to a value. A minimum of one expression must be provided.	Number, Timestamp, Duration **Note** While this function accepts arguments of several types, all expressions must evaluate to the same type within the same invocation of the function.

Returns: The smallest non-`NULL` value of all provided expressions. If all arguments are `NULL`, this function returns `NULL`.

Example 1

Assume the following test data (`THIS_PROCESS`):

City	T-shirts Sold	Caps Sold
New York	2,934	3,329
Boston	1,273	984

City	T-shirts Sold	Caps Sold
San Francisco	342	345
Tokyo	0	null
Atlantis	null	null

Cities which are not currently served by the company record sales as either 0 or `NULL`. The following query returns the number sold of the least-selling clothing type per city:

```
SELECT
    City,
    LEAST("T-shirts Sold", "Caps Sold") as "Number of Least-Selling Type"
FROM THIS_PROCESS
```

Output:

City	Number of Least-Selling Type
New York	2,934
Boston	984
San Francisco	342
Tokyo	0
Atlantis	null

Example 2

By providing an additional literal value, you can set a maximum value to be returned or provide a default value in case all other arguments are `NULL`.

Assuming the same input data as the previous example, consider the following query. It demonstrates the behavior of `LEAST` when the values of its arguments are adjusted:

```
SELECT
    City,
    LEAST("T-shirts Sold" - 1, "Caps Sold" - 1, 0) as "Number of Least-Selling
Type"
FROM THIS_PROCESS
```

Output:

City	Number of Least-Selling Type
New York	2,933
Boston	983
San Francisco	341
Tokyo	-1
Atlantis	0

1.6.2.4 NULLIF

Compares two arguments and returns `NULL` if those arguments are equal. Otherwise, the first argument is returned.

Syntax

```
NULLIF(<expression1>, <expression2>)
```

Parameter	Description	Valid Types
`expression1`	Any combination of values, functions, or operators that evaluates to a value. **Note** If the two arguments are not equal, `NULLIF` returns this value.	`NULLIF` accepts arguments of any type, but the types must match within the same invocation of the function.
`expression2`	Any combination of values, functions, or operators that evaluates to a value.	

Returns: `NULL` if `expression1` equals `expression2`. Otherwise, this function returns the value of `expression1`.

Example 1

Assuming the following input data (`THIS_PROCESS`):

case_id	Order Status
00001	Delivered
00002	Canceled
00003	Delivered
00004	Delivered
00005	Delivered
00006	Delivered
00007	Delivered
00008	Returned
00009	Canceled
00010	Returned

The following query returns a `NULL` status for each canceled order.

```
SELECT case_id,
    NULLIF("Order Status", 'Canceled') AS Status
FROM THIS_PROCESS
```

case_id	Status
00001	Delivered
00002	null
00003	Delivered
00004	Delivered
00005	Delivered
00006	Delivered
00007	Delivered
00008	Returned
00009	null
00010	Returned

Example 2

Assuming the same process data as the previous example, the next query counts the number of orders that were canceled and the number of orders that have any other status.

```
SELECT NULLIF("Order Status", 'Canceled') AS Status,
    COUNT(case_id) AS Total
FROM THIS_PROCESS
```

By introducing an aggregate function, the result set is grouped by the non-aggregate column.

Result:

Status	Total
null	2
Delivered	6
Returned	2

It's possible to further refine the query to obtain a count of all non-canceled orders.

```
SELECT COUNT(NULLIF("Order Status", 'Canceled')) AS "Non-Canceled Orders"
FROM THIS_PROCESS
```

Non-Canceled Orders
8

Related Information

COALESCE [page 146]

1.6.3 Conversion Functions

Use conversion functions to convert values from one data type to another.

DURATION_FROM_DAYS [page 154]
Converts a number of days into a Duration.

DURATION_FROM_MILLISECONDS [page 155]
Converts a number of milliseconds into a Duration.

DURATION_TO_DAYS [page 156]
Converts a Duration value into the equivalent number of days.

DURATION_TO_MILLISECONDS [page 157]
Converts a Duration value into the equivalent number of milliseconds.

TO_NUMBER [page 158]
Converts an expression to a number.

TO_STRING [page 159]
Converts a number or timestamp to a string.

TO_TIMESTAMP [page 161]
Converts a string to a timestamp.

Timestamp Pattern [page 164]
Timestamp patterns specify a format for showing date and time values.

Related Information

Data types [page 5]

1.6.3.1 DURATION_FROM_DAYS

Converts a number of days into a Duration.

Syntax

```
DURATION_FROM_DAYS(<expression>)
```

Parameter	Description	Valid Types
expression	Number of days.	Number

Returns: A Duration, the length of time represented by the original number of days.

Example

This query demonstrates converting a number of days to a Duration.

```
SELECT DURATION_FROM_DAYS(3.5) AS "3 and a half days"
FROM THIS_PROCESS
```

Output:

3 and a half days
3d 12h

1.6.3.2 DURATION_FROM_MILLISECONDS

Converts a number of milliseconds into a Duration.

Syntax

```
DURATION_FROM_MILLISECONDS(<expression>)
```

Parameter	Description	Valid Types
expression	Number of milliseconds.	Number

Returns: A Duration, the length of time represented by the original number of milliseconds.

Example

This query demonstrates two instances of converting a number of milliseconds to a Duration. In both cases, the argument has a value of 720,000 and therefore returns a duration of 12 minutes.

```
SELECT
    DURATION_FROM_MILLISECONDS(720000) AS "Duration 1",
    DURATION_FROM_MILLISECONDS(12*60*1000) AS "Duration 2"
FROM THIS_PROCESS
```

Output:

Duration 1	Duration 2
12m	12m

1.6.3.3 DURATION_TO_DAYS

Converts a Duration value into the equivalent number of days.

Syntax

```
DURATION_TO_DAYS(<expression>)
```

Parameter	Description	Valid Types
expression	The duration to be converted.	Duration

Returns: A Number, the days in the original duration. The value is a decimal and so may include fractions of a day.

Example

This query calculates a case's duration by subtracting the last event's timestamp from the first event's timestamp, which yields a duration. This duration becomes an argument in a call to `DURATION_TO_DAYS`.

```
SELECT case_id,
    (SELECT FIRST(end_time)) AS "Start",
    (SELECT LAST(end_time)) AS "End",
    (SELECT LAST(end_time)) - (SELECT FIRST(end_time)) AS "Duration",
    DURATION_TO_DAYS( (SELECT LAST(end_time)) - (SELECT FIRST(end_time)) ) AS
"Duration in days"
FROM THIS_PROCESS
```

Output:

case_id	Start	End	Duration	Duration in days
00001	01/08/2020, 12:32	09/08/2020, 09:19	7d 20h	7.866
00002	06/04/2020, 06:38	09/04/2020, 23:43	3d 17h	3.712
00003	21/02/2020, 09:14	02/03/2020, 20:15	10d 11h	10.459
00004	09/10/2020, 19:26	16/10/2020, 02:45	6d 7h	6.305
00005	25/12/2020, 18:21	07/01/2021, 01:24	12d 7h	12.293

Related Information

Duration Literal [page 76]

1.6.3.4 DURATION_TO_MILLISECONDS

Converts a Duration value into the equivalent number of milliseconds.

Syntax

```
DURATION_TO_MILLISECONDS(<expression>)
```

Parameter	Description	Valid Types
expression	The duration to be converted.	Duration

Returns: A Number, the milliseconds in the original duration.

Example

This query calculates a case's duration by subtracting the last event's timestamp from the first event's timestamps, which yields a duration. This duration becomes an argument in two calls to `DURATION_TO_MILLISECONDS`. The first call returns the number of milliseconds. The second call does the same but is followed by a conversion of the millisecond value into minutes.

```
SELECT case_id,
    (SELECT FIRST(end_time)) AS "Start",
    (SELECT LAST(end_time)) AS "End",
    (SELECT LAST(end_time)) - (SELECT FIRST(end_time)) AS "Duration",
    DURATION_TO_MILLISECONDS( (SELECT LAST(end_time)) - (SELECT
FIRST(end_time)) ) AS "In ms",
    DURATION_TO_MILLISECONDS( (SELECT LAST(end_time)) - (SELECT
FIRST(end_time)) ) / (1000*60) AS "In min"
FROM THIS_PROCESS
```

Output:

case_id	Start	End	Duration	In ms	In min
00001	01/08/2020, 12:32	09/08/2020, 09:19	7d 20h	679,612,000	11,326.87
00002	06/04/2020, 06:38	09/04/2020, 23:43	3d 17h	320,697,000	5,344.95

case_id	Start	End	Duration	In ms	In min
00003	21/02/2020, 09:14	02/03/2020, 20:15	10d 11h	903,668,000	15,061.13
00004	09/10/2020, 19:26	16/10/2020, 02:45	6d 7h	544,717,000	9,078.62
00005	25/12/2020, 18:21	07/01/2021, 01:24	12d 7h	1,062,150,000	17,702.50

1.6.3.5 TO_NUMBER

Converts an expression to a number.

> ⚠ Caution
>
> For very large data sets, this function may require excessive CPU activity, causing long query execution times.
>
> For more information, refer to Performance [page 294].

Syntax

```
TO_NUMBER(<expression>)
```

Parameter	Description	Valid Types
`expression`	Evaluates to a value to be converted into a number.	String

Returns: A numeric value, if the value of the string `expression` can be converted to a number. If `expression` can't be converted, then this function returns `NULL`.

Formatting

The `expression` argument must strictly follow a numeric format. The only accepted non-numeric character is a decimal point – commas aren't supported for separating the integer and fractional parts of a number.

Extraneous characters, such as currency symbols or thousands separators, prevent the number from being converted.

> → Tip
>
> Consider using the `SUBSTRING_BEFORE` or `SUBSTRING_AFTER` functions to remove extraneous characters from a string before conversion. For example, if your "Order Amount" column records

values with a leading currency symbol (like '$123.45'), the value could be converted like so: `TO_NUMBER(SUBSTRING_AFTER("Order Amount", '$'))`.

This function doesn't support formatting the output.

Example 1

Let's assume we have the following process data (`THIS_PROCESS`). The days and order amounts are stored as strings, but aren't formatted consistently.

case_id	Day	Order Amount
1	1	100
2	2nd	1000
3	3	1,500
4	null	$300
5	five	99.45

The following query converts those values into numbers where possible. In the case of the Order Amount, the query also performs arithmetic on the resulting value.

```
SELECT
    TO_NUMBER("Day") AS "Day",
    TO_NUMBER("Order Amount") * 0.9 AS "Discounted Amount"
FROM THIS_PROCESS
```

Output:

case_id	Day	Discounted Amount
00001	1	90
00002	2	900
00003	3	null
00004	4	null
00005	5	99.45

1.6.3.6 TO_STRING

Converts a number or timestamp to a string.

When converting a timestamp to a string, you can optionally provide its format using a timestamp pattern. If a pattern isn't provided, the timestamp value is parsed according to RFC 3339 .

Syntax

```
TO_STRING(<expression> [, <pattern>] )
```

Parameter	Description	Valid Types
`expression`	Evaluates to a value to be converted to a string.	Number, Timestamp
`pattern`	Optional. • If `expression` is a number, providing `pattern` leads to an error. • If `expression` is a timestamp, `pattern` defines the timestamp format. **Note** This pattern follows a defined syntax. Refer to the Timestamp Pattern topic.	String literal

Returns: The value of `expression` as a string. If `expression` evaluates to `NULL`, this function returns `NULL`.

Example 1

The query in this example shows conversion of numeric and timestamp values.

The order amount is a numeric value. To display the amount with a currency symbol, the value must first be converted to a string before it's concatenated to the symbol.

The date selected is that of the start event in each case. It's converted to a string without providing a timestamp pattern, meaning it's displayed using the default pattern.

```
SELECT
    case_id,
    "Order Amount",
    CONCAT('$', TO_STRING("Order Amount")) AS "Order Amount ($)",
    (SELECT FIRST(end_time)) AS "Creation Date",
    TO_STRING((SELECT FIRST(end_time))) AS "Date as String"
FROM THIS_PROCESS
```

Output:

case_id	Order Amount	Order Amount ($)	Creation Date	Date as String
00001	944.42	$944.42	01/08/2020, 12:32	2020-08-01T12:32:27.000Z

case_id	Order Amount	Order Amount ($)	Creation Date	Date as String
00002	270.04	$270.04	06/04/2020, 06:38	2020-04-06T06:38:52.000Z
00003	469.9	$469.9	21/02/2020, 09:14	2020-02-21T09:14:08.000Z
00004	268.34	$268.34	09/10/2020, 19:26	2020-10-09T19:26:33.000Z
00005	327.94	$327.94	25/12/2020, 18:21	2020-12-25T18:21:57.000Z

Example 2

The following query demonstrates how the timestamp pattern can be used to format the output of a timestamp when it's converted to a string.

```
SELECT
    case_id,
    (SELECT FIRST(end_time)) AS "Creation Date",
    TO_STRING((SELECT FIRST(end_time)), 'DD-MM-YYYY') AS "Date",
    TO_STRING((SELECT FIRST(end_time)), 'HH:mm:ss.SSS') AS "24hr Time w/ ms",
    TO_STRING((SELECT FIRST(end_time)), 'hh:mm aa') AS "12hr Time"
FROM THIS_PROCESS
```

Output:

case_id	Creation Date	Date	24hr Time w/ ms	12hr Time
00001	01/08/2020, 12:32	01/08/2020	12:32:27	12:32 PM
00002	06/04/2020, 06:38	06/04/2020	06:38:52	6:38 AM
00003	21/02/2020, 09:14	21/02/2020	09:14:08	9:14 AM
00004	09/10/2020, 19:26	09/10/2020	19:26:33	7:26 PM
00005	25/12/2020, 18:21	25/12/2020	18:21:57	6:21 PM

Related Information

Timestamp Pattern [page 164]

1.6.3.7 TO_TIMESTAMP

Converts a string to a timestamp.

You can optionally provide the format of the timestamp using a timestamp pattern. If a pattern isn't provided, the timestamp value is parsed according to RFC 3339.

> ⚠ Caution
>
> For very large data sets, this function may require excessive CPU activity when including a timestamp pattern, causing long query execution times.
>
> For more information, refer to Performance [page 294].

Syntax

```
TO_TIMESTAMP(<expression> [, <pattern>] )
```

Parameter	Description	Valid Types
`expression`	Evaluates to a value to be converted into a timestamp value.	String
`pattern`	Optional. Defines the timestamp format. > ⓘ Note > This pattern follows a defined syntax. Refer to the Timestamp Pattern topic.	String literal

Returns: The `expression` as a timestamp. If `expression` evaluates to `NULL`, this function returns `NULL`.

Example 1

Let's assume we have the following process data (THIS_PROCESS), where the timestamp and date information is stored as strings:

case_id	timestamp_string	date_string
1	1970-01-01T00:00:00.000Z	01/01/1970
2	1970-01-01T00:00:01.000Z	01/01/1970
3	null	null
4	2023-09-13T18:55:30.000+02:00	13/09/2023
5	2023-09-13T16:55:30.000Z	13/09/2023

The following query converts the strings into timestamp values.

```
SELECT
    case_id,
    TO_TIMESTAMP(timestamp_string) AS "Timestamp Value",
    TO_TIMESTAMP(date_string, 'DD/MM/YYYY') AS "Date Value"
FROM THIS_PROCESS
```

case_id	Timestamp Value	Date Value
00001	01/01/1970, 00:00	01/01/1970, 00:00
00002	01/01/1970, 00:00	01/01/1970, 00:00
00003	null	null
00004	13/09/2023, 16:55	13/09/2023, 00:00
00005	13/09/2023, 16:55	13/09/2023, 00:00

Note that the "Timestamp Value" in cases 1 and 2 are stored internally as 0 and 1000 respectively (since a timestamp is the number of milliseconds since midnight on Jan 1, 1970).

Example 2

Continuing with the data from the previous example, this query demonstrates how `TO_STRING` and `TO_TIMESTAMP` are inverses of one another.

The following query returns the original timestamp string value after conversion to and from a timestamp, but in the timezone UTC+0.

```
SELECT
    case_id,
    TO_STRING(TO_TIMESTAMP(timestamp_string)) AS "Original Timestamp"
FROM THIS_PROCESS
```

Output:

case_id	Original Timestamp
00001	1970-01-01T00:00:00.000Z
00002	1970-01-01T00:00:01.000Z
00003	null
00004	2023-09-13T16:55:30.000Z
00005	2023-09-13T16:55:30.000Z

Related Information

RFC 3339

Timestamp Pattern [page 164]
TO_STRING [page 159]

1.6.3.8 Timestamp Pattern

Timestamp patterns specify a format for showing date and time values.

A timestamp pattern is a string used when converting date and time values to strings. Each character in the pattern specifies which element of a timestamp features in the output and how it appears.

The following table shows all available pattern characters.

Pattern Character	Description
YYYY	Year (0000-9999)
YY	Two-digit year, that is, year modulo 100 (00-99) **Note** This pattern character is supported only in the `TO_STRING` function.
MM	Month, padded with zero (01-12)
M	Month (1-12)
DD	Day of month, padded with zero (01-31)
D	Day of month (1-31)
HH	Hour, 24h clock, padded with zero (00-23)
H	Hour, 24h clock (0-23)
hh	Hour, 12h clock, padded with zero (01-12)
h	Hour, 12h clock (1-12)
aa	am/pm (lowercase)
AA	AM/PM (uppercase)
mm	Minute, padded with zero (00-59)
m	Minute (0-59)
ss	Second, padded with zero (00-60)
s	Second (0-60)
SSS	Fractional seconds with millisecond precision (000-999)
SSSSSS	Fractional seconds with microsecond precision (000000-999999)
SSSSSSSSS	Fractional seconds with nanosecond precision (000000000-999999999)
Z	Timezone offset, hours only (-24-+24)
ZZ	Timezone offset, hours, and minutes (-2400-+2459)
ZZZ	Timezone offset, hours, and minutes separated by colon (-24:00-+24:59)
T	Literal **T**

Pattern Character	Description
Any other letter	Reserved for future extensions
[,], {, }, %, #	Reserved for future extensions
''	Literal single quote (`'`)
'string'	Literal `string`
Any other character	Printed literally

Remarks

Some pattern characters behave more leniently:

- Both padded and unpadded inputs are accepted. For example, the pattern `M/D/YYYY` also parses `01/01/1970` and `MM/DD/YYYY` also parses `1/1/1970`.
- Timezones are parsed in any supported format. `Z/ZZ/ZZZ` parses any of the following timezone offsets: `Z/+00/+0000/ +00:00`

The patterns for fractional sections (`SSS` **/** `SSSSSS` **/** `SSSSSSSSS`**) expect the exact number of fractional digits**:

- For example, the pattern `.SSS` doesn't accept an input of `.1`.

When the optional pattern parameter is omitted, the following applies:

- A default pattern is used, namely:`YYYY-MM-DDTHH:mm:ss.SSSZZZ`.
- The function `TO_TIMESTAMP` also accepts a space instead of the literal `T`.
- The function `TO_STRING` formats the timezone as a literal `Z`. All timestamps are stored using UTC internally.

1.6.4 Date Functions

Use date functions to execute operations on date and time values.

> **Note**
>
> The default timestamp format in SIGNAL is dd/mm/yyyy.

DATE_ADD [page 166]
Adds a given number of date or time units to a timestamp.

DATE_DIFF [page 170]
Returns the period between two timestamps as a number of date or time units.

DATE_PART [page 173]
Extracts part of a date from a given timestamp.

DATE_TRUNC [page 174]
Truncates the displayed precision level of timestamps.

DURATION_BETWEEN [page 176]
Calculates the working time between two timestamps.

DURATION_FROM_DAYS [page 178]
Converts a number of days into a Duration.

DURATION_FROM_MILLISECONDS [page 179]
Converts a number of milliseconds into a Duration.

DURATION_TO_DAYS [page 179]
Converts a Duration value into the equivalent number of days.

DURATION_TO_MILLISECONDS [page 180]
Converts a Duration value into the equivalent number of milliseconds.

NOW [page 181]
Returns the current UTC timestamp.

1.6.4.1 DATE_ADD

Adds a given number of date or time units to a timestamp.

Syntax

```
DATE_ADD(<unit>, <number>, <timestamp>)
```

Parameter	Description	Valid Types
`unit`	The date or time unit to be added. The following units are supported: • 'year' • 'quarter' • 'month' • 'week' • 'day' • 'hour' • 'minute' • 'second' • 'millisecond'	String literal

Parameter	Description	Valid Types
`number`	An expression determining the number of units to add to `timestamp`. This parameter accepts negative values, allowing you to subtract units. **Note** Fractional parts of this value are ignored.	Number
`timestamp`	A timestamp expression.	Timestamp

Returns: A new timestamp, the result of adding the specified number of units to the `timestamp` parameter. If `number` or `timestamp` is `NULL`, then this function returns `NULL`.

Handling Calendar Time

The function doesn't take calendar quarters or weeks into account:

- Adding a **year** is equivalent to adding 12 months.
- Adding a **quarter** is equivalent to adding 3 months.
- Adding a **week** is equivalent to adding 7 days.

Months are handled specially. Adding `N` **months** increases the month part of the timestamp by `N`, wrapping the value around as necessary if the year boundary is passed. If the day part of the resulting timestamp would be greater than the number of days in the month, then the day value becomes the last day of the month.

For example, simply adding a month to '2023-01-31' would yield '2023-02-31', an invalid date. Therefore, `DATE_ADD('month', 1, DATE '2023-01-31')` returns '2023-02-28'

The special handling of months means that this function isn't associative, meaning that chaining together calls to `DATE_ADD` can have different results depending on the order of the calls. Refer to Example 3 for a demonstration.

Example 1

Let's assume we have the following data set (`THIS_PROCESS`):

start_date
01/01/2023, 00:00
31/01/2023, 00:00
null

Running the query

```
SELECT start_date,
    DATE_ADD('year', 1, start_date) AS "+ 1 year",
    DATE_ADD('quarter', 1, start_date) AS "+ 1 quarter",
    DATE_ADD('month', 1, start_date) AS "+1 month",
    DATE_ADD('day', 1, start_date) AS "+1 day",
    DATE_ADD('hour', 1, start_date) AS "+1 hour"
FROM THIS_PROCESS
```

produces the following output:

start_date	+ 1 year	+ 1 quarter	+1 month	+1 day	+1 hour
01/01/2023, 00:00	01/01/2024, 00:00	01/04/2023, 00:00	01/02/2023, 00:00	02/01/2023, 00:00	01/01/2023, 01:00
31/01/2023, 00:00	31/01/2024, 00:00	30/04/2023, 00:00	28/02/2023, 00:00	01/02/2023, 00:00	31/01/2023, 01:00
null	null	null	null	null	null

Example 2

Using the process data from the previous example, the following query demonstrates subtracting time units from a timestamp:

```
SELECT start_date,
    DATE_ADD('year', -2, start_date) AS "-2 years",
    DATE_ADD('quarter', -3, start_date) AS "-3 quarters",
    DATE_ADD('month', -4, start_date) AS "-4 months",
    DATE_ADD('day', -5, start_date) AS "-5 days",
    DATE_ADD('hour', -6, start_date) AS "-6 hours"
FROM THIS_PROCESS
-- Filter out null values (demo'ed in Example 1)
WHERE start_date IS NOT NULL
```

Output:

start_date	-2 years	-3 quarters	-4 months	-5 days	-6 hours
01/01/2023, 00:00	01/01/2021, 00:00	01/04/2022, 00:00	01/09/2022, 00:00	27/12/2022, 00:00	31/12/2022, 18:00
31/01/2023, 00:00	31/01/2021, 00:00	30/04/2022, 00:00	30/09/2022, 00:00	26/01/2023, 00:00	30/01/2023, 18:00

Example 3

Because `DATE_ADD` both accepts and returns a timestamp, it's possible to chain together calls to this function. For example, the following expression is valid:

```
DATE_ADD('week', 1, DATE_ADD('hour', 1, start_date))
```

However, the special handling of months means DATE_ADD isn't associative, so chaining calls to this function can yield different results depending on the ordering of calls. Sticking with the previous input data, running the query

```
SELECT start_date,
    -- Column 2: Subtract a month, then add a month
    DATE_ADD('month', 1, DATE_ADD('month', -1, start_date)) as "-1 then +1
month",
    -- Column 3: Add a month, then subtract a month
    DATE_ADD('month', -1, DATE_ADD('month', 1, start_date)) as "+1 then -1 month"
FROM THIS_PROCESS
WHERE start_date IS NOT NULL
```

produces the following output:

start_date	-1 then +1 month	+1 then -1 month
01/01/2023, 00:00	01/01/2023, 00:00	01/01/2023, 00:00
31/01/2023, 00:00	31/01/2023, 00:00	28/01/2023, 00:00

In column 2, the inner call to DATE_ADD is executed first and subtracts a month from `start_date`:

- Subtracting a month from '2023-01-01' yields '2022-12-01'
- Subtracting a month from '2023-01-31' yields '2022-12-31'

The outer call is then executed and adds a month to the result:

- Adding a month to '2022-12-01' yields '2023-01-01'
- Adding a month to '2022-12-31' yields '2023-01-31'

In column 3, the inner call to DATE_ADD adds a month to `start_date`:

- Adding a month to '2023-01-01' yields '2023-02-01'
- Adding a month to '2023-01-31' yields '2023-02-28'. Note that simply increasing the month value would yield '2023-02-31', an invalid date, so DATE_ADD adjusts the day value to the last day in February 2023.

The second call then subtracts a month from the result:

- Subtracting a month from '2023-02-01' yields '2023-01-01'
- Subtracting a month from '2023-02-28' yields '2023-01-28'

Related Information

DATE_DIFF [page 170]
DATE_TRUNC [page 174]

1.6.4.2 DATE_DIFF

Returns the period between two timestamps as a number of date or time units.

> → Tip
>
> It's also possible to calculate the period between two timestamps as a duration value using one of the following methods:
>
> - Subtract one timestamp from the other.
> - Use the `DURATION_BETWEEN` function, which calculates the duration according to a given worktime calendar.

Syntax

```
DATE_DIFF( <unit>, <startTimestamp>, <endTimestamp> )
```

Parameter	Description	Valid Types
`unit`	The unit of measurement of the returned value. The following units are supported: • 'year' • 'quarter' • 'month' • 'week' • 'day' • 'hour' • 'minute' • 'second' • 'millisecond'	String literal
`startTimestamp`	A timestamp expression.	Timestamp
`endTimestamp`	A timestamp expression.	Timestamp

Returns: A number representing the period between the two timestamps. Note that:

- The number is always an integer. Fractional units aren't included.
- If a timestamp is `NULL`, this function returns `NULL`.

Handling Calendar Time

The function doesn't take calendar years, quarters, or weeks into account:

- A **year** is equivalent to 12 months.
- A **quarter** is equivalent to 3 months.
- A **week** is equivalent to 7 days.

The counting of months is handled specially. One **month** is considered elapsed each time the calendar month can be increased without reaching the month of `endTimestamp`. When the month prior to `endTimestamp` is reached, a further month is considered elapsed only if the day and time parts of `endTimestamp` are greater than or equal to those of `startTimestamp`.

For example, consider the dates '2024-01-15' and '2024-04-14' as `startTimestamp` and `endTimestamp` respectively. Beginning with January 2024, we count two increases in the calendar month until reaching March 2024, which is the month prior to April 2024. In this case, the day part of `endTimestamp` is less than that of `startTimestamp`, so a further month isn't considered elapsed. There's therefore a two-month period between '2024-01-15' and '2024-04-14'.

The special handling of months means that the behavior of `DATE_DIFF` isn't always consistent with the behavior of the `DATE_ADD` function. Using `DATE_ADD` to add `N` months to date `D` doesn't necessarily mean that `DATE_DIFF` returns `N` when comparing the result with `D`. Refer to Example 2 for a demonstration.

Example 1

Let's assume we have the following data set (`THIS_PROCESS`):

start_date	end_date
01/01/2023, 00:00	01/06/2024, 00:00
31/01/2023, 00:00	29/02/2024, 00:00
31/10/2023, 00:00	null

Running the query

```
SELECT start_date,
    end_date,
    DATE_DIFF('year', start_date, end_date) AS diff_year,
    DATE_DIFF('quarter', start_date, end_date) AS diff_quarter,
    DATE_DIFF('month', start_date, end_date) AS diff_month,
    DATE_DIFF('day', start_date, end_date) AS diff_day
FROM THIS_PROCESS
```

produces the following output:

start_date	end_date	diff_year	diff_quarter	diff_month	diff_day
01/01/2023, 00:00	01/06/2024, 00:00	1	5	17	517
31/01/2023, 00:00	29/02/2024, 00:00	1	4	12	394
31/10/2023, 00:00	null	null	null	null	null

Example 2

The special handling of months means that the behavior of the `DATE_DIFF` function isn't always consistent with that of the `DATE_ADD` function. Sticking with the previous input data, the following query shows three values in each case: `start_date`, `start_date` with one month added to it, and the difference between the two according to `DATE_DIFF`:

```
SELECT start_date,
    DATE_ADD('month', 1, start_date) AS start_plus_1_month,
    DATE_DIFF('month', start_date, (DATE_ADD('month', 1, start_date))) AS
diff_month
FROM THIS_PROCESS
```

Output:

start_date	start_plus_1_month	diff_month
01/01/2023, 00:00	01/02/2023, 00:00	1
31/01/2023, 00:00	28/02/2023, 00:00	0
31/10/2023, 00:00	30/11/2023, 00:00	0

In the second and third cases, the day part of `start_plus_1_month` is less than that of `start_date`, and so this difference isn't considered a month.

To avoid this behavior, you can use the `DATE_TRUNC` function to restrict the precision of your timestamps to the month level, as the following query does:

```
SELECT start_date,
    DATE_TRUNC('month', start_date) AS start_truncated,
    DATE_ADD('month', 1, DATE_TRUNC('month', start_date)) AS start_plus_1_month,
    DATE_DIFF('month',
        DATE_TRUNC('month', start_date),
        DATE_ADD('month', 1, DATE_TRUNC('month', start_date))
    ) AS diff_month
FROM THIS_PROCESS
```

Output:

start_date	start_truncated	start_plus_1_month	diff_month
01/01/2003, 00:00	01/01/2003, 00:00	01/02/2003, 00:00	1
31/01/2023, 00:00	01/01/2003, 00:00	01/02/2003, 00:00	1
31/10/2003, 00:00	01/10/2003, 00:00	01/11/2003, 00:00	1

Related Information

DATE_ADD [page 166]
DATE_TRUNC [page 174]
DURATION_BETWEEN [page 176]

1.6.4.3 DATE_PART

Extracts part of a date from a given timestamp.

For example, you can display the month and week of a timestamp in separate columns or you can display date information that is not displayed by default, for example the hour.

Syntax

```
DATE_PART(<precision>, <expression>)
```

Parameter	Description	Valid Types
precision	The precision level in single quotes for the timestamp that is returned. Available values: • 'year' • 'quarter' • 'month' • 'week' (ISO 8601-week numbering is applied) • 'day' • 'day_of_week' (returns 0 – 6, beginning with Sunday = 0) • 'hour' • 'minute' • 'second' • millisecond	String literal
expression	The timestamp from which you want to extract a part.	Timestamp

Returns: A Number corresponding to the value of the extracted part of the date.

Example

```
SELECT
    "case_id",
    (SELECT FIRST("end_time")) AS "Timestamp",
    DATE_PART('day', (SELECT FIRST("end_time"))) AS "Day",
    DATE_PART('day_of_week', (SELECT FIRST("end_time"))) AS "Day of week",
    DATE_PART('week', (SELECT FIRST("end_time"))) AS "Week",
    DATE_PART('month', (SELECT FIRST("end_time"))) AS "Month",
    DATE_PART('year', (SELECT FIRST("end_time"))) AS "Year",
    DATE_PART('hour', (SELECT FIRST("end_time"))) AS "Hour",
```

```
    DATE_PART('minute', (SELECT FIRST("end_time"))) AS "Minute",
    DATE_PART('second', (SELECT FIRST("end_time"))) AS "Second",
    DATE_PART('millisecond', (SELECT FIRST("end_time"))) AS "Millisecond"
FROM THIS_PROCESS
```

This query extracts date parts from the timestamp of the cases and displays them in separate columns. If the underlying data doesn't provide the queried precision, the value is displayed as "0", as in this example for milliseconds.

case_id	Time-stamp	Day	Day of week	Week	Month	Year	Hour	Minute	Second	Millisec-ond
00001	01/08/2020, 12:32	1	6	31	8	2020	12	32	27	0
00002	06/04/2020, 06:38	6	1	15	4	2020	6	38	52	0
00003	21/02/2020, 09:14	21	5	8	2	2020	9	14	8	0
00004	09/10/2020, 19:26	9	5	41	10	2020	19	26	33	0
00005	25/12/2020, 18:21	25	5	52	12	2020	18	21	57	0
00006	24/01/2020, 16:09	24	5	4	1	2020	16	9	56	0
00007	28/08/2020, 13:03	28	5	35	8	2020	13	3	14	0
00008	23/03/2020, 18:44	23	1	13	3	2020	18	44	31	0
00009	30/12/2020, 15:27	30	3	53	12	2020	15	27	42	0
00010	02/01/2020, 09:02	2	4	1	1	2020	9	2	53	0

1.6.4.4 DATE_TRUNC

Truncates the displayed precision level of timestamps.

Truncation in this context means that all precision levels below the truncated date part of the timestamp are displayed as "01". For example, if you truncate the timestamp with precision level "year", all months and days are set to "01".

Syntax

```
DATE_TRUNC(<precision>, <expression>)
```

Parameter	Description	Valid Types
precision	The precision level in single quotes for the timestamp that is returned. Available values: • 'year' • 'quarter' • 'month' • 'week' (ISO 8601-week numbering is applied) • 'day' • 'hour' • 'minute' • 'second' • 'millisecond'	String literal
expression	The timestamp you want to truncate.	Timestamp

Returns: A Timestamp with the specified truncated level of precision.

Example

```
SELECT
    DATE_TRUNC('day', (SELECT FIRST(END_TIME))) AS "Truncated (day)",
    DATE_TRUNC('month', (SELECT FIRST(END_TIME))) AS "Truncated (month)",
    DATE_TRUNC('year', (SELECT FIRST(END_TIME))) AS "Truncated (year)"
FROM THIS_PROCESS
```

This query returns the given timestamps with truncated date parts:

- DATE_TRUNC('day') returns the timestamps unmodified because precision levels below "day" are not displayed.
- DATE_TRUNC('month') returns the timestamps with all precision levels below month are set to "01", that is 01/mm/yyyy.
- DATE_TRUNC('year') returns the timestamps with all precision levels below year are set to "01", that is 01/01/yyyy.

Example output:

Truncated (day)	Truncated (month)	Truncated (year)
08/08/2017	01/08/2017	01/01/2017

Truncated (day)	Truncated (month)	Truncated (year)
12/07/2017	01/07/2017	01/01/2017
26/06/2017	01/06/2017	01/01/2017
23/09/2017	01/09/2017	01/01/2017
28/07/2017	01/07/2017	01/01/2017

1.6.4.5 DURATION_BETWEEN

Calculates the working time between two timestamps.

The calculation is done according to a specific weekday calendar of relevant 'working' and 'non-working' time. The built-in weekday calendars support a possible level of precision down to the millisecond.

> ⓘ Note
>
> The built-in calendars are defined at the day level. Working time for work days is considered as a full 24 hours.

Syntax

```
DURATION_BETWEEN(<timestampStart>, <timestampEnd>, <calendarID>)
```

Parameter	Description	Valid Types
timestampStart	The start of the duration.	Timestamp
timestampEnd	The end of the duration.	Timestamp
calendarID	Label referring to a weekday calendar. The following calendars are available: • Monday–Friday (use label 'WEEKDAY_MTWTF') • Monday–Saturday ('WEEKDAY_MTWTFS') • Sunday–Thursday ('WEEKDAY_SMTWT') • Saturday–Thursday ('WEEKDAY_SSMTWT') • Sunday–Friday ('WEEKDAY_SMTWTF') • Monday–Thursday and Saturday ('WEEKDAY_MTWT_S')	String (literal)

Returns: The difference between `timestampStart` and `timestampEnd` as a Duration.

Example

The query below demonstrates how the choice of weekday calendar affects calculated durations.

Three pairs of timestamps are selected. For each pair, the `duration` field, calculated without reference to working time, displays the simple time difference. Each subsequent column shows how the original time difference changes when taking into account a different weekday calendar.

Example

```
SELECT case_id,
    (SELECT FIRST(end_time)) AS "start",
    (SELECT LAST(end_time)) AS "end",
    (SELECT LAST(end_time)) - (SELECT FIRST(end_time)) AS "duration",
    DURATION_BETWEEN(
        (SELECT FIRST(end_time)),
        (SELECT LAST(end_time)),
        'WEEKDAY_MTWTF'
    ) AS "MTWTF",
    DURATION_BETWEEN(
        (SELECT FIRST(end_time)),
        (SELECT LAST(end_time)),
        'WEEKDAY_MTWTFS'
    ) AS "MTWTFS",
    DURATION_BETWEEN(
        (SELECT FIRST(end_time)),
        (SELECT LAST(end_time)),
        'WEEKDAY_SMTWT'
    ) AS "SMTWT",
    DURATION_BETWEEN(
        (SELECT FIRST(end_time)),
        (SELECT LAST(end_time)),
        'WEEKDAY_SSMTWT')
    AS "SSMTWT",
    DURATION_BETWEEN(
        (SELECT FIRST(end_time)),
        (SELECT LAST(end_time)),
        'WEEKDAY_SMTWTF'
    ) AS "SMTWTF",
    DURATION_BETWEEN(
        (SELECT FIRST(end_time)),
        (SELECT LAST(end_time)),
        'WEEKDAY_MTWT_S'
    ) AS "MTWT_S"
FROM THIS_PROCESS
LIMIT 3
```

The weekdays in each case are:

- Case 00001: Saturday (01/08/2020) and Sunday (09/08/2020)
- Case 00002: Monday (06/04/2020) and Thursday (09/04/2020)
- Case 00003: Friday (21/02/2020) and Monday (02/03/2020)

Example output:

case_id	start	end	duration	MTWTF	MTWTFS	SMTWT	SSMTWT	SMTWTF	MTWT_S
00001	01/08/2020, 12:32	09/08/2020, 09:19	7d 20h	4d 20h	6d 20h	4d 20h	6d 20h	5d 20h	5d 20h

case_id	start	end	duration	MTWTF	MTWTFS	SMTWT	SSMTWT	SMTWTF	MTWT_S
00002	06/04/2020, 06:38	09/04/2020, 23:43	3d 17h	3d 17h	3d 17h	3d 17h	3d 17h	3d 17h	3d 17h
00003	21/02/2020, 09:14	02/03/2020, 20:15	10d 11h	6d 11h	8d 11h	6d 11h	8d 11h	8d 11h	6d 11h

1.6.4.6 DURATION_FROM_DAYS

Converts a number of days into a Duration.

Syntax

```
DURATION_FROM_DAYS(<expression>)
```

Parameter	Description	Valid Types
expression	Number of days.	Number

Returns: A Duration, the length of time represented by the original number of days.

Example

This query demonstrates converting a number of days to a Duration.

```
SELECT DURATION_FROM_DAYS(3.5) AS "3 and a half days"
FROM THIS_PROCESS
```

Output:

3 and a half days
3d 12h

1.6.4.7 DURATION_FROM_MILLISECONDS

Converts a number of milliseconds into a Duration.

Syntax

```
DURATION_FROM_MILLISECONDS(<expression>)
```

Parameter	Description	Valid Types
expression	Number of milliseconds.	Number

Returns: A Duration, the length of time represented by the original number of milliseconds.

Example

This query demonstrates two instances of converting a number of milliseconds to a Duration. In both cases, the argument has a value of 720,000 and therefore returns a duration of 12 minutes.

```
SELECT
    DURATION_FROM_MILLISECONDS(720000) AS "Duration 1",
    DURATION_FROM_MILLISECONDS(12*60*1000) AS "Duration 2"
FROM THIS_PROCESS
```

Output:

Duration 1	Duration 2
12m	12m

1.6.4.8 DURATION_TO_DAYS

Converts a Duration value into the equivalent number of days.

Syntax

```
DURATION_TO_DAYS(<expression>)
```

Parameter	Description	Valid Types
expression	The duration to be converted.	Duration

Returns: A Number, the days in the original duration. The value is a decimal and so may include fractions of a day.

Example

This query calculates a case's duration by subtracting the last event's timestamp from the first event's timestamp, which yields a duration. This duration becomes an argument in a call to `DURATION_TO_DAYS`.

```
SELECT case_id,
    (SELECT FIRST(end_time)) AS "Start",
    (SELECT LAST(end_time)) AS "End",
    (SELECT LAST(end_time)) - (SELECT FIRST(end_time)) AS "Duration",
    DURATION_TO_DAYS( (SELECT LAST(end_time)) - (SELECT FIRST(end_time)) ) AS
"Duration in days"
FROM THIS_PROCESS
```

Output:

case_id	Start	End	Duration	Duration in days
00001	01/08/2020, 12:32	09/08/2020, 09:19	7d 20h	7.866
00002	06/04/2020, 06:38	09/04/2020, 23:43	3d 17h	3.712
00003	21/02/2020, 09:14	02/03/2020, 20:15	10d 11h	10.459
00004	09/10/2020, 19:26	16/10/2020, 02:45	6d 7h	6.305
00005	25/12/2020, 18:21	07/01/2021, 01:24	12d 7h	12.293

Related Information

Duration Literal [page 76]

1.6.4.9 DURATION_TO_MILLISECONDS

Converts a Duration value into the equivalent number of milliseconds.

Syntax

```
DURATION_TO_MILLISECONDS(<expression>)
```

Parameter	Description	Valid Types
expression	The duration to be converted.	Duration

Returns: A Number, the milliseconds in the original duration.

Example

This query calculates a case's duration by subtracting the last event's timestamp from the first event's timestamps, which yields a duration. This duration becomes an argument in two calls to `DURATION_TO_MILLISECONDS`. The first call returns the number of milliseconds. The second call does the same but is followed by a conversion of the millisecond value into minutes.

```
SELECT case_id,
    (SELECT FIRST(end_time)) AS "Start",
    (SELECT LAST(end_time)) AS "End",
    (SELECT LAST(end_time)) - (SELECT FIRST(end_time)) AS "Duration",
    DURATION_TO_MILLISECONDS( (SELECT LAST(end_time)) - (SELECT
FIRST(end_time)) ) AS "In ms",
    DURATION_TO_MILLISECONDS( (SELECT LAST(end_time)) - (SELECT
FIRST(end_time)) ) / (1000*60) AS "In min"
FROM THIS_PROCESS
```

Output:

case_id	Start	End	Duration	In ms	In min
00001	01/08/2020, 12:32	09/08/2020, 09:19	7d 20h	679,612,000	11,326.87
00002	06/04/2020, 06:38	09/04/2020, 23:43	3d 17h	320,697,000	5,344.95
00003	21/02/2020, 09:14	02/03/2020, 20:15	10d 11h	903,668,000	15,061.13
00004	09/10/2020, 19:26	16/10/2020, 02:45	6d 7h	544,717,000	9,078.62
00005	25/12/2020, 18:21	07/01/2021, 01:24	12d 7h	1,062,150,000	17,702.50

1.6.4.10 NOW

Returns the current UTC timestamp.

What is UTC?

UTC is Coordinated Universal Time, the main standard by which clocks are globally synchronized.
A UTC timestamp represents time measured at 0° longitude. All time zones are offset from UTC to calculate local time. For example, Central European Time (CET) is UTC+1. A local time of 09:00 CET would be 08:00 UTC.

UTC doesn't change with seasons and isn't affected by daylight saving. Therefore, Central European Summer Time (CEST) is UTC+2. A local time of 10:00 CEST would be 08:00 UTC.

Syntax

`NOW()`

Returns: A Timestamp corresponding to the date and time at which the query is executed.

Example

In the following example, the output of the call to `NOW` becomes the `expression` parameter value for `DATE_PART`. The result displays the current quarter at the time the query is executed.

```
SELECT
    NOW(),
    DATE_PART('quarter', NOW()) as "current_quarter"
FROM THIS_PROCESS
```

NOW	current_quarter
09/03/2023, 09:19	1

1.6.5 Mathematical Functions

Use these functions to perform a range of mathematical operations.

ABS [page 183]
: Returns the absolute value of an expression. For an expression, `x`, the non-negative value of `x` is returned regardless of its sign.

CEIL [page 184]
: Returns the nearest integer greater than or equal to the provided numeric value by rounding towards positive infinity.

FLOOR [page 186]
: Returns the nearest integer less than or equal to the provided numeric value by rounding towards negative infinity.

LOG [page 188]
: Calculates the exponent of the equation `x = b ^ (y) -> log (b,x) = y` of a numeric expression.

POW [page 188]
: Raises a base value to an exponent power.

ROUND [page 189]
: Returns a rounded value of the provided numeric value.

SIGN [page 192]
Returns the sign of a real number of a numeric expression.

SQRT [page 193]
Calculates the square root of a numeric expression

TRUNC [page 194]
Returns the integer part of the provided numeric value by rounding towards zero.

1.6.5.1 ABS

Returns the absolute value of an expression. For an expression, x, the non-negative value of x is returned regardless of its sign.

Syntax

```
ABS(<expression>)
```

Parameter	Description	Valid Types
`expression`	The value for which you want to calculate the absolute value.	Number

Returns: The absolute value of the argument as a Number.

Example 1

This example shows ABS applied to simple values.

```
SELECT ABS(-3) AS Negative, ABS(4) AS Positive
FROM THIS_PROCESS
```

The resulting table displays the absolute values of the two values:

Negative	Positive
3	4

Example 2

This example finds all changes to order quantities. Since changes can be negative or positive, this query returns only the magnitude of each change.

```
SELECT
    case_id,
    "Order Quantity Changed" AS Change,
    ABS("Order Quantity Changed") AS AbsChange
FROM FLATTEN(THIS_PROCESS)
WHERE event_name = 'Change Order Quantity'
```

Even where order sizes were reduced, those changes can be presented as absolute values.

case_id	Change	AbsChange
00001	1	1
00003	7	7
00004	5	5
00005	-7	7
00005	-9	9

1.6.5.2 CEIL

Returns the nearest integer greater than or equal to the provided numeric value by rounding towards positive infinity.

For example:

- 10.56 is rounded to 11
- -10.56 is rounded to -10

Syntax

```
CEIL(<expression>)
```

Parameter	Description	Valid Types
`expression`	The value you want to round up.	Number

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	944.42
00002	270.04
00003	469.9
00004	268.34
00005	327.94

The following query demonstrates all numeric rounding functions on an attribute.

```
SELECT "case_id",
    "Order Amount",
    CEIL("Order Amount") AS "Ceiling",
    FLOOR("Order Amount") AS "Floor",
    ROUND("Order Amount") AS "Rounded",
    TRUNC("Order Amount") AS "Truncated"
FROM THIS_PROCESS
```

The query returns the following result.

case_id	Order Amount	Ceiling	Floor	Rounded	Truncated
00001	944.42	945	944	944	944
00002	270.04	271	270	270	270
00003	469.9	470	469	470	469
00004	268.34	269	268	268	268
00005	327.94	328	327	328	327

Example 2

The following query demonstrates how each numeric rounding function deals with negative values.

```
SELECT
    -1.5,
    CEIL(-1.5) AS "Ceiling",
    FLOOR(-1.5) AS "Floor",
    ROUND(-1.5) AS "Rounded",
    TRUNC(-1.5) AS "Truncated"
FROM THIS_PROCESS
LIMIT 1
```

The query returns the following result.

-1.5	Ceiling	Floor	Rounded	Truncated
-1.5	-1	-2	-2	-1

Related Information

FLOOR [page 186]
ROUND [page 189]
TRUNC [page 194]

1.6.5.3 FLOOR

Returns the nearest integer less than or equal to the provided numeric value by rounding towards negative infinity.

For example,

- 10.56 is rounded to 10
- -10.56 is rounded to -11

Syntax

```
FLOOR(<expression>)
```

Parameter	Description	Valid Types
`expression`	The value you want to round down.	Number

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	944.42
00002	270.04
00003	469.9
00004	268.34

case_id	Order Amount
00005	327.94

The following query demonstrates all numeric rounding functions on an attribute.

```
SELECT "case_id",
    "Order Amount",
    CEIL("Order Amount") AS "Ceiling",
    FLOOR("Order Amount") AS "Floor",
    ROUND("Order Amount") AS "Rounded",
    TRUNC("Order Amount") AS "Truncated"
FROM THIS_PROCESS
```

The query returns the following result.

case_id	Order Amount	Ceiling	Floor	Rounded	Truncated
00001	944.42	945	944	944	944
00002	270.04	271	270	270	270
00003	469.9	470	469	470	469
00004	268.34	269	268	268	268
00005	327.94	328	327	328	327

Example 2

The following query demonstrates how each numeric rounding function deals with negative values.

```
SELECT
    -1.5,
    CEIL(-1.5) AS "Ceiling",
    FLOOR(-1.5) AS "Floor",
    ROUND(-1.5) AS "Rounded",
    TRUNC(-1.5) AS "Truncated"
FROM THIS_PROCESS
LIMIT 1
```

The query returns the following result.

-1.5	Ceiling	Floor	Rounded	Truncated
-1.5	-1	-2	-2	-1

Related Information

CEIL [page 184]
ROUND [page 189]
TRUNC [page 194]

1.6.5.4 LOG

Calculates the exponent of the equation `x = b ^ (y) -> log (b,x) = y` of a numeric expression.

Syntax

```
LOG(<expression>, <base>)
```

Parameter	Description	Valid Types
`expression`	The value for which the logarithm is calculated.	Number
`base`	The base to which the logarithm is calculated.	Number (literal)

Returns: A Number, the logarithm of the value calculated to the specified base.

Example

```
SELECT
    LOG(10, 1000),
    LOG(3, 27),
    LOG(2, 8)
FROM THIS_PROCESS
```

The table displays the results of the three calculations. In each case, the result is 3 (specifically, 2 ^ *3* = 8, 3 ^ *3* = 27, and 2 ^ *3* = 8).

10 LOG 1000	3 LOG 27	2 LOG 8
3	3	3

1.6.5.5 POW

Raises a base value to an exponent power.

Syntax

```
POW(<base>, <exponent>)
```

Parameter	Description	Valid Types
`base`	The value to be raised.	Number
`exponent`	The exponent power to raise by.	Number (literal)

Returns: A number, the base value raised to the exponent value.

Example

```
SELECT
    POW(9, 2),
    POW(9, 0.5),
    POW(9, 0),
    POW(9, 1)
FROM THIS_PROCESS
```

The table displays the results of the four calculations:

9 POW 2	9 POW 0.5	9 POW 0	9 POW 1
81	3	1	9

1.6.5.6 ROUND

Returns a rounded value of the provided numeric value.

For example:

- 10.45 is rounded to 10
- 10.56 is rounded to 11
- -10.45 is rounded to -10
- -10.5 is rounded to -11

It's also possible to specify the scale at which to perform rounding. You can, for example:

- Round 10.45 to 1 decimal place, yielding 10.5
- Round -10.563 to 2 decimal places, yielding -10.56

> **Note**
>
> The rounding method used is "round half away from zero."

Syntax

```
ROUND(<expression> [, <scale>])
```

Parameter	Description	Valid Types
`expression`	The value you want to round.	Number
`scale`	Optional. The number of digits to include after the decimal point. If a `scale` isn't provided, the default value is 0.	Number **Note** The scale must be an integer. Providing a fractional number produces an error.

Providing a Scale

When rounding a number, you can optionally provide a scale as an integer value. This scale specifies the number of decimal places at which to perform the rounding and how many digits to return.

- When the scale is 0, the expression is rounded to the nearest integer value.
- A positive scale specifies the number of decimal places **after** the decimal point. All digits following this position are removed. For example, `ROUND(17.11111111111111111117, 1)` returns 17.1
- A negative scale specifies the number of decimal places **before** the decimal point. The digit at this position and all subsequent digits become zero. For example, if the scale is -3, then the number is a multiple of 1000.

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	944.42
00002	270.04
00003	469.9
00004	268.34
00005	327.94

The following query demonstrates all numeric rounding functions on an attribute.

```
SELECT "case_id",
    "Order Amount",
    CEIL("Order Amount") AS "Ceiling",
    FLOOR("Order Amount") AS "Floor",
    ROUND("Order Amount") AS "Rounded",
    TRUNC("Order Amount") AS "Truncated"
FROM THIS_PROCESS
```

The query returns the following result.

case_id	Order Amount	Ceiling	Floor	Rounded	Truncated
00001	944.42	945	944	944	944
00002	270.04	271	270	270	270
00003	469.9	470	469	470	469
00004	268.34	269	268	268	268
00005	327.94	328	327	328	327

Example 2

The following query demonstrates how each numeric rounding function deals with negative values.

```
SELECT
    -1.5,
    CEIL(-1.5) AS "Ceiling",
    FLOOR(-1.5) AS "Floor",
    ROUND(-1.5) AS "Rounded",
    TRUNC(-1.5) AS "Truncated"
FROM THIS_PROCESS
LIMIT 1
```

The query returns the following result.

-1.5	Ceiling	Floor	Rounded	Truncated
-1.5	-1	-2	-2	-1

Example 3

Assuming the same process data as the first example, the following query demonstrates supplying an explicit scale.

```
SELECT "case_id",
    "Order Amount",
    ROUND("Order Amount") AS "Default",
    ROUND("Order Amount", 0) AS "0 places",
    ROUND("Order Amount", 1) AS "1 place",
    ROUND("Order Amount", -1) AS "-1 place"
FROM THIS_PROCESS
```

Output:

case_id	Order Amount	Default	0 places	1 place	-1 place
00001	944.42	944	944	944.4	940
00002	270.04	270	270	270	270
00003	469.9	470	470	469.9	470

case_id	Order Amount	Default	0 places	1 place	-1 place
00004	268.34	268	268	268.3	270
00005	327.94	328	328	327.9	330

Related Information

CEIL [page 184]
FLOOR [page 186]
TRUNC [page 194]

1.6.5.7 SIGN

Returns the sign of a real number of a numeric expression.

For a real number x, return:

- -1 if x < 0
- 0 if x = 0
- 1 if x > 0

Syntax

```
SIGN(<expression>)
```

Parameter	Description	Valid Types
`expression`	The value for which you want to extract the sign.	Number

Returns: A Number denoting the sign of the argument.

Example

```
SELECT
    SIGN(10) AS Positive,
    SIGN(0) AS Zero,
    SIGN(-8) AS Negative
FROM THIS_PROCESS
```

The resulting table displays the signs of the three numeric expressions:

Positive	Zero	Negative
1	0	-1

1.6.5.8 SQRT

Calculates the square root of a numeric expression

Syntax

```
SQRT(<expression>)
```

Parameter	Description	Valid Types
`expression`	The value for which you want to calculate the square root.	Number

Returns: The square root of the argument as a Number.

Example 1

This query calculates the square root from the given argument. For expression = 9, the result is 3.

```
SELECT SQRT(9)
FROM THIS_PROCESS
```

Example 2

This query calculates the square root from the order amounts.

```
SELECT SQRT("Order Amount in EUR")
FROM THIS_PROCESS
```

1.6.5.9 TRUNC

Returns the integer part of the provided numeric value by rounding towards zero.

For example:

- 10.45 is rounded to 10
- -10.5 is rounded to -10

Syntax

```
TRUNC(<expression>)
```

Parameter	Description	Valid Types
`expression`	The value you want to truncate.	Number

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	944.42
00002	270.04
00003	469.9
00004	268.34
00005	327.94

The following query demonstrates all numeric rounding functions on an attribute.

```
SELECT "case_id",
    "Order Amount",
    CEIL("Order Amount") AS "Ceiling",
    FLOOR("Order Amount") AS "Floor",
    ROUND("Order Amount") AS "Rounded",
    TRUNC("Order Amount") AS "Truncated"
FROM THIS_PROCESS
```

The query returns the following result.

case_id	Order Amount	Ceiling	Floor	Rounded	Truncated
00001	944.42	945	944	944	944
00002	270.04	271	270	270	270

case_id	Order Amount	Ceiling	Floor	Rounded	Truncated
00003	469.9	470	469	470	469
00004	268.34	269	268	268	268
00005	327.94	328	327	328	327

Example 2

The following query demonstrates how each numeric rounding function deals with negative values.

```
SELECT
    -1.5,
    CEIL(-1.5) AS "Ceiling",
    FLOOR(-1.5) AS "Floor",
    ROUND(-1.5) AS "Rounded",
    TRUNC(-1.5) AS "Truncated"
FROM THIS_PROCESS
LIMIT 1
```

The query returns the following result.

-1.5	Ceiling	Floor	Rounded	Truncated
-1.5	-1	-2	-2	-1

Related Information

CEIL [page 184]
FLOOR [page 186]
ROUND [page 189]

1.6.6 String Functions

Use string functions to manipulate strings in queries.

CHAR_INDEX [page 196]
: Returns the starting position of the first occurrence of a string within a second string.

CHAR_LENGTH [page 199]
: Returns the number of Unicode characters contained in an input string.

CONCAT [page 200]
: Combines two strings into a single string value by concatenating them.

LEFT [page 202]
: Returns a specified number of the leftmost Unicode characters in a string.

LOWER [page 203]
Converts all upper case characters in an input string to lower case characters.

LTRIM [page 204]
Removes all leading whitespaces from the input string.

REPLACE [page 206]
Searches a string, replacing all occurrences of a specified substring with an alternative string.

REVERSE [page 207]
Returns a new string with the order of all characters from the input string reversed.

RIGHT [page 209]
Returns a specified number of the rightmost Unicode characters in a string.

RTRIM [page 210]
Removes all trailing whitespaces from the input string.

SUBSTRING [page 212]
Extracts from a string a specified number of Unicode characters beginning from a given start position.

SUBSTRING_AFTER [page 213]
Extracts from a string all Unicode characters occurring after a delimiter string.

SUBSTRING_BEFORE [page 215]
Extracts from a string all Unicode characters occurring before a delimiter string.

TRIM [page 217]
Removes all leading and trailing whitespaces from the input string.

UPPER [page 218]
Converts all lower case characters in an input string to upper case characters.

1.6.6.1 CHAR_INDEX

Returns the starting position of the first occurrence of a string within a second string.

Syntax

```
CHAR_INDEX(<string>, <searchString>)
```

Parameter	Description	Valid Types
`string`	A string expression. The string to be searched. If this expression is `NULL`, the function returns `NULL`.	String

Parameter	Description	Valid Types
`searchString`	The string to be located in `string`. If this expression is `NULL`, the function returns `NULL`.	String

Returns: A number, specifically:

- if `searchString` appears in `string`, then this function returns the starting index of the first occurrence.
- if `searchString` doesn't appear in `string`, then this function returns 0.

> **Note**
>
> - The indexing is 1-based, meaning the first character in `stringExpression` occupies character index 1.
> - The indexing counts Unicode characters.
> - The function is case-sensitive. For example, the string 'case' wouldn't be found in the string 'UPPERCASE'.

Example 1

This example shows several invocations of the `CHAR_INDEX` function, demonstrating:

- Finding the only occurrence of a string: In this case, '`York`' is searched for within '`New York`'. Since the 'Y' in '`York`' occupies the fifth position in the string, the function returns 5.
- Finding the first of many occurrences of a string: In this case, '`iss`' is searched for within '`Mississippi`'. Even though '`iss`' occurs twice, it first appears at index 2, and so the function returns 2.
- Returning 0 when failing to find a string: In this case, the function fails to find '`San`' inside '`Los Angeles`' and so returns 0.
- Returning `NULL` when a parameter is `NULL`: In this case, '`Anywhere`' is searched for within `NULL`, which can't be searched, so the function returns `NULL`.

```
SELECT
    CHAR_INDEX('New York', 'York') AS "New YORK",
    CHAR_INDEX('Mississippi', 'iss') AS "MISSissippi",
    CHAR_INDEX('Los Angeles', 'San') AS "Los Angeles",
    CHAR_INDEX(NULL, 'Anywhere') AS "Nowhere"
FROM THIS_PROCESS
```

Output:

New YORK	MISSissippi	Los Angeles	Nowhere
5	2	0	null

Example 2

In this example, there are six types of goods for sale. We can think of these six types as being three garment types – cappy, hoody and t-shirt – each available in two varieties: with and without a print. This query outputs the types and the number sold.

```
SELECT
    "Type of Goods",
    COUNT("Type of Goods") AS "Number Sold"
FROM THIS_PROCESS
```

Output:

Type of Goods	Number Sold
Cappy with Print	425
Hoody with Print	372
T-shirt with Print	532
T-shirt	589
Cappy	1102
Hoody	264

Let's say we want to ignore the distinction between varieties and instead count the number sold per garment type.

The following query removes the first occurrence of a space character along with all text that follows, leaving only the first word. The condition in the first IF function uses `CHAR_INDEX` to determine whether the field includes a space character. If it doesn't, then the "`Type of Goods`" field is returned as is. Otherwise, a substring of that field is extracted, specifically between the first character and the index of the first occurring space. The index is decremented by 1 so that the space character isn't included in the substring.

```
SELECT
    IF(
        -- Checks whether a whitespace is present inside the "Type of Goods"
field.
        CHAR_INDEX("Type of Goods", ' ') = 0,
        -- If not, CHAR_INDEX returns 0, meaning "Type of Goods" is selected
unaltered.
        "Type of Goods",
        -- Otherwise, a substring of the "Type of Goods" field is returned.
        SUBSTRING(
            -- The substring begins at the first letter.
            "Type of Goods", 1,
            -- The substring ends one character before the first whitespace.
            IF(
                CHAR_INDEX("Type of Goods", ' ') > 0,
                CHAR_INDEX("Type of Goods", ' ') - 1,
                -- If neither whitespace (first IF statement) nor strings in
general
                -- are found, then return NULL.
                NULL
            )
        )
    ) AS "Type of Garment",
    COUNT("Type of Goods") AS "Number Sold"
FROM THIS_PROCESS
```

Output:

Garment Type	Number Sold
T-shirt	1121
Hoody	636
Cappy	1527

Related Information

SUBSTRING [page 212]
SUBSTRING_AFTER [page 213]
SUBSTRING_BEFORE [page 215]

1.6.6.2 CHAR_LENGTH

Returns the number of Unicode characters contained in an input string.

Syntax

```
CHAR_LENGTH(<string>)
```

Parameter	Description	Valid Types
`string`	A string expression.	String

Returns: The number of Unicode characters in the string. If the string is `NULL`, the function returns a `NULL` value.

Example

This query displays the names of all cities in the data (`THIS_PROCESS`) along with the number of Unicode characters in those names, including spaces.

```
SELECT
    "City",
    CHAR_LENGTH("City") AS "Length"
FROM THIS_PROCESS
GROUP BY 1
```

Output:

City	Length
Boston	6
New York	8
San Francisco	13
Houston	7
Washington	10
Miami	5

1.6.6.3 CONCAT

Combines two strings into a single string value by concatenating them.

`CONCAT` can only be applied to non-nested attributes. To use `CONCAT` on nested attributes, use the `FLATTEN` operator. Alternatively, use a nested query that returns a single string attribute as a result.

Syntax

```
CONCAT(<expression1>, <expression2>)
```

Parameter	Description	Valid Types
`expression1`	An expression, literal or non-nested attribute.	String
`expression2`	An expression, literal or non-nested attribute.	String

Returns: A String, the result of appending `expression2` to `expression1`.

Alternative syntax, which allows concatenation of an arbitrary number of strings:

```
<expression1> || <expression2> [ || <expressionN> ... ]
```

Example (Non-nested Attributes)

This query concatenates a dash to the name of every event which features the string '`ship`'. It then uses the output of that call as a parameter to a second call to `CONCAT`, appending the carrier name.

```
SELECT
    COUNT(DISTINCT case_id) AS "Case Count", event_name,
    "Shipment Carrier",
```

```
    CONCAT(CONCAT(event_name, ' - '), "Shipment Carrier") AS "Shipment Type and
Provider"
FROM FLATTEN(THIS_PROCESS)
WHERE event_name ILIKE '%ship%'
```

Output:

event_name	Shipment Carrier	Shipment Type and Provider	Case Count
Ship Goods Standard	UPS	Ship Goods Standard - UPS	140
	DHL	Ship Goods Standard - DHL	1,537
Ship Goods Express	DHL	Ship Goods Express - DHL	31
	UPS	Ship Goods Express - UPS	1,379

Example (Nested Attributes)

This query selects a field from a nested attribute, using a subquery to select the first event name. Using the alternate syntax for concatenation, it appends to this event name the case ID.

```
SELECT
    case_id,
    (SELECT FIRST(event_name)) AS "Nested Attribute: Event_Name",
    (SELECT FIRST(event_name)) || ' ' || case_id AS "Nested Attribute
Adjustment: Event_Name + Case_Id"
FROM THIS_PROCESS
```

Output:

case_id	Nested Attribute: Event_Name	Nested Attribute Adjustment: Event_Name + Case_Id
1	Receive Customer Order	Receive Customer Order 00001
2	Receive Customer Order	Receive Customer Order 00002
3	Receive Customer Order	Receive Customer Order 00003
4	Receive Customer Order	Receive Customer Order 00004
5	Receive Customer Order	Receive Customer Order 00005

1.6.6.4 LEFT

Returns a specified number of the leftmost Unicode characters in a string.

Syntax

```
LEFT(<string>, <numberOfCharacters>)
```

Parameter	Description	Valid Types
`string`	A string expression. If the value is `NULL`, the function returns `NULL`.	String
`numberOfCharacters`	The number of leftmost characters to return. The value must be a positive integer. Negative or fractional values result in an error.	Number

Returns: The specified number of leftmost characters from the `string` argument. If the value exceeds the length of `string`, then this function returns the complete string.

Example

This query displays the names of all cities in the data (`THIS_PROCESS`) as well as the six leftmost characters of each name. Names of length less than six are displayed in their entirety.

```
SELECT
    "City",
    LEFT("City", 6) AS "Leftmost6"
FROM THIS_PROCESS
GROUP BY 1
```

Output:

City	Leftmost6
Boston	Boston
New York	New Yo
San Francisco	San Fr
Houston	Housto

City	Leftmost6
Washington	Washin
Miami	Miami

Related Information

RIGHT [page 209]
SUBSTRING [page 212]

1.6.6.5 LOWER

Converts all upper case characters in an input string to lower case characters.

Syntax

```
LOWER(<string>)
```

Parameter	Description	Valid Types
`string`	The string whose upper case characters should be converted to lower case. If the value is `NULL`, this function returns `NULL`.	String

Returns: A new string with the same value as the `string` parameter but with all upper case characters converted to lower case.

Example

Assuming the following process data (`THIS_PROCESS`):

Type of Goods
Hoody
Cappy with Print
T-shirt

Type of Goods
Cappy
Cappy with Print

This query demonstrates converting all upper case characters in the "`Type of Goods`" attribute to lower case:

```
SELECT "Type of Goods",
    LOWER("Type of Goods") AS "Lower Case"
FROM THIS_PROCESS
```

Type of Goods	Lower Case
Hoody	hoody
Cappy with Print	cappy with print
T-shirt	t-shirt
Cappy	cappy
Cappy with Print	cappy with print

Related Information

UPPER [page 218]

1.6.6.6 LTRIM

Removes all leading whitespaces from the input string.

The definition of a whitespace character follows the Unicode Character Database definition and includes characters such as:

- " " (space)
- "\t" (tab)
- "\n" (newline)
- "\r" (carriage return)
- "\x0b" (line tabulation)
- "\x0c (form feed)
- "\xa0" (non-breaking space)

Syntax

```
LTRIM(<string>)
```

Parameter	Description	Valid Types
`string`	The string to be trimmed. If the value is `NULL`, then this function returns `NULL`.	String

Returns: A new string with the same value as the `string` parameter but with all leading whitespace characters removed.

Related Functions

The following functions provide additional means for trimming strings:

- `RTRIM`: Removes all trailing whitespace characters.
- `TRIM`: Removes all leading and trailing whitespace characters.

Example

> **Note**
>
> The data in this example has been rendered so as to make all whitespace characters explicit.

Assuming the following process data (`THIS_PROCESS`):

```
| Type of Goods        |
|----------------------|
| "Cappy"              |
| "  T-shirt\t"        |
| "Cappy with   Print" |
| "Hoody \r\n"         |
```

The following query demonstrates the functions available for trimming:

```
SELECT "Type of Goods",
    TRIM("Type of Goods") AS "TRIM",
    LTRIM("Type of Goods") AS "LTRIM",
    RTRIM("Type of Goods") AS "RTRIM"
FROM THIS_PROCESS
```

Output:

```
| Type of Goods        | LTRIM                | RTRIM                |
TRIM                 |
|----------------------|----------------------|----------------------|------------
---------|
| "Cappy"              | "Cappy"              | "Cappy"              |
"Cappy"              |
| "  T-shirt\t"        | "T-shirt\t"          | "  T-shirt"          | "T-
shirt"           |
| "Cappy with   Print" | "Cappy with   Print" | "Cappy with   Print" | "Cappy
with   Print" |
| "Hoody \r\n"         | "Hoody \r\n"         | "Hoody"              |
"Hoody"              |
```

Related Information

RTRIM [page 210]
TRIM [page 217]

1.6.6.7 REPLACE

Searches a string, replacing all occurrences of a specified substring with an alternative string.

REPLACE can only be applied to non-nested attributes. To use REPLACE on nested attributes, use the FLATTEN operator. Alternatively, use a nested query that returns a single string attribute as a result.

Syntax

```
REPLACE(<sourceExpression>, <searchExpression>, <replacementExpression>)
```

Parameter	Description	Valid Types
`sourceExpression`	The string to be searched. An expression or non-nested attribute.	String
`searchExpression`	The substring to be replaced. An expression or literal value.	String
`replacementExpression`	The string which replaces the substring. An expression or literal value.	String

Returns: A String, the result of replacing in `sourceExpression` all incidences of `searchExpression` with `replacementExpression`.

Example (Non-nested Attributes)

```
SELECT
    DISTINCT "Type of Goods",
    REPLACE("Type of Goods", 'Cappy', 'Cap') AS "Single Text Adjustment
(Literal): Cappy->Cap",
    REPLACE("Type of Goods", 'w', 'W') AS "Multiple Text Adjustment (Literal):
w->W"
FROM FLATTEN(THIS_PROCESS)
```

Output:

Type of Goods	Single Text Adjustment (Literal): Cappy->Cap	Multiple Text Adjustment (Literal): w->W
Hoody	Hoody	Hoody
T-shirt	T-shirt	T-shirt
Cappy with Print	Cap with Print	Cappy With Print
Hoody with Print	Hoody with Print	Hoody With Print
T-shirt with Print	T-shirt with Print	T-shirt With Print
Cappy	Cap	Cappy

Example (Nested Attributes)

```
SELECT
    case_id,
    (SELECT FIRST(event_name)) AS "Nested Attribute: Event_Name",
    REPLACE((SELECT FIRST(event_name)), 'Customer','Cust') AS "Nested Attribute
Adjustment: Customer->Cust"
FROM THIS_PROCESS
```

Output:

case_id	Nested Attribute: Event_Name	Nested Attribute Adjustment: Customer->Cust
00001	Receive Customer Order	Receive Cust Order
00002	Receive Customer Order	Receive Cust Order
00003	Receive Customer Order	Receive Cust Order
00004	Receive Customer Order	Receive Cust Order
00005	Receive Customer Order	Receive Cust Order

1.6.6.8 REVERSE

Returns a new string with the order of all characters from the input string reversed.

Syntax

```
REVERSE(<string>)
```

Parameter	Description	Valid Types
`string`	A string expression to be reversed. If the value is `NULL`, this function returns `NULL`.	String

Returns: A string containing all characters from the `string` parameter in reverse order.

Example

Assuming the following process data (`THIS_PROCESS`):

Type of Goods
Hoody
Cappy with Print
T-shirt
Cappy
Cappy with Print

This query demonstrates reversing the "`Type of Goods`" attribute in every row:

```
SELECT "Type of Goods",
    REVERSE("Type of Goods") AS "Reversed"
FROM THIS_PROCESS
```

Output:

Type of Goods	Reversed
Hoody	ydooH
Cappy with Print	tnirP htiw yppaC
T-shirt	trihs-T
Cappy	yppaC
Cappy with Print	tnirP htiw yppaC

1.6.6.9 RIGHT

Returns a specified number of the rightmost Unicode characters in a string.

Syntax

```
RIGHT(<string>, <numberOfCharacters>)
```

Parameter	Description	Valid Types
`string`	A string expression. If the value is `NULL`, the function returns `NULL`.	String
`numberOfCharacters`	The number of rightmost characters to return. The value must be a positive integer. Negative or fractional values result in an error.	Number

Returns: The specified number of rightmost characters from the `string` argument. If the value exceeds the length of `string`, then this function returns the complete string.

Example

This query displays the names of all cities in the data (`THIS_PROCESS`) as well as the six rightmost characters of each name. Names of length less than six are displayed in their entirety.

```
SELECT
    "City",
    RIGHT("City", 6) AS "Rightmost6"
FROM THIS_PROCESS
GROUP BY 1
```

Output:

City	Rightmost6
Boston	Boston
New York	w York
San Francisco	ncisco
Houston	ouston

City	Rightmost6
Washington	ington
Miami	Miami

Related Information

LEFT [page 202]
SUBSTRING [page 212]

1.6.6.10 RTRIM

Removes all trailing whitespaces from the input string.

The definition of a whitespace character follows the Unicode Character Database definition and includes characters such as:

- " " (space)
- "\t" (tab)
- "\n" (newline)
- "\r" (carriage return)
- "\x0b" (line tabulation)
- "\x0c (form feed)
- "\xa0" (non-breaking space)

Syntax

```
RTRIM(<string>)
```

Parameter	Description	Valid Types
`string`	The string to be trimmed. If the value is `NULL`, then this function returns `NULL`.	String

Returns: A new string with the same value as the `string` parameter but with all trailing whitespace characters removed.

Related Functions

The following functions provide additional means for trimming strings:

- `LTRIM`: Removes all leading whitespace characters.
- `TRIM`: Removes all leading and trailing whitespace characters.

Example

> **Note**
>
> The data in this example has been rendered so as to make all whitespace characters explicit.

Assuming the following process data (`THIS_PROCESS`):

```
| Type of Goods        |
|----------------------|
| "Cappy"              |
| "  T-shirt\t"        |
| "Cappy with   Print" |
| "Hoody \r\n"         |
```

The following query demonstrates the functions available for trimming:

```
SELECT "Type of Goods",
    TRIM("Type of Goods") AS "TRIM",
    LTRIM("Type of Goods") AS "LTRIM",
    RTRIM("Type of Goods") AS "RTRIM"
FROM THIS_PROCESS
```

Output:

```
| Type of Goods        | LTRIM                | RTRIM                |
TRIM                 |
|----------------------|----------------------|----------------------|-------------
---------|
| "Cappy"              | "Cappy"              | "Cappy"              |
"Cappy"              |
| "  T-shirt\t"        | "T-shirt\t"          | "  T-shirt"          | "T-
shirt"            |
| "Cappy with   Print" | "Cappy with   Print" | "Cappy with   Print" | "Cappy
with   Print" |
| "Hoody \r\n"         | "Hoody \r\n"         | "Hoody"              |
"Hoody"              |
```

Related Information

LTRIM [page 204]
TRIM [page 217]

1.6.6.11 SUBSTRING

Extracts from a string a specified number of Unicode characters beginning from a given start position.

Syntax

```
SUBSTRING(<string>, <startPosition>, <numberOfCharacters>)
```

Parameter	Description	Valid Types
`string`	A string expression from which the subset of characters is extracted. The value of the string isn't altered.	String
`startPosition`	The position from which to begin the extraction. The index of the first character is 1.	Number
`numberOfCharacters`	The number of characters to extract.	Number

Returns: A string containing a subset of characters from the `string` parameter beginning at index `startPosition` and with a length of up to `numberOfCharacters`. Exceptions to this include:

- If `string` is `NULL`, this function returns `NULL`.
- If `startPosition` is greater than the length of `string`, this function returns an empty string.
- If `numberOfCharacters` exceeds the length of `string`, this function returns the entire substring after `startPosition`.

SUBSTRING_BEFORE and SUBSTRING_AFTER

SIGNAL provides alternative means of obtaining a substring when delimiters are involved.

A common use case regarding substrings is the extraction of characters occurring before or after a delimiter or separator string. For example, let's say your process data contains a 'Department' column containing labels of the form "`DEPT_IT`", "`DEPT_HR`", "`DEPT_FINANCE`" and so on. From these labels, you wish to extract the characters following but not including the underscore.

You could achieve this by combining `SUBSTRING`, `CHAR_INDEX`, and `CHAR_LENGTH`:

```
SUBSTRING("Department", CHAR_INDEX("Department", '_') + 1,
CHAR_LENGTH("Department"))
```

However, SIGNAL provides a pair of functions making queries like this more convenient to write:

```
SUBSTRING_BEFORE(<string>, <searchString>)
SUBSTRING_AFTER(<string>, <searchString>)
```

These functions work by searching for the first occurrence of `searchString` within `string` and then returning all characters before (or after) that occurrence. The example above could instead be written like so:

```
SUBSTRING_AFTER("Department", '_')
```

Example

In this example, customer IDs have the format 'C_nnnnn'. This query selects only the numeric part of the ID by extracting the five characters from index position three onwards.

```
SELECT
    "Customer ID",
    SUBSTRING("Customer ID", 3, 5) AS "Number Part"
FROM THIS_PROCESS
```

Output:

Customer ID	Number Part
C_00218	218
C_00693	693
C_00655	655
C_00914	914
C_00115	115

Related Information

CHAR_INDEX [page 196]
SUBSTRING_AFTER [page 213]
SUBSTRING_BEFORE [page 215]

1.6.6.12 SUBSTRING_AFTER

Extracts from a string all Unicode characters occurring after a delimiter string.

Syntax

```
SUBSTRING_AFTER(<string>, <searchString>)
```

Parameter	Description	Valid Types
`string`	A string expression from which the subset of characters is extracted. The value of the string isn't altered.	String
`searchString`	A string expression that is searched for in the `string` parameter. **Note** If `searchString` occurs multiple times in `string`, the first occurrence is taken.	String

Returns: A string containing all characters from the `string` parameter that occur after `searchString`. Exceptions include:

- If the `string` parameter is `NULL`, the function returns `NULL`.
- If `searchString` doesn't occur, this function returns `NULL`.

Example 1

This example shows several invocations of `SUBSTRING_AFTER`, demonstrating:

- Returning a substring from a string containing one delimiter.
- Returning a substring from a string containing many delimiters.
- Returning `NULL` when a search string isn't found.
- Returning `NULL` when the input string is `NULL`.

```
SELECT
    SUBSTRING_AFTER('DEPT_FINANCE', '_') AS "Finance Dept.",
    SUBSTRING_AFTER('DEPT_SALES_MARKETING', '_') AS "Sales & Marketing Dept.",
    SUBSTRING_AFTER('DEPT_HR', '-') AS "Search String Not Found",
    SUBSTRING_AFTER(NULL, '_') AS "Input String NULL"
FROM THIS_PROCESS
```

Output:

Finance Dept.	Sales & Marketing Dept.	Search String Not Found	Input String NULL
FINANCE	SALES_MARKETING	null	null

Related Information

SUBSTRING [page 212]

1.6.6.13 SUBSTRING_BEFORE

Extracts from a string all Unicode characters occurring before a delimiter string.

Syntax

```
SUBSTRING_BEFORE(<string>, <searchString>)
```

Parameter	Description	Valid Types
`string`	A string expression from which the subset of characters is extracted. The value of the string isn't altered.	String
`searchString`	A string expression that is searched for in the `string` parameter. **Note** If `searchString` occurs multiple times in `string`, the first occurrence is taken.	String

Returns: A string containing all characters from the `string` parameter that occur before `searchString`. Exceptions include:

- If the `string` parameter is `NULL`, the function returns `NULL`.
- If `searchString` doesn't occur, this function returns `NULL`.

Example 1

This example shows several invocations of `SUBSTRING_BEFORE`, demonstrating:

- Returning a substring from a string containing one delimiter.
- Returning a substring from a string containing many delimiters.
- Returning `NULL` when a search string isn't found.
- Returning `NULL` when the input string is `NULL`.

```
SELECT
    SUBSTRING_BEFORE('FINANCE_DEPT', '_') AS "Finance Dept.",
    SUBSTRING_BEFORE('SALES_MARKETING_DEPT', '_') AS "Sales & Marketing Dept.",
    SUBSTRING_BEFORE('HR_DEPT', '-') AS "Search String Not Found",
    SUBSTRING_BEFORE(NULL, '_') AS "Input String NULL"
FROM THIS_PROCESS
```

Output:

Finance Dept.	Sales & Marketing Dept.	Search String Not Found	Input String NULL
FINANCE	SALES	null	null

Example 2

In this example, there are six types of goods for sale. We can think of these six types as being three garment types – cappy, hoody and t-shirt – each available in two varieties: with and without a print. This query outputs the types and the number sold.

```
SELECT
    "Type of Goods",
    COUNT("Type of Goods") AS "Number Sold"
FROM THIS_PROCESS
```

Output:

Type of Goods	Number Sold
Cappy with Print	425
Hoody with Print	372
T-shirt with Print	532
T-shirt	589
Cappy	1102
Hoody	264

Let's say we want to ignore the distinction between varieties and instead count the number sold per garment type.

The following query ignores all text following the first space, if a space exists.

```
SELECT
    IF(
        SUBSTRING_BEFORE("Type of Goods", ' ') IS NOT NULL,
        SUBSTRING_BEFORE("Type of Goods", ' '),
        "Type of Goods"
    ) AS "Garment Type",
    COUNT("Type of Goods") AS "Number Sold"
FROM THIS_PROCESS
```

Output:

Garment Type	Number Sold
T-shirt	1121
Hoody	636
Cappy	1527

Related Information

SUBSTRING [page 212]

1.6.6.14 TRIM

Removes all leading and trailing whitespaces from the input string.

The definition of a whitespace character follows the Unicode Character Database definition and includes characters such as:

- " " (space)
- "\t" (tab)
- "\n" (newline)
- "\r" (carriage return)
- "\x0b" (line tabulation)
- "\x0c (form feed)
- "\xa0" (non-breaking space)

Syntax

```
TRIM(<string>)
```

Parameter	Description	Valid Types
`string`	The string to be trimmed. If the value is `NULL`, then this function returns `NULL`.	String

Returns: A new string with the same value as the `string` parameter but with all leading and trailing whitespace characters removed.

Related Functions

The following functions provide additional means for trimming strings:

- `LTRIM`: Removes all leading whitespace characters.
- `RTRIM`: Removes all trailing whitespace characters.

Example

> **Note**
>
> The data in this example has been rendered so as to make all whitespace characters explicit.

Assuming the following process data (`THIS_PROCESS`):

```
| Type of Goods        |
|----------------------|
| "Cappy"              |
| "  T-shirt\t"        |
| "Cappy with   Print" |
| "Hoody \r\n"         |
```

The following query demonstrates the functions available for trimming:

```
SELECT "Type of Goods",
    TRIM("Type of Goods") AS "TRIM",
    LTRIM("Type of Goods") AS "LTRIM",
    RTRIM("Type of Goods") AS "RTRIM"
FROM THIS_PROCESS
```

Output:

```
| Type of Goods        | LTRIM                | RTRIM                |
TRIM                |
|----------------------|----------------------|----------------------|-------------
---------|
| "Cappy"              | "Cappy"              | "Cappy"              |
"Cappy"             |
| "  T-shirt\t"        | "T-shirt\t"          | "  T-shirt"          | "T-
shirt"           |
| "Cappy with   Print" | "Cappy with   Print" | "Cappy with   Print" | "Cappy
with   Print" |
| "Hoody \r\n"         | "Hoody \r\n"         | "Hoody"              |
"Hoody"             |
```

Related Information

LTRIM [page 204]
RTRIM [page 210]

1.6.6.15 UPPER

Converts all lower case characters in an input string to upper case characters.

Syntax

```
UPPER(<string>)
```

Parameter	Description	Valid Types
`string`	The string whose lower case characters should be converted to upper case. If the value is `NULL`, this function returns `NULL`.	String

Returns: A new string with the same value as the `string` parameter but with all lower case characters converted to upper case.

Example

Assuming the following process data (`THIS_PROCESS`):

Type of Goods	Lower Case
Hoody	hoody
Cappy with Print	cappy with print
T-shirt	t-shirt
Cappy	cappy
Cappy with Print	cappy with print

This query demonstrates converting all lower case characters in the "`Type of Goods`" attribute to upper case:

```
SELECT "Type of Goods",
    UPPER("Type of Goods") AS "Upper Case"
FROM THIS_PROCESS
```

Type of Goods	Upper Case
Hoody	HOODY
Cappy with Print	CAPPY WITH PRINT
T-shirt	T-SHIRT
Cappy	CAPPY
Cappy with Print	CAPPY WITH PRINT

Related Information

LOWER [page 203]

1.6.7 Window Functions

Window functions perform calculations on a window, which is a subset of rows that are defined as being related.

During the execution of a query, a window function progresses through a data set, applying the function to the current row and all other rows in the current row's window. A window can have zero, one, or multiple rows.

The general form of a window function is:

```
<function> OVER (
    [ PARTITION BY <partitionExpressions> ]
    [ ORDER BY <orderExpressions> [ <windowFrame> ] ]
)
```

Element	Description
`<function>`	The function applied to the current row and all rows in the current row's window.
`PARTITION BY <partitionExpressions>`	Groups a data set into partitions. This allows you to define multiple windows. The function calculates values only within the current window. Learn more in Window Partition [page 221].
`ORDER BY <orderExpressions>`	Sorts the data set into ascending or descending order. When the data set is partitioned into multiple windows, each window is sorted separately. Learn more in Window Sort Order [page 223].
`<windowFrame>`	Defines a subset set of rows in the current window, restricting further which rows are related to the current row. The functon calculates values only within the current window frame. Learn more in Window Frame [page 224] **Note** If no window frame is specified explicitly, then the function is applied over the entire window.

Considerations

When using window functions in SIGNAL queries, keep the following in mind:

- Window functions are evaluated after aggregate functions in a `SELECT` clause.
- Window functions can only occur in `SELECT` clauses and on flat data. Nested data structures aren't supported.

> → Tip
>
> If you want to use window functions on event level, you can use the `FLATTEN` operator to flatten the table and then partition by `case_id`.

- In `ORDER BY` clauses, only use expressions such as column names. Don't use numeric values for indices.

Functions

Several categories of function can be used as window functions:

- A number of aggregate functions support the use of windows. For more information, see Window-Supporting Aggregate Functions [page 229].
- Non-Aggregate Functions [page 271] don't aggregate collections of values into a single value, but are instead applied to individual rows within a window.
- Ranking Functions [page 280] sort data partitions and assign a rank value to each row within a result set.

1.6.7.1 Window Partition

Use partitioning to split a data set into multiple windows based on one or more attributes. Each partitioned window includes only rows whose attributes share a common value.

The `PARTITION BY` clause allows you to perform the grouping.

Syntax

```
PARTITION BY <expression> [, <expression> ... ]
```

Using Partitions

Consider the follow example data set:

case_id	Customer ID	city	Order Amount
00001	1	Boston	100
00002	2	Boston	250
00003	4	Miami	1000
00004	8	Miami	240

To this data set, we'll apply the following query:

```
SELECT "Customer ID",
    city,
    SUM("Order Amount") OVER() AS "Total Sales",
    SUM("Order Amount") OVER(PARTITION BY city) AS "Total Sales per City"
FROM THIS_PROCESS
ORDER BY 2
```

The query selects using two window functions:

- The first instance (column number 3) performs no partitioning.
- The second window function (column number 4) partitions the data set into windows based on the city. This groups together rows sharing the same city.

Result:

Customer ID	city	Total Sales	Total Sales per City
1	Boston	1590	350
2	Boston	1590	350
4	Miami	1590	1240
8	Miami	1590	1240

The '`Total Sales`' column, which doesn't partition the data, applies one window over the entire data set. The total in each row is therefore the sum of **all order amounts in the data set**.

The '`Total Sales per City`' column partitions the data into as many windows as there are distinct cities, in this case two. The total in each row is therefore the sum of **all order amounts in the same window as the current row**.

Example

Consider the following process data (`THIS_PROCESS`):

City	Value
Berlin	1000
Berlin	1800
Paris	3000
London	2500
Paris	1500
London	1200
Berlin	1300

And consider the following query, which groups rows into partitions based on the city:

```
SELECT city,
    SUM(value) OVER (PARTITION BY city)
```

```
FROM THIS_PROCESS
```

A grouped sum is returned:

City	Value
Berlin	4100
Berlin	4100
Berlin	4100
Paris	4500
Paris	4500
London	3700
London	3700

1.6.7.2 Window Sort Order

In a window function, you can sort the rows included within the current window. Use the `ORDER BY` clause to sort the data within a window into ascending or descending order.

If the data set has been partitioned into multiple windows, each window is sorted separately.

When sorting, you can further restrict which rows are related to the current row by using a window frame [page 224]. Otherwise all rows in the current window are considered as related.

Syntax

```
ORDER BY <expression> [ ASC | DESC ] [, <expression>, ...] [ NULLS FIRST | NULLS LAST ] [ <windowFrame> ]
```

Example

Consider the following example data (`THIS_PROCESS`):

City	Value
Berlin	1000
Paris	3000
London	2500
Rome	1500

And consider the following query:

```
SELECT city,
    SUM(value) OVER (ORDER BY value ASC)
FROM THIS_PROCESS
```

This query returns the cumulative sum:

City	Value
Berlin	1000
Rome	2500
London	5000
Paris	8000

1.6.7.3 Window Frame

A window frame is a subset of rows from the current window.

You can define the window frame by using the `ROW` and `RANGE` modes.

> ⓘ Note
>
> If no frame is specified explicitly, then the window function is applied over the entire window.

Syntax

```
{ RANGE | ROWS } BETWEEN <startFrame> AND <endFrame>
```

Types of Window Frame

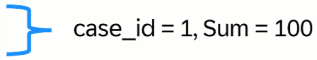
Two types of window frame are available:

- Cumulative
- Sliding

A cumulative window frame starts at a fixed row in the data set and grows with each row. For example, the following query applies a cumulative window frame when calculating a sum of the order amount (the window frame definition is emphasized):

```
SELECT case_id,
    "Order Amount",
    SUM("Order Amount") OVER (ORDER BY case_id ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW)
FROM THIS_PROCESS
```

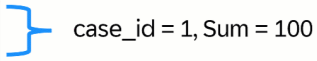
case_id	Order Amount	Sum
1	100	100
2	200	300
3	400	700
4	800	1500



A sliding window frame has a fixed range that slides across the window with each row. The following query also sums order amounts, but does so across a sliding window frame:

```
SELECT case_id,
    "Order Amount",
    SUM("Order Amount") OVER (ORDER BY case_id ROWS BETWEEN 1 PRECEDING AND
CURRENT ROW)
FROM THIS_PROCESS
```

case_id	Order Amount	Sum
1	100	100
2	200	300
3	400	600
4	800	1200



Defining a Start and End

The boundaries of a window frame are marked by a frame start and a frame end. You can define the start of a frame using the following keywords:

Keyword	Description
`CURRENT ROW`	Starts the frame with the current row.
`UNBOUNDED PRECEDING`	Starts the frame with the first row of the peer group within the partition.
`<offset> PRECEDING`	Starts the frame the specified number of rows before the current row. An `offset` of 0 refers to the current row.

You can define the end of a frame using the following keywords:

Keyword	Description
`CURRENT ROW`	Ends the frame with the current row.
`UNBOUNDED FOLLOWING`	Ends the frame with the last row of the peer group within the partition.
`<offset> FOLLOWING`	Ends the frame with the specified number of rows after the current row. An `offset` of 0 refers to the current row.

The offset expression's data type can vary depending on the data type of the ordering column.

- If you use numeric ordering columns, the offset's type is the same as the ordering column.
- If the ordering column is a timestamp, the offset must be a non-null and non-negative value. For example: `RANGE BETWEEN DURATION '1 day' PRECEDING AND DURATION '10 days' FOLLOWING`

> → Remember
>
> The frame end can't precede the frame start. For example:
>
> - `ROWS BETWEEN 1 PRECEDING AND 2 PRECEDING`
> - `ROWS BETWEEN 1 FOLLOWING AND 1 PRECEDING`
> - `ROWS BETWEEN 1 FOLLOWING AND CURRENT ROW`
>
> All these examples are invalid because the right boundary precedes the left boundary.

ROWS and RANGE

When using the `ROWS` mode, the window frame is defined as a number of rows preceding or following the current row.

> ⚠ Caution
>
> If no `ORDER BY` clause is provided when using `ROWS`, the returned results are undefined and the order in which the rows are processed isn't uniform.

Alternatively, the `RANGE` mode defines a window frame by including all rows whose value falls within a specified range of the current row's value, the value being the attribute chosen to sort the window frame. This means that the `ORDER BY` clause is required when using `RANGE`. Only numeric or timestamp type attributes can be used in `RANGE`.

> → Tip
>
> `RANGE` is useful when working with time series or when there are many gaps or duplicates in your tables.

Example 1

Consider the following input data (`THIS_PROCESS`):

City	Value
Berlin	1000
Paris	3000
London	2500
Paris	1500

And consider the following query, which uses a sliding window three values long consisting of the current row's value, plus the individual rows immediately before and after:

```
SELECT city,
    SUM(value) OVER (ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
FROM THIS_PROCESS
```

Output:

City	Value
Berlin	4000
Paris	6500
London	7000
Paris	4000

Example 2

Consider the following input data (`THIS_PROCESS`):

City	Value
Berlin	1000
Paris	3000
London	3000
Paris	2000

And consider the following query, which uses a window consisting of all rows whose value is either 1000 less than or 1000 more than that of the current row:

```
SELECT city,
    SUM(value) OVER (ORDER BY value RANGE BETWEEN 1000 PRECEDING AND 1000
FOLLOWING)
FROM THIS_PROCESS
```

Result:

City	Value
Berlin	3000
Paris	9000
London	8000
Paris	8000

Example 3

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Customer Type
00681	01/01/2020, 04:36	Standard
01153	01/01/2020, 10:31	Standard
01524	02/01/2020, 00:54	Standard
03055	02/01/2020, 03:09	Premium
03015	02/01/2020, 04:09	Standard
00854	02/01/2020, 04:29	Premium
02060	02/01/2020, 06:48	Standard
00010	02/01/2020, 09:02	Standard
03040	02/01/2020, 12:27	Premium
01665	02/01/2020, 16:44	Premium

The following query calculates, at the time each premium order was placed, how many premium orders were opened within the previous 12 hours.

```
SELECT case_id,
    "Order Date",
    "Customer Type",
    COUNT(case_id)
        OVER (ORDER BY "Order Date" RANGE BETWEEN DURATION '12hours' PRECEDING
AND CURRENT ROW)
        AS "Premium Orders Over 12 Hrs"
FROM THIS_PROCESS
WHERE "Customer Type" = 'Premium'
```

Result:

case_id	Order Date	Customer Type	Premium Orders Over 12 Hrs
03055	02/01/2020, 03:09	Premium	1
00854	02/01/2020, 04:29	Premium	2

case_id	Order Date	Customer Type	Premium Orders Over 12 Hrs
03040	02/01/2020, 12:27	Premium	3
01665	02/01/2020, 16:44	Premium	2

1.6.7.4 Window-Supporting Aggregate Functions

The functions listed in this section support windows.

Some of the standard aggregate functions can be applied to windows. This allows the function to be applied to defined groups of rows instead of the entire table.

For more general information about windows, refer to the Window Functions overview.

> ⚠ Caution
>
> For very large data sets, these functions may require excessive CPU activity, causing long query execution times.
>
> For more information, refer to Performance [page 294].

AVG [page 230]
Calculates the average of a collection of numeric values. `NULL` values are ignored

BOOL_AND [page 232]
Returns true if the supplied expression evaluates to true for all input rows, otherwise it returns false.

BOOL_OR [page 235]
Returns true if the supplied expression evaluates to true for any input row, otherwise it returns false.

COUNT [page 238]
Counts the number of values in a specified column. `NULL` values aren't counted.

FIRST [page 240]
Returns the first element from a collection of values.

LAST [page 244]
Returns the last element from a collection of values.

MAX [page 248]
Finds the maximum value in a collection of values.

MIN [page 250]
Finds the minimum value in a collection of values.

STDDEV [page 253]
Calculates the standard deviation for a collection of values.

SUM [page 255]
Calculates the sum of all values in a collection of numeric values. `NULL` values are ignored.

TRIMMED_AVG [page 257]
Calculates the average of a group of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and calculating the average.

TRIMMED_STDDEV [page 261]
: Calculates the population standard deviation of a group of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and calculating the average.

TRIMMED_VARIANCE [page 265]
: Calculates the population variance for a collection of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and performing the calculation.

VARIANCE [page 269]
: Calculates the population variance for a collection of values. `NULL` values are ignored.

Related Information

Aggregate Functions [page 92]
Window Functions [page 220]

1.6.7.4.1 AVG

Calculates the average of a collection of numeric values. `NULL` values are ignored

Syntax

```
AVG(<expression>)
```

Parameter	Description	Valid Types
`expression`	The column of values to be averaged.	Number, Timestamp, Duration

Returns: The average as a Number, Timestamp, or Duration. The return type matches the type of the `expression` parameter.

Use as a Window Function

`AVG` can also be used as a window function:

```
AVG(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
AVG(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount
02302	01/01/2020, 00:48	404.54
01553	01/01/2020, 03:00	482.85
00681	01/01/2020, 04:36	1030.71
01153	01/01/2020, 10:31	700.68
01524	02/01/2020, 00:54	706.96

This query returns the average order amount of all cases.

```
SELECT AVG("Order Amount") AS "Average Order Amount"
FROM THIS_PROCESS
```

Output:

Average Order Amount
665.148

Example 2

This example demonstrates a windowed average. Assuming the same data set as the previous example, the following query calculates a running average at the point of each sale. It does so by defining for each row a window between the first row and the current row, calculating the average order amount within that window.

```
SELECT case_id,
    "Order Date",
    "Order Amount",
    AVG("Order Amount")
```

```
        OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS "AVG All Orders to Date"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Order Amount	AVG All Orders to Date
02302	01/01/2020, 00:48	404.54	404.54
01553	01/01/2020, 03:00	482.85	443.695
00681	01/01/2020, 04:36	1030.71	639.367
01153	01/01/2020, 10:31	700.68	654.695
01524	02/01/2020, 00:54	706.96	665.148

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT AVG("Order Amount") FILTER (WHERE "Order Amount" < 500) AS "Average"
FROM THIS_PROCESS
```

Average
443.70

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.4.2 BOOL_AND

Returns true if the supplied expression evaluates to true for all input rows, otherwise it returns false.

Syntax

```
BOOL_AND(<expression>)
```

Parameter	Description	Valid Types
`expression`	An expression applied to a collection of rows.	Boolean

Returns: A Boolean. Result is true if all input rows evaluate to true for the supplied expression, otherwise false.

Use as a Window Function

`BOOL_AND` can also be used as a window function:

```
BOOL_AND(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
BOOL_AND(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount	Customer Type
03015	02/01/2020, 04:09	269.4	Standard
00854	02/01/2020, 04:29	374.01	Premium
02060	02/01/2020, 06:48	84.33	Standard
00010	02/01/2020, 09:02	319.18	Standard
03040	02/01/2020, 12:27	20	Premium
01665	02/01/2020, 16:44	492.27	Premium
01951	02/01/2020, 21:32	521.2	Standard

The following query determines whether all cases have order amounts over 30.

```
SELECT BOOL_AND("Order Amount" > 30) AS "All Over 30?"
FROM THIS_PROCESS
```

The answer is false because one of the amounts is lower than 30.

All Over 30?
FALSE

Example 2

Assuming the same data set as the previous example, the following query demonstrates the windowed version of `BOOL_AND`. It considers two cases at a time: the current case and its immediate predecessor. If "`Customer Type`" is listed as "`Premium`" in both, the function returns true.

```
SELECT case_id,
    "Order Date",
    "Customer Type",
    BOOL_AND("Customer Type" = 'Premium')
        OVER (ROWS BETWEEN 1 PRECEDING AND CURRENT ROW)
        AS "Both Premium?"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Customer Type	Both Premium?
03015	02/01/2020, 04:09	Standard	FALSE
00854	02/01/2020, 04:29	Premium	FALSE
02060	02/01/2020, 06:48	Standard	FALSE
00010	02/01/2020, 09:02	Standard	FALSE
03040	02/01/2020, 12:27	Premium	FALSE
01665	02/01/2020, 16:44	Premium	TRUE
01951	02/01/2020, 21:32	Standard	FALSE

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT BOOL_AND("Order Amount" > 50) FILTER (WHERE "Customer Type" = 'Standard')
AS "All Standard Over 50?"
FROM THIS_PROCESS
```

All Standard Over 50?
TRUE

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.4.3 BOOL_OR

Returns true if the supplied expression evaluates to true for any input row, otherwise it returns false.

Syntax

```
BOOL_OR(<expression>)
```

Parameter	Description	Valid Types
`expression`	An expression applied to a collection of rows.	Boolean

Returns: A Boolean. Result is true if any input row evaluates to true for the supplied expression, otherwise false.

Use as a Window Function

`BOOL_OR` can also be used as a window function:

```
BOOL_OR(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
BOOL_OR(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount	Customer Type
03015	02/01/2020, 04:09	269.4	Standard
00854	02/01/2020, 04:29	374.01	Premium
02060	02/01/2020, 06:48	84.33	Standard
00010	02/01/2020, 09:02	319.18	Standard
03040	02/01/2020, 12:27	20	Premium
01665	02/01/2020, 16:44	492.27	Premium
01951	02/01/2020, 21:32	521.2	Standard

The following query determines whether any cases have order amounts over 30.

```
SELECT BOOL_OR("Order Amount" > 30) AS "Any Over 30?"
FROM THIS_PROCESS
```

The answer is true since at least one of the amounts is greater than 30:

Any Over 30?
TRUE

Example 2

Assuming the same data set as the previous example, the following query demonstrates the windowed version of `BOOL_OR`. It considers two cases at a time: the current case and its immediate predecessor. If "`Customer Type`" is listed as "`Premium`" in either, the function returns true.

```
SELECT case_id,
    "Order Date",
    "Customer Type",
    BOOL_OR("Customer Type" = 'Premium')
        OVER (ROWS BETWEEN 1 PRECEDING AND CURRENT ROW)
        AS "Either Premium?"
```

```
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Either Premium?	Customer Type
03015	02/01/2020, 04:09	FALSE	Standard
00854	02/01/2020, 04:29	TRUE	Premium
02060	02/01/2020, 06:48	TRUE	Standard
00010	02/01/2020, 09:02	FALSE	Standard
03040	02/01/2020, 12:27	TRUE	Premium
01665	02/01/2020, 16:44	TRUE	Premium
01951	02/01/2020, 21:32	TRUE	Standard

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT BOOL_OR("Order Amount" < 20) FILTER (WHERE "Customer Type" = 'Premium')
AS "Any Premium Under 20?"
FROM THIS_PROCESS
```

Any Premium Under 20?
FALSE

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.4.4 COUNT

Counts the number of values in a specified column. `NULL` values aren't counted.

Syntax

```
COUNT(<expression>)
```

Parameter	Description	Valid Types
`expression`	The column whose values are to be counted.	Any

Returns: The number of values in the provided collection of values.

Use as a Window Function

`COUNT` can also be used as a window function:

```
COUNT(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
COUNT(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Customer Type
00681	01/01/2020, 04:36	Standard
01153	01/01/2020, 10:31	Standard
01524	02/01/2020, 00:54	Standard
03055	02/01/2020, 03:09	Premium
03015	02/01/2020, 04:09	Standard
00854	02/01/2020, 04:29	Premium
02060	02/01/2020, 06:48	Standard
00010	02/01/2020, 09:02	Standard
03040	02/01/2020, 12:27	Premium
01665	02/01/2020, 16:44	Premium

The following query counts the number of rows.

```
SELECT COUNT(case_id) AS "Num. of Orders"
FROM THIS_PROCESS
```

Output:

Num. of Orders
10

Example 2

This example demonstrates a windowed count. Assuming the same data set as the previous example, the following query calculates how many premium customers placed orders between the time of each order and the previous twelve hours.

```
SELECT case_id,
    "Order Date",
    "Customer Type",
    COUNT(case_id)
        OVER (ORDER BY "Order Date" RANGE BETWEEN DURATION '12hours' PRECEDING
AND CURRENT ROW)
        AS "Premium Orders Over 12 Hrs"
FROM THIS_PROCESS
WHERE "Customer Type" = 'Premium'
```

The first three premium orders arrive within less than twelve hours, hence the count climbs from 1 to 3. By the time of the fourth premium order, the first two orders have dropped out of the twelve-hour window and so the count drops to 2.

case_id	Order Date	Customer Type	Premium Orders Over 12 Hrs
03055	02/01/2020, 03:09	Premium	1
00854	02/01/2020, 04:29	Premium	2
03040	02/01/2020, 12:27	Premium	3
01665	02/01/2020, 16:44	Premium	2

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT COUNT(case_id) FILTER (WHERE "Customer Type" = 'Premium') AS "Count"
FROM THIS_PROCESS
```

Count
4

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.4.5 FIRST

Returns the first element from a collection of values.

Syntax

```
FIRST(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection of values from which the first is chosen.	Number, Timestamp, Duration, Text, Boolean

Returns: The first value as a Number, Timestamp, Duration, Text or Boolean. The return type matches the type of the `expression` parameter.

Use as a Window Function

`FIRST` can also be used as a window function:

```
FIRST(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
FIRST(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	event_name
00606	Receive Customer Order
	Receive Payment
	Receive Payment
	Send items to Printing
	Order Canceled
02178	Receive Customer Order
	Ship Goods Standard
	Receive Payment
	Receive Delivery Confirmation

case_id	event_name
01200	Receive Customer Order
	Receive Payment
	Ship Goods Express
	Ship Goods Express
	Receive Delivery Confirmation
	Receive Delivery Confirmation
01750	Receive Customer Order
	Change Order Quantity
	Receive Payment
	Ship Goods Standard
	Receive Delivery Confirmation

The following query returns the first event that occurs in each case after receipt of an order (which is always the first event). It does so by filtering out those receipt events and then taking the first event from what remains.

```
SELECT
    case_id,
    (SELECT FIRST(event_name)) AS "First Action After Receipt"
FROM THIS_PROCESS
FILTER EVENTS WHERE event_name <> 'Receive Customer Order'
```

Output:

case_id	First Action After Receipt
00606	Receive Payment
02178	Ship Goods Standard
01200	Receive Payment
01750	Change Order Quantity

Example 2

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. The following query achieves the same goal as the previous example by using the`FILTER` clause in a subquery instead of using `FILTER EVENTS`:

```
SELECT case_id,
    (SELECT FIRST(event_name) FILTER (WHERE event_name <> 'Receive Customer
Order')) AS "First Action After Receipt"
FROM THIS_PROCESS
```

Result:

case_id	First Action After Receipt
00606	Receive Payment
01200	Receive Payment
01750	Change Order Quantity
02178	Ship Goods Standard

Example 3

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Customer Type
00681	01/01/2020, 04:36	Standard
01153	01/01/2020, 10:31	Standard
01524	02/01/2020, 00:54	Standard
03055	02/01/2020, 03:09	Premium
03015	02/01/2020, 04:09	Standard
00854	02/01/2020, 04:29	Premium
02060	02/01/2020, 06:48	Standard
00010	02/01/2020, 09:02	Standard
03040	02/01/2020, 12:27	Premium
01665	02/01/2020, 16:44	Premium

The following query finds the date of the second order before the current one. It does so by using a window comprising the two previous rows and taking the order date of the first row in the window.

```
SELECT case_id,
    "Order Date",
    FIRST("Order Date")
        OVER (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
        AS "Date Two Orders Prior"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Date Two Orders Prior
1553	01/01/2020, 03:00	01/01/2020, 03:00
681	01/01/2020, 04:36	01/01/2020, 03:00
1153	01/01/2020, 10:31	01/01/2020, 03:00

case_id	Order Date	Date Two Orders Prior
1524	02/01/2020, 00:54	01/01/2020, 04:36
3055	02/01/2020, 03:09	01/01/2020, 10:31
3015	02/01/2020, 04:09	02/01/2020, 00:54
854	02/01/2020, 04:29	02/01/2020, 03:09

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.4.6 LAST

Returns the last element from a collection of values.

Syntax

```
LAST(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection of values from which the last is chosen.	Number, Timestamp, Duration, Text, Boolean

Returns: The last value as a Number, Timestamp, Duration, Text or Boolean. The return type matches the type of the `expression` parameter.

Use as a Window Function

LAST can also be used as a window function:

```
LAST(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
LAST(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Status	event_name
00606	Canceled	Receive Customer Order
		Receive Payment
		Receive Payment
		Send items to Printing
		Order Canceled
02178	Delivered	Receive Customer Order
		Ship Goods Standard
		Receive Payment
		Receive Delivery Confirmation
01200	Delivered	Receive Customer Order
		Receive Payment
		Ship Goods Express
		Ship Goods Express
		Receive Delivery Confirmation
01750	Delivered	Receive Customer Order
		Change Order Quantity
		Receive Payment
		Ship Goods Standard
		Receive Delivery Confirmation

The following query returns the last event that occurs in each case.

```
SELECT
    case_id,
```

```
    (SELECT LAST(event_name)) AS "Last Action"
FROM THIS_PROCESS
```

Output:

case_id	Last Action
00606	Order Canceled
02178	Receive Delivery Confirmation
01200	Receive Delivery Confirmation
01750	Receive Delivery Confirmation

Example 2

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. The following query finds the last event in all delivered orders but filters the final delivery confirmation event, resulting in it returning the penultimate event:

```
SELECT case_id,
    (SELECT LAST(event_name) FILTER (WHERE event_name <> 'Receive Delivery
Confirmation')) AS "Last Action Before Delivery"
FROM THIS_PROCESS
WHERE "Order Status" = 'Delivered'
```

Result:

case_id	Last Action Before Delivery
01200	Ship Goods Express
01750	Ship Goods Standard
02178	Receive Payment

Example 3

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Customer Type
00681	01/01/2020, 04:36	Standard
01153	01/01/2020, 10:31	Standard
01524	02/01/2020, 00:54	Standard
03055	02/01/2020, 03:09	Premium
03015	02/01/2020, 04:09	Standard

case_id	Order Date	Customer Type
00854	02/01/2020, 04:29	Premium
02060	02/01/2020, 06:48	Standard
00010	02/01/2020, 09:02	Standard
03040	02/01/2020, 12:27	Premium
01665	02/01/2020, 16:44	Premium

The following query finds the date of the second order after the current one. It does so by using a window comprising the two following rows and taking the order date of the last row in the window.

```
SELECT case_id,
    "Order Date",
    LAST("Order Date")
        OVER (ROWS BETWEEN CURRENT ROW AND 2 FOLLOWING)
        AS "Date Two Orders Later"
ORDER BY 2
```

Output:

case_id	Order Date	Date Two Orders Later
01553	01/01/2020, 03:00	01/01/2020, 10:31
00681	01/01/2020, 04:36	02/01/2020, 00:54
01153	01/01/2020, 10:31	02/01/2020, 03:09
01524	02/01/2020, 00:54	02/01/2020, 04:09
03055	02/01/2020, 03:09	02/01/2020, 04:29
03015	02/01/2020, 04:09	02/01/2020, 04:29
00854	02/01/2020, 04:29	02/01/2020, 04:29

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.4.7 MAX

Finds the maximum value in a collection of values.

Syntax

```
MAX(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection from which the maximum value is chosen.	Number, Timestamp, Duration

Returns: The maximum value as a Number, Timestamp or Duration. The return type matches the type of the `expression` parameter.

Use as a Window Function

`MAX` can also be used as a window function:

```
MAX(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
MAX(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount
02302	01/01/2020, 00:48	404.54
01553	01/01/2020, 03:00	482.85
00681	01/01/2020, 04:36	1030.71
01153	01/01/2020, 10:31	700.68
01524	02/01/2020, 00:54	706.96

This query returns the maximum order amount found in all cases.

```
SELECT MAX("Order Amount") AS "Maximum Order Amount"
FROM THIS_PROCESS
```

Output:

Maximum Order Amount
1030.71

Example 2

This example demonstrates a windowed maximum. Assuming the same data set as the previous example, the following query calculates the maximum order amount to date when each order was placed. It does so by defining for each row a window between the first row and the current row, finding the maximum order amount within that window.

```
SELECT case_id,
    "Order Date",
    "Order Amount",
    MAX("Order Amount")
        OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS "MAX All Orders to Date"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Order Amount	MAX All Orders to Date
02302	01/01/2020, 00:48	404.54	404.54
01553	01/01/2020, 03:00	482.85	482.85
00681	01/01/2020, 04:36	1030.71	1030.71
01153	01/01/2020, 10:31	700.68	1030.71

case_id	Order Date	Order Amount	MAX All Orders to Date
01524	02/01/2020, 00:54	706.96	1030.71

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT MAX("Order Amount") FILTER (WHERE "Order Amount" < 1000) AS "Maximum
Under 1000"
FROM THIS_PROCESS
```

Maximum Under 1000
706.96

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.4.8 MIN

Finds the minimum value in a collection of values.

Syntax

```
MIN(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection from which the minimum value is chosen.	Number, Timestamp, Duration

Returns: The minimum value as a Number, Timestamp or Duration. The return type matches the type of the `expression` parameter.

Use as a Window Function

`MIN` can also be used as a window function:

```
MIN(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
MIN(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount
02302	01/01/2020, 00:48	404.54
01553	01/01/2020, 03:00	482.85
00681	01/01/2020, 04:36	1030.71
01153	01/01/2020, 10:31	700.68
01524	02/01/2020, 00:54	706.96

This query returns the minimum order amount found in all cases.

```
SELECT MIN("Order Amount") AS "Minimum Order Amount"
FROM THIS_PROCESS
```

Output:

Minimum Order Amount
404.54

Example 2

This example demonstrates a windowed minimum. Assuming the same data set as the previous example, the following query calculates the minimum order amount to date at the point when each order was placed. It does so by defining for each row a window between the first row and the current row, finding the minimum order amount within that window.

```
SELECT case_id,
    "Order Date",
    "Order Amount",
    MIN("Order Amount")
        OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS "MIN All Orders to Date"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Order Amount	MIN All Orders to Date
02302	01/01/2020, 00:48	404.54	404.54
01553	01/01/2020, 03:00	482.85	404.54
00681	01/01/2020, 04:36	1030.71	404.54
01153	01/01/2020, 10:31	700.68	404.54
01524	02/01/2020, 00:54	706.96	404.54

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT MIN("Order Amount") FILTER (WHERE "Order Amount" > 500) AS "Minimum over
500"
FROM THIS_PROCESS
```

Minimum over 500
700.68

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.4.9 STDDEV

Calculates the standard deviation for a collection of values.

The standard deviation describes the average deviation of all measured values from the mean value. A low standard deviation indicates that the values tend to be close to the mean value. A high standard deviation indicates that the values are spread out over a wide range.

Syntax

```
STDDEV(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection of values for which you want to determine the standard deviation.	Number, Duration

Returns: The standard deviation as a Number, Timestamp, or Duration. The return type matches the type of the `expression` parameter.

Use as a Window Function

`STDDEV` can also be used as a window function:

```
STDDEV(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
STDDEV(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

This query returns the standard deviation of all order amounts.

```
SELECT
    MIN("Order Amount") AS "Min",
    MAX("Order Amount") AS "Max",
    AVG("Order Amount") AS "Avg",
    STDDEV("Order Amount") AS "StdDev"
FROM THIS_PROCESS
```

Output:

Min	Max	Avg	StdDev
20	2,166.14	385.668	357.452

Example 2

This query shows how the cycle time of a process changes week-to-week. Specifically, it calculates both the average and the standard deviation of the cycle time on a weekly basis.

To smooth out short-term fluctuations in the data, it uses windowed versions of the average (`AVG`) and standard deviation (`STDDEV`) functions. These versions calculate a moving average and moving standard deviation for the weekly mean cycle time respectively.

In the nested query, `sq1`, all cases are clustered together by week. The mean cycle time within each cluster is calculated.

The outer query applies the windowed `AVG` and `STDDEV` functions to the weekly mean cycle time. In both cases, the window includes the three weeks preceding and the three weeks following the current week.

```
SELECT
    "Week",
    "Avg Cycle Time",
    AVG("Avg Cycle Time") OVER (ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING) as
mean,
    STDDEV("Avg Cycle Time") OVER (ROWS BETWEEN 3 PRECEDING AND 3 FOLLOWING) as
std
FROM
(
    SELECT
        DATE_TRUNC('WEEK', (SELECT LAST (END_TIME))) AS "Week",
        AVG((SELECT LAST(end_time) - FIRST(end_time))) AS "Avg Cycle Time"
    FROM THIS_PROCESS
    ORDER BY 1 ASC NULLS FIRST
    FILL timeseries('WEEK')
) as sq1
```

Week	Avg Cycle Time	mean	std
06/01/2020, 00:00	6d 14h	9d 20h	1d 23h
13/01/2020, 00:00	9d 20h	10d	1d 19h

Week	Avg Cycle Time	mean	std
20/01/2020, 00:00	11d 5h	10d	1d 15h
27/01/2020, 00:00	11d 16h	10d 4h	1d 13h
03/02/2020, 00:00	10d 16h	10d 18h	13h 40m
10/02/2020, 00:00	10d 3h	10d 20h	11h 12m
17/02/2020, 00:00	10d 23h	10d 18h	10h 45m
24/02/2020, 00:00	10d 18h	10d 17h	9h 18m

Related Information

Window Functions [page 220]

1.6.7.4.10 SUM

Calculates the sum of all values in a collection of numeric values. `NULL` values are ignored.

Syntax

```
SUM(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection of values to be summed.	Number, Duration

Returns: The sum as a Number or Duration. The return type matches the type of the `expression` parameter.

Use as a Window Function

`SUM` can also be used as a window function:

```
SUM(<expression>) OVER (
    [ PARTITION BY [, <partitionExpression>, ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression>, ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
SUM(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following data set (`THIS_PROCESS`):

case_id	Order Date	Order Amount
02302	01/01/2020, 00:48	404.54
01553	01/01/2020, 03:00	482.85
00681	01/01/2020, 04:36	1030.71
01153	01/01/2020, 10:31	700.68
01524	02/01/2020, 00:54	706.96

This query returns the total order amount in all cases.

```
SELECT SUM("Order Amount") AS "Total Order Amount"
FROM THIS_PROCESS
```

Output:

Total Order Amount
3325.74

Example 2

This example demonstrates a running total. Assuming the same data set as the previous example, the following query calculates the running total at the point when each order was placed. It does so by defining for each row a window between the first row and the current row, finding the sum of all order amounts within that window.

```
SELECT case_id,
    "Order Date",
    "Order Amount",
    SUM("Order Amount")
        OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
        AS "SUM All Orders to Date"
FROM THIS_PROCESS
ORDER BY 2
```

Output:

case_id	Order Date	Order Amount	SUM All Orders to Date
02302	01/01/2020, 00:48	404.54	404.54
01553	01/01/2020, 03:00	482.85	887.39
00681	01/01/2020, 04:36	1030.71	1918.1
01153	01/01/2020, 10:31	700.68	2618.78
01524	02/01/2020, 00:54	706.96	3325.74

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT SUM("Order Amount") FILTER (WHERE "Order Amount" < 1000) AS "Sum"
FROM THIS_PROCESS
```

Sum
1407.64

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.4.11 TRIMMED_AVG

Calculates the average of a group of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and calculating the average.

A cutoff ranges from 0 to 1 and expresses the percentage of values to be trimmed before the calculation is performed. For example, if a group contains 10 values, setting the lower and upper cutoffs to 0.2 means that up to 20% of the lowest values and up to 20% of the highest values are trimmed. 20% of 10 is 2, so the 2 lowest and 2 highest values are trimmed, excluding 4 values in total from the calculation.

If a cutoff results in a non-integer number of values to be trimmed, the resulting number is rounded down to the next lower whole number. For a group of 10 values, a cutoff of 0.15 results in 1.5, in which case the result is rounded down to 1.

Syntax

```
TRIMMED_AVG(<expression>, <lowerCutoff>, <upperCutoff>)
```

Parameter	Description	Valid Types
`expression`	The group of values to be trimmed and averaged. `NULL` values are ignored.	Number, Timestamp, Duration
`lowerCutoff`	A number between 0 and 1. Specifies what percentage of the smallest values are trimmed before calculating the average. **Note** • 0 means that no values are trimmed. • 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number
`upperCutoff`	A number between 0 and 1. Specifies what percentage of the largest values are trimmed before calculating the average. **Note** • 0 means that no values are trimmed. • 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number

Returns: The trimmed average as a Number, Timestamp, or Duration. The return type matches the type of the `expression` parameter. If all values of `expression` are `NULL` or no values remain after trimming, then `NULL` is returned.

Use as a Window Function

This function can also be used as a window function:

```
TRIMMED_AVG(<expression>, <lowerCutoff>, <upperCutoff>) OVER (
    [ PARTITION BY <partitionExpression> [, <partitionExpression> ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression> ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
TRIMMED_AVG(<expression>, <lowerCutoff>, <upperCutoff>) FILTER (WHERE
<condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	20
00002	40
00003	40
00004	20
00005	5
00006	115
00007	null

The following query demonstrates the difference between calculating an average and a trimmed average.

```
SELECT
    AVG("Order Amount") AS "Average",
    TRIMMED_AVG("Order Amount", 0.2, 0.2) AS "Trimmed Average"
FROM THIS_PROCESS
```

- When calculating the **average**, all (non-`NULL`) values are included in the calculation.
- In this example, the **trimmed average** has upper and lower cutoffs of 0.2, meaning that up to 20% of the largest and smallest values are excluded from the calculation. There are 6 non-`NULL` values and 20% of 6 is 1.2. This result is therefore rounded down to the next lowest whole number, 1, and so the single highest and lowest values (115 and 5) are excluded.

Output:

Average	Trimmed Average
40	30

Example 2

Because this function is also available as a window function, it's possible to calculate the trimmed average of a window frame.

In the following query, the lower cutoff of 0.4 removes up to 40% of the smallest values in the specified window frame. That window frame includes the current row and the two preceding rows.

```
SELECT
    case_id,
    "Order Amount",
    TRIMMED_AVG("Order Amount", 0.4, 0.0)
        OVER (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
        AS "Trimmed Average - Window"
FROM THIS_PROCESS
ORDER BY 1
```

Output:

case_id	Order Amount	Trimmed Average - Window
00001	20	20
00002	40	30
00003	40	40
00004	20	40
00005	5	30
00006	115	67.5
00007	null	60

The trimmed average values are calculated as follows:

- For the first row (case_id = 00001), the window frame includes only one value (20). Based on the defined cutoffs, no value can be removed from the window frame, so the average is 20.
- For the second row (case_id = 00002), the window frame includes the values 20 and 40. Removing the lowest value would exceed the defined 40 percent cutoff limit, so no values are trimmed. Therefore, the average is 30 (40 + 20 / 2).
- For the third row (case_id = 00003), the window frame includes the values 20, 40, and 40. It's now possible to trim values because 40% of 3 is 1.2, which is rounded down to 1. That means the lowest value (20) is excluded from the calculation. The average of the remaining values (40 and 40) is 40.
- Calculations continue like this for all remaining rows in the data set. In the final row (case_id = 00007) `NULL` is ignored, which means the window contains only two values, neither of which are trimmed.

Example 3

As with all aggregate functions, the `FILTER` clause can be applied to this function to remove unwanted values from the aggregation. For example:

```
SELECT TRIMMED_AVG("Order Amount", 0.2, 1) FILTER (WHERE "Order Amount" < 100)
AS "Average"
```

```
FROM THIS_PROCESS
```

Average
443.70

Related Information

FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.4.12 TRIMMED_STDDEV

Calculates the population standard deviation of a group of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and calculating the average.

A cutoff ranges from 0 to 1 and expresses the percentage of values to be trimmed before the calculation is performed. For example, if a group contains 10 values, setting the lower and upper cutoffs to 0.2 means that up to 20% of the lowest values and up to 20% of the highest values are trimmed. 20% of 10 is 2, so the 2 lowest and 2 highest values are trimmed, excluding 4 values in total from the calculation.

If a cutoff results in a non-integer number of values to be trimmed, the resulting number is rounded down to the next lower whole number. For a group of 10 values, a cutoff of 0.15 results in 1.5, in which case the result is rounded down to 1.

Syntax

```
TRIMMED_STDDEV(<expression>, <lowerCutoff>, <upperCutoff>)
```

Parameter	Description	Valid Types
`expression`	The group of values to be trimmed and have their standard deviation calculated. `NULL` values are ignored.	Number, Duration

Parameter	Description	Valid Types
`lowerCutoff`	A number between 0 and 1. Specifies what percentage of the smallest values are trimmed before calculating the standard deviation. **Note** • 0 means that no values are trimmed. • 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number
`upperCutoff`	A number between 0 and 1. Specifies what percentage of the largest values are trimmed before calculating the standard deviation. **Note** • 0 means that no values are trimmed. • 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number

Returns: The standard deviation as a Number or Duration. The return type matches the type of the `expression` parameter. If all values of `expression` are `NULL` or no values remain after trimming, then `NULL` is returned.

Use as a Window Function

This function can also be used as a window function:

```
TRIMMED_STDDEV(<expression>, <lowerCutoff>, <upperCutoff>) OVER (
    [ PARTITION BY <partitionExpression> [, <partitionExpression> ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression> ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
TRIMMED_STDDEV(<expression>, <lowerCutoff>, <upperCutoff>) FILTER (WHERE
<condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	20
00002	40
00003	40
00004	20
00005	5
00006	115
00007	null

The following query demonstrates the difference between calculating a standard deviation and a trimmed standard deviation.

```
SELECT
    STDDEV("Order Amount") AS "Std. Dev",
    TRIMMED_STDDEV("Order Amount", 0.1, 0.2) AS "Trimmed Std. Dev"
FROM THIS_PROCESS
```

- When calculating the **standard deviation**, all (non-`NULL`) values are included in the calculation.
- In this example, the **trimmed standard deviation** has lower and upper cutoffs of 0.1 and 0.2 respectively.
  - Up to 10% of the smallest values are excluded from the calculation. There are 6 values and 10% of 6 is 0.6. This result is therefore rounded down to the next lowest whole number, 0, so none of the lowest values are excluded.
  - Up to 20% of the largest values are excluded from the calculation. 20% of 6 values is 1.2. This result is therefore rounded down to the next lowest whole number, 1, and so the single highest value (115) is excluded.

Output:

Std. Dev.	Trimmed Std. Dev.
35.707	13.416

Example 2

Because this function is also available as a window function, it's possible to calculate the trimmed standard deviation of a window frame.

In the following query, the lower cutoff of 0.4 removes up to 40% of the smallest values in the specified window frame. That window frame includes the current row and the two preceding rows.

```
SELECT
    case_id,
    "Order Amount",
    TRIMMED_STDDEV("Order Amount", 0.4, 0.0)
        OVER (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
        AS "Trimmed Std. Dev. - Window"
FROM THIS_PROCESS
ORDER BY 1
```

Output:

case_id	Order Amount	Trimmed Std. Dev. - Window
00001	20	0
00002	40	10
00003	40	0
00004	20	0
00005	5	10
00006	115	47.5
00007	null	55

The trimmed standard deviation values are calculated as follows:

- For the first row (case_id = 00001), the window frame includes only one value (20). Based on the defined cutoffs, no value can be removed from the window frame, so the standard deviation is 0.
- For the second row (case_id = 00002), the window frame includes the values 20 and 40. Removing the lowest value would exceed the defined 40 percent cutoff limit, so no values are trimmed. Therefore, the standard deviation of values 20 and 40 is 10.
- For the third row (case_id = 00003), the window frame includes the values 20, 40, and 40. It's now possible to trim values because 40% of 3 is 1.2, which is rounded down to 1. That means the lowest value (20) is excluded from the calculation. The standard deviation of the remaining values (40 and 40) is 0.
- Calculations continue like this for all remaining rows in the data set. In the final row (case_id = 00007) `NULL` is ignored, which means the window contains only two values, neither of which are trimmed.

Related Information

Window Functions [page 220]

1.6.7.4.13 TRIMMED_VARIANCE

Calculates the population variance for a collection of values, trimming all outliers above and below defined cutoffs, and excluding them from the calculation. `NULL` values are ignored, both when determining cutoffs and performing the calculation.

A cutoff ranges from 0 to 1 and expresses the percentage of values to be trimmed before the calculation is performed. For example, if a group contains 10 values, setting the lower and upper cutoffs to 0.2 means that up to 20% of the lowest values and up to 20% of the highest values are trimmed. 20% of 10 is 2, so the 2 lowest and 2 highest values are trimmed, excluding 4 values in total from the calculation.

If a cutoff results in a non-integer number of values to be trimmed, the resulting number is rounded down to the next lower whole number. For a group of 10 values, a cutoff of 0.15 results in 1.5, in which case the result is rounded down to 1.

> ⓘ Note
>
> This function calculates **population** variance, which measures dispersion using all values in a data set. This shouldn't be confused with **sample** variance, which measures dispersion using a subset of those values and yields a different result.

Syntax

```
TRIMMED_VARIANCE(<expression>, <lowerCutoff>, <upperCutoff>)
```

Parameter	Description	Valid Types
`expression`	The group of values for which you want to determine the variance. `NULL` values are ignored.	Number → Tip To calculate the variance of duration values, you can first convert them into numeric values using the functions `DATE_DIFF` or `DURATION_TO_DAYS`.

Parameter	Description	Valid Types
`lowerCutoff`	A number between 0 and 1. Specifies what percentage of the smallest values are trimmed before calculating the variance. **Note** • 0 means that no values are trimmed. • 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number
`upperCutoff`	A number between 0 and 1. Specifies what percentage of the largest values are trimmed before calculating the variance. **Note** • 0 means that no values are trimmed. • 1 means that all values are trimmed. In this case, the function returns `NULL`.	Number

Returns: The trimmed variance as a Number. If all values of `expression` are `NULL` or no values remain after trimming, then `NULL` is returned.

Use as a Window Function

This function can also be used as a window function:

```
TRIMMED_VARIANCE(<expression>, <lowerCutoff>, <upperCutoff>) OVER (
    [ PARTITION BY <partitionExpression> [, <partitionExpression> ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression> ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
TRIMMED_VARIANCE(<expression>, <lowerCutoff>, <upperCutoff>) FILTER (WHERE
<condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assume the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	20
00002	40
00003	40
00004	20
00005	5
00006	115
00007	null

The following query demonstrates the difference between calculating variance and trimmed variance.

```
SELECT
    VARIANCE("Order Amount") AS "Variance",
    TRIMMED_VARIANCE("Order Amount", 0.2, 0.2) AS "Trimmed Variance"
FROM THIS_PROCESS
```

- When calculating the **variance**, all (non-`NULL`) values are included in the calculation.
- In this example, the **trimmed variance** has upper and lower cutoffs of 0.2, meaning that up to 20% of the largest and smallest values are excluded from the calculation. There are 6 (non-`NULL`) values and 20% of 6 is 1.2. This result is therefore rounded down to the next lowest whole number, 1, and so the single highest and lowest values (115 and 5) are excluded.

Output:

Variance	Trimmed Variance
1275	100

Example 2

Because this function is also available as a window function, it's possible to calculate the trimmed variance of a window frame.

In the following query, the lower cutoff of 0.4 removes up to 40% of the smallest values in the specified window frame. That window frame includes the current row and the two preceding rows.

```
SELECT
    case_id,
    "Order Amount",
    TRIMMED_VARIANCE("Order Amount", 0.4, 0.0)
        OVER (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
        AS "Trimmed Variance - Window"
FROM THIS_PROCESS
ORDER BY 1
```

Output:

case_id	Order Amount	Trimmed Average - Window
00001	20	0
00002	40	100
00003	40	0
00004	20	0
00005	5	100
00006	115	2256.25
00007	null	3.025

The trimmed variance values are calculated as follows:

- For the first row (case_id = 00001), the window frame includes only one value (20). Based on the defined cutoffs, no value can be removed from the window frame, so the variance is 0.
- For the second row (case_id = 00002), the window frame includes the values 20 and 40. Removing the lowest value would exceed the defined 40 percent cutoff limit, so no values are trimmed. Therefore, the variance of values 20 and 40 is 100.
- For the third row (case_id = 00003), the window frame includes the values 20, 40, and 40. It's now possible to trim values because 40% of 3 is 1.2, which is rounded down to 1. That means the lowest value (20) is excluded from the calculation. The variance of the remaining values (40 and 40) is 0.
- Calculations continue like this for all remaining rows in the data set. In the final row (case_id = 00007) `NULL` is ignored, which means the window contains only two values, neither of which are trimmed.

Related Information

Conversion Functions [page 154]
FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.4.14 VARIANCE

Calculates the population variance for a collection of values. `NULL` values are ignored.

Variance measures the dispersion of values within a data set, determining how far those values are spread out from their average value.

> ⓘ Note
>
> This function calculates **population** variance, which measures dispersion using all values in a data set. This shouldn't be confused with **sample** variance, which measures dispersion using a subset of those values and yields a different result.

Syntax

```
VARIANCE(<expression>)
```

Parameter	Description	Valid Types
`expression`	The collection of values for which you want to determine the variance.	Number → Tip To calculate the variance of duration values, you can first convert them into numeric values using the functions `DATE_DIFF` or `DURATION_TO_DAYS`.

Returns: The variance as a Number. If all values in `expression` are `NULL`, this function returns `NULL`.

Use as a Window Function

This function can also be used as a window function:

```
VARIANCE(<expression>) OVER (
    [ PARTITION BY <partitionExpression> [, <partitionExpression> ...] ]
    [ ORDER BY <orderExpression> [ { ASC | DESC } ] [, <orderExpression> ...] [
<windowFrame> ] ]
)
```

For more details about this syntax, refer to the Window Functions [page 220] overview.

Filtering the Input

The input data to this function can be filtered before aggregation takes place, allowing you to exclude specific cases or events from the result.

```
VARIANCE(<expression>) FILTER (WHERE <condition>)
```

See FILTER Clause [page 20] for more information.

Example 1

Assuming the following process data (`THIS_PROCESS`):

case_id	Order Amount
00001	20
00002	40
00003	40
00004	20
00005	5
00006	115
00007	null

The following query calculates the variance of all non-`NULL` order amounts.

```
SELECT VARIANCE("Order Amount")
FROM THIS_PROCESS
```

Result:

VARIANCE(Order Amount)
1275

Example 2

Because this function is also available as a window function, it's possible to calculate the variance of a window frame.

The following query calculates the variance of three order values, namely the one in the current row and those in the individual rows immediately preceding and following.

```
SELECT
    case_id,
    "Order Amount",
    VARIANCE("Order Amount")
```

```
        OVER (ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
        AS "Windowed Variance"
FROM THIS_PROCESS
```

Result:

case_id	Order Amount	Windowed Variance
00001	20	100
00002	40	88,889
00003	40	88,889
00004	20	205,556
00005	5	2,372,222
00006	115	3,025
00007	null	0

Related Information

Conversion Functions [page 154]
FILTER Clause [page 20]
Window Functions [page 220]

1.6.7.5 Non-Aggregate Functions

The functions in this section do not aggregate collections of values into a single value. They are applied to individual rows within a window.

For more general information about windows, refer to the Window Functions overview.

> ⚠ Caution
>
> For very large data sets, these functions may require excessive CPU activity, causing long query execution times.
>
> For more information, refer to Performance [page 294].

LAG [page 272]
Returns a single column value from the preceding row according to the window partition and sort criteria. When no preceding row exists, a null value is returned.

LEAD [page 273]
Returns a single column value from the succeeding row according to the window partition and sort criteria. When no succeeding row exists, a null value is returned.

OCCURRENCE [page 274]
Returns a list of numbers representing a running occurrence count of distinct items in an event list.

Related Information

Window Functions [page 220]

1.6.7.5.1 LAG

Returns a single column value from the preceding row according to the window partition and sort criteria. When no preceding row exists, a null value is returned.

Syntax

```
LAG(<columnName>) OVER (
    [ PARTITION BY <partitionExpression> [, <partitionExpression>, ... ] ]
    ORDER BY <sortExpression>
)
```

Parameter	Description
`columnName`	The column from the preceding row to be included in the current row.
`partitionExpression`	An expression determining how the rows are grouped.
`sortExpression`	An expression determining how each partition is sorted.

Example

This query partitions data by case ID, sorting each partition chronologically. Each row includes the case ID and event name along with two additional columns:

- The event name from the preceding row
- The difference between the timestamps of the preceding and current rows

The result shows the name of each event that occurred, the name of the preceding event and how much time separates the two events. For events having no predecessor, a null value is displayed.

```
SELECT
    case_id,
    LAG(event_name) OVER (PARTITION BY case_id ORDER BY end_time) AS Predecessor,
    event_name AS Event,
    end_time - LAG(end_time) OVER (PARTITION BY case_id ORDER BY end_time) AS
cycle_time
FROM FLATTEN(THIS_PROCESS)
```

Example

Example output:

case_id	Predecessor	Event	cycle_time
00001	null	Receive Customer Order	null
00001	Receive Customer Order	Change Order Quantity	14h 47m
00001	Change Order Quantity	Receive Payment	2d 13h
00001	Receive Payment	Ship Goods Express	17h 9m
00001	Ship Goods Express	Receive Delivery Confirmation	3d 23h
00002	null	Receive Customer Order	null
00002	Receive Customer Order	Receive Payment	1d 15h
00002	Receive Payment	Send items to Printing	18h 49m
00002	Send items to Printing	Order Canceled	1d 6h

1.6.7.5.2 LEAD

Returns a single column value from the succeeding row according to the window partition and sort criteria. When no succeeding row exists, a null value is returned.

Syntax

```
LEAD(<columnName>) OVER (
    [ PARTITION BY <partitionExpression> [, <partitionExpression>, ... ] ]
    ORDER BY <sortExpression>
)
```

Parameter	Description
`columnName`	The column from the succeeding row to be included in the current row.
`partitionExpression`	An expression determining how the rows are grouped.
`sortExpression`	An expression determining how each partition is sorted.

Example

This query partitions data by case ID, sorting each partition chronologically. Each row includes the case ID and event name along with two additional columns:

- The event name from the next row
- The difference between the timestamps of the current and succeeding rows

The result shows the name of each event that occurred, the name of the succeeding event and how much time separates the two events. For events having no successor, a null value is displayed.

```
SELECT
    case_id,
    event_name AS Event,
    LEAD(event_name) OVER (PARTITION BY case_id ORDER BY end_time) AS Successor,
    LEAD(end_time) OVER (PARTITION BY case_id ORDER BY end_time) - end_time AS
CycleTime
FROM FLATTEN(THIS_PROCESS)
```

Example

Example output:

case_id	Event	Successor	CycleTime
00001	Receive Customer Order	Change Order Quantity	14h 47m
00001	Change Order Quantity	Receive Payment	2d 13h
00001	Receive Payment	Ship Goods Express	17h 9m
00001	Ship Goods Express	Receive Delivery Confirmation	3d 23h
00001	Receive Delivery Confirmation	null	null
00002	Receive Customer Order	Receive Payment	1d 15h
00002	Receive Payment	Send items to Printing	18h 49m
00002	Send items to Printing	Order Canceled	1d 6h
00002	Order Canceled	null	null

1.6.7.5.3 OCCURRENCE

Returns a list of numbers representing a running occurrence count of distinct items in an event list.

The function operates at event level. When applied to an event column, it calculates a running total of occurrences for each distinct value in the event list. At each step, it adds to a list the number of item occurrences up to that point.

Syntax

```
<aggregateFunction> ( <occurrenceAlias> )
FROM ( SELECT OCCURRENCE( <expression> ) AS <occurrenceAlias> ) AS
<subqueryAlias>
```

Parameter	Description
`aggregateFunction`	The name of an aggregate function.
`occurrenceAlias`	An alias for the `OCCURRENCE` function's output for the aggregate function to use.
`expression`	An expression that evaluates to an event-level column.
`subqueryAlias`	An alias for the subquery.

Returns: A list of numbers containing the running totals of each distinct item in `expression`. Note that:

- If `expression` is `NULL`, then `OCCURRENCE` returns `NULL`.
- Any `NULL` values in the `expression` column are ignored.
- If the `expression` column contains only `NULL` values, `OCCURRENCE` returns `NULL`.

Behavior

To illustrate the function's behavior, let's imagine an event column containing six values: `["A", "B", "A", "C", "B", "A"]`. The following table shows how `OCCURRENCE` proceeds to calculate a return value for this column as it encounters each value:

Value Number	Value	Remark	Return Value of `OCCURRENCE` So Far
1	"A"	Encounters 1st occurrence of A	`[1]`
2	"B"	Encounters 1st occurrence of B	`[1, 1]`
3	"A"	Encounters 2nd occurrence of A	`[1, 1, 2]`
4	"C"	Encounters 1st occurrence of C	`[1, 1, 2, 1]`
5	"B"	Encounters 2nd occurrence of B	`[1, 1, 2, 1, 2]`
6	"A"	Encounters 3rd occurrence of A	`[1, 1, 2, 1, 2, 3]`

The function therefore returns the list `[1, 1, 2, 1, 2, 3]`.

Example 1

The `OCCURRENCE` function is useful for identifying cases containing rework, indicated by repeated events. The following query selects the ID and the list of event names for each case.

```
SELECT case_id, "event_name"
FROM THIS_PROCESS
```

```
WHERE (
    SELECT MAX(occ) FROM (
        SELECT OCCURRENCE(event_name) AS occ
    ) AS sub
) >= 4
```

The result set is filtered by the subquery in the `WHERE` clause. The `OCCURRENCE` function is applied to the `event_name` column, which returns a list of running occurrences of each event. The `MAX` function finds the largest number in this list, indicating the number of times the most repeated event occurred per case. Only cases containing an event that was worked more than or equal to four times are included in the result.

case_id	event_name
00018	Receive Customer Order
	Change Order Quantity
	Change Order Quantity
	Change Order Quantity
	Change Order Quantity
	Receive Payment
	Ship Goods Standard
	Ship Goods Standard
	Receive Delivery Confirmation
	Receive Delivery Confirmation
00098	Receive Customer Order
	Receive Payment
	Receive Payment
	Receive Payment
	Receive Payment
	Ship Goods Express
	Receive Delivery Confirmation
00238	Receive Customer Order
	Receive Payment
	Receive Payment
	Receive Payment
	Receive Payment
	Send items to Printing
	Order Canceled

A refinement of this example demonstrates this function's utility as a window function. Instead of the maximum, let's select the average event occurrence and restrict the result set to the case with ID '00098', the events from which are in the previous image.

```
SELECT
    case_id,
    (
        SELECT AVG(occ) FROM (
            SELECT OCCURRENCE(event_name) AS occ
        ) AS sub
    ) AS "Average"
FROM THIS_PROCESS
WHERE case_id = '00098'
```

Output:

case_id	Average
00098	1.857

The `OCCURRENCE` function determines for each value the number of occurrences up to the current point. The occurrence values for case '00098' are `[1, 1, 2, 3, 4, 1, 1]`. Consequently, the `AVG` function calculates case '00098' as 1.857, the sum of its occurrence values (13) divided by the number of elements (7).

Example 2

It's possible to use `OCCURRENCE` to search for specific events where rework occurred. The following snippet of code can be used in a complete query to select the number of times that a particular event occurs in each case. In this snippet, the event of interest is `'Change Order Quantity'`.

```
-- Event-level subquery
( SELECT MAX(occ)
    FROM (
        -- General subquery
        SELECT CASE
            WHEN event_name = 'Change Order Quantity' THEN
OCCURRENCE(event_name)
            ELSE NULL
        END AS occ
    ) as sub
) as "Times order size changed"
```

The inner, general subquery creates a table of values containing a single column named `occ`. These values are the result of applying the `OCCURRENCE` function to events named `'Change Order Quantity'` within a single case.

The outer, event-level subquery selects from this table. Being an event-level subquery, its selected data must be aggregated somehow when incorporated into a complete query. Since this example seeks the number of occurrences within a case, the maximum value is chosen.

In addition to selection, this same logic can be applied within filtering. The following complete query uses the snippet in a filter and compares the result of `MAX(occ)` to a desired minimum value.

```
SELECT case_id,
    event_name,
    (
```

```
        SELECT MAX(occ)
        FROM (
            SELECT CASE
                WHEN event_name = 'Change Order Quantity' THEN
OCCURRENCE(event_name)
                ELSE NULL
            END AS occ
        ) as sub
    ) as "Times order size changed"
FROM THIS_PROCESS
WHERE (
    SELECT MAX(occ)
    FROM (
        SELECT CASE
            WHEN event_name = 'Change Order Quantity' THEN
OCCURRENCE(event_name)
            ELSE NULL
        END AS occ
    ) as sub
) >= 3
```

In this case, the minimum value is three. This has the effect of filtering out all cases containing fewer than three instances of the `'Change Order Quantity'` event.

Let's apply this query to some example data. Assume the following test data (`THIS_PROCESS`):

case_id	event_name
00012	Receive Customer Order
	Change Order Quantity
	Change Order Quantity
	Change Order Quantity
	Receive Payment
	Receive Payment
	Ship Goods Standard
	Receive Delivery Confirmation
00013	Receive Customer Order
	Ship Goods Express
	Receive Payment
	Receive Delivery Confirmation
00014	Receive Customer Order
	Change Order Quantity
	Change Order Quantity
	Ship Goods Standard
	Receive Payment
	Receive Delivery Confirmation

case_id	event_name
00018	Receive Customer Order
	Change Order Quantity
	Change Order Quantity
	Change Order Quantity
	Change Order Quantity
	Receive Payment
	Ship Goods Standard
	Ship Goods Standard
	Receive Delivery Confirmation
	Receive Delivery Confirmation

The query returns the following result:

case_id	event_name	Times order size changed
00012	Receive Customer Order	3
	Change Order Quantity	
	Change Order Quantity	
	Change Order Quantity	
	Receive Payment	
	Receive Payment	
	Ship Goods Standard	
	Receive Delivery Confirmation	
00018	Receive Customer Order	4
	Change Order Quantity	
	Change Order Quantity	
	Change Order Quantity	
	Change Order Quantity	
	Receive Payment	
	Ship Goods Standard	
	Ship Goods Standard	
	Receive Delivery Confirmation	
	Receive Delivery Confirmation	

Related Information

Aggregate Functions [page 92]
Event-Level Subqueries [page 52]
RANK [page 283]

1.6.7.6 Ranking Functions

The functions in this section are window functions that sort data partitions and assign a rank value for each row within a result set.

Using these functions, you can filter the results and sort them for consistent stacking in visualizations such as stacked bar charts.

> ⚠ Caution
>
> For very large data sets, these functions may require excessive CPU activity, causing long query execution times.
>
> For more information, refer to Performance [page 294].

DENSE_RANK [page 280]
: Returns the rank of each row in a result set based on the order defined in the OVER clause for each partition

RANK [page 283]
: Returns the rank of each row in a result set based on the order defined in the OVER clause for each partition.

ROW_NUMBER [page 284]
: Returns the calculated row number based on partitioned and sorted set of values.

1.6.7.6.1 DENSE_RANK

Returns the rank of each row in a result set based on the order defined in the OVER clause for each partition

The DENSE_RANK function assigns the same ranking for rows with identical values and doesn't skip the rank positions for these identical rows. For example, rows with rank values, 1, 2, 2, 2, 3, 4. The rank value of the next nonidentical row will have the succeeding rank.

Syntax

```
DENSE_RANK() OVER (
    [ PARTITION BY <expression> [, <expression>, ...] ]
    [ ORDER BY <expression> [ { ASC | DESC } ] [, <expression>, ...] ]
```

```
)
```

Parameters	Description
`expression`	The column or argument determining how the rows are grouped and how each partition is sorted. Mandatory if `PARTITION BY` or `ORDER BY` clauses are included in the query.

`PARTITION BY`: Splits the result set into partitions. Optional.

`ORDER BY`: Sorts each partition based on the defined criteria. Optional. If `ORDER BY` clause isn't included in your query, each row will have the same rank value, one.

`ASC | DESC`: `ASC` sorts the result set in ascending order and `DESC` sorts it in descending order based on the expressions defined in `ORDER BY`.

Example 1

In the following query, the cities are first partitioned by the type of good and sorted by the count of high value orders per '`Type of Goods`'. The query then returns the cities with order amount greater than 1500 ranked in descending order.

```
SELECT "Type of Goods",
       "City",
       count(case_id) AS "Orders > $1500",
       ROW_NUMBER() OVER (PARTITION BY "Type of Goods" ORDER BY count(case_id)
DESC) AS "ROW_NUMBER",
       RANK() OVER (PARTITION BY "Type of Goods" ORDER BY count(case_id) DESC)
AS "RANK",
       DENSE_RANK() OVER (PARTITION BY "Type of Goods" ORDER BY count(case_id)
DESC) AS "DENSE_RANK"
FROM THIS_PROCESS
WHERE "Order Amount" > 1500
GROUP BY 1,2
```

Example

Result:

Type of Goods	City	ROW_NUMBER	RANK	DENSE_RANK	Orders > $1500
T-shirt with Print	Boston	1	1	1	1
Hoody	Houston	1	1	1	8
	Boston	2	2	2	5
	New York	3	3	3	3
	San Francisco	4	4	4	2
	Washington	5	4	4	2
	Miami	6	6	5	1
Hoody with Print	Houston	1	1	1	18

Type of Goods	City	ROW_NUMBER	RANK	DENSE_RANK	Orders > $1500
	New York	2	2	2	15
	San Francisco	3	3	3	4
	Washington	4	4	4	3
	Miami	5	4	4	3
	Boston	6	6	5	1

Example 2

The following query partitions the result set by the type of goods and returns the list of cities with order amount greater than 1500 within a partition. In the result set, the `RANK` and `DENSE_RANK` of all the returned rows is one as the `ORDER BY` clause isn't included in the query.

```
SELECT "Type of Goods",
       "City",
        count(case_id) AS "Orders > $1500",
        ROW_NUMBER() OVER () AS "ROW_NUMBER",
        RANK() OVER () AS "RANK",
        DENSE_RANK() OVER () AS "DENSE_RANK"
FROM THIS_PROCESS
WHERE "Order Amount" > 1500
```

Example

Result:

Type of Goods		ROW_NUMBER	RANK	DENSE_RANK	Orders > $1500
Hoody with Print	Houston	1	1	1	18
	New York	4	1	1	15
	Washington	5	1	1	3
	Miami	7	1	1	3
	Boston	8	1	1	1
	San Francisco	9	1	1	4
Hoody	Miami	2	1	1	1
	San Francisco	6	1	1	2
	Houston	10	1	1	8
	Boston	11	1	1	5
	Washington	12	1	1	2
	New York	13	1	1	3
T-shirt with Print	Boston	3	1	1	1

1.6.7.6.2 RANK

Returns the rank of each row in a result set based on the order defined in the `OVER` clause for each partition.

The `RANK` function assigns the same ranking for rows with identical values and skips the rank positions for these identical rows. For example, rows with rank values, 1, 2, 2, 2, 5, 6. The rank value of the next nonidentical row depends on the number of rows with same ranking. The number of rows with the same ranking determines the number of rank values to be skipped.

Syntax

```
RANK() OVER (
    [ PARTITION BY <expression> [, <expression>, ...] ]
    [ ORDER BY <expression> [ { ASC | DESC } ] [, <expression>, ...] ]
)
```

Parameters	Description
`expression`	The column or argument determining how the rows are grouped and how each partition is sorted. Mandatory if `PARTITION BY` or `ORDER BY` clauses are included in the query.

`PARTITION BY`: Splits the result set into partitions. Optional.

`ORDER BY`: Sorts each partition based on the defined criteria. Optional.

`ASC | DESC`: `ASC` sorts the result set in ascending order and `DESC` sorts it in descending order based on the expressions defined in `ORDER BY`.

Example

In the following query, the cities are first partitioned by the type of good and sorted by the count of high value orders per '`Type of Goods`'. The query then returns the cities with order amount greater than 1500 ranked in descending order.

```
SELECT "Type of Goods",
       "City",
        count(case_id) AS "Orders > $1500",
       ROW_NUMBER() OVER (PARTITION BY "Type of Goods" ORDER BY count(case_id)
DESC) AS "ROW_NUMBER",
       RANK() OVER(PARTITION BY "Type of Goods" ORDER BY count(case_id) DESC) AS
"RANK"
FROM THIS_PROCESS
WHERE "Order Amount" > 1500
GROUP BY 1, 2
```

> ✤ Example
>
> *Result:*

Type of Goods	City	ROW_NUMBER	RANK	Orders > $1500
T-shirt with Print	Boston	1	1	1
Hoody	Houston	1	1	8
	Boston	2	2	5
	New York	3	3	3
	San Francisco	4	4	2
	Washington	5	4	2
	Miami	6	6	1
Hoody with Print	Houston	1	1	18
	New York	2	2	15
	San Francisco	3	3	4
	Washington	4	4	3
	Miami	5	4	3
	Boston	6	6	1

1.6.7.6.3 ROW_NUMBER

Returns the calculated row number based on partitioned and sorted set of values.

The ranking happens sequentially based on the order defined in the `OVER` clause for each partition. The rank value is different even if the rows contain the same values.

Syntax

```
ROW_NUMBER() OVER (
    [ PARTITION BY <expression> [, <expression>, ...] ]
    [ ORDER BY expression [ { ASC | DESC } ] [, <expression>, ...] ]
)
```

Parameters	Description
`expression`	The column or argument that determines how the rows are grouped and how each partition is sorted. Mandatory if `PARTITION BY` or `ORDER BY` clauses are included in the query.

`PARTITION BY`: Splits the result set into partitions. Optional.

`ORDER BY`: Sorts each partition based on the defined criteria. Optional.

`ASC | DESC`: `ASC` sorts the result set in ascending order and `DESC` sorts it in descending order based on the expressions defined in `ORDER BY`.

Example

In the following query, the order amount is first partitioned based on the city and the type of good. The query then returns the order amount ranked in descending order in the result set.

```
SELECT "Type of Goods",
       "City",
       "Order Amount",
       ROW_NUMBER() OVER (PARTITION BY "Type of Goods", "City"  ORDER BY "Order
Amount" DESC)
FROM THIS_PROCESS
WHERE "Order Amount" > 1500
ORDER BY 1, 2, 4
```

Result:

Type of Goods	City	Order Amount	ROW_NUMBER() OVER()
Hoody	Boston	1,738.26	1
Hoody	Boston	1,732.03	2
Hoody	Boston	1,721.64	3
Hoody	Boston	1,684.40	4
Hoody	Boston	1,528.04	5
Hoody	Houston	1,719.93	1
Hoody	Houston	1,644.87	2
Hoody	Houston	1,612.53	3
Hoody	Houston	1,539.99	4
Hoody	Houston	1,539.11	5

1.6.8 BUCKET Function

Learn about the BUCKET() function in SIGNAL, the process mining query language of SAP Signavio Process Intelligence.

The BUCKET() function calculates the indexes of values in a range of values (as buckets) that are equal in size. The bucket indexes can then be applied to an aggregate function, for example a COUNT() function, which counts the values within that bucket.

The BUCKET() function uses following parameters to calculate the bucket indexes as a positive integer number:

- The value to be bucketed (expression)
- The minimum value within the result range (min)
- The fixed width for each bucket (bucket_width)
- The total number of consecutive, non-overlapping buckets (#_inlier_buckets)

Syntax:

```
BUCKET(expression, min, bucket_width, #_inlier_buckets)
```

Parameter	Description
expression	Specify a column with numeric or duration values in the expression.
min	The starting value of the bucket. The values must be either numeric or duration values.
bucket_width	The width of the bucket. The value must be either a positive numeric or positive duration [page 6] that is greater than zero.
#_inlier_buckets	The number of required buckets. The value must be a positive integer that is greater than or equal to one.

To understand the range of values in the bucket, you can first determine the bucket boundaries and then apply the boundaries to the BUCKET() function. To determine the bucket boundaries and then apply the boundaries to the BUCKET() function, use the following pattern:

```
SELECT
IF (b = 0, NULL, ((b - 1) * bucket_width) + min) AS bucket_start,
IF (b = #_inlier_buckets + 1, NULL, (b * bucket_width) + min) AS bucket_end,
value
FROM (
SELECT BUCKET(expression, bucket_start, bucket_width, #_inlier_buckets) AS b,
COUNT(1) AS value
FROM ...
) AS sub
```

The bucket boundaries pattern isn't required for the BUCKET() function to work. It provides a useful way to understand the range of values in the bucket. The pattern defines the minimum and maximum values being aggregated in the bucket.

Example 1:

The following example calculates the bucket boundaries (bucket_start and bucket_end). Then aggregates the total number of cases based on 'Order Amount'. The cases are bucketed into intervals of 100€ (bucket_width).

The results are then calculated for 100 buckets (#_inlier_buckets) where the 'Order Amount' is greater than 1€ (min) and less than 10001€ (max). The maximum value is calculated with the following formula: min + bucket width x #_inlier_buckets = 1+ 100 x 100 = 10001.

```
SELECT
total_cases,
bucket_id,
IF (bucket_id > 100, -1, ((bucket_id-1) * 100) + 0) AS bucket_start,
IF (bucket_id > 100, -1, (bucket_id * 100) + 0) AS bucket_end
FROM (
SELECT
BUCKET("Order Amount", 1, 100,100) as bucket_id,
COUNT(case_id) as total_cases
FROM FLATTEN(THIS_PROCESS)
WHERE "Order Amount" is not null
GROUP BY 1
ORDER BY 1
) as sub
```

Result:

bucket_id	bucket_start	bucket_end	total_cases
9	800	900	550
10	900	1,000	11,768
11	1,000	1,100	12,221
12	1,100	1,200	485
15	1,400	1,500	2,362
16	1,500	1,600	2,039

Example 2:

The following example aggregates the total number of cases based on the 'Order Amount'. The cases are bucketed into intervals of 100€ (bucket_width). The results are calculated for 10 buckets (#_inlier_buckets) where the 'Order Amount' is greater than 1€ (min) and less than 1001€ (max). The maximum value is calculated with the following formula: min + bucket width x #_inlier_buckets = 1 + 100 x 10 = 1001

The result displays in a breakdown widget. The values on the X axis represent the bucket ID. Buckets that contain at least one case within them are displayed. The cases that have a bucket value of 10 but exceed the maximum value are added into the outlier bucket. The outlier bucket has a bucket ID of 11.

```
SELECT
BUCKET("Order Amount", 1, 100,10) as order_amount,
COUNT(case_id)
FROM FLATTEN(THIS_PROCESS)
WHERE "Order Amount" is not null
GROUP BY 1
ORDER BY 1
```

Result:

1.7 Keywords

Keywords are words with special significance in SIGNAL, meaning they cannot be used as identifiers in a query. If you wish to use a keyword as an identifier, you must enclose it in double quotation marks.

For example, the query

```
SELECT SUM("Order Value") AS Sum FROM THIS_PROCESS
```

is invalid because it tries to give the selected column an alias of '`Sum`'. However, '`SUM`' – being the name of a function – is a SIGNAL keyword. (SIGNAL keywords are case-insensitive, so the difference in case is of no consequence.) To use '`Sum`' as an identifier, you would have to enclose it in double quotation marks, like so:

```
SELECT SUM("Order Value") AS "Sum" FROM THIS_PROCESS
```

The following alphabetically organized lists contain all SIGNAL keywords.

A

- ABS
- ALL
- ANALYZE
- AND
- ANY
- AS
- ASC
- AVG

B

- BARRIER
- BEHAVIOR
- BEHAVIOUR
- BETWEEN
- BOOL_AND
- BOOL_OR
- BUCKET
- BY

C

- CASE
- CASE_ID
- CATEGORY
- CEIL
- CHAR_INDEX
- CHAR_LENGTH
- COALESCE
- CONCAT
- COUNT
- CREATE
- CURRENT

D

- DATE_ADD
- DATE_DIFF
- DATE_PART
- DATE_TRUNC
- DEFAULT
- DENSE_RANK
- DESC
- DESCRIBE
- DISTINCT
- DROP
- DURATION
- DURATION_BETWEEN
- DURATION_FROM_DAYS
- DURATION_FROM_MILLISECONDS
- DURATION_TO_DAYS
- DURATION_TO_MILLISECONDS

E

- ELSE
- END
- END_TIME
- EVENT_ID

- EVENT_NAME
- EVENTS
- EXACT
- EXPLAIN
- EXTERNAL

F

- FALSE
- FILL
- FILTER
- FIRST
- FLATTEN
- FLOOR
- FOLLOWING
- FORMAT
- FROM

G

- GRANT
- GREATEST
- GROUP

H

- HAVING

I

- IF
- ILIKE
- IN
- INVOKER
- IS

J

- JOIN
- JSON

L

- LAG
- LAST
- LEAD
- LEAST
- LEFT
- LIKE
- LIMIT
- LOCATION
- LOG
- LOWER
- LTRIM

M

- MATCHES
- MAX
- MEDIAN
- MIN

N

- NOT
- NOW
- NULL
- NULLIF
- NULLS

O

- OCCURRENCE
- ODATA
- OFFSET
- ON
- ONLY
- OR
- ORDER
- OUTER
- OVER

P

- PARQUET
- PARTITION
- PERCENT
- PERCENTILE_CONT
- PERCENTILE_DESC
- PERMISSIONS
- POW
- PRECEDING
- PRIVATE
- PUBLIC

R

- RANGE
- RANK
- REGR_INTERCEPT
- REGR_SLOPE
- REPEATABLE
- REPLACE
- REVERSE
- RIGHT
- ROUND
- ROW
- ROW_NUMBER
- ROWS

- RTRIM

S

- SECURITY
- SELECT
- SIGN
- SQRT
- STDDEV
- SUBSTRING
- SUBSTRING_AFTER
- SUBSTRING_BEFORE
- SUM

T

- TABLE
- TABULAR
- TEXT
- THEN
- TIMESERIES
- TIMESTAMP
- TO
- TO_NUMBER
- TO_STRING
- TO_TIMESTAMP
- TRIM
- TRIMMED_AVG
- TRIMMED_STDDEV
- TRIMMED_VARIANCE
- TRUE
- TRUNC

U

- UNBOUNDED
- UNION
- UPPER

- USING

V

- VARIANCE
- VIEW

W

- WHEN
- WHERE
- WITH
- WITHIN

1.8 Performance

The SIGNAL Engine processes queries and can handle very large data sets rapidly, although some factors can affect performance.

Event Log Size

The SIGNAL Engine performs numerous optimizations automatically to maximize query performance. In general, the engine can comfortably handle event logs consisting of **up to 1 billion entries**.

For a certain subset of language features, performance can suffer when operating on a data set of very large size. This can compromise the execution time of the containing query. A function or operator is considered vulnerable to performance issues if its execution time tends to be **greater than five seconds when running on a data set of 1 billion events**. All functions and operators prone to such performance issues are flagged accordingly in their documentation.

Timeout

In SIGNAL, the timeout length of a query is **10 minutes**. Once a query has run for 10 minutes without result, that query is terminated and an error is returned.

→ Tip

If you receive a timeout error, refreshing your browser triggers a restart of the query. A result may be returned by this subsequent attempt, however it isn't guaranteed.

2 Tutorial

Learn about using SIGNAL in the following tutorial, the process mining query language of SAP Signavio Process Intelligence.

This tutorial supports you to get started with SIGNAL.

Based on a sample process with test data, this tutorial introduces you to the main principles of SIGNAL, starting from simple case-attribute based queries to more complex event-based queries.

The tutorial is structured as follows:

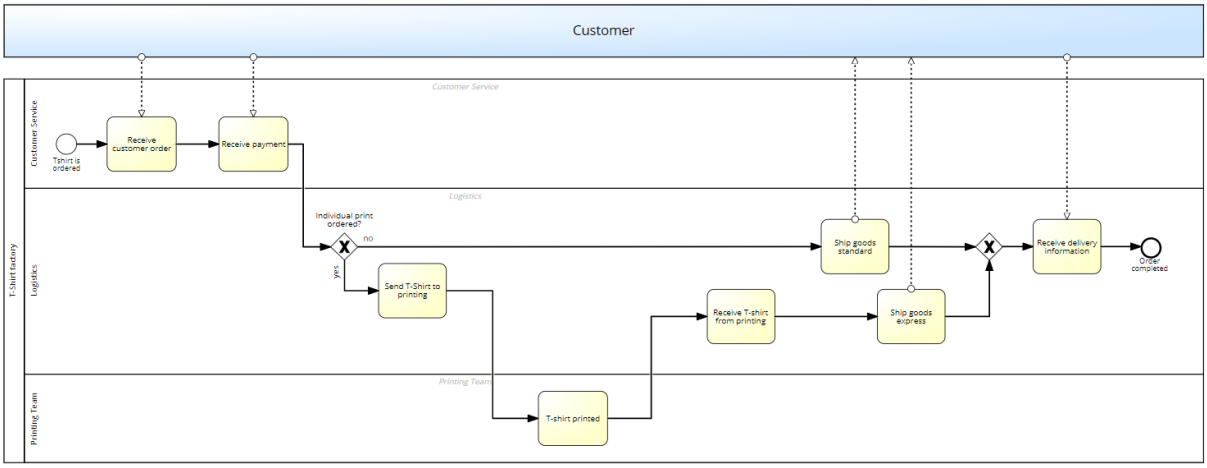
Section	Examples	Learning success	Introduced keywords
Count cases and cities	• How many cases exist for this process? • How many different cities are involved in this process? • How many cases exist for each city? • How many cases exist for New York and Miami?	• Count a case attribute • Count the distinct records of a case attribute • Count two or more case attributes • Rename case attribute names with alias names • Sort the result set • Filter a result set to include only records that fulfill a specified condition	COUNT COUNT DISTINCT AS ORDER BY WHERE IN
Analyze order amounts	• What is the average order amount of this process? • What is the average order amount in Houston? • What is the total order amount in Boston? • What is the percentage order amount in Boston compared to the total order amount?	• Determine the average value of a case attribute • Determine a filtered average case attribute • Sum up a case attribute • Determine the percentage value of a case attribute compared to the overall value • Apply filter condition(s) within a query	AVG SUM FILTER
Determine process cycle times	• How long is the average cycle time of all cases? • How long is the average cycle time by city? • What are the maximum / minimum cycle times by city?	• Calculate cycle times • Perform subqueries on event attributes • Determine cycle time by case attribute • Calculate the largest / smallest values	MAX MIN

Section	Examples	Learning success	Introduced keywords
Investigate events	• How many cases have been closed / canceled? • What is the drop-out rate? • How many cases follow the standard process? • How many cases are canceled although the T-shirts have been sent for printing?	• Counting the total number of events • Filter for not true conditions • Determine the process conformance	IF NOT MATCHES

2.1 Understand the sample process

Understand the sample process used in the SIGNAL tutorial.

This tutorial is using the following sample process:



> **Note**
>
> For a more detailed view of the process flow, check the *Process Discovery* widget of the sample process.

In this tutorial, the following case and event attributes are used:

- Case attributes
  - Case ID
  - City
  - Order Amount in EUR
- Event attributes
  - EventName

- Timestamp

Example (extract):

Imported data

CaseId Type Case ID	EventName Type 'ENT_NAME	Timestamp Type JD_TIME	City Type Choice	Customer ID Type Choice
10100	Receive Customer Order	2017-08-08 22:52	Houston	10100
10100	Receive Customer Order	2017-08-08 22:52	Houston	10100
10100	Receive Payment	2017-08-12 13:04	Houston	10100
10100	Receive Payment	2017-08-12 13:04	Houston	10100
10100	Ship Goods Standard	2017-08-16 02:20	Houston	10100
10100	Ship Goods Standard	2017-08-16 02:20	Houston	10100
10100	Receive Delivery Confirmation	2017-08-22 05:18	Houston	10100
10100	Receive Delivery Confirmation	2017-08-22 05:18	Houston	10100
10101	Receive Customer Order	2017-07-12 13:30	San Francisco	10099
10101	Receive Customer Order	2017-07-12 13:30	San Francisco	10099
10101	Receive Payment	2017-07-15 07:15	San Francisco	10099
10101	Receive Payment	2017-07-15 07:15	San Francisco	10099
10101	Ship Goods Standard	2017-07-19 08:03	San Francisco	10099
10101	Ship Goods Standard	2017-07-19 08:03	San Francisco	10099
10101	Receive Delivery Confirmation	2017-08-06 01:47	San Francisco	10099
10101	Receive Delivery Confirmation	2017-08-06 01:47	San Francisco	10099
10102	Receive Customer Order	2017-06-26 17:33	Washington	10198

> **Note**
>
> For more detailed information about the test data, check them under *Process Settings > Data* in the sample process.

2.2 Count cases and cities

Read about the overview of case attributes used in the SIGNAL tutorial.

You want to get an overview about the case attributes of the process: How many cases exist for this process? How many different cities are involved in this process? How many cases exist for each city? How many cases exist for New York and Miami?

Example 1: How many cases exist for this process?

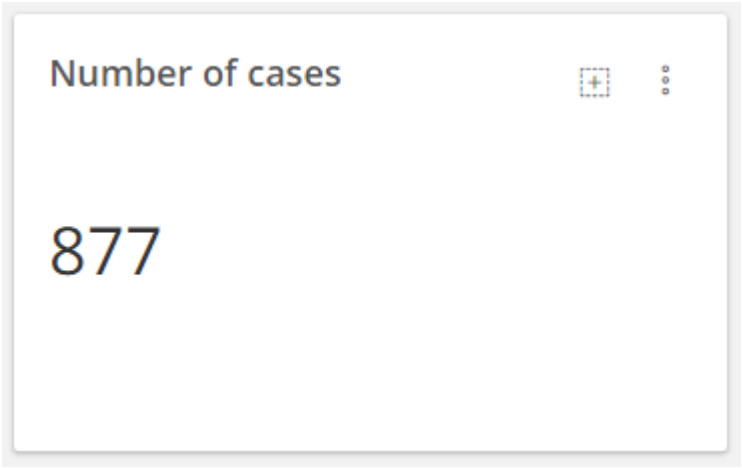
Learning success: Count a case attribute.

Query instruction: Count (keyword: COUNT) all cases (expression: case_id) in this process.

SIGNAL syntax:

```
SELECT
COUNT(case_id)
FROM THIS_PROCESS
```

Query result: The total number of cases, displayed in a *Value* widget.



Example 2: How many different cities are involved in this process?

Learning success: Count the distinct records of a case attribute.

Query instruction: Count (keyword: COUNT) all cities (expression: City) in this process. Unlike example 1, only distinct (keyword: DISTINCT) records are counted.

SIGNAL syntax:

```
SELECT
COUNT(DISTINCT City)
FROM THIS_PROCESS
```

Query result: The total number of distinct cities, displayed in a *Value* widget:

Example 3: How many cases exist for each city?

Learning success:

- Count two or more case attributes.
- Rename case attribute names with alias names.
- Sort the result set.

Query instruction: Count (keyword: COUNT) the number of cases (expression: case id) by city (expression: City). Rename the case attribute names with aliases (keyword: AS) to "Case Number" and "Site". This makes labels for widgets easier to understand. Finally sort the result set by Case numbers (keywords: ORDER BY 1) in descending order (keyword: DESC).

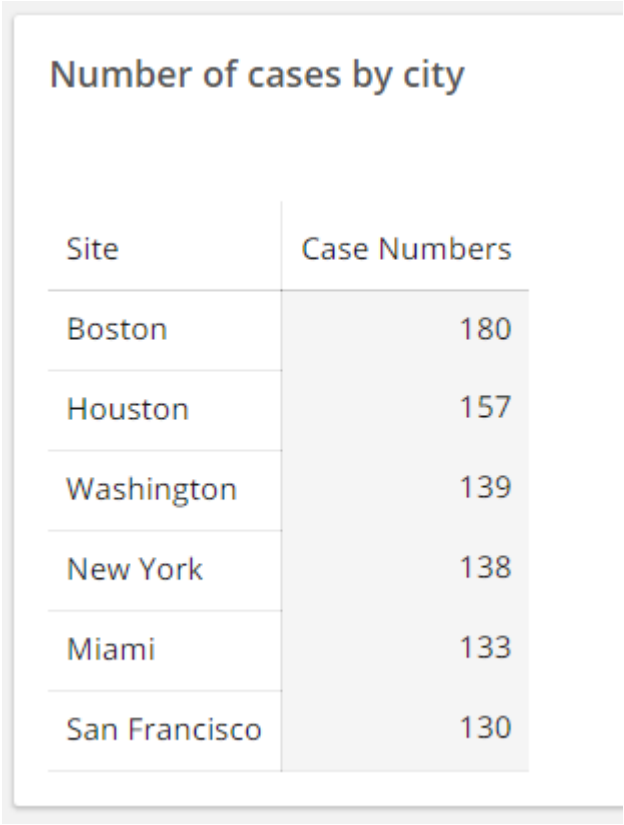
The order expression 1 selects the column, which has to be sorted. You can sort in ascending order (keyword: ASC), descending order (keyword: DESC), by null values first (keyword: NULLS FIRST,) or by null values last (keyword: NULLS LAST).

SIGNAL syntax:

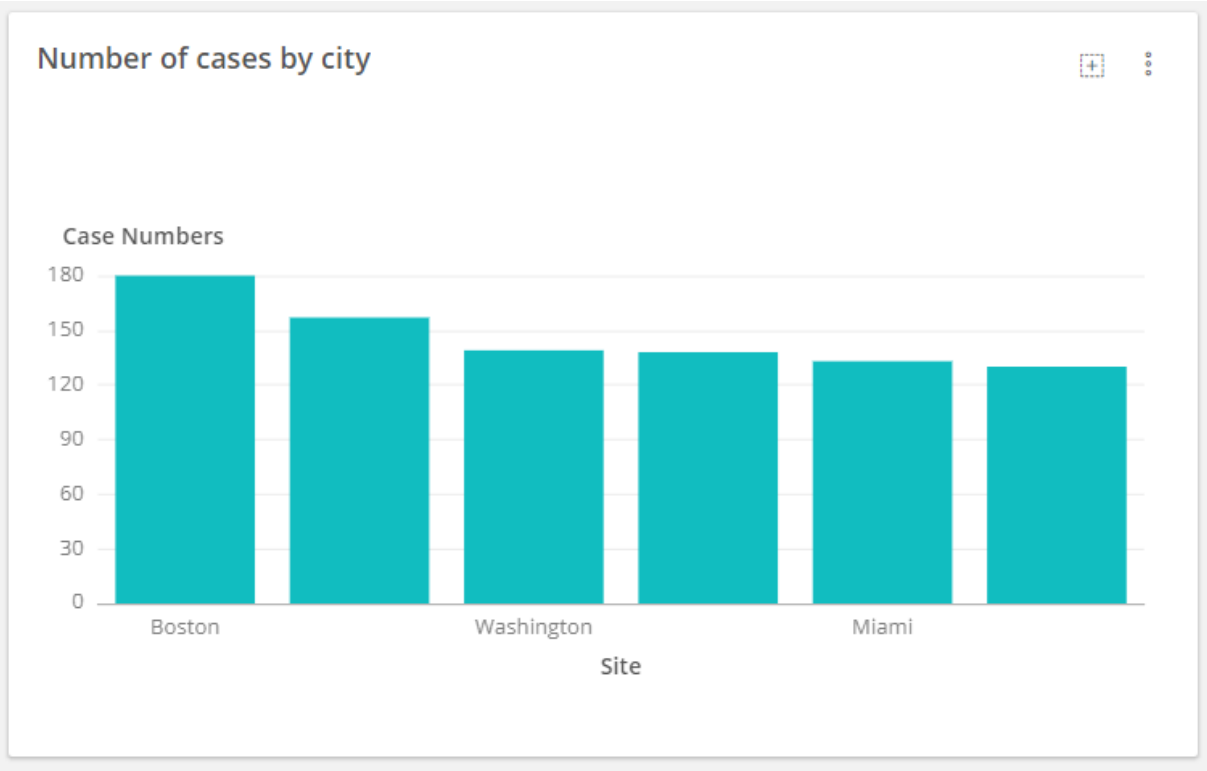
```
SELECT
COUNT(case_id) AS "Case Numbers", City AS "Site"
FROM THIS_PROCESS
ORDER BY 1 DESC
```

Query display: The total number of cases by city, displayed in a SIGNAL table widget and in a *Breakdown* widget.

Table:



Breakdown (bar chart):



Example 4: How many cases exist for New York and Miami?

Learning success: Filter a result set to include only records that fulfill a specified condition.

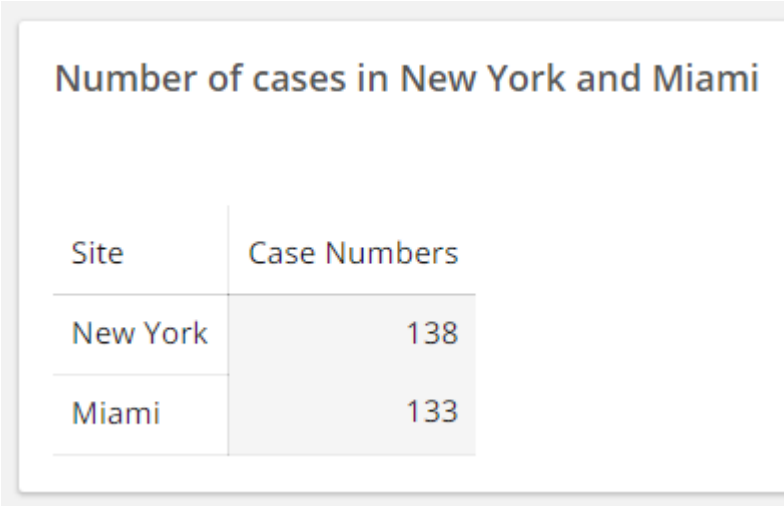
Query instruction: This query is very similar to example 3, but in this case, you do not query all cases but only the cases for New York and Miami. To filter the result, introduce the filter (keyword: WHERE) and specify the filter condition (keyword: IN + expression: 'New York', 'Miami'). Sort the result set by cities (keyword: ORDER BY 1), in descending order(keyword: DESC).

SIGNAL syntax:

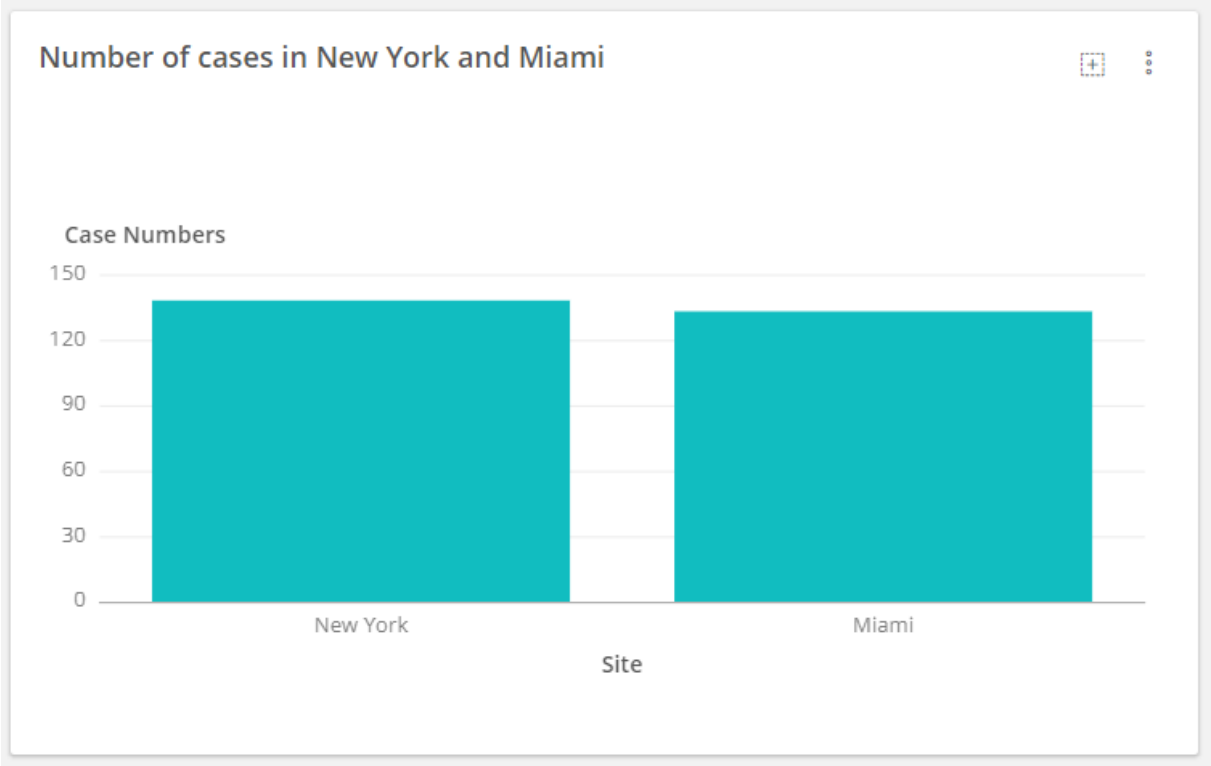
```
SELECT
COUNT(case_id) AS "Case Numbers", "City" AS "Site"
FROM THIS_PROCESS
WHERE City IN('New York', 'Miami')
ORDER BY 1 DESC
```

Query display: The total number of cases by filtered cities, displayed in a SIGNAL table widget and in a *Breakdown* widget.

Table:



Breakdown (bar chart):



2.3 Analyze order amounts

You want to get insights about the order amount: What is the average order amount of this process? What is the average order amount in Houston? What is the total order amount in Boston? What is the order amount in Boston related to the total order amount?

Example 1: What is the average order amount of this process?

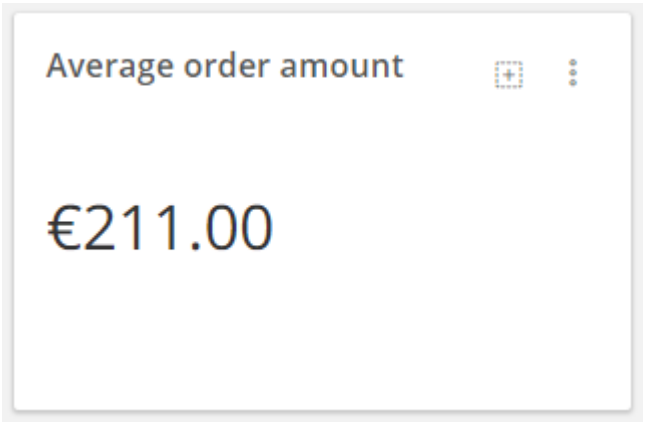
Learning success: Determine the average value of a case attribute.

Query instruction: Determine the average value (keyword: AVG) for the order amount(expression: Order Amount in EUR).

SIGNAL syntax:

```
SELECT
AVG("Order Amount in EUR")
FROM THIS_PROCESS
```

Query result: The aggregated value, displayed in a *Value* widget:



Example 2: What is the average order amount in Houston?

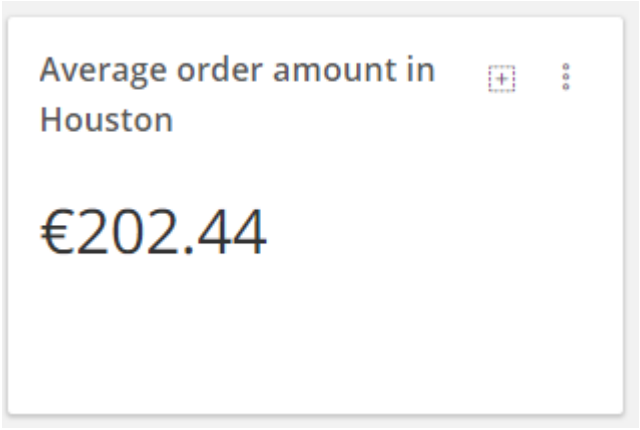
Learning success: Determine a filtered average case attribute.

Query instruction: Determine the average order amount (see example 1). Introduce the filter (keyword: WHERE) and specify the filter condition (expression: "City"='Houston').

SIGNAL syntax:

```
SELECT
AVG("Order Amount in EUR")
FROM THIS_PROCESS
WHERE("City"='Houston')
```

Query result: The average order amount in Houston, displayed in *Value* widget:



Example 3: What is the total order amount in Boston?

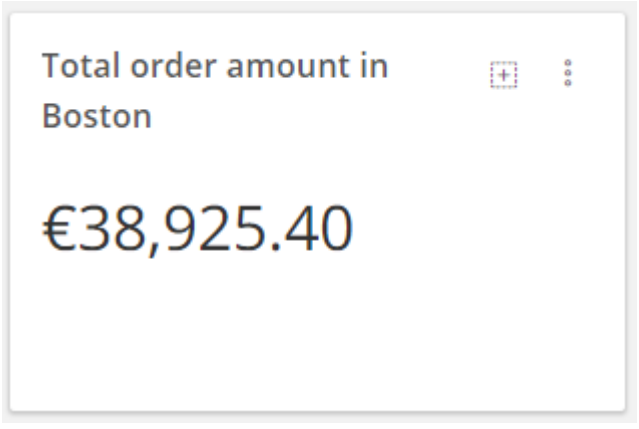
Learning success: Sum up a case attribute.

Query instruction: Sum up (keyword: SUM) the total order amount (expression: "Order Amount in EUR"). Introduce the filter (keyword: WHERE) and specify the filter condition (expression: "City"='Boston').

SIGNAL syntax:

```
SELECT
SUM("Order Amount in EUR")
FROM THIS_PROCESS
WHERE("City"='Boston')
```

Query result: The total order amount in Boston, displayed in a *Value* widget:



Example 4: What is the percentage order amount in Boston compared to the total order amount?

Learning success:

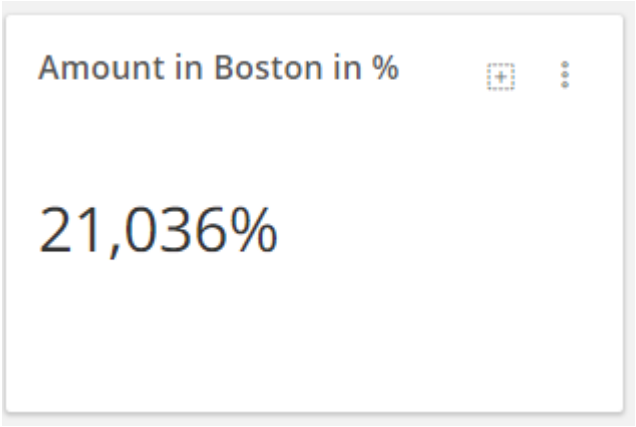
- Determine the percentage value of a case attribute compared to the overall value.
- Apply filter condition(s) within a query.

Query instruction: Sum up the order amount in Boston (see example 3). Unlike example 3, you cannot apply the filter condition as the last step. You have to filter (keyword: FILTER + filter condition: (WHERE "City"='Boston')) the result set before you can calculate the percentage value.

SIGNAL syntax:

```
SELECT
SUM("Order Amount in EUR")
FILTER (WHERE "City"='Boston')
/SUM("Order Amount in EUR")
* 100
FROM THIS_PROCESS
```

Query result: The percentage order amount of Boston, displayed in a *Value* widget:



2.4 Determine case cycle times

Read about how cycle times are determined in the SIGNAL tutorial.

You want to determine the cycle times of your process: How long is the average cycle time of all cases? How long is the average cycle time by city? What are the maximum / minimum cycle times by city?

Example 1: How long is the average cycle time of all cases?

Learning success:

- Calculate cycle times
- Perform subqueries on event attributes

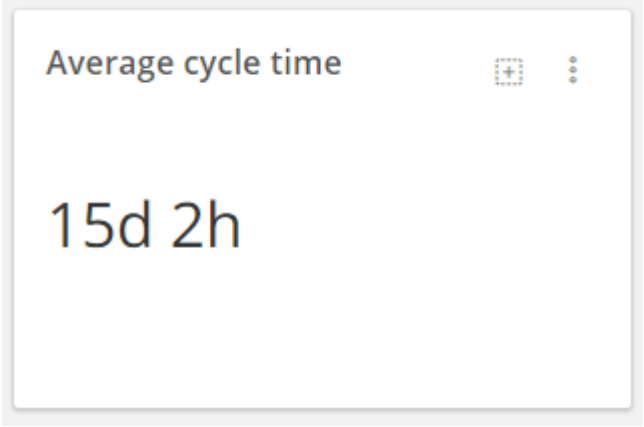
Query instruction: You have to calculate the cycle times first and then aggregate them to an average value. The cycle times are calculated from the event-based timestamps, so you have to perform a subquery (keyword: (SELECT).

To calculate the cycle time, you have to subtract the first event timestamp (keyword: SELECT FIRST + expression: end_time) from the last event timestamp (keyword: SELECT LAST + expression: end_time). From these values, you aggregate the average value (keyword: AVG) .

SIGNAL syntax:

```
SELECT
AVG(
(SELECT LAST(end_time))
-
(SELECT FIRST(end_time)))
FROM THIS_PROCESS
```

Query result: The average cycle time, displayed in a *Value* widget:



Example 2: How long is the average cycle time by city?

Learning success: Determine cycle time by case attribute.

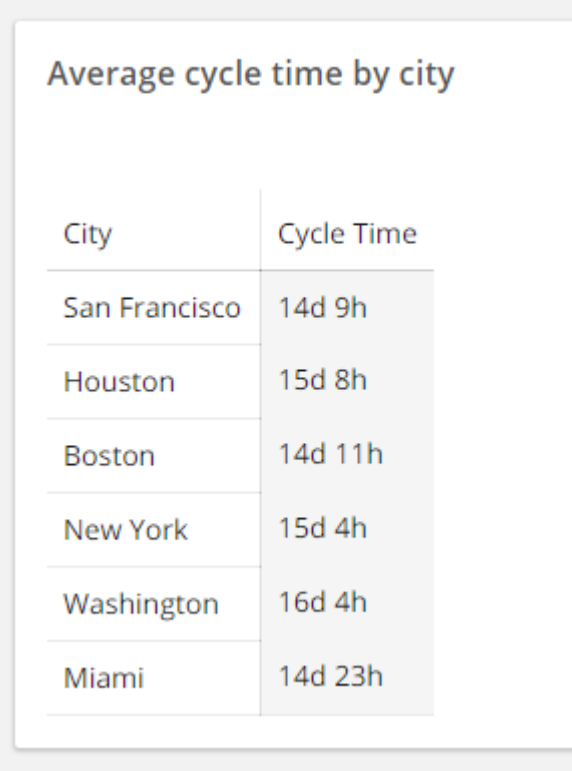
Query instruction: Determine the average cycle time (see example 1) by city (expression: City). Rename the case attribute with an alias (keyword: AS) to "Cycle Time".

SIGNAL syntax:

```
SELECT
AVG(
(SELECT LAST(end_time))
-
(SELECT FIRST(end_time))) AS "Cycle Time", "City"
```

```
FROM THIS_PROCESS
```

Query result: The average cycle times by city, displayed in a SIGNAL table widget:



Example 3: What are the maximum / minimum cycle times by city?

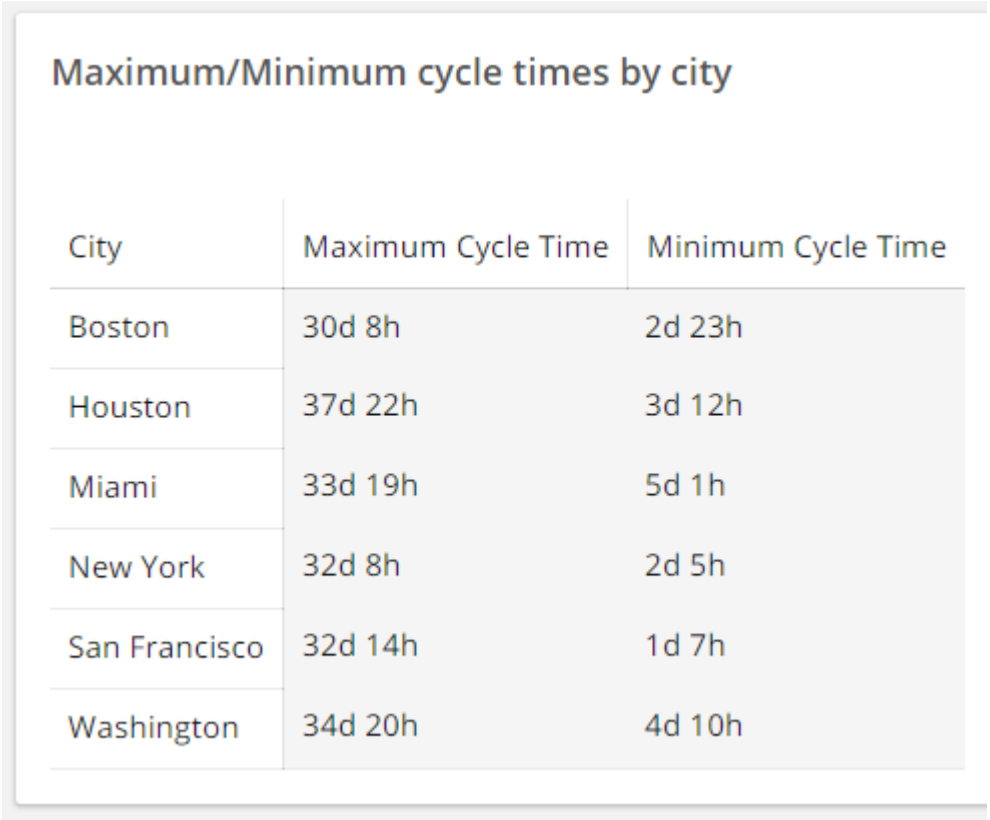
Learning success: Calculate the largest / smallest values.

Query instruction: According to example 1, you calculate the cycle times by city. From these values, you determine the smallest value(keyword: MIN) and the largest values(keyword: MAX). Rename the values with aliases (keyword: AS) to "Maximum Cycle Time" and "Minimum Cycle Time". Finally sort the result set by city(keywords: ORDER BY 3) in ascending order(keyword: ASC).

SIGNAL syntax:

```
SELECT
MAX(
(SELECT LAST(end_time))
-
(SELECT FIRST(end_time))) AS "Maximum Cycle Time",
MIN(
SELECT LAST(end_time))
-
(SELECT FIRST(end_time))) AS "Minimum Cycle Time",
"City"
FROM THIS_PROCESS
ORDER BY 3 ASC
```

Query result: The maximum /minimum cycle times, displayed in a SIGNAL table widget:



Maximum/Minimum cycle times by city

City	Maximum Cycle Time	Minimum Cycle Time
Boston	30d 8h	2d 23h
Houston	37d 22h	3d 12h
Miami	33d 19h	5d 1h
New York	32d 8h	2d 5h
San Francisco	32d 14h	1d 7h
Washington	34d 20h	4d 10h

2.5 Investigate events

Read about how events of the process are used in the SIGNAL tutorial.

You want to get an overview about the events of the process: How many cases have been closed / canceled? What is the drop-out rate? How many cases follow the standard process? How many cases are canceled although the T-shirt has been sent for printing?

Example 1: How many cases have been closed / canceled?

Learning success: Counting the total number of events.

You perform two subqueries for the last event names and sum up all cases for which the respective condition is fulfilled. The event names are event-based attributes, so you perform a subquery (keyword: (SELECT).

Select the last events of the cases (keyword: SELECT LAST + expression: event_name).

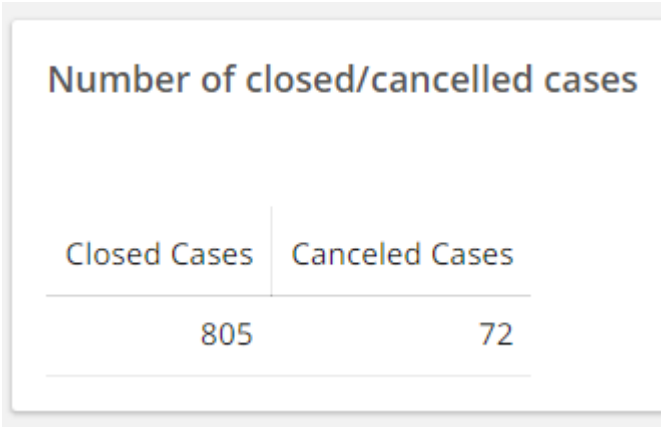
If (keyword: IF) the last event name is "Receive Delivery Confirmation" (keyword: IN + expression: ('Receive Delivery Confirmation')) count 1, otherwise 0 (keyword: 1,0)). Sum up (keyword: SUM) the value for all cases. Rename the value with an alias(keyword: AS) to "Closed Cases".

To count the value for the canceled cases, reverse the count condition (keyword: 0,1). Rename the column header with an alias (keyword: AS) to "Canceled Cases".

SIGNAL syntax:

```
SELECT
SUM
(IF
((SELECT LAST("event_name"))
IN('Receive Delivery Confirmation'),1,0)) AS "Closed Cases",
SUM
(IF
((SELECT LAST("event_name"))
IN('Receive Delivery Confirmation'),0,1)) AS "Canceled Cases"
FROM THIS_PROCESS
```

Query result: The total number of closed and canceled cases, displayed in a SIGNAL table widget:



Example 2: What is the drop-out rate?

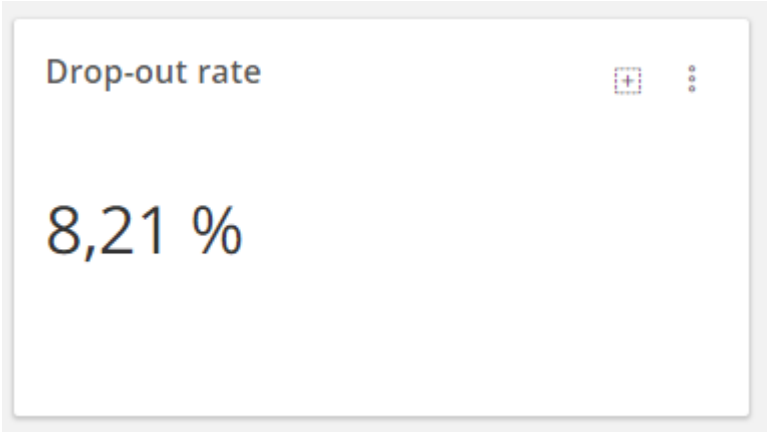
Learning success: Filter for not true conditions.

Query instruction: Select the last events of the cases (keyword: SELECT LAST + expression: event_name). Filter the result set(keyword: FILTER (WHERE) for event names other than (keyword: NOT) "Receive Delivery Confirmation" (keyword: IN+ expression: ('Receive Delivery Confirmation')) and calculate the percentage.

SIGNAL syntax:

```
SELECT
(COUNT(case_id) FILTER
(WHERE NOT(SELECT LAST(event_name)
IN('Receive Delivery Confirmation'))))
/
COUNT(case_id)
*100
FROM THIS_PROCESS
```

Query results: The drop-out rate, displayed in a *Value* widget:



Example 3: How many cases follow the standard process?

Learning success: Determine the process conformance.

Query instruction: This query is similar to example 2, but in this case, you do not only search for the first or last event but for a certain pattern of events.

You query the following pattern:

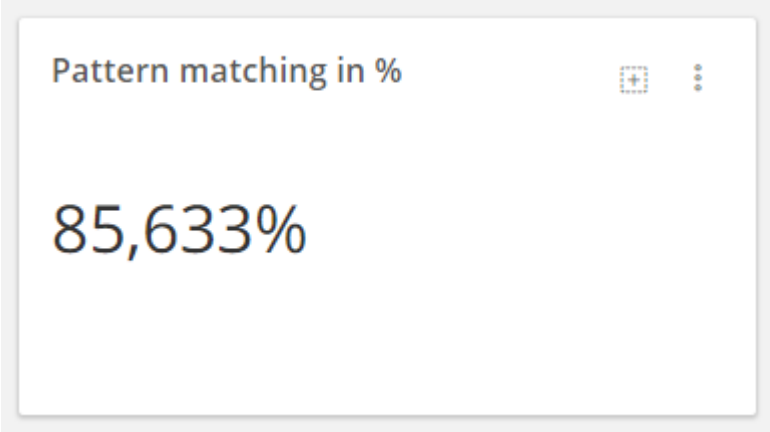
1. The starting event is "Receive Customer Order" (expression: ^'Receive Customer Order')
2. The next event (directly or indirectly following the start event) is "Receive Payment" (expression: ~> 'Receive Payment').
3. The next event (directly or indirectly following the preceding event) is either "Ship Goods Standard" or "Ship Goods Express"(expression: ~>('Ship Goods Standard | 'Ship Goods Express')) .
4. The final event (directly or indirectly following the preceding event) is "Receive Delivery Confirmation" (expression: 'Receive Delivery Confirmation'$).

For the conformance to this pattern, you calculate the percentage.

SIGNAL syntax:

```
SELECT
(COUNT(case_id) FILTER
(WHERE event_name MATCHES
(^ 'Receive Customer Order'
~>'Receive Payment'
~>('Ship Goods Standard'|'Ship Goods Express')
~> 'Receive Delivery Confirmation'$)))
/
COUNT(case_id)
*100
FROM THIS_PROCESS
```

Query result: The pattern matching, displayed in a *Value* widget.



Example 4: How many cases are canceled although T-shirt has been sent for printing?

Learning success: Determine the process conformance.

Query instruction: This query is similar to example 3, but for a different type of pattern: You want to determine how many orders have been canceled while the T-Shirt has already been sent for printing.

You query the following pattern:

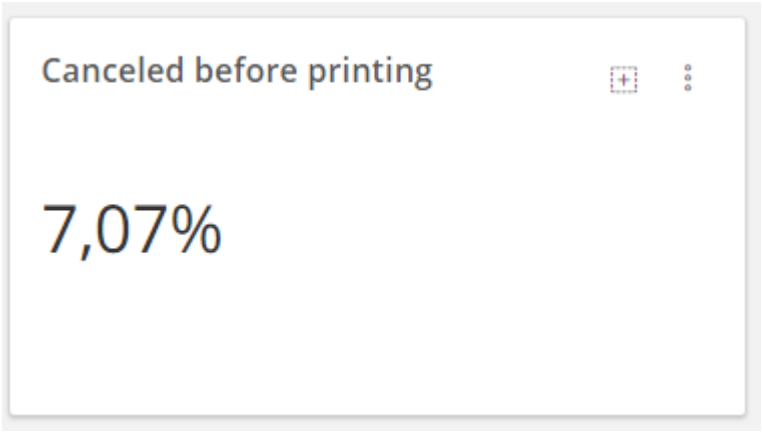
1. The starting event is "Receive Customer Order"(expression: ^'Receive Customer Order')
2. The next event (directly or indirectly following the start event) is "Receive Payment"(expression: ~> 'Receive Payment').
3. The next event (directly or indirectly following the preceding event) is "Send T-shirt to Printing"(expression: ~>'Ship T-shirt to Printing') .
4. The final event (directly or indirectly following the preceding event) is "Order Canceled"(expression: 'Order Canceled'$).

For the conformance to this pattern, you calculate the percentage.

SIGNAL syntax:

```
SELECT
(COUNT(case_id) FILTER
(WHERE event_name MATCHES
(^ 'Receive Customer Order'
~>'Receive Payment'
~>'Send T-shirt to Printing')
~>'Order Canceled'$)))
/
COUNT(case_id)
*100
FROM THIS_PROCESS
```

Query result: The pattern matching, displayed in a *Value* widget.


Canceled before printing
7,07%

3 SIGNAL Cookbook

What is the cookbook, who is it for and what will they get out of it?

Audience

The intended audience of the cookbook is process analysts who already have a basic familiarity with SIGNAL and knowledge of similar querying languages, like SQL. We recommend that the reader already knows the core keywords and functions available, and has already used the language to construct simple queries.

Purpose

SIGNAL, the SAP Signavio Analytics Language, is the process mining query language of SAP Signavio Process Intelligence. It's a powerful tool with numerous features that can be used and combined in many different ways. This cookbook aims to guide early-stage users who already have some basic familiarity with SIGNAL and help them to develop their abilities with the language.

Structure

The cookbook's approach is to guide the reader through a series of examples of process analysis using SIGNAL. You can think of each example as a use case or recipe demonstrating how to use the language to achieve some nontrivial goal.

They're organized into categories corresponding to typical process analysis activities, such as determining conformance, variants, or rework. This way, you can use the cookbook not only as a learning resource but also a reference guide, one that you can browse in order to learn about achieving specific, commonly occurring goals.

3.1 Cycle Time

Learn about different ways of calculating cycle times from your process data.

Cycle time refers to the recorded time between two events in a case. By itself, cycle time generally refers to the duration of a whole case, namely the time between that cases's first and last events. However, cycle time can also refer to the duration between two arbitary events of interest to the user.

3.1.1 Average Cycle Time

Calculate the average cycle time in a process.

Goal

Discover how long it takes on average to execute a process.

Solution

Calculate the complete cycle time for all cases and then find the average value from this collection.

Discussion

Every event has an associated `end_date` value. We can locate the earliest event in a case by using the `FIRST` function, which returns the first element in a collection.

```
FIRST(end_date)
```

Similarly, we can locate the latest event in a case using the `LAST` function.

To calculate the duration of a case, subtract the earliest `event_time` from the latest `event_time`. Keep in mind that `event_time` is an event-level attribute, so this subtraction is done in a subquery.

```
(SELECT LAST(end_time) - FIRST(end_time))
```

This expression would return the cycle times of all cases. To calculate the average of all these values, supply the expression as input to the `AVG` function.

```
AVG( (SELECT LAST(end_time) - FIRST(end_time)) )
```

We can now complete the query.

```
SELECT AVG( (SELECT LAST(end_time) - FIRST(end_time)) ) AS "Average Cycle Time"
FROM THIS_PROCESS
```

Example Output

Average Cycle Time
11d 7h

Related Information

AVG [page 96]
FIRST [page 109]
LAST [page 113]

3.1.2 Cycle Time Between All Events

Calculate the cycle times between all consecutive events in a case.

Goal

Discover how long each step of a case took. In other words, we want to know the cycle times between each pair of consecutive events.

Solution

For each case, order the events chronologically and compare the timestamp of each event to its predecessor (where it has one).

Discussion

To get the timestamp of the current row, we can select its `end_date` attribute.

As part of the solution, we want to compare each row and its immediate neighbor. For comparing a row and its predecessor, the `LAG` window function is available.

> → Tip
>
> Window functions allow you to group your data into partitions and carry out operations within each partition. You can use a window function to perform calculations on the set of rows in the same partition as the current row.

The LAG function accesses a column value from the row preceding the current row. Selecting `LAG(end_time)` returns the timestamp of the previous row. Bear in mind that the first row in each partition has no predecessor, in which case LAG returns `NULL`.

Additionally, we need to configure two things in the LAG function:

1. The partition. We're examining all events on a per-case basis, so we partition by `case_id`.
2. The sorting. We need the events in chronological order, so we sort by `end_time`.

This gives us the complete form:

```
LAG(end_time) OVER (PARTITION BY case_id ORDER BY end_time)
```

To get the cycle time between the two consecutive events, we can subtract the timestamp of the previous (earlier) row from that of the current one.

```
end_time - LAG(end_time) OVER (PARTITION BY case_id ORDER BY end_time)
```

Selecting this alone would give us only a list of durations. Let's also associate each duration with the names of the two enclosing events. We can get the event name of the current event by selecting `event_name`.

To get the name of a previous event, we can again use LAG. We partition and order the rows in the same way as before, but instead select the `event_name`.

```
LAG(event_name) OVER (PARTITION BY case_id ORDER BY end_time)
```

We have now defined the columns of our result set, which we can select along with `case_id`:

```
SELECT
    case_id,
    LAG(event_name) OVER (PARTITION BY case_id ORDER BY end_time) AS Predecessor,
    event_name AS Event,
    end_time - LAG(end_time) OVER (PARTITION BY case_id ORDER BY end_time) AS
cycle_time
```

All that remains is to add a FROM clause. Keep in mind that window functions can't work with nested data, so the process data table must be flattened.

```
SELECT
    case_id,
    LAG(event_name) OVER (PARTITION BY case_id ORDER BY end_time) AS Predecessor,
    event_name AS Event,
    end_time - LAG(end_time) OVER (PARTITION BY case_id ORDER BY end_time) AS
cycle_time
FROM FLATTEN(THIS_PROCESS)
```

Example Output

Given this example input data:

case_id	event_name	end_time
00001	Receive Customer Order	01/08/2020, 12:32

case_id	event_name	end_time
00001	Change Order Quantity	02/08/2020, 03:19
00001	Receive Payment	04/08/2020, 16:47
00001	Ship Goods Express	05/08/2020, 09:57
00001	Receive Delivery Confirmation	09/08/2020, 09:19
00002	Receive Customer Order	06/04/2020, 06:38
00002	Receive Payment	07/04/2020, 22:36
00002	Send items to Printing	08/04/2020, 17:26
00002	Order Canceled	09/04/2020, 23:43

our query returns the following output:

case_id	Predecessor	Event	cycle_time
00001	null	Receive Customer Order	null
00001	Receive Customer Order	Change Order Quantity	14h 47m
00001	Change Order Quantity	Receive Payment	2d 13h
00001	Receive Payment	Ship Goods Express	17h 9m
00001	Ship Goods Express	Receive Delivery Confirmation	3d 23h
00002	null	Receive Customer Order	null
00002	Receive Customer Order	Receive Payment	1d 15h
00002	Receive Payment	Send items to Printing	18h 49m
00002	Send items to Printing	Order Canceled	1d 6h

Related Information

LAG [page 272]
Window Functions [page 220]

3.1.3 Cycle Time Between Two Specific Events

Calculate the cycle time between two specific events in a case.

Goal

Discover the duration on average between an arbitrary pair of events in a process.

Solution

For each case, use two event-level subqueries to select the specific events of interest. Then, calculate the time difference of each pair before calculating the average of all differences.

Discussion

When writing this query, it's important to know how to identify exactly the events of interest. Let's say, for example, you want to know how long it normally takes for an order to be fully paid once it's received.

We can select the order receipt event using a query something like this:

```
(SELECT FIRST(end_time) FILTER (WHERE event_name = 'Receive Customer Order'))
```

The `FILTER` clause filters out all events without the name `'Receive Customer Order'`. In the example data set, every case has only one event with this name, so the filter actually returns one value per case. Nevertheless, this is an event-level subquery, so we must use an aggregation function to ensure just one `end_time` value is returned. We'll choose `FIRST`, although `LAST` would return the same value.

However, the same isn't true when selecting the payment event. A customer is allowed to pay in multiple installments, so there may be more than one event in each case with the name `'Receive Payment'`. Since we want to know the cycle time until the order is **fully** paid, we'll select the last payment event:

```
(SELECT LAST(end_time) FILTER (WHERE event_name = 'Receive Payment'))
```

To calculate the cycle time, subtract the earlier timestamp from the later one:

```
(SELECT LAST(end_time) FILTER (WHERE event_name = 'Receive Payment'))
- (SELECT FIRST(end_time) FILTER (WHERE event_name = 'Receive Customer Order'))
```

> → Tip
>
> The function `DURATION_BETWEEN` allows you to calculate the working time between the two timestamps, rather than calendar time as is done here.

Finally, use the `AVG` function to calculate the average cycle time between these two events:

```
SELECT AVG(
    (SELECT LAST(end_time) FILTER (WHERE event_name = 'Receive Payment'))
    - (SELECT FIRST(end_time) FILTER (WHERE event_name = 'Receive Customer
Order'))
    ) AS "Cycle time between receipt and payment"
FROM THIS_PROCESS
```

Example

Assume the following example data (`THIS_PROCESS`):

case_id	event_name	end_time
00001	• Receive Customer Order • Change Order Quantity • Receive Payment • Ship Goods Express • Receive Delivery Confirmation	• 23/03/2020, 18:44 • 24/03/2020, 19:24 • 26/03/2020, 04:49 • 26/03/2020, 15:32 • 27/12/2020, 20:37
00002	• Receive Customer Order • Receive Payment • Receive Payment • Ship Goods Express • Receive Delivery Confirmation	• 30/12/2020, 15:27 • 03/01/2021, 05:48 • 05/01/2021, 00:35 • 06/01/2021, 00:49 • 09/01/2021, 07:46
00003	• Receive Customer Order • Change Order Quantity • Receive Payment • Ship Goods Express • Receive Delivery Confirmation	• 02/01/2020, 09:02 • 03/01/2020, 04:43 • 04/01/2020, 14:00 • 05/01/2020, 16:30 • 07/01/2020, 04:27

Applying our query returns the following result:

Cycle time between receipt and payment
3d 8h

This being the average of the durations '2d 10h', '5d 9h', and '2d 4h'.

Related Information

Aggregate Functions [page 92]
Event-Level Subqueries [page 52]
FILTER Clause [page 20]

3.2 Variants

Learn about the different ways of analyzing variants in a process.

A process variant is a unique sequence of recorded events in a process.

3.2.1 Top N Variants

Show an arbitrary number of variants ranked according to their prevalence.

Goal

Determine which variants are the most prevalent within the process data. We'd like to choose how many variants to see and also query other data associated with each variant, in this case the average cycle time.

Solution

Find which distinct sequences of events exist in the process data, count them, and order the result numerically. Also, for each distinct sequence, calculate its average cycle time.

Discussion

As specified by the data model, each case contains a nested table of event data. One of the columns of this nested table is `event_name`. Selecting `event_name` in a query therefore returns a list of event names.

```
SELECT case_id, event_name AS "Variant"
FROM THIS_PROCESS
```

Examples of such variants might be as follows:

case_id	Variant
00001	• Receive Customer Order • Receive Payment • Ship Goods Standard • Receive Delivery Confirmation
00002	• Receive Customer Order • Receive Payment • Ship Goods Standard • Receive Delivery Confirmation
00003	• Receive Customer Order • Receive Payment • Ship Goods Express • Receive Delivery Confirmation

Although this example contains three event sequences, there are only two **distinct** sequences – notice how the first two cases ship goods standard, while the third case ships goods express. To show only the distinct variants, you can group them by their sequence of event names.

```
SELECT event_name AS "Variant"
FROM THIS_PROCESS
GROUP BY 1
```

To count the number of distinct variants in this grouped data set, apply `COUNT` to a case-level attribute.

```
SELECT
    event_name AS "Variant",
    COUNT(case_id)
FROM THIS_PROCESS
GROUP BY 1
```

To rank the variants starting with the most prevalent, sort this column in descending order of value. To choose only the most prevalent N values, limit the result set to N.

```
SELECT
    event_name AS "Variant",
    COUNT(case_id)
FROM THIS_PROCESS
GROUP BY 1
ORDER BY 2 DESC
LIMIT 5
```

Finally, we want to find the average cycle time per variant, which is explained in the Average Cycle Time recipe.

```
AVG( SELECT LAST(end_time) - FIRST(end_time) )
```

`AVG` is an aggregate function. When used with a `GROUP BY` clause, an aggregate function computes values across all rows of each group and returns a separate value for each group. Since we're grouping by sequence of event names, this calculates the average of each distinct variant.

We can now complete the query.

```
SELECT event_name AS "Variant",
    COUNT(case_id) as "Case Count",
    AVG( (SELECT LAST(end_time) - FIRST(end_time)) ) AS "Average Cycle Time"
FROM THIS_PROCESS
GROUP BY 1
ORDER BY 2 DESC
LIMIT 5
```

Example Output

Variant	Case Count	Average Cycle Time
Receive Payment	445	1w 1d
Ship Goods Standard		
Receive Delivery Confirmation		

Variant	Case Count	Average Cycle Time
Receive Customer Order Receive Payment Ship Goods Express Receive Delivery Confirmation	305	6d 1h
Receive Customer Order Receive Payment Send items to Printing Items Printed Receive items from Printing Ship Goods Express Receive Delivery Confirmation	278	1w 4d
Receive Customer Order Receive Payment Send items to Printing Items Printed Receive items from Printing Ship Goods Standard Receive Delivery Confirmation	188	1w 6d
Receive Customer Order Receive Payment Ship Goods Standard Ship Goods Standard Receive Delivery Confirmation Receive Delivery Confirmation	174	2w 3d

Related Information

Average Cycle Time [page 314]
AVG [page 96]
COUNT [page 104]
Data Model [page 4]
GROUP BY Clause [page 27]
LIMIT Clause [page 38]

ORDER BY Clause [page 30]

3.3 Rework and Repeated Events

Learn about different ways of analyzing rework in a process.

Rework within a case is identified by a number of repeated process steps exceeding a given threshold.

3.3.1 Cases With More Than N Repeated Events

Show an arbitrary number of cases containing rework, treating the threshold value of rework as a parameter.

Goal

Identify cases containing an amount rework that exceeds a given threshold.

Solution

In each case, count the number of occurrences of each event. Specify the minimum number of event occurrences to qualify them as rework and filter out cases whose value is insufficient.

Discussion

To count occurrences of events, we can use the `OCCURRENCE` function. This function calculates a running total of occurrences for a specified event-level attribute. For example, consider a case with the following list of event names:

case_id	event_name
00017	Receive Customer Order
	Change Order Quantity
	Change Order Quantity
	Change Order Quantity
	Receive Payment
	Ship Goods Standard
	Ship Goods Standard
	Receive Delivery Confirmation

The `OCCURRENCE` function would return the following values for `event_name`: [1, 1, 2, 3, 1, 1, 2, 1]. These values show that each event occurs between one and three times in that case.

Because our goal is to include only cases above the threshold value, let's use the `OCCURRENCE` function as part of a `WHERE` clause. Let's choose 3 as the threshold value. Since we're operating at event level, the clause must use an event-level subquery. However, we can't simply use the `OCCURRENCE` function like so:

```
SELECT case_id, "event_name", "end_time"
FROM THIS_PROCESS
WHERE (SELECT OCCURRENCE(event_name) AS occ) > 3
```

That's because a subquery must return an aggregated value instead of a collection of values. Since our goal is to identify cases meeting a minimum threshold value, let's take the maximum number of occurrences by using the `MAX` function. Doing this determines if at least one event meets the threshold.

```
SELECT case_id, "event_name", "end_time"
FROM THIS_PROCESS
WHERE (
    SELECT MAX(occ) FROM (
        SELECT OCCURRENCE(event_name) AS occ)
    AS sub) > 3
```

Finally, we can arbitrarily limit the number of cases returned. Let's take the first 10 cases from the process data.

```
SELECT case_id, "event_name", "end_time"
FROM THIS_PROCESS
WHERE (
    SELECT MAX(occ) FROM (
        SELECT OCCURRENCE(event_name) AS occ)
    AS sub) > 3
ORDER BY 1
LIMIT 10
```

Example Output

case_id	event_name	end_time
00018	• Receive Customer Order • Change Order Quantity • Change Order Quantity • Change Order Quantity • Change Order Quantity • Receive Payment • Ship Goods Standard • Ship Goods Standard • Receive Delivery Confirmation • Receive Delivery Confirmation	• 09/03/2020, 07:22 • 10/03/2020, 16:42 • 11/03/2020, 22:52 • 12/03/2020, 22:21 • 14/03/2020, 00:31 • 17/03/2020, 17:39 • 19/03/2020, 01:03 • 20/03/2020, 09:11 • 24/03/2020, 11:22 • 28/03/2020, 20:44
00098	• Receive Customer Order • Receive Payment • Receive Payment • Receive Payment • Receive Payment • Ship Goods Express • Receive Delivery Confirmation	• 27/06/2020, 12:22 • 01/07/2020, 05:08 • 03/07/2020, 05:26 • 05/07/2020, 06:35 • 08/07/2020, 13:24 • 09/07/2020, 09:25 • 12/07/2020, 09:02
00238	• Receive Customer Order • Receive Payment • Receive Payment • Receive Payment • Receive Payment • Send items to Printing • Order Canceled	• 12/04/2020, 04:11 • 13/04/2020, 19:56 • 15/04/2020, 01:37 • 16/04/2020, 04:18 • 18/04/2020, 07:35 • 20/04/2020, 01:02 • 21/04/2020, 08:00

(The subsequent seven cases are not depicted.)

Related Information

MAX [page 116]
OCCURRENCE [page 274]
WHERE Clause [page 18]

3.3.2 Counting Intermediate Events

Find out how many times an event occurred between two other events.

> **Note**
>
> This recipe uses a general subquery. It's therefore suitable only where general subqueries are supported, such as in the SIGNAL Editor.

Goal

For each case, determine how many times a specific event, E, occurs between occurrences of a pair of other events, A and B, where B chronologically follows A.

Solution

Find all cases featuring events A and B. Restrict the events under consideration to those that both succeed the first instance of A and precede the last instance of B. From this, count all instances of E.

As a demonstration, we'll write a query counting how many times an order quantity was changed between initial receipt of the order and final payment.

Discussion

We want to count a number of events in each case, so let's begin our query like this:

```
SELECT
    case_id,
    (SELECT COUNT(event_name)) AS "No. of Intermediate Events"
FROM ?
```

We don't want to count all events, only instances of a specific event (in this example, the event '`Change Order Quantity`'). Furthermore, we're only interested in those between the first instance of an event, A ('`Receive Customer Order`'), and the last instance of an event, B ('`Receive Payment`'). So, we filter the events of each case appropriately.

```
SELECT
    case_id,
    (SELECT COUNT(event_name)) AS "No. of Intermediate Events"
FROM ?
FILTER EVENTS WHERE event_name = 'Change Order Quantity'
    AND end_time >= "Time of First Event A"
    AND end_time <= "Time of Last of Event B"
```

"`Time of First Event A`" and "`Time of Last of Event B`" aren't columns in our process data, but we can define them using other columns. We also need to select some additional columns for this, so let's create a general subquery in which to do all this.

Putting aside the outer query for a moment, we begin the inner query by selecting the columns that the outer query references.

```
SELECT
    case_id,
    event_name,
    end_time,
    "Time of First Event A",
    "Time of Last of Event B"
FROM THIS_PROCESS
```

The first three columns exist in the process data. The last two will become aliases for columns defined using the process data. Respectively, we want:

- The timestamp of the earliest event whose name is '`Receive Customer Order`'.
- The timestamp of the latest event whose name is '`Receive Payment`'.

Remember that these columns are selected at event-level, so must be written as event-level subqueries:

```
SELECT
    case_id,
    event_name,
    end_time,
    (SELECT FIRST(end_time) FILTER (WHERE event_name = 'Receive Customer
Order')) AS "Time of First Event A",
    (SELECT LAST(end_time) FILTER (WHERE event_name = 'Receive Payment')) AS
"Time of Last of Event B"
FROM THIS_PROCESS
```

To ensure that we consider only cases where event A is followed at some point by event B, add a `WHERE` clause that uses an appropriate matching expression.

```
SELECT
    case_id,
    event_name,
    end_time,
    (SELECT FIRST(end_time) FILTER (WHERE event_name = 'Receive Customer
Order')) AS "Time of First Event A",
    (SELECT LAST(end_time) FILTER (WHERE event_name = 'Receive Payment')) AS
"Time of Last of Event B"
FROM THIS_PROCESS
WHERE event_name MATCHES ('Receive Customer Order' ~> 'Receive Payment')
```

Our inner query is now complete.

Returning to the outer query:

```
SELECT
    case_id,
    (SELECT COUNT(event_name)) AS "No. of Intermediate Events"
FROM ?
FILTER EVENTS WHERE event_name = 'Change Order Quantity'
    AND end_time >= "Time of First Event A"
    AND end_time <= "Time of Last of Event B"
```

We can now replace the '?' with the definition of the inner query, which gives us the overall query.

```
SELECT
    case_id,
    (SELECT COUNT(event_name)) as "No. of Intermediate Events",
    event_name
FROM (
    SELECT
        case_id,
        event_name,
        end_time,
        (SELECT FIRST(end_time) FILTER (WHERE event_name = 'Receive Customer
Order')) AS "Time of First Event A",
        (SELECT LAST(end_time) FILTER (WHERE event_name = 'Receive Payment')) AS
"Time of Last of Event B"
    FROM THIS_PROCESS
    WHERE event_name MATCHES ('Receive Customer Order' ~> 'Receive Payment')
) AS sub
FILTER EVENTS WHERE event_name = 'Change Order Quantity'
    AND end_time >= "Time of First Event A"
    AND end_time <= "Time of Last of Event B"
```

Example

Assuming the following process data (`THIS_PROCESS`):

case_id	event_name
00002	• Receive Customer Order • Receive Payment • Send items to Printing • Order Canceled
00003	• Receive Customer Order • Change Order Quantity • Receive Payment • Ship Goods Standard • Receive Delivery Confirmation
00004	• Receive Customer Order • Change Order Quantity • Receive Payment • Ship Goods Express • Receive Delivery Confirmation

case_id	event_name
00005	• Receive Customer Order • Change Order Quantity • Change Order Quantity • Receive Payment • Send items to Printing • Items Printed • Receive items from Printing • Ship Goods Express • Receive Delivery Confirmation

Our query produces the following output:

case_id	No. of Intermediate Events	event_name
00002	0	
00003	1	• Change Order Quantity
00004	1	• Change Order Quantity
00005	2	• Change Order Quantity • Change Order Quantity

Related Information

COUNT [page 104]
FILTER Clause [page 20]
FILTER EVENTS Clause [page 22]
FIRST [page 109]
General Subqueries [page 50]
LAST [page 113]

3.4 Compliance

See examples of analyzing compliance in a process.

Compliance measures to what extent a process exhibits specific patterns and behaviors.

3.4.1 Compliance Rate for a Subset of Cases

Show the compliance rate for only a subset of cases you're interested in. As a demonstration, we'll consider the subset as sales orders using standard invoicing.

Goal

See what rate of cases receive payment when standard invoicing is applied.

Solution

Calculate a count of cases exhibiting a specific pattern of events, then divide this by the total number of cases. In this instance, a case is counted if it uses standard invoicing and has received payment.

Discussion

To count cases, we can use the `COUNT` function. However, this counts all cases, so we need to exclude cases whose events don't follow the event pattern we're interested in.

We can exclude cases from a count by applying the `FILTER` clause to the `COUNT` function. This clause has the following syntax:

```
FILTER (WHERE <condition>)
```

To identify event patterns, SIGNAL provides matching expressions. These yield true or false depending on whether event-level attributes follow a specific sequence. In this example, we're interested in cases following the standard invoicing pattern: a customer order is received and payment follows later, either directly or indirectly. In our example data, this information is captured in the event name. We can express this as:

```
event_name MATCHES ('Receive Customer Order' ~> 'Receive Payment')
```

This becomes the condition in the `FILTER` clause. Putting this together with the `COUNT` function gives:

```
COUNT(1) FILTER (WHERE event_name MATCHES ('Receive Customer Order' ~> 'Receive
Payment'))
```

Since we'd like the compliance rate, we divide the filtered number of cases by the total number of cases and multiply by 100.

```
(
    COUNT(1) FILTER (WHERE event_name MATCHES ('Receive Customer Order'~>
'Receive Payment'))
    /
    COUNT(1)
) * 100
```

Finally, we form this as a query.

```
SELECT
(
    COUNT(1) FILTER (WHERE event_name MATCHES ('Receive Customer Order'~>
'Receive Payment'))
    /
    COUNT(1)
) * 100 AS "Standard Invoicing Compliance Rate"
FROM THIS_PROCESS
```

Related Information

COUNT [page 104]
FILTER Clause [page 20]
Matching Expressions [page 80]

3.5 Pattern Matching and Deviations

Learn how you can specify behaviour patterns, using them to find matches or deviations in process data.

A behaviour is an expression evaluating case or event level attributes and can be used in a pattern matching expression.

3.5.1 Incomplete Cases

Find the number of cases whose behavior shows them to be incomplete. As a demonstration, we'll consider sales invoices, which are considered incomplete if they remain unpaid and haven't been canceled..

Goal

Obtain a count of how many invoices remain unpaid and aren't in any other way resolved.

Solution

Find all cases where an order was received but no payment followed, ignoring canceled orders. Count the resultant cases.

Discussion

In our example process data, receipt of an order is given the event name '`Receive Customer Order`'. Using pattern matching, we can identify all received orders by matching against this event name

```
event_name MATCHES ('Receive Customer Order')
```

We need to refine the matching so that, from these cases, only those which are either unpaid or not canceled are matched. In the case of unpaid orders, receipt events aren't followed by payment events, either directly or indirectly.

```
NOT event_name MATCHES ('Receive Customer Order' ~> 'Receive Payment')
```

Similarly, in the case of non-canceled orders, receipt events aren't followed by cancellation events, either directly or indirectly.

```
NOT event_name MATCHES ('Receive Customer Order' ~> 'Cancel Invoice')
```

For an order to be considered incomplete, the sequence of events in each case must match all three of these conditions. Therefore, we connect them using the logical connector `AND`.

```
event_name MATCHES ('Receive Customer Order')
AND
NOT event_name MATCHES ('Receive Customer Order' ~> 'Receive Payment')
AND
NOT event_name MATCHES ('Receive Customer Order' ~> 'Cancel Invoice')
```

Finally, this matching expression can become part of a `WHERE` clause. Because we're counting matches, we can simply count the number of rows that match that expression.

```
SELECT COUNT(1) AS "Case Count"
FROM THIS_PROCESS
WHERE
    event_name MATCHES ('Receive Customer Order')
    AND
    NOT event_name MATCHES ('Receive Customer Order' ~> 'Receive Payment')
    AND
    NOT event_name MATCHES ('Receive Customer Order' ~> 'Cancel Invoice')
```

Related Information

COUNT [page 104]
Logical Expressions [page 78]
Matching Expressions [page 80]

3.6 Time Series

See examples of analyzing time series process data.

A time series arranges process data into chronologically ordered cases and events.

3.6.1 Filtered Events With Gaps Filled

Filter events to find specific cases of interest and order them chronologically without leaving any gaps in the time series.

Goal

Identify cases containing a specific pattern of events and put them into a time series, summarizing them at a particular level of precision. To demonstrate this, we'll write a query that selects only sales orders paid for in multiple installments and counts them on a per day basis.

Solution

Select the timestamps of the cases in the process data, filtering for cases whose event patterns contain more than one payment event. Summarize them per day and fill any gaps in the time series, so that days without such cases are also included in the result set.

Discussion

First, select the timestamp. Take the date of the case's first event, in other words when the order was created. Since we're selecting an event, use an event-level subquery.

```
(SELECT FIRST(end_time))
```

The timestamps are to be summarized per day, so truncate the timestamp to that level of precision.

```
DATE_TRUNC(
    'day',
    (SELECT FIRST(end_time))
) AS "Date"
```

To count cases per day, use the `COUNT` function to count `case_id` in the result set. (Recall that when the `SELECT` clause includes both aggregate and non-aggregate expressions, the result is automatically grouped by the non-aggregate expressions, in this case the timestamp.)

Also, order the result set chronologically.

```
SELECT
    DATE_TRUNC(
        'day',
        (SELECT FIRST(end_time))
    ) AS "Date",
    COUNT(case_id) AS "# Cases"
FROM THIS_PROCESS
ORDER BY 1
```

At this stage, the query counts all cases and returns the following result on the example data:

Date	# Cases
01/01/2020, 00:00	4
02/01/2020, 00:00	10
03/01/2020, 00:00	8
04/01/2020, 00:00	9
05/01/2020, 00:00	15
06/01/2020, 00:00	15
07/01/2020, 00:00	7

To include only the cases where multiple payments were made, filter using a matching expression. Include only cases where a payment event was followed at some point by another payment event.

```
SELECT
    DATE_TRUNC(
        'day',
        (SELECT FIRST(end_time))
    ) AS "Date",
    COUNT(case_id) AS "# Cases"
FROM THIS_PROCESS
WHERE event_name MATCHES ('Receive Payment' ~> 'Receive Payment')
ORDER BY 1
```

Now that the query filters out certain cases, executing it on the example data leaves gaps in the time series:

Date	# Cases
01/01/2020, 00:00	1
02/01/2020, 00:00	2
04/01/2020, 00:00	2
05/01/2020, 00:00	4
06/01/2020, 00:00	4
07/01/2020, 00:00	1
09/01/2020, 00:00	2

To remove the gaps, use the `FILL` clause to insert rows where dates are missing. The query selects two columns, so the `FILL` clause must provide one specification for each:

1. `TIMESERIES('day')`: The first column of an inserted row is filled with the day missing at that point in the time series.
2. `NULL`: The second column of an inserted row is filled with `NULL`.

Adding the `FILL` clause puts `NULL` into the gaps:

```
SELECT
    DATE_TRUNC(
        'day',
        (SELECT FIRST(end_time))
    ) AS "Date",
    COUNT(case_id) AS "# Cases"
FROM THIS_PROCESS
WHERE event_name MATCHES ('Receive Payment' ~> 'Receive Payment')
ORDER BY 1
FILL TIMESERIES ('day'), NULL
```

Date	# Cases
01/01/2020, 00:00	1
02/01/2020, 00:00	2
03/01/2020, 00:00	null
04/01/2020, 00:00	2
05/01/2020, 00:00	4
06/01/2020, 00:00	4
07/01/2020, 00:00	1
08/01/2020, 00:00	null
09/01/2020, 00:00	2

Finally, to fill the gaps with zeroes instead – so, for example, the result can be plotted suitably on a chart – use the `COALESCE` function to replace any `NULL` values in the result set.

```
SELECT "Date", COALESCE("# Cases", 0) AS "# Cases"
FROM (
    SELECT
        DATE_TRUNC(
            'day',
            (SELECT FIRST(end_time))
        ) AS "Date",
        COUNT(case_id) AS "# Cases"
    FROM THIS_PROCESS
    WHERE event_name MATCHES ('Receive Payment' ~> 'Receive Payment')
    ORDER BY 1
    FILL TIMESERIES ('day'), NULL
) AS sub
```

Example Output

Date	# Cases
01/01/2020, 00:00	1
02/01/2020, 00:00	2
03/01/2020, 00:00	0
04/01/2020, 00:00	2
05/01/2020, 00:00	4
06/01/2020, 00:00	4
07/01/2020, 00:00	1
08/01/2020, 00:00	0

Related Information

COALESCE [page 146]
DATE_TRUNC [page 174]
FILL Clause [page 32]
Matching Expressions [page 80]

3.6.2 Active Cases Within a Period

Obtain a summary of active cases over time.

Goal

See a month-by-month overview of how many cases were active in the respective time period. Also, we'd like running totals of the number of started and finished cases.

Solution

The solution has three parts:

1. On a monthly basis, find which cases were opened and which were closed.
2. Count the found cases, filling any gaps in time series.
3. Calculate from these figures how many cases were active.

Discussion

Identifying Opened and Closed Cases

First, identify for each case the dates when the order was opened and when it was closed.

```
SELECT (SELECT FIRST(end_time)) AS "Month"
FROM THIS_PROCESS
SELECT (SELECT LAST(end_time)) AS "Month"
FROM THIS_PROCESS
```

We're interested in a month-by-month overview, so let's truncate the precision of the dates to the level of month.

```
SELECT DATE_TRUNC('month', (SELECT FIRST(end_time)) AS "Month"
FROM THIS_PROCESS
SELECT DATE_TRUNC('month', (SELECT LAST(end_time)) AS "Month"
FROM THIS_PROCESS
```

Next, we label each selected row with a number. This label, `started`, indicates if the row records an order being opened (1) or closed (0).

```
SELECT DATE_TRUNC('month', (SELECT FIRST(end_time))) AS "Month",
    1 AS started
FROM THIS_PROCESS
SELECT DATE_TRUNC('month', (SELECT LAST(end_time))) AS "Month",
    0 AS started
FROM THIS_PROCESS
```

Finally, unite these two queries using the `UNION ALL` clause so that they return one combined result set.

```
SELECT DATE_TRUNC('month', (SELECT FIRST(end_time))) AS "Month",
    1 AS started
FROM THIS_PROCESS
UNION ALL
SELECT DATE_TRUNC('month', (SELECT LAST(end_time))) AS "Month",
    0 AS started
FROM THIS_PROCESS
```

This query serves as a subquery, providing a data source for an outer query that we construct in the following section.

Counting the Cases per Month

Let's start by selecting from that result set the month, which we also use to group the data.

```
SELECT "Month"
FROM (
    -- The subquery from the previous section goes here.
) AS sub1
GROUP BY 1
```

Next, the labels we created become useful. We use them to count how many cases were opened (`started = 1`) and how many were closed (`started = 0`) by filtering on those labels before counting.

```
SELECT "Month",
    COUNT(1) FILTER (WHERE started = 1) AS count_started,
    COUNT(1) FILTER (WHERE started = 0) AS count_finished
FROM (
    -- The subquery from the previous section goes here.
```

```
) AS sub1
GROUP BY 1
```

If a time period happens to contain no rows, it appears as a gap in the time series. It's a good idea therefore to use a `FILL` clause, which inserts missing periods into any gaps. In this case, we're querying at the level of months, so we'll use a specification of `FILL TIMESTAMP('month')` to ensure the regularity of the time series.

```
SELECT "Month",
    COUNT(1) FILTER (WHERE started = 1) AS count_started,
    COUNT(1) FILTER (WHERE started = 0) AS count_finished
FROM (
    -- The subquery from the previous section goes here.
) AS sub1
GROUP BY 1
FILL TIMESERIES('month')
```

At this point, running the query produces an output something like this:

Month	count_started	count_finished
01/01/2020, 00:00	267	179
01/02/2020, 00:00	282	283
01/03/2020, 00:00	264	249
01/04/2020, 00:00	291	283
01/05/2020, 00:00	274	285

Once again, this query will become a subquery, acting as a data source for an outer query that we'll construct in the following section.

Calculating Active Cases

Let's begin our new outer query by selecting the month and the two count columns from the previous query. Doing this gives us two of the three columns stated in our original goal.

```
SELECT "Month",
    count_started AS "Started, Current & Previous Months",
    count_finished AS "Completed, Current & Previous Months",
FROM (
        -- Subquery from the previous section goes here.
    ) AS sub2
) AS sub3
```

The third column should display the number of active cases for a particular month in time. For any given month, that number is calculated by taking the number of cases started up to that month and subtracting the cases finished **up to the previous month**. Whenever you need to consider the values from the row immediately before, you can use the `LAG` function.

The following expression performs the desired calculation:

```
count_started - LAG(count_finished) OVER (ORDER BY "Month")
```

Be mindful that if no previous row exists, `LAG` returns `NULL`. So, we use the `COALESCE` function to ensure that a numeric value is returned in such a case.

```
count_started - COALESCE(LAG(count_finished) OVER (ORDER BY "Month"), 0)
```

We can add this to our `SELECT` clause as the "`Active Cases`" column.

```
SELECT "Month",
    count_started AS "Started, Current & Previous Months",
    count_finished AS "Completed, Current & Previous Months",
    count_started - COALESCE(LAG(count_finished) OVER (ORDER BY "Month"), 0) AS
"Active Cases"
FROM (
    -- Subquery from previous section goes here.
    ) AS sub2
) AS sub3
```

Putting It All Together

Our final query therefore looks like this:

```
SELECT "Month",
    count_started AS "Started, Current & Previous Months",
    count_finished AS "Completed, Current & Previous Months",
    count_started - COALESCE(LAG(count_finished) OVER (ORDER BY "Month"), 0) AS
"Active Cases"
FROM (
    SELECT "Month",
        SUM(count_started) OVER (ORDER BY "Month") AS count_started,
        SUM(count_finished) OVER (ORDER BY "Month") AS count_finished
    FROM (
        SELECT "Month",
            COUNT(1) FILTER (WHERE started = 1) AS count_started,
            COUNT(1) FILTER (WHERE started = 0) AS count_finished
        FROM (
            SELECT DATE_TRUNC('month', (SELECT FIRST(end_time))) AS "Month",
                1 AS started
            FROM THIS_PROCESS
            UNION ALL
            SELECT DATE_TRUNC('month', (SELECT LAST(end_time))) AS "Month",
                0 AS started
            FROM THIS_PROCESS
        ) AS sub1
        GROUP BY 1
        FILL TIMESERIES('month')
    ) AS sub2
) AS sub3
```

Example Output

Month	Started, Current & Previous Months	Completed, Current & Previous Months	Active Cases
01/01/2020, 00:00	268	179	268
01/02/2020, 00:00	550	462	371
01/03/2020, 00:00	814	711	352
01/04/2020, 00:00	1105	994	394
01/05/2020, 00:00	1379	1279	385

Related Information

COALESCE [page 146]
COUNT [page 104]
DATE_TRUNC [page 174]
FILL Clause [page 32]
FILTER Clause [page 20]
FIRST [page 109]
LAG [page 272]
LAST [page 113]
UNION ALL Clause [page 47]

Important Disclaimers and Legal Information

Hyperlinks

Some links are classified by an icon and/or a mouseover text. These links provide additional information.
About the icons:

- Links with the icon  : You are entering a Web site that is not hosted by SAP. By using such links, you agree (unless expressly stated otherwise in your agreements with SAP) to this:
  - The content of the linked-to site is not SAP documentation. You may not infer any product claims against SAP based on this information.
  - SAP does not agree or disagree with the content on the linked-to site, nor does SAP warrant the availability and correctness. SAP shall not be liable for any damages caused by the use of such content unless damages have been caused by SAP's gross negligence or willful misconduct.
- Links with the icon  : You are leaving the documentation for that particular SAP product or service and are entering an SAP-hosted Web site. By using such links, you agree that (unless expressly stated otherwise in your agreements with SAP) you may not infer any product claims against SAP based on this information.

Videos Hosted on External Platforms

Some videos may point to third-party video hosting platforms. SAP cannot guarantee the future availability of videos stored on these platforms. Furthermore, any advertisements or other content hosted on these platforms (for example, suggested videos or by navigating to other videos hosted on the same site), are not within the control or responsibility of SAP.

Beta and Other Experimental Features

Experimental features are not part of the officially delivered scope that SAP guarantees for future releases. This means that experimental features may be changed by SAP at any time for any reason without notice. Experimental features are not for productive use. You may not demonstrate, test, examine, evaluate or otherwise use the experimental features in a live operating environment or with data that has not been sufficiently backed up.
The purpose of experimental features is to get feedback early on, allowing customers and partners to influence the future product accordingly. By providing your feedback (e.g. in the SAP Community), you accept that intellectual property rights of the contributions or derivative works shall remain the exclusive property of SAP.

Example Code

Any software coding and/or code snippets are examples. They are not for productive use. The example code is only intended to better explain and visualize the syntax and phrasing rules. SAP does not warrant the correctness and completeness of the example code. SAP shall not be liable for errors or damages caused by the use of example code unless damages have been caused by SAP's gross negligence or willful misconduct.

Bias-Free Language

SAP supports a culture of diversity and inclusion. Whenever possible, we use unbiased language in our documentation to refer to people of all cultures, ethnicities, genders, and abilities.

www.sap.com/contactsap

THE BEST RUN SAP