SQL for Analytics Developer Guide

Salesforce, Summer '24

Last updated: May 3, 2024

CONTENTS

INTRODUCING SQL FOR CRM ANALYTICS

SQL for CRM Analytics is a query language that lets you execute queries against your CRM Analytics datasets. SQL (Structured Query Language) is typically used for working with data stored in relational databases—you might already be familiar with variants like MySQL and PostGreSQL. CRM Analytics uses the ANSI SQL interface to access CRM Analytics's fast in-memory data store. It has native support of Salesforce features used in CRM Analytics, including sharing inheritance and custom fiscal calendars.

SQL for CRM Analytics supports these standard SQL features.

- `SELECT` Clause
- `FROM` Clause
- `WHERE` Clause
- `GROUP BY` Clause
- `GROUP BY ROLLUP`
- `GROUPING()`
- `HAVING` Clause
- `ORDER BY` Clause
- `LIKE` Clause
- `LIMIT` Clause
- `FETCH` Clause
- `OFFSET` Clause
- `CASE` statements
- `COALESCE()`
- `NULLIF()`
- `UNION` Operator
- Aggregate, windowing, math, and string functions
- Subqueries

Note: These features don't support the full range of cases covered by ANSI SQL. Refer to each statement in the SQL for CRM Analytics reference for more information.

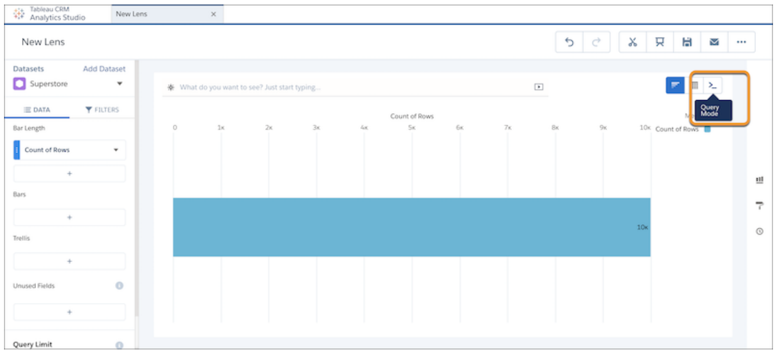
SEE ALSO:

Add Row-Level Security by Inheriting Sharing Rules

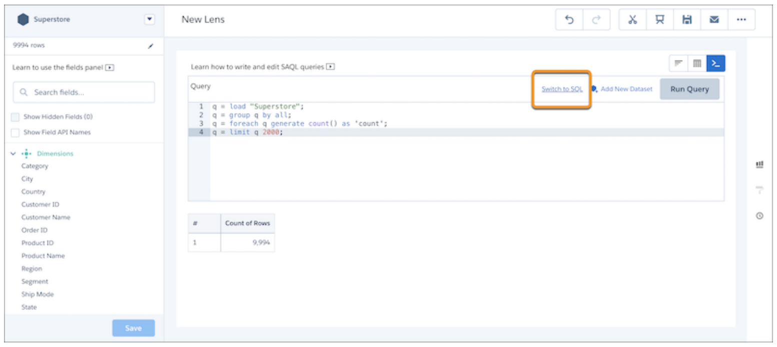
QUICK START: RUN YOUR FIRST QUERY

Let’s write a basic query for SQL for CRM Analytics in Analytics Studio.

1. In CRM Analytics Studio, open a Lens on a dataset.
2. Click the **Query Mode** button.



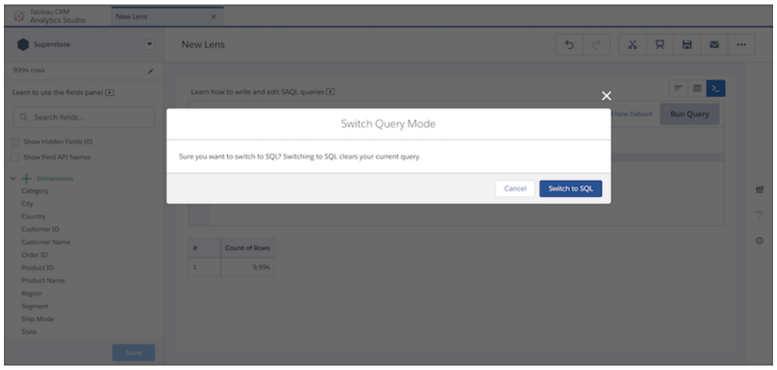
3. SAQL is the default query language. To use SQL, click **Switch to SQL**.



```
q = load "Superstore";
q = group q by all;
q = foreach q generate count() as 'count';
q = limit q 2000;
```

#	Count of Rows
1	9,994

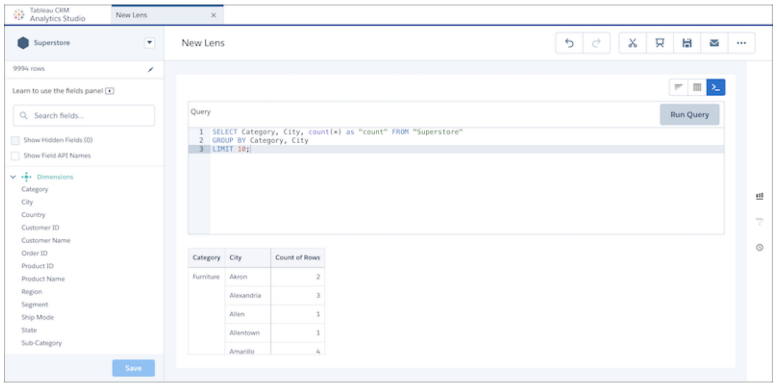
4. Switching to SQL provides a warning that switching clears your current query. If you don't want to lose the SAQL query, copy it. Then click **Switch to SQL**.



Sure you want to switch to SQL? Switching to SQL clears your current query.

Note: To return to SAQL query language, click the **Chart Mode** button, then click the **Query Mode** button. If you return to SAQL, you lose any SQL query changes that you made.

5. The default query for SQL is the row count in your dataset. Let's update the query to count the number of times a category occurs for each city. To run the query, click the **Run Query** button. Because we limit the results to 10 rows, the results display the first 10 cities in the furniture category.



```
SELECT Category, City, count(*) as "count" FROM "Superstore"
GROUP BY Category, City
LIMIT 10;
```

Category	City	Count of Rows
Furniture	Akron	2
	Alexandria	3
	Allen	1
	Allentown	1
	Amarillo	4

```
SELECT Category, City, count(*) as "count" FROM "Superstore"
GROUP BY Category, City
LIMIT 10;
```

Category	City	count
Furniture	Akron	2
	Alexandria	3
	Allen	1
	Allentown	1
	Amarillo	4
	Anaheim	8
	Andover	1
	Apopka	1
	Apple Valley	2
	Arlington	13

SEE ALSO:

Find Analytics and Insights

SQL VS SAQL

SQL for CRM Analytics and Salesforce Analytics Query Language (SAQL) are both query languages intended for use with CRM Analytics. Here's a guide to their different behaviors and limitations.

You can only write SQL queries against datasets.

SQL for CRM Analytics doesn't offer support for these features available in SAQL.

- Random sampling
- Cogrouping
- SAQL functions `string_to_number()` and `toString()`. Type casting is unavailable in SQL for CRM Analytics.
- Relative date filtering, for example, `Closed_Date in ["2 years ago".."1 day ago"]`
- Date functions such as `day_in_week()`, `date_diff()`, `now()`, `date_to_epoch()`, and `date_to_string()` tc

SQL and SAQL have the following differences in behavior.

- In a grouped query, SQL returns a group for null values in a grouped query by default.
- In SQL, counting all records in an empty dataset returns 0. In SAQL, it returns an empty response.
- In SQL, the `GROUPING()` function can take multiple arguments. In SAQL, it takes one argument only.
- SAQL's `DateOnly` type corresponds with SQL for CRM Analytics's `Date` type.
- SAQL's `DateTime` and `Date` types correspond with SQL's `Timestamp` type.
- In SQL, the `Date` and `Timestamp` types are treated as dates. In SAQL, when timezone is enabled, the `DateOnly` and `DateTime` types are treated as dates. When timezone isn't enabled, the `DateOnly` and `DateTime` types are treated as strings.
- When columns in a `SELECT` statement have the same data types, but different aliases, `UNION ALL` in SQL returns the results in a single column. In SAQL, `union` returns all of the columns.
- If your values query (one without a grouping) includes aggregated and non-aggregated fields, SQL throws an error. SAQL handles such queries differently by taking the value of individual rows.

  In this example, `City` is a non-aggregated field. `Count` and `sum_Profit` are aggregated fields.

  This SQL query throws an error.

  ```
  SELECT City AS "City", COUNT(*) AS "Count", sum(Profit) AS "sum_Profit"
  FROM "Superstore"
  LIMIT 100;
  ```

  The corresponding SAQL query yields the sum of `Profit` for each `City`.

  ```
  q = load "Superstore";
  q = foreach q generate 'City' as 'City', count() as 'count', sum('Profit') as
  'sum_Profit';
  ```

SEE ALSO:

Analytics SAQL Developer Guide

SQL STATEMENTS

Use a SQL statement to access and perform operations on data from one or more tables in your database.

A statement is made up of a series of clauses. It must include the keywords `SELECT` and `FROM`. `SELECT` indicates which columns to retrieve, and `FROM` indicates which tables the columns are in. A statement follows this syntax:

```
SELECT column_name FROM table_name
```

Note: Einstein Analytics SQL does not support the use of `*` with the `SELECT` clause. You must specify which columns to retrieve.

`SELECT` Clause

The `SELECT` clause retrieves columns from a table.

`SELECT` query syntax takes the form of a required `SELECT` statement followed by one or more optional clauses, such as `WHERE`, `GROUP BY`, and `ORDER BY`.

For a simple column projection, SQL outputs the results with the name of the column provided. If you include additional expressions and functions, you must use an alias to name the column output. Include an alias by following the structure ***`expression AliasName`*** or ***`expression as AliasName`***. The "as" is not required.

Note: Unlike other versions of SQL, the CRM Analytics SQL server throws an error if you don't include an alias. It does not automatically provide an alias name.

Let's look at an example.

```
SELECT City, COUNT(*) as StoreCount
FROM "Superstore"
GROUP BY City;
```

Here are the first ten results.

City	StoreCount
Aberdeen	1
Abilene	1
Akron	21
Albuquerque	14
Alexandria	16
Allen	4
Allentown	7

City	StoreCount
Altoona	2
Amarillo	10
Anaheim	27

This query returns a table with two columns, `City` and the alias `StoreCount`. `StoreCount` is the column name given to the expression, `COUNT(*)`, which means count the number of rows for each city in the Superstore table. The `GROUP BY` clause groups all of the store counts by the `City` column.

`SELECT` supports the following operators.

Type	Operator Name	Supported Types
Arithmetic Operators	+	Numeric, String
	-	Numeric
	*	Numeric
	/	Numeric
	%	Numeric
Comparison Operators	!=, =	Numeric, String, Boolean
	<, <=, >, >=	Numeric, String
	`IN`, `NOT IN`	Numeric, String
Logical Operators	`AND`, `OR`, `NOT`	Boolean
Case Operators	`CASE ... END`	Numeric, String, Boolean

`SELECT` supports arithmetic operators between measure fields and using "+" between string fields.

FROM Clause
The `FROM` clause defines which table or table function to query. CRM Analytics SQL supports using `FROM` with a single table only. You can't select from multiple tables.

WHERE Clause
By default, a SQL query retrieves every row in your dataset. Use the optional `WHERE` clause to restrict your query results to a conditional expression.

Boolean Expressions in SELECT Clause
A boolean expression is a logical statement that returns either `true` or `false`.

GROUP BY Clause
The `GROUP BY` clause organizes the rows returned from a `SELECT` statement into groups. Within each group, you can apply an aggregate function, like `count()` or `sum()` to get the number of items or sum, respectively.

GROUP BY ROLLUP

`ROLLUP` is a sub-clause of `GROUP BY` that creates and displays aggregations of column data. The results output of `ROLLUP` is based on column order in your query.

GROUPING()

Use the `GROUPING()` clause with `ROLLUP` to determine whether null values are the result of a `ROLLUP` operation or part of your dataset. `GROUPING()` returns 1 if the null value is a subtotal generated by a rollup. It returns 0 otherwise.

HAVING Clause

Use the `HAVING` clause to filter grouped results from grouped columns, aggregate functions, or grouping functions.

ORDER BY Clause

`SELECT` returns rows in an unspecified order by default. To sort returned rows in ascending or descending order, use the `ORDER BY` clause.

LIKE Clause

Use `LIKE` to match single characters and patterns found anywhere in a string—beginning, ending, or somewhere in between.

BETWEEN Operator

Use `BETWEEN` to check whether values fall within a given range. `BETWEEN` accepts numeric, string, and date data types, and can be used with aggregate, window, and math functions.

LIMIT Clause

The `LIMIT` clause specifies the maximum number of rows to return. If no `LIMIT` clause is specified, then SQL returns all rows.

FETCH Clause

The `FETCH` clause specifies the number of rows to return. If no `FETCH` clause is specified, then SQL returns all rows.

OFFSET Clause

By default, a SQL query retrieves every row in your dataset. Use the optional `WHERE` clause to restrict your query results to a conditional expression.

CASE Statements

Use case statements to express if/then logic. A case statement always has a pair of `WHERE` and `THEN` statements. CRM Analytics supports the simple and searched forms of case expressions in a `SELECT` statement. Case statements have two formats: simple and searched.

COALESCE()

You can use the `coalesce()` function as shorthand for case statements. The `coalesce()` function replaces null values in your dataset with another value. The function takes a series of arguments and returns the first value that is not null.

NULLIF()

Use the `nullif()` function as shorthand for a searched case statement where two equal expressions return null.

UNION Operator

Use the `UNION` operator to combine the results of two or more `SELECT` statements. The joined statements must have the same number of columns and the same data types for corresponding columns.

Subquery

A subquery is a query that is nested inside a `SELECT` statement. Nesting queries allows you to perform multi-step operations.

Window Functions

A window function lets you perform calculations on a selection—or "window"—of rows that are related to the current row. Unlike a regular aggregate function, such as `avg()`, `sum()`, or `count()`, the row output of a window function isn't grouped into a single row.

FROM Clause

The FROM clause defines which table or table function to query. CRM Analytics SQL supports using FROM with a single table only. You can't select from multiple tables.

CRM Analytics SQL supports the following table functions.

FILL
Use the FILL() table function to fill in any gaps in date fields. By specifying the date fields to check, FILL() creates rows that contain the missing month, day, week, quarter, or year and null data. Use it with TIMESERIES() to forecast future results when there are gaps in input data.

TIMESERIES
Use the TIMESERIES() table function to predict future values based on existing ones tracked over time. Optionally choose a prediction model, confidence interval, and seasonality among other parameters.

FILL

Use the FILL() table function to fill in any gaps in date fields. By specifying the date fields to check, FILL() creates rows that contain the missing month, day, week, quarter, or year and null data. Use it with TIMESERIES() to forecast future results when there are gaps in input data.

FILL() takes the following syntax.

```
SELECT <PROJECTION_LIST>|* FROM FILL(
    INPUT=>(SELECT STATEMENT),
    DATE_COLS=>ARRAY[<DATE_FIELDS>, 'DATE_COLUMN_TYPE'],
    [PARTITION=>'FIELD_NAME']
)
```

Name	Description
INPUT	Required. A SELECT statement that includes date information and is the input to the FILL() function.
DATE_COLS	Required. ***DATE_FIELDS***—The array of date fields in which to check for gaps. The ***DATE_COLUMN_TYPE*** string accepts these values. • '***YEAR_FIELD***', '***MONTH_FIELD***', 'Y-M' • '***YEAR_FIELD***', '***QUARTER_FIELD***', 'Y-Q' • '***YEAR_FIELD***', 'Y' • '***YEAR_FIELD***', '***WEEK_FIELD***', 'Y-W' • '***YEAR_FIELD***', '***MONTH_FIELD***', '***DAY_FIELD***', 'Y-M-D'
PARTITION	Optional. A field used to split query results into smaller partitions. The FILL() function resets when the field value changes. After

Name	Description
	each group of rows is completed for a given partition, `FILL()` runs on the next partition.

Example

This example uses `FILL()` to add missing quarter and year values to tourist data.

```
SELECT "year", "quarter", tourists FROM FILL(
    INPUT=>(SELECT EXTRACT(YEAR FROM "date") as "year", EXTRACT(QUARTER FROM "date") as
"quarter",
tourists FROM "TouristsData"),
    DATE_COLS=>ARRAY['year', 'quarter', 'Y-Q']);
```

The input `SELECT` statement returns the year, quarter, and number of tourists for each quarter. Based on the results from the first three years represented in the dataset, the only date data available is for the first quarter.

Tip: Another way to project all columns from the `INPUT` query is by using `SELECT * FROM FILL(...)`.

These are the results from executing only the `INPUT SELECT` query.

year	quarter	tourists
2001	1	4127
2002	1	4173
2003	1	4621

`FILL()` specifies in the `DATE_COLS` array to check for gaps in year and quarter fields in the input data. To have a complete dataset of years and quarters, `FILL()` adds the 2nd, 3rd, and 4th quarters for each year and a null value for the number of tourists.

year	quarter	tourists
2001	1	4127
2001	2	-
2001	3	-
2001	4	-
2002	1	4173
2002	2	-
2002	3	-
2002	4	-
2003	1	4621
2003	2	-

year	quarter	tourists
2003	3	-
2003	4	-

TIMESERIES

Use the `TIMESERIES()` table function to predict future values based on existing ones tracked over time. Optionally choose a prediction model, confidence interval, and seasonality among other parameters.

`TIMESERIES()` takes the following syntax.

```
SELECT * FROM TIMESERIES(
    INPUT=>(SELECT STATEMENT),
    FIELDS=>ARRAY['MEASURE1', 'MEASURE2,...'],
    LENGTH=>NUMBER,
    [DATE_COLS=>ARRAY[<DATE_FIELDS>, 'DATE_COLUMN_TYPE'],]
    [IGNORE_LAST=BOOLEAN,]
    [ORDER=>ARRAY['FIELD_NAME1', 'DESC'|'ASC', 'FIELD_NAME2', 'DESC'|'ASC',...],],
    [PARTITION=>'FIELD_NAME',]
    [PREDICTION_INTERVAL=>ARRAY[NUMBER],]
    [MODEL=>'MODEL_NAME',]
    [SEASONALITY=NUMBER]
)
```

Name	Description
`INPUT`	Required. A `SELECT` statement that includes date information and is the input to the `TIMESERIES()` function.
`FIELDS`	Required. The field values in the input query from which to predict future values. Predicted values are automatically named ***`predicted_fieldName`***.
`LENGTH`	Required. The number of points to predict.
`DATE_COLS`	Optional. ***`DATE_FIELDS`***—The array of date fields from which to base future predictions. The ***`DATE_COLUMN_TYPE`*** string accepts these values. • `'`***`YEAR_FIELD`***`'`, `'`***`MONTH_FIELD`***`'`, `'Y-M'` • `'`***`YEAR_FIELD`***`'`, `'`***`QUARTER_FIELD`***`'`, `'Y-Q'` • `'`***`YEAR_FIELD`***`'`, `'Y'` • `'`***`YEAR_FIELD`***`'`, `'`***`WEEK_FIELD`***`'`, `'Y-W'` • `'`***`YEAR_FIELD`***`'`, `'`***`MONTH_FIELD`***`'`, `'`***`DAY_FIELD`***`'`, `'Y-M-D'`
`ORDER`	Optional, unless no `DATE_COLS` are specified.

Name	Description
`PARTITION`	Optional. A field used to split query results into smaller partitions. The `TIMESERIES()` function resets when the field value changes. After each group of rows is completed for a given partition, `TIMESERIES()` runs on the next partition.
`PREDICTION_INTERVAL`	Optional. The confidence interval to display for each forecasted data point.
`IGNORE_LAST`	Optional. If set to `TRUE`, excludes the last time period from timeseries calculations.
`MODEL`	Optional. Choose from these prediction models: • `NONE` • `ADDITIVE` • `MULTIPLICATIVE`
`SEASONALITY`	Optional. Use with `DATE_COLS` to specify the seasonality for the prediction.

Example

This example predicts the annual number of tourists for two years based on available data for previous years.

```
SELECT * FROM TIMESERIES(
   INPUT=>(
        SELECT EXTRACT(YEAR FROM "date") AS date_Year, SUM(tourists) AS sum_tourists FROM
 "TouristsData" GROUP BY
   EXTRACT(YEAR FROM "date")),
   FIELDS=>ARRAY['sum_tourists'],
   LENGTH=>2,
   DATE_COLS=>ARRAY['date_Year', 'Y']
)
```

date (Year)	predicted_sum_tourists	Sum of tourists
2001	13008	13140
2002	13964	13543
2003	14934	15502
2004	15894	15894
2005	16855	16784
2006	17816	17713
2007	18777	18719
2008	19738	-
2009	20700	-

date (Year)	predicted_sum_tourists	Sum of tourists
2010	21662	-

Example with `TIMESERIES()` and `FILL()`

To fill in the gaps, the `FILL()` function takes the input of the annual tourist sums and fills in the gaps (if any) in date fields. `TIMESERIES()` takes the input from the `FILL()` function and predicts the number of tourists expected for the specified length. Here, `LENGTH` is set to 3, meaning three years. `IGNORE_LAST` is set to `TRUE` to exclude the last row for 2008 from generating timeseries predictions. `TIMESERIES()` predicts values from 2008–2010.

```
SELECT * FROM TIMESERIES(
    INPUT=>(
        SELECT * FROM FILL(
            INPUT=>(SELECT EXTRACT(YEAR FROM "date") AS date_Year, SUM(tourists) AS
sum_tourists
FROM "TouristsData"
GROUP BY EXTRACT(YEAR FROM "date")
            ),
            DATE_COLS=>ARRAY['date_Year', 'Y']
        )
    ),
    FIELDS=>ARRAY['sum_tourists'],
    LENGTH=>3,
    DATE_COLS=>ARRAY['date_Year', 'Y'],
    IGNORE_LAST=>TRUE
)
```

date (Year)	predicted_sum_tourists	Sum of tourists
2001	13008	13140
2002	13964	13543
2003	14934	15502
2004	15894	15894
2005	16855	16784
2006	17816	17713
2007	18777	18719
2008	19738	-
2009	20700	-
2010	21662	-

`WHERE` Clause

By default, a SQL query retrieves every row in your dataset. Use the optional `WHERE` clause to restrict your query results to a conditional expression.

The conditional expression in the `WHERE` clause takes the following syntax.

```
fieldExpression [logicalOperator fieldExpression2][...]
```

```
SELECT COUNT(*) as Count
FROM "Superstore"
WHERE City = 'San Francisco';
```

Count
510

You can add multiple field expressions to conditional expressions with logical operators.

```
SELECT COUNT(*) as Count FROM "Superstore"
WHERE City = 'Los Angeles' OR City = 'San Francisco';
```

Count
1,257

The `WHERE` clause supports these operators.

Type	Operator Name	Supported Types
Comparison Operators	!=, =	Numeric, String, Boolean
	<, <=, >, >=	Numeric, String
	`IN`, `NOT IN`	Numeric, String
Null Operators	`IS NULL, IS NOT NULL`	Numeric
Logical Operators	`AND, OR, NOT`	Boolean

The left operand of a comparison operator must be a valid field. The right operand cannot be a field. Arithmetic operators (e.g. +, -, /, *, %) between supported types can be used on the right hand side.

`WHERE` is limited to comparisons within a single column.

In a non-nested query (a non-subquery), you can't pass fields to a math or string function in a `WHERE` clause. You can pass a constant. To pass fields to a math or string function in a subquery, include it in the outer query's `WHERE` clause. If you use it in the innermost query, SQL throws an error.

Boolean Expressions in `SELECT` Clause

A boolean expression is a logical statement that returns either `true` or `false`.

A boolean expression in `SELECT` follows this syntax: `SELECT` ***`boolean_expression`*** `as` ***`alias`***.

This example demonstrates a query that satisfies one numerical logical condition.

```
SELECT Profit > 0 as Profits FROM "Superstore" LIMIT 5;
```

Profits
true
true
true
false
true

This example demonstrates a query that satisfies two numerical logical conditions.

```
SELECT Profit > 0 and Profit < 100 as Profits FROM "Superstore" LIMIT 5;
```

Profits
true
false
true
false
true

This example demonstrates a text comparison.

```
SELECT Customer_Name LIKE 'A%' as "A Names" FROM "Superstore" LIMIT 5;
```

A Names
false
false
false
false
true

SEE ALSO:

SELECT Clause

`GROUP BY` Clause

The `GROUP BY` clause organizes the rows returned from a `SELECT` statement into groups. Within each group, you can apply an aggregate function, like `count()` or `sum()` to get the number of items or sum, respectively.

In this example, the `SELECT` query counts the number of rows for each category and groups the counts by category.

```
SELECT Category, COUNT(*) AS "cnt"
FROM "Superstore"
GROUP BY Category;
```

Category	cnt
Furniture	2,121
Office Supplies	6,026
Technology	1,847

You can also group by multiple columns. This query generates the group of cities that contain stores in each category. The count column shows how many stores there are in each city.

```
SELECT Category, City, count(*) as cnt
FROM "Superstore"
GROUP BY Category, City
```

Here are the first ten results.

Category	City	cnt
Furniture	Akron	2
	Alexandria	3
	Allen	1
	Allentown	1
	Amarillo	4
	Anaheim	8
	Andover	1
	Apopka	1
	Apple Valley	2

Note: Grouping by ordinal number is not supported. You can't refer to a results column or an expression created by input values by its index value. To shorten a lengthy expression in your query, use an alias. For example, this query throws an error.

```
SELECT Category, City, count(*) as cnt
FROM "Superstore"
GROUP BY 1, 2
```

GROUP BY ROLLUP

ROLLUP is a sub-clause of GROUP BY that creates and displays aggregations of column data. The results output of ROLLUP is based on column order in your query.

ROLLUP supports the following aggregate functions.

- Average
- Count
- Min
- Max
- Sum

This example first groups the results by category and sub-category, and runs Sum(Profit), an aggregate function on each resulting row. By modifying the GROUP BY clause with ROLLUP, the query "rolls up" the results into subtotals and grand totals.

```
SELECT Category, Sub_Category, Sum(Profit) as TotalProfit
FROM "Superstore"
GROUP BY ROLLUP(Category, Sub_Category);
```

Notice how GROUP BY ROLLUP (Category, Sub_Category) creates groups for each column combination:

```
Category, Sub-Category
Category
() — null
```

The query first groups the total profit for each subcategory of a given category. Next, it groups is the total profit for a single category. Once each category's total profit is accounted for, the query generates the total profit for all categories.

Category	Sub-Category	TotalProfit
Furniture	Bookcases	-3,472.56
	Chairs	25,590.17
	Furnishings	13,059.14
	Tables	-17,725.48
	-	18,451.27
Office Supplies	Appliances	18,138.01
	Art	6,527.79
	Binders	30,221.76
	Envelopes	6,964.18
	Fasteners	949.52
	Labels	5,546.25
	Paper	34,053.57
	Storage	21,278.83
	Supplies	-1,189.1

Category	Sub-Category	TotalProfit
	-	122,490.8

You can also group by multiple columns. This query generates the group of cities that contain stores in each category. The count column shows how many stores there are in each city.

```
SELECT Category, City, count(*) as cnt
FROM "Superstore"
GROUP BY Category, City
```

Here are the first ten results.

Category	City	cnt
Furniture	Akron	2
	Alexandria	3
	Allen	1
	Allentown	1
	Amarillo	4
	Anaheim	8
	Andover	1
	Apopka	1
	Apple Valley	2

GROUPING()

Use the `GROUPING()` clause with `ROLLUP` to determine whether null values are the result of a `ROLLUP` operation or part of your dataset. `GROUPING()` returns 1 if the null value is a subtotal generated by a rollup. It returns 0 otherwise.

This example uses the `GROUPING()` function and `CASE` statements together to label the subtotal and grand total categories. The first case checks for a null value generated by the rollup in the `Category` field. If true, then the query labels the field "All Categories." Similarly, the second case checks whether a `Sub-Category` field is null. If true, the query labels the field "All Sub-Categories."

```
SELECT
(CASE WHEN GROUPING("Category") = 1 THEN 'All Categories' ELSE "Category" END) as "Category",
(CASE WHEN GROUPING("Sub_Category") = 1 THEN 'All Sub-Categories' ELSE "Sub_Category" END)
 as "SubCategory",
SUM("Sales") as "sum_sales"
FROM "Superstore"
GROUP BY ROLLUP("Category", "Sub_Category");
```

Category	Sub-Category	sum_sales
Furniture	Bookcases	114,880

Category	Sub-Category	sum_sales
	Chairs	328,449.1
	Furnishings	91,705.16
	Tables	206,965.53
	All Sub-Categories	741,999.8
Office Supplies	Appliances	107,532.16
	Art	27,118.73
	Binders	203,412.73
	Envelopes	16,476.4
	Fasteners	3,024.28
	Labels	12,486.31
	Paper	78,479.21
	Storage	223,843.61
	Supplies	46,673.544
	All Sub-Categories	719,047.03
Technology	Accessories	167,380.32
	Copiers	149,528.03
	Machines	189,238.63
	Phones	330,007.05
	All Sub-Categories	838,154.03
All Categories	All Sub-Categories	2,297,200.86

SEE ALSO:

- Aggregate Functions
- GROUP BY Clause

`HAVING` Clause

Use the `HAVING` clause to filter grouped results from grouped columns, aggregate functions, or grouping functions.

`HAVING` follows this syntax: `HAVING` ***`search-condition`***.

A search condition can contain boolean expressions, comparison operators, and scalar functions.

Grouped Columns

This query returns groups of cities that contain furniture stores and their counts. The `HAVING` clause filters for the `Furniture` category.

```
SELECT Category, City, count(*) as "cnt" FROM "Superstore" GROUP BY Category, City HAVING
 Category = 'Furniture' LIMIT 10
```

Category	City	cnt
Furniture	Akron	2
	Alexandria	3
	Allen	1
	Allentown	1
	Amarillo	4
	Anaheim	8
	Andover	1
	Apopka	1
	Apple Valley	2
	Arlington	13

You can also write this query using `WHERE`.

```
SELECT Category, City, count(*) as "cnt" FROM "Superstore" WHERE Category = 'Furniture'
GROUP BY Category, City LIMIT 10
```

In the logical execution of a `SELECT` statement, `WHERE` filters out rows before any grouping or aggregate function runs. In this example, it selects the rows that can be passed to the aggregate function `count()`. The `WHERE` clause comes before a `GROUP BY` statement. With `HAVING`, the filtering occurs after all of the rows have been passed to `count()`. `HAVING` always comes after a `GROUP BY` statement. Both `HAVING` and `WHERE` run before projection.

Aggregate Functions

This query returns groups of cities that have more than 150 stores in each category. The `HAVING` clause filters out values greater than 150 in the aggregate column, `cnt`.

```
SELECT Category, City, count(*) as cnt FROM "Superstore"
GROUP BY Category, City HAVING count(*) > 150
```

Category	City	cnt
Furniture	Los Angeles	154
	New York City	192
Office Supplies	Houston	231

Category	City	cnt
	Los Angeles	443
	New York City	552
	Philadelphia	312
	San Francisco	322
	Seattle	249
Technology	New York City	171

Using `HAVING` with an aggregate function automatically implies `GROUP BY()`. These two queries return the same result.

```
SELECT Sum(Profit) as TotalProfit
FROM "Superstore"
HAVING Sum(Profit) > 0
```

```
SELECT Sum(Profit) as TotalProfit
FROM "Superstore"
GROUP BY ()
HAVING Sum(Profit) > 0
```

TotalProfit
286,397.02

GROUP BY ROLLUP

Here's a simple example of how to use `HAVING` with `GROUP BY ROLLUP`. The query returns the profit for each `Sub-Category`, rolls them up into subtotals of each `Category`, and then sums up the grand total. Here, `HAVING` filters on positive profits.

```
SELECT Category, Sub_Category, Sum(Profit) as TotalProfit FROM "Superstore" GROUP BY
ROLLUP(Category, Sub_Category) HAVING Sum(Profit) > 0;
```

Category	Sub-Category	TotalProfit
Furniture	Chairs	26,590.17
	Furnishings	13,059.14
	-	18,451.27
Office Supplies	Appliances	18,138.01
	Art	6,527.79
	Binders	30,221.76
	Envelopes	6,964.18
	Fasteners	949.52

Category	Sub-Category	TotalProfit
	Labels	5,546.25
	Paper	34,053.57
	Storage	21,278.83
	-	122,490.8
Technology	Accessories	41,936.64
	Copiers	55,617.82
	Machines	3,384.76
	Phones	44,515.73
	-	145,454.95
	-	286,397.02

`GROUP BY ROLLUP` with `GROUPING()`

In SQL for CRM Analytics, you can use `HAVING()` with the `GROUP BY ROLLUP` subclause and the `GROUPING()` function. You can use `GROUPING()` with `ROLLUP` only; you can't use it on its own.

This example uses the `GROUPING()` function with the `HAVING()` function to filter for the subtotals of each category and return the grand total. The `GROUPING()` function returns 1 for null values that are the result of a `ROLLUP`, and not null values in your dataset. Here, by setting `GROUPING(Sub_Category) = 1`, we know that these `Sub-Category` values refer to the subtotals of each category.

```
SELECT Category, Sub_Category, Sum(Profit) as TotalProfit
FROM "Superstore"
GROUP BY ROLLUP(Category, Sub_Category)
HAVING GROUPING(Sub_Category) = 1
```

Category	Sub-Category	TotalProfit
Furniture	-	18,451.27
Office Supplies	-	122,490.8
Technology	-	145,454.95
-	-	286,397.02

SEE ALSO:

WHERE Clause

`ORDER BY` Clause

`SELECT` returns rows in an unspecified order by default. To sort returned rows in ascending or descending order, use the `ORDER BY` clause.

`ORDER BY` takes the following syntax.

```
ORDER BY field_name(s) [ ASC | DESC ] [ NULLS { FIRST | LAST }]
```

This example shows how `ORDER BY` works with multiple fields.

```
SELECT State, Category, Sub_Category FROM "Superstore" ORDER BY State, Category;
```

Here, the `SELECT` statement orders by the State and Category columns. Alabama and Furniture are the first alphabetized fields for their respective columns. For all of the rows where State is equal to Alabama and Category is equal to Furniture, SQL returns the Sub-Category column values without any order. When State is equal to Alabama, and Category has displayed all of the Furniture values, the Category column displays Office Supplies, the next value in alphabetical order.

State	Category	Sub-Category
Alabama	Furniture	Chairs
Alabama	Furniture	Chairs
Alabama	Furniture	Chairs
Alabama	Furniture	Chairs
Alabama	Furniture	Furnishings
Alabama	Furniture	Tables
Alabama	Furniture	Chairs
Alabama	Furniture	Furnishings
Alabama	Furniture	Tables
Alabama	Furniture	Furnishings
Alabama	Furniture	Tables
Alabama	Furniture	Chairs
Alabama	Office Supplies	Appliances

To order the columns in ascending or descending order, use the keywords `ASC` and `DSC`. The order is ascending by default. To sort fields that contain null values, specify `NULLS FIRST` or `NULLS LAST`. By default, null values are treated as the largest. In ascending order, they are ordered last. In descending order, they are first.

`LIKE` Clause

Use `LIKE` to match single characters and patterns found anywhere in a string—beginning, ending, or somewhere in between.

`LIKE` takes this syntax.

`[NOT] LIKE` ***`pattern`***

`NOT`—Optional.

Returns `True` if the left operand matches the pattern on the right. This operator is case-sensitive. A pattern must be at least two characters.

To match any single character in the string, include an underscore (_). To match any pattern, include a percent sign (%). Starting a pattern with a percent sign returns all words that end with the specified characters. The percent sign indicates that any number of characters can precede the specified ones. Ending a pattern with a percent sign returns all the words that begin with the specified characters. Any number of characters can follow the specified ones. To match a pattern anywhere in a string, the pattern must start and end with a percent sign.

For example, let's say we have a list of countries. A query that filters on the pattern `%a%a%` returns Bahamas, Jamaica, Slovakia, and Trinidad and Tobago. If we change the pattern to `_a_a____`, the query returns Bahamas and Jamaica.

To include a literal percent sign or underscore in a pattern, escape them with a backwards slash (`\`).

Additional wildcard characters and regular expressions aren't supported.

Here are some more examples.

Simple Pattern in a String

This example checks whether a customer name contains the characters "ni" anywhere in the string. If the string contains "ni", then the condition evaluates to `True` and the query returns the name.

```
SELECT Customer_Name as 'name'
FROM "Superstore"
WHERE Customer_Name LIKE "%ni%";
GROUP BY Customer_Name;
LIMIT 5;
```

name
Annie Thurman
Annie Zypern
Benjamin Venier
Berenike Kampe
Chad Cunningham

One Underscore

This query returns city names that contain any single character preceding "lb" and zero or more characters following it.

```
SELECT City as 'city'
FROM "Superstore"
WHERE City LIKE "_lb%";
GROUP BY City;
```

city
Albuquerque

Multiple Underscores

If we precede the characters "lb" with two underscores, the query returns results that have any two characters before "lb."

```
SELECT City as 'city'
FROM "Superstore"
WHERE City LIKE "__lb%";
GROUP BY City;
```

city
Gilbert
Melbourne

End with Percent Sign

This query matches names that begin with "An." These names include Andrew Levine, Annette Boone, Annette Cline, and Annie Horne.

```
WHERE Customer_Name LIKE "An%"
```

Exclude Records with `NOT`

This query shows all customer names that don't contain "po." These names include Aaron Davies Bruce, Aaron Day, Aaron Dillon, and Aaron Riggs.

```
WHERE Customer_Name NOT LIKE "%po%"
```

SEE ALSO:

String Functions and Operators

`BETWEEN` Operator

Use `BETWEEN` to check whether values fall within a given range. `BETWEEN` accepts numeric, string, and date data types, and can be used with aggregate, window, and math functions.

`BETWEEN` takes this syntax.

```
<expr> BETWEEN (SYMMETRIC | ASYMMETRIC) <lower_bound> AND <upper_bound>
```

Numeric Example

This example filters results on flights whose prices are between $300 and $600.

```
SELECT price, origin, dest
FROM FlightsData
WHERE price BETWEEN 300 and 600
ORDER BY price ASC
```

Dest	Origin	Price
PHX	LAX	300
LAX	PHX	400
PHX	LAX	400
LAX	PHX	500
LAX	SFO	550
LAX	OAK	560
SFO	LAX	600
OAK	LAX	600
LAX	PHX	600

Aggregate and Window Functions Example

This example checks whether the sum of profits for each account is less than 5% of the total sum of profits for all accounts. It returns the account name, account ID, and `IsLowPercent`, a boolean value that is `true` if the sum of profits is less than 5%.

```
SELECT Account_ID, Account_Name, SUM(Profit) BETWEEN 0 AND 5 * SUM(SUM(Profit)) OVER (ROWS
 BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) / 100 AS IsLowPercent
FROM "Opportunity"
GROUP BY Account_ID, Account_Name
LIMIT 10;
```

Account ID	Account Name	isLowPercent
00137000003dT6qAAE	Alphenymp	true
00137000003dT6rAAE	AboveRosa	true
00137000003dT6sAAE	Acidyles	true
00137000003dT6tAAE	Angelstage	true
00137000003dT6uAAE	Animorror	true
00137000003dT6vAAE	AnnouncerKing	true
00137000003dT6wAAE	ApenguinInca	true
00137000003dT6xAAE	AttractivePenguin	true
00137000003dT6yAAE	BearDigestAir	true
00137000003dT6zAAE	BertramWillow	true

Date Example with **TIMESTAMP**

This example returns ten `CloseDate` values that are between two `TIMESTAMP` values, `2014-12-31` and `2015-12-31`.

```
SELECT CloseDate
FROM "Opportunity"
WHERE CloseDate BETWEEN TIMESTAMP '2014-12-31 00:00:00' AND TIMESTAMP '2015-12-31 00:00:00'
LIMIT 10;
```

CloseDate
2014-12-31 16:00:00
2015-01-01 16:00:00
2015-01-30 16:00:00
2015-01-31 16:00:00
2015-02-28 16:00:00
2015-03-30 17:00:00
2015-03-31 17:00:00
2015-04-29 17:00:00
2015-04-30 17:00:00
2015-05-30 17:00:00

Date Example with **EXTRACT**

This example uses SQL for CRM Analytics's `EXTRACT` function to access the `MONTH` values from the `CloseDate` field and return dates between April (4) and July (7).

```
SELECT CloseDate
FROM "Opportunity"
WHERE EXTRACT(MONTH FROM CloseDate) BETWEEN 4 AND 7
LIMIT 10;
```

CloseDate
2015-04-29 17:00:00
2015-04-30 17:00:00
2015-05-30 17:00:00
2015-05-31 17:00:00
2015-07-31 17:00:00

Asymmetric and Symmetric Example

By default, the `BETWEEN` statement evaluates a value that falls between the lower and upper bounds in order of least to greatest as `true`. If the lower and upper bound values are reversed, `BETWEEN` evaluates the statement as `false`. To specify this behavior, use the `ASYMMETRIC` operator.

Let's go back to the numeric example.

Here, the range is specified as `ASYMMETRIC BETWEEN 300 and 600`. The statement evaluates to `true`.

```
SELECT price, origin, dest
FROM FlightsData
WHERE price ASYMMETRIC BETWEEN 300 and 600
ORDER BY price ASC
```

When the range is specified as `ASYMMETRIC BETWEEN 600 and 300`, the statement evaluates to `false`.

```
SELECT price, origin, dest
FROM FlightsData
WHERE price ASYMMETRIC BETWEEN 600 and 300
ORDER BY price ASC
```

By using the `SYMMETRIC` operator, the order of the upper and lower bounds of your `BETWEEN` statement remain true if reversed. The order is irrelevant.

```
SELECT price, origin, dest
FROM FlightsData
WHERE price SYMMETRIC BETWEEN 600 and 300
ORDER BY price ASC
```

By using `SYMMETRIC`, the `BETWEEN` statement evaluates to `true`.

Here's the date example using `EXTRACT` and the `SYMMETRIC` operator. By including `SYMMETRIC`, the query returns the same results, even though the months are filtered `BETWEEN 7 AND 4`.

```
SELECT CloseDate
FROM "Opportunity"
WHERE EXTRACT(MONTH FROM CloseDate) SYMMETRIC BETWEEN 7 AND 4
LIMIT 10;
```

SEE ALSO:

- Access Parts of a Date
- WHERE Clause

`LIMIT` Clause

The `LIMIT` clause specifies the maximum number of rows to return. If no `LIMIT` clause is specified, then SQL returns all rows.

`LIMIT` follows the syntax `LIMIT {count}`.

Important: When working with two or more columns, make sure to use `LIMIT` with an `ORDER BY` clause, so that your results are returned in a specified order. Otherwise, results return in an order you may not want. ORDER BY is optional for working with a single column.

Here's an example.

```
SELECT City
FROM "Superstore"
ORDER BY City
LIMIT 5;
```

This query returns the cities in the first three rows of the dataset.

City
Aberdeen
Abilene
Akron
Akron
Akron

`FETCH` Clause

The `FETCH` clause specifies the number of rows to return. If no `FETCH` clause is specified, then SQL returns all rows.

`FETCH` follows this syntax. `FETCH {FIRST||NEXT}` ***`count`*** `{ROW|ROWS} ONLY`

FIRST
Returns the count of rows beginning from the first row in the dataset.

NEXT
Returns the count of rows that immediately follow the current row.

count
Optional. The number of rows to return. If you don't specify a count, `FETCH` returns one row by default.

Note: `FETCH FIRST ROW ONLY` is the same as `FETCH FIRST 1 ROW ONLY`.

In this example, because there's an offset of one row, `FETCH` starts from the second value of the `City` column, ordered ascending. The query returns the next three rows, which are the second, third, and fourth rows.

```
SELECT City
FROM "Superstore"
OFFSET 1
ORDER BY City
FETCH NEXT 3 ROWS ONLY;
```

City
Abilene
Abilene
Akron

Here, the query returns the first row.

```
SELECT City
FROM "Superstore"
ORDER BY City
FETCH FIRST ROW ONLY;
```

City
Aberdeen

`OFFSET` Clause

By default, a SQL query retrieves every row in your dataset. Use the optional `WHERE` clause to restrict your query results to a conditional expression.

`OFFSET` takes the syntax `OFFSET {`***`start`***`}`. *`Start`* refers to the number of rows to skip before returning rows.

This example skips over rows 1-5. It returns the three subsequent rows, 6-8.

```
SELECT City
FROM "Superstore"
ORDER BY City
OFFSET 5
LIMIT 3;
```

City
Aberdeen
Abilene
Abilene

`CASE` Statements

Use case statements to express if/then logic. A case statement always has a pair of `WHERE` and `THEN` statements. CRM Analytics supports the simple and searched forms of case expressions in a `SELECT` statement. Case statements have two formats: simple and searched.

Simple Form

A simple statement compares a case expression against a set of expressions. The result is the matched expression.

```
SELECT Category, CASE Category
WHEN 'Furniture' THEN 'Available'
WHEN 'Office Supplies' THEN 'Unavailable'
ELSE 'Unknown' END AS Availability
FROM "Superstore"
GROUP BY Category;
```

Here are the results.

Category	Availability
Furniture	Available
Office Supplies	Unavailable
Technology	Unknown

Searched Form

The searched form evaluates to a result. A searched statement compares an expression against a series of boolean expressions. If it matches a boolean expression, the result is the corresponding THEN clause. If the expression does not return true for any of the boolean expressions, then the result is the corresponding `ELSE` clause.

```
SELECT Region, CASE
WHEN sum(Profit) <= -500000 THEN 'Huge Loss'
WHEN sum(Profit) <= 0 THEN 'Loss'
WHEN sum(Profit) > 500000 THEN 'Huge Profit'
ELSE 'Profit' END AS ProfitLoss
FROM "Superstore"
GROUP BY Region;
```

Region	ProfitLoss
Central	Profit
East	Profit
South	Profit
West	Profit

COALESCE()

You can use the `coalesce()` function as shorthand for case statements. The `coalesce()` function replaces null values in your dataset with another value. The function takes a series of arguments and returns the first value that is not null.

In this query, the first case statement says that if the City value is not null, return the City value. Otherwise, return `NULL`. For the second case, it says if a Country value is not null, return the Country. If it is null, return the string "Unknown."

```
SELECT CASE WHEN City
IS NOT NULL
THEN City
ELSE NULL
END AS City,
CASE WHEN Country
IS NOT NULL
ELSE 'Unknown'
END As Country
FROM "Superstore"
Group by City, Country;
```

Here's the same query rewritten with `coalesce()`.

```
SELECT COALESCE(City, NULL) as City, COALESCE(Country, 'Unknown') as Country
FROM "Superstore"
GROUP BY City, Country;
```

There are no null City or Country values in the first five results returned from the query.

City	Country
Aberdeen	United States
Abilene	United States
Akron	United States
Albuquerque	United States
Alexandria	United States

NULLIF()

Use the `nullif()` function as shorthand for a searched case statement where two equal expressions return null.

`nullif()` takes the following syntax. Its two arguments are the expressions that you want to compare.

```
 nullif(fieldName1, fieldName2)
```

In this example, the query returns null for all City values that are Aberdeen. If the City value is not Aberdeen, then it returns the City value.

Here’s how to write the query as a case statement.

```
SELECT CASE WHEN City='Aberdeen'
THEN NULL
ELSE City
END AS City
FROM "Superstore"
GROUP BY City;
```

Here's how to write the same query using `nullif()`.

```
SELECT NULLIF(City, 'Aberdeen') as City
FROM "Superstore"
GROUP BY City;
```

City
Aberdeen
Akron
Albuquerque
Alexandria

`UNION` Operator

Use the `UNION` operator to combine the results of two or more `SELECT` statements. The joined statements must have the same number of columns and the same data types for corresponding columns.

SQL for CRM Analytics supports `UNION ALL` only. This clause returns all results—it doesn't remove duplicate values. Let's see what happens when we combine two `SELECT` statements that retrieve the `Cities` field.

```
SELECT City From "Superstore" UNION ALL SELECT City from "Superstore";
```

The query returns all `City` results in an unspecified order.

City
Henderson
Henderson
Los Angeles
Fort Lauderdale
Fort Lauderdale
Los Angeles
Los Angeles
Los Angeles
Los Angeles
Los Angeles

To include an `ORDER BY` or `LIMIT` clause when using the `UNION` operator, you must enclose the first `SELECT` statement in parentheses so that the SQL parser can identify the `UNION` operator.

In this example, `ORDER BY City DESC` applies to the results of the full query, not only the second `SELECT` clause. The results display in reverse alphabetical order for the combined set of the `City` field.

```
(SELECT City
FROM "Superstore"
ORDER BY City ASC)
UNION ALL
SELECT City
FROM "Superstore"
ORDER BY City DESC;
```

City
Yuma
Yuma
Yuma
Yuma

City
Yuma
Yuma
Yuma
Yuma
Yucaipa
Yucaipa
York

If you include parentheses around the second `SELECT` statement, the query returns results from the first `SELECT` statement in ascending order and the second in descending order.

```
(SELECT City
FROM "Superstore"
ORDER BY City ASC)
UNION ALL
(SELECT City
FROM "Superstore"
ORDER BY City DESC);
```

The first ten results show values from the first `SELECT` statement in ascending order.

City
Aberdeen
Abilene
Akron
Akron
Akron
Akron
Akron
Akron
Akron
Akron

Subquery

A subquery is a query that is nested inside a `SELECT` statement. Nesting queries allows you to perform multi-step operations.

A subquery follows this syntax.

```
SELECT field
FROM ( select_statement || union_all )
```

To pass fields to a math or string function in a subquery, include it in the outer query's `WHERE` clause. If you use it in the innermost query, SQL throws an error.

Simple Example

This query returns the sum of all profits and the max profit value from the Superstore dataset where all the profits are greater than zero. Values greater than zero mean that no losses are reflected in the results.

```
SELECT d.sumProfit, maxProfit
FROM (
   SELECT SUM(Profit) as sumProfit, MAX(Profit) as maxProfit
   FROM (
       SELECT Profit
       FROM "Superstore"
       WHERE Profit > 0
       )
   ) as d;
```

CRM Analytics first runs the innermost query: `SELECT Profit FROM "Superstore" WHERE Profit > 0`.

The output from this query becomes the dataset on which the outer query operates.

```
SELECT SUM(Profit) as sumProfit, MAX(Profit) as maxProfit
FROM (innermost query);
```

Optionally, you can refer to an inner query by its alias, if you choose to use one. In the outermost query, you can refer to `sumProfit` as `d.sumProfit`, as shown in the example, or you can refer to it simply as `sumProfit`.

```
SELECT d.sumProfit, maxProfit FROM (nested subqueries as d);
```

sumProfit	maxProfit
442,649.60	8,399.98

Subqueries with `UNION ALL`

The `UNION ALL` operator combines the results of two datasets into one. It doesn't remove any duplicate rows. This example includes `UNION ALL` after a subquery.

The first part of this example, up to the `UNION ALL` statement, returns the total profit and the maximum profit values from the Superstore dataset. After the `UNION ALL` statement, the query returns the total and maximum profit values from the ClearanceSales dataset. Note that ClearanceSales is intended for example purposes only.

```
SELECT d.sumProfit as total, maxProfit as max_profit
   FROM (
       SELECT SUM(Profit) as sumProfit, MAX(Profit) as maxProfit
       FROM (
           SELECT Profit
           FROM "Superstore"
           WHERE Profit > 0
```

```
            )
        ) as d
UNION ALL (
   SELECT SUM("Profit") as total, MAX("Profit") as max_profit
   FROM "ClearanceSales"
   GROUP BY ()
)
```

total	max_profit
442,649.60	8,399.98
286,397.02	4,519.50

You can also include `UNION ALL` within a subquery. This example finds all the names in the Superstore and Opportunity datasets that contain "an." The outer query then filters for names that have counts over 30 and returns that subset of results.

```
SELECT Customer_Name as "name", "count"
FROM (
   (SELECT Customer_Name, count(*) as "count"
   FROM "Superstore"
   WHERE Customer_Name LIKE '%an%'
   GROUP BY Customer_Name
) UNION ALL (
   SELECT User_Name as "name", count(*) as "count" FROM "Opportunity" WHERE User_Name LIKE
 '%an%' GROUP BY User_Name
   )
) WHERE "count" > 30
```

name	count
Emily Phan	31
Jonathan Doherty	32
Matt Abelman	34
Han Solo	654
Indiana Jones	583

Window Functions

A window function lets you perform calculations on a selection—or "window"—of rows that are related to the current row. Unlike a regular aggregate function, such as `avg()`, `sum()`, or `count()`, the row output of a window function isn't grouped into a single row.

A window function takes this syntax.

```
<window function>(<projection expression>) OVER ([PARTITION BY <reset groups>] [ORDER
BY order_clause] <frame_clause>) AS <label>
```

Name	Description
window function	There are two types of window functions. An aggregate function performed over a specified group of rows related to the current row. SQL for CRM Analytics supports these aggregate functions. • `avg()` • `sum()` • `min()` • `max()` • `count()` • `percentile_disc()` • `percentile_cont()` A ranking function that returns a rank value for each row in the partition. SQL for CRM Analytics supports these rank functions. • `rank()` • `dense_rank()` • `cume_dist()` • `row_number()`
projection expression	The expression that serves as the input to the window function.
`PARTITION BY`	Optional. Splits query results into smaller partitions based on the ***`reset_group`***. When provided, the window function resets after it runs on each part. When you don't include `PARTITION BY`, all rows are treated as part of a single partition and the function doesn't reset.
reset_group	One or more columns that reset the windowing aggregation function when their value (or values) change.
`ORDER BY`	Optional. Provides a sorting order for rows in each partition. If ordering rows in a partition isn't relevant to your query, then don't include this clause. `ORDER BY` within a window function isn’t the sorting order for your query’s results. To specify the order of query results, include the `ORDER BY` clause at the end of your query.
order_clause	The expressions by which to order the results within a window function partition.
frame_clause	Specifies a subset of rows within the current partition that make up the window "frame." The frame has a start and end value, such as `ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`. If no end value is specified, the frame's end value

Name	Description
	defaults to the current row. For example, for `ROWS UNBOUNDED PRECEDING`, the frame starts at the beginning of the partition and ends at the current row. See the Example Frame Clause Values table for more examples.
label	The output column name.

Table 1: Example Frame Clause Values

Frame Clause Value	Description
`ROWS UNBOUNDED PRECEDING`	From the beginning of the partition to the current row.
`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`	From the beginning of the partition to the current row.
`ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`	From the current row to the last row in the partition.
`ROWS BETWEEN UNBOUNDED PRECEDING TO` ***number*** `PRECEDING`	From the beginning of the partition to number rows before the current row.
`ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`	From the beginning of the partition to the end of the partition. Aggregates the full partition.
`ROWS BETWEEN CURRENT ROW AND` ***number*** `FOLLOWING`	From the current row to number rows after the current row in the partition.

A ranking function returns a ranking value for each row in a partition.

Table 2: Supported Ranking Functions

Function Name	Description
`RANK()`	Returns a ranking value for each row within a partition. If values are tied, `RANK()` assigns them the same ranking value and skips over the next ranking. For example, if 2 items are ranked at 2, the next-ranked value is 4.
`DENSE_RANK()`	Returns a ranking value for each row within a partition. Unlike `RANK()`, `DENSE_RANK()` doesn't skip over a rank value if multiple entries are tied. If 2 items are ranked at 2, the next-ranked value is 3.
`CUME_DIST()`	Returns the cumulative distribution between entries in a reset group.
`ROW_NUMBER()`	Returns the sequential number of a row in a partition, starting at 1.

Aggregation Example

Windowing in SQL for CRM Analytics is available for grouped queries only. This example looks at flights grouped by origin. The query has 3 window functions that demonstrate variations on the 3 parts of a windowing function: partitioning, ordering, and framing.

```
SELECT origin, sum(price) as sum1,
sum(sum(price)) OVER (ORDER BY sum(price) ROWS UNBOUNDED PRECEDING) as sum2,
sum(sum(price)) OVER (PARTITION BY origin ORDER BY sum(price) ROWS UNBOUNDED PRECEDING)
as sum3,
sum(sum(price)) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS sum4
FROM "FlightsData"
WHERE dest='LAX'
GROUP BY origin
ORDER BY sum1
```

origin	sum1	sum2	sum3	sum4
PHX	1500	1500	1500	5430
SFO	1950	3450	1950	5430
OAK	1980	5430	1980	5430

This query first executes the `WHERE` and `GROUP BY` clauses, so that each expression operates on flights whose destination is LAX. Each group has a column that contains the sum of all of its respective flights' prices.

Note: The `WHERE`, `GROUP`, and `HAVING` clauses execute before windowing functions.

The subset of data with which we're concerned looks like this.

Origin	Price
PHX	400
PHX	500
PHX	600
SFO	550
SFO	650
SFO	750
OAK	560
OAK	660
OAK	760

We've included `sum(price)` as `sum1` in this example for demonstration purposes only. The window functions return the same results regardless of whether `sum(price)` is included in the `SELECT` statement. `sum(price)` is a regular aggregate function. The expressions that follow it use `sum()` as a window function.

Here's the output for the first part of the query calculation.

```
SELECT origin, sum(price) as sum1
FROM "FlightsData"
WHERE dest='LAX'
GROUP BY origin
```

Origin	sum1
PHX	1500
SFO	1950
OAK	1980

Let's break down the windowing functions line by line.

The first windowing function is: `sum(sum(price)) OVER (ORDER BY sum(price) ROWS UNBOUNDED PRECEDING) as sum2`. The window function `sum()` takes the argument `sum(price)` as the input for the window function `sum2`. The clause after `OVER` specifies the rows and the order in which the function operates on them. Since there's no partitioning clause, all of the rows belong to the same partition.

The rows are sorted in ascending order of `sum(price)`. `ROWS UNBOUNDED PRECEDING` says apply this operation—`sum(sum(price))`—from the beginning of the partition to the current row. Because the query is already grouped by origin, there's only one value corresponding to each origin. For `PHX`, that means taking the sum of `1500`. For the next row, `SFO`, the function calculates the sum of `sum1` in addition to the unbounded preceding ones—the `sum1` value of `PHX`, `1500 + 1950`. For the last group, `OAK`, the function sums up the `OAK` total in addition to the two preceding rows, `1500 + 1950 + 1980`. This window function computes a running sum of `sum(price)`, ordered by `sum(price)`.

The second windowing function is: `sum(sum(price)) OVER (PARTITION BY origin ORDER BY sum(price) ROWS UNBOUNDED PRECEDING)`. Partitioning by origin returns the groupings that the query established with the initial `GROUP BY` clause. By including the clause `ROWS UNBOUNDED PRECEDING`, we take the sum of all of the preceding rows within a particular origin. Since each origin's prices are already summed up, taking the sum is just each summed value. The results are the same as the results for `sum1`.

The third windowing function is: `sum(sum(price)) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) as sum4`. Since partitioning isn't specified, all rows are part of the same partition. `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` starts the frame at the first row of the partition and continues to the last row. The function aggregates the full partition, which in this example is all of the rows.

Ranking Example

This example demonstrates the differences between the `RANK()` and `DENSE_RANK()` functions.

```
SELECT origin, SUM(price) as s,
    RANK() OVER (ORDER BY SUM(price) ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
FOLLOWING) as rnk,
    DENSE_RANK() OVER (ORDER BY SUM(price) DESC ROWS BETWEEN UNBOUNDED PRECEDING AND
UNBOUNDED FOLLOWING) as denserank
FROM "FlightsData"
GROUP BY origin
```

origin	s	rnk	denserank
LAX	5400	8	1
HI	3300	6	2
SJC	3300	6	2
ASE	2700	5	3
ORD	2400	4	4
OAK	1980	3	5
SFO	1950	2	6
PHX	1500	1	7

The `RANK()` function assigns the sum of the prices for all flights from `LAX` the highest rank. Because the `DENSE_RANK()` function specifies to order the rows descending, it assigns `LAX` the rank `1`.

Since `HI` and `SJC`'s sums are the same value, 3300, instead of assigning them values of `7` and `6`, `RANK()` skips over `7`. It assigns them both the rank `6`. By contrast, `DENSE_RANK()` assigns consecutive ranking values. It assigns `HI` and `SJC` the value `2`. `HI` and `SJC` have the next highest descending value after `LAX`, which has the ranking value `1`. The next value `DENSE_RANK()` assigns to `ASE` is `3`.

SEE ALSO:

Aggregate Functions

DATE FUNCTIONS

SQL for CRM Analytics supports `DateTime`, `DateOnly`, and `Date` date types. If you're familiar with standard SQL, the `DateOnly` type corresponds with SQL's `Date` type and the `DateTime` type corresponds with SQL's `Timestamp` type. `DateTime` and `DateOnly` have custom time zone support. If your org doesn't support custom time zones, then use the `Date` type, which supports GMT date information only.

A `DateTime` date type follows this format.

```
DateTimeFormat = "2006-01-02 15:04:05"
```

A `DateOnly` date type follows this format.

```

DateOnlyFormat = "2006-01-02"
```

Access Parts of a Date
Use the `EXTRACT()` function to access parts of a date. You can use `EXTRACT()` in projections, filtering, grouping and ordering.

Project a Date Field
To project a date field of type `DateTime`, use the `EXTRACT()` function on the date field in the `SELECT` statement.

Filter By Date Parts or Date Field
To filter by a date part, use the `WHERE` clause with the `EXTRACT()` function, and pass it the desired date parts (year, month, or day). To filter by a date field, use the `WHERE` clause with a logical operator.

Group By Date Part
To group by date part, use the `GROUP BY` clause and the `EXTRACT()` function. Pass `EXTRACT()` the date parts to isolate.

Order By Date Part
To order by date part, use `EXTRACT()` on the date field with the `ORDER BY` clause. The query returns the count of rows containing these same values and orders the results by count descending.

Project a Custom Fiscal Date Part
To project a custom fiscal date part, pass custom fiscal date parts to the `EXTRACT()` function.

Day in Week, Month, Quarter or Year Functions
Use the following functions to find the position of a day within a week, month, quarter, or year.

First and Last Day in the Week, Month, Quarter or Year Functions
Use the following functions to find the first and last week, month, quarter, or year for both standard and fiscal calendars. These functions accept `DateTime`, `DateOnly`, and legacy input values. They return the same type as the input value.

SEE ALSO:

Enable Custom Time Zones

Access Parts of a Date

Use the `EXTRACT()` function to access parts of a date. You can use `EXTRACT()` in projections, filtering, grouping and ordering.

`EXTRACT()` follows this syntax.

```
EXTRACT(date_part FROM date_field)
```

A date part can be one of the following values.

- `YEAR`
- `QUARTER`
- `MONTH`
- `WEEK`
- `DAY`
- `HOUR`
- `MINUTE`
- `SECOND`
- `EPOCH_DAY`
- `EPOCH`
- `FISCAL_YEAR`
- `FISCAL_QUARTER`
- `FISCAL_MONTH`
- `FISCAL_WEEK`

Note: If you query a date field that doesn't contain time information, it will return 0.

Project a Date Field

To project a date field of type `DateTime`, use the `EXTRACT()` function on the date field in the `SELECT` statement.

```
SELECT CloseDate
FROM "OpportunityFiscalEMTimezoned"
LIMIT 1;
```

CloseDate
2014-12-30 16:00:00

This example projects only the year from the `DateTime` field.

```
SELECT EXTRACT(Year FROM CloseDate) As CloseDate_Year
FROM "OpportunityFiscalEMTimezoned"
LIMIT 1;
```

CloseDate_Year
2014

Filter By Date Parts or Date Field

To filter by a date part, use the `WHERE` clause with the `EXTRACT()` function, and pass it the desired date parts (year, month, or day). To filter by a date field, use the `WHERE` clause with a logical operator.

This example filters by year.

```
SELECT CloseDate
FROM "OpportunityFiscalEMTimezoned"
WHERE EXTRACT(YEAR FROM CloseDate) = 2014;
```

CloseDate
2014-12-31 15:00:00
2014-12-30 16:00:00

This example returns CloseDate fields that occur on or before the given timestamp.

```
SELECT CloseDate
FROM "OpportunityFiscalEMTimezoned"
WHERE CloseDate <= TIMESTAMP '2014-12-31 15:00:00';
```

CloseDate
2014-12-30 16:00:00

Group By Date Part

To group by date part, use the `GROUP BY` clause and the `EXTRACT()` function. Pass `EXTRACT()` the date parts to isolate.

```
SELECT EXTRACT(YEAR FROM CloseDate) AS CloseDate_Year, EXTRACT(MONTH FROM CloseDate) AS
CloseDate_Month, count(*) AS cnt
FROM "OpportunityFiscalEMTimezoned"
GROUP BY EXTRACT(YEAR FROM CloseDate), EXTRACT(MONTH FROM CloseDate)
LIMIT 10;
```

The results are grouped by year and month parts of the CloseDate field.

CloseDate_Year	CloseDate_Month	cnt
2014	12	1
2015	1	3
2015	2	1
2015	3	2
2015	4	2
2015	5	2

CloseDate_Year	CloseDate_Month	cnt
2015	6	1
2015	8	1
2015	10	1
2015	11	1

Order By Date Part

To order by date part, use `EXTRACT()` on the date field with the `ORDER BY` clause. The query returns the count of rows containing these same values and orders the results by count descending.

```
SELECT EXTRACT(YEAR FROM CloseDate) AS CloseDate_Year, EXTRACT(MONTH FROM CloseDate) AS
CloseDate_Month, COUNT(*) AS cnt
FROM "OpportunityFiscalEMTimezoned"
GROUP BY EXTRACT(YEAR FROM CloseDate), EXTRACT(MONTH FROM CloseDate)
ORDER BY cnt DESC
LIMIT 10;
```

CloseDate_Year	CloseDate_Month	cnt
2015	1	3
2015	3	2
2015	4	2
2015	5	2
2014	12	1
2015	2	1
2015	6	1
2015	8	1
2015	10	1
2015	11	1

Project a Custom Fiscal Date Part

To project a custom fiscal date part, pass custom fiscal date parts to the `EXTRACT()` function.

```
SELECT EXTRACT(FISCAL_YEAR FROM CloseDate) AS CloseDate_Fiscal_Year, EXTRACT(FISCAL_QUARTER
 FROM CloseDate) AS CloseDate_Fiscal_Quarter, EXTRACT(FISCAL_MONTH FROM CloseDate) AS
CloseDate_Fiscal_Month, EXTRACT(FISCAL_WEEK FROM CloseDate) AS CloseDate_Fiscal_Week
FROM "OpportunityFiscalEMTimezoned"
LIMIT 10;
```

CloseDate_Fiscal_Year	CloseDate_Fiscal_Quarter	CloseDate_Fiscal_Month	CloseDate_Fiscal_Week
2015	4	11	49
2015	4	11	49
2015	4	12	49
2015	4	12	53
2015	4	12	54
2016	1	1	5
2016	1	2	9
2016	1	2	9
2016	1	3	13
2016	1	3	13

Day in Week, Month, Quarter or Year Functions

Use the following functions to find the position of a day within a week, month, quarter, or year.

Date position functions take this syntax.

```
SELECT EXTRACT (FunctionName FROM Date) AS DateAlias
FROM dataset;
```

Day of Week
Returns an integer that represents the day of the week for a specific date.

Day of Month
Returns an integer that represents the day of the month for a specific date.

Day of Quarter
Returns an integer that represents the day of the quarter for a specific date. The first quarter of the year begins on January 1.

Day of Year
Returns an integer that represents the day of the year for a specific date.

Day of Week

Returns an integer that represents the day of the week for a specific date.

Example:

```
SELECT OrderDate as "date", EXTRACT (DOW FROM OrderDate) as "day_of_week" FROM
"dates_sample_data";
```

date	day_of_week
2015-01-21 15:30:00	3
2015-01-21 00:00:00	3
2015-01-31 10:00:30	6
2015-02-03 15:30:00	2
2016-01-21 23:59:59	4
2015-10-31 23:59:59	6
2015-12-03 00:00:00	4
2016-01-11 03:30:00	1
2016-01-11 03:30:00	1

Day of Month

Returns an integer that represents the day of the month for a specific date.

Example:

```
SELECT OrderDate as "date", EXTRACT (DAY FROM OrderDate) as "day_of_month" FROM
"dates_sample_data";
```

date	day_of_month
2015-01-21 15:30:00	21
2015-01-21 00:00:00	21
2015-01-31 10:00:30	31
2015-02-03 15:30:00	3
2016-01-21 23:59:59	21
2015-10-31 23:59:59	31
2015-12-03 00:00:00	3
2016-01-11 03:30:00	11
2016-01-11 03:30:00	11

Day of Quarter

Returns an integer that represents the day of the quarter for a specific date. The first quarter of the year begins on January 1.

Example:

```
SELECT OrderDate as "date", EXTRACT (DOQ FROM OrderDate) as "day_of_quarter" FROM
"dates_sample_data";
```

date	day_of_quarter
2015-01-21 15:30:00	21
2015-01-21 00:00:00	21
2015-01-31 10:00:30	31
2015-02-03 15:30:00	34
2016-01-21 23:59:59	21
2015-10-31 23:59:59	31
2015-12-03 00:00:00	64
2016-01-11 03:30:00	11
2016-01-11 03:30:00	11

Day of Year

Returns an integer that represents the day of the year for a specific date.

Example:

```
SELECT OrderDate as "date", EXTRACT (DOY FROM OrderDate) as "day_of_year" FROM
"dates_sample_data";
```

date	day_of_year
2015-01-21 15:30:00	21
2015-01-21 00:00:00	21
2015-01-31 10:00:30	31
2015-02-03 15:30:00	34
2016-01-21 23:59:59	21
2015-10-31 23:59:59	304
2015-12-03 00:00:00	337
2016-01-11 03:30:00	11
2016-01-11 03:30:00	11

First and Last Day in the Week, Month, Quarter or Year Functions

Use the following functions to find the first and last week, month, quarter, or year for both standard and fiscal calendars. These functions accept `DateTime`, `DateOnly`, and legacy input values. They return the same type as the input value.

Date position functions take this syntax.

```
SELECT functionName(Date) AS DateAlias FROM dataset;
```

week_first_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day (Sunday) of the week that contains the specified date.

fiscal_week_first_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day (Monday) of the fiscal week that contains the specified date.

month_first_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy object as input. Returns an object of the same type that corresponds to the first day of the month that contains the specified date.

fiscal_month_first_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day of the fiscal month that contains the specified date.

quarter_first_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day of the quarter that contains the specified date.

fiscal_quarter_first_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day of the fiscal quarter that contains the specified date. By default, the first fiscal quarter is defined as February, March, April. The second quarter is May, June, July. The third is August, September, October. The fourth is November, December, January.

year_first_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day of the year that contains the specified date.

fiscal_year_first_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day of the fiscal year that contains the specified date. By default, the fiscal year begins on February 1.

week_last_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day (Saturday) of the week that contains the specified date.

fiscal_week_last_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day (Sunday) of the fiscal week that contains the specified date.

month_last_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the month that contains the specified date.

fiscal_month_last_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the month that contains the specified date.

quarter_last_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the quarter that contains the specified date.

fiscal_quarter_last_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the fiscal quarter that contains the specified date.

year_last_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the year that contains the specified date.

fiscal_year_last_day(date)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the fiscal year that contains the specified date.

SEE ALSO:

Project a Custom Fiscal Date Part

week_first_day(*date*)

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day (Sunday) of the week that contains the specified date.

Example:

```
SELECT OrderDate as "date", week_first_day(OrderDate) as "week_first_day" FROM
"dates_sample_data";
```

date	week_first_day
2015-01-21 15:30:00	2015-01-18 00:00:00
2015-01-21 00:00:00	2015-01-18 00:00:00
2015-01-31 10:00:30	2015-01-25 00:00:00
2015-02-03 15:30:00	2015-02-01 00:00:00
2016-01-21 23:59:59	2016-01-17 00:00:00
2015-10-31 23:59:59	2015-10-25 00:00:00
2015-12-03 00:00:00	2015-11-29 00:00:00
2016-01-11 03:30:00	2016-01-10 00:00:00
2016-01-11 03:30:00	2016-01-10 00:00:00

`fiscal_week_first_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day (Monday) of the fiscal week that contains the specified date.

Example:

```
SELECT OrderDate as "date", fiscal_week_first_day(OrderDate) as "fiscal_week_first_day"
 FROM "dates_sample_data";
```

date	fiscal_week_first_day
2015-01-21 15:30:00	2015-01-19 00:00:00
2015-01-21 00:00:00	2015-01-19 00:00:00
2015-01-31 10:00:30	2015-01-26 00:00:00
2015-02-03 15:30:00	2015-02-02 00:00:00
2016-01-21 23:59:59	2016-01-18 00:00:00
2015-10-31 23:59:59	2015-10-26 00:00:00
2015-12-03 00:00:00	2015-11-30 00:00:00
2016-01-11 03:30:00	2016-01-11 00:00:00
2016-01-11 03:30:00	2016-01-11 00:00:00

`month_first_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy object as input. Returns an object of the same type that corresponds to the first day of the month that contains the specified date.

Example:

```
SELECT OrderDate as "date", month_first_day(OrderDate) as "month_first_day" FROM
"dates_sample_data";
```

date	month_first_day
2015-01-21 15:30:00	2015-01-01 00:00:00
2015-01-21 00:00:00	2015-01-01 00:00:00
2015-01-31 10:00:30	2015-01-01 00:00:00
2015-02-03 15:30:00	2015-02-01 00:00:00
2016-01-21 23:59:59	2016-01-01 00:00:00
2015-10-31 23:59:59	2015-10-01 00:00:00
2015-12-03 00:00:00	2015-11-01 00:00:00

date	month_first_day
2016-01-11 03:30:00	2016-01-01 00:00:00
2016-01-11 03:30:00	2016-01-01 00:00:00

`fiscal_month_first_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day of the fiscal month that contains the specified date.

Example:

```
SELECT OrderDate as "date", fiscal_month_first_day(OrderDate) as "fiscal_month_first_day"
 FROM "dates_sample_data";
```

date	fiscal_month_first_day
2015-01-21 15:30:00	2015-01-01 00:00:00
2015-01-21 00:00:00	2015-01-01 00:00:00
2015-01-31 10:00:30	2015-01-01 00:00:00
2015-02-03 15:30:00	2015-02-01 00:00:00
2016-01-21 23:59:59	2016-01-01 00:00:00
2015-10-31 23:59:59	2015-10-01 00:00:00
2015-12-03 00:00:00	2015-12-01 00:00:00
2016-01-11 03:30:00	2016-01-01 00:00:00
2016-01-11 03:30:00	2016-01-01 00:00:00

`quarter_first_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day of the quarter that contains the specified date.

Example:

```
SELECT OrderDate as "date", quarter_first_day(OrderDate) as "quarter_first_day" FROM
"dates_sample_data";
```

date	quarter_first_day
2015-01-21 15:30:00	2015-01-01 00:00:00
2015-01-21 00:00:00	2015-01-01 00:00:00

date	quarter_first_day
2015-01-31 10:00:30	2015-01-01 00:00:00
2015-02-03 15:30:00	2015-01-01 00:00:00
2016-01-21 23:59:59	2016-01-01 00:00:00
2015-10-31 23:59:59	2015-10-01 00:00:00
2015-12-03 00:00:00	2015-10-01 00:00:00
2016-01-11 03:30:00	2016-01-01 00:00:00
2016-01-11 03:30:00	2016-01-01 00:00:00

`fiscal_quarter_first_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day of the fiscal quarter that contains the specified date. By default, the first fiscal quarter is defined as February, March, April. The second quarter is May, June, July. The third is August, September, October. The fourth is November, December, January.

Example:

```
SELECT OrderDate as "date", fiscal_quarter_first_day(OrderDate) as
"fiscal_quarter_first_day" FROM "dates_sample_data";
```

date	fiscal_quarter_first_day
2015-01-21 15:30:00	2014-11-01 00:00:00
2015-01-21 00:00:00	2014-11-01 00:00:00
2015-01-31 10:00:30	2014-11-01 00:00:00
2015-02-03 15:30:00	2015-02-01 00:00:00
2016-01-21 23:59:59	2015-11-01 00:00:00
2015-10-31 23:59:59	2015-08-01 00:00:00
2015-12-03 00:00:00	2015-11-01 00:00:00
2016-01-11 03:30:00	2015-11-01 00:00:00
2016-01-11 03:30:00	2015-11-01 00:00:00

`year_first_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day of the year that contains the specified date.

Example:

```
SELECT OrderDate as "date", year_first_day(OrderDate) as "year_first_day" FROM
"dates_sample_data";
```

date	year_first_day
2015-01-21 15:30:00	2015-01-01 00:00:00
2015-01-21 00:00:00	2015-01-01 00:00:00
2015-01-31 10:00:30	2015-01-01 00:00:00
2015-02-03 15:30:00	2015-01-01 00:00:00
2016-01-21 23:59:59	2016-01-01 00:00:00
2015-10-31 23:59:59	2015-01-01 00:00:00
2015-12-03 00:00:00	2015-01-01 00:00:00
2016-01-11 03:30:00	2016-01-01 00:00:00
2016-01-11 03:30:00	2016-01-01 00:00:00

`fiscal_year_first_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the first day of the fiscal year that contains the specified date. By default, the fiscal year begins on February 1.

Example:

```
SELECT OrderDate as "date", fiscal_year_first_day(OrderDate) as "fiscal_year_first_day"
 FROM "dates_sample_data";
```

date	year_first_day
2015-01-21 15:30:00	2014-02-01 00:00:00
2015-01-21 00:00:00	2014-02-01 00:00:00
2015-01-31 10:00:30	2014-02-01 00:00:00
2015-02-03 15:30:00	2015-02-01 00:00:00
2016-01-21 23:59:59	2015-02-01 00:00:00
2015-10-31 23:59:59	2015-02-01 00:00:00
2015-12-03 00:00:00	2015-02-01 00:00:00
2016-01-11 03:30:00	2015-02-01 00:00:00
2016-01-11 03:30:00	2015-02-01 00:00:00

`week_last_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day (Saturday) of the week that contains the specified date.

Example:

```
SELECT OrderDate as "date", week_last_day(OrderDate) as "week_last_day" FROM
"dates_sample_data";
```

date	week_last_day
2015-01-21 15:30:00	2015-01-24 00:00:00
2015-01-21 00:00:00	2015-01-24 00:00:00
2015-01-31 10:00:30	2015-01-31 00:00:00
2015-02-03 15:30:00	2015-02-07 00:00:00
2016-01-21 23:59:59	2016-01-23 00:00:00
2015-10-31 23:59:59	2015-10-31 00:00:00
2015-12-03 00:00:00	2015-12-05 00:00:00
2016-01-11 03:30:00	2016-01-16 00:00:00
2016-01-11 03:30:00	2016-01-16 00:00:00

`fiscal_week_last_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day (Sunday) of the fiscal week that contains the specified date.

Example:

```
SELECT OrderDate as "date", fiscal_week_last_day(OrderDate) as "fiscal_week_last_day"
 FROM "dates_sample_data";
```

date	fiscal_week_last_day
2015-01-21 15:30:00	2015-01-25 00:00:00
2015-01-21 00:00:00	2015-01-25 00:00:00
2015-01-31 10:00:30	2015-02-01 00:00:00
2015-02-03 15:30:00	2015-02-08 00:00:00
2016-01-21 23:59:59	2016-01-24 00:00:00
2015-10-31 23:59:59	2015-11-01 00:00:00
2015-12-03 00:00:00	2015-12-06 00:00:00

date	fiscal_week_last_day
2016-01-11 03:30:00	2016-01-17 00:00:00
2016-01-11 03:30:00	2016-01-17 00:00:00

`month_last_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the month that contains the specified date.

Example:

```
SELECT OrderDate as "date", month_last_day(OrderDate) as "month_last_day" FROM
"dates_sample_data";
```

date	month_last_day
2015-01-21 15:30:00	2015-01-31 00:00:00
2015-01-21 00:00:00	2015-01-31 00:00:00
2015-01-31 10:00:30	2015-01-31 00:00:00
2015-02-03 15:30:00	2015-02-28 00:00:00
2016-01-21 23:59:59	2016-01-31 00:00:00
2015-10-31 23:59:59	2015-10-31 00:00:00
2015-12-03 00:00:00	2015-12-31 00:00:00
2016-01-11 03:30:00	2016-01-31 00:00:00
2016-01-11 03:30:00	2016-01-31 00:00:00

`fiscal_month_last_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the month that contains the specified date.

Example:

```
SELECT OrderDate as "date", fiscal_month_last_day(OrderDate) as "fiscal_month_last_day"
 FROM "dates_sample_data";
```

date	fiscal_month_last_day
2015-01-21 15:30:00	2015-01-31 00:00:00
2015-01-21 00:00:00	2015-01-31 00:00:00

date	fiscal_month_last_day
2015-01-31 10:00:30	2015-01-31 00:00:00
2015-02-03 15:30:00	2015-02-28 00:00:00
2016-01-21 23:59:59	2016-01-31 00:00:00
2015-10-31 23:59:59	2015-10-31 00:00:00
2015-12-03 00:00:00	2015-12-31 00:00:00
2016-01-11 03:30:00	2016-01-31 00:00:00
2016-01-11 03:30:00	2016-01-31 00:00:00

`quarter_last_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the quarter that contains the specified date.

Example:

```
SELECT OrderDate as "date", quarter_last_day(OrderDate) as "quarter_last_day" FROM
"dates_sample_data";
```

date	quarter_last_day
2015-01-21 15:30:00	2015-03-31 00:00:00
2015-01-21 00:00:00	2015-03-31 00:00:00
2015-01-31 10:00:30	2015-03-31 00:00:00
2015-02-03 15:30:00	2015-03-31 00:00:00
2016-01-21 23:59:59	2016-03-31 00:00:00
2015-10-31 23:59:59	2015-12-31 00:00:00
2015-12-03 00:00:00	2015-12-31 00:00:00
2016-01-11 03:30:00	2016-03-31 00:00:00
2016-01-11 03:30:00	2016-03-31 00:00:00

`fiscal_quarter_last_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the fiscal quarter that contains the specified date.

Example:

```
SELECT OrderDate as "date", fiscal_quarter_last_day(OrderDate) as
"fiscal_quarter_last_day" FROM "dates_sample_data";
```

date	fiscal_quarter_last_day
2015-01-21 15:30:00	2015-01-31 00:00:00
2015-01-21 00:00:00	2015-01-31 00:00:00
2015-01-31 10:00:30	2015-01-31 00:00:00
2015-02-03 15:30:00	2015-04-30 00:00:00
2016-01-21 23:59:59	2016-01-31 00:00:00
2015-10-31 23:59:59	2015-10-31 00:00:00
2015-12-03 00:00:00	2015-01-31 00:00:00
2016-01-11 03:30:00	2016-01-31 00:00:00
2016-01-11 03:30:00	2016-01-31 00:00:00

`year_last_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the year that contains the specified date.

Example:

```
SELECT OrderDate as "date", year_last_day(OrderDate) as "year_last_day" FROM
"dates_sample_data";
```

date	year_last_day
2015-01-21 15:30:00	2015-12-31 00:00:00
2015-01-21 00:00:00	2015-12-31 00:00:00
2015-01-31 10:00:30	2015-12-31 00:00:00
2015-02-03 15:30:00	2015-12-31 00:00:00
2016-01-21 23:59:59	2016-12-31 00:00:00
2015-10-31 23:59:59	2015-12-31 00:00:00
2015-12-03 00:00:00	2015-12-31 00:00:00
2016-01-11 03:30:00	2016-12-31 00:00:00
2016-01-11 03:30:00	2016-12-31 00:00:00

`fiscal_year_last_day(`*`date`*`)`

Accepts a `DateTime`, `DateOnly`, or legacy Date object as input. Returns an object of the same type that corresponds to the last day of the fiscal year that contains the specified date.

Example:

```
SELECT OrderDate as "date", fiscal_year_last_day(OrderDate) as "fiscal_year_last_day"
 FROM "dates_sample_data";
```

date	fiscal_year_last_day
2015-01-21 15:30:00	2015-01-31 00:00:00
2015-01-21 00:00:00	2015-01-31 00:00:00
2015-01-31 10:00:30	2015-01-31 00:00:00
2015-02-03 15:30:00	2016-01-31 00:00:00
2016-01-21 23:59:59	2016-01-31 00:00:00
2015-10-31 23:59:59	2016-01-31 00:00:00
2015-12-03 00:00:00	2016-01-31 00:00:00
2016-01-11 03:30:00	2016-01-31 00:00:00
2016-01-11 03:30:00	2016-01-31 00:00:00

AGGREGATE FUNCTIONS

Aggregate functions perform operations across columns.

CRM Analytics supports the following aggregate functions.

avg()
: Takes the average of row values in a given column.

count()
: Returns the number of rows that match a specified condition in a column.

min()
: Returns the lowest row value in a numeric column.

max()
: Returns the highest row value in a numeric column.

sum()
: Returns the sum of all rows values in a numeric column.

stddev_pop()
: Returns the population standard deviation of the values in a field. Accepts measure fields as input. This function does not accept expressions.

stddev_samp()
: Returns the standard deviation of values in a field. Accepts measure fields as input. This function does not accept expressions.

var_pop()
: Returns the population variance of the values in a field. Accepts measure fields as input. This function does not accept expressions.

var_samp()
: Returns the sample variance of the values in a field. Accepts measure fields as input. This function does not accept expressions.

regr_intercept()
: Returns the y-intercept value of a regression line.

regr_slope()
: Returns the slope of a regression line between two numerical fields.

regr_r2()
: Returns the R-squared, or goodness-of-fit value for a regression.

percentile_cont
: Calculates a percentile based on a continuous distribution of the column value.

percentile_disc
: Returns the value corresponding to a given percentile.

`avg()`

Takes the average of row values in a given column.

`avg()` follows this syntax.

```
SELECT avg(column_name)
FROM dataset
WHERE condition;
```

`count()`

Returns the number of rows that match a specified condition in a column.

`count()` follows this syntax.

```
SELECT count(column_name)
FROM dataset
WHERE condition;
```

`min()`

Returns the lowest row value in a numeric column.

`min()` follows this syntax.

```
SELECT min(column_name)
FROM dataset
WHERE condition;
```

`max()`

Returns the highest row value in a numeric column.

`max()` follows this syntax.

```
SELECT max(column_name)
FROM dataset
WHERE condition;
```

`sum()`

Returns the sum of all rows values in a numeric column.

`sum()` follows this syntax.

```
SELECT sum(column_name)
FROM dataset
WHERE condition;
```

stddev_pop()

Returns the population standard deviation of the values in a field. Accepts measure fields as input. This function does not accept expressions.

`stddev_pop()` follows this syntax.

```
SELECT stddev_pop(field1, field2)
FROM dataset;
```

stddev_samp()

Returns the standard deviation of values in a field. Accepts measure fields as input. This function does not accept expressions.

`stddev_samp()` follows this syntax.

```
SELECT stddev_pop(field1, field2)
FROM dataset;
```

var_pop()

Returns the population variance of the values in a field. Accepts measure fields as input. This function does not accept expressions.

`var_pop()` follows this syntax.

```
SELECT var_pop(field1, field2)
FROM dataset;
```

var_samp()

Returns the sample variance of the values in a field. Accepts measure fields as input. This function does not accept expressions.

`var_samp()` follows this syntax.

```
SELECT var_samp(field1, field2)
FROM dataset;
```

regr_intercept()

Returns the y-intercept value of a regression line.

`regr_intercept()` follows this syntax.

```
SELECT regr_intercept(field1, field2)
FROM dataset;
```

`regr_slope()`

Returns the slope of a regression line between two numerical fields.

`regr_slope()` follows this syntax.

```
SELECT regr_slope(field1, field2)
FROM dataset;
```

`regr_r2()`

Returns the R-squared, or goodness-of-fit value for a regression.

`regr_r2()` follows this syntax.

```
SELECT regr_r2(field1, field2)
FROM dataset;
```

`percentile_cont`

Calculates a percentile based on a continuous distribution of the column value.

`percentile_cont()` follows this syntax.

```
SELECT percentile_cont(field1, field2)
FROM dataset;
```

`percentile_disc`

Returns the value corresponding to a given percentile.

`percentile_disc()` follows this syntax.

```
SELECT percentile_disc(field1, field2)
FROM dataset;
```

MATH FUNCTIONS

SQL for CRM Analytics supports the following math functions.

abs(n)
: Returns the absolute number of `n` as a numeric value. `n` can be any real numeric value in the range of -1.797e308 <= `n` <= 1.797e308.

acos(n)
: Returns the arccosine value of radians value ***n***. ***n*** can be any real number in the range of -1 <= ***n*** <= 1. If `null` is passed as an argument, `acos()` returns `null`.

asin(n)
: Returns the arcsine value of radians value ***n***. ***n*** can be any real number in the range of -1 <= ***n*** <= 1. If `null` is passed as an argument, `sin()` returns `null`.

atan(n)
: Returns the arctan value of radians value ***n***. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `atan()` returns `null`.

ceil(n), ceiling(n)
: Returns the nearest integer of equal or greater value to `n`. `n` can be any real numeric value in the range of -1.797e308 <= `n` <= 1.797e308.

cos(n)
: Returns the cosine value of radians value ***n***. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `cos()` returns `null`.

degrees(n)
: Returns the degrees value of ***n*** radians. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `radians()` returns `null`.

exp(n)
: Returns the value of Euler's number e raised to the power of `n`, where e = 2.71828183… The smallest value for `n` that does not result in 0 is 3e-324. `n` can be any real numeric value in the range of -1.797e308 <= `n` <= 700.

floor(n)
: Returns the nearest integer of equal or lesser value to n. n can be any real numeric value in the range of -1.797e308<= n <= 1.797e308.

ln(n)
: Returns the base e (Euler's number) logarithm of a number `n`. The value `n` can be any positive, non-zero numeric value in the range 0 < `n` <= 1.797e308.

log(m,n)
: Returns the natural logarithm (base m) of a number n. The values m and n can be any positive, non-zero numeric value in the range 0 < m, n <= 1.797e308 and m ≠ 1.

log10(n)
: Returns the base 10 logarithm of a number `n`. The value `n` can be any positive, non-zero numeric value in the range 0 < `n` <= 1.797e308.

pi()
Returns the value of constant π, where π=3.14159.

power(m,n)
Returns *m* raised to the *n*th power. *m*, *n* can be any numeric value in the range of -1.797e308 <= *m*, *n* <= 1.797e308. Returns null if *m* = 0 and *n* < 0.

radians(n)
Returns the radians value of ***n*** degrees. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `radians()` returns `null`.

round(n[,m])
Returns the value of *n* rounded to *m* decimal places. *m* can be negative, in which case the function returns *n* rounded to -*m* places to the left of the decimal point. If *m* is omitted, it returns *n* rounded to the nearest integer. For tie-breaking, it follows round half way from zero convention. *n* can be any real numeric value in the range of -1.797e308 <= *n* <= 1.797e308. *m* can be an integer value between -15 and 15, inclusive.

sign(n)
Returns 1 if the numeric value, ***n*** is positive. It returns -1 if the ***n*** is negative, and 0 if ***n*** is 0. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308.

sin(n)
Returns the sine value of radians value ***n***. ***n***can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `sin()` returns `null`.

sqrt(n)
Returns the square root of a number *n*. The value *n* can be any non-negative numeric value in the range of 0 <= *n* <= 1.797e308.

tan(n)
Returns the tan value of radians value ***n***. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `tan()` returns `null`.

trunc(n[,m])
Returns the value of the numeric expression *n* truncated to *m* decimal places. *m* can be negative, in which case the function returns *n* truncated to -*m* places to the left of the decimal point. If *m* is omitted, it returns *n* truncated to the integer place. *n* can be any real numeric value in the range of -1.797e308 <= *n* <= 1.797e308. *m* can be an integer value between -15 and 15 inclusive.

abs(*n*)

Returns the absolute number of *n* as a numeric value. *n* can be any real numeric value in the range of -1.797e308 <= *n* <= 1.797e308.

`abs(n)` takes the following syntax.

```
SELECT abs(column_name) as AliasName
FROM dataset;
```

Example: This example takes the absolute value of the Profit field.

```
SELECT abs(Profit) as absProfit FROM Superstore
LIMIT 1;
```

absProfit
41.9136

`acos(n)`

Returns the arccosine value of radians value ***n***. ***n*** can be any real number in the range of -1 <= ***n*** <= 1. If `null` is passed as an argument, `acos()` returns `null`.

`acos(n)` takes the following syntax.

```
SELECT ACOS(n) as Alias
FROM dataset;
```

Example: This example takes the arccosine of 35 degrees.

```
SELECT ACOS(RADIANS(90)) as acosValue
FROM "Opportunity"
LIMIT 1;
```

acosValue
0.913643

`asin(n)`

Returns the arcsine value of radians value ***n***. ***n*** can be any real number in the range of -1 <= ***n*** <= 1. If `null` is passed as an argument, `sin()` returns `null`.

`asin(n)` takes the following syntax.

```
SELECT ASIN(n) as Alias
FROM dataset;
```

Example: This example takes the arcsine of 35 degrees.

```
SELECT ASIN(RADIANS(35)) as asinValue
FROM "Opportunity"
LIMIT 1;
```

asinValue
0.60746

`atan(n)`

Returns the arctan value of radians value ***n***. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `atan()` returns `null`.

`atan(n)` takes the following syntax.

```
SELECT ATAN(n) as Alias
FROM dataset;
```

Example: This example takes the arctan of 90 degrees.

```
SELECT ATAN(RADIANS(90)) as arctanValue
FROM "Opportunity"
LIMIT 1;
```

arctanValue
1.00388

`ceil(n)`, `ceiling(n)`

Returns the nearest integer of equal or greater value to *n*. *n* can be any real numeric value in the range of -1.797e308 <= *n* <= 1.797e308.

`ceil(n)` takes the following syntax.

```
SELECT ceil(column_name) as aliasName
FROM dataset;
```

Example: This example takes the ceiling of the Profit field.

```
SELECT ceil(Profit) as ceilProfit
FROM Superstore
LIMIT 1;
```

ceilProfit
42

`cos(n)`

Returns the cosine value of radians value ***n***. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `cos()` returns `null`.

`cos(n)` takes the following syntax.

```
SELECT COS(n) as Alias
FROM dataset;
```

Example: This example takes the cosine of 90 degrees.

```
SELECT COS(RADIANS(90)) as cosValue
FROM "Opportunity"
LIMIT 1;
```

cosValue
6.12323e-17

`degrees(n)`

Returns the degrees value of ***n*** radians. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `radians()` returns `null`.

`degrees(n)` takes the following syntax.

```
SELECT DEGREES(n) as Alias
FROM dataset;
```

Example: This example returns the degrees of 1.57079 radians.

```
SELECT DEGREES(1.57079) as degreesValue
FROM "Opportunity"
LIMIT 1;
```

degreesValue
90

`exp(n)`

Returns the value of Euler's number e raised to the power of *n*, where e = 2.71828183… The smallest value for *n* that does not result in 0 is 3e-324. *n* can be any real numeric value in the range of -1.797e308 <= *n* <= 700.

`exp(n)` takes the following syntax.

```
SELECT exp(n) as AliasName
FROM dataset;
```

Example: This example returns the value of e raised to the 5th power.

```
SELECT exp(5) as expExample FROM Superstore
LIMIT 1;
```

expExample
148.413

`floor(n)`

Returns the nearest integer of equal or lesser value to n. n can be any real numeric value in the range of -1.797e308<= n <= 1.797e308.

`floor(n)` takes the following syntax.

```
SELECT floor(column_name) as AliasName
FROM dataset;
```

Example: This example returns values in the Discount field rounded down to the nearest integer. In the first three rows, the Discount value is 0. The fourth and fifth values are 0.45 and 0.2, respectively, which are rounded to 0.

```
SELECT floor(Discount) as floorDiscount
FROM Superstore
LIMIT 5;
```

floorDiscount
0
0
0
0
0

`ln(n)`

Returns the base e (Euler's number) logarithm of a number *n*. The value *n* can be any positive, non-zero numeric value in the range 0 < *n* <= 1.797e308.

`ln(n)` takes the following syntax.

```
SELECT ln(numericValue) as AliasName
FROM dataset;
```

Example: This example returns the base e logarithm for the number 8.

```
SELECT ln(8.0) as LnExample FROM Superstore
LIMIT 1;
```

LnExample
2.079

`log(m,n)`

Returns the natural logarithm (base m) of a number n. The values m and n can be any positive, non-zero numeric value in the range 0 < m, n <= 1.797e308 and m ≠ 1.

`log(mn)` takes the following syntax.

```
SELECT log(baseNumber, numericValue) as AliasName
FROM dataset;
```

Example: This example returns the base 2 logarithm for the number 8.

```
SELECT log(2.0,8.0) as natlLogExample FROM Superstore
LIMIT 1;
```

natlLogExample
3.0

`log10(n)`

Returns the base 10 logarithm of a number *n*. The value *n* can be any positive, non-zero numeric value in the range 0 < *n* <= 1.797e308.

`log10(n)` takes the following syntax.

```
SELECT log10(numeric_value) as AliasName
FROM dataset;
```

Example: This example returns the base 10 logarithm for the number 5.

```
SELECT log10(5) as logExample FROM Superstore
LIMIT 1;
```

logExample
0.698

`pi()`

Returns the value of constant π, where π=3.14159.

`PI()` takes the following syntax.

```
SELECT PI() as Alias
FROM dataset;
```

Example:

```
SELECT PI() as piValue
FROM "Opportunity"
LIMIT 1;
```

piValue
3.14159

`power(m,n)`

Returns *m* raised to the *n*th power. *m*, *n* can be any numeric value in the range of -1.797e308 <= *m*, *n* <= 1.797e308. Returns null if *m* = 0 and *n* < 0.

- If *m* = 0, *n* must be a non-negative value.
- If *m* < 0, *n* must be an integer value.
- The result of power(*m*, *n*) must be within the range expressed by a float64 number.

`power(m,n)` takes the following syntax.

```
SELECT power(m,n) as AliasName
FROM dataset;
```

Example: This example returns 2 raised to the 5th power.

```
SELECT power(2, 5) as powerExample FROM Superstore
LIMIT 1;
```

powerExampe
32.0

`radians(n)`

Returns the radians value of ***n*** degrees. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `radians()` returns `null`.

`radians(n)` takes the following syntax.

```
SELECT RADIANS(n) as Alias
FROM dataset;
```

Example: This example returns the radians of 90 degrees.

```
SELECT RADIANS(90) as radiansValue
FROM "Opportunity"
LIMIT 1;
```

radiansValue
1.57079

round(*n*[,*m*])

Returns the value of *n* rounded to *m* decimal places. *m* can be negative, in which case the function returns *n* rounded to -*m* places to the left of the decimal point. If *m* is omitted, it returns *n* rounded to the nearest integer. For tie-breaking, it follows round half way from zero convention. *n* can be any real numeric value in the range of -1.797e308 <= *n* <= 1.797e308. *m* can be an integer value between -15 and 15, inclusive.

`round(n[,m])` takes the following syntax.

```
SELECT round(n[,m]) as AliasName
FROM dataset;
```

Example: This example returns 47.385 rounded to the first decimal place.

```
SELECT round(47.385, 1) as roundExample FROM Superstore
LIMIT 1;
```

roundExample
47.4

sign(*n*)

Returns 1 if the numeric value, ***n*** is positive. It returns -1 if the ***n*** is negative, and 0 if ***n*** is 0. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308.

`SIGN(n)` takes the following syntax.

```
SELECT SIGN(n) as Alias
FROM dataset;
```

Example: This example returns the `sign` of -12.

```
SELECT SIGN(-12) as signValue
FROM "Opportunity"
LIMIT 1;
```

signValue
-1

`sin(`***n***`)`

Returns the sine value of radians value ***n***. ***n***can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `sin()` returns `null`.

`sin(`***n***`)` takes the following syntax.

```
SELECT SIN(n) as Alias
FROM dataset;
```

Example: This example takes the sine of 90 degrees.

```
SELECT SIN(RADIANS(90)) as sinValue
FROM "Opportunity"
LIMIT 1;
```

sinValue
1

`sqrt(`*n*`)`

Returns the square root of a number *n*. The value *n* can be any non-negative numeric value in the range of 0 <= *n* <= 1.797e308.

`sqrt(`***n***`)` takes the following syntax.

```
SELECT sqrt(n) as AliasName
FROM dataset;
```

Example: This example returns the square root of 64.

```
SELECT sqrt(64.0) as SqrtExample FROM Superstore
LIMIT 1;
```

SqrtExample
8.0

`tan(`*n*`)`

Returns the tan value of radians value ***n***. ***n*** can be any real number in the range of -1e308 <= ***n*** <= 1e308. If `null` is passed as an argument, `tan()` returns `null`.

`tan(`***n***`)` takes the following syntax.

```
SELECT TAN(n) as Alias
FROM dataset;
```

Example: This example takes the tan of 90 degrees.

```
SELECT TAN(RADIANS(90)) as tanValue
FROM "Opportunity"
LIMIT 1;
```

tanValue
163312

trunc(*n*[,*m*])

Returns the value of the numeric expression *n* truncated to *m* decimal places. *m* can be negative, in which case the function returns *n* truncated to -*m* places to the left of the decimal point. If *m* is omitted, it returns *n* truncated to the integer place. *n* can be any real numeric value in the range of -1.797e308 <= *n* <= 1.797e308. *m* can be an integer value between -15 and 15 inclusive.

`trunc(n[,m])` takes the following syntax.

```
SELECT trunc(n[,m]) as AliasName
FROM dataset;
```

Example: This example returns 47.385 truncated to the second decimal place.

```
SELECT trunc(47.385, 2) as truncExample FROM Superstore
LIMIT 1;
```

truncExample
47.38

STRING FUNCTIONS AND OPERATORS

SQL for CRM Analytics supports the following string functions and operators.

ascii(char)
: Returns UTF-8 numeric value of the specified character. Returns `null` if ***n*** is `null`.

chr(int)
: Returns the UTF-8 character for integer `n`. Returns `null` if ***n*** is `null`.

char_length(str)
: Returns the number of characters in a given string.

ends_with(source_str, search_str)
: Returns a Boolean indicating whether a string ends with the search string.

index_of(source_str, search_str, [position,occurrence])
: Returns a boolean indicating whether a string ends with the search string.

lower(str)
: Returns a copy of string *`str`* with all cased characters converted to lowercase.

mv_to_string(multivalue_column_name, [delimeter])
: Converts multivalue fields to string fields.

position(search_str IN source_str)
: Returns an integer that indicates the first occurrence of a substring in a given string. If the substring is not found, the function returns 0.

replace(str, old_str, new_str)
: Replaces all occurrences in string *`str`* of a substring *`old_str`* with a new substring *`new_str`*. Returns a new string.

starts_with(source_str, search_str)
: Returns a Boolean indicating whether a string begins with the search string.

substring(str FROM start FOR length)
: Returns a substring from string *`str`* that begins with the character at the *`start`* position.

trim(LEADING | TRAILING | BOTH, chars, str)
: Removes leading or trailing characters from a string. If no characters are specified, the function removes blank spaces.

upper(str)
: Returns a copy of string *`str`* with all cased characters converted to uppercase.

ascii(*char*)

Returns UTF-8 numeric value of the specified character. Returns `null` if ***n*** is `null`.

Example: This example returns the ASCII value for the letter “a.”

```
SELECT ASCII('C') as "asciiValue"
FROM "Superstore"
LIMIT 1;
```

asciiValue
67

`chr(int)`

Returns the UTF-8 character for integer `n`. Returns `null` if ***n*** is `null`.

Example: This example returns the character value of 67.

```
SELECT CHR(67) as "charValue"
FROM "Superstore"
LIMIT 1;
```

charValue
C

`char_length(str)`

Returns the number of characters in a given string.

Example: This example returns the number of characters in the first five entries in the City column.

```
SELECT City, CHAR_LENGTH(City) as "CityLen"
FROM "Superstore"
GROUP BY City
LIMIT 5;
```

City	CityLen
Aberdeen	8
Abilene	7
Akron	5
Albuquerque	11
Alexandria	10

`ends_with(source_str, search_str)`

Returns a Boolean indicating whether a string ends with the search string.

`ENDS_WITH()` follows this syntax.

`source_str`
: The string to be searched.

`search_str`
: The string to search for within the source string.

Example: This example returns a Boolean that confirms whether the value in the `City` field ends with "ale."

```
SELECT City, ENDS_WITH(City, "ale") as "endValue"
FROM "Superstore"
LIMIT 5;
```

City	endValue
Henderson	false
Henderson	false
Los Angeles	false
Fort Lauderdale	true
Fort Lauderdale	true

`index_of(source_str, search_str, [position,occurrence])`

Returns a boolean indicating whether a string ends with the search string.

`INDEX_OF()` follows this syntax.

`source_str`
: The string to be searched.

`search_str`
: The string to search for within the source string.

`position`
: Optional. The index from which to begin searching the string. The default *`position`* is set to 1.

`occurrence`
: Optional. If there's more than one instance of the searched string, you can specify occurrence to choose which instance to return. The default *`occurrence`* is set to 1.

The function returns 0 if *`search_str`* isn't found.

Example: This example returns the index of the second occurrence of the letter “e” in the first five cities in the dataset.

```
SELECT City, INDEX_OF(City, 'e', 1, 2) as "indexValue"
FROM "Superstore"
LIMIT 5;
```

City	indexValue
Henderson	5
Henderson	5
Los Angeles	10
Fort Lauderdale	15
Fort Lauderdale	15

`lower(`*`str`*`)`

Returns a copy of string `str` with all cased characters converted to lowercase.

Example: This example returns the first five entries in the City column in lowercase.

```
SELECT lower(City) as "lCity"
FROM "Superstore"
GROUP BY City
LIMIT 5;
```

lCity
aberdeen
abilene
akron
albuquerque
alexandria

`mv_to_string(`***`multivalue_column_name, [delimeter]`***`)`

Converts multivalue fields to string fields.

`MV_TO_STRING()` takes this syntax.

`MV_TO_STRING(`***`multivalue_column_name, delimeter`***`)`

multivalue_column_name
Name of the multivalue field to be converted to a string.

delimiter

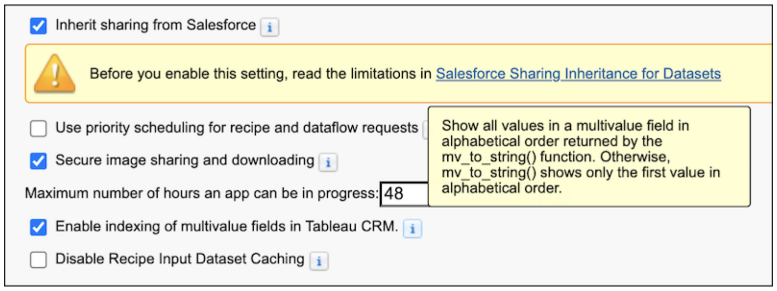
Optional. The characters used to delimit values in the converted string. Maximum length is 2 characters.

Returns an alphabetically sorted, delimited string representation of a multivalue field. The default delimiter is a comma followed by a space (`, `).

You can't use `MV_TO_STRING()` in a `WHERE` or `GROUP BY` clause. Use it in the `SELECT` statement only. If you run `MV_TO_STRING()` on single-value dimensions or on unindexed multivalue dimensions, the function returns a single value.

Note: To enable multivalue fields, you must select the **Enable indexing of multivalue fields** in CRM Analytics preference in Setup. If you run `MV_TO_STRING()` without the preference selected, the function returns the first value in the first field only.

1. From Settings, in the Quick Find box, enter *`Analytics`*, and then select **Settings** from the list of Analytics options.
2. In Settings, select **Enable indexing of multivalue fields in CRM Analytics**.



[✓] Inherit sharing from Salesforce

Before you enable this setting, read the limitations in Salesforce Sharing Inheritance for Datasets

[ ] Use priority scheduling for recipe and dataflow requests

[✓] Secure image sharing and downloading

Maximum number of hours an app can be in progress: 48

[✓] Enable indexing of multivalue fields in Tableau CRM.

[ ] Disable Recipe Input Dataset Caching

Show all values in a multivalue field in alphabetical order returned by the mv_to_string() function. Otherwise, mv_to_string() shows only the first value in alphabetical order.

Simple Example

This query converts the multivalue field `flight_attendants` to a string and returns values that contain the name "maria."

```
SELECT MV_TO_STRING(flight_attendants) as Flight_Attendants
FROM "FlightsWithNullDim"
WHERE "flight_attendants" IN ('maria')
```

Flight_Attendants
kate, maria, mark, martin, sara
maria, sarah
kate, maria, mark, martin, sara
maria, sarah
kate, maria, mark, martin, sara
maria, sarah

Example with Custom Delimeter

This query returns all values of the `flight_classes` field and delimits them with "`;;`".

```
SELECT MV_TO_STRING(flight_classes, ';;') as Flight_Classes
FROM "FlightsWithNullDim"
LIMIT 10;
```

Flight_Classes
business;;economy
business;;economy;;first
business;;economy
business;;economy
business;;economy
business;;economy;;first
business;;economy
business;;economy
business;;economy
business;;economy;;first

Example with Grouping

The custom delimeter example returns two result values: business;;economy and business;;economy;;first. Let's group by the `flight_classes` field and display the counts for each of these values. Since `MV_TO_STRING()` throws an error if included in a `GROUP BY` statement, we can nest the `SELECT` statement containing `MV_TO_STRING()` as a subquery and group the results in the outer query.

```
SELECT Flight_Classes, count() as cnt
FROM (
    SELECT MV_TO_STRING(flight_classes, ';;') as Flight_Classes
    FROM "FlightsWithNullDim"
)
GROUP BY Flight_Classes;
```

Flight_Classes	cnt
business;;economy	18
business;;economy;;first	6

`position(`*`search_str`* `IN` *`source_str`*`)`

Returns an integer that indicates the first occurrence of a substring in a given string. If the substring is not found, the function returns 0.

Example: This example returns the position of the substring "der" in the City field.

```
SELECT City, POSITION('der' IN City) as "Pos"
FROM "Superstore"
LIMIT 5;
```

City	Pos
Henderson	4
Henderson	4
Los Angeles	0
Fort Lauderdale	9
Fort Lauderdale	9

`replace(`*`str, old_str, new_str`*`)`

Replaces all occurrences in string *`str`* of a substring *`old_str`* with a new substring *`new_str`*. Returns a new string.

Example: This example replaces instances of 'Al' with 'AL' in the City column. The query returns the first five entries.

```
SELECT replace(City, 'Al', 'AL') as "City"
From "Superstore"
GROUP BY City
LIMIT 5;
```

City
Aberdeen
Abilene
Akron
ALbuquerque
ALexandria

`starts_with(`*`source_str, search_str`*`)`

Returns a Boolean indicating whether a string begins with the search string.

`STARTS_WITH()` follows this syntax.

`source_str`
    The string to be searched.

`search_str`
    The string to search for within the source string.

Example: This example returns a Boolean that confirms whether the value in the `City` field begins with "Hen."

```
SELECT City, STARTS_WITH(City, "Hen") as "startValue"
FROM "Superstore"
LIMIT 5;
```

City	startValue
Henderson	true
Henderson	true
Los Angeles	false
Fort Lauderdale	false
Fort Lauderdale	false

substring(*str* FROM *start* FOR *length*)

Returns a substring from string *str* that begins with the character at the *start* position.

Note: The substring function isn't supported in Analytics Studio. An error occurs when the function is used in the UI. The function is available using the Query API. For more information, see Query CRM Analytics Data with the Query API.

start
Required. The starting position of the substring. This is the first character of the substring. The value can be positive or negative. If positive, the position is taken from the beginning of the string. If negative, the position is taken from the end of the string.

length
Optional. The length of the substring, beginning at the *start* position. If no value is provided, `substring()` extracts characters from the *start* position through the end of the string.

Example: This example extracts the first three letters from each City field.

```
SELECT City, SUBSTRING(City FROM 1 FOR 3) as CityCode FROM "Superstore"
GROUP BY City LIMIT 5;
```

City	CityCode
Aberdeen	Abe
Abilene	Abi
Akron	Akr
Albuquerque	Alb
Alexandria	Ale

`trim(LEADING | TRAILING | BOTH, *chars*, *str*)`

Removes leading or trailing characters from a string. If no characters are specified, the function removes blank spaces.

Note: The trim function isn’t supported in Analytics Studio. An error occurs when the function is used in the UI. The function is available using the Query API. For more information, see Query CRM Analytics Data with the Query API.

`LEADING`
Optional. Specify to remove characters from the beginning of the string.

`TRAILING`
Optional. Specify to remove characters from the end of the string.

`BOTH`
Optional. Specify to remove characters from both the start and end of the string.

If `LEADING`, `TRAILING`, or `BOTH` aren't specified, `trim()` defaults to `BOTH`.

Example: This example shows all variations of the `trim()` function for removing the characters "A", "b", and "n" from the beginning, end, or both sides of the City field string.

```
SELECT City, Trim('Abn' FROM City) as TrimCity, Trim(LEADING 'Abn' FROM City) as
LTrimCity, Trim(TRAILING 'Abn' FROM City) as RTrimCity, Trim(BOTH 'Abn' FROM City) as
 BothTrimCity
FROM "Superstore"
GROUP BY City
LIMIT 5;
```

City	TrimCity	LTrimCity	RTrimCity	BothTrimCity
Aberdeen	erdee	erdeen	Aberdee	erdee
Abilene	ilene	ilene	Abilene	ilene
Akron	kro	kron	Akro	kro
Albuquerque	lbuquerque	lbuquerque	Albuquerque	lbuquerque
Alexandria	lexandria	lexandria	Alexandria	lexandria

`upper(*str*)`

Returns a copy of string `str` with all cased characters converted to uppercase.

Example: This example returns the first five entries in the City column in uppercase.

```
SELECT upper(City) as upperCity FROM "Superstore"
GROUP BY City
LIMIT 5;
```

upperCity
ABERDEEN

upperCity
ABILENE
AKRON
ALBUQUERQUE
ALEXANDRIA

SQL FOR ANALYTICS RELEASE NOTES

Use the Salesforce Release Notes to learn about the most recent updates and changes to SQL for Analytics.

For a list of all current developer changes, including SQL for Analytics, see CRM Analytics in the Salesforce Release Notes.

Note: If the Analytics Development section in the Salesforce Release Notes isn't present, there aren't any updates for that release.