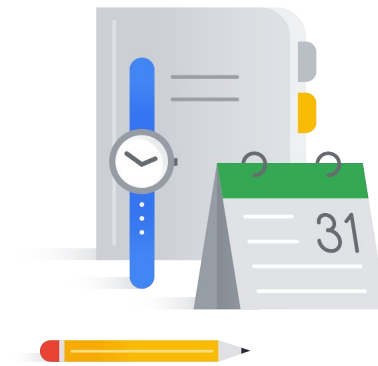
Google Cloud

Advanced Features and Partitioning your Queries and Tables for Advanced Insights

Agenda

01 Advanced Functions
(Statistical, Analytic, User-defined)

02 Date-Partitioned Tables
- Demo: Creating Partitioned Tables
- Lab: Creating Date-Partitioned Tables in BigQuery



In this module we continue our journey with SQL by delving into a few more advanced concepts like statistical approximation functions and user-defined functions. Then we will explore how to break apart really complex data questions into step-by-step modular pieces in SQL with common table expressions and subqueries.

Let’s start by revisiting the SQL functions we’ve covered so far.

01

Advanced Functions (Statistical, Analytic, User-Defined)

Use the right function for the right job

- String Manipulation Functions - FORMAT( )
- Aggregation Functions - SUM( ) COUNT( ) AVG( ) MAX( )
- Data Type Conversion Functions - CAST( )
- Date Functions - PARSE_DATETIME( )
- Statistical Functions
- Analytic Functions
- User-defined Functions

BigQuery Functions Reference

Aggregation = perform calculations over a set of values (like SUM, COUNT, MIN, MAX)

String Manipulation = make every letter uppercase, pull the left 5 characters, format

Statistical = standard deviation, variance, and more

Analytic = perform aggregations over a subset or window of data

User-defined = write your own function recipe in SQL or even Javascript

Use **Analytic Window Functions** for advanced analysis

- Standard aggregations
  - `SUM`, `AVG`, `MIN`, `MAX`, `COUNT`, `etc.`
- Navigation functions
  - `LEAD()` – Returns the value of a row ***n*** rows ahead of the current row
  - `LAG()` – Returns the value of a row ***n*** rows behind the current row
  - `NTH_VALUE()` – Returns the value of the ***n***th value in the window
- Ranking and numbering functions
  - `CUME_DIST()` – Returns the cumulative distribution of a value in a group
  - `DENSE_RANK()` – Returns the integer rank of a value in a group
  - `ROW_NUMBER()` – Returns the current row number of the query result
  - `RANK()` – Returns the integer rank of a value in a group of values
  - `PERCENT_RANK()` – Returns the rank of the current row, relative to the other rows in the partition

Proprietary + Confidential

Example: **RANK() Function** for aggregating over groups of rows

Get the oldest ranking employee *by each department*

firstname	department	startdate
Andrew	1	1/23/1999
Jacob	1	7/11/1990
Daniel	2	6/24/2004
Anna	1	10/7/2001
Pierre	1	2/22/2009
Ruth	2	6/6/1998
Anthony	1	11/29/1995
Isabella	2	9/28/1997
Jose	2	3/17/2013

`PARTITION BY department`

firstname	department	startdate
Andrew	1	1/23/1999
Jacob	1	7/11/1990
Anna	1	10/7/2001
Pierre	1	2/22/2009
Anthony	1	11/29/1995

firstname	department	startdate
Ruth	2	6/6/1998
Daniel	2	6/24/2004
Jose	2	3/17/2013
Isabella	2	9/28/1997

Sometimes called a "window" function

`ORDER BY startdate`

firstname	department	startdate
Jacob	1	7/11/1990
Anthony	1	11/29/1995
Andrew	1	1/23/1999
Anna	1	10/7/2001
Pierre	1	2/22/2009

firstname	department	startdate
Isabella	2	9/28/1997
Daniel	2	6/24/2004
Jose	2	3/17/2013
Ruth	2	6/6/2013

`RANK()`

firstname	department	startdate	rank
Jacob	1	7/11/1990	1
Anthony	1	11/29/1995	2
Andrew	1	1/23/1999	3
Anna	1	10/7/2001	4
Pierre	1	2/22/2009	5

firstname	department	startdate	rank
Isabella	2	9/28/1997	1
Daniel	2	6/24/2004	2
Jose	2	3/17/2013	3
Ruth	2	6/6/2013	4

More SQL Analytic Functions

Google Cloud

Get the oldest ranking employee by each department

https://cloud.google.com/bigquery/docs/reference/standard-sql/functions-and-operators#supported-functions
RANK( )

In databases, an analytic function is a function that computes aggregate values over a group of rows. Unlike aggregate functions, which return a single aggregate value for a group of rows, analytic functions return a single value for each row by computing the function over a group of input rows.

Analytic functions are a powerful mechanism for succinctly representing complex analytic operations, and they enable efficient evaluations that otherwise would involve expensive self-JOINs or computation outside the SQL query.

Analytic functions are also called "(analytic) window functions" in the SQL standard and some commercial databases. This is because an analytic function is evaluated over a group of rows, referred to as a window or window frame. In some other databases, they may be referred to as Online Analytical Processing (OLAP) functions.

Proprietary + Confidential

Example: **RANK() Function** for aggregating over groups of rows

```
SELECT firstname, department, startdate,
  RANK() OVER ( PARTITION BY department ORDER BY startdate ) AS rank
FROM Employees;
```

Logical partitioning or "windowing" of rows BY department

Google Cloud

Option to do demo with IRS dataset:

```
#standardSQL
# Largest employer per U.S. state per 2015 filing
WITH employer_per_state AS (
SELECT
  ein,
  name,
  state,
  noemplyeesw3cnt AS number_of_employees,
  RANK() OVER (PARTITION BY state ORDER BY noemplyeesw3cnt DESC ) AS
rank
FROM
  `bigquery-public-data.irs_990.irs_990_2015`
JOIN
  `bigquery-public-data.irs_990.irs_990_ein`
USING(ein)
GROUP BY 1,2,3,4 #remove duplicates
)

# Get the top employer per state and order highest to lowest states
SELECT *
FROM employer_per_state
WHERE rank = 1
```

```
ORDER BY number_of_employees DESC;
```

Components of a User-Defined Function (UDF)

- **CREATE FUNCTION.**
  Creates a new function. A function can contain zero or more named_parameters
- **RETURNS [data_type].**
  Specifies the data type that the function returns.
- **Language [language].**
  Specifies the language for the function.
- **AS [external_code].**
  Specifies the code that the function runs.

```
CREATE FUNCTION d2i_demo.nlp_compromise_people(str STRING)
RETURNS ARRAY<STRING> LANGUAGE js AS '''
  return nlp(str).people().out('topk').map(x=>x.normal)
'''
OPTIONS (
library="gs://cloud-training/datatoinsights/assets/compromise.
min.11.14.0.js");

SELECT name, COUNT(*) c
FROM ( SELECT d2i_demo.nlp_compromise_people(title) names
  FROM `d2i_demo.reddit_posts`
  WHERE subreddit = 'movies'
), UNNEST(names) name
WHERE name LIKE '% %' GROUP BY 1
ORDER BY 2 DESC LIMIT 10
```

Row	name	c
1	captain marvel	198
2	will smith	118
3	ary digital	109
4	star wars	84

The example code in the slide is a Persistent UDF, one that can be used by any BigQuery user via a simple SQL query.

https://cloud.google.com/bigquery/docs/reference/standard-sql/user-defined-functions

Pitall: User-Defined Functions hurt performance

- Use native SQL functions whenever possible
- Concurrent rate limits:
  - for non-UDF queries: 50
  - for UDF-queries: 6

BigQuery Quota Policy

Amount of data UDF outputs per input row should be <=5 MB

Each user can run 6 concurrent JavaScript UDF queries per project

Native code JavaScript functions aren't supported

JavaScript handles only the most significant 32 bits

A query job can have a maximum of 50 JavaScript UDF resources

Each inline code blob is limited to maximum size of 32 KB

Each external code resource limited to maximum size of 1 MB

Proprietary + Confidential

User-Defined Function limitations

Persistent & Temporary

- DOM objects such as Window, Document, and Node (and functions that require them) are not supported.
- JavaScript functions that rely on native code can fail.
- Bitwise operations in JavaScript are limited to 32 bits.

Persistent

- Each dataset can only contain one persistent UDF with the same name.
- Persistent UDFs must be qualified with the name of the dataset.

Temporary

- The "function_name" cannot contain periods.
- Views and persistent UDFs cannot reference temporary UDFs.

Google Cloud

There are a variety of limitations to User-Defined functions. These can dramatically impact potential use cases and must be considered when deciding to use a UDF.

https://cloud.google.com/bigquery/docs/reference/standard-sql/user-defined-functions#limitations

02

Date-Partitioned Tables

In this module we continue our journey with SQL by delving into a few more advanced concepts like statistical approximation functions and user-defined functions. Then we will explore how to break apart really complex data questions into step-by-step modular pieces in SQL with common table expressions and subqueries.

Let’s start by revisiting the SQL functions we’ve covered so far.

Proprietary + Confidential

A common challenge with 1M+ record tables is querying the entire table for just last week's metrics

All ecommerce site visits

```
SELECT
  COUNT(transactionId) AS total_transactions,
  date
FROM
  `data-to-insights.ecommerce.all_sessions`
WHERE
  transactionId IS NOT NULL
  AND PARSE_DATE("%Y%m%d", date) >= '2018-01-01'
GROUP BY date
ORDER BY date DESC
```

Google Cloud

A partitioned table is a table that is divided into segments, called partitions, that make it easier to manage and query your data. By dividing a large table into smaller partitions, you can improve query performance, and control costs by reducing the number of bytes read by a query.

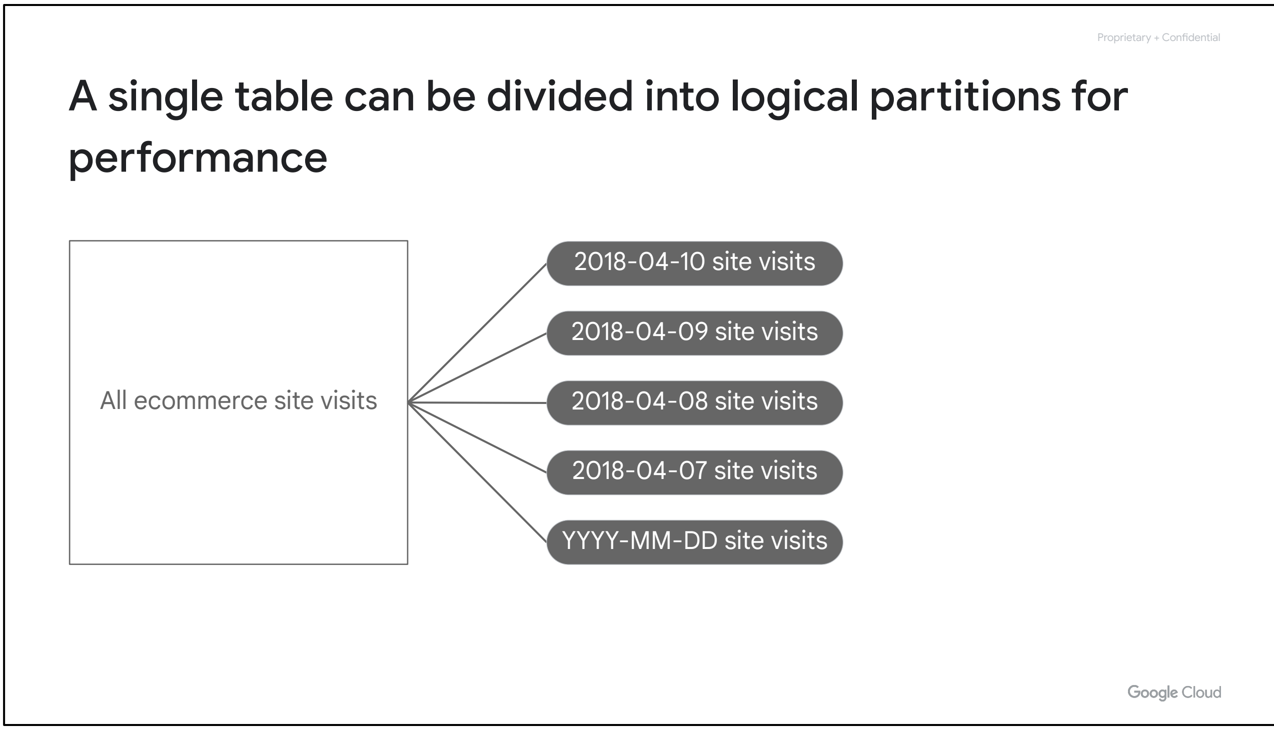
To satisfy the WHERE condition, our query must look at **every** date value to see if it's after '2018-01-01'

All ecommerce site visits

```
SELECT
  COUNT(transactionId) AS total_transactions,
  date
FROM
  `data-to-insights.ecommerce.all_sessions`
WHERE
  transactionId IS NOT NULL
  AND PARSE_DATE("%Y%m%d", date) >= '2018-01-01'
GROUP BY date
ORDER BY date DESC
```

This query will process 205.9 MB when run.

A partitioned table is a table that is divided into segments, called partitions, that make it easier to manage and query your data. By dividing a large table into smaller partitions, you can improve query performance, and control costs by reducing the number of bytes read by a query.



A partitioned table is a table that is divided into segments, called partitions, that make it easier to manage and query your data. By dividing a large table into smaller partitions, you can improve query performance, and control costs by reducing the number of bytes read by a query.

A single table can be divided into logical partitions for performance

```
CREATE OR REPLACE TABLE ecommerce.partitions
 PARTITION BY date_formatted
 OPTIONS(
   description="a table partitioned by date"
 ) AS

SELECT
  COUNT(transactionId) AS total_transactions,
  PARSE_DATE("%Y%m%d", date) AS date_formatted
FROM
  `data-to-insights.ecommerce.all_sessions`
WHERE
  transactionId IS NOT NULL
GROUP BY date
```


Instead of scanning the entire dataset and filtering on a date field like we did in the earlier queries, we will now set up a date-partitioned table. This will allow us to completely ignore scanning records in certain partitions if they are irrelevant to our query.

Now the exact same query will first reference the partition list before processing any data!

2018-04-10 site visits

2018-04-09 site visits

2018-04-08 site visits

2018-04-07 site visits

YYYY-MM-DD site visits

```
SELECT
  total_transactions,
  date_formatted
FROM
  `data-to-insights.ecommerce.partitions`
WHERE date_formatted >= '2018-01-01'
ORDER BY date_formatted DESC
```

This query will process 0 B when run.

Why 0?

A partitioned table is a table that is divided into segments, called partitions, that make it easier to manage and query your data. By dividing a large table into smaller partitions, you can improve query performance, and control costs by reducing the number of bytes read by a query.

Our query knew there wasn't any transactions after 2017-08-01 in our dataset just by looking at our existing partitions

- 2018-04-10 site visits
- 2018-04-09 site visits
- 2018-04-08 site visits
- 2018-04-07 site visits
- YYYY-MM-DD site visits

```
SELECT
  total_transactions,
  date_formatted
FROM
  `data-to-insights.ecommerce.partitions`
ORDER BY
  date_formatted DESC
```

Row	total_transactions	date_formatted
1	279	2017-08-01
2	321	2017-07-31
3	139	2017-07-30
4	111	2017-07-29
5	249	2017-07-28
6	230	2017-07-27

The latest partition of data is from ***2017***-08-01 so no 2018 would ever show up

A partitioned table is a table that is divided into segments, called partitions, that make it easier to manage and query your data. By dividing a large table into smaller partitions, you can improve query performance, and control costs by reducing the number of bytes read by a query.

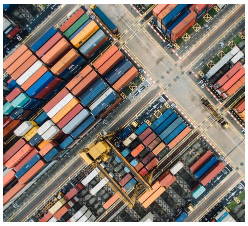
Demo

Creating Partitioned Tables

Review: Date-partitioned tables

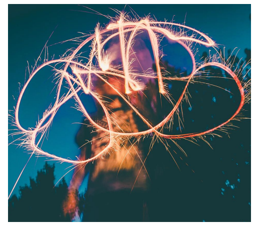
Refer to
https://github.com/GoogleCloudPlatform/training-data-analyst/tree/master/courses/data-to-insights/demos/date-partitioned-tables.sql

Summary: Answer more complex questions with advanced SQL

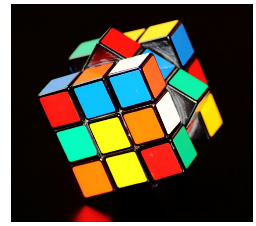


Consider using approximation functions for really large datasets.

Operate over sub-groups of rows with analytical window functions.



User-defined functions add sophistication at the expense of performance.



Use partitions to break apart tables logically for performance.

To wrap up, we finished covering SQL functions which included some pretty neat ones that allow you to statistically estimate with great accuracy across huge datasets. It's your option here whether to trade processing time for 100% accuracy.

Next we covered an example where we wanted to break apart a single table into sub groups of rows and perform a ranking inside each sub-group by using analytical window functions.

After that we introduced UDFs or user-defined functions which can be written in SQL or Javascript. Remember the caveat that query performance is impacted.

Let's practice this in our next lab.

Lab Intro

Creating Date-Partitioned Tables
in BigQuery

Lab objectives

01 Creating tables with date partitions

02 View data processed with a partitioned table

03 Creating an auto-expiring partitioned table using NOAA Weather data

04 Your turn: Create a Partitioned Table