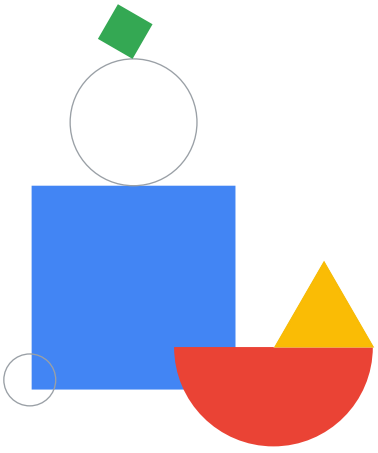
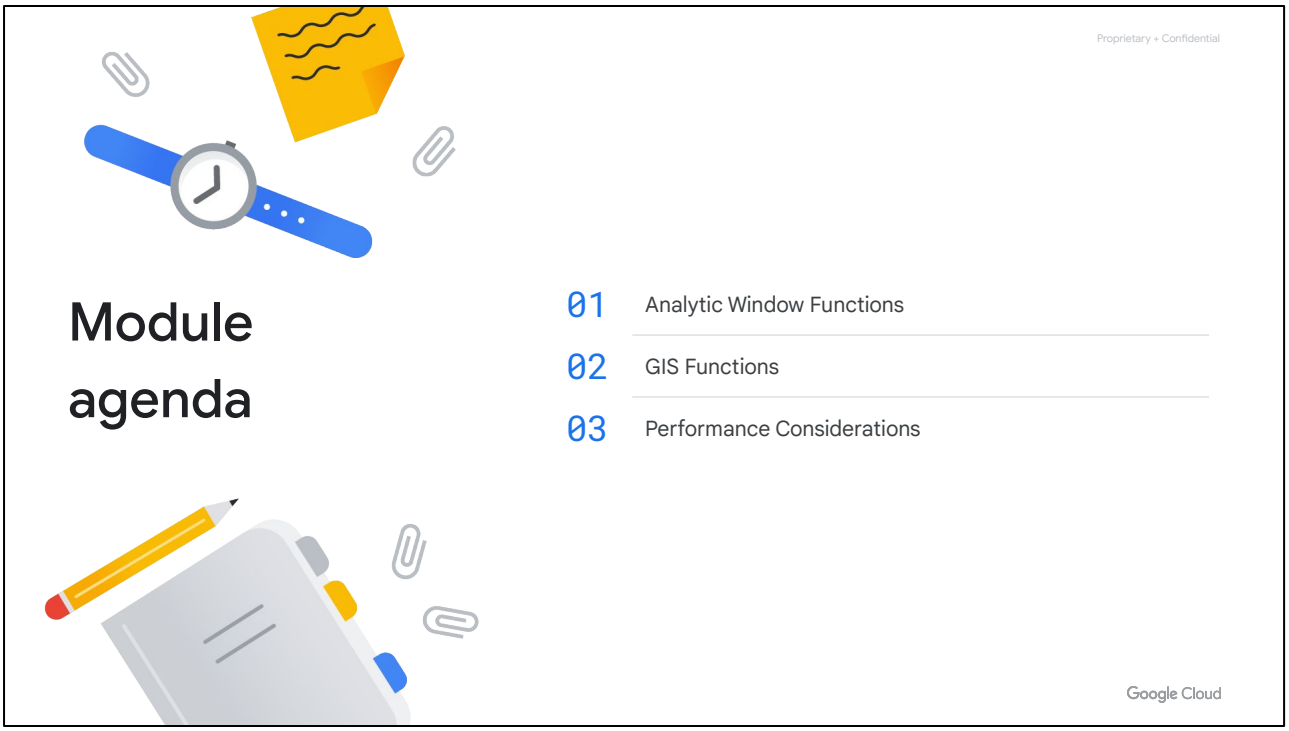
Google Cloud

Advanced BigQuery Functionality and Performance



In this final module of the **Building Resilient Streaming Systems on Google Cloud** course, you'll learn about some of the advanced features of BigQuery.



In this module, you'll learn about analytic window functions and the use of WITH clauses to make complex queries more manageable.

We'll introduce you to some of the GIS functions built into BigQuery, and finally, we'll share best practices to consider for BigQuery performance.

Analytic Window Functions

Let’s start by looking at how to use analytic window functions for advanced analysis.

Use analytic window functions for advanced analysis

- Standard aggregations
- Navigation functions
- Ranking and Numbering functions

BigQuery, like other databases, has built-in functions to allow for rapid calculation of results.

These include window functions to support advanced analysis.

Three groups of functions exist:

- Standard aggregations,
- Navigation functions, and
- Ranking and Numbering Functions.

Example: The COUNT function (self-explanatory)

```
SELECT
  start_station_name,
  end_station_name,
  APPROX_QUANTILES(duration, 10)[OFFSET(5)] AS
typical_duration,
  COUNT(*) AS num_trips
FROM
  `bigquery-public-data`.london_bicycles.cycle_hire
GROUP BY
  start_station_name,
  end_station_name
```

Query results SAVE RESULTS EXPLORE WITH DATA STUDIO

Query complete (13.5 sec elapsed, 1.5 GB processed)

Job information | Results | JSON | Execution details

Row	start_station_name	end_station_name	typical_duration	num_trips
1	Borough High Street, The Borough	Bell Street , Marylebone	4320	3
2	William IV Street, Strand	Little Brook Green, Brook Green	4500	3
3	Baker Street, Marylebone	George Place Mews, Marylebone	180	95
4	Waterloo Station 2, Waterloo	Westbourne Grove, Bayswater	3240	7
5	Imperial Wharf Station	Upcerne Road, West Chelsea	180	94
6	Kennington Road , Vauxhall	Westminster Bridge Road, Elephant & Castle	180	38
7	Whiston Road, Haggerston	Pitfield Street North,Hoxton	180	99
8	Gloucester Street, Pimlico	Rampayne Street, Pimlico	180	330
9	Harrington Square 1, Camden Town	Drummond Street , Euston	180	76

The Count function is a frequently used and self-explanatory function.

Some other “standard” aggregation functions

- SUM
- AVG
- MIN
- MAX
- BIT_AND
- BIT_OR
- BIT_XOR
- COUNTIF
- LOGICAL_OR
- LOGICAL_AND

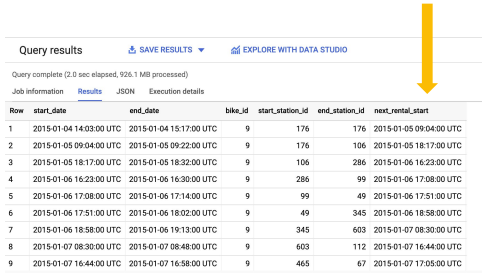
https://cloud.google.com/bigquery/docs/reference/standard-sql/aggregate_functions

Other standard aggregation functions are listed here with more detail and how to properly use them available in the documentation.

[https://cloud.google.com/bigquery/docs/reference/standard-sql/aggregate_functions]

Example: The LEAD function returns the value of a row n rows ahead of the current row

```
SELECT
  start_date,
  end_date,
  bike_id,
  start_station_id,
  end_station_id,
  LEAD(start_date, 1) OVER(PARTITION BY bike_id ORDER BY start_date ASC ) AS
next_rental_start
FROM
  `bigquery-public-data`.london_bicycles.cycle_hire
WHERE
  bike_id = 9
```



Query results SAVE RESULTS EXPLORE WITH DATA STUDIO

Query complete (2.0 sec elapsed, 926.1 MB processed)

Job information | Results | JSON | Execution details

Row	start_date	end_date	bike_id	start_station_id	end_station_id	next_rental_start
1	2015-01-04 14:03:00 UTC	2015-01-04 15:17:00 UTC	9	176	176	2015-01-05 09:04:00 UTC
2	2015-01-05 09:04:00 UTC	2015-01-05 09:22:00 UTC	9	176	106	2015-01-05 18:17:00 UTC
3	2015-01-05 18:17:00 UTC	2015-01-05 18:32:00 UTC	9	106	286	2015-01-06 16:23:00 UTC
4	2015-01-06 16:23:00 UTC	2015-01-06 16:30:00 UTC	9	286	99	2015-01-06 17:08:00 UTC
5	2015-01-06 17:08:00 UTC	2015-01-06 17:14:00 UTC	9	99	49	2015-01-06 17:51:00 UTC
6	2015-01-06 17:51:00 UTC	2015-01-06 18:02:00 UTC	9	49	345	2015-01-06 18:58:00 UTC
7	2015-01-06 18:58:00 UTC	2015-01-06 19:13:00 UTC	9	345	603	2015-01-07 08:30:00 UTC
8	2015-01-07 08:30:00 UTC	2015-01-07 08:48:00 UTC	9	603	112	2015-01-07 16:44:00 UTC
9	2015-01-07 16:44:00 UTC	2015-01-07 16:58:00 UTC	9	465	67	2015-01-07 17:05:00 UTC

The LEAD function will return a value for a subsequent row in relation to the current row. In the example, the next bike rental time is listed along with the current rental row.

Some other navigation functions

- NTH_VALUE
- LAG
- FIRST_VALUE
- LAST_VALUE

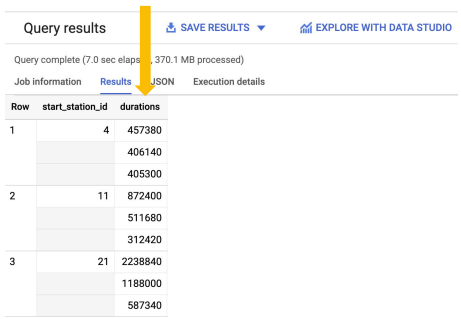
https://cloud.google.com/bigquery/docs/reference/standard-sql/navigation_functions

Navigation functions generally compute some value expression over a different row in the window frame from the current row. Here are a few commonly used Navigation functions.

https://cloud.google.com/bigquery/docs/reference/standard-sql/navigation_functions

Example: The RANK function returns the integer rank of a value in a group of values

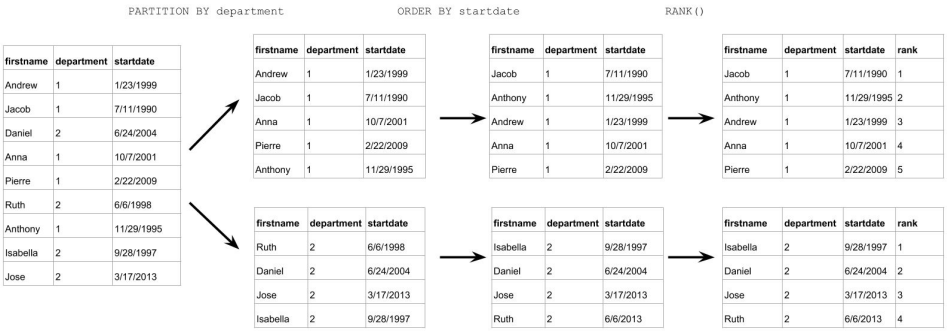
```
WITH
  longest_trips AS (
  SELECT
    start_station_id,
    duration,
    RANK() OVER(PARTITION BY start_station_id ORDER BY duration DESC) AS
nth_longest
  FROM
    `bigquery-public-data`.london_bicycles.cycle_hire )
SELECT
  start_station_id,
  ARRAY_AGG(duration
  ORDER BY
    nth_longest
  LIMIT
    3) AS durations
FROM
  longest_trips
GROUP BY
  start_station_id
```



Row	start_station_id	durations
1	4	457380
		406140
		405300
2	11	872400
		511680
		312420
3	21	2238840
		1188000
		587340

Rank returns the ordinal (1-based) rank of each row within the ordered partition. In the example, each station's duration is returned in ranked order descending - the longest duration returned first..

Example: RANK() function for aggregating over groups of rows



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

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

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

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

Get the oldest ranking employee ***by each department***

https://cloud.google.com/bigquery/docs/reference/standard-sql/functions-and-operators#analytic-functions

This example shows the ranking of employees by tenure using the start date within each department.

First the rows are partitioned by department, then ordered by start date, and then finally ranked.

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;
```


This is the SQL code used to perform the RANK operation from the ‘ranking of employees by tenure’ example on the previous slide.

Some other ranking and numbering functions

- CUME_DIST
- DENSE_RANK
- ROW_NUMBER
- PERCENT_RANK

https://cloud.google.com/bigquery/docs/reference/standard-sql/numbering_functions

Addition Ranking and Numbering exist for specific use cases. The examples seen here are frequently used when determining relationships between the rows of data rather than by an external measurement.

Use WITH clauses and subqueries to modularize

```
WITH

  # 2015 filings joined with organization details
  irs_990_2015_ein AS (
  SELECT *
  FROM
    `bigquery-public-data.irs_990.irs_990_2015`
  JOIN
    `bigquery-public-data.irs_990.irs_990_ein` USING (ein)
    ),

  # duplicate EINs in organization details
  duplicates AS (
  SELECT
    ein AS ein,
    COUNT(ein) AS ein_count
  FROM
    irs_990_2015_ein
  GROUP BY
    ein
  HAVING
    ein_count > 1
    )

# return results to store in a permanent table
SELECT
  irs_990.ein AS ein,
  irs_990.name AS name,
  irs_990.noemployeesw3cnt AS num_employees,
  irs_990.grsrcptspublicuse AS gross_receipts
  # more fields ommited for brevity
FROM irs_990_2015_ein AS irs_990
LEFT JOIN duplicates
ON
  irs_990.ein=duplicates.ein
WHERE
  # filter out duplicate records
  duplicates.ein IS NULL
```

- WITH is simply a named subquery (or Common Table Expression)
- Acts as a temporary table
- Breaks up complex queries
- Chain together multiple subqueries in a single WITH
- You can reference other subqueries in future subqueries

https://cloud.google.com/bigquery/docs/reference/standard-sql/query-syntax#with-clause

WITH clauses are instances of a named subquery in BigQuery.

WITH clauses are an easy way to isolate SQL operations and make complex queries more manageable.

GIS Functions

BigQuery has many built-in Geographic Information System, or GIS, features. You'll learn about some of them in this lesson.

BigQuery has built-in GIS functionality

Example: Can we find the zip codes best served by the New York Citibike system by looking for the number of stations within 1 km of each zip code that have at least 30 bikes?

```
SELECT
  z.zip_code,
  COUNT(*) AS num_stations
FROM
  `bigquery-public-data.new_york_citibike.citibike_stations` AS s,
  `bigquery-public-data.geo_us_boundaries.zip_codes` AS z
WHERE
  ST_DWithin(z.zcta_geom,
    ST_GeogPoint(s.longitude, s.latitude),
    1000) -- 1km
  AND num_bikes_available > 30
GROUP BY
  z.zip_code
ORDER BY
  num_stations DESC
```

Query results SAVE RESULTS EXPLORE WITH DATA STUDIO

Query complete (1.1 sec elapsed, 128.3 MB processed)

Job information | Results | JSON | Execution details

Row	zip_code	num_stations
1	10003	21
2	10002	19
3	10026	17
4	10012	16
5	10029	16

Google Cloud

In the example shown, a zip code is used to determine how many bike stations are within 1 kilometer of the zip code and have at least 30 bikes available.

ST_GeogPoint and **ST_DWithin** are used together to pinpoint the stations of interest. ST simply means spatial type.

Use ST_DWITHIN to check if two locations objects are within some distance

- ST_DWithin(geography_1, geography_2, distance)

in meters

```
SELECT
  z.zip_code,
  COUNT(*) AS num_stations
FROM
    `bigquery-public-data.new_york_citibike.citibike_stations` AS s,
    `bigquery-public-data.geo_us_boundaries.zip_codes` AS z
WHERE
  ST_DWithin(z.zip_code_geom,
    ST_GeogPoint(s.longitude, s.latitude),
    1000) -- 1km
  AND num_bikes_available > 30
GROUP BY
  z.zip_code
ORDER BY
  num_stations DESC
```


ST_DWithin is used in conjunction with the Geospatial boundaries of US zip codes,

The latitude and longitude of the bike stations joined together in ST_GeogPoint to create a geospatial object, and the value of 1000 for 1000 meters, which is 1 kilometer as the distance between the objects - Zip Code boundary and Station point.

This will return only those within 1 kilometer.

Represent longitude and latitude points as Well Known Text (WKT) using the function ST_GeogPoint

Queries in BigQuery are much more efficient if geographic data is stored as geography types rather than as basics.

```
SELECT
  z.zip_code,
  COUNT(*) AS num_stations
FROM
    `bigquery-public-data.new_york_citibike.citibike_stations` AS s,
    `bigquery-public-data.geo_us_boundaries.zip_codes` AS z
WHERE
  ST_DWithin(z.zcta_geom,
    ST_GeogPoint(s.longitude, s.latitude),
    1000) -- 1km
  AND num_bikes_available > 30
GROUP BY
  z.zip_code
ORDER BY
  num_stations DESC
```

If your data is stored in JSON, use **ST_GeogFromGeoJSON**

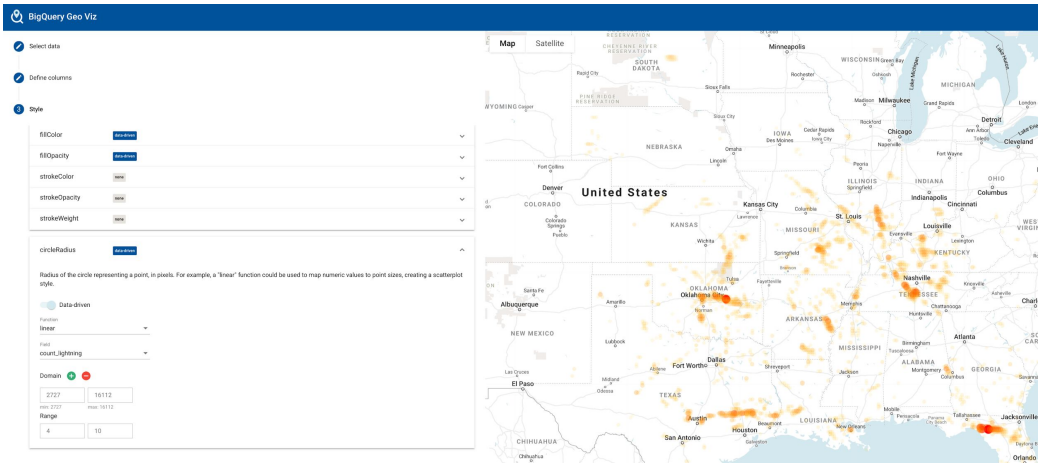
```
SELECT ST_GeogFromGeoJSON('{"type": "Point", "coordinates": [-73.967416,40.756014]}')
```

Google Cloud

ST_GeogPoint creates a geospatial object in Well Known Text (or WKT) from values provided within the database. In this case we use latitude and longitude.

If the latitude and longitude are provided in JSON format, the function ST_GeogFromGeoJSON can be used to generate a geospatial object.

Visualization with BigQuery Geo Viz



Google Cloud

To allow for quick testing of Geospatial data, Google Cloud provides the lightweight BigQuery GeoViz application.

This application will allow rendering of GIS data with minimal configuration.

Represent points with ST_GeogPoint

- Represented as **WKT (Well Known Text)**

```
SELECT
  id,
  longitude,
  latitude,
  ST_GeogPoint(longitude, latitude) AS location
FROM
  `bigquery-public-data.london_bicycles.cycle_stations`
LIMIT
  100
```

id	longitude	latitude	location
171	-0.186753859	51.4916156	POINT(-0.186753859 51.4916156)
165	-0.183716959	51.517932	POINT(-0.183716959 51.517932)
261	-0.191351186	51.5134891	POINT(-0.191351186 51.5134891)
131	-0.136792671	51.53300545	POINT(-0.136792671 51.53300545)
467	-0.030556	51.523538	POINT(-0.030556 51.523538)
43	-0.157183945	51.52026	POINT(-0.157183945 51.52026)
212	-0.199004026	51.50658458	POINT(-0.199004026 51.50658458)
517	-0.033085	51.532513	POINT(-0.033085 51.532513)
704	-0.202802098	51.45682071	POINT(-0.202802098 51.45682071)
721	-0.026262677	51.53603947	POINT(-0.026262677 51.53603947)

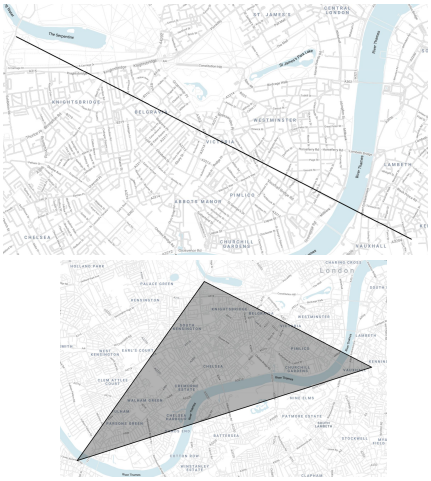
As mentioned earlier, ST_GeogPoint is used to create a Geospatial object from relevant data.

The image shows the exact coordinates of the ID values on a map of London.

Represent regions with ST_MakeLine and ST_MakePolygon

```
WITH stations AS (
SELECT
  (SELECT ST_GeogPoint(longitude, latitude) FROM `bigquery-public-da
      AS loc300,
  (SELECT ST_GeogPoint(longitude, latitude) FROM `bigquery-public-da
      AS loc302,
  (SELECT ST_GeogPoint(longitude, latitude) FROM `bigquery-public-da
      AS loc305
)
SELECT
  ST_MakeLine(loc300, loc305) AS seg1,
  ST_MakePolygon(ST_MakeLine([loc300, loc305, loc302])) AS poly
FROM
  stations
```

seg1	poly
LINESTRING(-0.17306032 51.505014, -0.115853961 51.48677988)	POLYGON((-0.216573 51.466907, -0.115853961 51.48677988, -0.17306032 ...



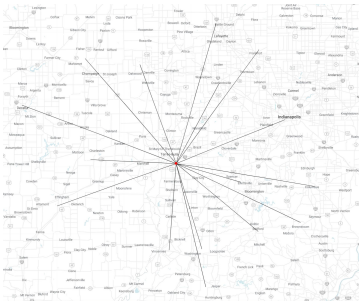
ST_MakeLine and ST_MakePolygon are two additional geospatial functions that can be used to overlay information on a map to help highlight relationships in the data.

Are locations within some distance?

- ST_DWithin

```
SELECT
  m.name AS city,
  m.int_point AS city_coords,
  ST_MakeLine(
    n.int_point,
    m.int_point) AS segs
FROM
  `bigquery-public-data.geo_us_boundaries.us_msa` AS n,
  `bigquery-public-data.geo_us_boundaries.us_msa` AS m
WHERE
  n.name='Terre Haute, IN'
  AND ST_DWithin(
    n.int_point,
    m.int_point,
    1.5e5) --150km
```

city	city_coords	segs
Crawfordsville, IN	POINT(-86.8927145 40.0402962)	LINESTRING(-87.3470958 39.392389, -86.892714...
Decatur, IL	POINT(-88.9615288 39.8602372)	LINESTRING(-87.3470958 39.392389, -88.961528...
Washington, IN	POINT(-87.076944 38.696089)	LINESTRING(-87.3470958 39.392389, -87.076944 ...
Indianapolis-Carmel-Anderson, IN	POINT(-86.2045408 39.7449323)	LINESTRING(-87.3470958 39.392389, -86.204540...
Lafayette-West Lafayette, IN	POINT(-86.9304747 40.5147171)	LINESTRING(-87.3470958 39.392389, -86.930474...
Bloomington, IN	POINT(-86.6717544 39.2417362)	LINESTRING(-87.3470958 39.392389, -86.671754...
Seymour, IN	POINT(-86.0425161 38.9119571)	LINESTRING(-87.3470958 39.392389, -86.042516...
Frankfort, IN	POINT(-86.4775665 40.309944)	LINESTRING(-87.3470958 39.392389, -86.477566...
Charleston-Mattoon, IL	POINT(-88.2422121 39.4231628)	LINESTRING(-87.3470958 39.392389, -88.242212...
Jasper, IN	POINT(-87.0359535 38.3841559)	LINESTRING(-87.3470958 39.392389, -87.035953...



As mentioned in our earlier example, ST_DWithin can be used to determine the relative location of two points or objects.

This images shows cities that are all within 150 Kilometers linear distance from Terre Haute, Indiana.

Other predicate functions

- Do locations intersect?
  - ST_Intersects
- Is one geometry contained inside another?
  - ST_Contains
- Does a geography engulf another?
  - ST_CoveredBy

```
WITH geos AS (
  SELECT
    (SELECT state_geom FROM `bigquery-public-data.geo_us_boundaries.us_states`
       WHERE state_name='Massachusetts') AS ma_poly,
    (SELECT msa_geom FROM `bigquery-public-data.geo_us_boundaries.us_msa`
       WHERE name='Boston-Cambridge-Newton, MA-NH') AS boston_poly,
    (SELECT msa_geom FROM `bigquery-public-data.geo_us_boundaries.us_msa`
       WHERE name='Seattle-Tacoma-Bellevue, WA') AS seattle_poly
)
SELECT
  ST_Intersects(boston_poly, ma_poly) boston_in_ma,
  ST_Intersects(seattle_poly, ma_poly) seattle_in_ma
FROM
  geos
```

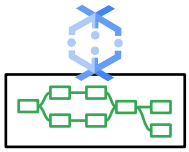
boston_in_ma	seattle_in_ma
true	false

The functions ST_Intersects, ST_Contains, and ST_CoveredBy allow reporting on the overlay or co-location of Geospatial objects.

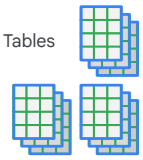
Performance Considerations

This lesson is a recap on BigQuery performance and pricing topics.

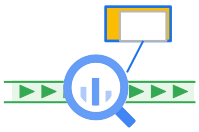
Best practices for fast, smart, data-driven decisions



Use Dataflow to do the processing and data transformations

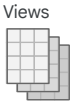


Create multiple tables for easy analysis



Use BigQuery for streaming analysis and dashboards

Store data in BigQuery for low cost long term storage



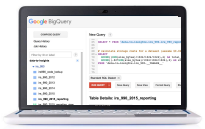
Create views for common queries

The goal for virtually every information system is to promote fast and smart decisions. Here are a few best practices to consider:

- Use Dataflow to do the processing and data transformations.
- Create multiple tables for easy analysis.
- Use BigQuery for streaming analysis and dashboards, and store data in BigQuery for low cost, long-term storage.
- Also, create views for common queries.

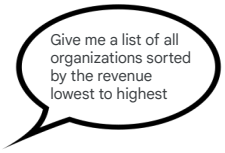
Best practices for analyzing data with BigQuery

Dataset



Know your data

Questions



Ask good questions

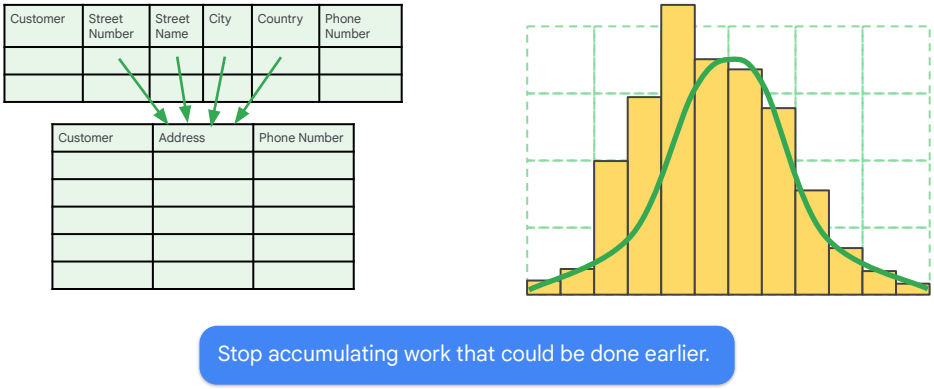
SQL

```
SELECT
  name,
  revenue
FROM dataset
ORDER BY revenue;
```

Use queries effectively

Exploring a dataset through SQL is more than just writing good code. You need to know what destination you're heading towards and the general layout of your data. Good data analysts will explore how the dataset is structured even before writing a single line of code.

How to optimize in production? Revisit the schema. Revisit the data.

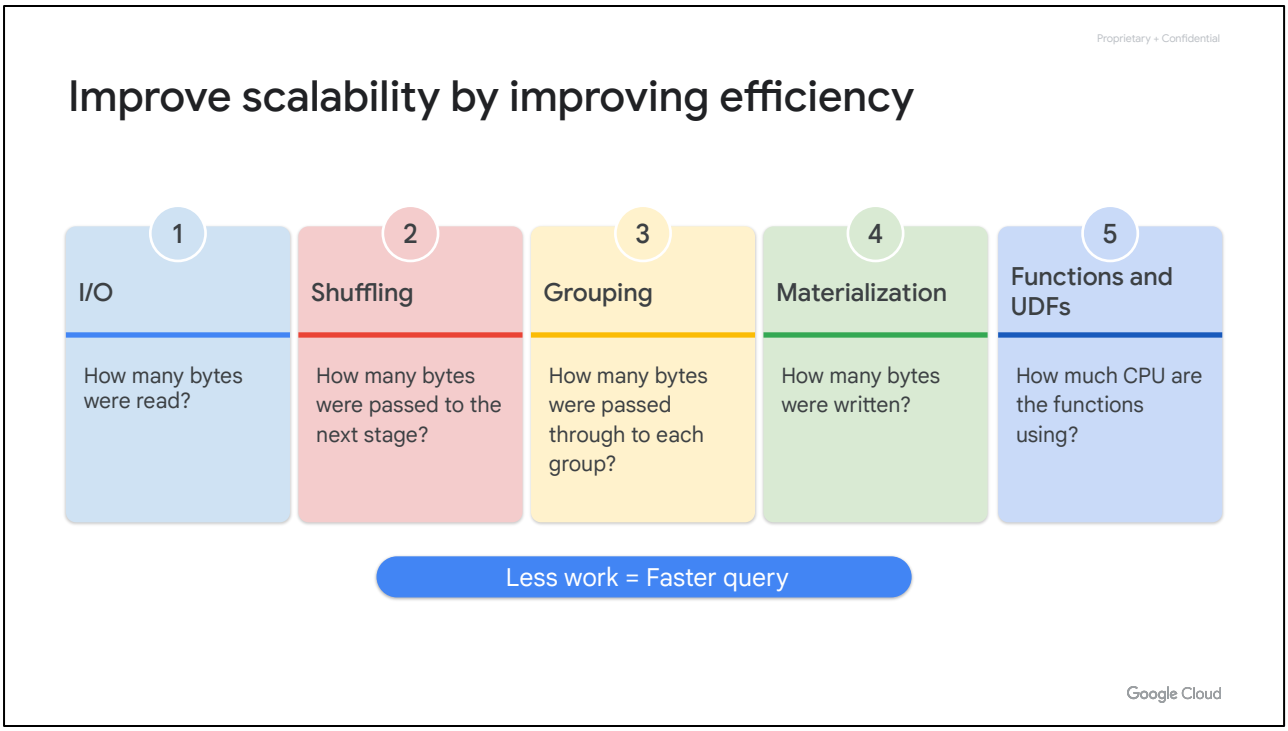


People often analyze data and develop a schema at the beginning of a project and never revisit those decisions. The assumptions they made at the beginning may have changed and are no longer true. So they attempt to adjust the downstream processes without ever reviewing and considering changing some of the original decisions.

Look at the data. Perhaps it was evenly distributed at the start of the project but as the work has grown, the data may have become skewed.

Look at the schemas. What were the goals then? Are those the same goals now? Is the organization of the data optimized for current operations?

Stop accumulating work that could be done earlier. Analogy: dirty dishes. If you clean them as you use them, the kitchen remains clean. If you save them, you end up with a sink full of dirty dishes and a lot of work.



There are five key areas for performance optimization in BigQuery and they are:

- **Input and output** - how many bytes were read from disk?
- **Shuffling** - how many bytes were passed to the next query processing stage?
- **Grouping** - how many bytes were passed through to each group?
- **Materialization** - how many bytes are written permanently out to disk?
- Lastly, **Functions and UDFs**, how computationally expensive is the query on your CPU?

There's an old Silicon Valley saying: "Don't scale up your problems. Solve them early while they are small."

Here's a cheat sheet of best practices that you should follow.

- Don't select more data columns than you need, that means avoid SELECT * at all costs when you can.
- If you have a very large dataset, consider using approximate aggregation functions instead of regular ones.
- Next, make liberal use of the WHERE clause at all times to filter data.
- Then, don't use an ORDER BY on a wide clause or sub-queries or any other sub-queries that you have, only apply ORDER BY as the last operation that you will perform.
- For joins, put the larger table on the left if you can, that'll help BigQuery optimize it and how it does its joins. If you forget, BigQuery will likely do those optimizations for you so you might not even see any difference.
- You can use wildcards in table suffixes to query multiple tables, but try to be as specific as possible as you can with those wildcards.
- For your GROUP BYs, if you're grouping by the names of every Wikipedia author ever, which means high distinct values or high cardinality, that's a bad practice or an anti-pattern. Stick to low unique value group bys.
- Lastly, use partition tables whenever you can.

[https://cloud.google.com/bigquery/docs/best-practices-performance-compute]

BigQuery supports three ways of partitioning tables

Ingestion time

```
bq query --destination_table mydataset.mytable
--time_partitioning_type=DAY
...
```

Any column that is of type DATE or TIMESTAMP

```
bq mk --table --schema a:STRING,tm:TIMESTAMP
--time_partitioning_field tm
```

Integer-typed column

```
bq mk --table --schema "customer_id:integer,value:integer"
--range_partitioning=customer_id,0,100,10 my_dataset.my_table
```


Earlier we talked about how to build a data warehouse. We mentioned that you can optimize the tables in your data warehouse by reducing the cost and amount of data read. This can be achieved by partitioning your tables.

Partitioning tables is very relevant to performance so let's revisit in the next few slides some of the main points already covered.

You enable partitioning during the table-creation process.

The slide shows how to migrate an existing table to an ingestion-time-partitioned table: just use destination table. It will cost you one table scan.

BigQuery creates new date-based partitions automatically, with no need for additional maintenance. In addition, you can specify an expiration time for data in the partitions.

New data that is inserted into a partitioned table is written to the raw partition at the time of insert. To explicitly control which partition the data is loaded to, your load job can specify a particular date partition.

Partitioning can improve query cost and performance by cutting down on data being queried

```
SELECT
  field1
FROM
  mydataset.table1
WHERE
  _PARTITIONTIME > TIMESTAMP_SUB(TIMESTAMP('2016-04-15'), INTERVAL 5 DAY)
```

Isolate the partition field in the left-hand side of the query expression!

```
bq query \
--destination_table mydataset.mytable
--time_partitioning_type=DAY
--require_partition_filter
...
```

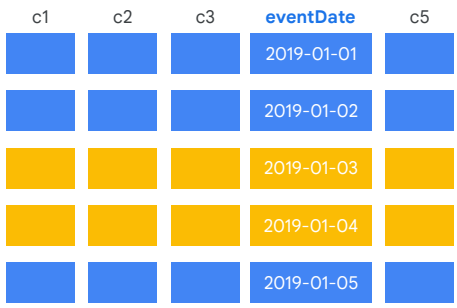

In a table partitioned by a date or timestamp column, each partition contains a single day of data. When the data is stored, BigQuery ensures that all the data in a block belongs to a single partition. A partitioned table maintains these properties across all operations that modify it: query jobs, Data Manipulation Language (DML) statements, Data Definition Language (DDL) statements, load jobs, and copy jobs. This requires BigQuery to maintain more metadata than a non-partitioned table. As the number of partitions increases, the amount of metadata overhead increases.

Although more metadata must be maintained, by ensuring that data is partitioned globally, BigQuery can more accurately estimate the bytes processed by a query before you run it. This cost calculation provides an upper bound on the final cost of the query.

A good practice is to require that queries always include the partition filter.

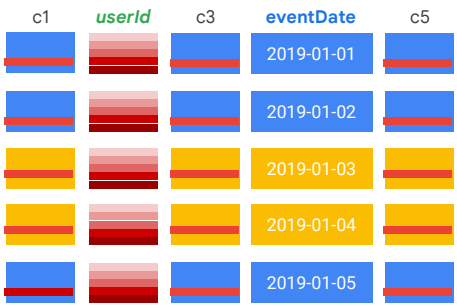
Make sure that the partition field is isolated on the left-hand-side since that's the only way BigQuery can quickly discard unnecessary partitions.

BigQuery automatically sorts the data based on values in the clustering columns



SELECT c1, c3 FROM ...
WHERE **eventDate** BETWEEN "2019-01-03" AND "2019-01-04"

Partitioned tables



SELECT c1, c3 FROM ... WHERE ***userId*** BETWEEN 52 and 63 AND **eventDate** BETWEEN "2019-01-03" AND "2019-01-04"

Clustered tables

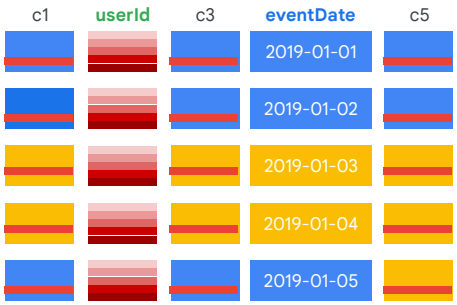
Clustering can improve the performance of certain types of queries, such as queries that use filter clauses and those that aggregate data. When data is written to a clustered table by a query or load job, BigQuery sorts the data using the values in the clustering columns. These values are used to organize the data into multiple blocks in BigQuery storage. When you submit a query containing a clause that filters data based on the clustering columns, BigQuery uses the sorted blocks to eliminate scans of unnecessary data.

Similarly, when you submit a query that aggregates data based on the values in the clustering columns, performance is improved because the sorted blocks co-locate rows with similar values.

In this example, the table is partitioned by eventDate and clustered by userId. Now, because the query looks for partitions in a specific range, only 2 of the 5 partitions are considered.

Because the query looks for userId in a specific range, BigQuery can jump to the row range and read only those rows for each of the columns needed.

Set up clustering at table creation time



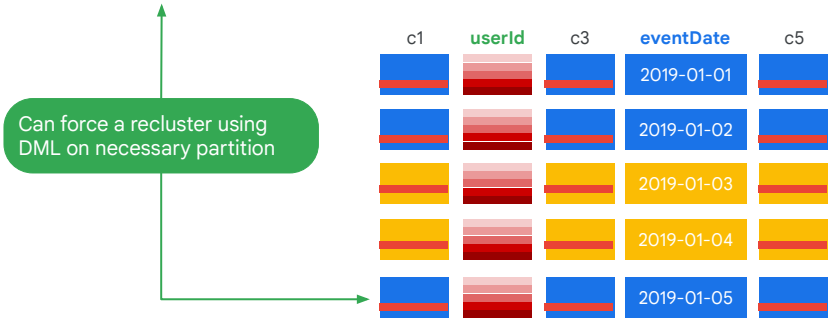

```
CREATE TABLE mydataset.myclusteredtable
 (
   c1 NUMERIC,
   userId STRING,
   c3 STRING,
   eventDate TIMESTAMP,
   C5 GEOGRAPHY
  )
 PARTITION BY DATE(eventDate)
 CLUSTER BY userId
 OPTIONS (
   partition_expiration_days=3,
   description="cluster")
 AS SELECT * FROM mydataset.myothertable
```


You set up clustering at table creation time. Here, we are creating the table, partitioning by eventDate, and clustering by userId. We are also telling BigQuery to expire partitions that are more than 3 days old.

The columns you specify in the cluster are used to co-locate related data. When you cluster a table using multiple columns, the order of columns you specify is important. The order of the specified columns determines the sort order of the data.

In streaming tables, the sorting fails over time, and so BigQuery has to recluster

```
UPDATE ds.table
SET c1 = 300
WHERE c1 = 300
AND eventDate > TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 DAY)
```



Over time, as more and more operations modify a table, the degree to which the data is sorted begins to weaken, and the table becomes only partially sorted. In a partially sorted table, queries that use the clustering columns may need to scan more blocks compared to a table that is fully sorted. You can re-cluster the data in the entire table by running a SELECT * query that selects from and overwrites the table, but guess what! You don't need to do that anymore.

BigQuery will automatically recluster your data

Automatic re-clustering

free

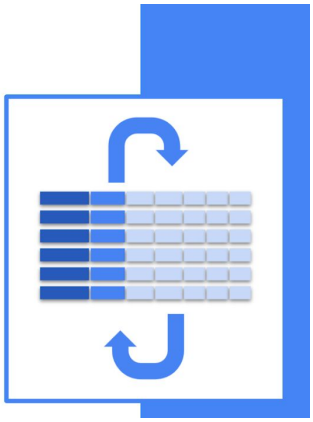
Doesn't consume your query resources

maintenance-free

Requires no setup or maintenance

autonomous

Automatically happens in the background



The great news is that BigQuery now periodically does auto-reclustering for you so you don't need to worry about your clusters getting out of date as you get new data.

Automatic re-clustering is absolutely free and automatically happens in the background -- you don't need to do anything additional to enable it.

[https://cloud.google.com/blog/products/data-analytics/whats-happening-bigquery-adding-speed-and-flexibility-10x-streaming-quota-cloud-sql-federation-and-more
https://cloud.google.com/bigquery/docs/clustered-tables#automatic_re-clustering]

Organize data through managed tables

Partitioning

Filtering storage before query execution begins to reduce costs.

Reduces a full table scan to the partitions specified.

A single column results in lower cardinality (e.g., thousands of partitions).

- Time partitioning (Pseudocolumn)
- Time partitioning (User Date/Time column)
- Integer range partitioning

Clustering

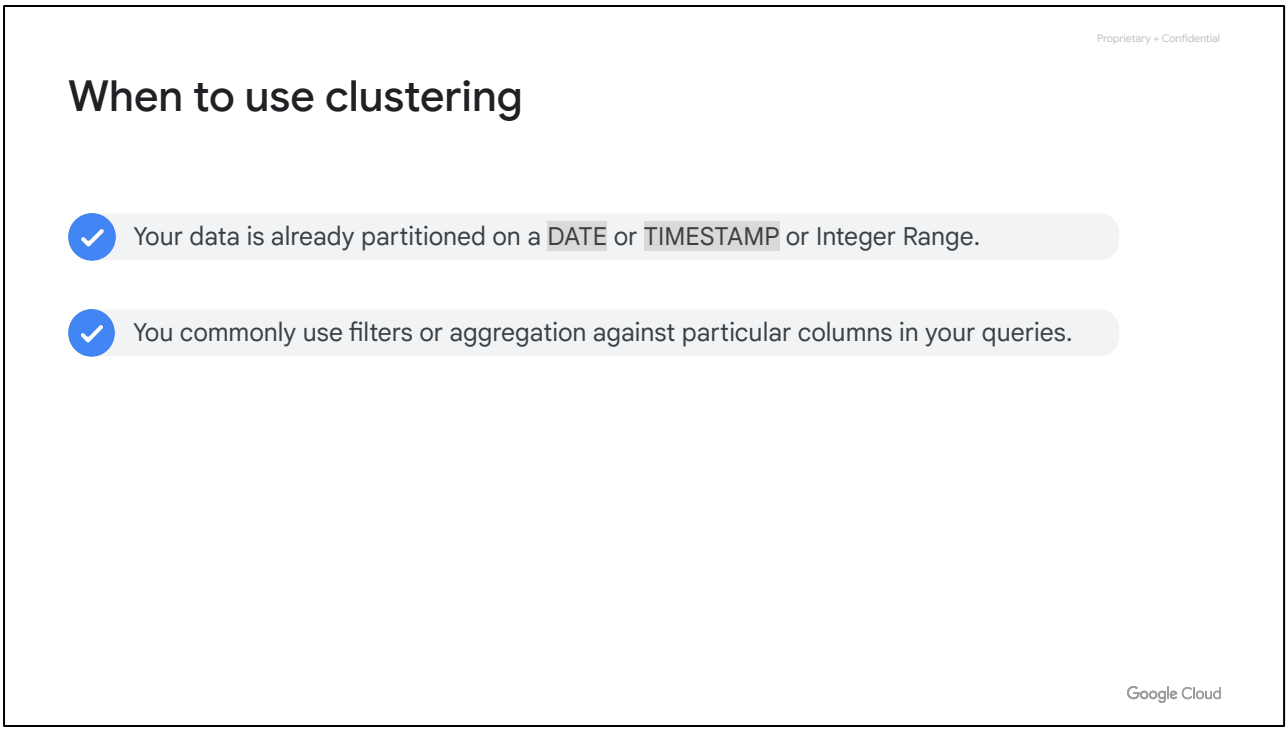
Storage optimization within columnar segments to improve filtering and record colocation.

Clustering performance and cost savings can't be assessed before query begins.

Prioritized clustering of up to 4 columns, on more diverse types (but no nested columns).

Partitioning provides a way to obtain accurate cost estimates for queries and guarantees improved cost and performance.

Clustering provides additional cost and performance benefits in addition to the partitioning benefits.

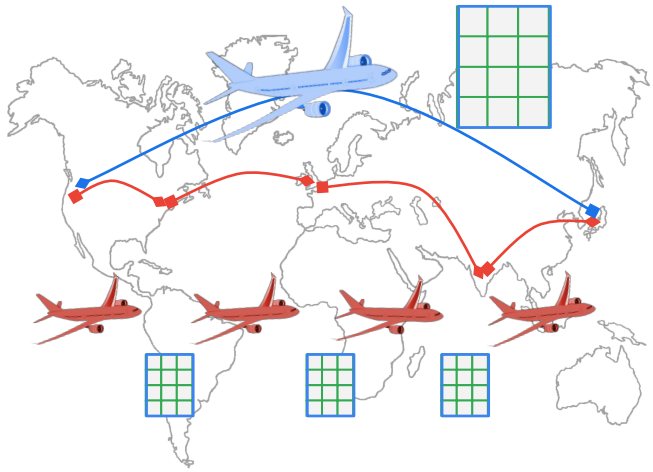


BigQuery supports clustering for both partitioned and non-partitioned tables.

When you use clustering and partitioning together, the data can be partitioned by a date or timestamp column and then clustered on a different set of columns. In this case, data in each partition is clustered based on the values of the clustering columns. Partitioning provides a way to obtain accurate cost estimates for queries.

Keep in mind if you don't have partitioned columns and you want the benefits of clustering, you can create a fake_date column of type DATE and have all the values be NULL.

Break queries into stages using intermediate tables



If you create a large, multi-stage query, each time you run it, BigQuery reads all the data that is required by the query. All the data that is read each time the query is run.

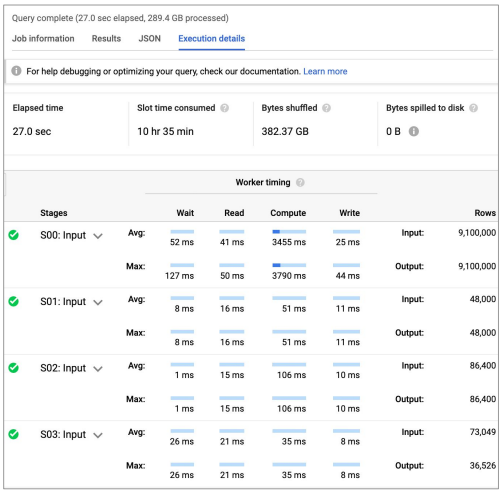
Intermediate table materialization is where you break the query into stages. Each stage materializes the query results by writing them to a destination table.

Querying the smaller destination table reduces the amount of data that is read. In general, storing the smaller materialized results is more efficient than processing the larger amount of data.

The analogy is air travel from Sunnyvale, California, USA to Japan. There is one direct flight. Or a series of four shorter connecting flights. The direct flight has to carry the fuel for the entire journey. The connecting flights only need enough fuel for each leg of the trip. The total fuel used in landing and taking off (an analogy for storing the intermediate tables) was less than the total fuel used for carrying everything in the entire journey.

Here is a tip: Compare costs of storing the data with costs of processing the data. Processing the large dataset will use more processing resources. Storing the intermediate tables will use more storage resources. In general, processing data is more expensive than storing data. But you can do the calculations yourself to establish a breakeven for your particular use case.

Track input and output counts with Execution details tab



Query complete (27.0 sec elapsed, 289.4 GB processed)

Job information | Results | JSON | Execution details

For help debugging or optimizing your query, check our documentation. Learn more

Elapsed time	Slot time consumed	Bytes shuffled	Bytes spilled to disk
27.0 sec	10 hr 35 min	382.37 GB	0 B

Worker timing

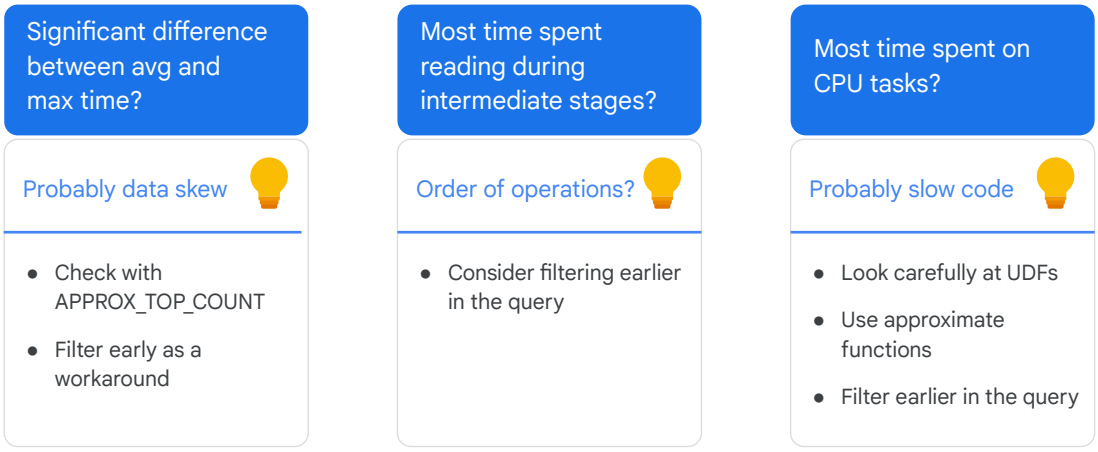
Stages		Wait	Read	Compute	Write		Rows
S00: Input	Avg:	52 ms	41 ms	3455 ms	25 ms	Input:	9,100,000
	Max:	127 ms	50 ms	3790 ms	44 ms	Output:	9,100,000
S01: Input	Avg:	8 ms	16 ms	51 ms	11 ms	Input:	48,000
	Max:	8 ms	16 ms	51 ms	11 ms	Output:	48,000
S02: Input	Avg:	1 ms	15 ms	106 ms	10 ms	Input:	86,400
	Max:	1 ms	15 ms	106 ms	10 ms	Output:	86,400
S03: Input	Avg:	26 ms	21 ms	35 ms	8 ms	Input:	73,049
	Max:	26 ms	21 ms	35 ms	8 ms	Output:	36,526

A different way to check how many records are being processed is by clicking on the Explanation tab in the BigQuery UI after running a query.

We started with 9.1 million rows and filtered down to roughly 36,000.

The query stages represent how BigQuery mapped out the work required to perform the query job.

Using BigQuery plans to optimize



Approximate Functions are a great way to improve performance. The: APPROX_COUNT_DISTINCT function returns an approximate result for COUNT(DISTINCT expression). The result is less accurate but it performs much more efficiently.

Analyze BigQuery performance in Cloud Monitoring

Custom Dashboards

Metrics include:

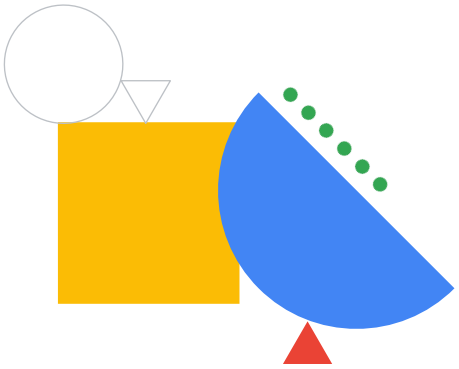
- slots utilization
- queries in flight
- upload bytes
- stored bytes

An easy way to understand the performance of your BigQuery operations is through Cloud Monitoring, a default component of every Google Cloud project.

These charts show Slot Utilization, Slots available and queries in flight for a 1 hour period.

Lab Intro

Optimizing your BigQuery Queries for Performance



In this lab, you'll optimize your BigQuery queries for performance. Specifically, you'll use BigQuery to minimize input and output from your queries.

You'll cache results from your previous queries, learn about performing efficient joins, avoid overwhelming single workers with your query, and lastly, use approximate aggregation functions. Good luck.

Once your data is loaded into BigQuery, you are charged for storing it

Active Storage Pricing

Storage pricing is prorated per MB, per second. For example, if you store:

- 100 MB for half a month, you pay $0.001 (a tenth of a cent)
- 500 GB for half a month, you pay $5 ($0.02/GB per month)
- 1 TB for a full month, you pay $20

Long-term Storage Pricing

- Table or partition not edited 90+ consecutive days
- Pricing drops ≈ 50%
- No degradation (performance, durability, availability, functionality)
- Applies to BigQuery storage only

Storage pricing is based on the amount of data stored in your tables when it is uncompressed. The size of the data is calculated based on the data types of the individual columns. Active storage pricing is prorated per MB, per second.

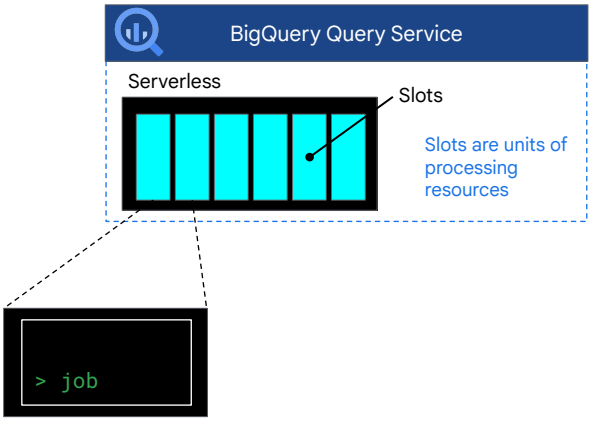
If a table is not edited for 90 consecutive days it is considered long-term storage. The price of storage for that table automatically drops by approximately 50 percent. There is no degradation of performance, durability, availability, or any other functionality.

If the table is edited, the price reverts back to the regular storage pricing, and the 90-day timer starts counting from zero. Anything that modifies the data in a table resets the timer, including:
- Loading data into a table
- Copying data into a table
- Writing query results to a table
- Using the Data Manipulation Language
- Using Data Definition Language
- Streaming data into the table

All other actions do not reset the timer, including:
- Querying a table
- Creating a view that queries a table
- Exporting data from a table
- Copying a table (to another destination table)
- Patching or updating a table resource

https://cloud.google.com/bigquery/pricing

Slots are units of resources that are consumed when a query is run



Because you don't get to see the VMs running behind the scenes, BigQuery exposes slots to help you manage resource consumption and costs. BigQuery automatically calculates how many slots are required by each query, depending on query size and complexity. The default slot capacity and allocation work well in most cases. You can monitor slot usage in Cloud Monitoring.

Candidate circumstances where additional slot capacity might improve performance are solutions with very complex queries on very large datasets with highly concurrent workloads. You can read more about slots in the online documentation or contact a sales representative.

Which BigQuery pricing model to pick?

On-Demand Pricing

- $5/TB of data processed
- Quota limit of 2,000 slots
- Slots shared amongst all on demand users
- 1st TB of data processed is free each month

Good for **exploratory work** and discovery

Flat-Rate Pricing

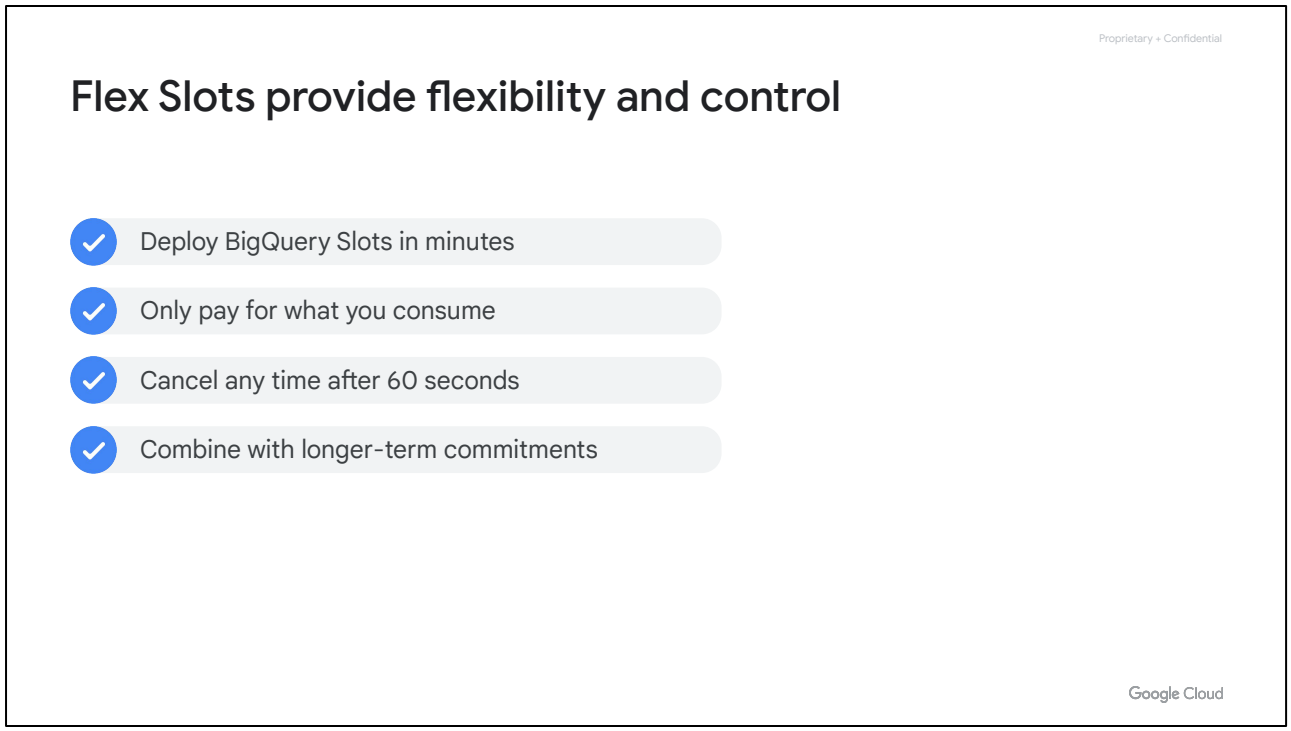
- Fixed rate pricing is $10k* per 500 slots per month
- Slots are dedicated 24/7
- Starting at 500 slots
- Unlimited use of slots

Good for **multiple large workloads** and **consistent monthly billing**

Fixed rate pricing is $10k per 500 slots per month. A 25% discount is offered for customers choosing a term length of at least 1-year ($7.5k per 500 slots).

Capacity is sold in increments of 500 slots with a current minimum of 500 slots.

Flex Slots are an option available for you to purchase BigQuery slots for short durations.



Flex Slots allow you to purchase BigQuery slots for short durations, as little as 60 seconds at a time. A slot is the unit of BigQuery analytics capacity. Flex Slots let you quickly respond to rapid demand for analytics and prepare for business events such as retail holidays and app launches.

Flex Slots give BigQuery Reservations users immense flexibility without sacrificing cost predictability or control. Flex Slots are priced at $0.04 per slot per hour, and are available in increments of 100 slots. It usually takes just a few minutes to deploy Flex Slots in BigQuery Reservations.

Once deployed, you can cancel after just 60 seconds, and you will only be billed for the seconds Flex Slots are deployed.

Flex Slots enable you to scale

- Planning for major calendar events, such as the tax season, Black Friday, popular media events, and video game launches.
- Meeting cyclical periods of high demand for analytics, like Monday mornings.
- Completing data warehouse evaluations and dialing in the optimal number of slots to use.

You can seamlessly combine Flex Slots with existing annual and monthly commitments to supplement steady-state workloads with bursty analytics capability.

For many businesses, specific days or weeks of the year are crucial. Retailers care about Black Friday and Cyber Monday, gaming studios focus on the first few days of launching new titles, and financial services companies worry about quarterly reporting and tax season. Flex Slots enable such organizations to scale up their analytics capacity for the few days necessary to sustain the business event, and scale down thereafter, only paying for what they consumed.

Considerations when enrolling for flat-rate pricing

- Flex slots are a special commitment type and a subject to capacity availability.
- Monthly commitments cannot be canceled or downgraded for 30 calendar days from the purchase confirmation date.
- Annual commitments cannot be canceled or downgraded for one calendar year.
- To purchase additional BigQuery slots, you must enter into a new commitment.
- Flat-rate pricing is purchased for a specific BigQuery location.
- Flat-rate and on-demand pricing can be used together.
- To discontinue a flat-rate pricing plan, you must cancel or downgrade your commitment.

There are a number of considerations for flat-rate pricing:

- **Flex slots** are a special commitment type.
  - The commitment duration is only 60 seconds.
  - You can cancel Flex slots any time thereafter.
  - You are charged only for the seconds your commitment was deployed.
  - Flex slots are subject to capacity availability. When you attempt to purchase Flex Slots, success of this purchase is not guaranteed. However, once your commitment purchase is successful, your capacity is guaranteed until you cancel it.

- **Monthly commitments** cannot be canceled for 30 days after your commitment is active. After the first 30 calendar days, you can cancel or downgrade at any time. If you cancel or downgrade, charges are prorated per-second at the monthly rate. For example:
  - You cannot cancel on day 29,
  - If you cancel during the first second of day 31, you're charged for 30 days and 1 second, and
  - If you cancel at the midpoint of the third month, you're charged 50% of your monthly rate for that month.

- Prior to the anniversary of your commitment date, you can choose to renew for

- another year, or convert it to a monthly or flex commitment. If you move to the monthly rate, you can cancel any time, and you're charged per-second at the monthly rate. For example:
    - If you renew for another year after your annual commitment date, you enter into a new annual commitment, and you continue to be charged the yearly commitment rate.
    - Also, if you don't renew for another year after your annual commitment date, you can cancel at any time, and charges are prorated per-second at the monthly rate.

- If you determine you need more BigQuery slots, you can purchase additional increments of 500. However, doing so will create a new commitment.

- When you purchase a flat-rate plan, you specify the allocation of slots by location. To use slots in multiple locations, you must purchase slots in each location.

- A project can use either flat-rate or on-demand pricing. If you have multiple projects in a given location, you can choose which projects use flat-rate pricing and which projects use on-demand pricing.

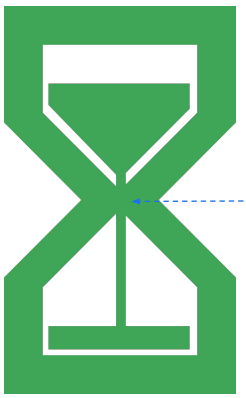
- Lastly, to discontinue a flat-rate pricing plan, you must cancel or downgrade your commitment, but only AFTER the initial commitment period (30 days or 1 year).

Estimating the right BigQuery slots allocation is critical

Guideline:
It is recommended to plan for 2,000 BigQuery slots for every 50 medium-complexity queries simultaneously

50 queries = 2,000 slots

(quota: 100 concurrent queries, but this can be raised)



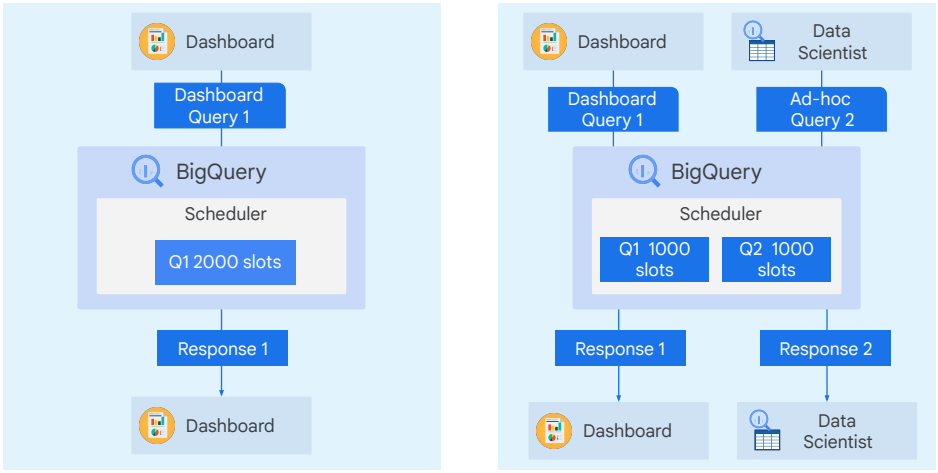
Slots available

More slots = Faster depletion rate

More concurrent queries = Slower depletion rate

BigQuery doesn't support fine-grained prioritization of interactive or batch queries. To avoid a pile up of BigQuery jobs and timely execution, estimating the right BigQuery slots allocation is critical. Currently, BigQuery times out any query taking longer than 6 hours.

BigQuery has a fair scheduler



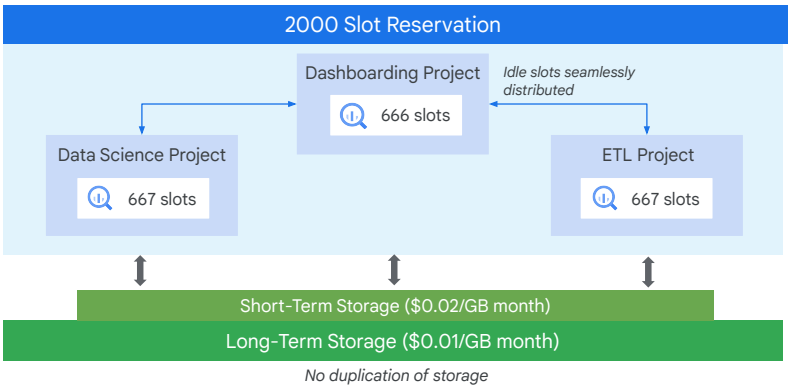
If one query is executing within BigQuery, it has full access to the amount of slots available to the project or reservation, by default 2000.

If we suddenly execute a second query, BigQuery will split the slots between the 2 queries, with each getting half the total amount of slots available, in this case 1000 each.

This subdividing of compute resources will continue to happen as more queries are executed.

This is a long way of saying, it's unlikely that one resource-heavy query will overpower the system and steal resources from other running queries.

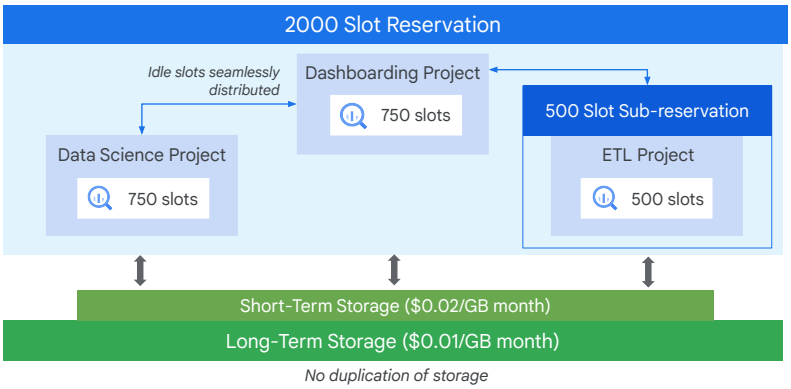
Concurrency is fair across projects, users, and queries, with unused capacity fairly divided among existing tasks



In flat rate pricing, organizations have a fixed number of slots. Concurrency is fair across projects, users, and queries. That is if you have 2,000 slots and 2 projects, each project can get up to 1,000 slots. If one project uses less, the other project will be able to use all of the remainder. If you have 2 users in each project, each user will be able to get 500 slots. And if each of the two users of the two projects runs two queries, they'll each get 500 slots.

This is a long way of saying they won't likely degrade performance by adding projects.

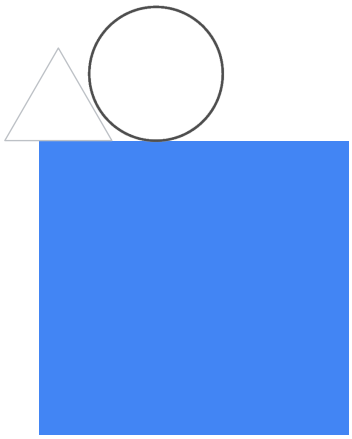
To prioritize projects, set up a hierarchical reservation



Note that if you want to prioritize one project over another, you can set up a hierarchical reservation. Let's say you have an ETL project that is somewhat lower priority than your dashboarding project. You can give the ETL project 500 slots as a sub-reservation and the dashboarding project will be in the outer one. If both projects are fully using their reservations, the ETL project can never get more than 500 slots. When one project is lightly used, the other project will be able to take the remaining slots.

Lab Intro

Partitioned Tables in BigQuery



In this lab, you practice creating date-partitioned tables in BigQuery. Specifically, you query a partitioned dataset, and then you'll create dataset partitions to improve the query performance and reduce the overall cost.