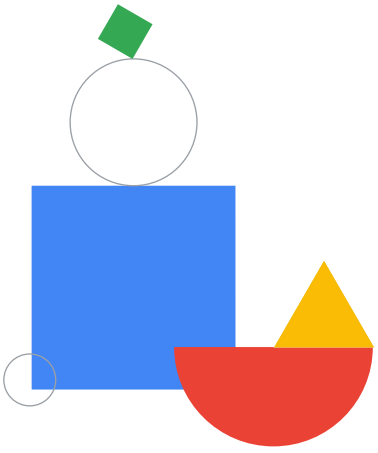
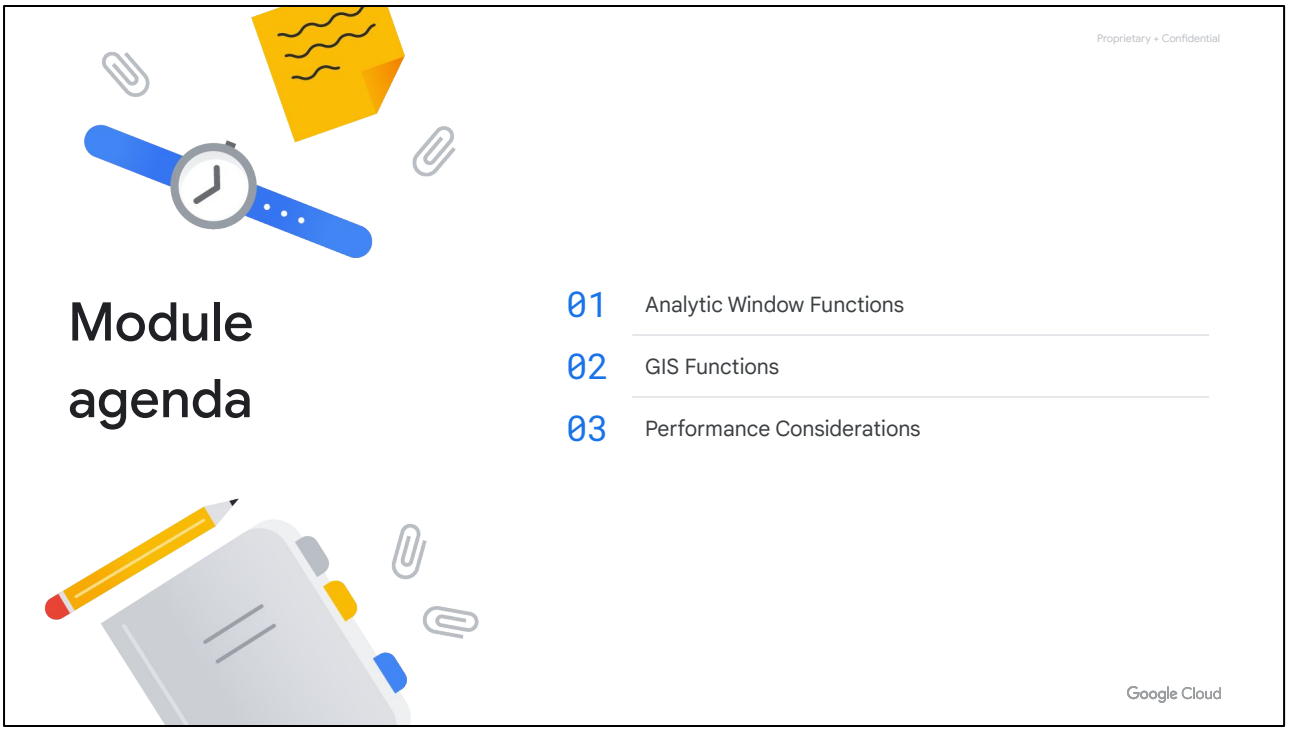
Google Cloud

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

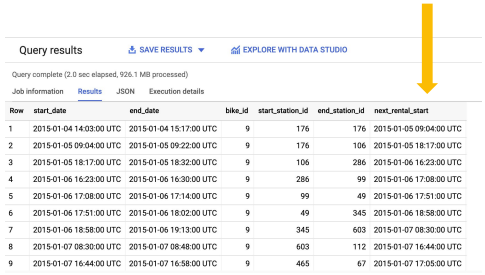
https://cloud.google.com/bigquery/docs/reference/standard-sql/aggregate_functions

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.

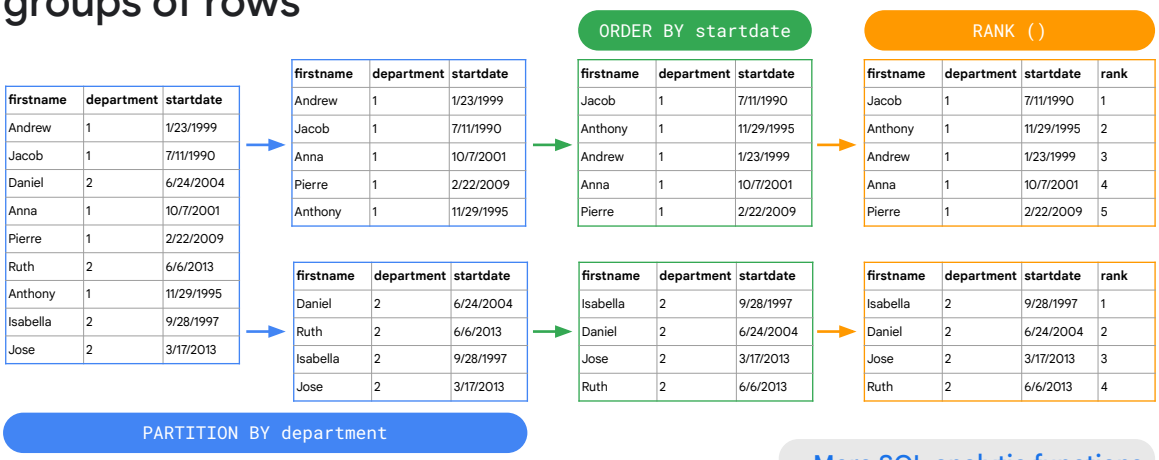
https://cloud.google.com/bigquery/docs/reference/standard-sql/navigation_functions

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

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


In this example we want to find out who has the longest tenure by department. To do so, we will use the RANK function again. By department we display the startdate in ascending order, so that we can see the employee with the oldest start date first in the result list.

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

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

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

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.

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 omitted 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
```

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

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 the distance between two points is within the distance parameter

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**

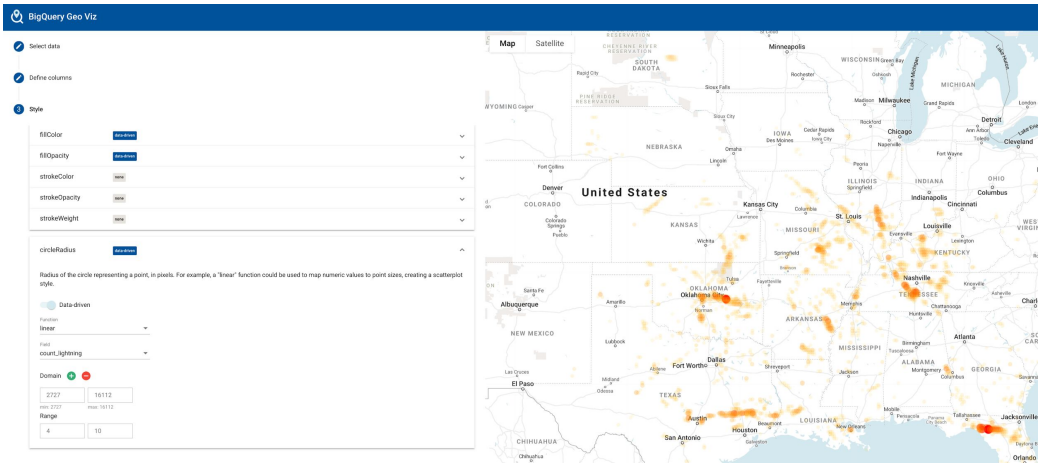
```
SELECT ST_GeogFromGeoJSON('{"type": "Point", "coordinates": [-73.967416,40.756014]}')
```

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



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)

Google Cloud

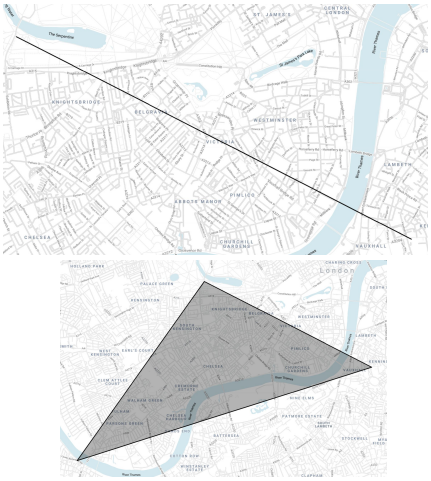
As mentioned earlier, ST_GeogPoint is used to create a Geospatial object from relevant data.

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 ...



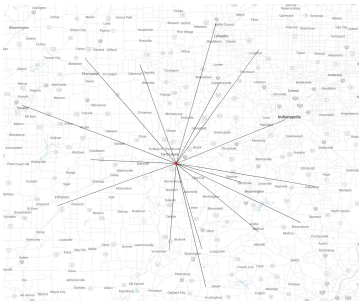
ST_MakeLine and ST_MakePolygon are two additional geospatial functions that can be used to overlay information on a map to help highlight relationships in the data.

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
```

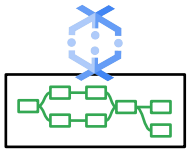
boston_in_ma	seattle_in_ma
true	false

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

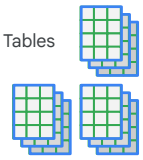
Performance Considerations

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

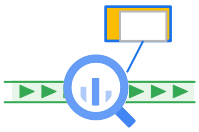
Best practices for fast, smart, data-driven decisions



Use Dataflow to do the processing and data transformations

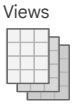


Create multiple tables for easy analysis



Use BigQuery for streaming analysis and dashboards

Store data in BigQuery for low cost long term storage



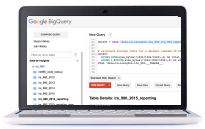
Create views for common queries

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.

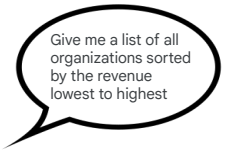
Best practices for analyzing data with BigQuery

Dataset



Know your data

Questions



Ask good questions

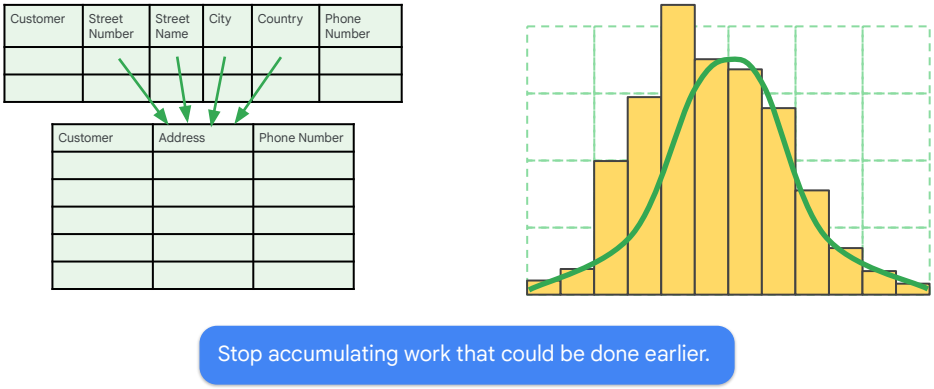
SQL

```
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.

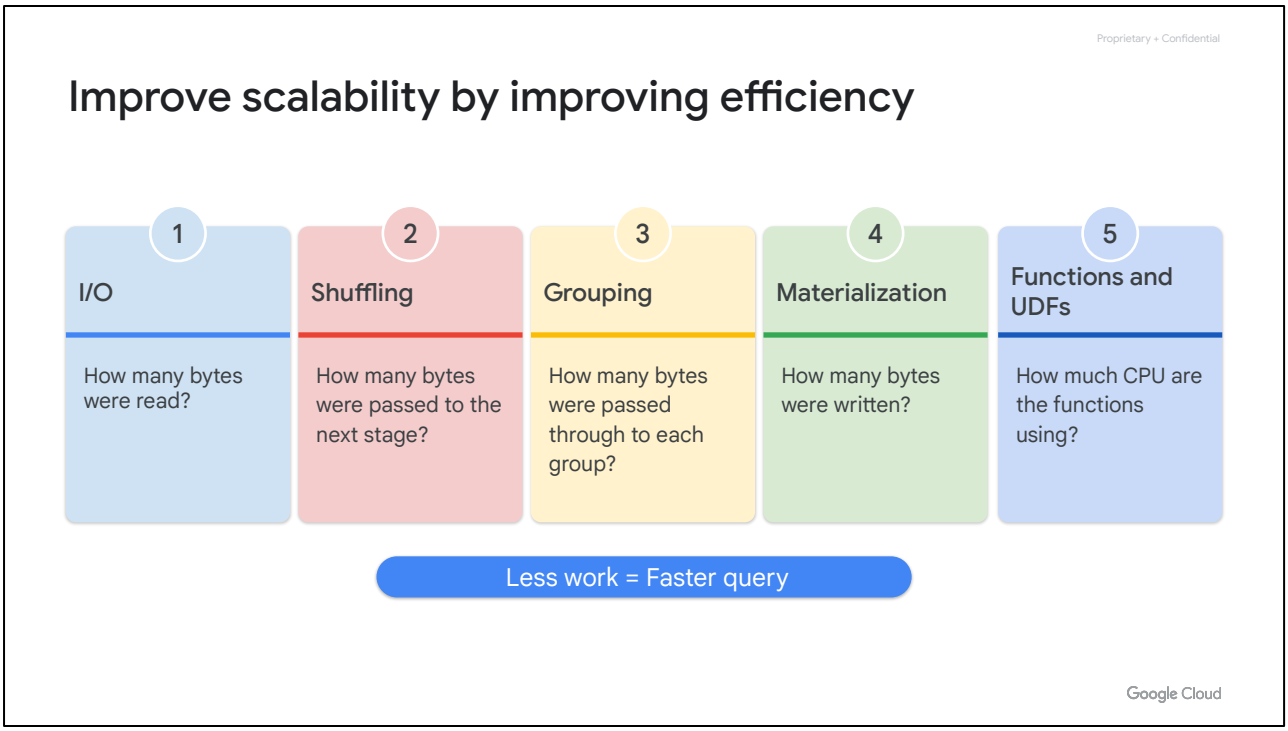


People often analyze data and develop a schema at the beginning of a project and never revisit those decisions. The assumptions they made at the beginning may have changed and are no longer true. So they attempt to adjust the downstream processes without ever reviewing and considering changing some of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

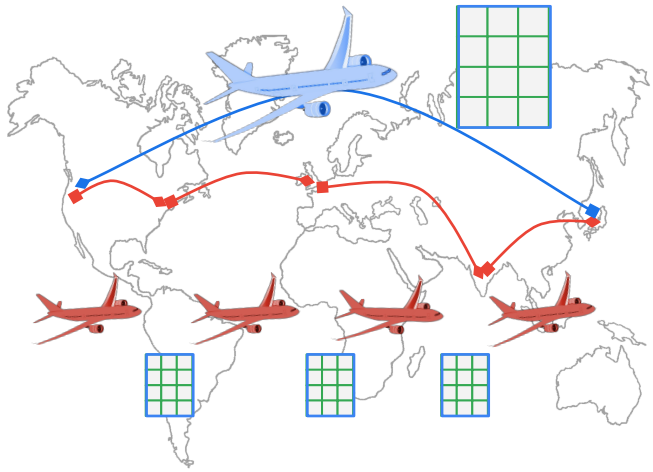
There's an old Silicon Valley saying: "Don't scale up your problems. Solve them early while they are small."

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. It is also important to understand data distribution to avoid data skew. Low cardinality means less values by key, but these values may occur very often, for example a status column has values of "available", "do not disturb", and "offline". The value "available" might occur 5,000 times but the value "do not disturb" might occur just 10 times.
- Lastly, use partition tables whenever you can.

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

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.

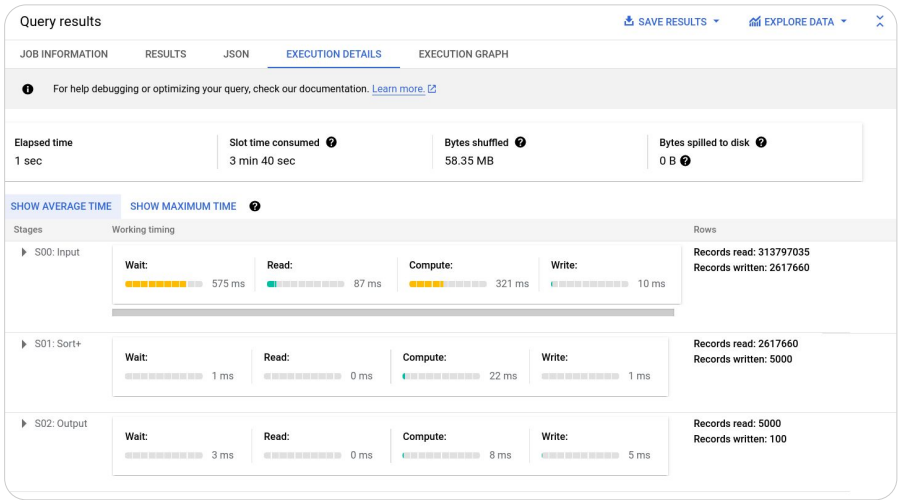
Intermediate table materialization is where you break the query into stages. Each stage materializes the query results by writing them to a destination table.

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



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 313 million rows and filtered down to 100 as our output result.

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

Using BigQuery plans to optimize

Significant difference between avg and max time?	Most time spent reading during intermediate stages?	Most time spent on CPU tasks?
Probably data skew	Order of operations?	Probably slow code
• Check with APPROX_TOP_COUNT • Filter early as a workaround	• Consider filtering earlier in the query	• Look carefully at UDFs • Use approximate functions • Filter earlier in the query

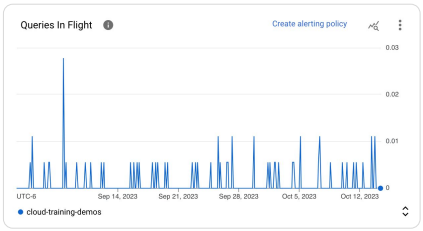
Approximate Functions are a great way to improve performance. The: APPROX_COUNT_DISTINCT function returns an approximate result for the COUNT(DISTINCT expression). The result is less accurate but it performs much more efficiently.

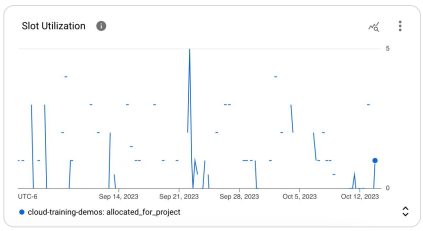
Analyze BigQuery performance in Cloud Monitoring

Custom Dashboards

Metrics include:

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





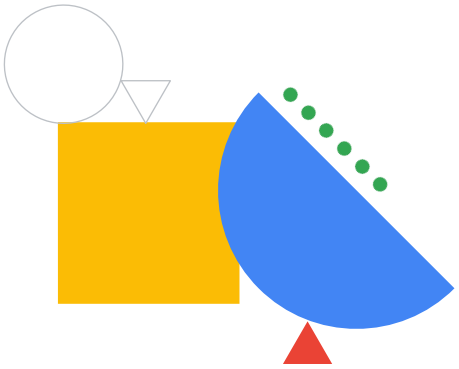
Google Cloud

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 **Queries in Flight** and **Slot Utilization** for the period of September 2 to October 14.

Lab Intro

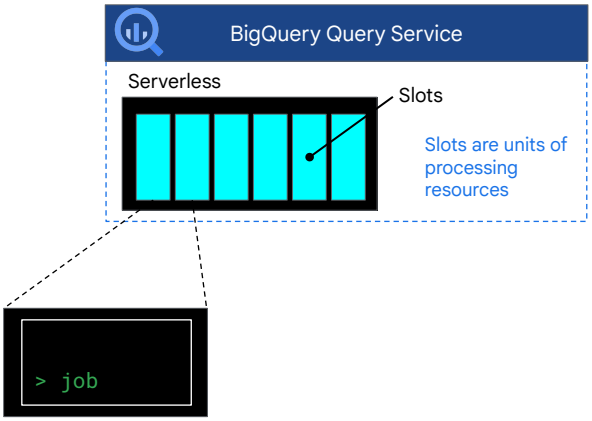
Optimizing your BigQuery Queries for Performance



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

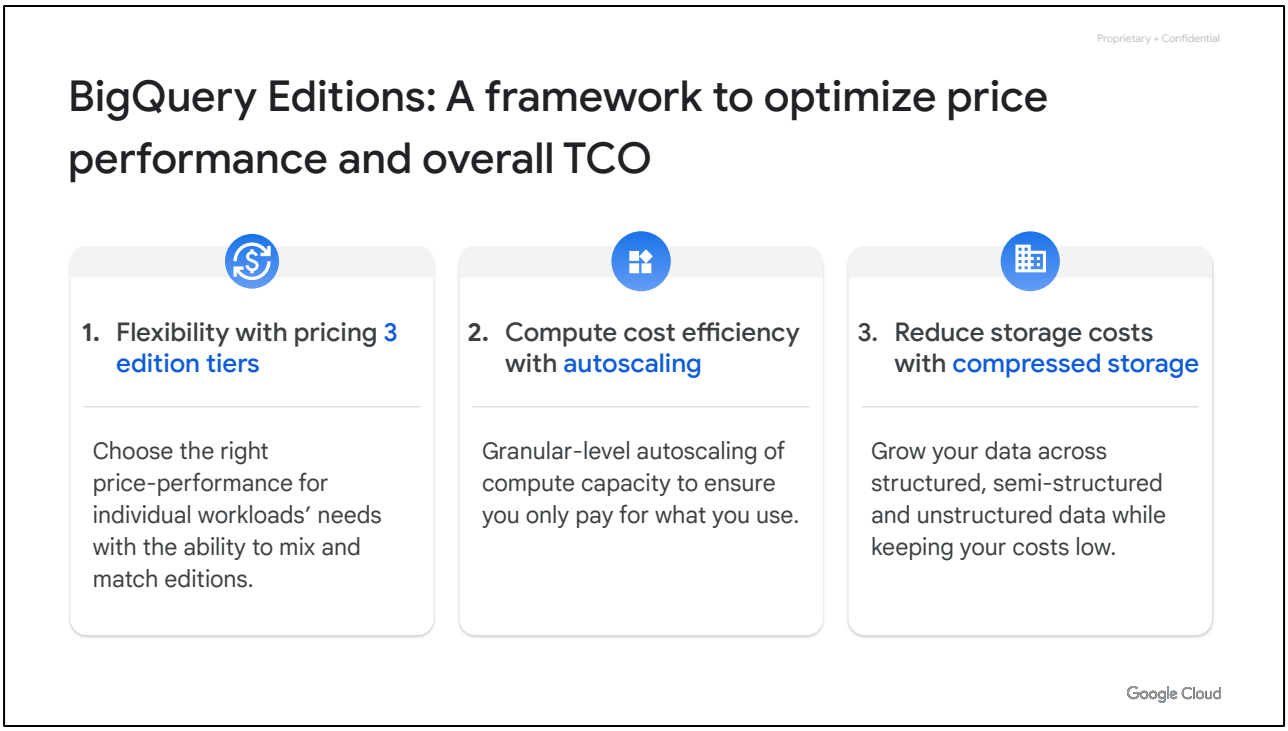
You'll cache results from your previous queries, learn about performing efficient joins, avoid overwhelming single workers with your query, and lastly, use approximate aggregation functions. Good luck.

BigQuery Architecture: 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.

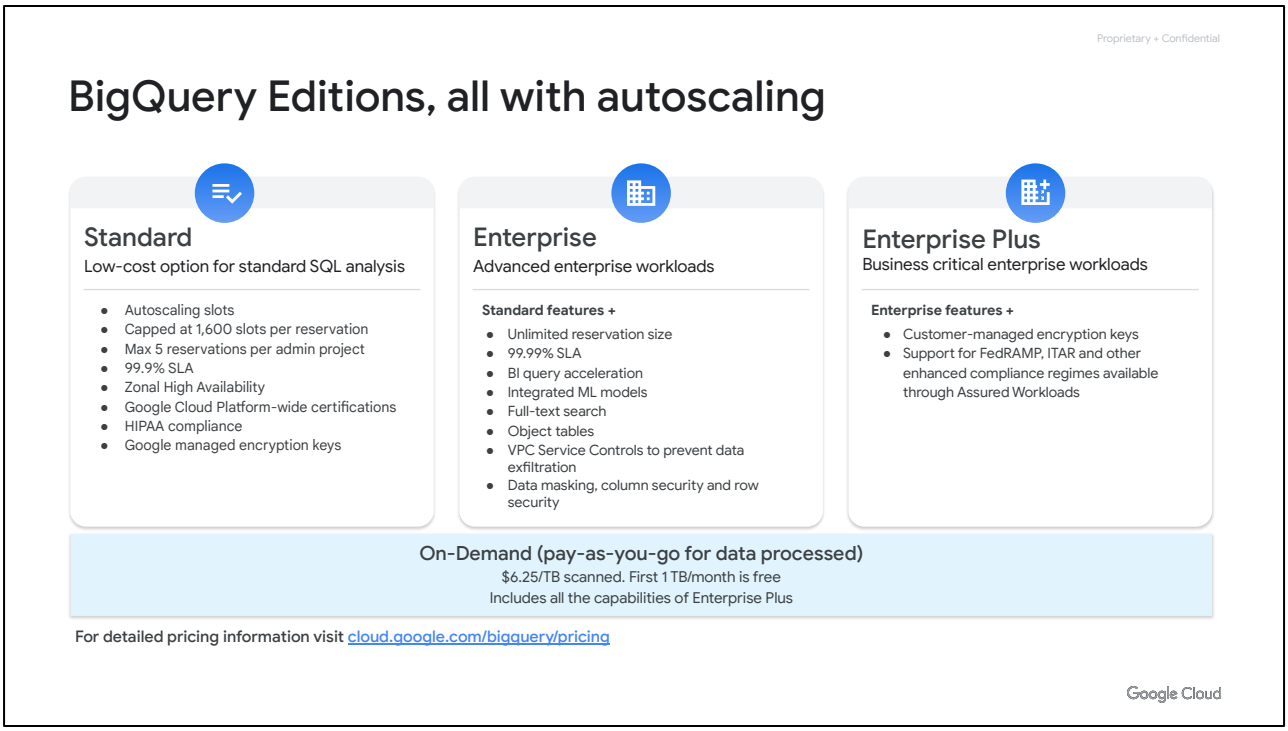
Candidate circumstances where additional slot capacity might improve performance are solutions with very complex queries on very large datasets with highly concurrent workloads. You can read more about slots in the online documentation.



BigQuery Editions is a new framework that allows users the ability to optimize price performance and overall TCO.

There are three Edition tiers and a pay-as-you-go model. All options feature an autoscaling capability.

Lastly, with Editions you have the option to lower storage costs with compressed storage.



BigQuery Editions offers choices to use BigQuery aligned with organizational and end-user needs.

The Standard tier is an entry-level, low-cost option for standard SQL analysis and offers features that meet core requirements of basic workloads.

The Enterprise tier offers a broad range of analytics features for workloads that demand a high level of scalability, flexibility, and reliability.

The Enterprise Plus tier is designed with advanced features for mission critical workloads that require multi-region support, cross-cloud analytics, advanced security and regulatory compliance.

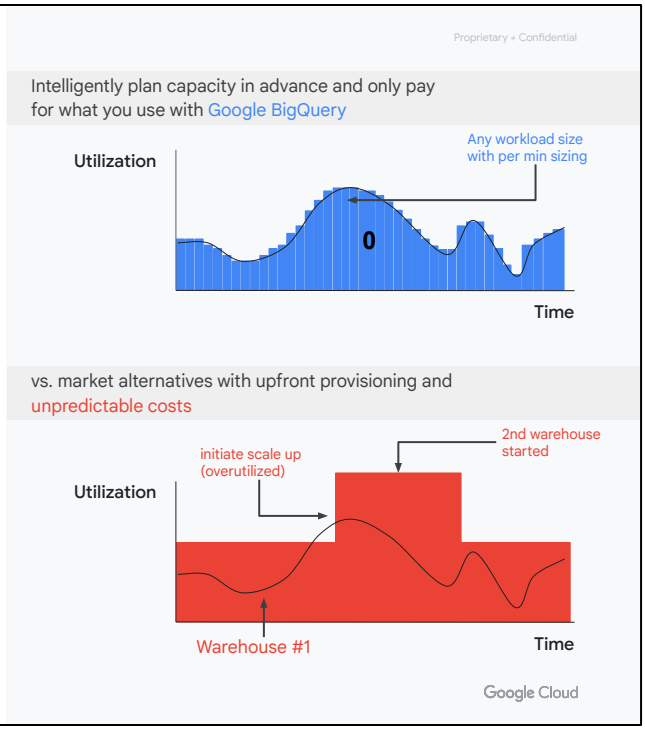
It is also possible to mix and match editions based on individual workload demands.

Customers are able to purchase these tiers with discounts based on commitment length.

In addition to the three pricing tiers, there is an on-demand pricing option that allows customers to pay for data processed.

BigQuery autoscaling

- Autoscaling with BigQuery dynamically adjusts the capacity in response to planned or unplanned changes in demand to ensure you **pay only for what you need.**
- Autoscaling will deliver up to **40% compute efficiency** gains over our current fixed capacity offering.

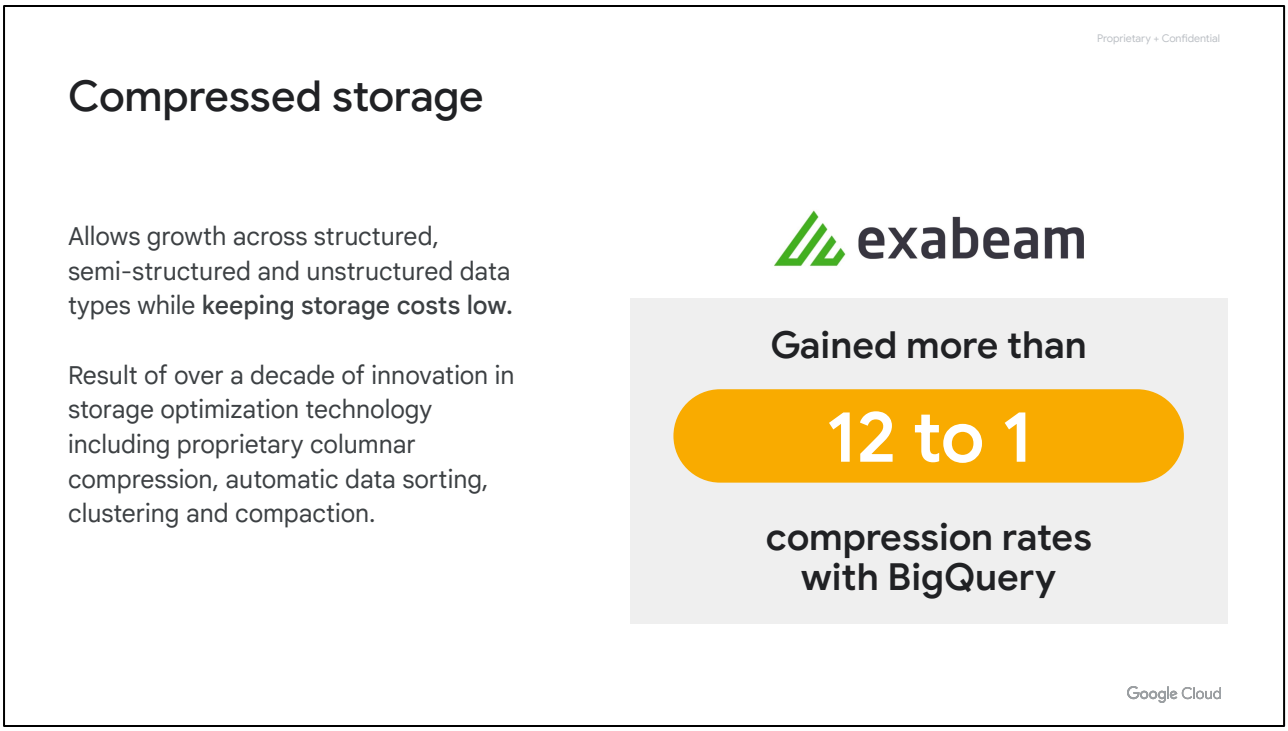


The autoscaling feature allows BigQuery to dynamically manage compute capacity.

Users only need to set up a maximum size and an optional baseline for a reservation.

BigQuery will take care of provisioning and optimizing the capacity based on workload demands without needing any additional manual intervention. This is due to its serverless architecture.

Slots are added and removed based on demand. Therefore, users only pay for the slots they consume.



A new storage option now exists, **Compressed storage**.

At a high level, this provides more flexibility in billing for data stored in BigQuery and rebalancing analytics costs between storage and compute.

An early adopter customer, Exabeam, saw a 12 to 1 size reduction with the new compression option.

Difference: Uncompressed vs. compressed storage pricing

Uncompressed (logical) storage

- Uncompressed Storage is priced at $0.02/GB for active and $0.01/GB for long term storage.
- Time travel and fail-safe bytes are included in the list price.

Compressed (physical) storage

- Compressed storage is priced at $0.04/GB for active and $0.02/GB for long term storage in US multi-region (higher list price in all other regions).
- Customers pay for time travel but have option to reduce from 7 days to 2 days. Fail-safe is a non-configurable 7 day window.
- Enabled at the Dataset level.

The per unit pricing of GB of storage is actually higher in compressed storage but with the compression ratios the overall savings can be quite significant.

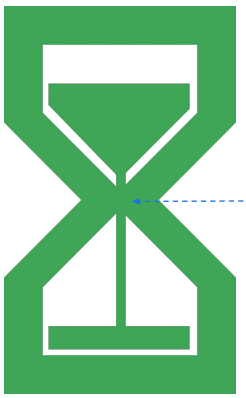
One important point to note is that the time travel window is shorter when the data is stored compressed. Time travel is only available for 2 days in the case of compressed storage.

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: 1,000 multi-statement, concurrent queries)



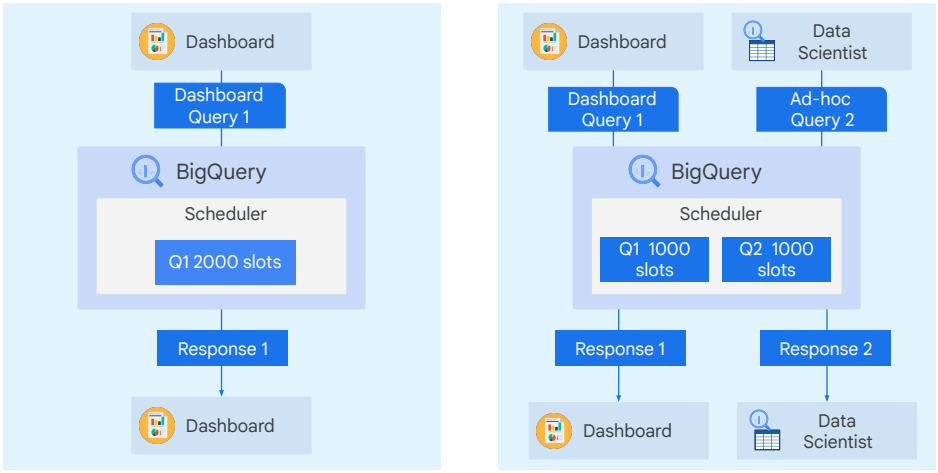
Slots available

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



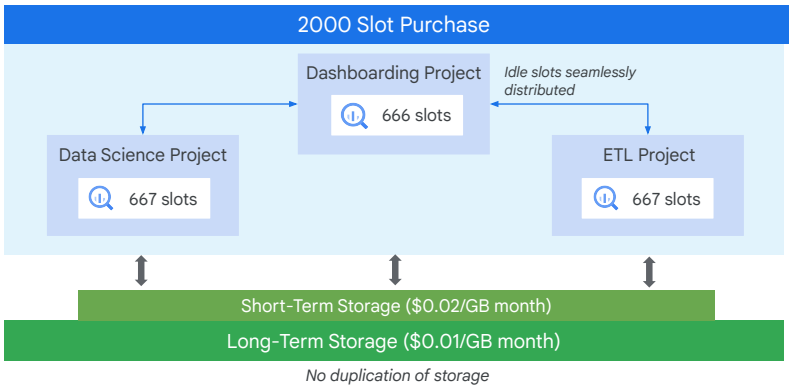
If one query is executing within BigQuery, it has full access to the amount of slots available to the project or reservation. In this case 2,000 slots were purchased.

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.

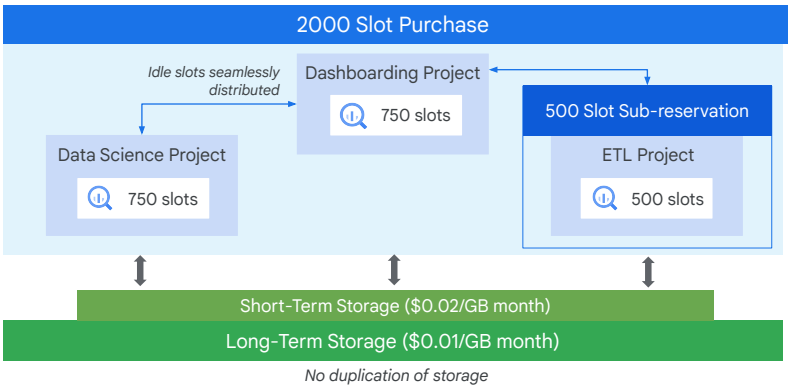
This is a long way of saying, it's unlikely that one resource-heavy query will overpower the system and steal resources from other running queries.

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



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.

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.