



# Google Cloud

---

## Advanced BigQuery Functionality

# Agenda

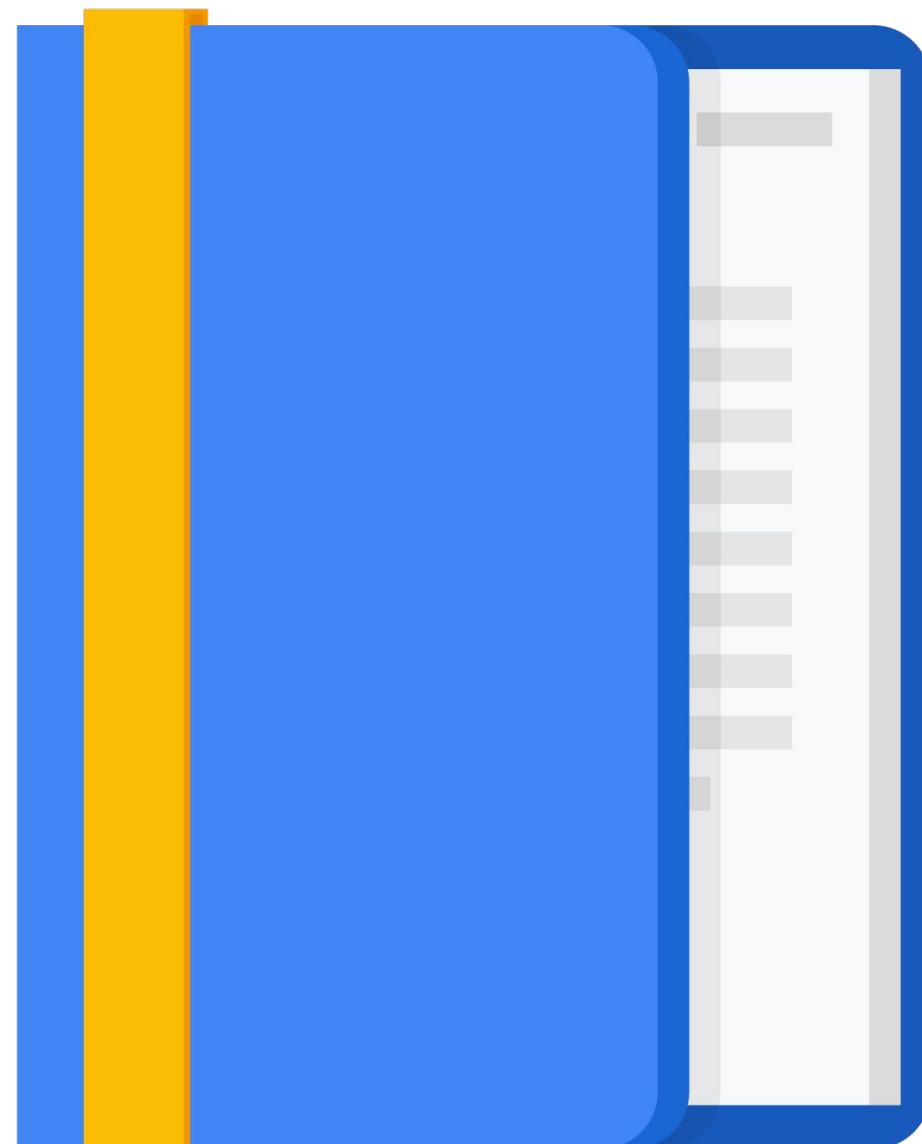
---

Analytic Window Functions

Using With Clauses

GIS Functions

Performance Considerations



# Use analytic window functions for advanced analysis

- Standard aggregations
- Navigation functions
- 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



# 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](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



# Some other navigation functions

- `NTH_VALUE`
- `LAG`
- `FIRST_VALUE`
- `LAST_VALUE`

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

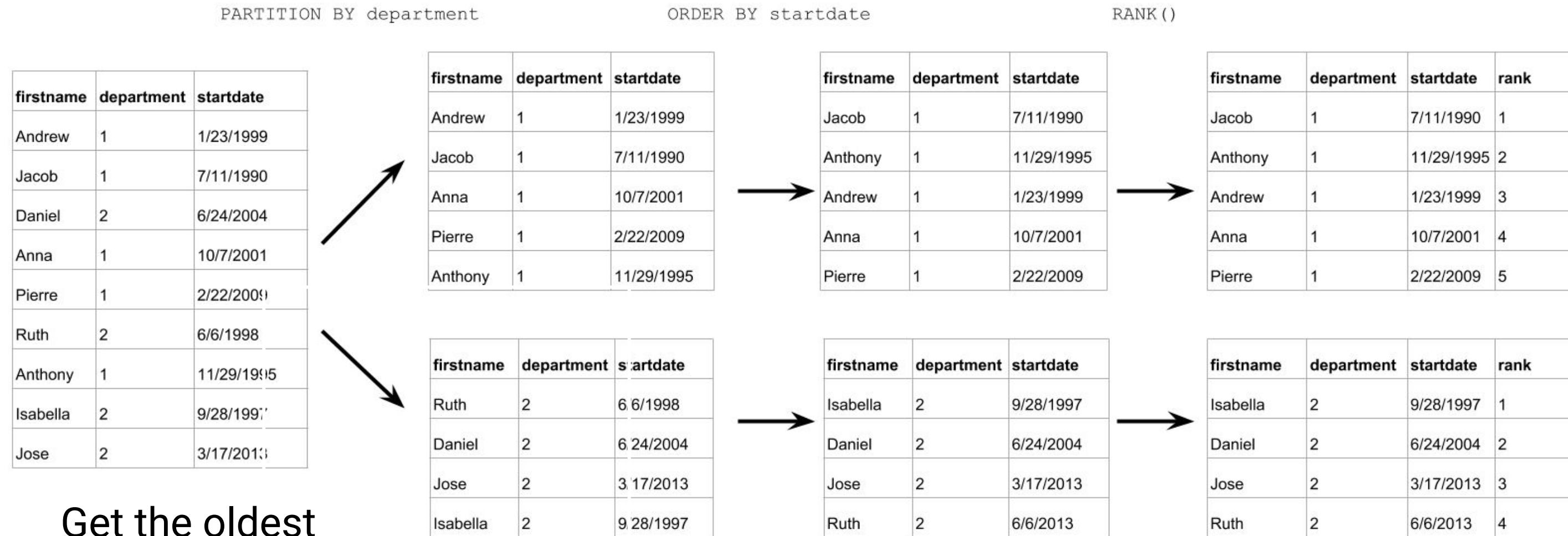
Query results  SAVE RESULTS ▾ EXPLORE WITH DATA STUDIO

Query complete (7.0 sec elapsed, 370.1 MB processed)

Job information Results JSON Execution details

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

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



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

# 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;
```

# 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](https://cloud.google.com/bigquery/docs/reference/standard-sql/numbering_functions)

# Agenda

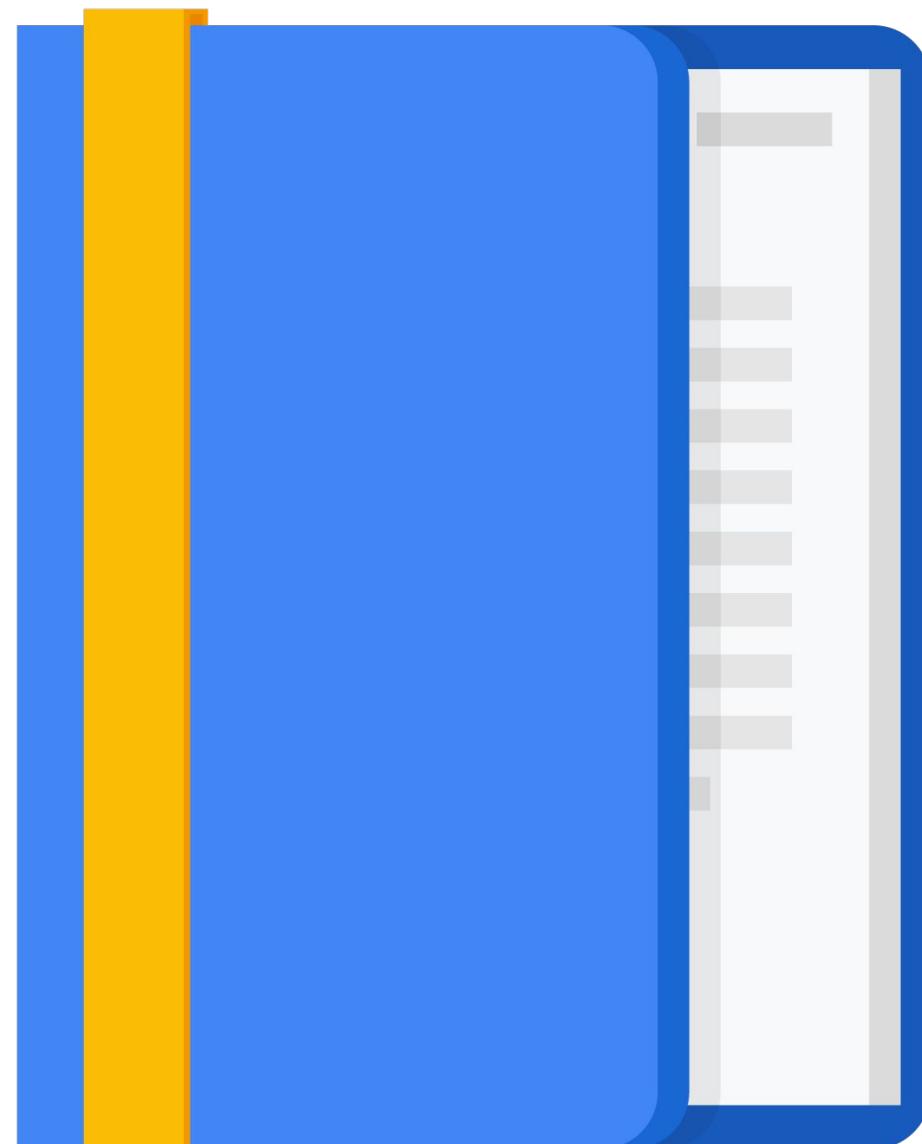
---

Analytic Window Functions

Using With Clauses

GIS Functions

Performance Considerations



# Use WITH clauses and subqueries to modularize

```
3   WITH
4
5     # 2015 filings joined with organization details
6     irs_990_2015_ein AS (
7       SELECT *
8       FROM
9         `bigquery-public-data.irs_990.irs_990_2015`
10      JOIN
11        `bigquery-public-data.irs_990.irs_990_ein` USING (ein)
12      ),
13
14     # duplicate EINs in organization details
15     duplicates AS (
16       SELECT
17         ein AS ein,
18         COUNT(ein) AS ein_count
19       FROM
20         irs_990_2015_ein
21       GROUP BY
22         ein
23       HAVING
24         ein_count > 1
25       )
26
27     # return results to store in a permanent table
28     SELECT
29       irs_990.ein AS ein,
30       irs_990.name AS name,
31       irs_990.noemployeesw3cnt AS num_employees,
32       irs_990.grsrcptspublicuse AS gross_receipts
33       # more fields omitted for brevity
34     FROM irs_990_2015_ein AS irs_990
35     LEFT JOIN duplicates
36     ON
37       irs_990.ein=duplicates.ein
38     WHERE
39       # filter out duplicate records
40       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

# Agenda

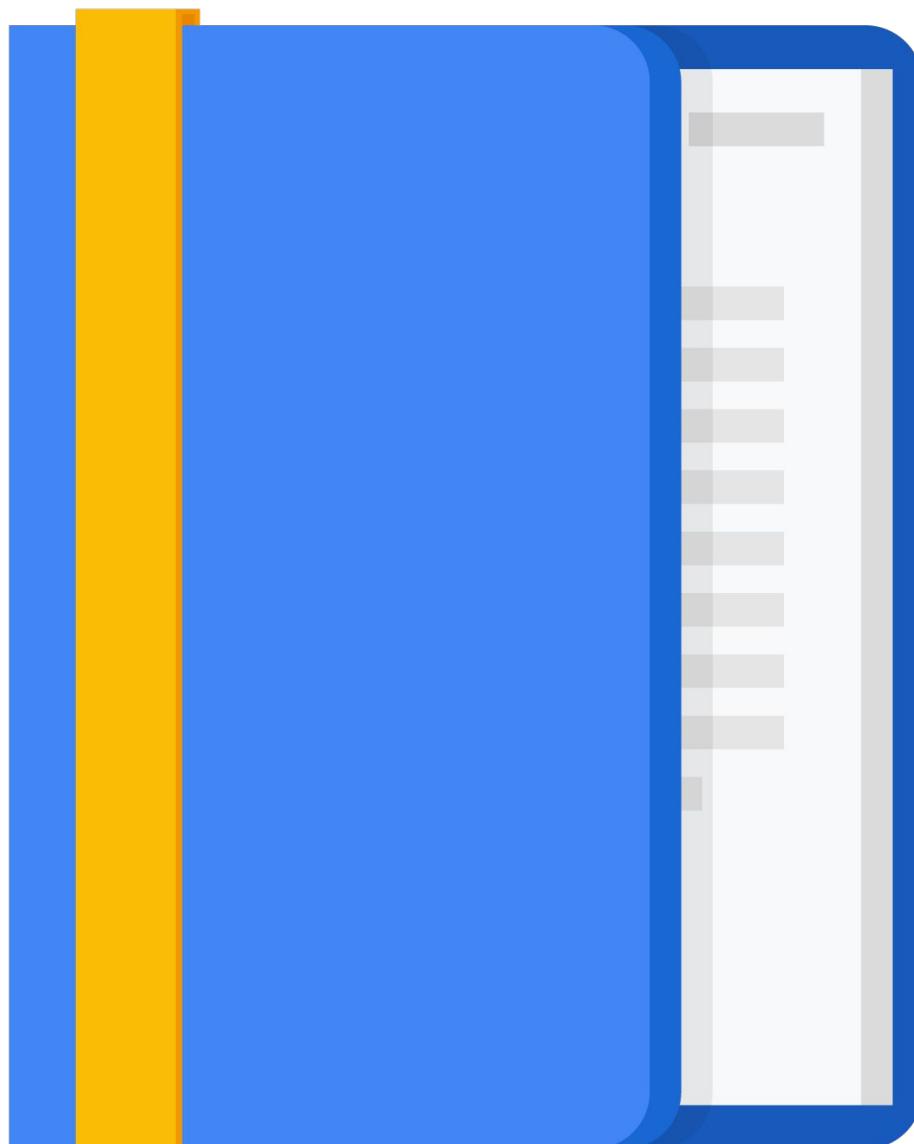
---

Analytic Window Functions

Using With Clauses

**GIS Functions**

Performance Considerations



# 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

# Use ST\_DWITHIN to check if two locations objects are within some distance

- `ST_DWithin(geography_1, geography_2, distance)`  
  
in meters
- Can lessen resolution of geo coordinates with `ST_SnapToGrid`
  - `ST_SnapToGrid(pt, 0.01)` rounds off the latitude and longitude of pt to the second decimal place

```
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
```

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

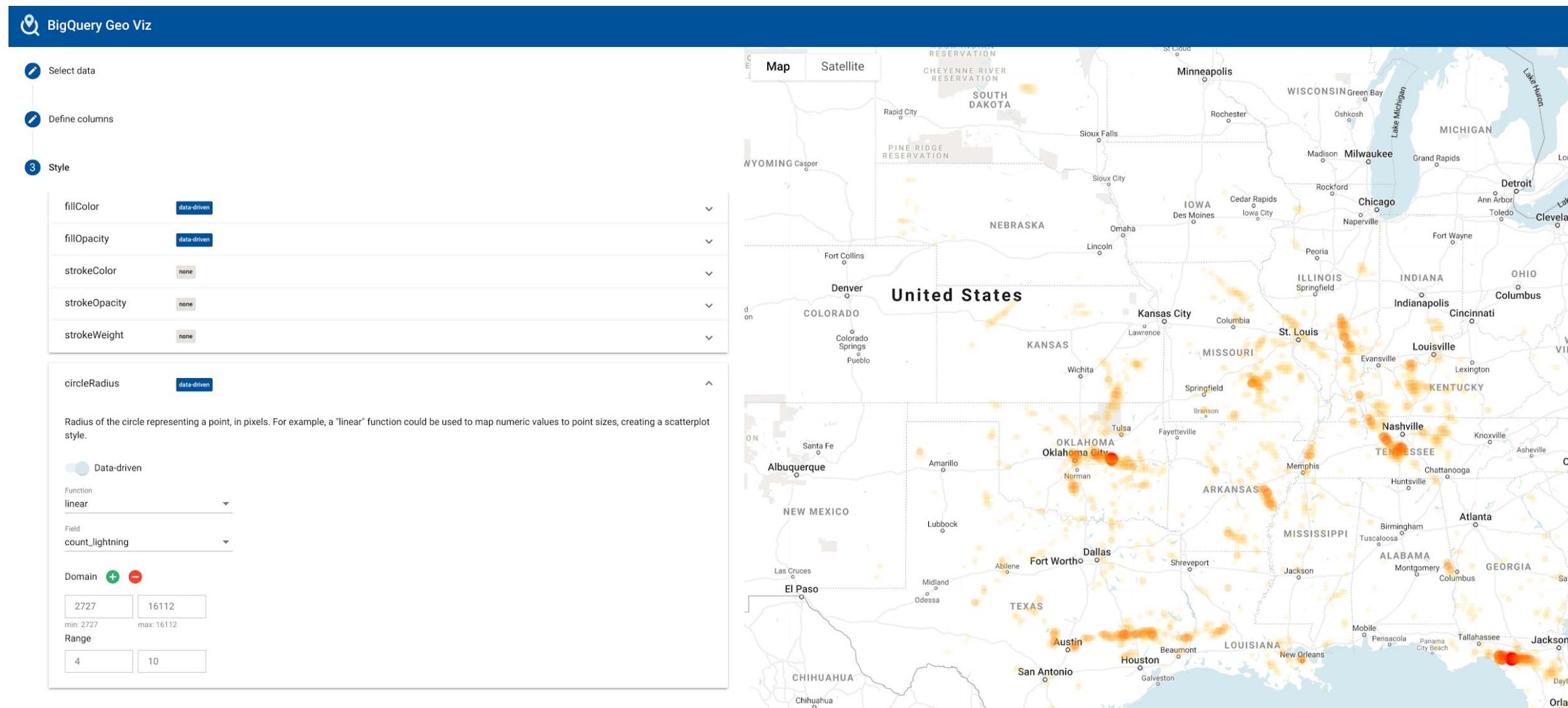
- Queries in BigQuery are much more efficient if geographic data is stored as geography types rather than as primitives

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

```
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
```

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

# Visualization with BigQuery Geo Viz



# Demo

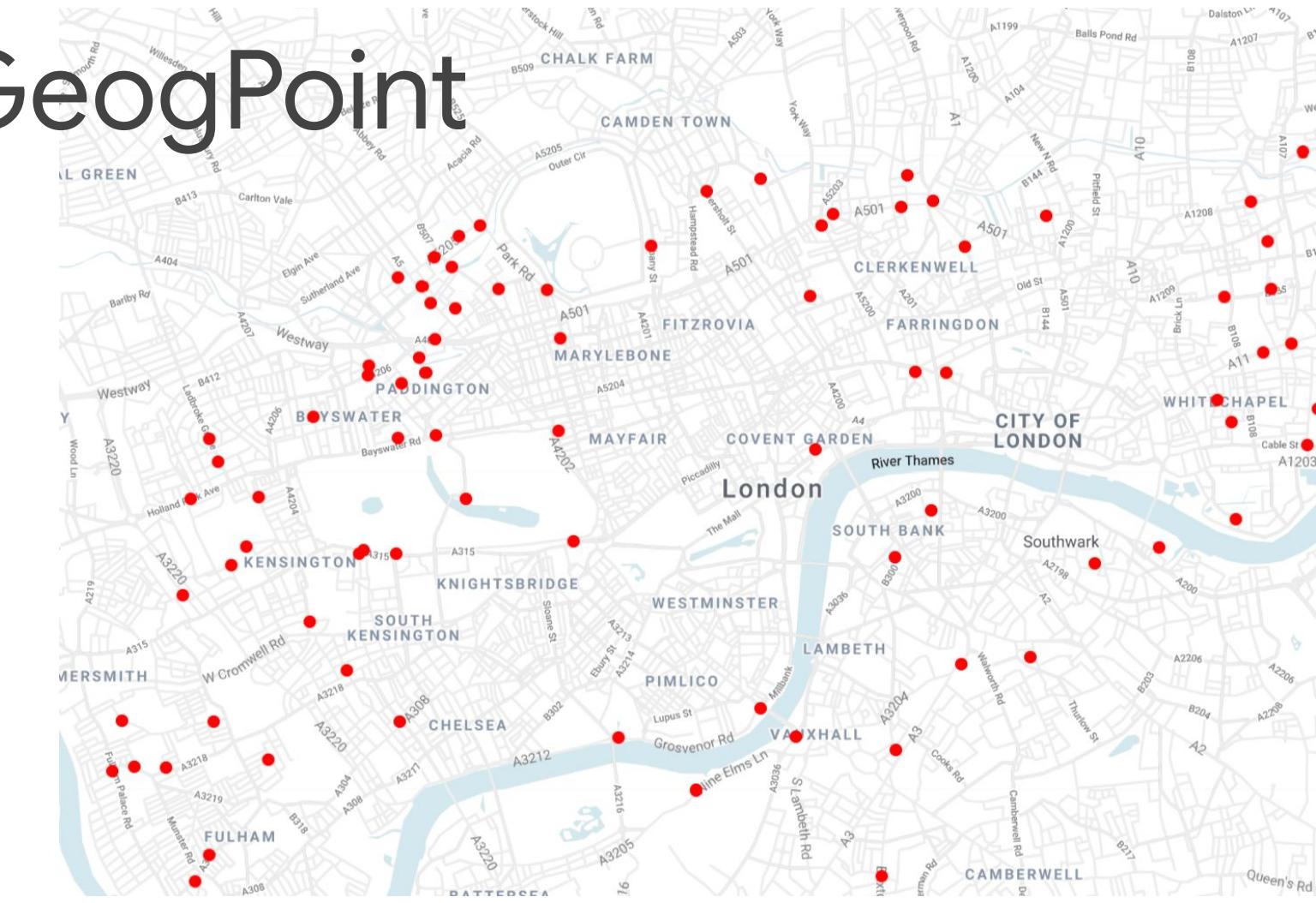
Mapping Fastest Growing Zip  
Codes with BigQuery GeoViz

# 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_static`
LIMIT
    100
```

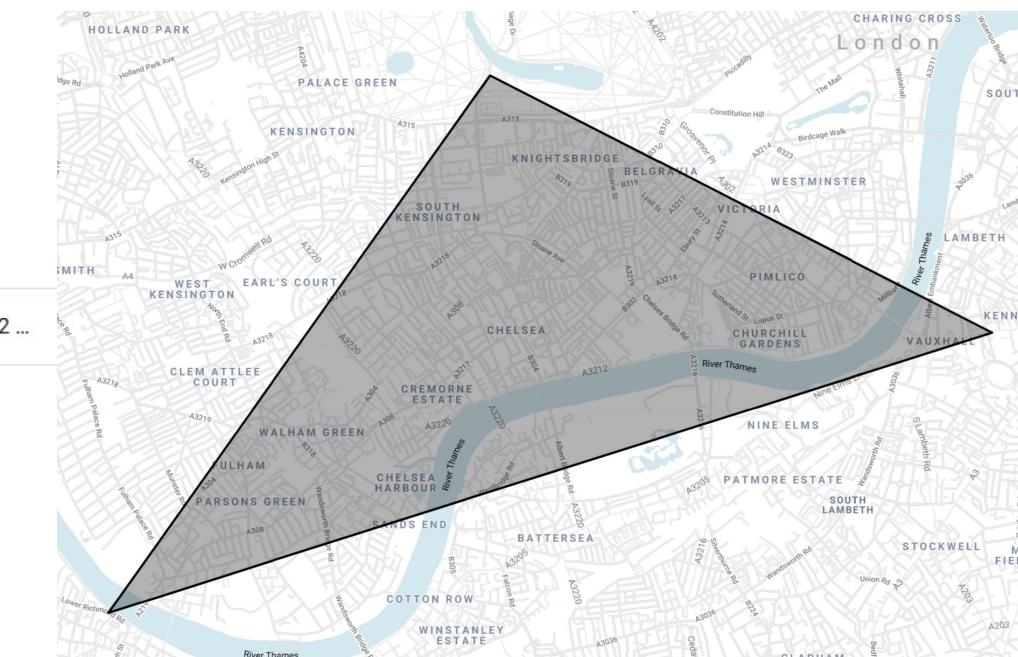
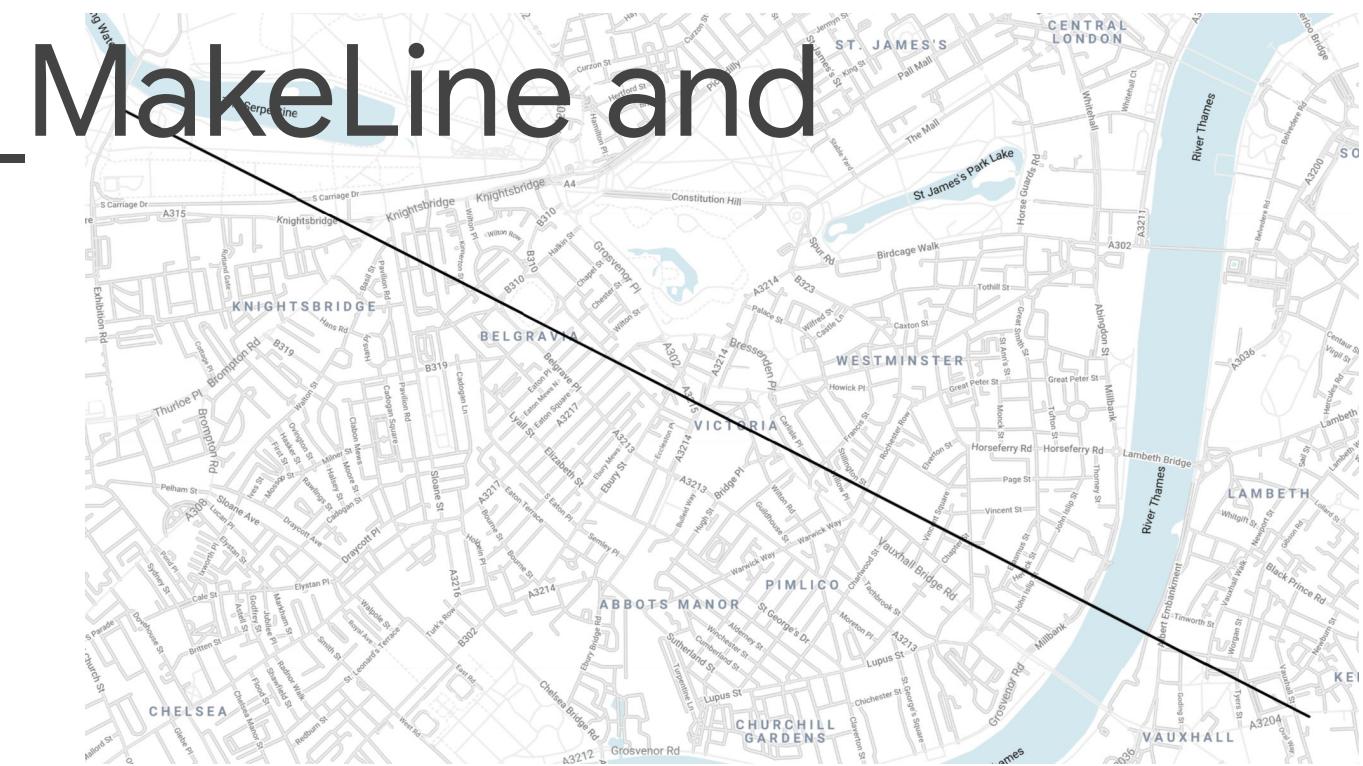
id	longitude	latitude	location
171	-0.186753859	51.4916156	POINT(-0.186753859 51.4916156)
165	-0.183716959	51.517932	POINT(-0.183716959 51.517932)
261	-0.191351186	51.5134891	POINT(-0.191351186 51.5134891)
131	-0.136792671	51.53300545	POINT(-0.136792671 51.53300545)
467	-0.030556	51.523538	POINT(-0.030556 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)



# Represent regions with ST\_MakeLine and ST\_MakePolygon

```
WITH stations AS (
  SELECT
    (SELECT ST_GeogPoint(longitude, latitude) FROM `bigquery-public-data.london_buses.stations` WHERE stop_id = 300) AS loc300,
    (SELECT ST_GeogPoint(longitude, latitude) FROM `bigquery-public-data.london_buses.stations` WHERE stop_id = 302) AS loc302,
    (SELECT ST_GeogPoint(longitude, latitude) FROM `bigquery-public-data.london_buses.stations` WHERE stop_id = 305) 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 ...))

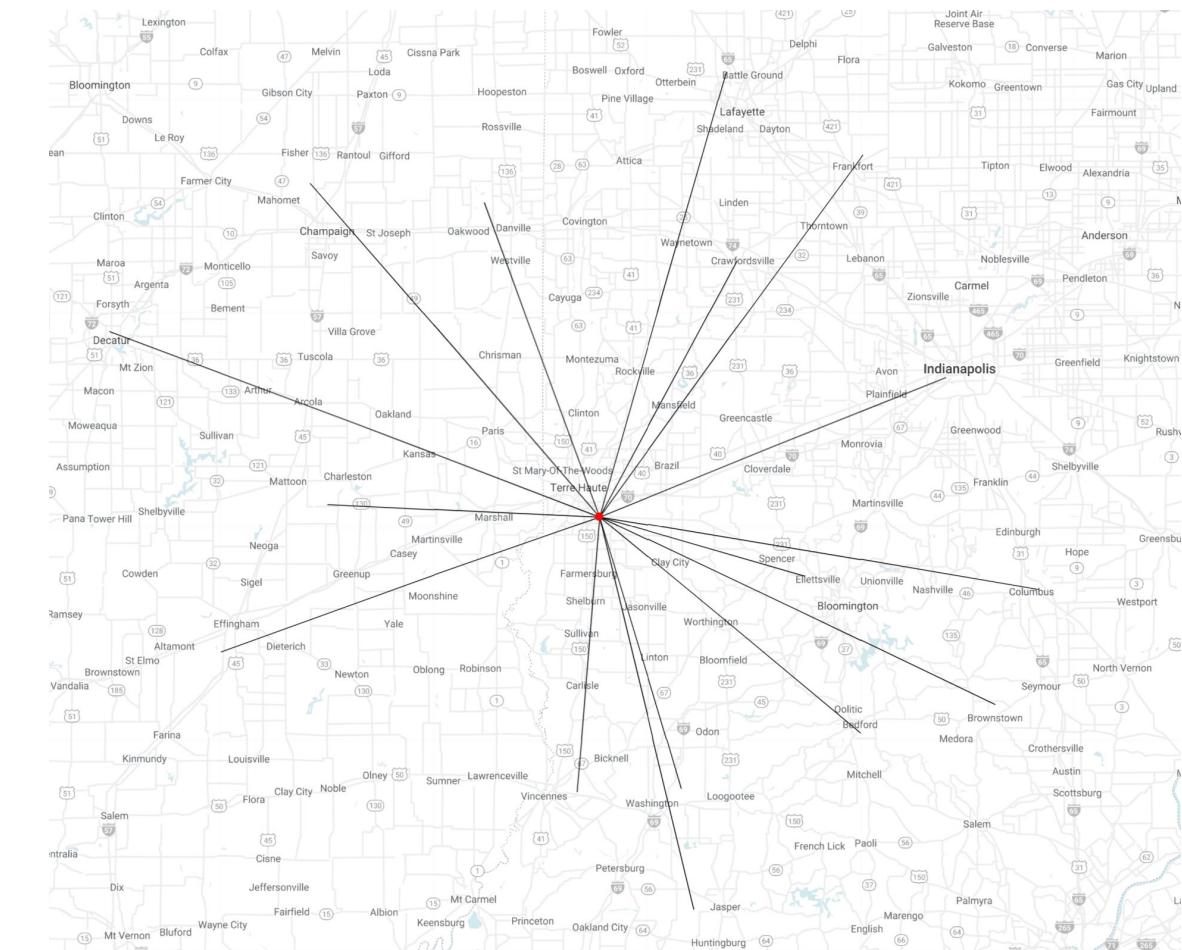


# Are Locations Within Some Distance?

- 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 40.305944)	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...



# 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

# Distance between points

- `ST_Distance`
- Can lessen resolution of geo coordinates with `ST_SnapToGrid`
  - `ST_SnapToGrid(pt, 0.01)` rounds off the latitude and longitude of pt to the second decimal place.

```
WITH geos AS (
  SELECT
    (SELECT int_point FROM `bigquery-public-data.geo_us_boundaries.us_msa`  

     WHERE name='Ann Arbor, MI') AS ann_arbor_loc,  

    (SELECT int_point FROM `bigquery-public-data.geo_us_boundaries.us_msa`  

     WHERE name='Columbus, OH') AS columbus_loc
)
SELECT
  ST_Distance(ann_arbor_loc, columbus_loc)/1000 AS dist_in_km
FROM
  geos
```

dist\_in\_km

---

267.63049397704333

---

```
SELECT
  name,
  int_point as point_full,
  ST_SnapToGrid(int_point, .01) as point_2decs
FROM
  `bigquery-public-data.geo_us_boundaries.us_msa`
WHERE
  name='Ann Arbor, MI'
```

name	point_full	point_2decs
Ann Arbor, MI	POINT(-83.8446336 42.2523272)	POINT(-83.84 42.25)

# Combine multiple polygons into a single unit

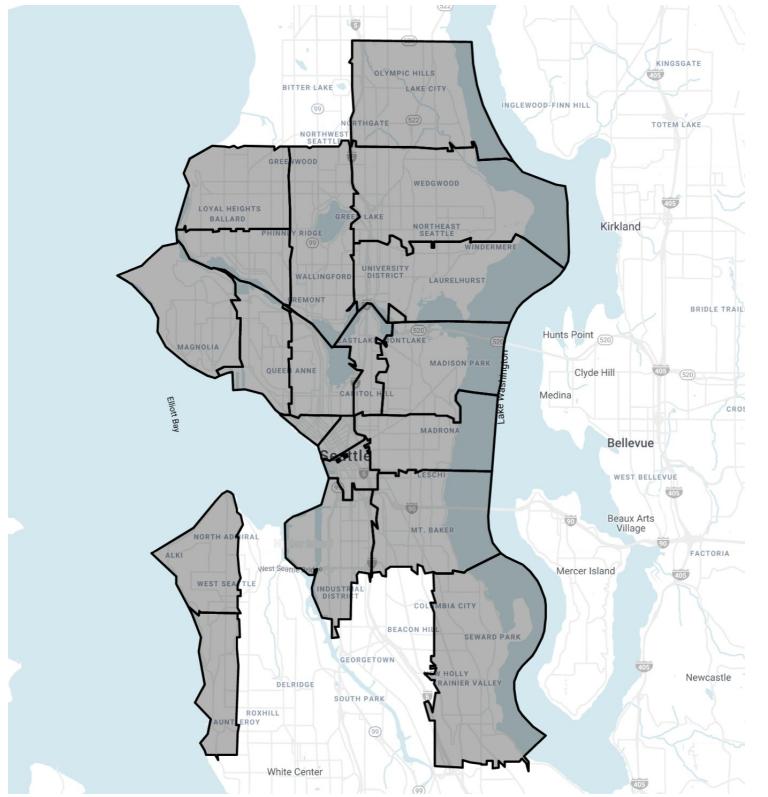
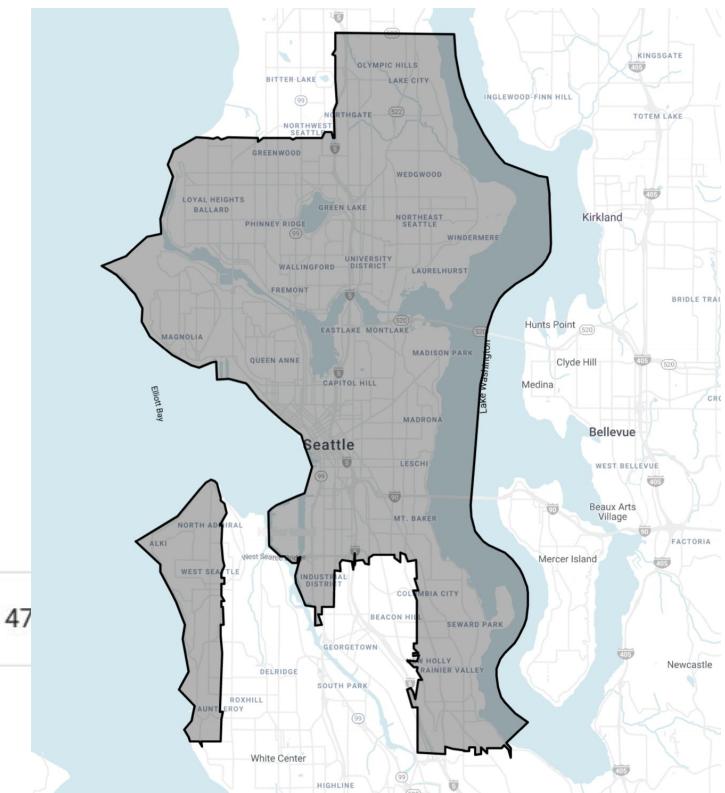
- ST\_Union\_Agg
- Find centroid of the union with ST\_Centroid\_Agg

```
SELECT
    zip_code,
    zcta_geom AS loc
FROM
    `bigquery-public-data.geo_us_boundaries.us_zip_codes` `zips`
WHERE
    city = 'Seattle city'
    AND state_code = 'WA'
```

zip_code	loc
98195	POLYGON((-122.312412 47.647196, -122.310437 47.64712, -122.304698 47...
98121	POLYGON((-122.361823642306 47.6187252120307, -122.359893520553 47...
98199	POLYGON((-122.429841 47.658919, -122.421367 47.652949, -122.4213666...
98154	POLYGON((-122.334671 47.606236, -122.332011 47.606096, -122.332561 4...

```
SELECT
    ST_UNION_AGG(zcta_geom) as all_zips,
    ST_CENTROID_AGG(zcta_geom) as centroid
FROM
    `bigquery-public-data.geo_us_boundaries.us_zip_codes` `zips`
WHERE
    city = 'Seattle city'
    AND state_code = 'WA'
```

all_zips	centroid
	MULTIPOLYGON((( -122.421139 47.57602, -122.409839 47.56892, -122.4097... POINT(-122.319780150055 47...



# Agenda

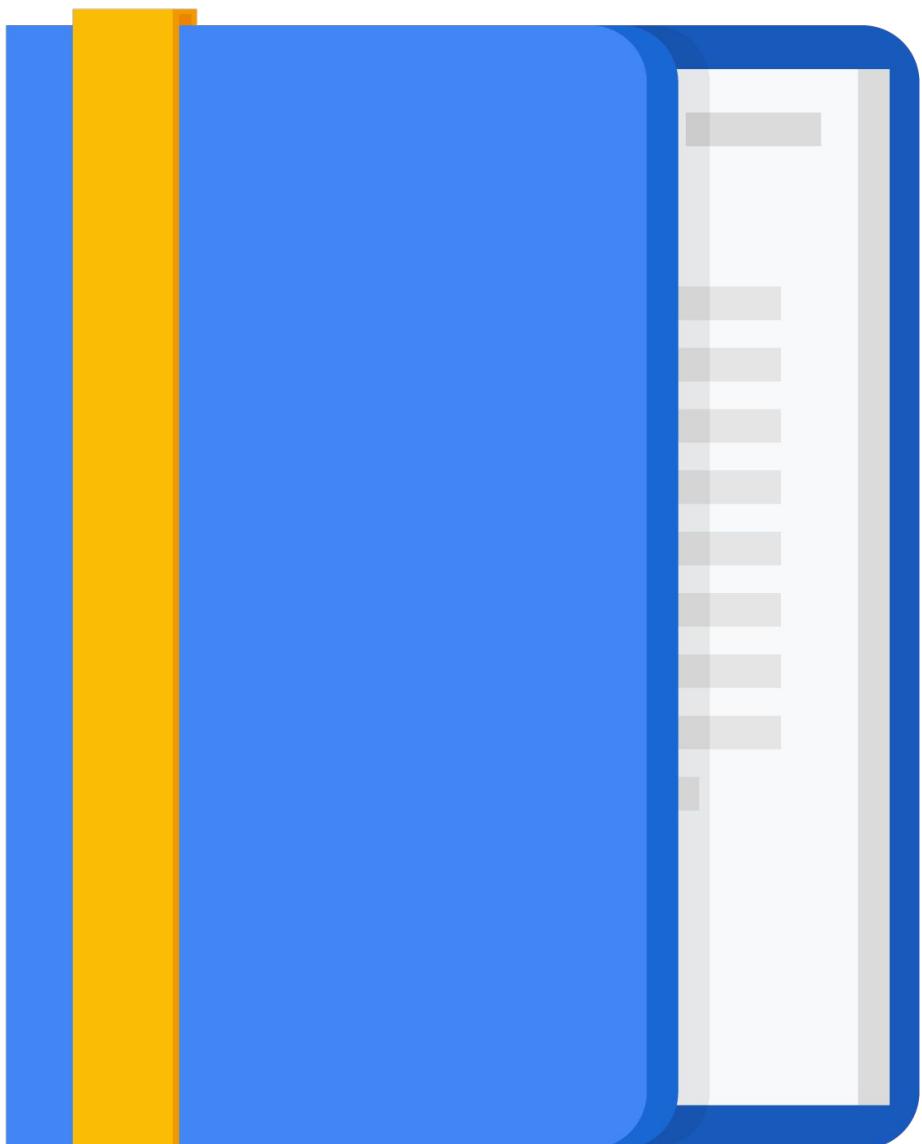
---

Analytic Window Functions

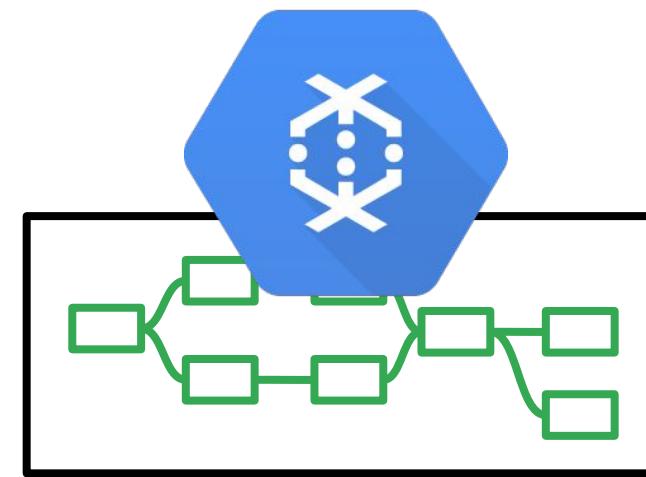
Using With Clauses

GIS Functions

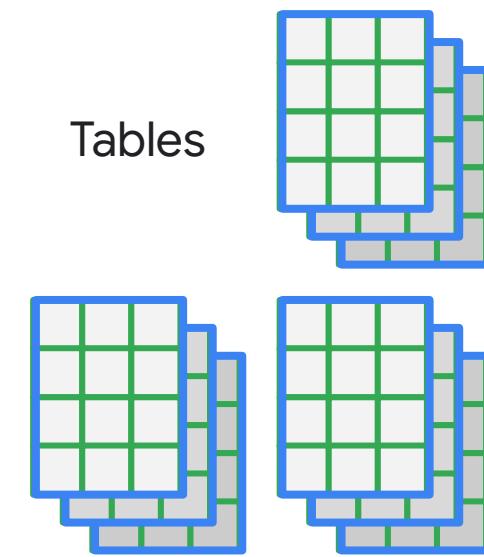
Performance Considerations



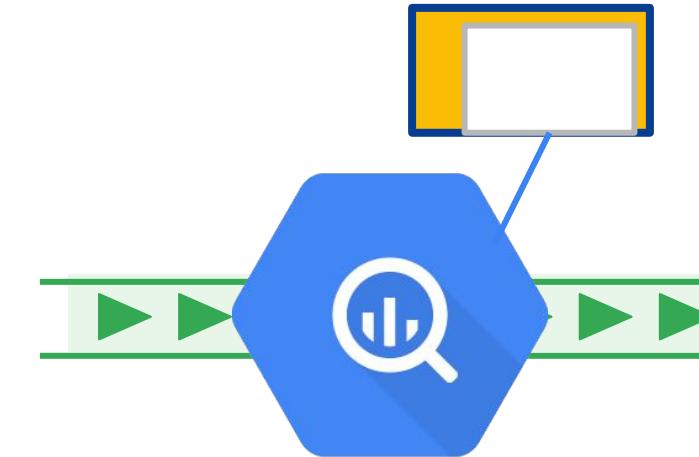
# Best practices for fast, smart, data-driven decisions



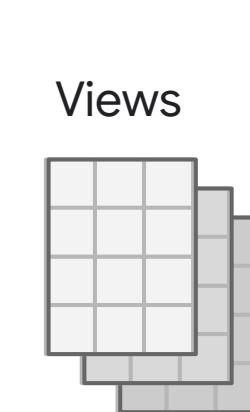
Use Dataflow to do the processing and data transformations



Create multiple tables for easy analysis



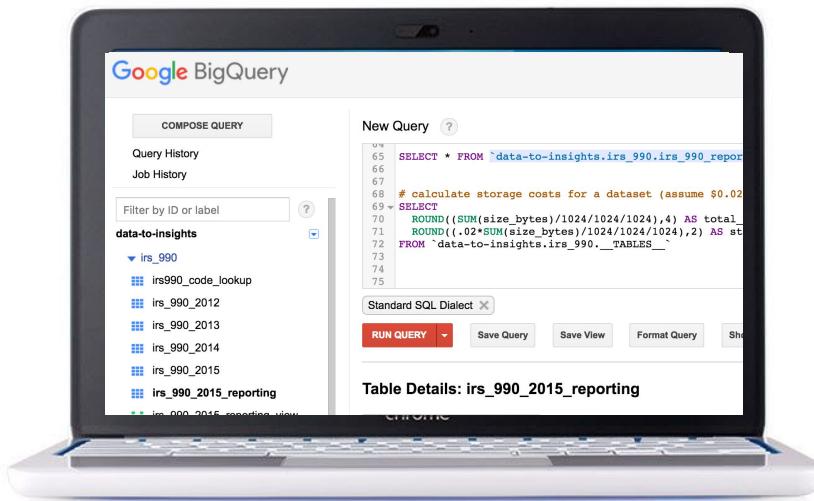
Use BigQuery for streaming analysis and dashboards



Store data in BigQuery for low cost long term storage

# Best practices for analyzing data with BigQuery

## Dataset



Know your data

## Questions

Give me a list of all organizations sorted by the revenue lowest to highest

## SQL

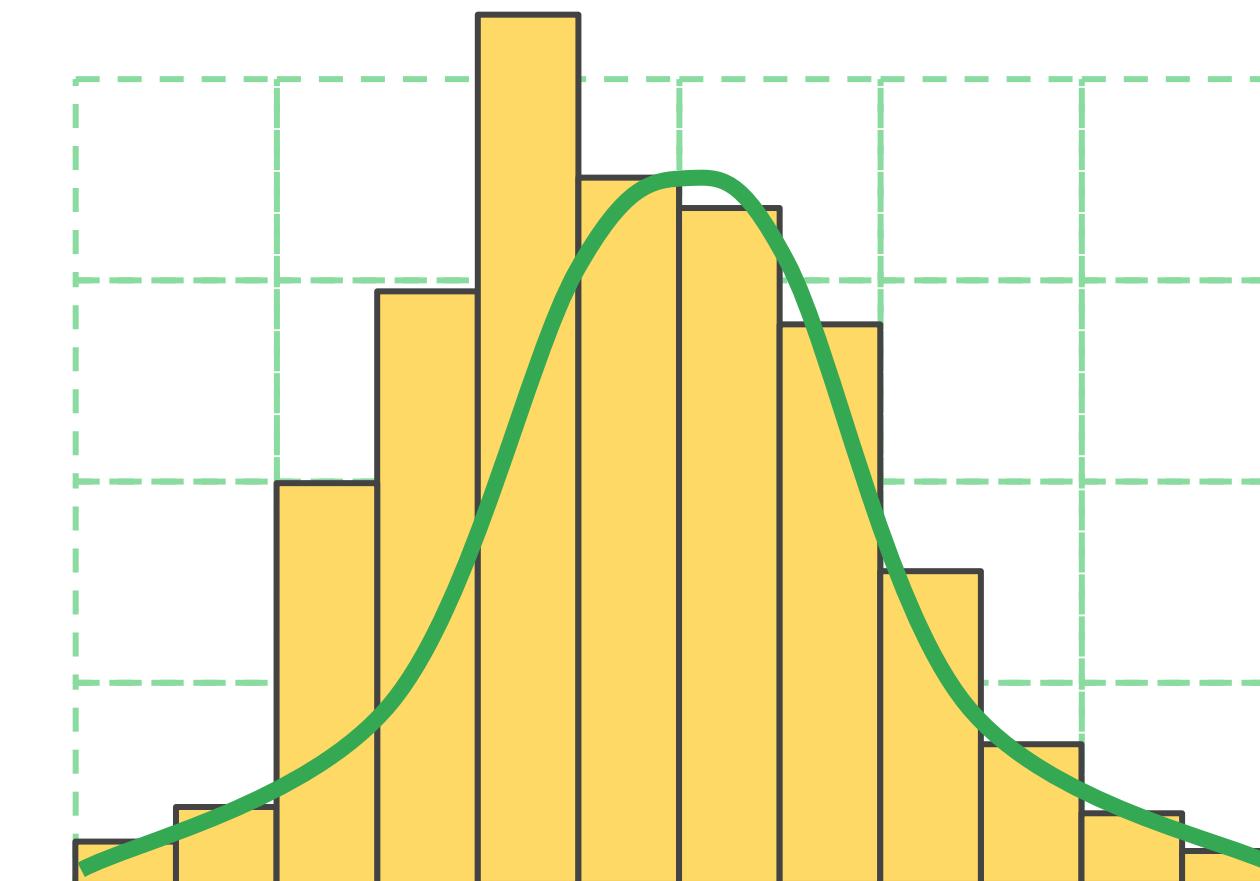
```
SELECT
  name,
  revenue
FROM dataset
ORDER BY revenue;
```

Use queries effectively

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

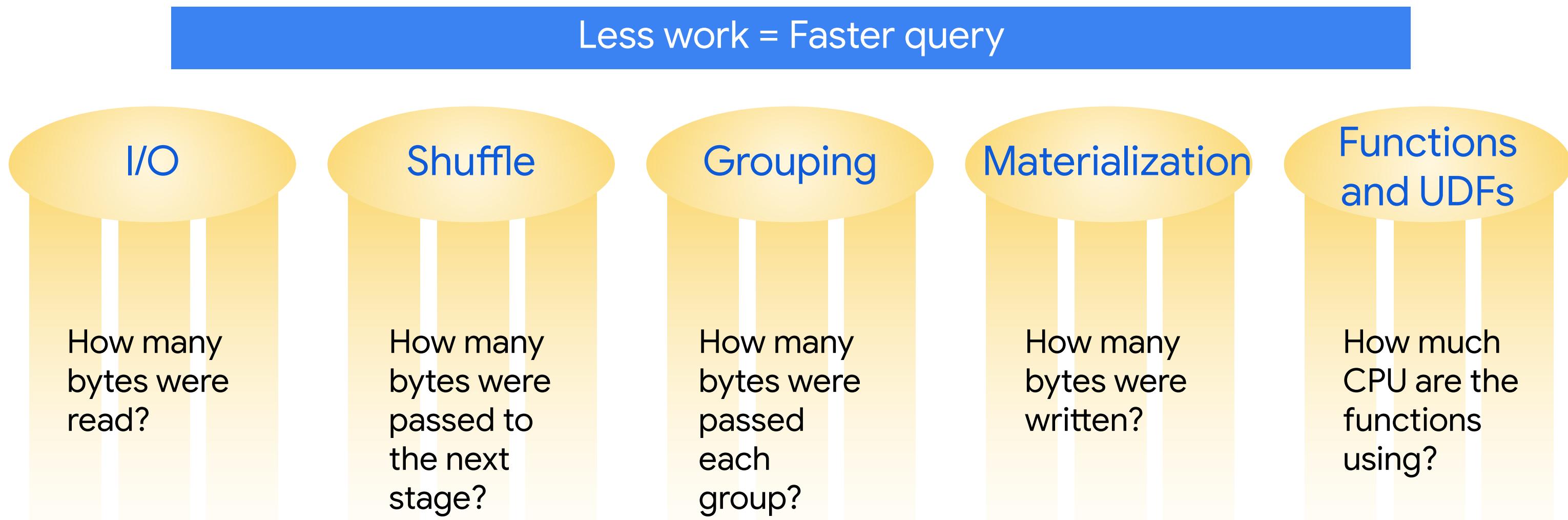
Customer	Street Number	Street Name	City	Country	Phone Number

Customer	Address	Phone Number



Stop accumulating work that could be done earlier.

# Improve scalability by improving efficiency



# Optimize BigQuery queries

~~SELECT \*~~

Avoid using unnecessary columns

WHERE

Filter early and often

JOIN

Put the largest table on the left

GROUP BY

Low cardinality is faster than high cardinality

APPROX\_COUNT\_DISTINCT

Is faster than

COUNT (DISTINCT)

ORDER

On the outermost query

\*

Use wildcards to query multiple tables

--time\_partitioning\_type

Time partitioning tables for easier search

# 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
```

# 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  
...
```

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

c1	c2	c3	eventDate	c5
			2019-01-01	
			2019-01-02	
			2019-01-03	
			2019-01-04	
			2019-01-05	

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

Partitioned tables

c1	userId	c3	eventDate	c5
			2019-01-01	
			2019-01-02	
			2019-01-03	
			2019-01-04	
			2019-01-05	

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

Clustered tables

# Set up clustering at table creation time

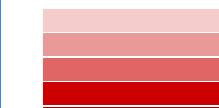
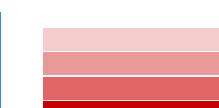
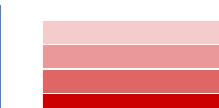
c1	userId	c3	eventDate	c5
			2019-01-01	
			2019-01-02	
			2019-01-03	
			2019-01-04	
			2019-01-05	

```
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
```

# 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)
```

Can force a recluster using DML on necessary partition

c1	userId	c3	eventDate	c5
			2019-01-01	
			2019-01-02	
			2019-01-03	
			2019-01-04	
			2019-01-05	

# BigQuery will automatically recluster your data

## Automatic re-clustering

**free**

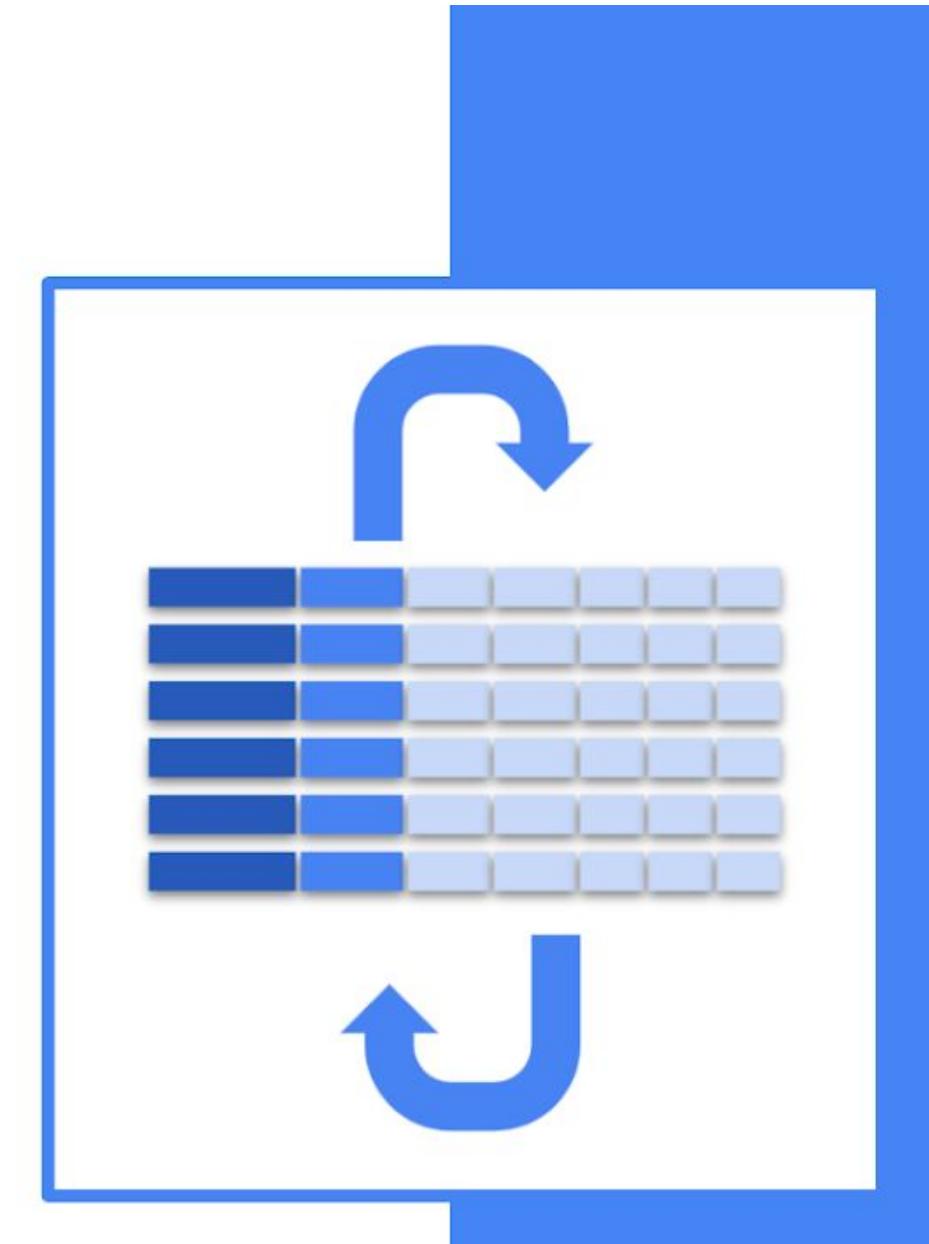
Doesn't consume your query resources

**maintenance-free**

Requires no setup or maintenance

**autonomous**

Automatically happens in the background



# 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).

# When to use clustering



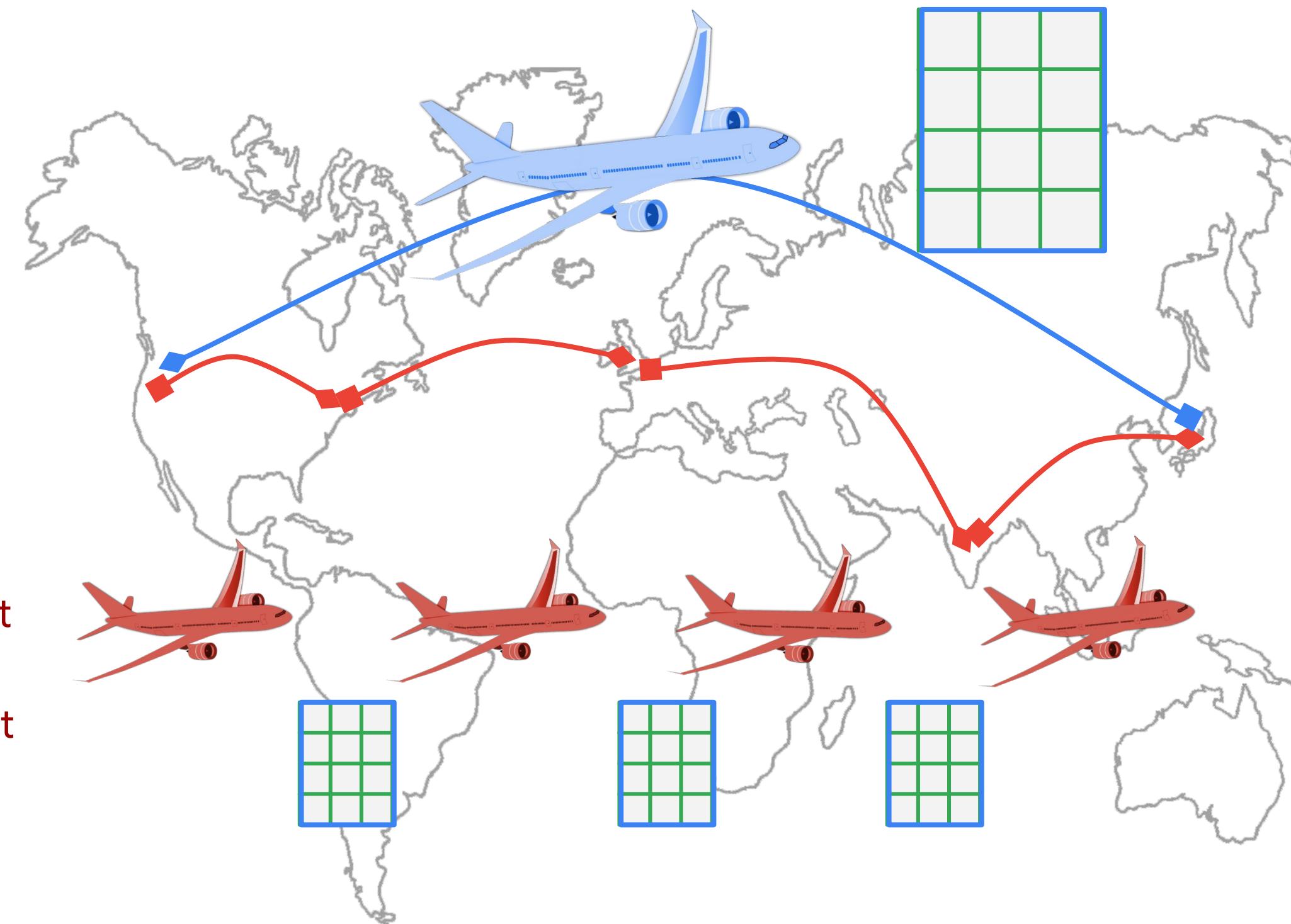
Your data is already partitioned on a DATE or TIMESTAMP or Integer Range



You commonly use filters or aggregation against particular columns in your queries.

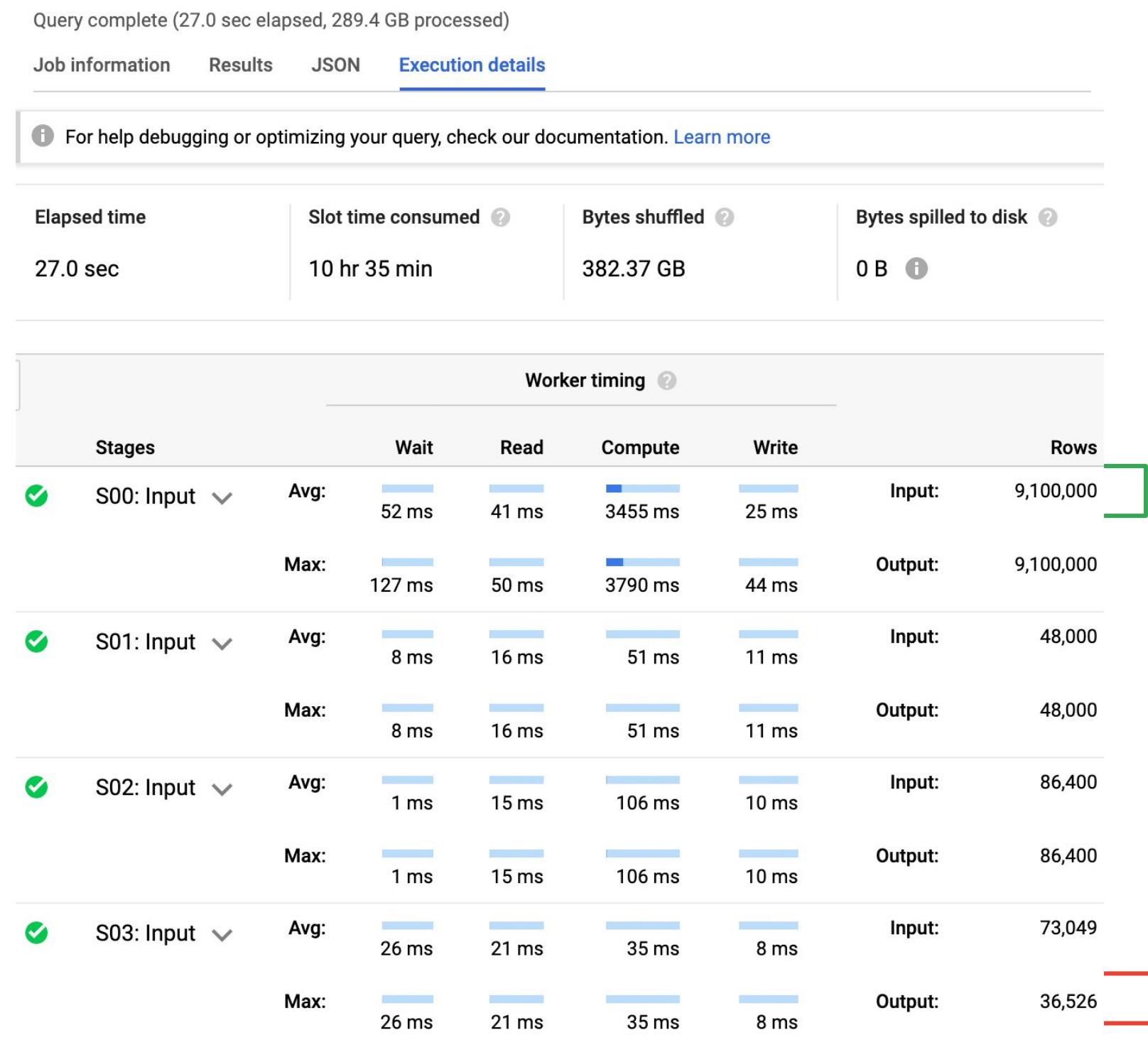
# Break queries into stages using intermediate tables

Read only what  
is needed and  
store results at  
each step.



Read a huge  
dataset and  
process it.

# Track input and output counts with Execution details tab



# Using BigQuery plans to optimize

Significant difference between avg and max time?

Probably data skew

Check with APPROX\_TOP\_COUNT

Filter early as a workaround

Most time spent reading during intermediate stages?

Order of operations?

Consider filtering earlier in the query

Most time spent on CPU tasks?

Probably slow code

Look carefully at UDFs

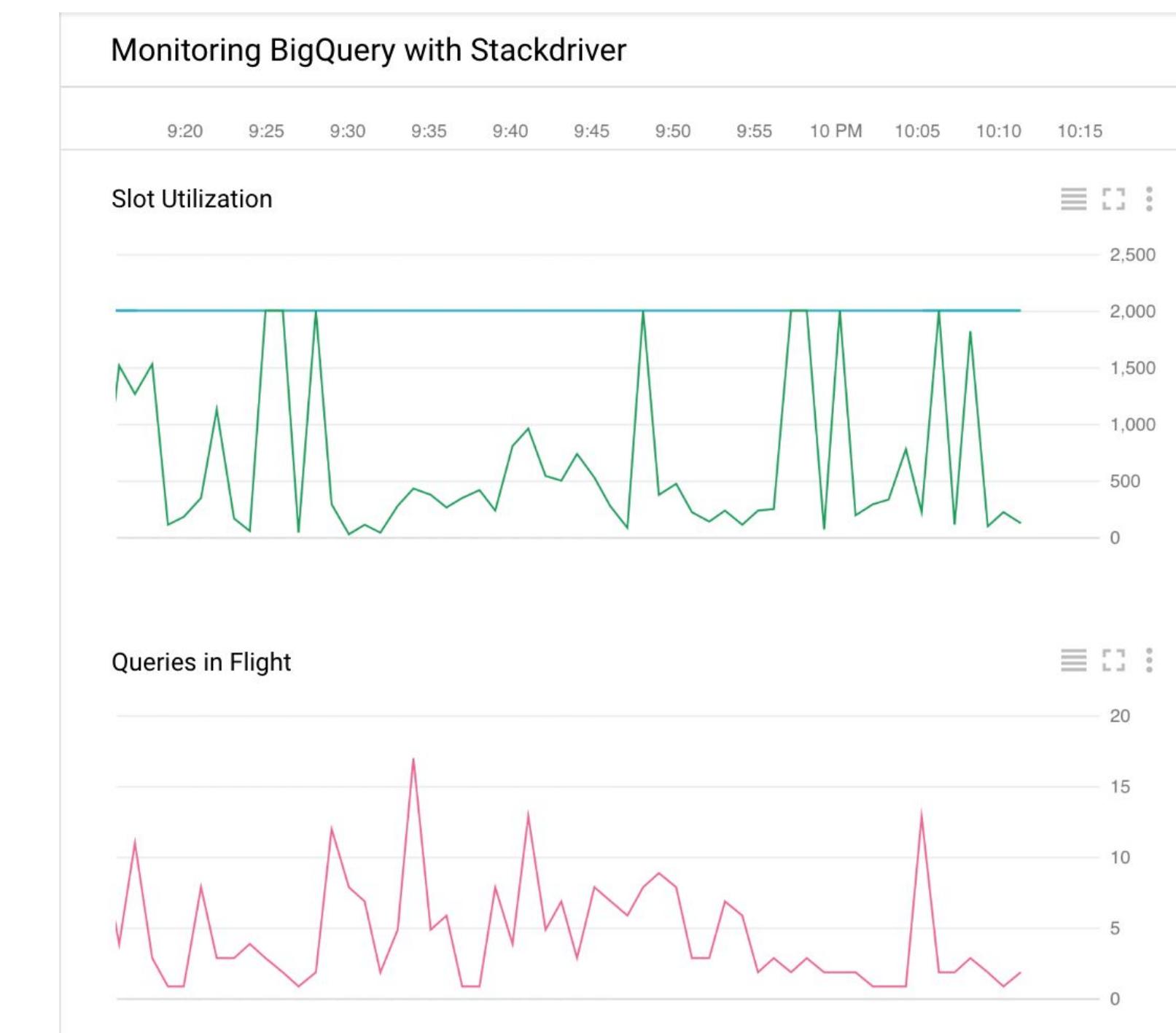
Use approximate functions  
Filter earlier in the query

# Analyze BigQuery performance in Stackdriver

## Custom Dashboards

Metrics include:

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





# Lab

---

## Optimizing your BigQuery Queries for Performance

### Objectives

- Use BigQuery to
  - Minimize I/O
  - Cache results of previous queries
  - Perform efficient joins
  - Avoid overwhelming single workers
  - Use approximate aggregation functions

# 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

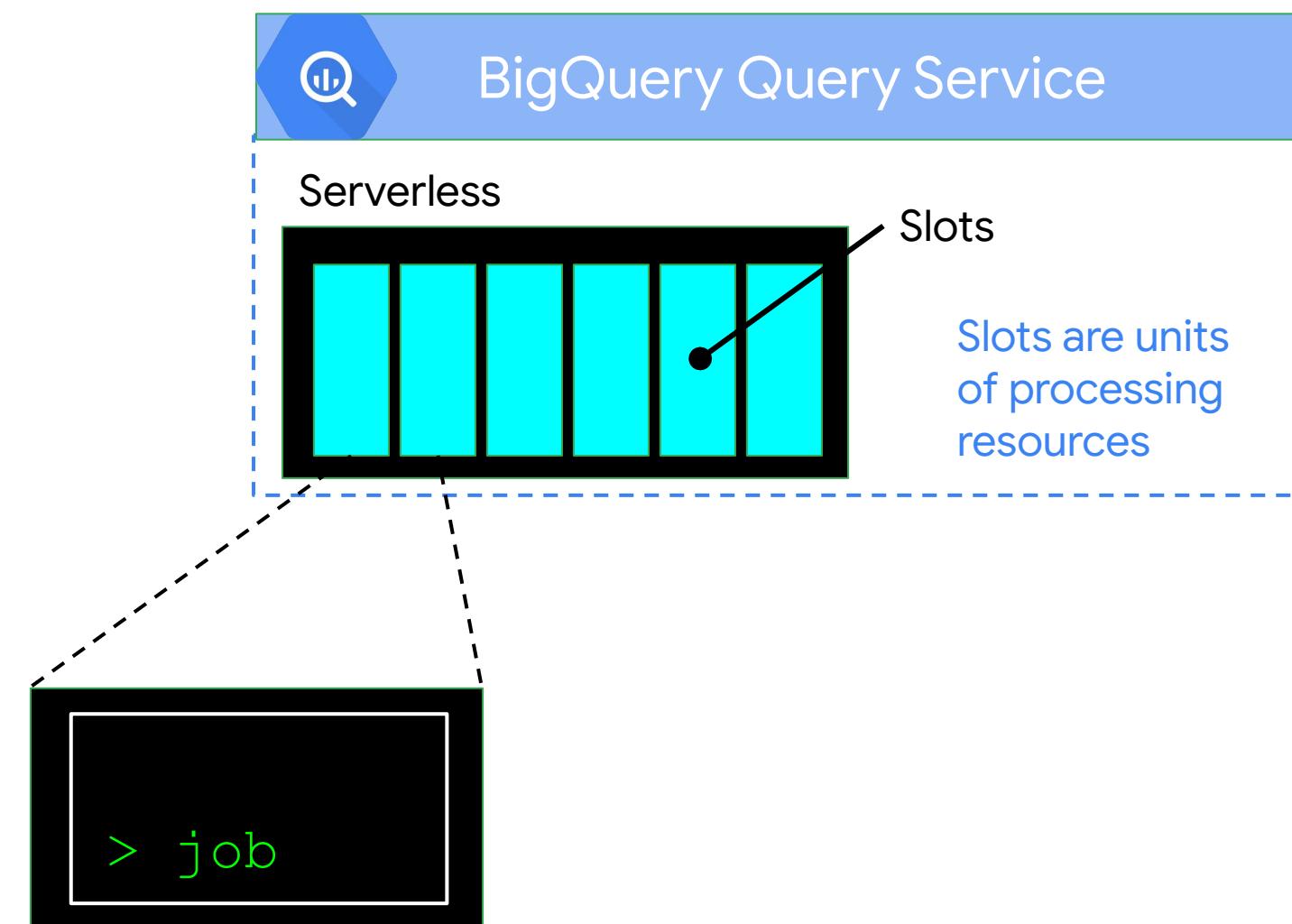
## Long-term Storage Pricing

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

# Which BigQuery pricing model to pick?

On-Demand Pricing	Flat-Rate Pricing
<ul style="list-style-type: none"><li>• \$5/TB of data processed</li><li>• Quota limit of 2,000 slots</li><li>• Slots shared amongst all on demand users</li><li>• 1st TB of data processed is free each month</li></ul>	<ul style="list-style-type: none"><li>• Fixed rate pricing is \$10k* per 500 slots per month</li><li>• Slots are dedicated 24/7</li><li>• Starting at 500 slots</li><li>• Unlimited use of slots</li></ul>
Good for <b>exploratory work</b> and discovery	Good for <b>multiple large workloads</b> and <b>consistent monthly billing</b>

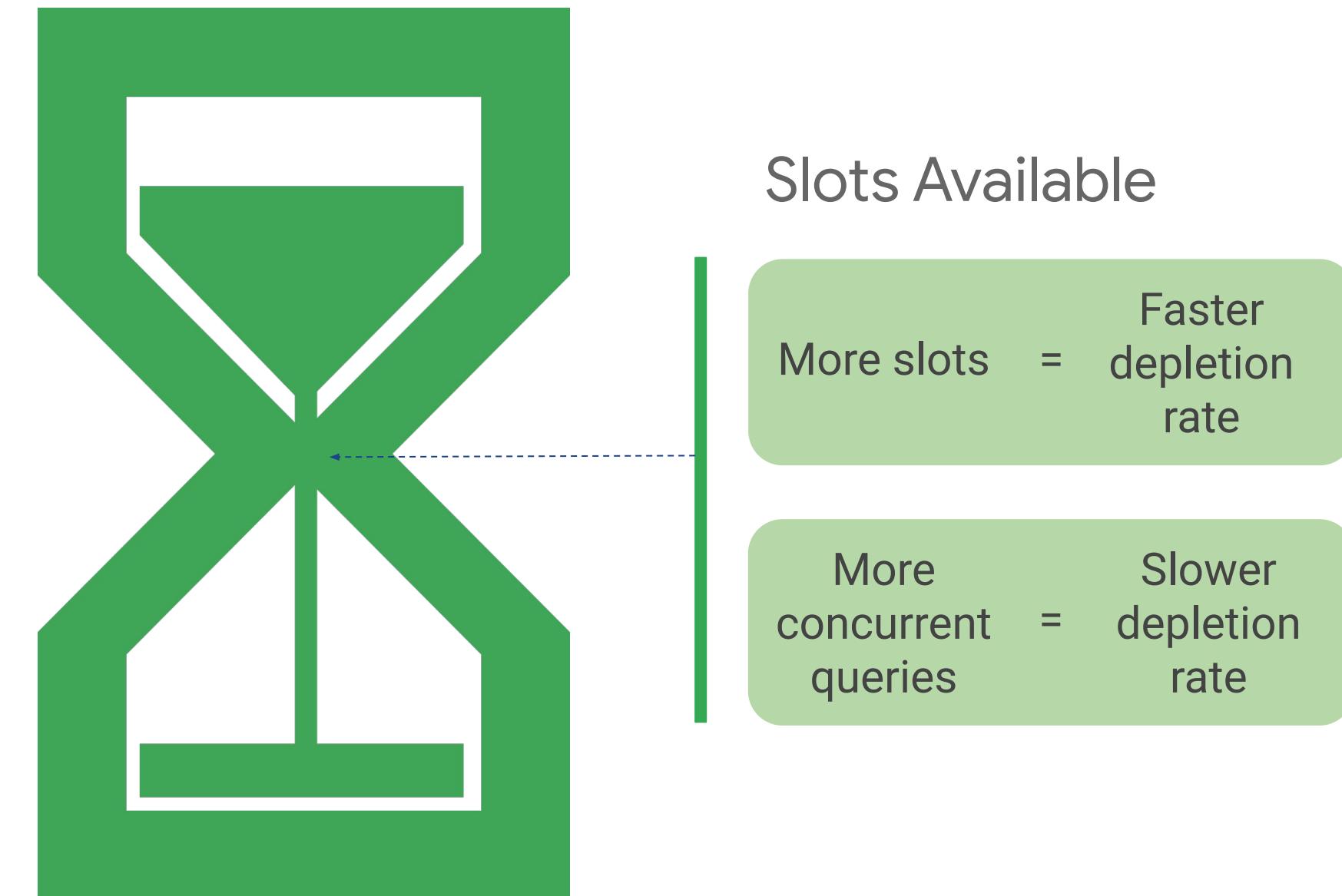
# Slots are units of resources that are consumed when a query is run



# Estimating the right BigQuery slots allocation is critical

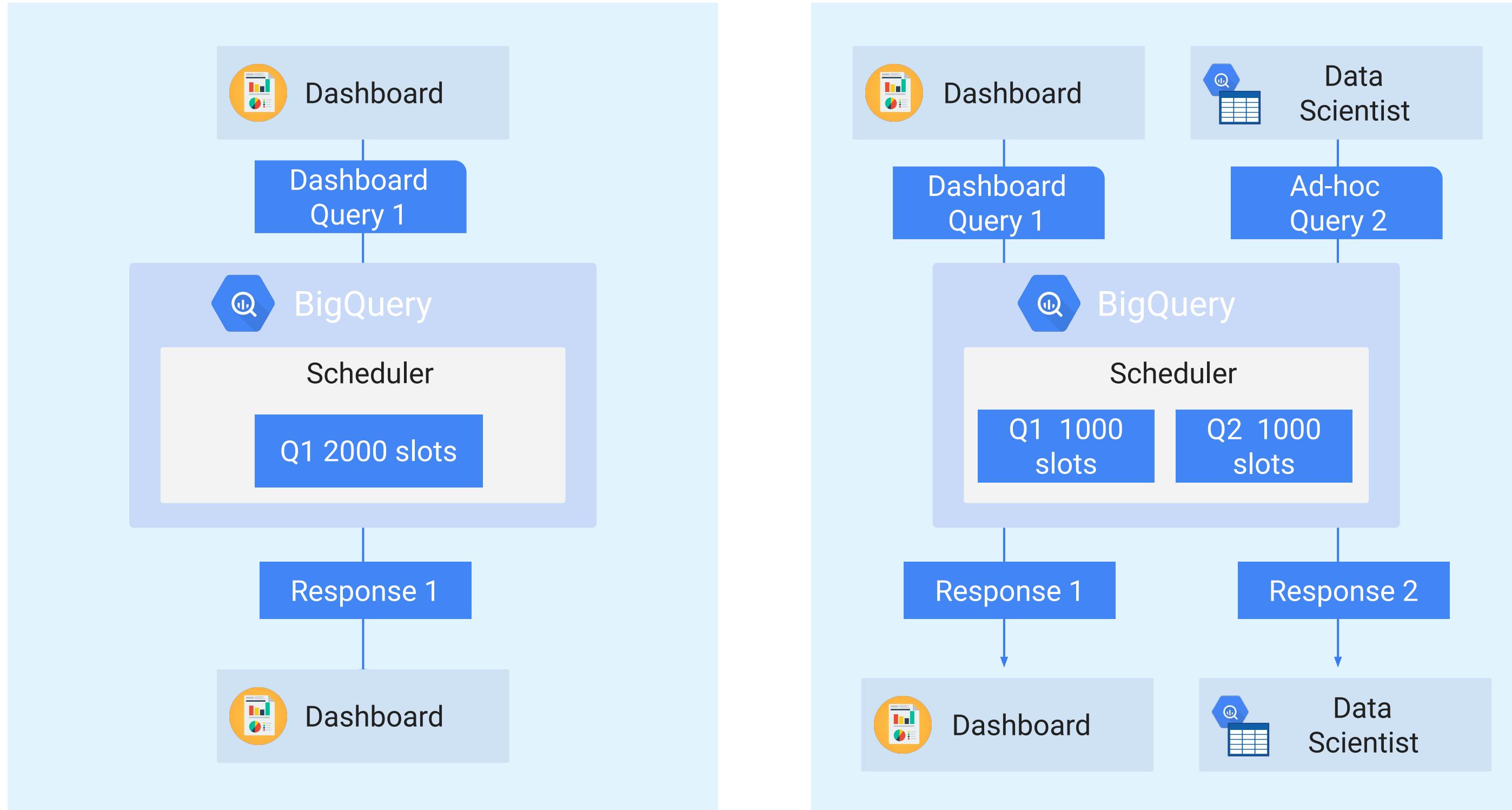
Guideline: It is recommended to plan for 2,000 BigQuery slots for every 50 medium-complexity queries simultaneously

50 queries = 2,000 Slots

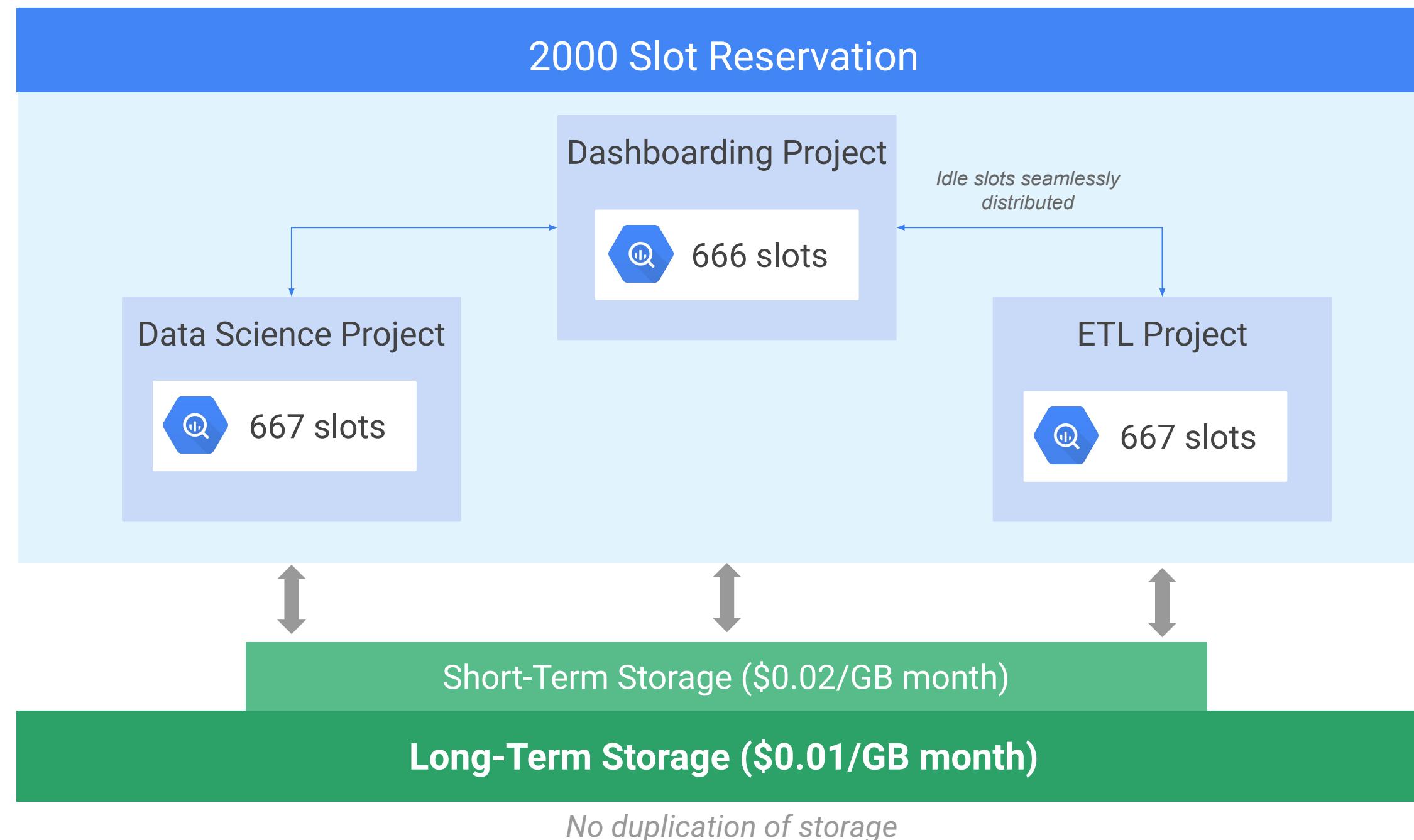


(quota: 100 concurrent queries, but this can be raised)

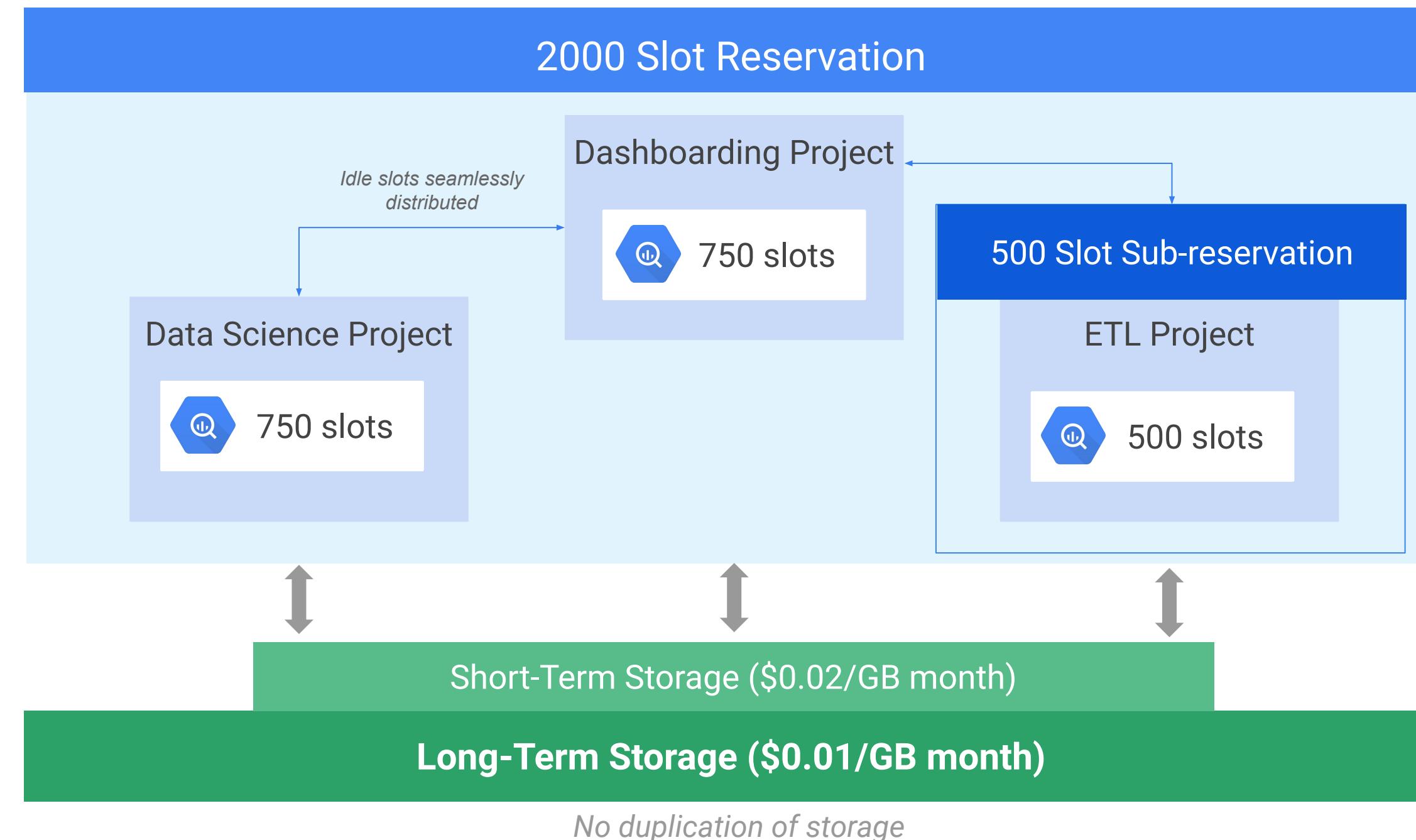
# BigQuery has a fair scheduler



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



# To prioritize projects, set up a hierarchical reservation





---

# Lab

## Creating Date-Partitioned Tables in BigQuery

### Objectives

- Query a partitioned dataset
- Create dataset partitions to improve query performance and reduce cost

# Module Summary

- Use BigQuery's analytic window functions for advanced analysis
- BigQuery has built in GIS functionality
- Use BigQuery's WITH clause to help modularize your queries
- There are many ways to optimize your query performance