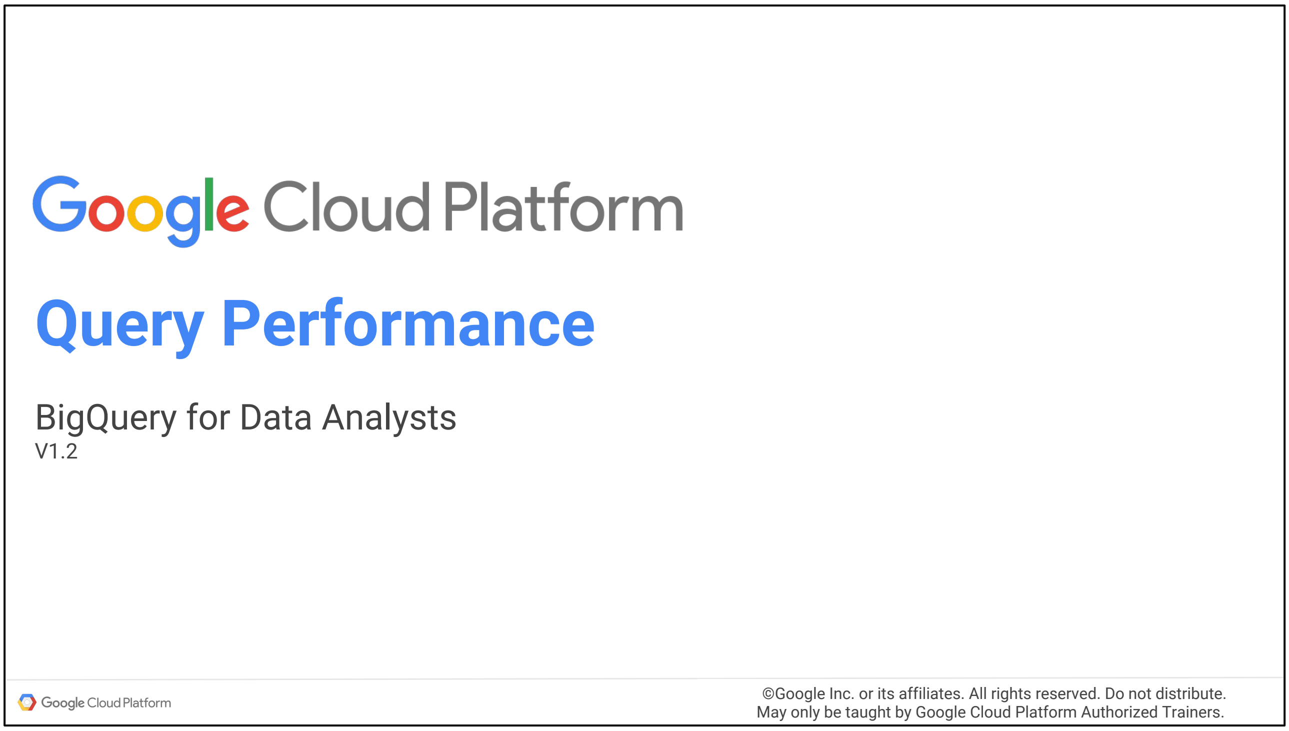


Google Cloud Platform

Query Performance

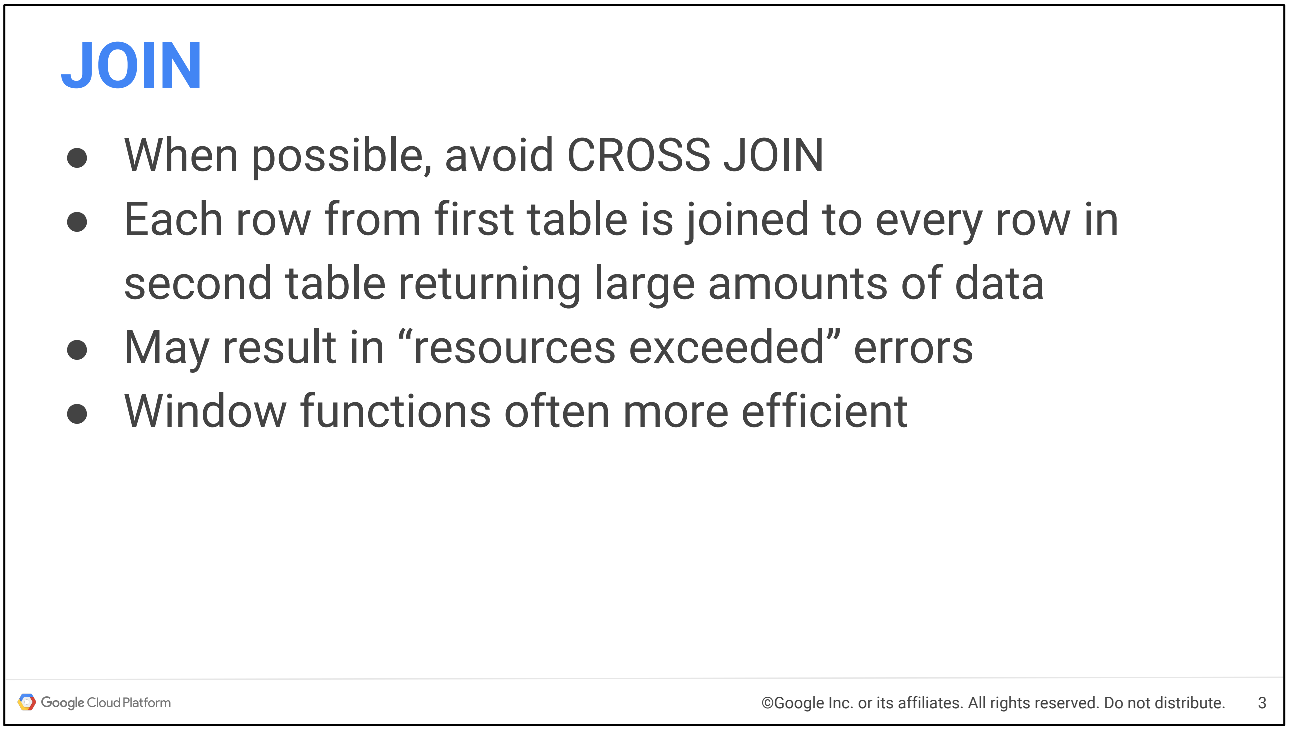
BigQuery for Data Analysts
V1.2

Google Cloud Platform

Approximate timing: 60 minutes

Agenda

1. JOIN and GROUP BY – How They Affect Performance
2. Table Decorators
3. Wildcards
4. Partitions
5. Query Performance Tips
6. Quiz & Lab



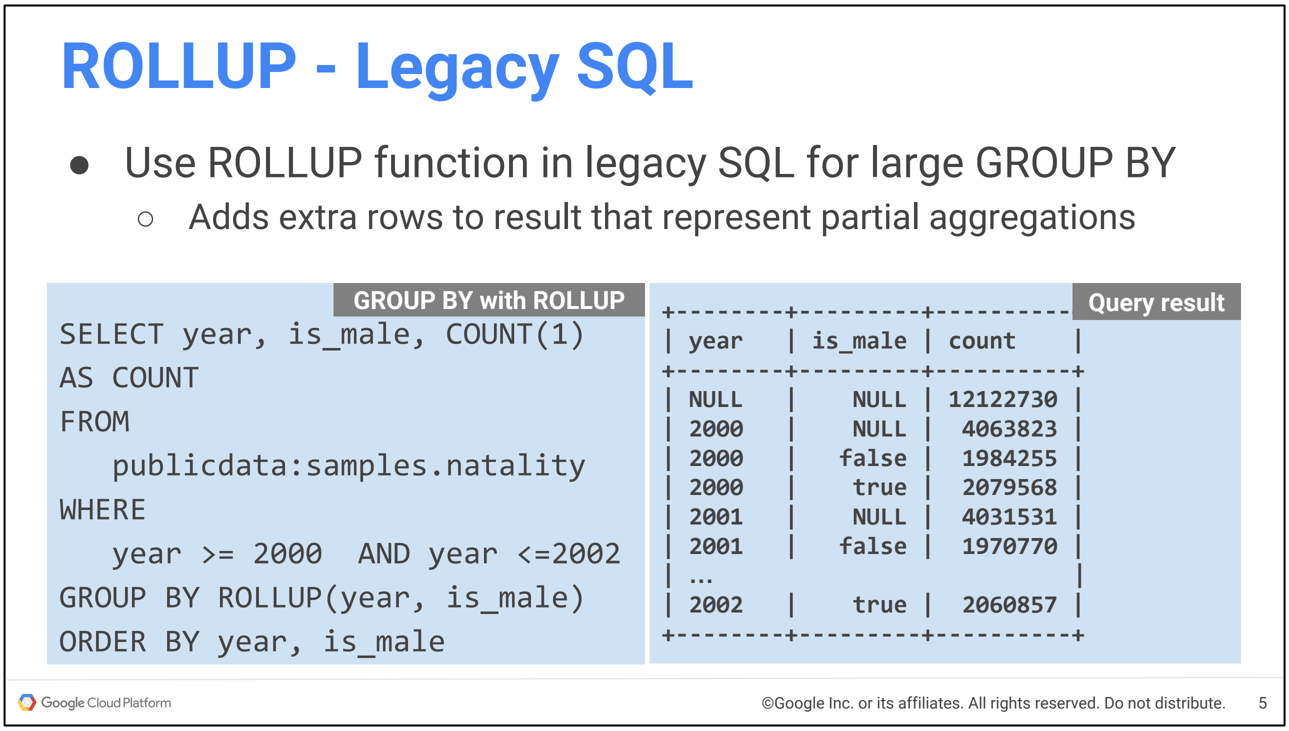
JOIN

- When possible, avoid CROSS JOIN
- Each row from first table is joined to every row in second table returning large amounts of data
- May result in “resources exceeded” errors
- Window functions often more efficient

Notes:
The 8MB right-side table join limit no longer applies.

GROUP BY

- Use GROUP BY when the number of distinct groups is small (low cardinality)
  - Aggregation of data performed in shards
  - Low cardinality means shards do not shuffle data
  - High performance
- Large GROUP BY is less optimal
  - High cardinality requires aggregation performed by multiple shards
  - Shards produce hash key for each value and shuffle data



ROLLUP - Legacy SQL

- Use ROLLUP function in legacy SQL for large GROUP BY
  - Adds extra rows to result that represent partial aggregations

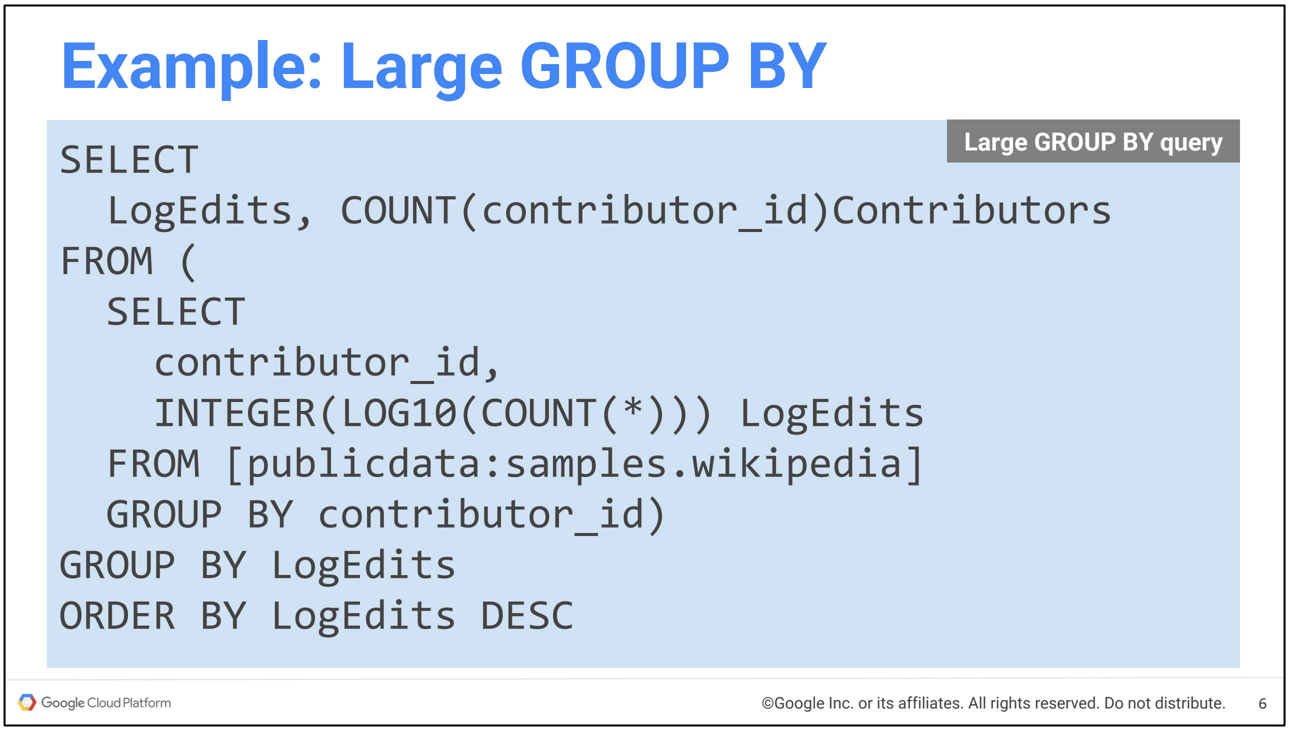
GROUP BY with ROLLUP

```
SELECT year, is_male, COUNT(1)
AS COUNT
FROM
   publicdata:samples.natality
WHERE
   year >= 2000  AND year <=2002
GROUP BY ROLLUP(year, is_male)
ORDER BY year, is_male
```

Query result

```
+--------+---------+----------+
| year   | is_male | count    |
+--------+---------+----------+
| NULL   |    NULL | 12122730 |
| 2000   |    NULL |  4063823 |
| 2000   |   false |  1984255 |
| 2000   |    true |  2079568 |
| 2001   |    NULL |  4031531 |
| 2001   |   false |  1970770 |
| …                           |
| 2002   |    true |  2060857 |
+--------+---------+----------+
```

Notes:
The fields in the GROUP BY must be in the SELECT (declares which columns to process).

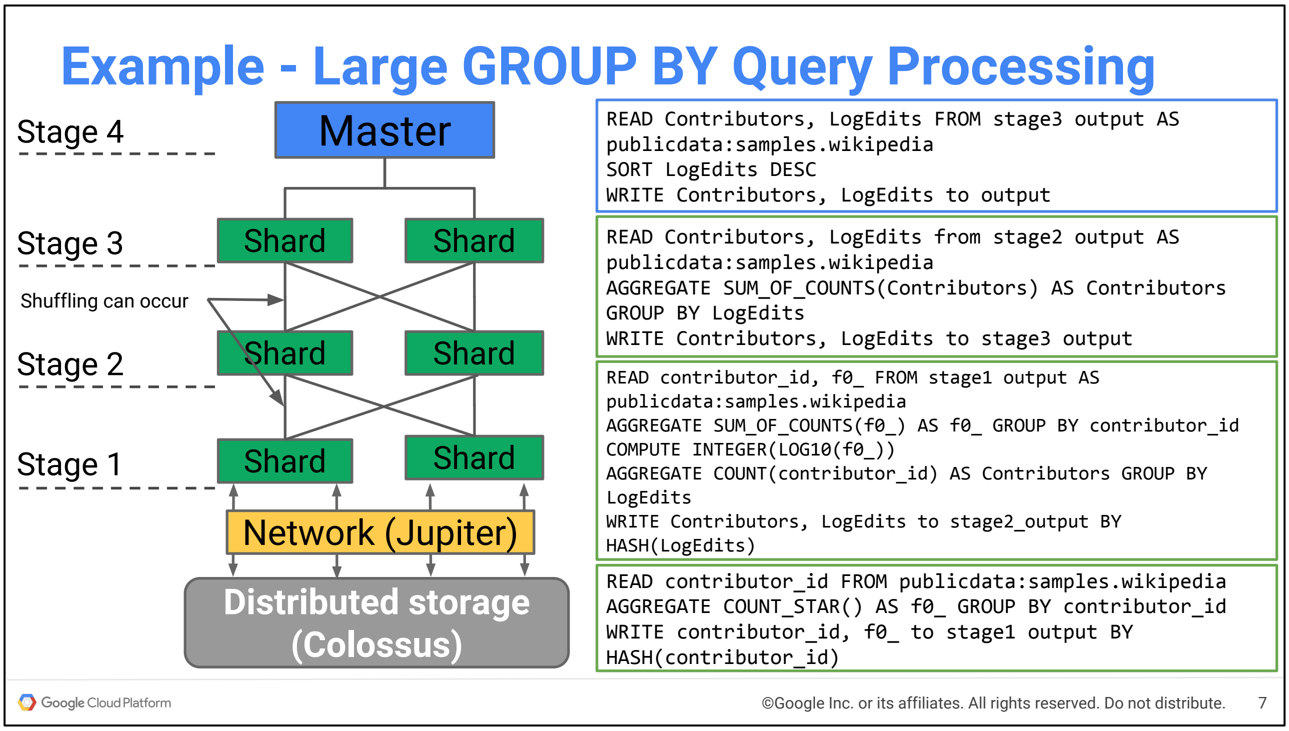


Example: Large GROUP BY

Large GROUP BY query

```
SELECT
  LogEdits, COUNT(contributor_id)Contributors
FROM (
  SELECT
    contributor_id,
    INTEGER(LOG10(COUNT(*))) LogEdits
  FROM [publicdata:samples.wikipedia]
  GROUP BY contributor_id)
GROUP BY LogEdits
ORDER BY LogEdits DESC
```

Notes:
This is a more complicated query to find out how many authors contribute to wikipedia by the number of edits that he/she made. The LOG10(COUNT(*)) is a way to broadly categorize the number of edits into buckets. For example, if a user contributed 120 edits, then INTEGER(LOG10(120)) will yield 2. Similarly, if a user contributed 185 edits, he will be grouped into the same bucket. If a user contributed 1500 edits, then INTEGER(LOG10(1500) will yield 3. So, the inner SELECT statement groups each contributor into buckets and the final SELECT statement counts the number of contributors for each bucket.



Example - Large GROUP BY Query Processing

```
READ Contributors, LogEdits FROM stage3 output AS
publicdata:samples.wikipedia
SORT LogEdits DESC
WRITE Contributors, LogEdits to output
```

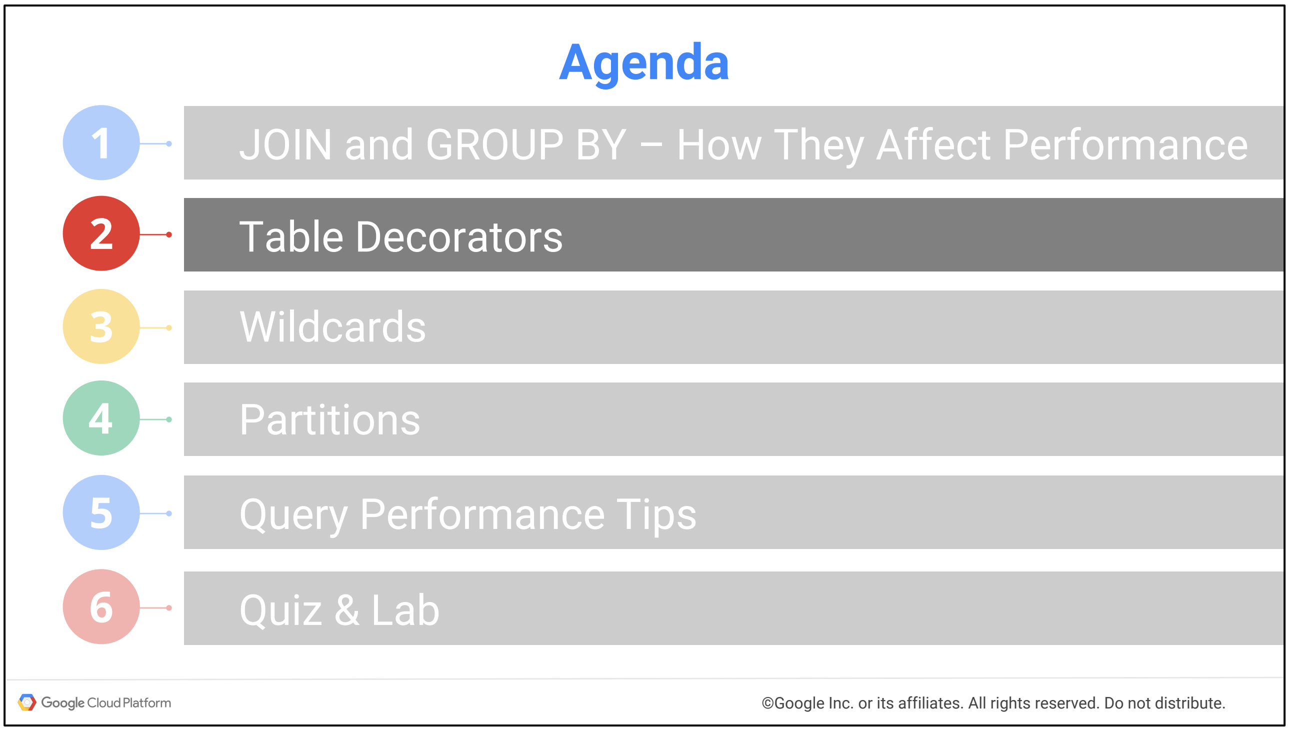
```
READ Contributors, LogEdits from stage2 output AS
publicdata:samples.wikipedia
AGGREGATE SUM_OF_COUNTS(Contributors) AS Contributors
GROUP BY LogEdits
WRITE Contributors, LogEdits to stage3 output
```

```
READ contributor_id, f0_ FROM stage1 output AS
publicdata:samples.wikipedia
AGGREGATE SUM_OF_COUNTS(f0_) AS f0_ GROUP BY contributor_id
COMPUTE INTEGER(LOG10(f0_))
AGGREGATE COUNT(contributor_id) AS Contributors GROUP BY
LogEdits
WRITE Contributors, LogEdits to stage2_output BY
HASH(LogEdits)
```

```
READ contributor_id FROM publicdata:samples.wikipedia
AGGREGATE COUNT_STAR() AS f0_ GROUP BY contributor_id
WRITE contributor_id, f0_ to stage1 output BY
HASH(contributor_id)
```

Google Cloud Platform

Notes:
Because the number of contributors to Wikipedia is huge, the INNER SELECT used ‘GROUP BY’. The large GROUP BY triggered shuffling between stages. Shuffling can be viewed as ‘partitioning’. Shuffling guarantees that all records which have the same value will be stored in the same shard. This allows those operations to be performed efficiently. In the example, each contributor will be shuffled to the same shard. Assuming the distribution is quite even, the implication is that each shard will have fewer contributors to process, hence less memory consumption.



Agenda

1. JOIN and GROUP BY – How They Affect Performance
2. **Table Decorators**
3. Wildcards
4. Partitions
5. Query Performance Tips
6. Quiz & Lab

BigQuery Table Decorators

- Use to perform the most cost-effective query of a subset of your data
- Table decorators can be used whenever a table is read
  - Copying a table, exporting a table, or listing data
- Can also be used to undelete a table within 2 days on a best-effort basis
- Currently supported in legacy SQL only

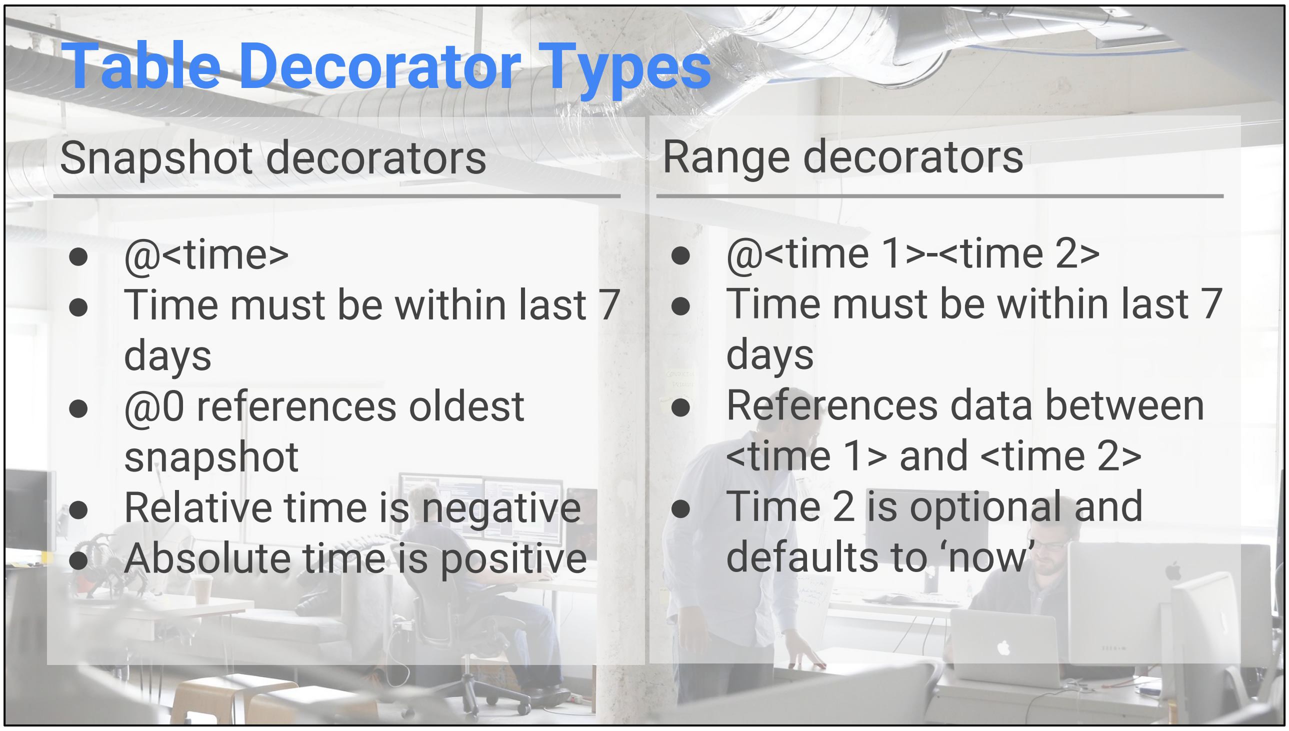


Table Decorator Types

Snapshot decorators

- @<time>
- Time must be within last 7 days
- @0 references oldest snapshot
- Relative time is negative
- Absolute time is positive

Range decorators

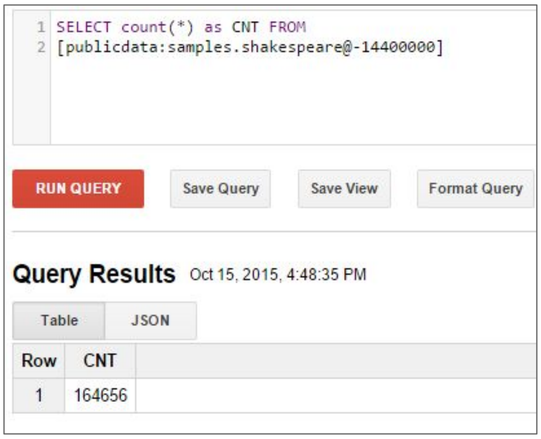
- @<time 1>-<time 2>
- Time must be within last 7 days
- References data between <time 1> and <time 2>
- Time 2 is optional and defaults to ‘now’

Example: Snapshot Table Decorator

- **@-14400000** - is a reference to a snapshot of the table at -14400000 milliseconds since the current time

Snapshot decorator - Legacy SQL

```
SELECT count(*)FROM
[publicdata:samples.shakespe
are@-14400000]
```

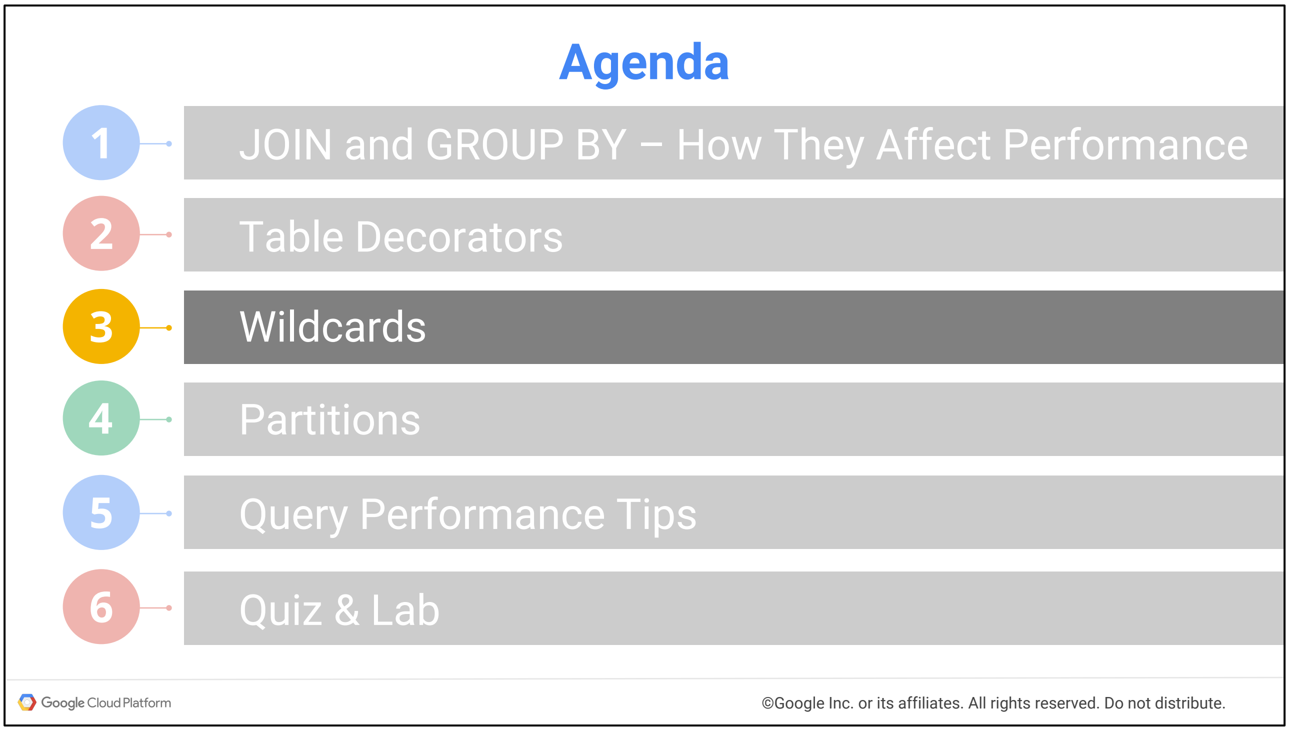


```
SELECT count(*) as CNT FROM
[publicdata:samples.shakespeare@-14400000]
```

Query Results Oct 15, 2015, 4:48:35 PM

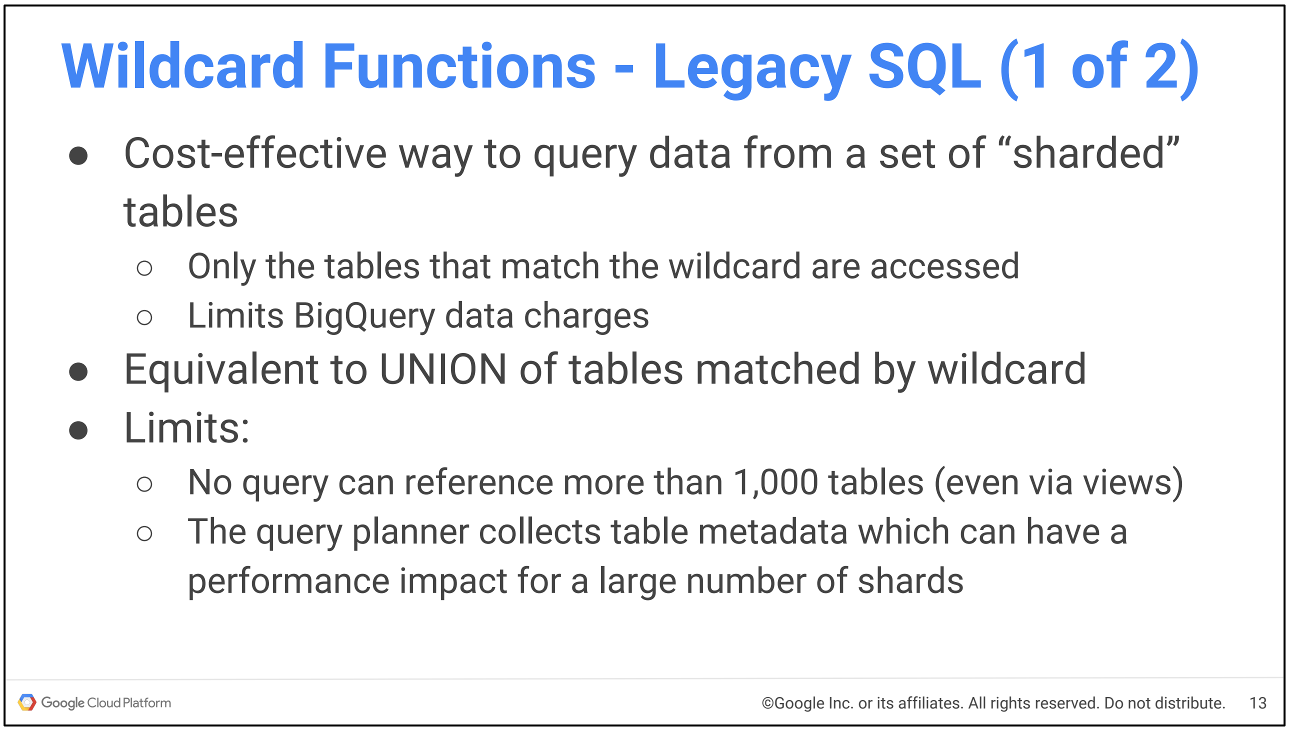
Row	CNT
1	164656

Notes:
The %-s flag can be used with table decorators to remove streaming data from the response.



Agenda

Google Cloud Platform



Wildcard Functions - Legacy SQL (1 of 2)

- Cost-effective way to query data from a set of “sharded” tables
    - Only the tables that match the wildcard are accessed
    - Limits BigQuery data charges
- Equivalent to UNION of tables matched by wildcard
- Limits:
    - No query can reference more than 1,000 tables (even via views)
    - The query planner collects table metadata which can have a performance impact for a large number of shards

Notes:
For more information on table wildcard functions, see:
https://cloud.google.com/bigquery/query-reference?hl=en#tablewildcardfunctions.

Wildcard Functions - Legacy SQL (1 of 2)

Function	Description
`TABLE_DATE_RANGE(`*`prefix`*`,` *`timestamp1`*`,` *`timestamp2`*`)`	Queries daily tables that overlap with the time range between <*timestamp1*> and <*timestamp2*>. Table names must have the following format: <*prefix*><*day*>, where <*day*> is in the format YYYYMMDD. You can use date and time functions to generate the timestamp parameters. For example: ● TIMESTAMP('2012-10-01 02:03:04') ● DATE_ADD(CURRENT_TIMESTAMP(), -7, 'DAY')
`TABLE_DATE_RANGE_STRICT(`*`prefix`*`,` *`timestamp1`*`,` *`timestamp2`*`)`	This function is equivalent to TABLE_DATE_RANGE. The only difference is that if any daily table is missing in the sequence, TABLE_DATE_RANGE_STRICT fails and returns a Not Found: Table <*table_name*> error.
`TABLE_QUERY(`*`dataset`*`,` *`expr`*`)`	Queries tables whose names match the supplied *expr*. The *expr* parameter must be represented as a string and must contain an expression to evaluate. For example, 'length(table_id) < 3'.

Wildcard Function Examples (1 of 2)

- Wildcards can be used with conditional logic
  - Pulls data from tables that match condition
  - Tables need to contain ‘common’ columns
  - Table_Query must contain an expression to evaluate
- Assuming the following tables: `mydata.partsales`, `mydata.laborsales`, `mydata.partnersales`:

Wildcard example

```
SELECT month, SUM(sales), SUM(discount)
FROM (TABLE_QUERY(mydata,’table_id CONTAINS “sales” and
length(table_id < 10’))
GROUP BY month
```

Wildcard Function Examples (2 of 2)

log_20150101
ip_address
browser
start_time
end_time
URL
return_code

log_20150102
ip_address
browser
start_time
end_time
URL
return_code

log_20150103
ip_address
browser
start_time
end_time
URL
return_code

Wildcard example

```
SELECT …
FROM
(TABLE_DATE_RANGE(dataset.log,TIMESTAMP(‘2015-01-01’),TIMESTAMP
(‘2015-01-03’))
```

Wildcard Tables - Standard SQL (1 of 2)

- Query multiple tables using concise SQL statements
- Wildcard table represents union of all tables that match the wildcard expression (like wildcard functions)
- Useful when dataset contains multiple, similarly named tables with compatible schemas
- Each row in wildcard table contains special column containing value matched by wildcard character

Notes:
For more information on wildcard tables in standard SQL, see:
https://cloud.google.com/bigquery/docs/wildcard-tables.

Wildcard Tables - Standard SQL (2 of 2)

- Example:

  `` FROM `bigquery-public-data.noaa_gsod.gsod*` ``

  - Matches all tables in `noaa_gsod` that begin with string 'gsod'
  - ` character is required (single, double quotes are invalid)
- Longer prefixes generally perform better than shorter prefixes
  - For example: `.gsod200*` versus `.*`

Agenda

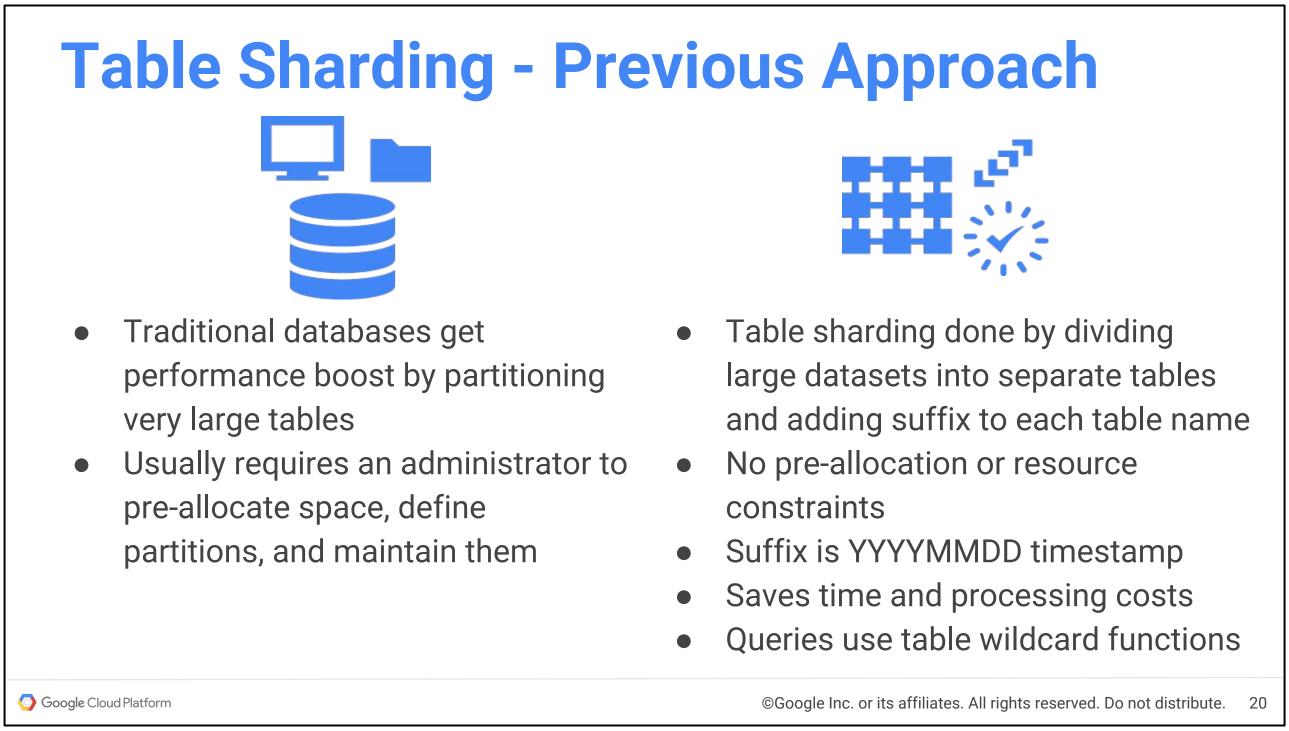


Table Sharding - Previous Approach

- Traditional databases get performance boost by partitioning very large tables
- Usually requires an administrator to pre-allocate space, define partitions, and maintain them

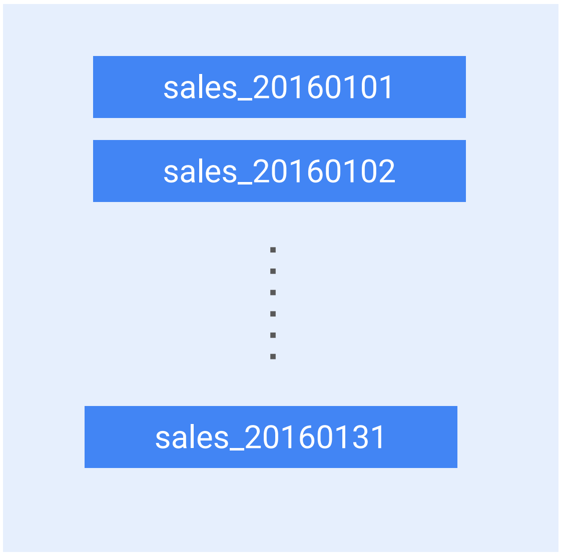
- Table sharding done by dividing large datasets into separate tables and adding suffix to each table name
- No pre-allocation or resource constraints
- Suffix is YYYYMMDD timestamp
- Saves time and processing costs
- Queries use table wildcard functions

Notes:
Dividing a dataset into daily tables helped to reduce the amount of data scanned when querying a specific date range. For example, if you have a a year's worth of data in a single table, a query that involves the last seven days of data still requires a full scan of the entire table to determine which data to return. However, if your table is divided into daily tables, you can restrict the query to the seven most recent daily tables.

Daily tables, however, have several disadvantages. You must manually, or programmatically, create the daily tables. SQL queries are often more complex because your data can be spread across hundreds of tables. Performance degrades as the number of referenced tables increases. There is also a limit of 1,000 tables that can be referenced in a single query.

Example - Sharding

```
SELECT ...
FROM TABLE_DATE_RANGE(sales_,
  DATE(“20160101”),
  DATE(“20160131”))
```



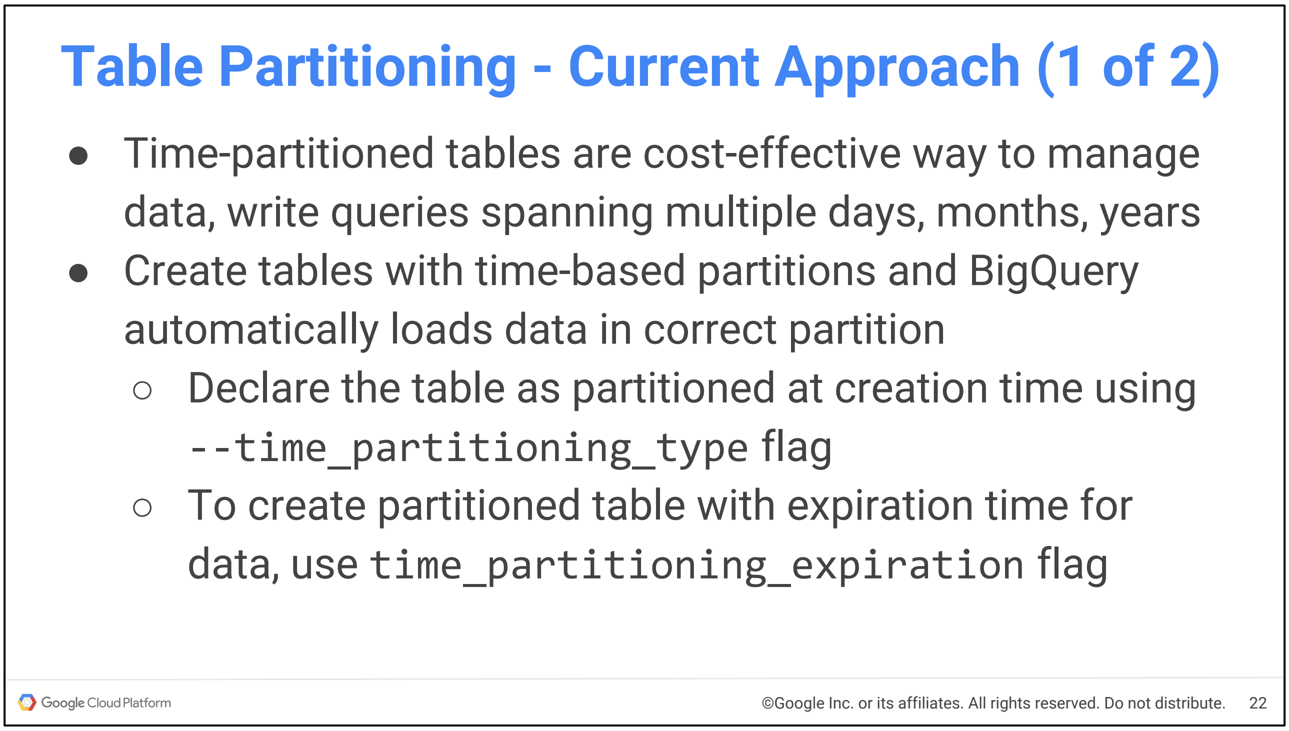


Table Partitioning - Current Approach (1 of 2)

- Time-partitioned tables are cost-effective way to manage data, write queries spanning multiple days, months, years
- Create tables with time-based partitions and BigQuery automatically loads data in correct partition
  - Declare the table as partitioned at creation time using `--time_partitioning_type` flag
  - To create partitioned table with expiration time for data, use `time_partitioning_expiration` flag

Google Cloud Platform

Notes:
Partitioned tables include a pseudo column named _PARTITIONTIME that contains a date-based timestamp for data loaded into the table. The timestamp is based on UTC time and represents the number of microseconds since the unix epoch. For example, if data is appended to a table on April 15, 2016, all of the rows of data appended on that day contain the value TIMESTAMP("2016-04-15") in the _PARTITIONTIME column.

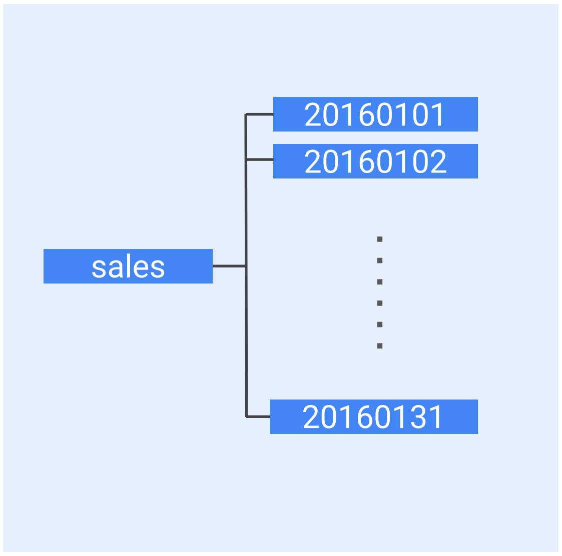
Table Partitioning - Current Approach (2 of 2)

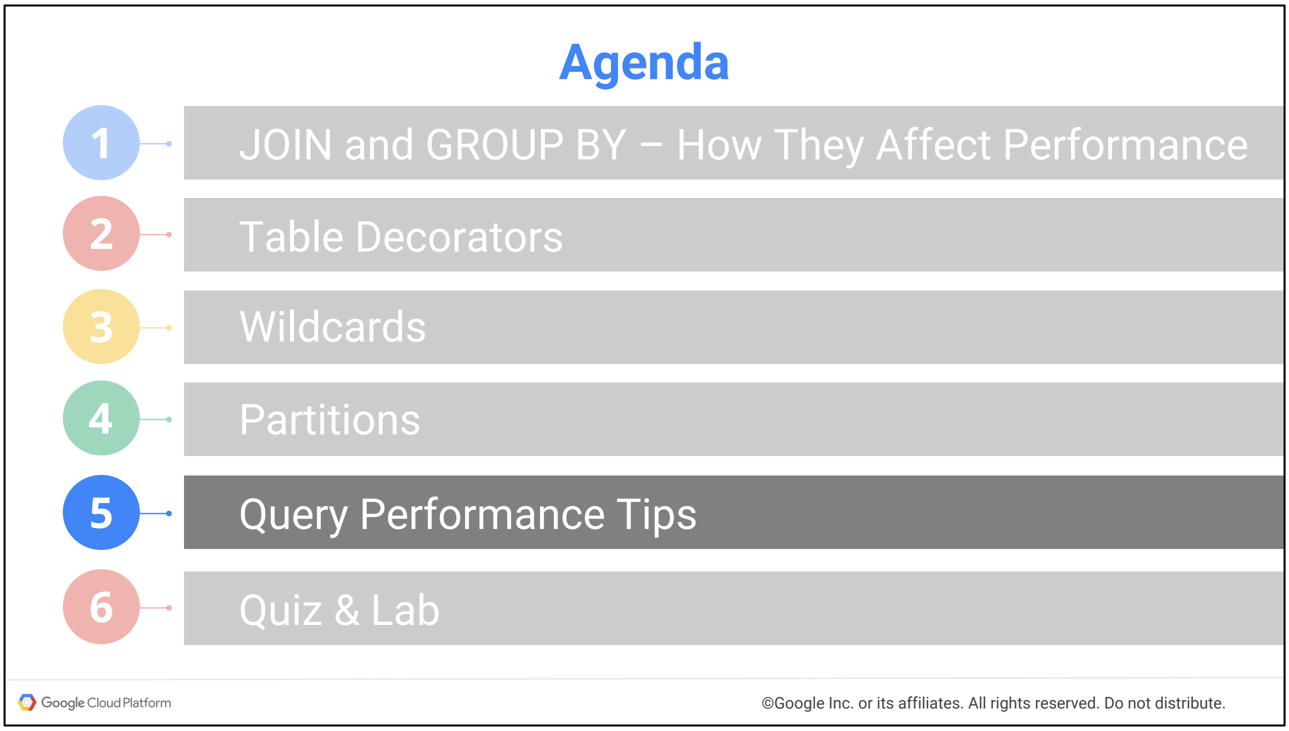
- To query partitioned table, provide date or range of dates and query processes data for interval specified
- Only data scanned is in partitions specified by interval
- Queries are more performant, cheaper
- Currently only supported by legacy SQL

Notes:
For information on table partitioning best practices, see:
https://cloud.google.com/bigquery/docs/partitioned-tables#best_practices.

Example - Table Partitioning

```
SELECT ...
FROM sales
WHERE _PARTITIONTIME
BETWEEN TIMESTAMP(“20160101”)
    AND TIMESTAMP(“20160131”)
```





Agenda

1. JOIN and GROUP BY – How They Affect Performance
2. Table Decorators
3. Wildcards
4. Partitions
5. **Query Performance Tips**
6. Quiz & Lab

Query Performance Tips (1 of 3)

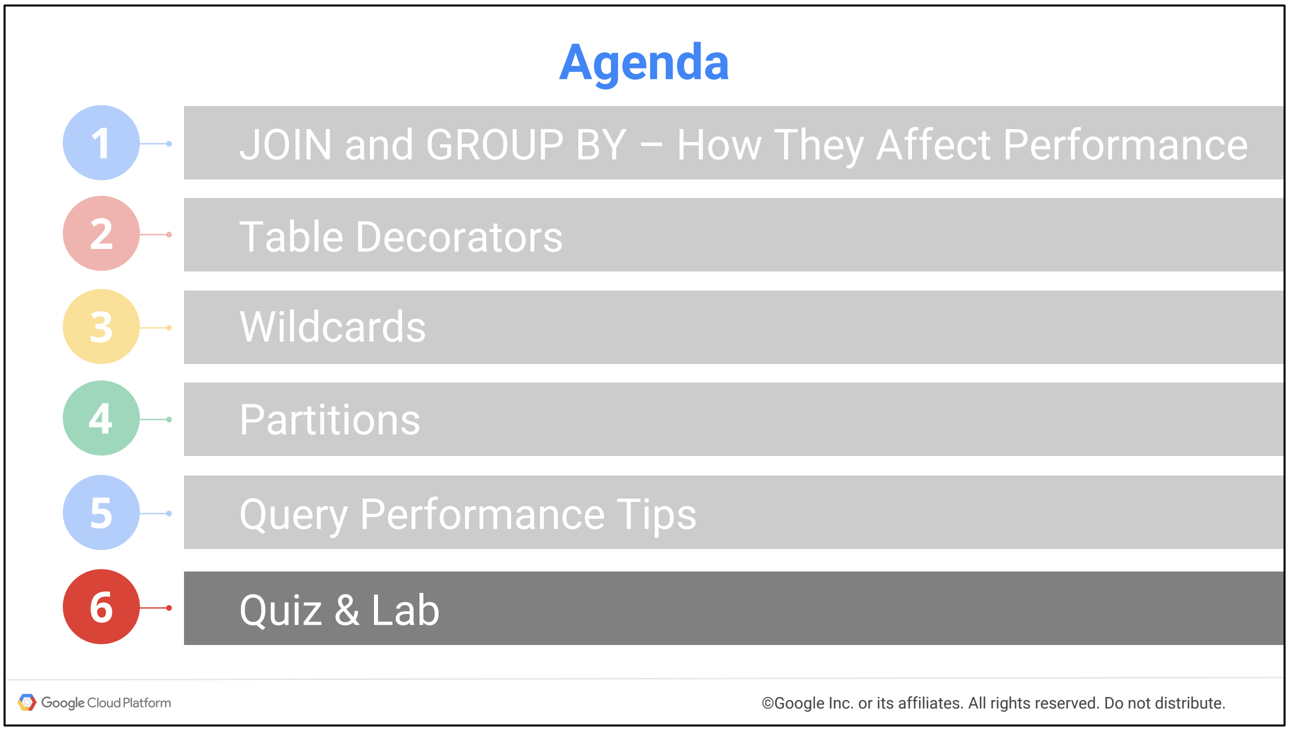
- Denormalize tables for performance
- Select only needed columns - Do not use `Select *`
- Schedule batch queries at off-peak hours using jobs
- Use caching when possible
  - Caching is best effort
  - If table data changes, cache is invalidated
  - Use `jobs.getQueryResults` to page through cached query results in a temporary table (no charge)

Query Performance Tips (2 of 3)

- Try to use ORDER BY and LIMIT in outermost queries
  - LIMIT is applied to results by Master
- Build queries from the inside out by using subqueries
  - Filter data in subqueries
  - Perform arithmetic, ordering, case logic in outer query
- Use queries to create materialized intermediate tables
  - Create subset of complex data in destination table
  - Partially aggregate data in destination table

Query Performance Tips (3 of 3)

- Move heavyweight filters, such as `regexp`, to the end
- Avoid grouping on unbounded possible values
  - Example: Web logs with arbitrary GET parameters in the suffix
- Consider using IF/CASE instead of self-joins because IF/CASE has lower processing overhead
  - Self-joins require multiple disk reads
- Apply WHERE filters prior to JOINs
  - Predicate pushdown



Agenda

1. JOIN and GROUP BY – How They Affect Performance
2. Table Decorators
3. Wildcards
4. Partitions
5. Query Performance Tips
6. Quiz & Lab

Module Review (1 of 2)

Which of the following statements are true?
*(select **2** of the available options)*

- ❑ In legacy SQL, you can use table decorators to retrieve data from a deleted table
- ❑ A “broadcast” table join invokes a shuffler operation
- ❑ A file in distributed storage contains data for only one column
- ❑ Shuffling is always invoked when a GROUP BY is used
- ❑ Table wildcards must have a date-stamp in the table name

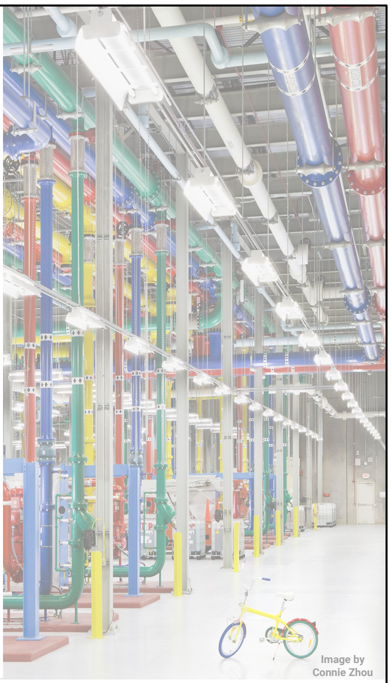
Module Review (2 of 2)

Which query clauses would cause shuffling to occur?
*(select **2** of the available options)*

- ❑ ORDER BY
- ❑ GROUP BY with few distinct values
- ❑ CROSS JOIN
- ❑ GROUP BY with many distinct values

Lab

BigQuery best practices and optimization techniques



Resources

- Release notes
  https://cloud.google.com/bigquery/release-notes
- Launch checklist for BigQuery
  https://cloud.google.com/bigquery/launch-checklist

Module Review Answers (1 of 2)

Which of the following statements are true?
*(select **2** of the available options)*

- ✓ In legacy SQL, you can use table decorators to retrieve data from a deleted table
- ❑ A “broadcast” table join invokes a shuffler operation
- ✓ A file in distributed storage contains data for only one column
- ❑ Shuffling is always invoked when a GROUP BY is used
- ❑ Table wildcards must have a date-stamp in the table name

Module Review Answers (2 of 2)

Which query clauses would cause shuffling to occur?
*(select **2** of the available options)*

- ❑ ORDER BY
- ❑ GROUP BY with few distinct values
- ✓ CROSS JOIN
- ✓ GROUP BY with many distinct values

cloud.google.com