

# Improving Read Performance in BigQuery

05

# Topics

01	BigQuery's Cache
02	Materialized Views
03	BI Engine
04	High Throughput Reads
05	BigQuery Storage Read API




# Topics

06


Considerations for External Data Sources




# To view only a few rows, use the Preview feature




Run







Save query




Save view




Schedule query





More



all\_sessions

Schema

Details

Preview

Row	fullVisitorId	channelGrouping	time	country	city
1	6897161498212019070	Social	17044	Canada	Toronto
2	9309061623954862081	Referral	3452427	United States	Mountain View
3	2759894513130601779	Direct	328842	United States	Mountain View
4	7057542698176321878	Referral	885958	United States	Seattle
5	4660977972422099228	Referral	796702	United States	Mountain View



## BigQuery's Cache

# If you run the exact same query twice, you will get the advantage of query cache...

```
SELECT
  fullVisitorId,
  country,
  timeOnSite
FROM
  `data-to-insights.ecommerce.all_sessions`
LIMIT 10
```

Query complete (0.1 sec elapsed, cached)

# Unless you have non-deterministic elements

```
SELECT
  current_timestamp(),
  fullVisitorId,
  country,
  timeOnSite
FROM
  `data-to-insights.ecommerce.all_sessions`
LIMIT 10
```

# Query cache

## Hash of

- Data modification times
- Tables used
- Query string

## Cache skipped if

- Referenced tables or views have changed
- Non-deterministic function used (e.g., NOW())
- Permanent result table requested
- Source tables have streaming buffers

## Hash becomes output table name

- Cache is per-user

### Query settings

#### Query engine

- ☒ BigQuery engine
- ☐ Cloud Dataflow engine  
Deploy your data processing pipelines on the Cloud Dataflow service.

#### Destination

☐ Set a destination table for query results

Project name

danny-bq

Dataset name

big\_query\_testing

Table name

Letters, numbers, and underscores allowed

#### Advanced options

#### Resource management

Job priority ?

- ☒ Interactive
- ☐ Batch

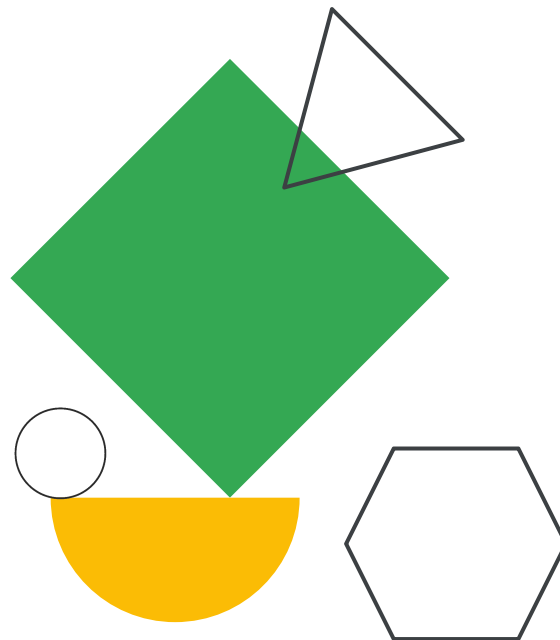
Cache preference ?

☒ Use cached results



# Demo

Using BigQuery's cache and  
temporary tables





# Materialized Views

# What are materialized views?

- Materialized views are precomputed views that periodically cache the results of a query for increased performance and efficiency.
- Materialized views can be queried directly or can be used by the BigQuery optimizer to process queries to the base tables.
  - Queries that use materialized views are generally faster and consume fewer resources.
- Materialized views are updated regularly to ensure fresh results.

# Using materialized views

base\_table

date	user	item	amount
20180801	Tom	Toaster	40
20180801	Polly	Toaster	60
20180802	Bob	Kettle	30
20180804	Wendy	Kettle	10

1 Create materialized view.

materialized\_view

date	item	amt
20180801	Toaster	100
20180802	Kettle	30
20180804	Kettle	10

2

By default, queries continue to go to the base table unaffected.

3

Applicable queries are rerouted automatically to materialized view to achieve faster execution.



# Why use materialized views in BigQuery?

- No maintenance:
  - Materialized views are updated automatically when the base table changes.
  - No user action is required!
  
- Data is always fresh:
  - If there is a change to the base table that may invalidate the materialized view, BigQuery will read directly from the base table.
  - Otherwise, BigQuery will only read the changes from the base table.

# Why use materialized views in BigQuery?

- Smart tuning:
  - If any part of a query against the source table can be resolved by querying the materialized view, BigQuery reroutes the query to use the materialized view.
- Cost estimation using dry run:
  - A dry run repeats query rewrite logic using the available materialized views and provides a cost estimate.
  - This can be used to test whether a specific query uses any materialized views.

# Use materialized views for reporting in BigQuery

- Improve query efficiency (slots usage and bytes processed) and reduce the execution time for complex queries with aggregate functions.
- Automatically improve a query execution plan.
- Inherit the same benefits of resilience and high availability of BigQuery tables.
- Provide real-time aggregation.
- Analytical queries against large volumes of data become more performant.

# Creating materialized views in BigQuery

Use the **CREATE MATERIALIZED VIEW** DDL statement to create materialized views.

```
CREATE MATERIALIZED VIEW movielens_demo.recent_movie_ratings
AS
(
SELECT
  movieId,
  AVG(rating) AS avg_rating,
  COUNT(rating) AS no_rating
FROM `movielens_demo.ratings`
WHERE timestamp > 874355425 # 1997/9/15 20:30:25 UTC
GROUP BY movieId
)
```



# Leverage materialized views for analysis

Materialized views can be queried directly.

```
SELECT
  movieId,
  avg_rating,
  no_rating
FROM `movielens_demo.recent_movie_ratings`
WHERE movieId = 3114 # Toy Story 2
```

Elapsed time

302 ms

Slot time consumed ?

29 ms

Bytes shuffled ?

61 B

Bytes spilled to disk ?

0 B ?

# Leverage materialized views for analysis

BigQuery can leverage materialized views in queries that don't directly reference the materialized view in some cases.

```
SELECT
  movieId,
  AVG(rating) AS avg_rating
FROM `movielens_demo.ratings`
WHERE timestamp > 874355425 AND movieId = 3114
GROUP BY movieId
```

Elapsed time

292 ms

Slot time consumed ?

64 ms

Bytes shuffled ?

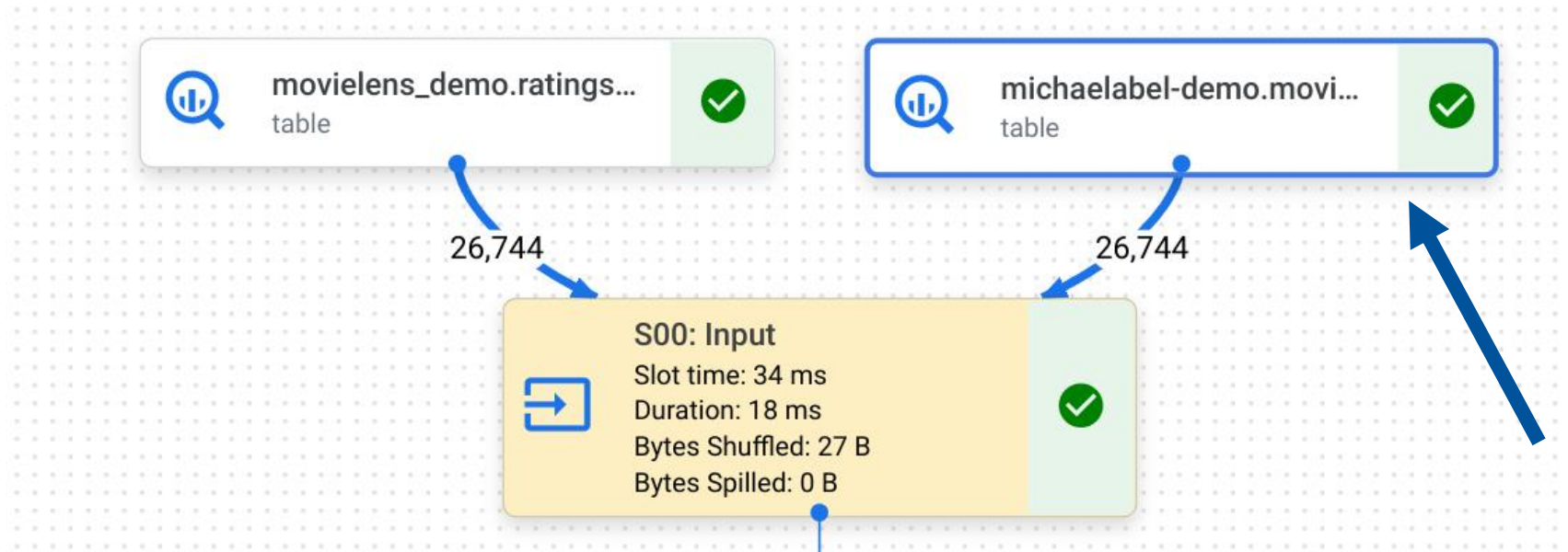
44 B

Bytes spilled to disk ?

0 B ?

# Leverage materialized views for analysis

BigQuery can leverage materialized views in queries that don't directly reference the materialized view in some cases.



# Additional features for materialized views

- WITH statements (Common Table Expressions) are supported.
- You can use both partitioning and clustering with materialized views.
  - A partition column for a materialized view must be a partition column for the base table.
  - Any non-aggregate column of the materialized view can be used for clustering.

# Limitations of materialized views

- Computing or filtering based on an aggregated value is not supported.
- Only certain aggregations are currently supported (see documentation for list).
- Materialized views cannot be nested on other materialized views, nor can they query external tables.
- The following operations are not supported (see documentation for complete list):
  - Outer JOINS and self JOINS
  - Analytic functions
  - ARRAY subqueries
  - Non-deterministic functions and user-defined functions

For more information, see:

<https://cloud.google.com/bigquery/docs/materialized-views-create>

# Considerations of using materialized views

- Put the largest or most frequently changing table first:
  - Materialized views with joins support incremental queries and refresh when the first or leftmost table in the query is appended.
- Avoid joining on clustering keys:
  - For selective queries (e.g., when joining on a clustering key), BigQuery is often already able to perform the join efficiently.

# Managing materialized views

- Use the **ALTER MATERIALIZED VIEW SET OPTIONS** DDL statement to change options for your materialized view.
- Example options:
  - `enable_refresh`
  - `refresh_interval_minutes`
  - `expiration_timestamp`

```
ALTER MATERIALIZED VIEW movielens_demo.recent_movie_ratings  
SET OPTIONS(enable_refresh=TRUE,  
             refresh_interval_minutes=30)
```

# Managing materialized views

- By default, materialized views are automatically refreshed within 5 minutes of a change to the base tables, but not more frequently than every 30 minutes.
  - Changes include inserting or deleting rows in the base table.
  - The refresh interval can be changed at any time with an **ALTER MATERIALIZED VIEW** statement.
  - Automatic refresh is treated similarly to a query with batch priority
- You can manually refresh a materialized view at any time.

```
CALL BQ.REFRESH_MATERIALIZED_VIEW('movielens_demo.recent_movie_ratings');
```



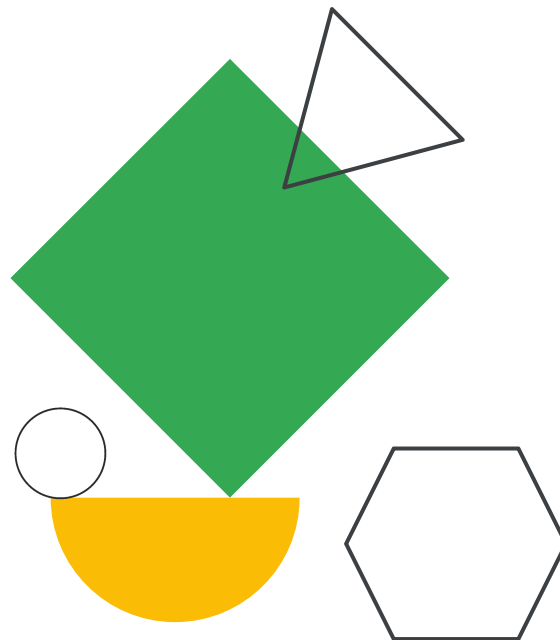
# Resources consumed and pricing for materialized views

Costs are associated with the following aspects of materialized views:

- Querying materialized views
- Maintaining materialized views, such as when materialized views are refreshed
  - The cost for automatic refresh is billed to the project where the view resides.
  - The cost for manual refresh is billed to the project in which the manual refresh job is run.
- Storing materialized view tables

# Demo

Using materialized views in  
BigQuery





03

BI Engine

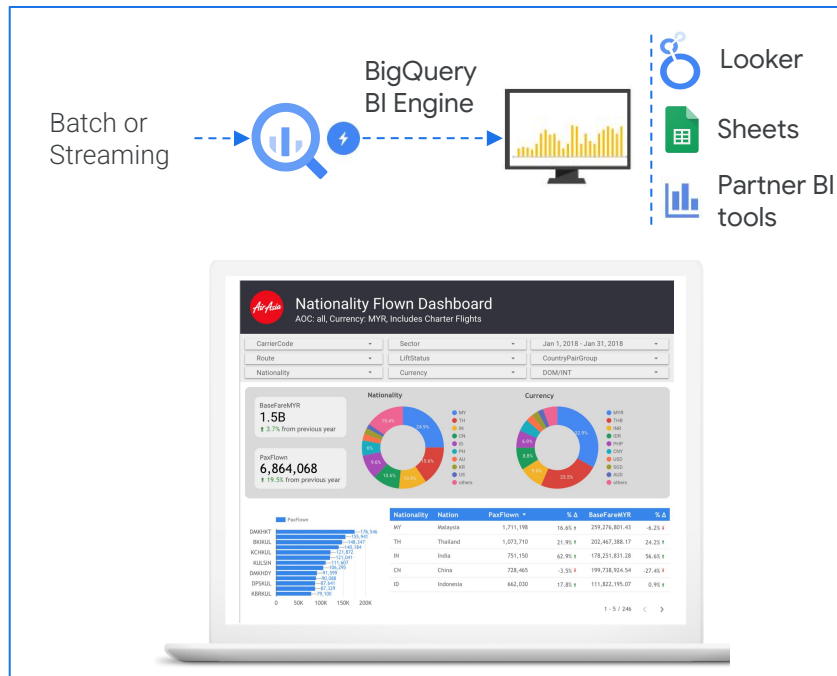
# BigQuery BI Engine vision

Always **fresh**  
Always **fast**

**Democratize BI** by enabling  
data and business analysts  
to perform interactive  
analytics in **real time** at  
scale.

# Add value: BI Engine for dashboard performance

- Managing OLAP cubes or separate BI servers for dashboard performance is not necessary.
- BI Engine natively integrates with BigQuery streaming for real-time data refresh.
- BI Engine is an In-memory analysis service that allows you to execute your business intelligence queries with sub-second query response time for data stored in BigQuery.

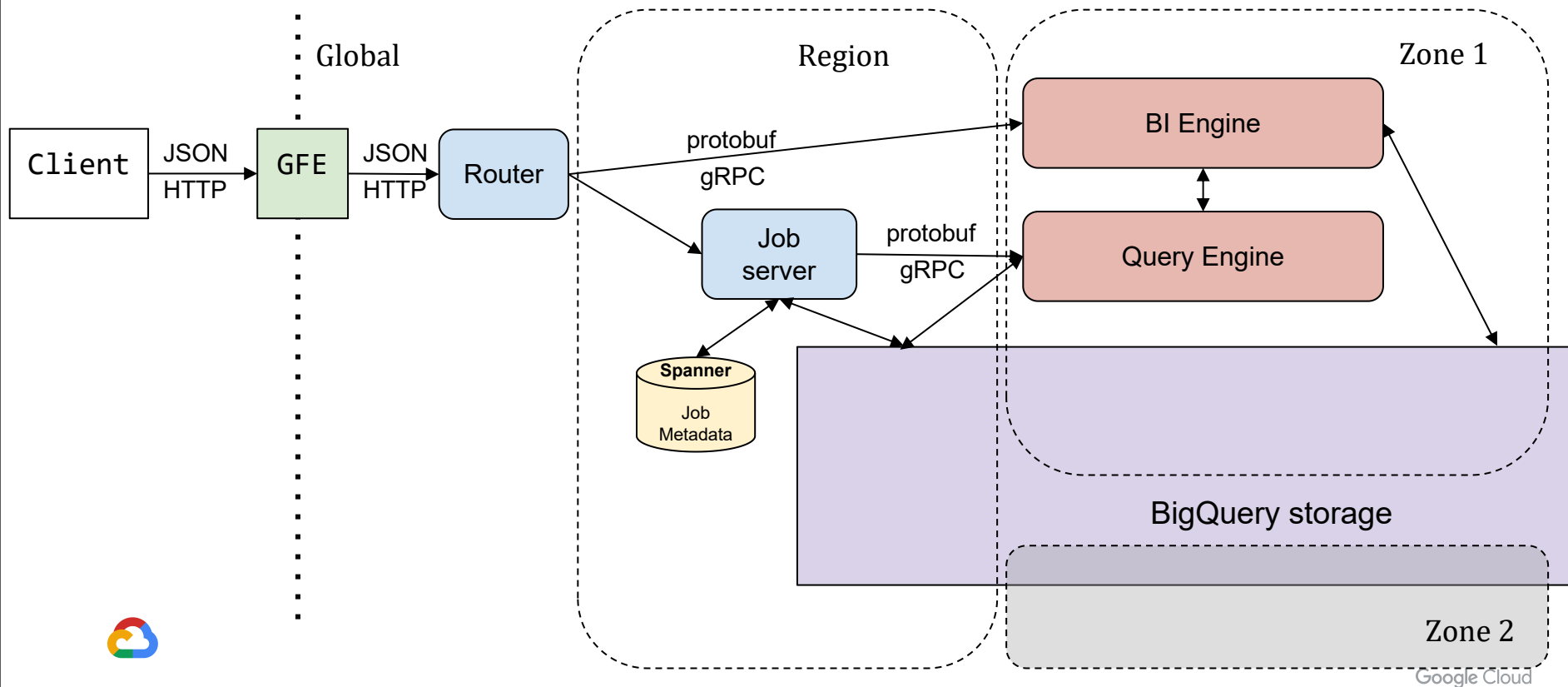


# BI Engine | Low latency queries for BI dashboards

## Design Approach:

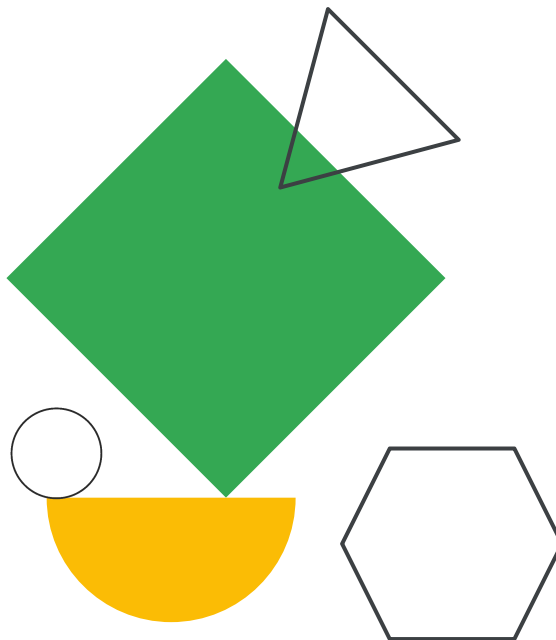
- In-memory data caching co-located with vectorized query processing
- Work seamlessly with BigQuery storage and regular query execution
- Share many components with the regular query execution engine

# Running a BI query in BigQuery



# Demo

Using BI Engine







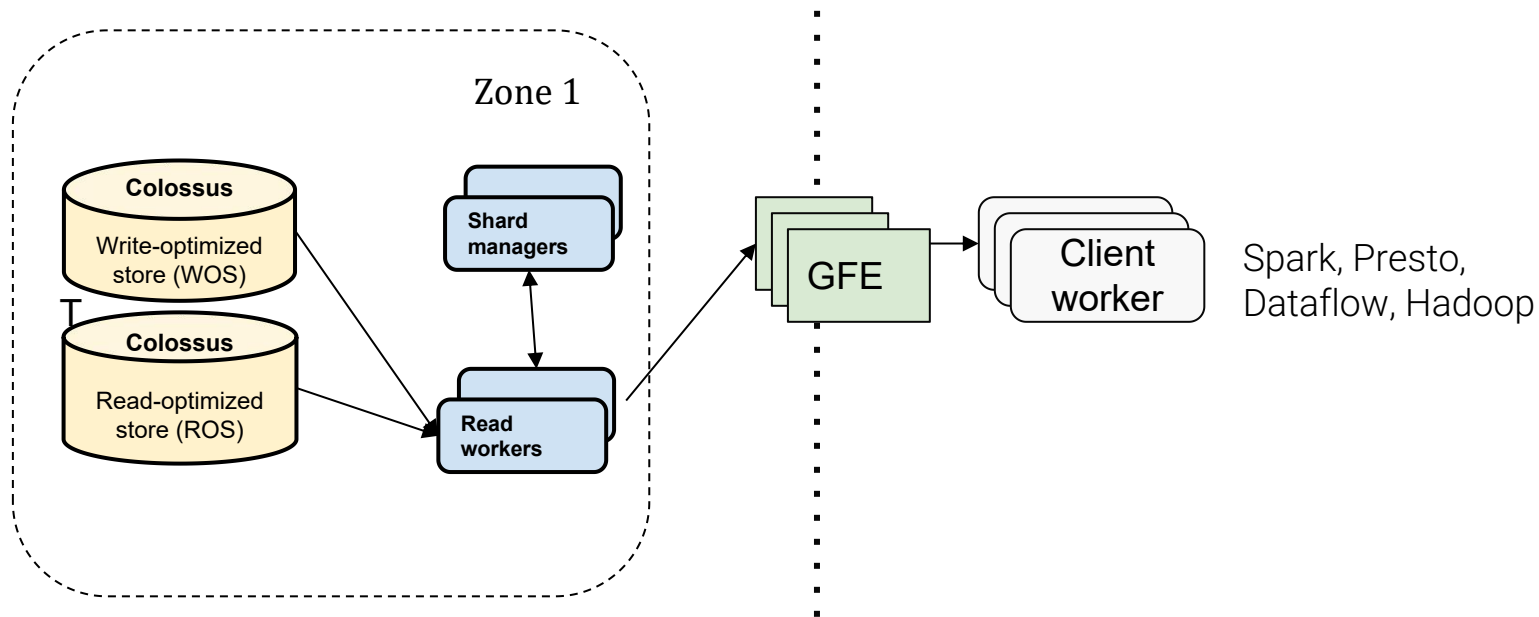
# High Throughput Reads

# High-throughput read

Provide parallel, high-throughput data access to third-party systems

- Dataflow (BigQueryIO Connector)
- OSS tools through Dataproc (Spark, Presto, Hadoop, Hive)
- BigQuery as data lake storage (BigQuery Storage Read API)
- TensorFlow IO for distributed training of machine learning models

# High-throughput read architecture



A blue square containing the white text '05'. To the right of the square is a yellow triangle pointing to the right.

05

# BigQuery Storage Read API

# Three ways of accessing BigQuery table data

- Record-based paginated access

- Using `tabledata.list` or `jobs.getQueryResults` REST API methods
- Best for small result sets
- Method that is used in the web UI

- Bulk data export

- Using BigQuery extract jobs
- Data is exported to Cloud Storage as a batch job
- Limited by daily quotas

# Three ways of accessing BigQuery table data

- BigQuery Storage Read API
  - Data is sent using a binary serialization format (gRPC protocol)
  - Parallelism is built into the API
  - Preferred way of reading data from BigQuery at scale

# Features of the BigQuery Storage Read API

- Multiple streams:
  - Consumers can read disjoint sets of rows from the same table within a session.
  - Advantageous for distributed processing frameworks.
  
- Column projection:
  - Users can select a subset of columns to read.

# Features of the BigQuery Storage Read API

- Column filtering:
  - Supports simple filter predicates to filter data on the server side.
- Snapshot consistency:
  - Storage sessions read based on a snapshot isolation model.
  - Default snapshot time is based on session create time.



# Basic flow of the BigQuery Storage Read API

Google provides client libraries to make it easy for users to leverage the Storage Read API.

Regardless of your choice of language, using the Storage Read API involves three key steps:

- Create a session.
- Read from a session stream.
- Decode row blocks.

# Create a session

```
from google.cloud.bigquery_storage import BigQueryReadClient, types

client = BigQueryReadClient()
table_ref = f'projects/{PROJECT}/datasets/{DATASET}/tables/{TABLE}'

requested_session = types.ReadSession()

requested_session.table = table_ref
requested_session.data_format = types.DataFormat.AVRO
requested_session.read_options.selected_fields = ["name", "number", "state"]
requested_session.read_options.row_restriction = 'state = "WA"'
requested_session.table_modifiers.snapshot_time = SNAPSHOT_TIME
```

# Create a session

```
parent = f"projects/{PROJECT}"  
  
session = client.create_read_session(  
    parent=parent,  
    read_session=requested_session,  
    max_stream_count=20)
```

# Read from a session stream and decode rows

```
reader = client.read_rows(session.streams[0].name)

frames = []

for message in reader.rows().pages:
    frames.append(message.to_dataframe())

dataframe = pandas.concat(frames)

# Code to process data from BigQuery
```



## Considerations for External Data Sources

# External data sources

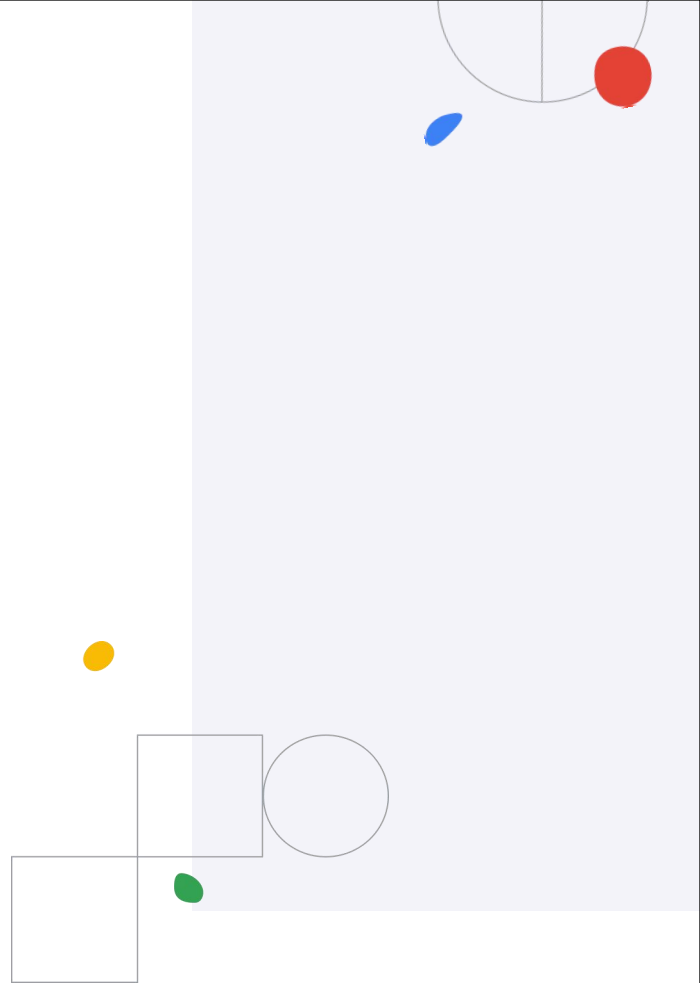
- Query OSS data in place; loading is not required.
- Convenient for ETL workloads, data exploration, and lift and shift use cases.
- Caveats:
  - Performance
  - Consistency
  - Mutability

# Recap: Limitations of querying from external data sources directly

- Strong performance disadvantages
- Data consistency not guaranteed
- Cannot use table wildcards



# Questions?





**Lab (45 min)**

# **Working with BI Engine and Looker**

**45:00**