BigQuery Architecture Fundamentals

Rule #1

Don’t optimize prematurely

Ignore all other best practices.
Try it out
If it is fast/inexpensive enough, leave it alone

BigQuery: Fun with numbers

350PB of data

Stored by one customer

100,000,000,000,000
(one hundred trillion rows)

Queried by multiple customers

10,000 concurrent queries

Run by another customer

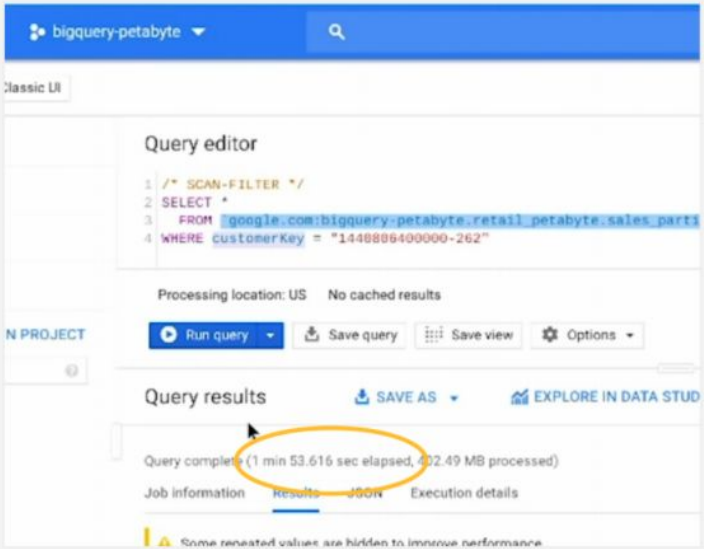
Rule #2

BigQuery is always getting better/faster. Read Rule #1

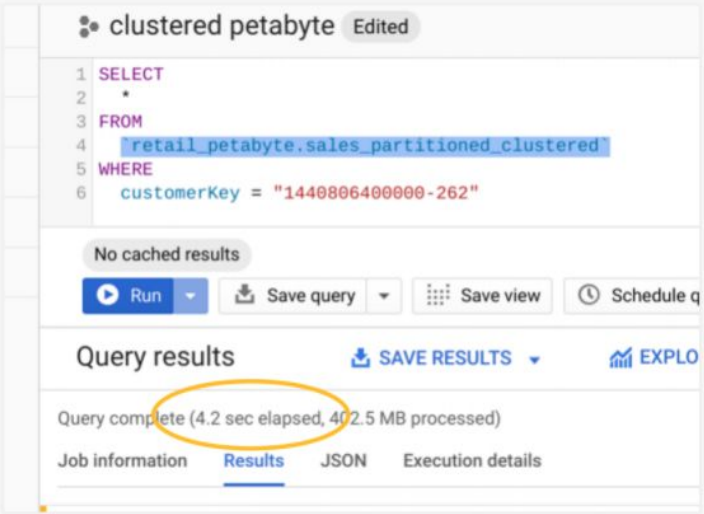
Continuous performance improvement

Performance

2018



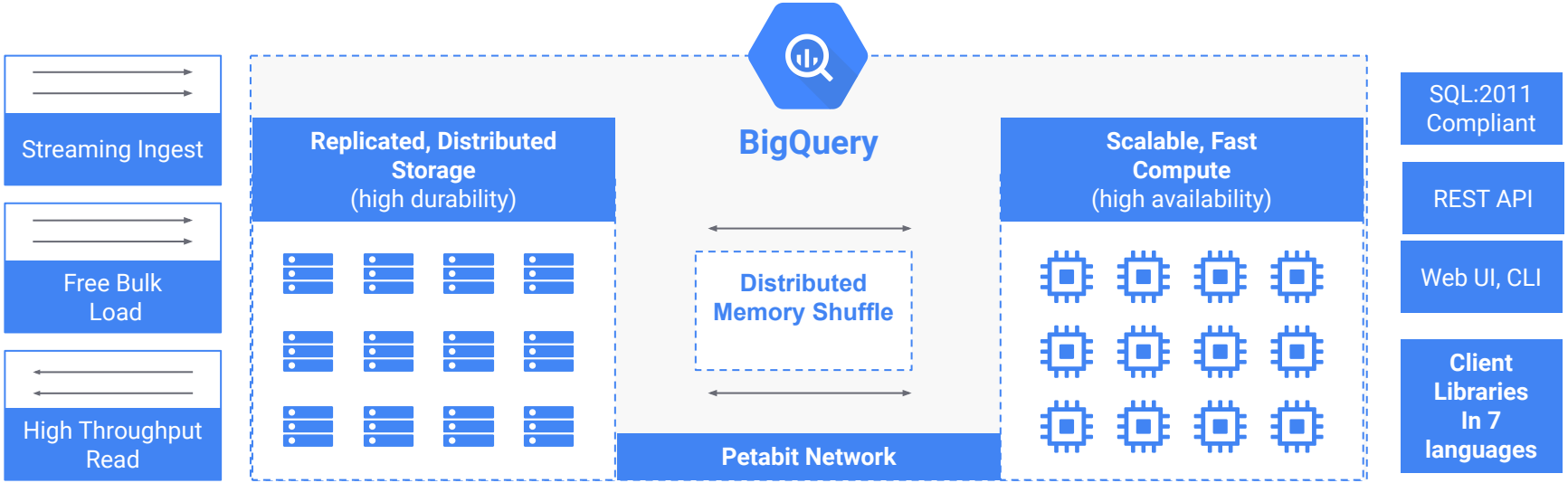
2020



Rule #3

If you’re in this training, occasionally Rule #1 and #2 aren’t enough.

BigQuery Architecture



Key Architecture Design Principles

Storage and Compute Separation

Petabyte-scale
High availability
Serverless and multi-tenant

Colocation and Caching

High performance at low cost

Integrated Hardware/Software Stack

Take advantage of hardware primitives
High performance at low cost

Integration with GCP

Common security and privacy policies across products
Seamless GCP experience

BigQuery Service Locations

BigQuery is a **regional** service

- **Regions** - (us-east4, europe-west5)
  - Multiple zones, one or more campuses, single metropolitan area, single jurisdiction
  - Data residence and colocation guarantees
- **Multi-Regions** - US, EU
  - Multiple zones, multiple campuses, multiple metropolitan areas
  - Flexible capacity planning
  - Generally less expensive
  - Improved durability due to off-region backups

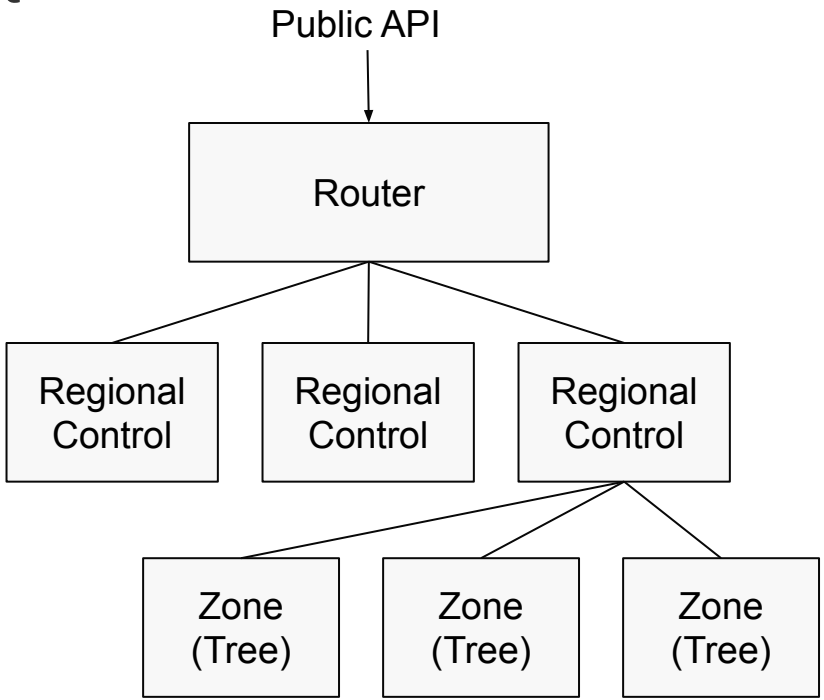
GCP Zone: *A zone is a deployment area for Cloud Platform resources within a region. Zones should be considered a single failure domain within a region.*

BigQuery Service Deployment

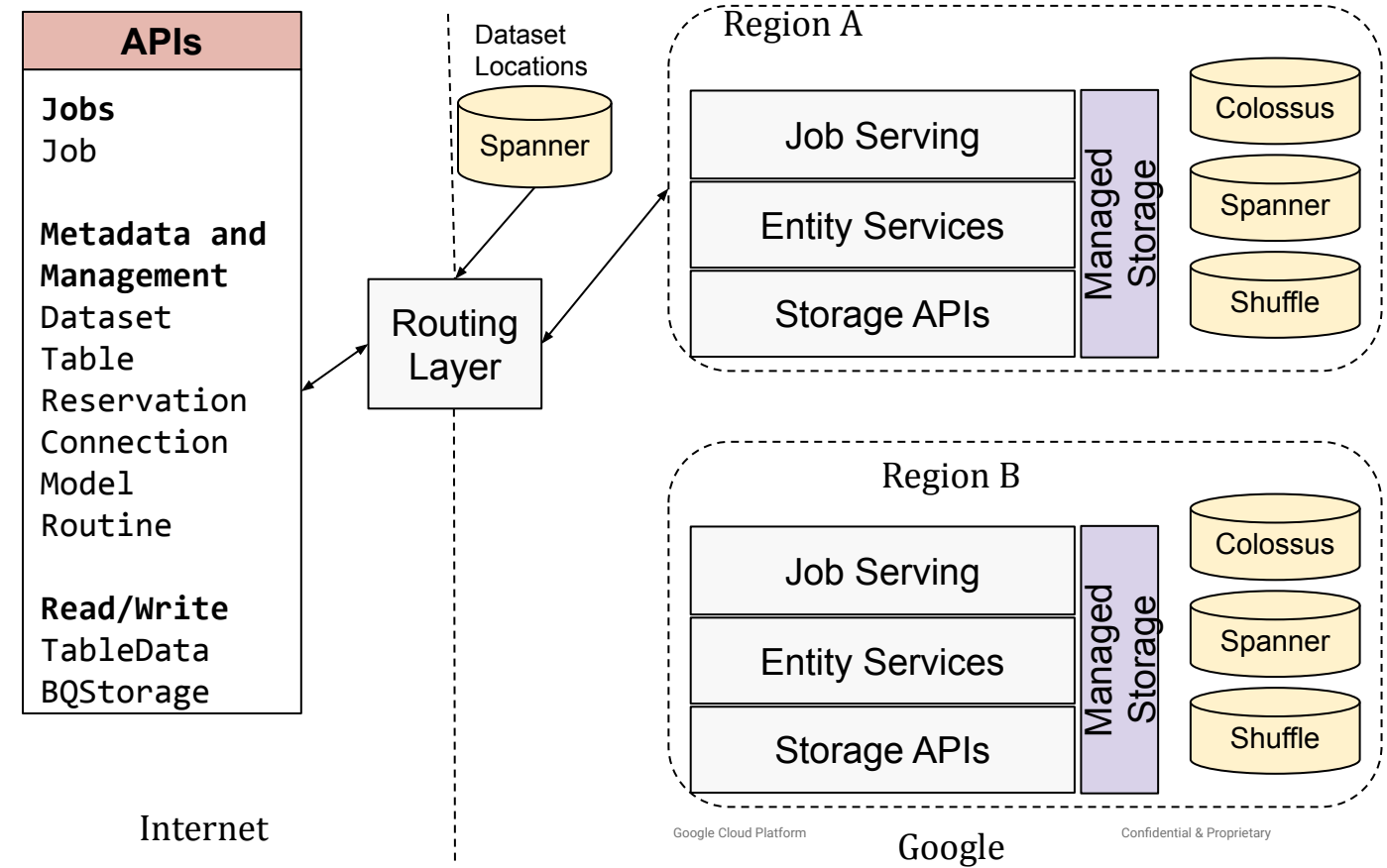
Global Layer
Components needed to route to the right region
Spread across zones globally

Regional Layer
Manage capacity, data and metadata redundancy
Spread across zones within the region

Zonal Layer
Compute and storage backend



Service Layers and Components



APIs

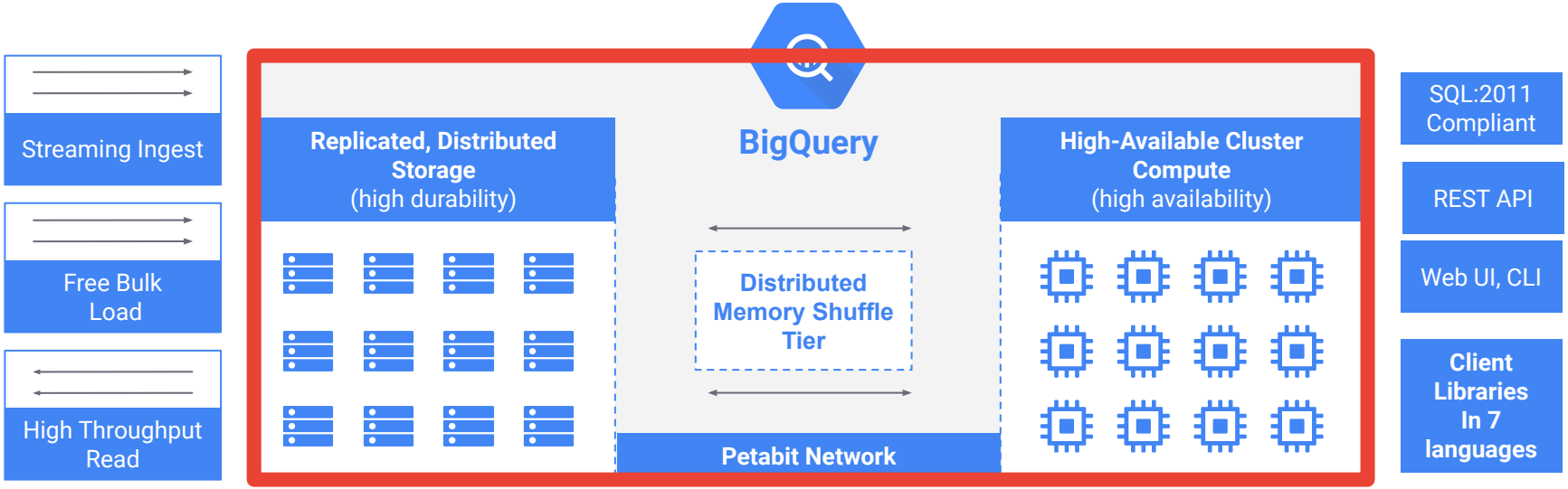
Job/Query - Run a single SQL query or a script. Load or export data.
UI, ODBC/JDBC, Command line client, Looker, Data Studio, …
Example: job.insert(), job.query(), job.getQueryResults()

Storage - Read from and write to BigQuery tables.
UI, Dataflow and Dataproc, custom code
Example: storage.read(), tabledata.insertall().

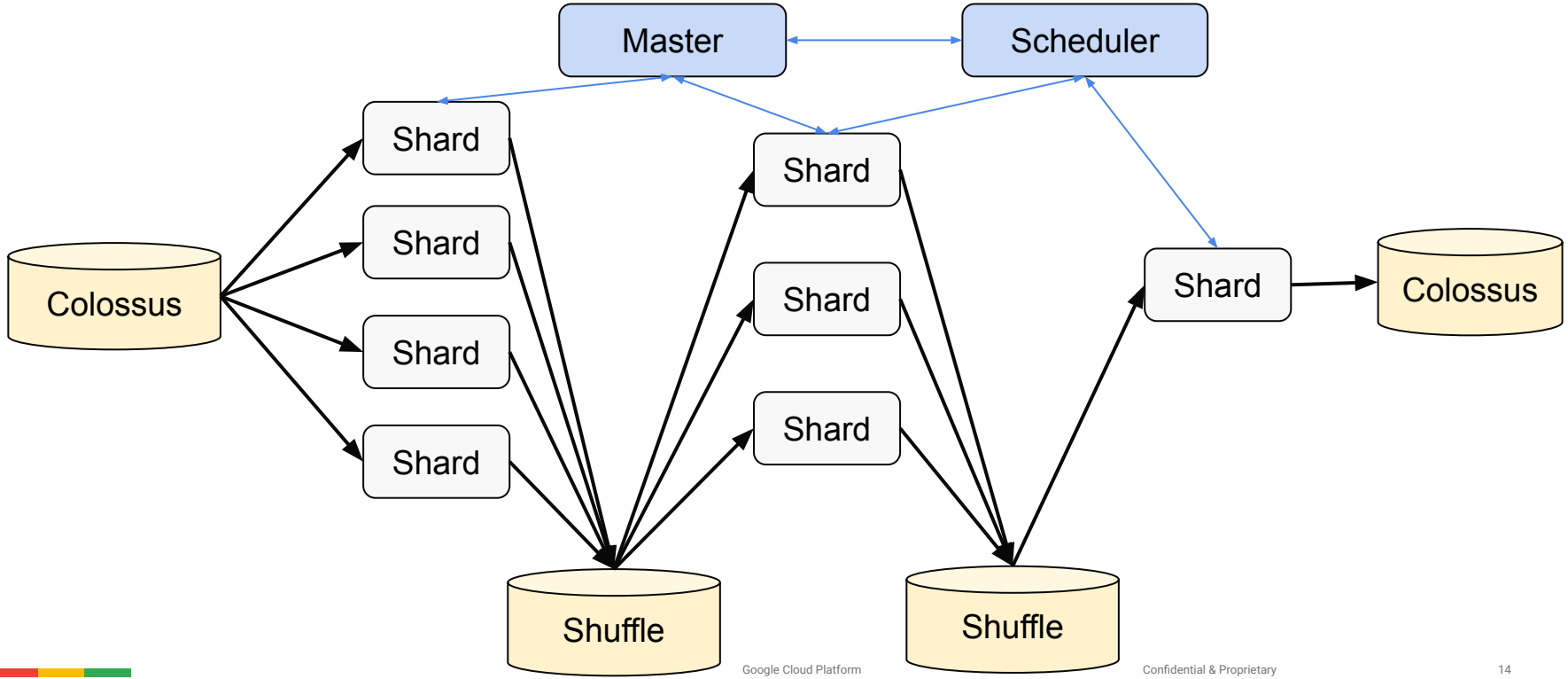
Metadata - Create dataset. Add a routine or script.
UI, SQL, CLI
Example: datasets.insert(), tables.list()

Management - Create and modify reservations
UI, CLI

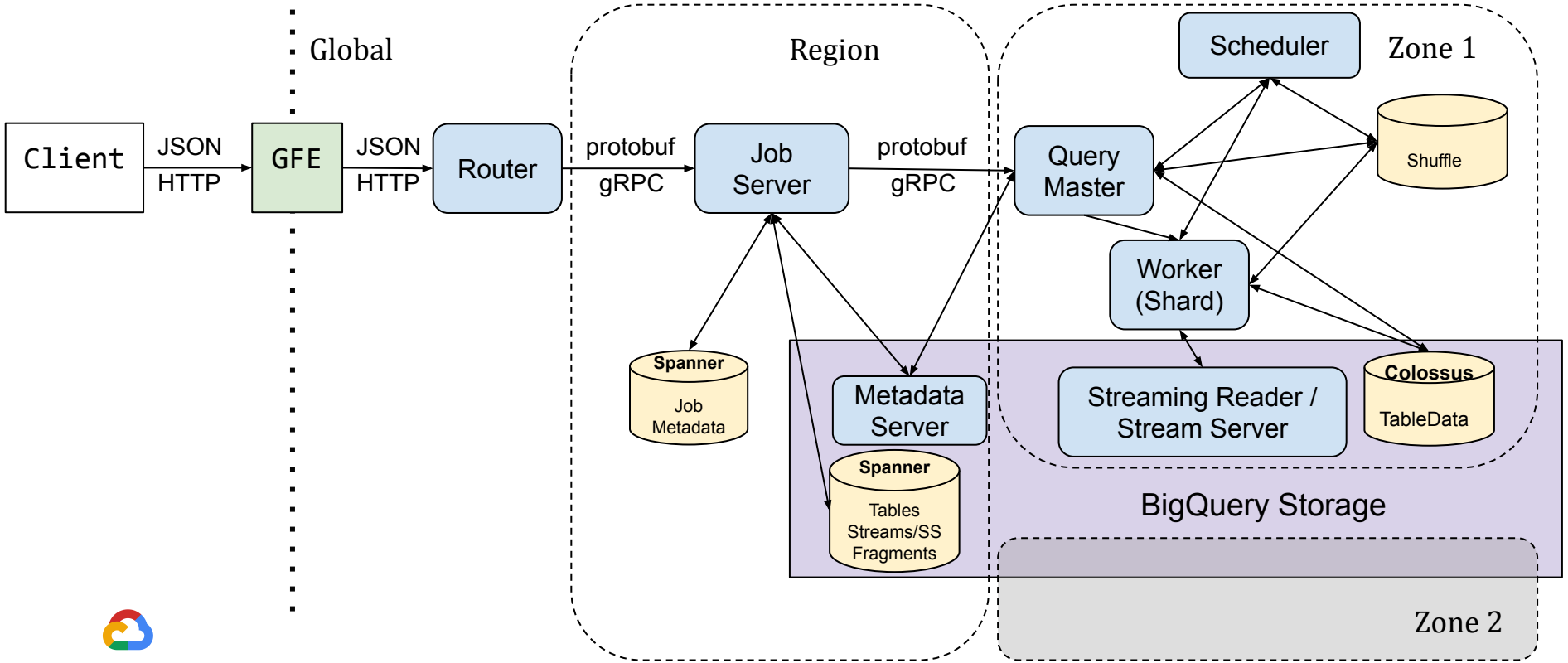
Google Core Infrastructure



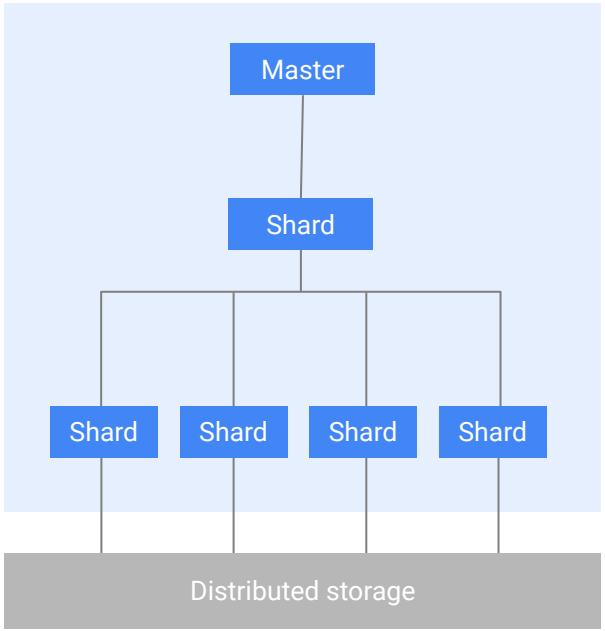
Query Engine Execution Flow



Running a Query Job in BigQuery



Simple Query Execution

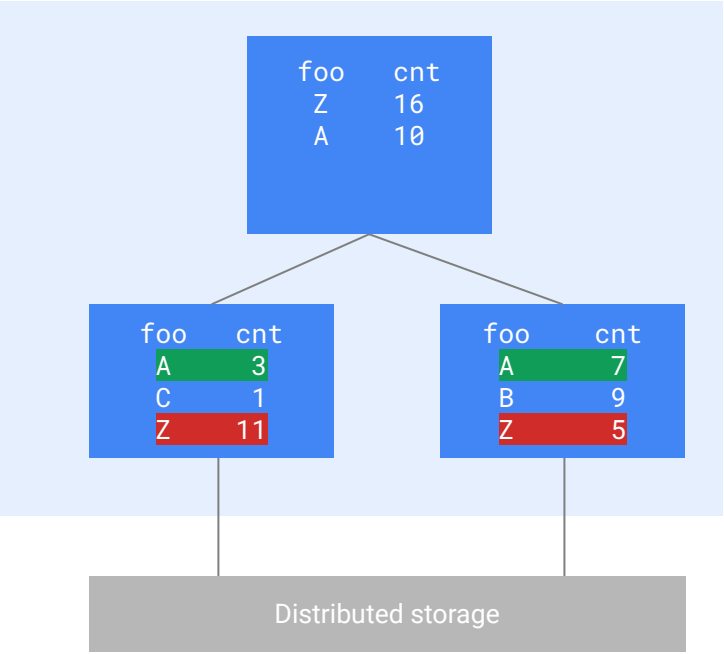



```
SELECT COUNT(*) FROM
wikipedia_benchmark.Wiki1B
WHERE title LIKE "G%o%o"
```

Stage 2: Sum

Stage 1: Filter, Count

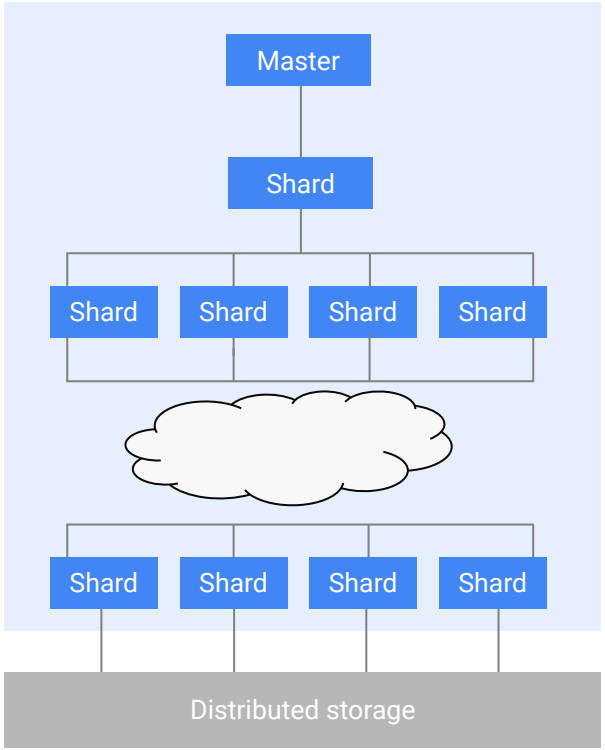
Aggregation with High Cardinality




```
SELECT foo, COUNT(*) as cnt
FROM `...`
GROUP BY 1
ORDER BY 2 DESC
LIMIT 2
```

- Can’t discard ‘B’ or ‘C’ until after all previous stages are complete.
- High Cardinality ‘foo’ will overwhelm the root node.

Shuffle Aggregation Execution




```
SELECT language, MAX(views) as views
FROM `wikipedia_benchmark.Wiki1B`
WHERE title LIKE "G%o%"
GROUP BY 1 ORDER BY 2 DESC LIMIT 100
```

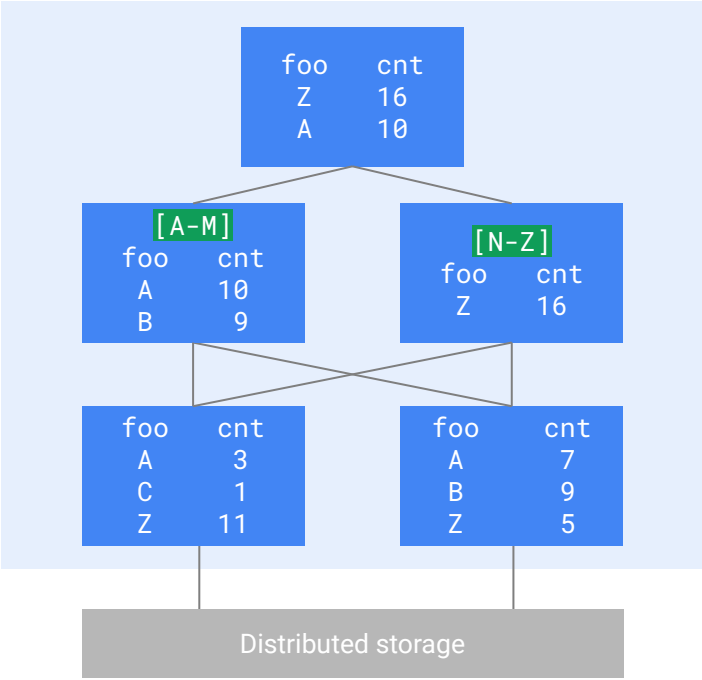
Stage 3: SORT, LIMIT (1 slots)

Stage 2: GROUP BY, SORT, LIMIT (289 slots)

Shuffle

Stage 1: Partial GROUP BY (40,859 sinks)

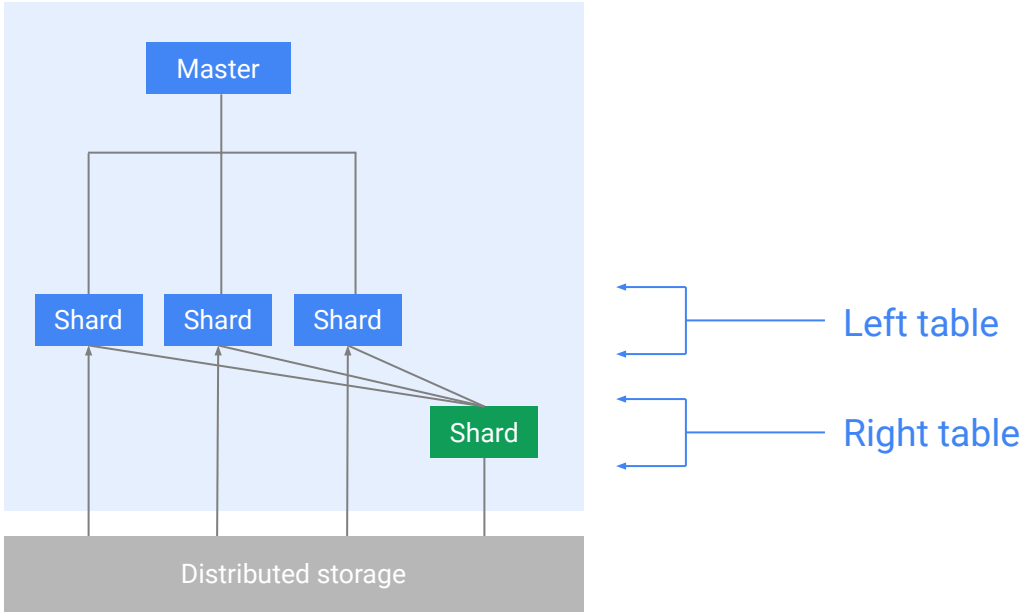
Aggregation with Shuffle




```
SELECT foo, COUNT(*) as cnt
FROM `...`
GROUP BY 1
ORDER BY 2 DESC
LIMIT 2
```

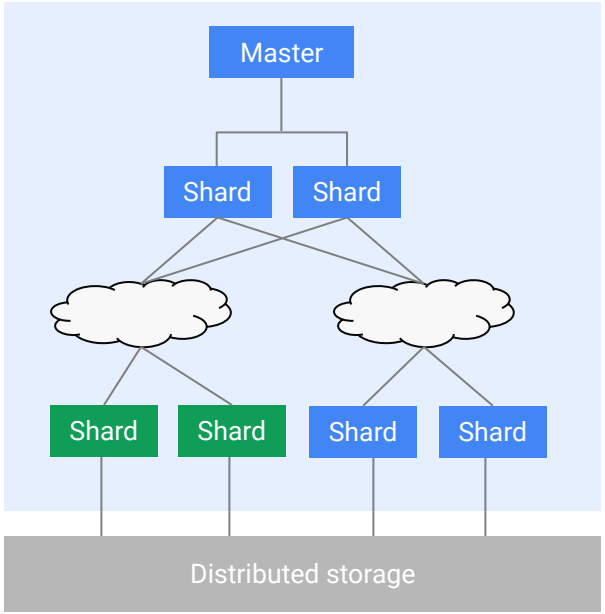
- Shuffle puts like values in the same node
- Scalable, since you never have to return more than the LIMIT value from each node in middle tier

Small JOIN (Broadcast)




```
SELECT
  c.author.name a, c2.a m
FROM github_repos.commits c
JOIN (
  SELECT
     committer.name a,
     commit
  FROM
github_repos.commits) c2
ON
  c.commit = c2.commit
WHERE c2.a = 'tom'
LIMIT 1000
```

Large JOIN (Shuffle)



Hash join

Independent shuffles

```
SELECT c.author.name a,  c2.a m
FROM github_repos.commits c
JOIN (SELECT committer.name a, commit
      FROM github_repos.commits) c2
ON c.commit = c2.commit
LIMIT 1000
```

Query Execution Design Choices

Shuffle is the **data transfer mechanism** between workers

- Allows flexible query planning and execution
- Can act as staging area or partitioning mechanism

Query optimization using **dynamic query execution**

- Observe execution and quickly react
- More robust than static (cost based) query optimization

Decouple **scheduling** from query **planning**

BigQuery BI Engine Vision

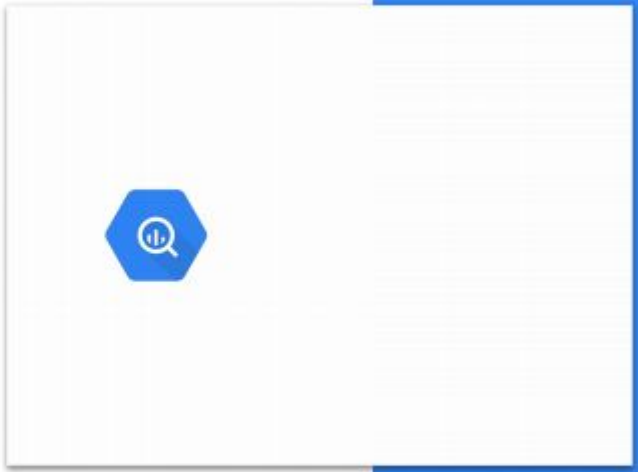
Always **Fresh**
Always **Fast**

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

BigQuery BI Engine – API

Sub-second queries

Simplified architecture

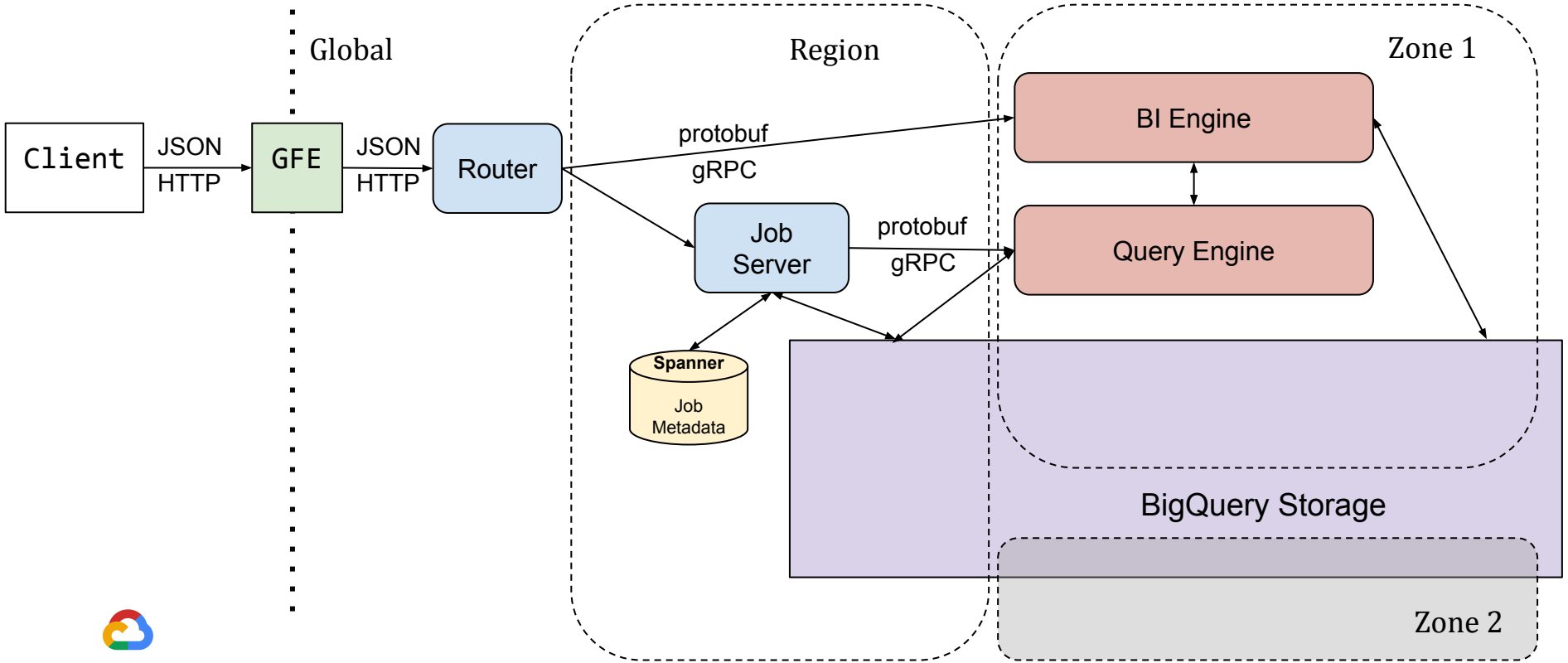


Low latency queries for BI dashboards

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

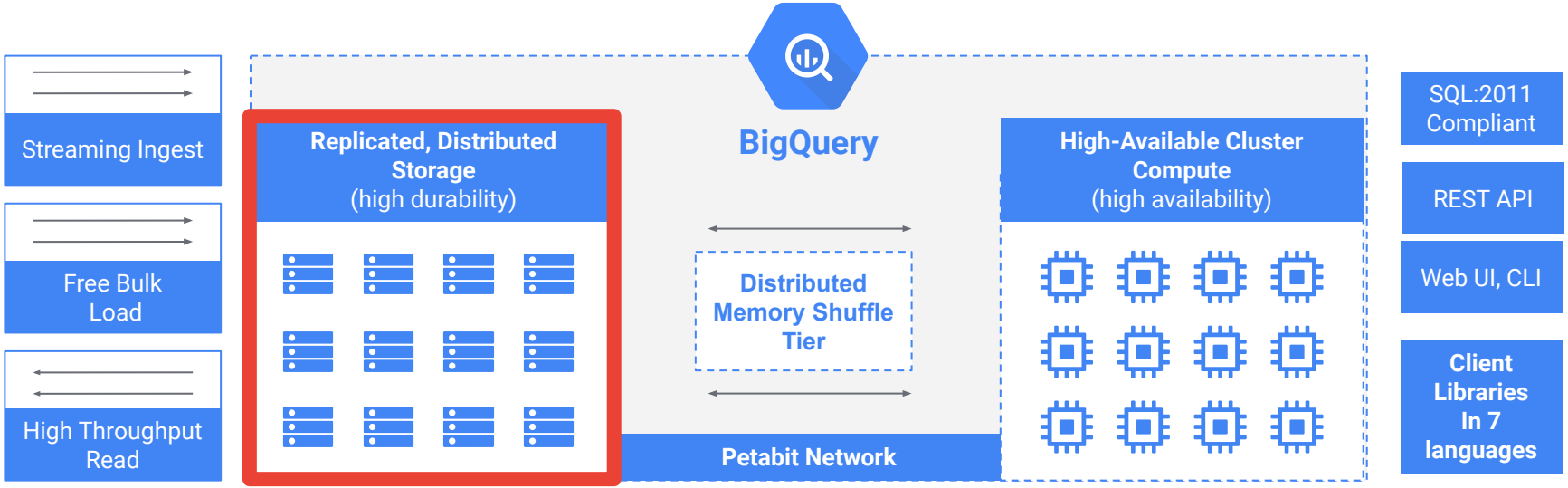


BigQuery Embedded ML – Machine Learning in SQL without leaving your Data Warehouse

Embedded ML

1. Execute ML initiatives without moving data from BigQuery
2. Iterate on models in SQL in BigQuery to increase development speed
3. Automate common ML tasks, and hyperparameter tuning

BigQuery Storage



Column Storage

Record Oriented Storage

Column Oriented Storage

Column-oriented vs Row-oriented storage

- Read less data faster
- Skip unused columns
- Column compression > Row Compression
- Supports vectorized columnar processing

Physical Layout

Capacitor: our proprietary columnar format.

- Maintain optimum sharding structure.
- Implement the logical metadata hints: partitioning/clustering

Why a new format?

- Can improve it under the covers
- Deeply tied to execution engine
- Apply what we've learned over last 10 years

Capacitor Features

- Dictionary encoding (low cardinality)
- Constraints and Bloom Filters (high cardinality)
- Run Length Encoding
- Compression
- Row Reordering

Physical Metadata

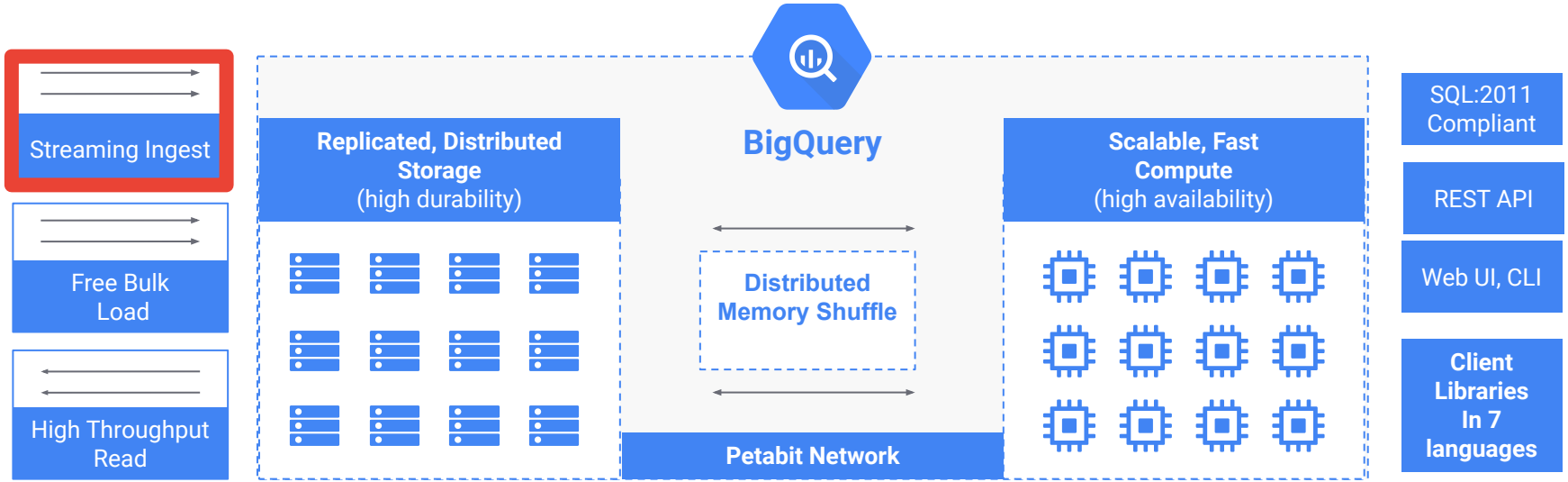
Critical part of BigQuery storage that is designed to support:

- Streaming
- ACID Commits
- Time Travel
- Backups
- Active Storage Management
- Storage Optimization
- Partitioning and clustering
- DML

GCS Federation

- Query OSS data in-place, no loading required.
- Convenient for ETL workloads, data exploration, and lift and shift use cases.
- Caveats
  - Performance
  - Consistency
  - Mutability

Streaming Ingestion

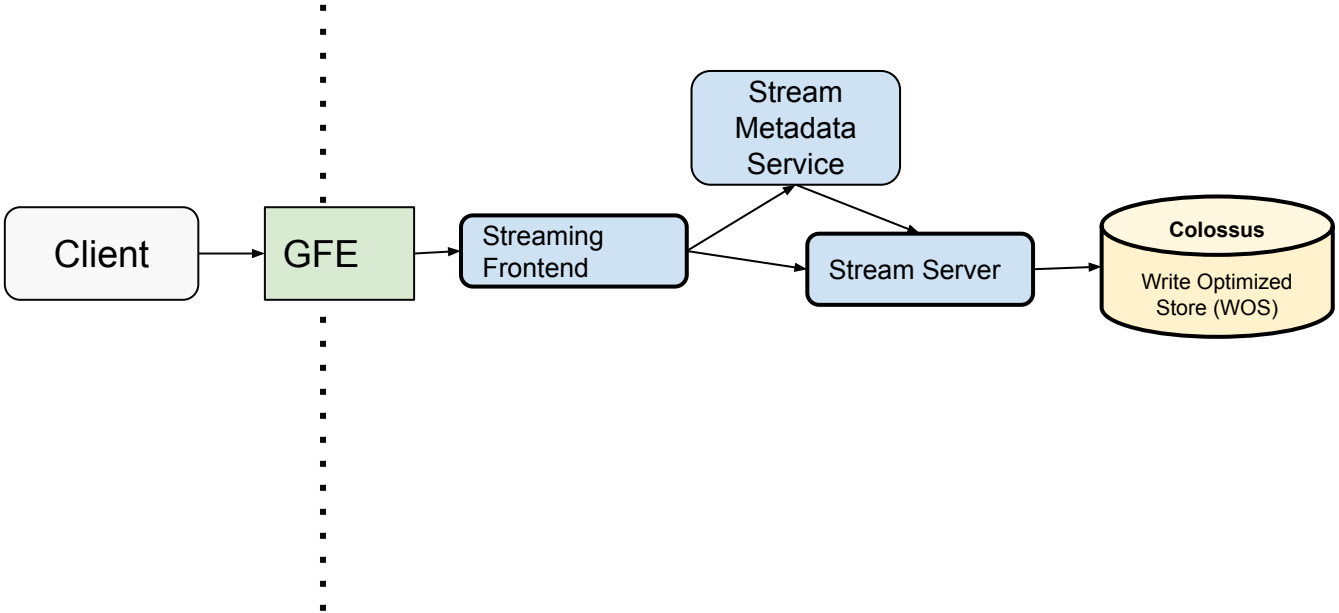


All large datasets are generated over time

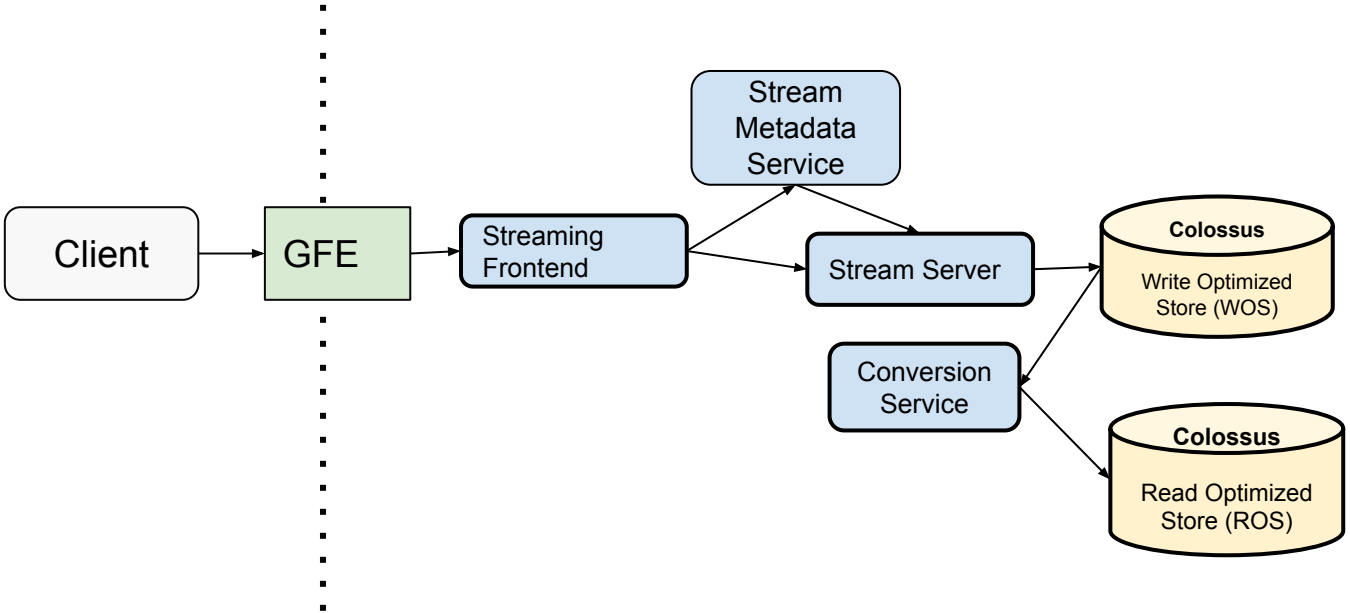
BigQuery Streaming Ingestion

- HTTP Post individual rows or groups of rows
- Designed for high throughput
- Durable
  - Data is acknowledge once committed to disk
  - Same durability guarantees as the rest of the BigQuery storage
- Streaming data immediately available for query
- (soon) Exactly once-ingestion

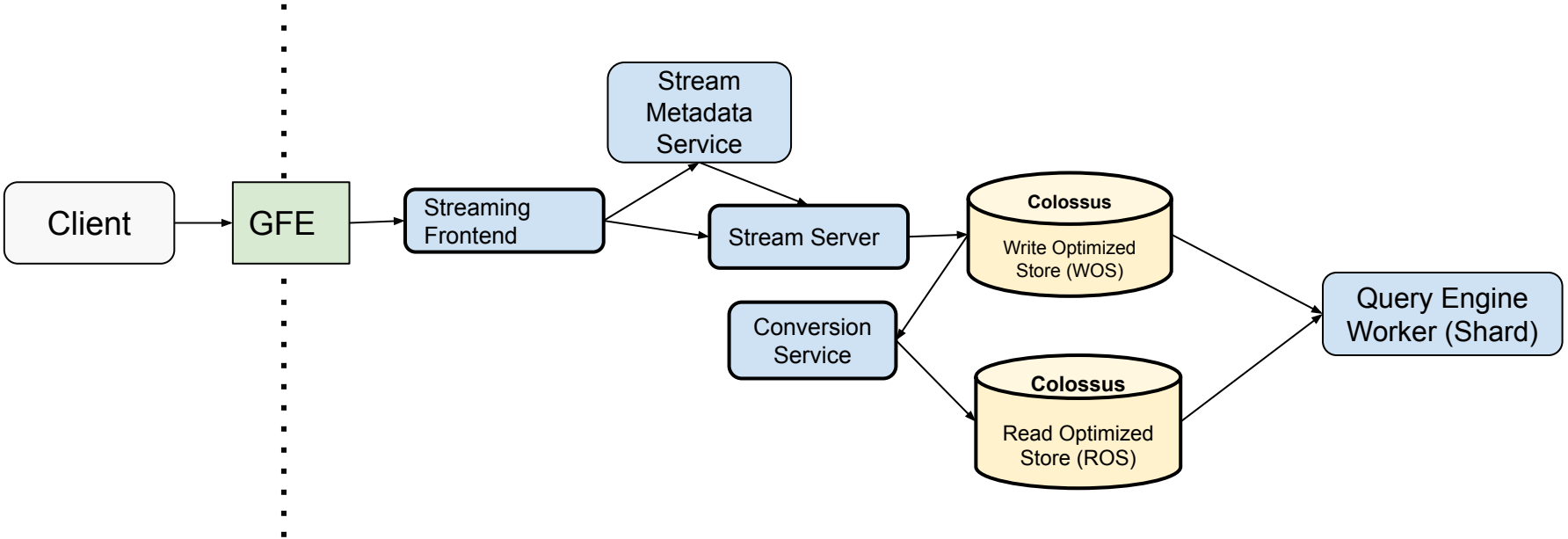
Streaming High Level Architecture



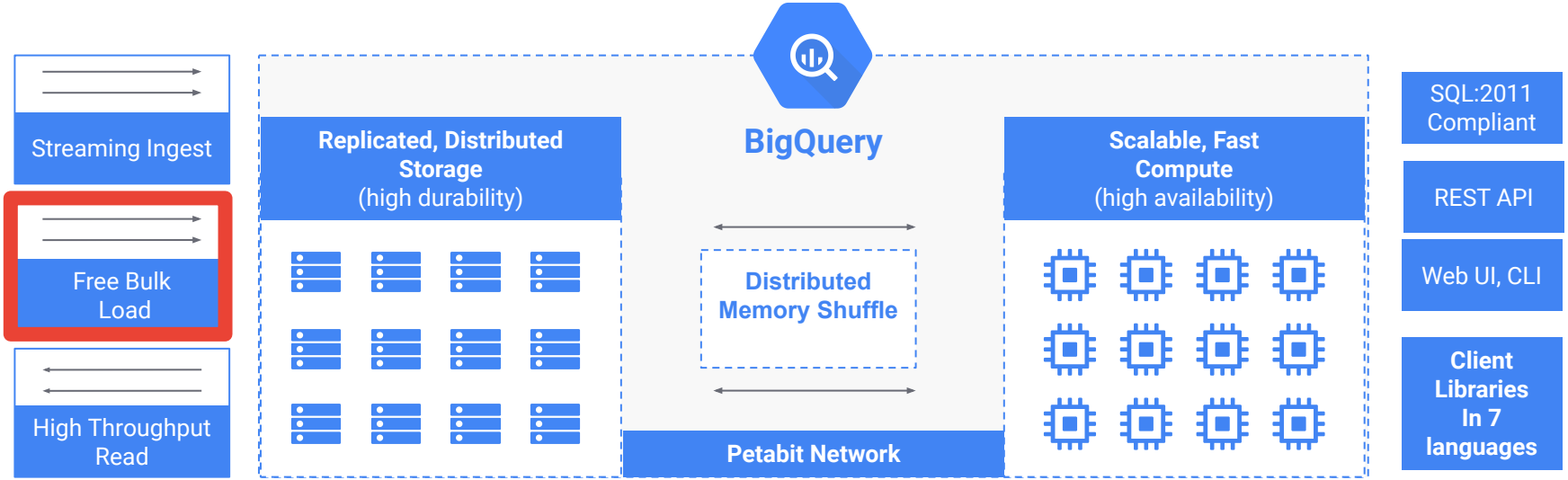
Streaming High Level Architecture



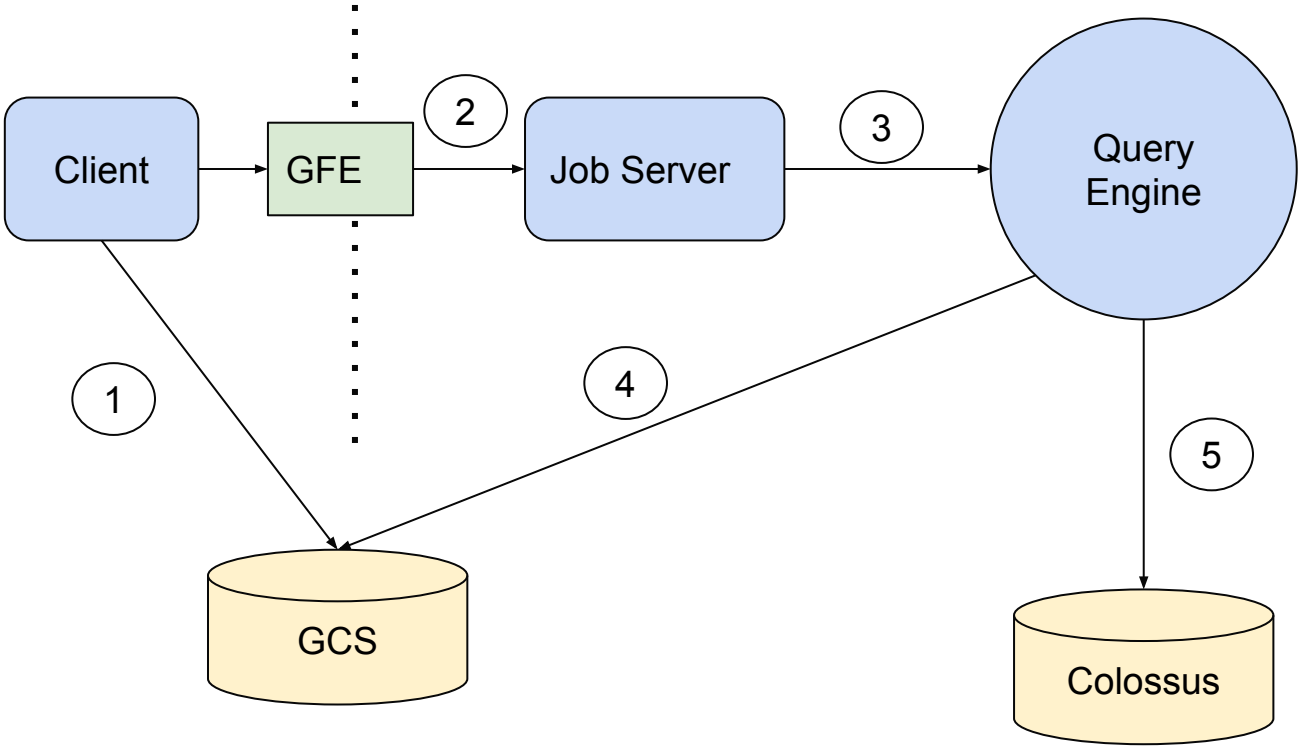
Streaming High Level Architecture



Batch Ingestion



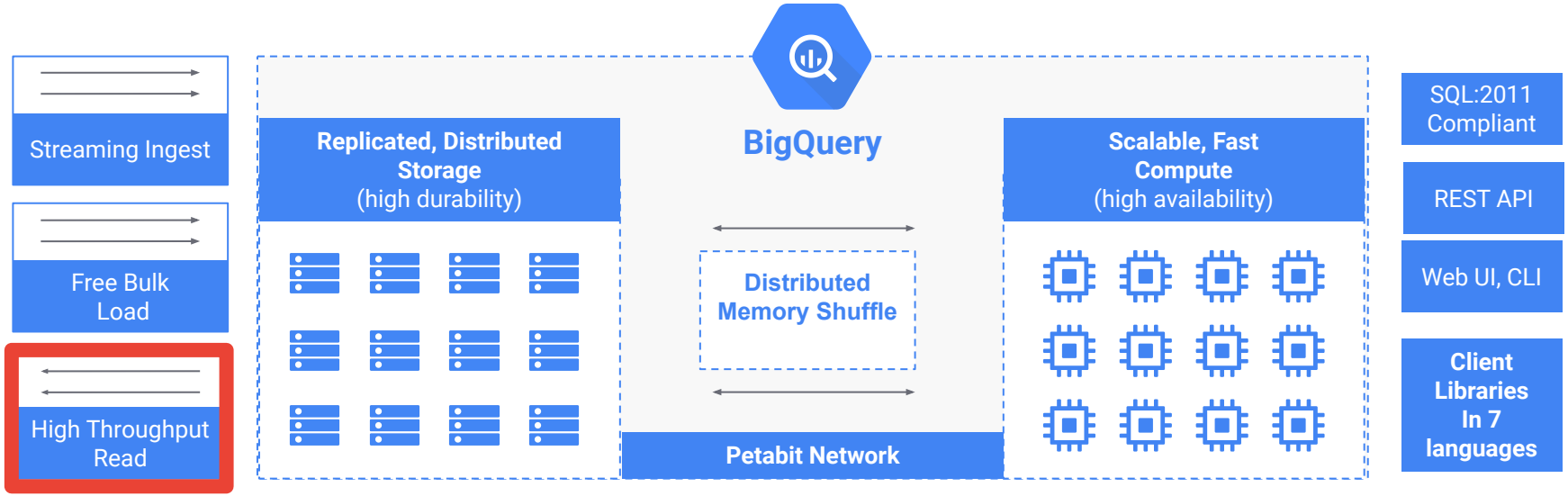
Batch Ingestion



Batch Ingestion

- Read from multiple formats
  - CSV, JSON, Avro, Parquet, ORC
- Multiple data sources
  - GCS, Cloud BigTable, Datastore Backup, Direct upload
- Uses the Query Engine to perform the ingestion and the required recoding
- Performance can be managed through reservations

High Throughput Read

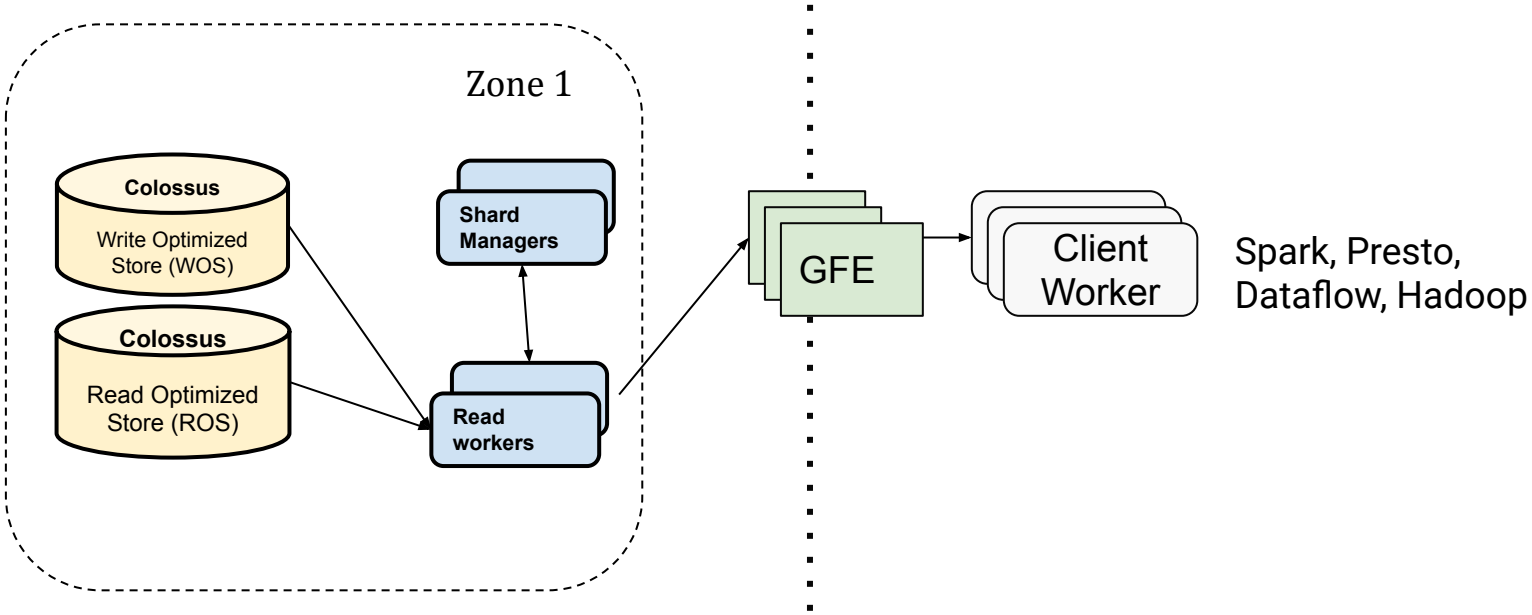


High Throughput Read

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

- Google Dataflow
- OSS tools through Dataproc (Spark, Presto, Hadoop, Hive)
- Using the TensorFlow IO package for training ML models with large datasets
- BigQuery as Data Lake storage

High Throughput Read Architecture



Questions?