BigQuery Deep Dive
H&M - May 2025

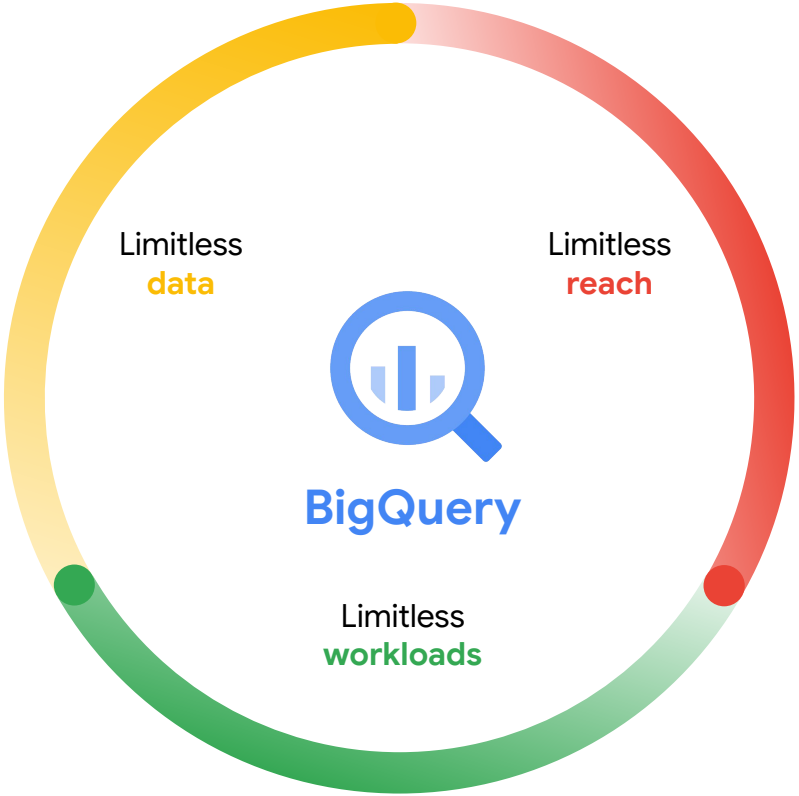
Henrik Warfvinge - Data & AI Architect
henrikw@google.com

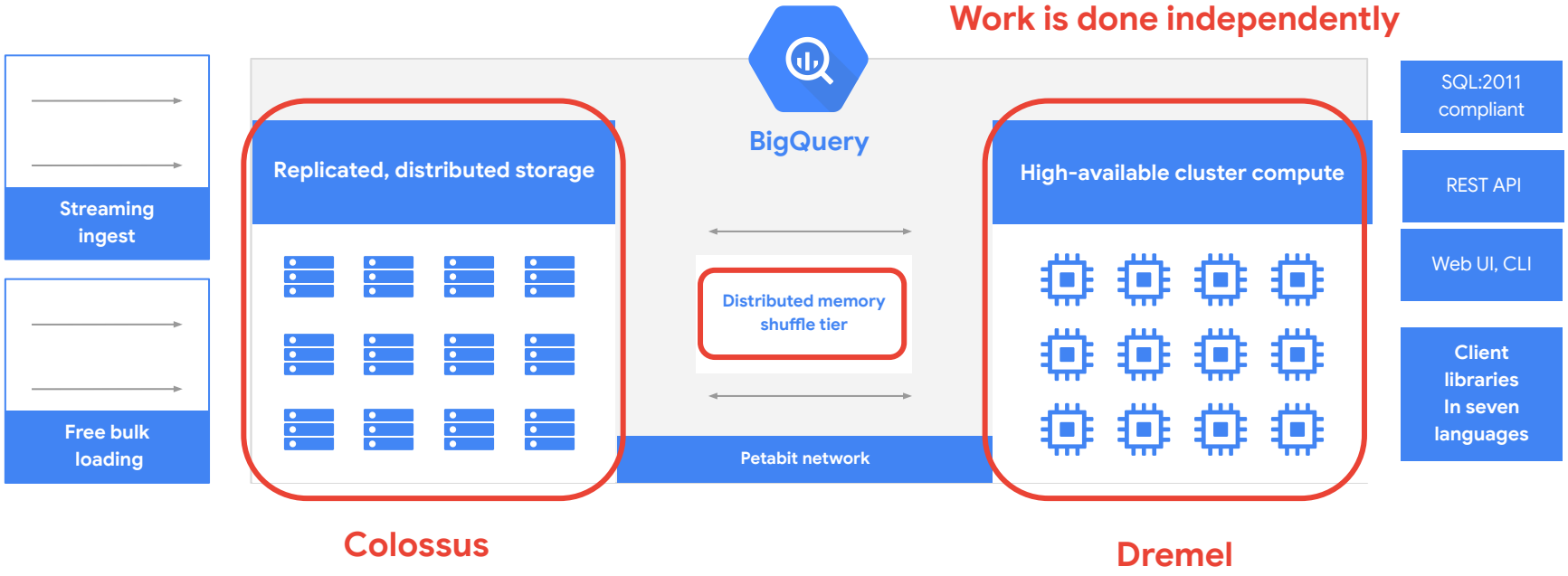
Google Cloud

Recap

BigQuery

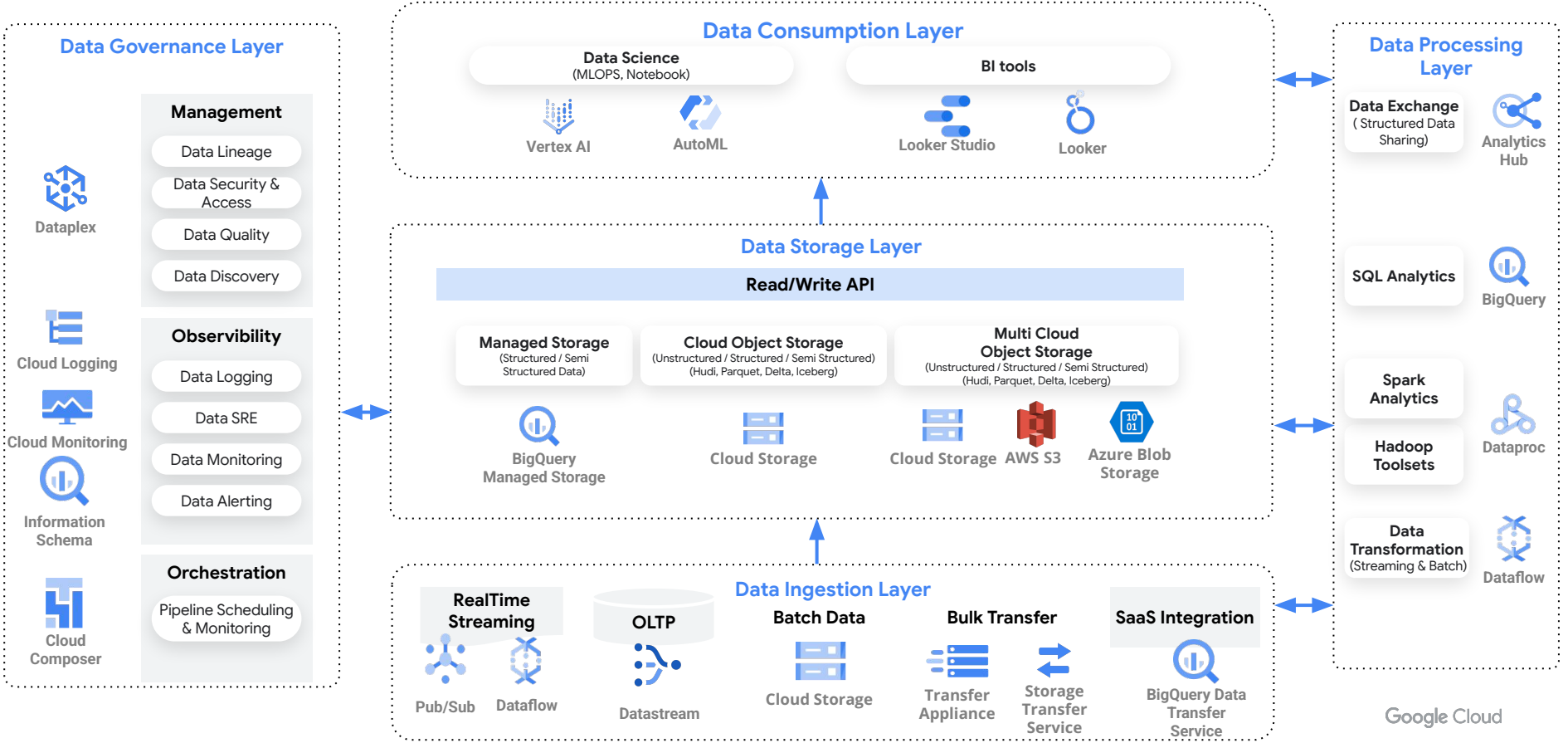
The core of Google's Data Cloud to power your data-driven innovation.




Work is done independently
Streaming ingest
Free bulk loading
Replicated, distributed storage
BigQuery
Distributed memory shuffle tier
Petabit network
High-available cluster compute
SQL:2011 compliant
REST API
Web UI, CLI
Client libraries In seven languages
Colossus
Dremel

https://cloud.google.com/blog/products/data-analytics/understanding-a-google-cloud-analytics-lakehouse

Google Analytics Lakehouse Architecture

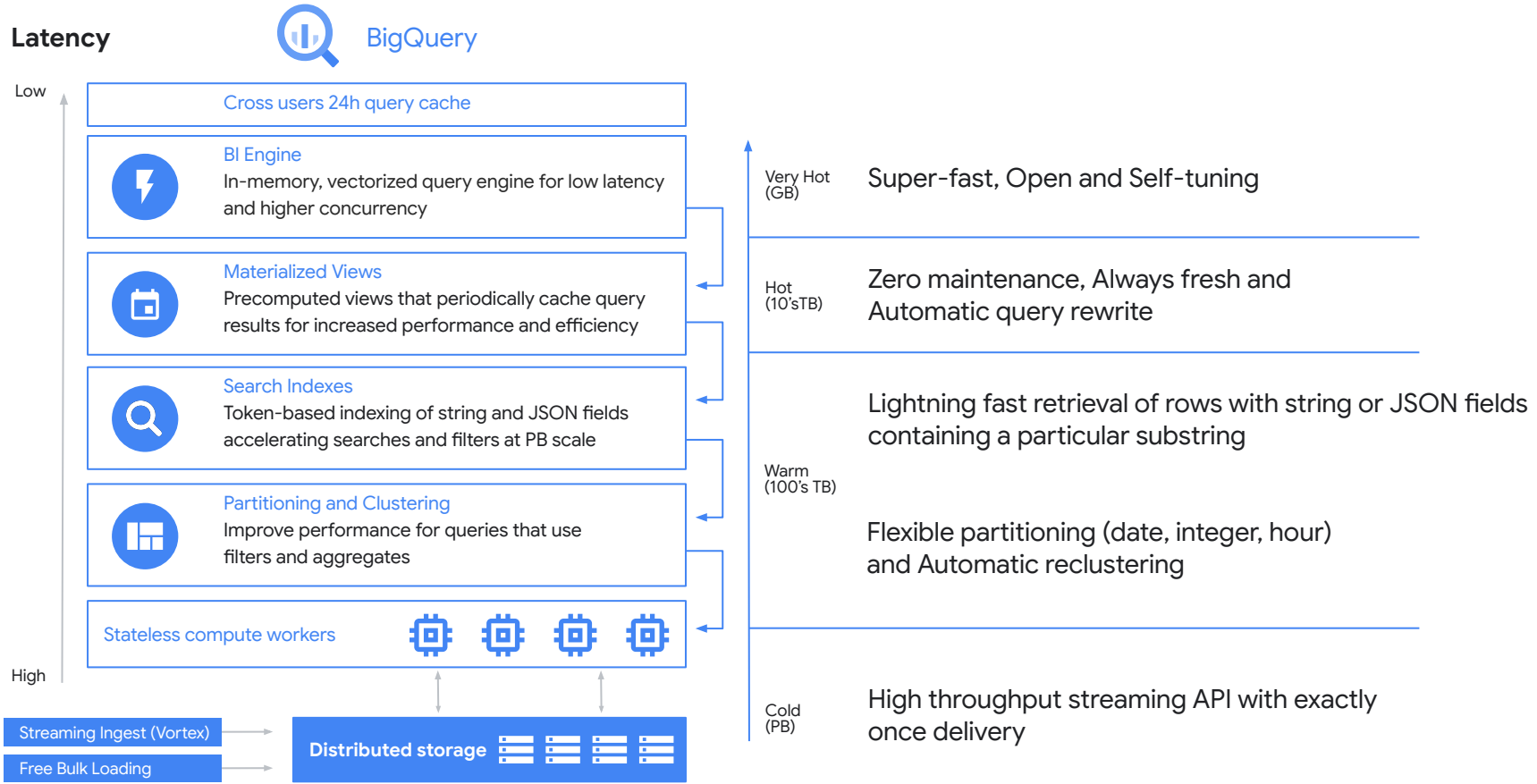


Agenda

- Storage
  - Partitioning
  - Clustering
  - Search indexes
  - Materialized views
  - Bi Engine
- Compute
  - Slots
  - Query execution
- SQL best practices
- Q&A

Going Deep
Storage Internals and optimizations

Adaptive Caching layers in BigQuery



Key features of BigQuery Storage

Compressed

Industry leading compression, result of over a decade of innovation in storage optimization technology including proprietary columnar compression, automatic data sorting, clustering and compaction.

Managed

As a completely managed service, you don't need to provision storage resources or reserve units. BigQuery automatically allocates storage when you load data into the system. You only pay for the amount of storage that you use.

Durable

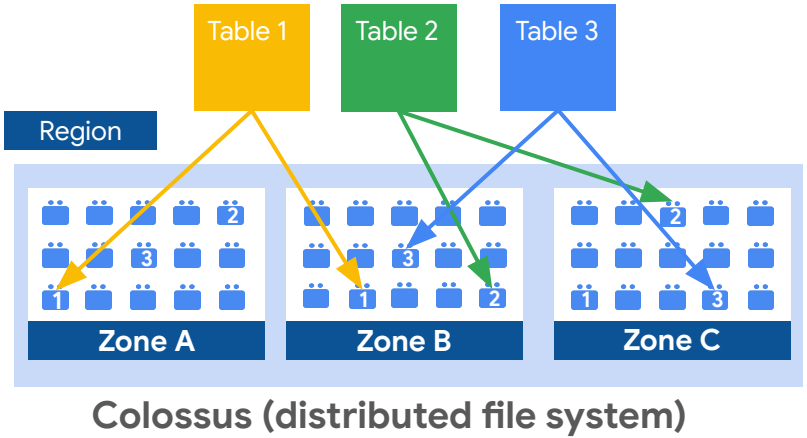
BigQuery storage is designed for 99.999999999% (11 9's) annual durability. It replicates your data across multiple availability zones to protect from data loss due to machine-level or zonal failures.

Encrypted

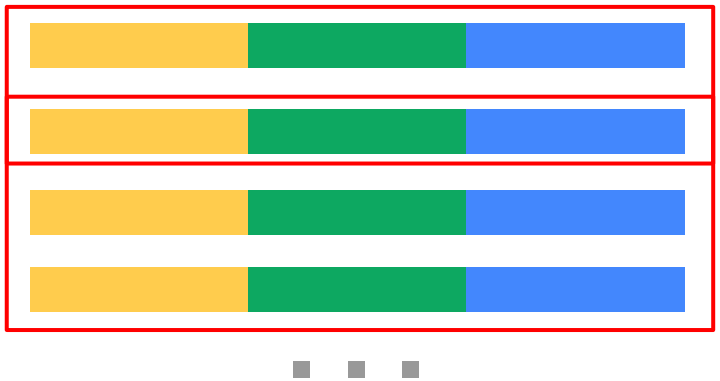
BigQuery automatically encrypts all data before it is written to disk. You can provide your own encryption key or let Google manage the encryption key.

Efficient

BigQuery storage uses an innovative adaptive file sizing and efficient columnar format that is optimized for analytic workloads.

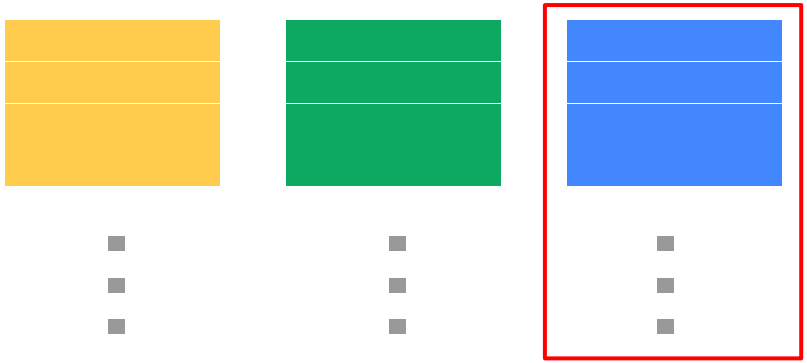


Traditional RDBMS storage



Record-oriented storage
OLTP

BigQuery storage



Columnar storage
OLAP

Data Backup and Recovery

Making it easy to recover from mistakes

GA

Time Travel

Configurable 2-7 day window to access all deleted or changed data.

GA

Table snapshots

Read-only copies of base tables that can be restored for modifications. Useful for logical backups or archiving beyond time-travel window.

GA

Table clones

Read/write copies of base tables. Useful for testing with production databases.

Preview

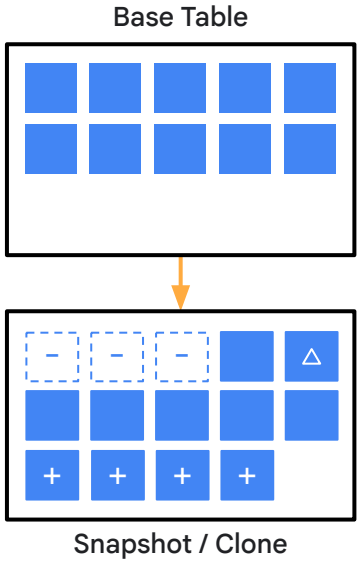
Cross-region table copy

Copies entire table (including CMEK) between regions

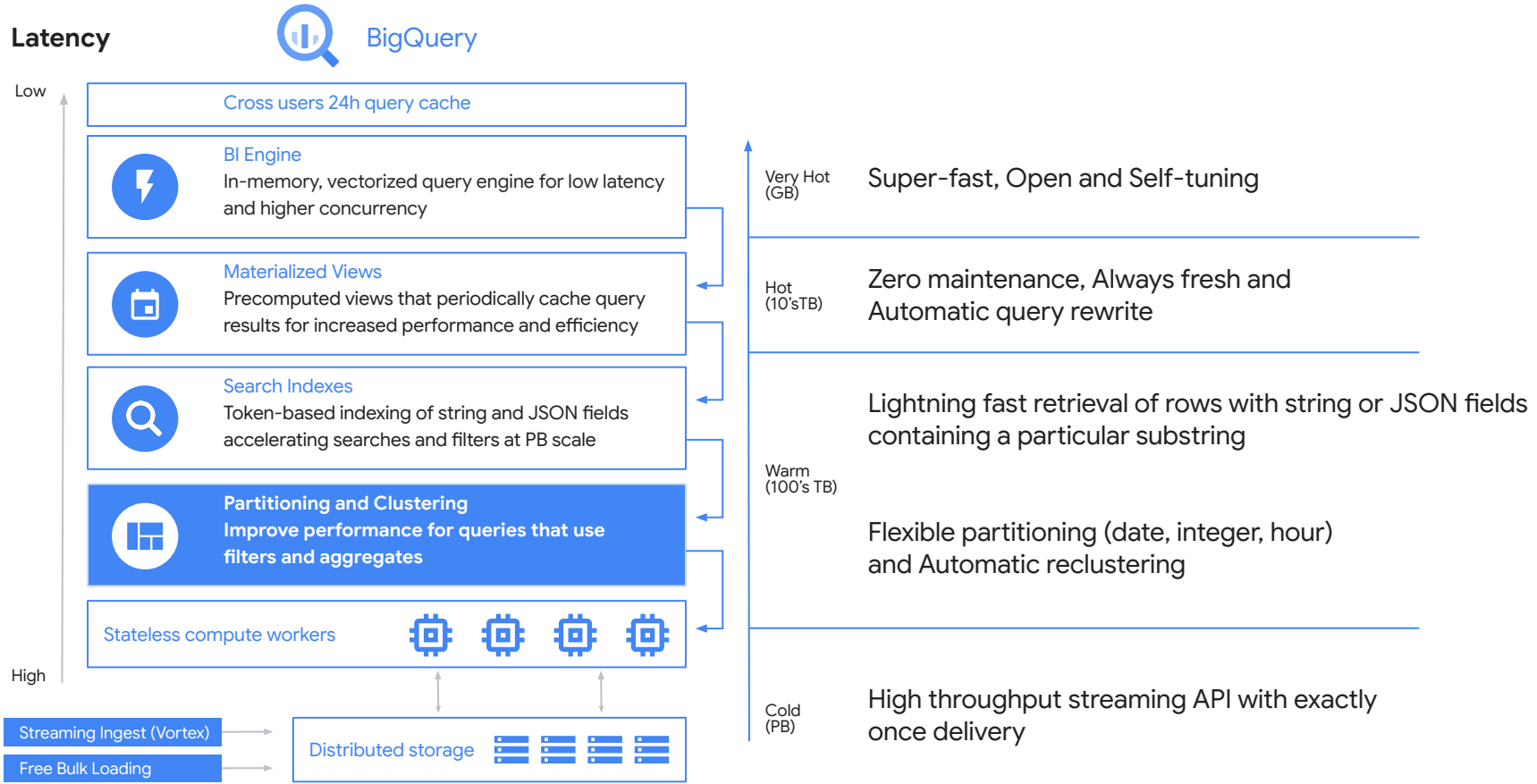
Preview

Dataset undelete

Easily recover entire datasets (all objects) within time travel window.



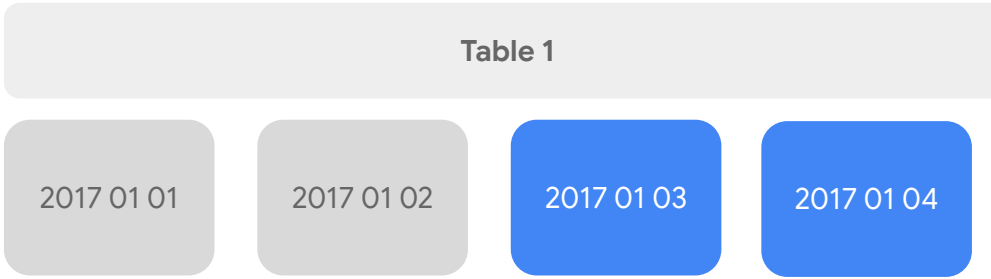
Adaptive Caching layers in BigQuery



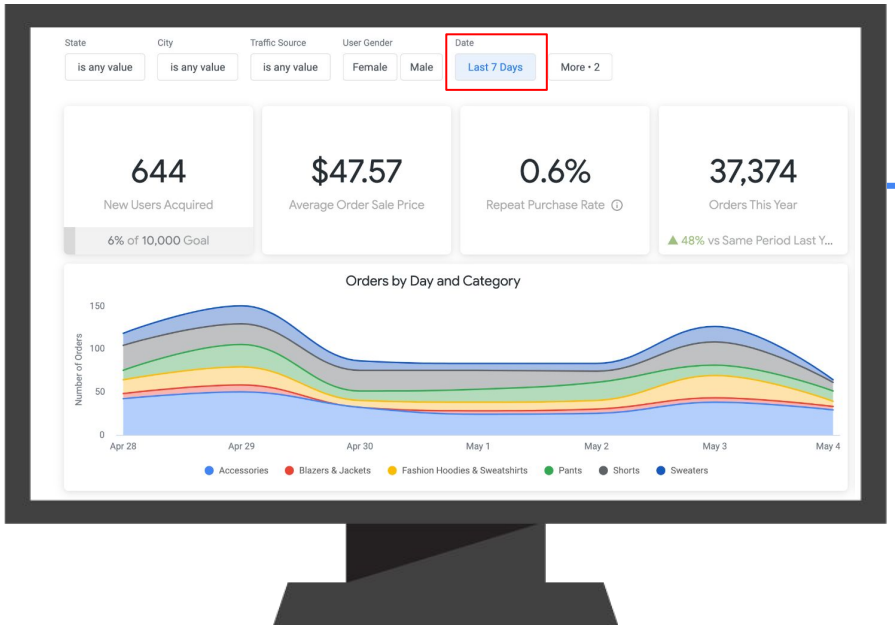
Storage Optimization

Partitioning

- Ingestion Time
- Date / Timestamp Column
- Integer Column



SELECT ...
WHERE date >= “20170103”




```
CREATE TABLE
`project-id.my_dataset.order_items
`
 (
   ORDER_ID INT64,
      ...
 )
PARTITION BY DATE(created_at)
```

Storage Optimization

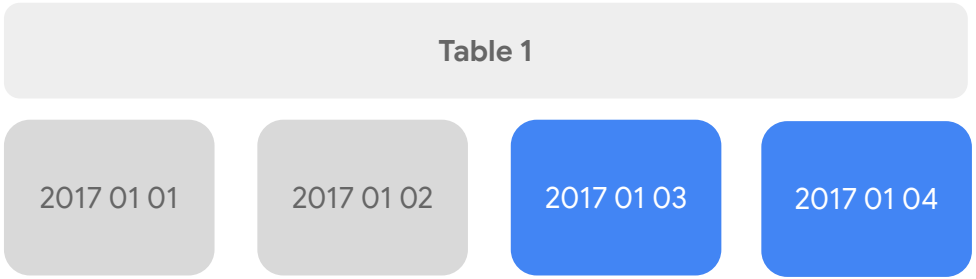
Clustering

Customer_ID	Created_At	Order_ID
1	2017-01-01	1
1	2017-01-01	1
1	2017-01-02	2
2	2017-01-01	3
3	2017-01-02	4
3	2017-01-02	4

Partitioning vs. Clustering

Partitioning

Lower Cardinality, Lots of Data



SELECT ...
WHERE date >= "20170103"

Clustering

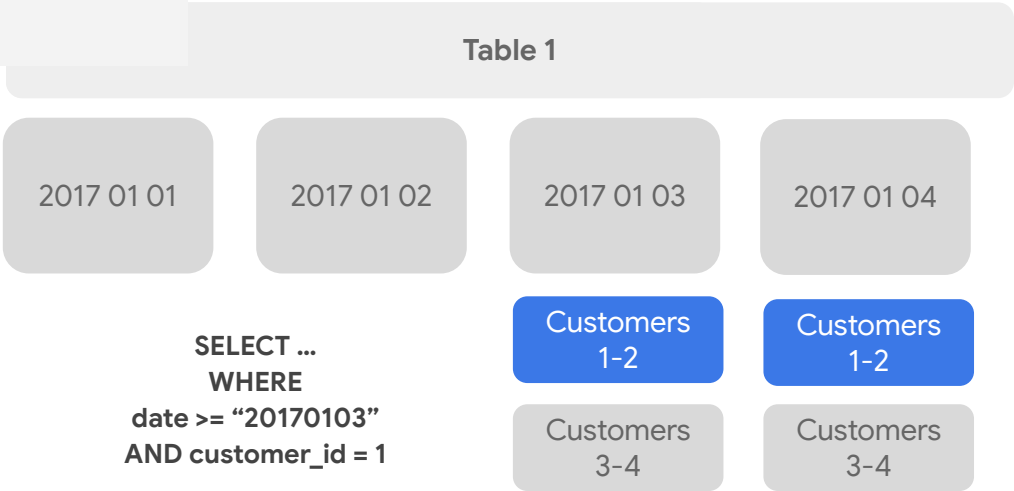
Higher Cardinality, Doesn't Need a Lot of Data

SELECT ...
WHERE customer_id = 1

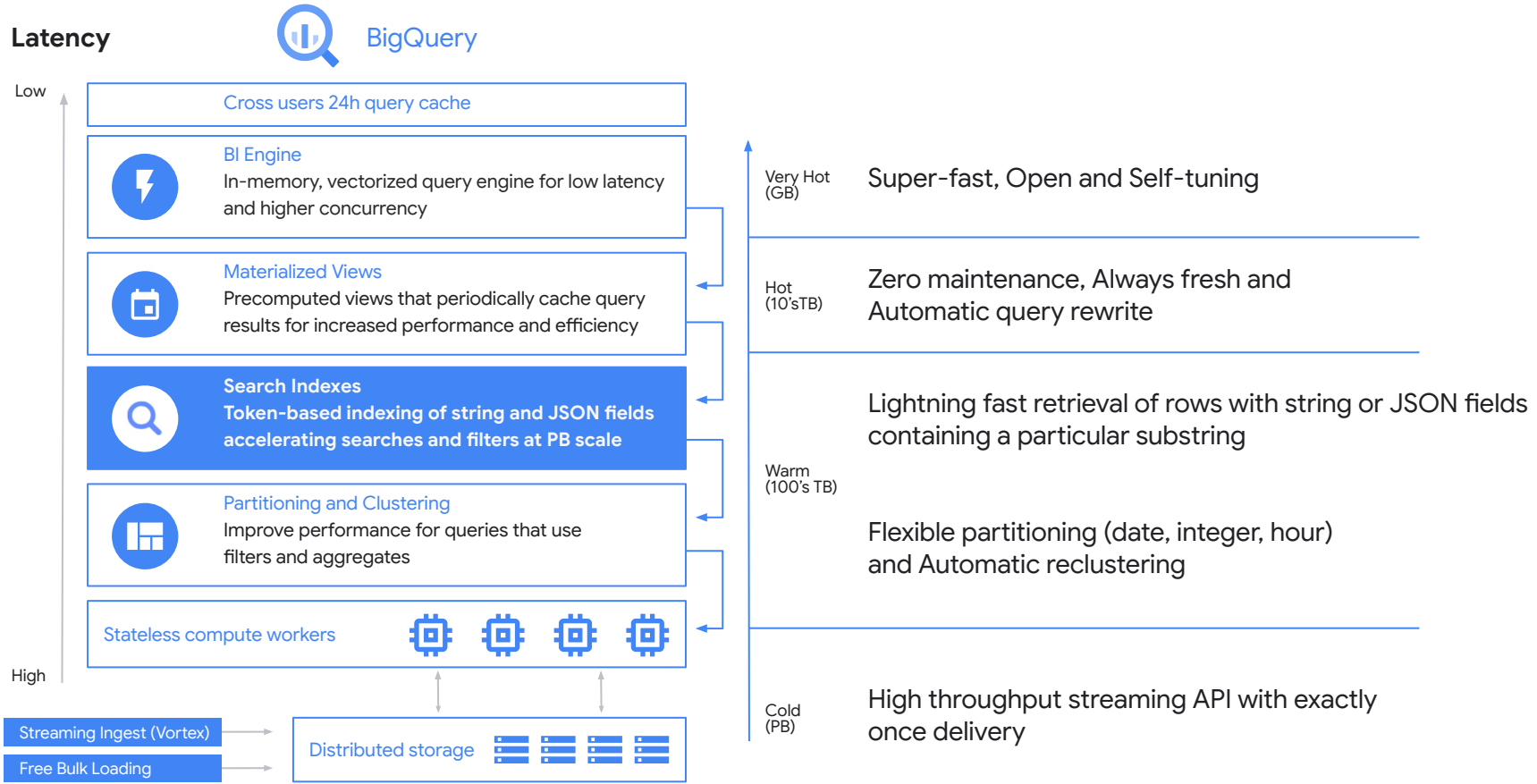
(Max 10,000 partitions per table)

Partitioning and Clustering in the same query

```
CREATE TABLE `project-id.my_dataset.order_items`
 (
   ORDER_ID INT64,
     ...
 )
PARTITION BY DATE(created_at)
CLUSTER BY customer_id
```



Adaptive Caching layers in BigQuery



Search Indexes for BigQuery

```
CREATE SEARCH INDEX bq_demo_index
ON bq_table
```

SQL

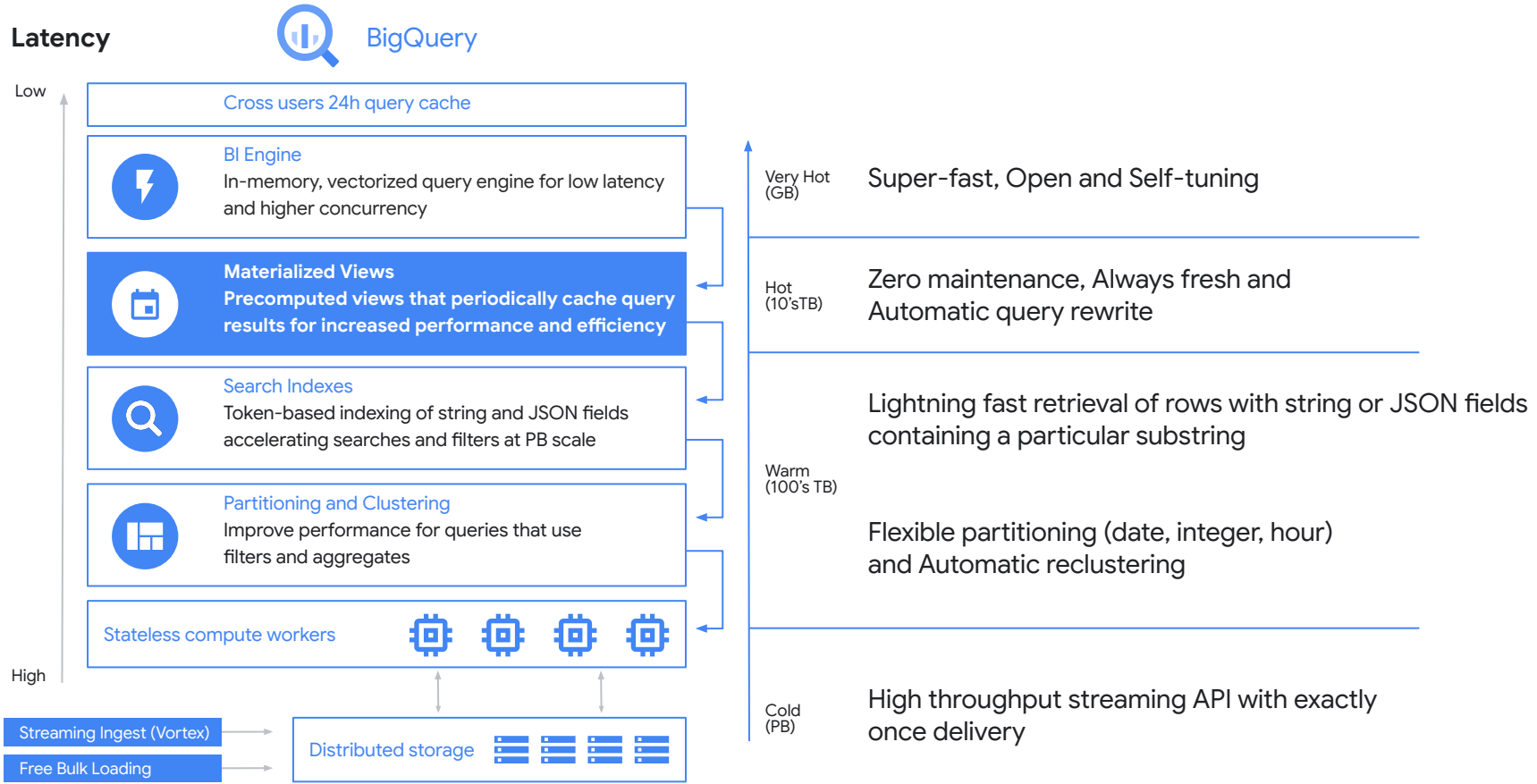
```
SELECT * FROM bq_table t
WHERE SEARCH (t, 'term')
      t.field = 'term'
      t.field like 'term'
      t.field IN('term1','term2')
      ...
```

Use Search Indexes to accelerate performance of "needle in the haystack" queries (point lookups)

Indexes auto-refresh as data changes and you are always guaranteed correct results. No management required.

Business Case	Example
Queries with filters on **JSON**	Select only rows where one of the JSON's sub-elements contains a value
Dashboards that require **highly selective filters**	Slice and dice data by different inventory items
Identify **targeted subsets** of data within a large dataset	Create a cohort of patients with a precise gene mutation.
Regulatory processes that require **finding specific data elements**	Find and delete a user's records for GDPR
Developer **troubleshooting**	Identify the application log entries associated with a particular error code
Security **audits**	Review all network activity from a specific IP address

Adaptive Caching layers in BigQuery



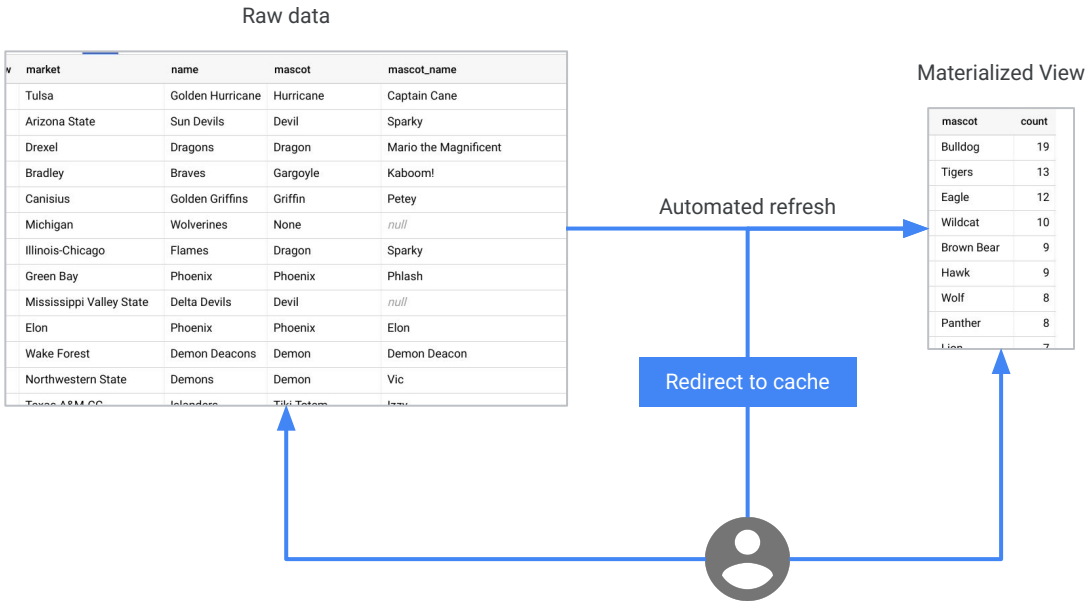
BigQuery Materialized Views

Performance - MVs are a caching layer for faster, cheaper queries

Ease of use - users define MVs, BigQuery automatically maintains and refreshes MVs

Power - Queries are automatically redirected to MVs when applicable

Consistency - data is never stale



Google Cloud

Materialized Views - Query Rewrite

Base_Table				
Col 1	Col 2	Col 3	Col 4	Col_5

BQ Optimizer

```
SELECT
COL1, SUM(COL3)
FROM Base_Table
GROUP BY COL1
```

Run Query

Materialized_View		
Col 1	Col 3	Col 4

Incremental Materialized Views

SQL Limitations

Unsupported SQL features

- Left/right/full outer joins.
- Self-joins (joins using the same table more than once).
- Window functions.
- ARRAY subqueries.
- UNION_ALL
- Non-deterministic functions such as RAND(), CURRENT_DATE(), SESSION_USER(), or CURRENT_TIME().
- User-defined functions (UDFs).
- TABLESAMPLE.
- FOR SYSTEM_TIME AS OF.

Non Incremental Materialized Views

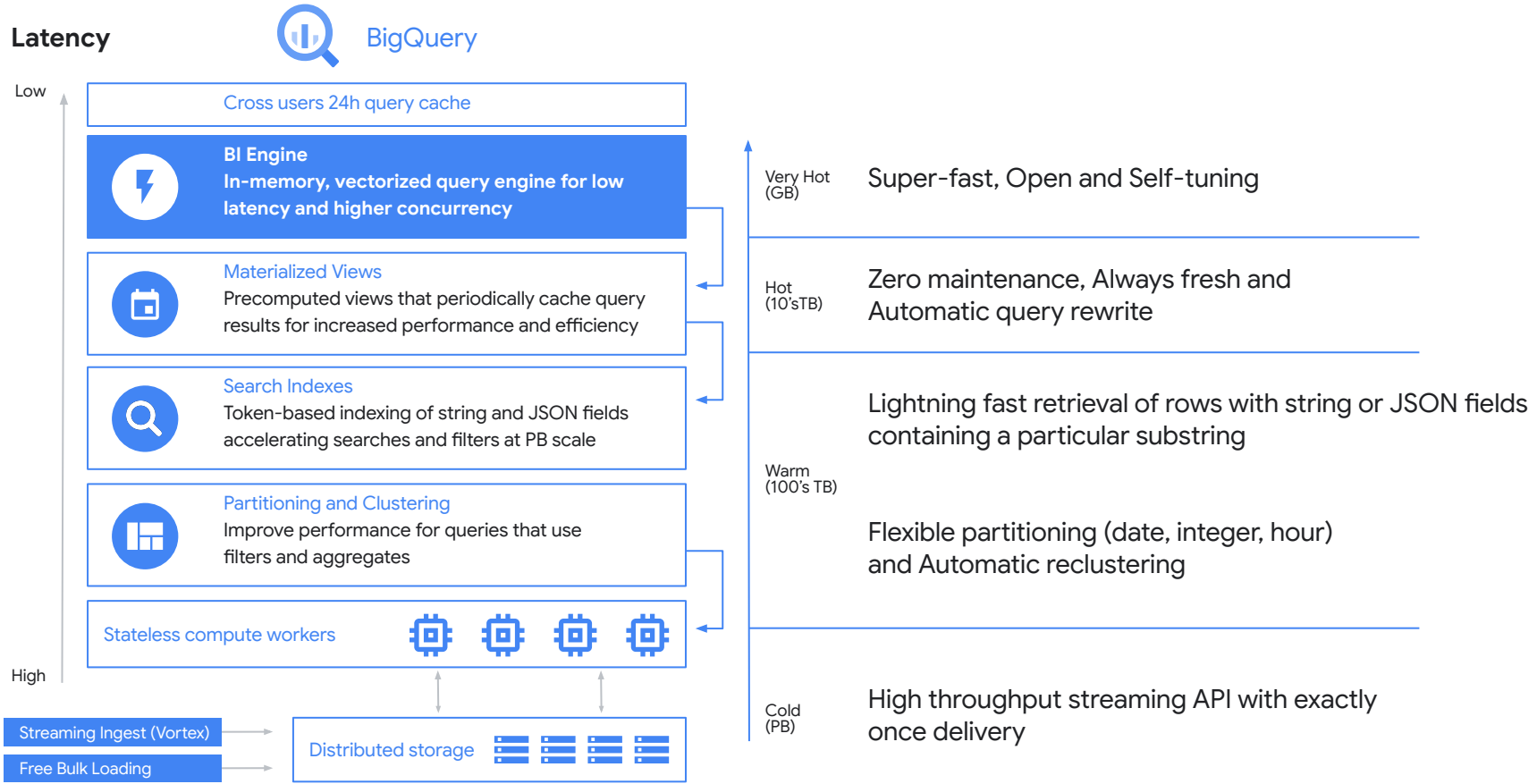
```
CREATE MATERIALIZED VIEW  project-id.my_dataset.my_mv_table
  OPTIONS
(
    enable_refresh = true, refresh_interval_minutes = 60,
    max_staleness = INTERVAL "4" HOUR,
    allow_non_incremental_definition = true
)
AS SELECT
  employee_id,
  DATE(transaction_time),
  COUNT(1) AS count
FROM my_dataset.my_base_table
GROUP BY 1, 2;
```

Allows much wider SQL Support

N.B.

- Refresh will involve the entire materialized view.
- Can only directly query the materialized view, query rewrite will not work.

Adaptive Caching layers in BigQuery



BigQuery BI Engine

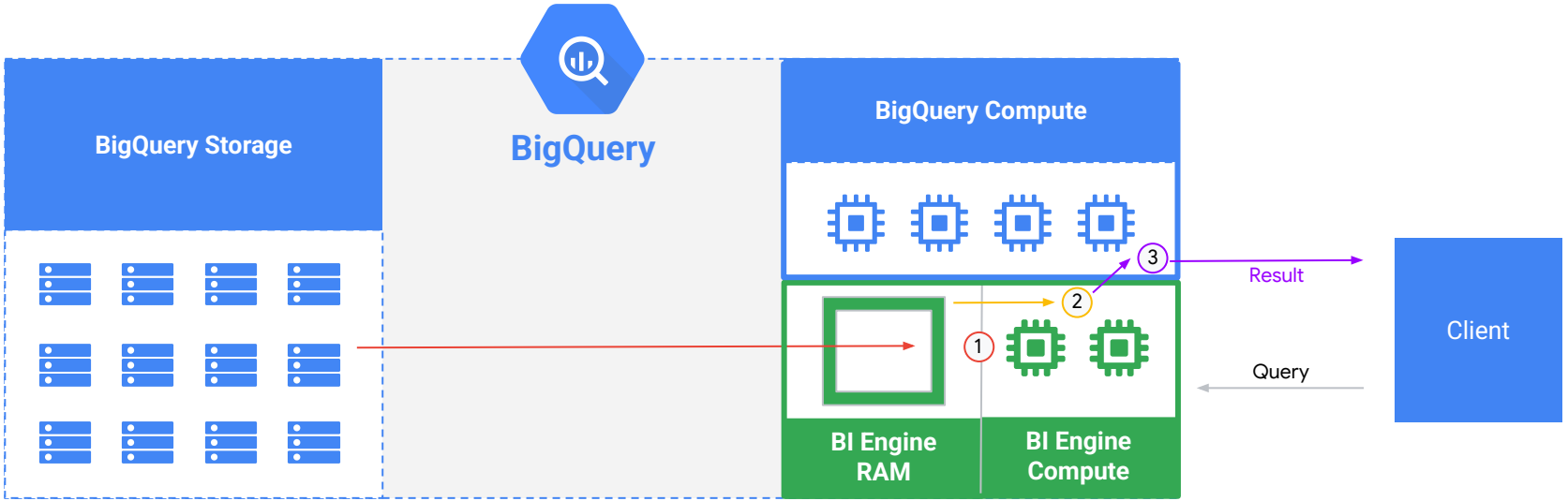
Sub-second queries

High Concurrency

Smart tuning

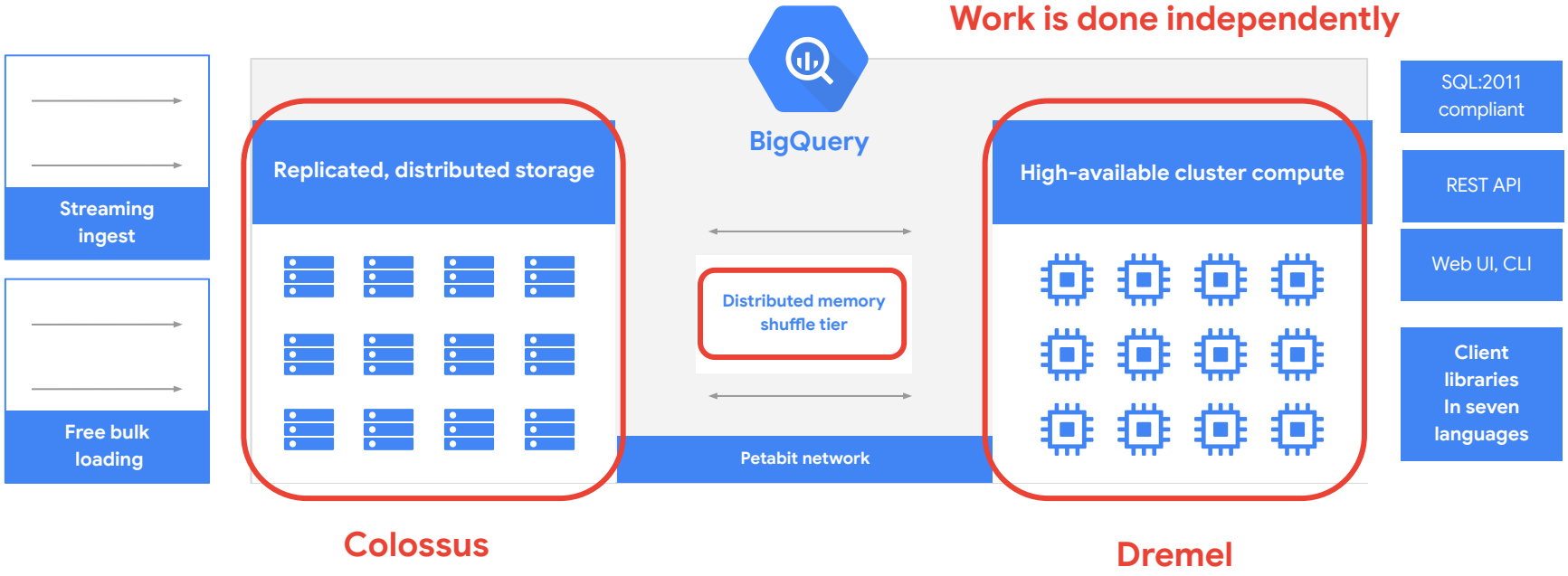
Google Cloud

BI Engine

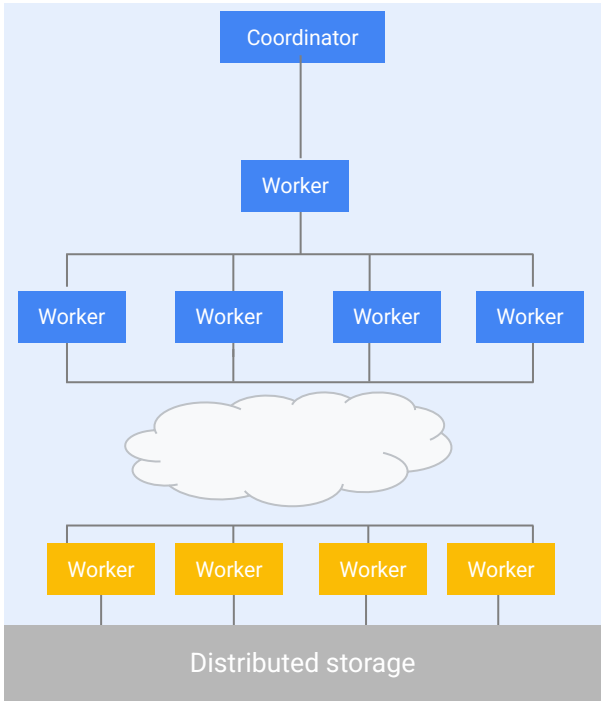


1. Check if all data required to resolve query is in RAM. If not, scan additional from storage (bytes scanned are not billed).
2. Use BI Engine compute to resolve leaf queries.
3. For complex queries, BQ compute may be used for final stages of query.

Going Deep - Query Processing Internals


Work is done independently
Streaming ingest
Free bulk loading
Replicated, distributed storage
BigQuery
Distributed memory shuffle tier
Petabit network
High-available cluster compute
SQL:2011 compliant
REST API
Web UI, CLI
Client libraries In seven languages
Colossus
Dremel

Flexible Query 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

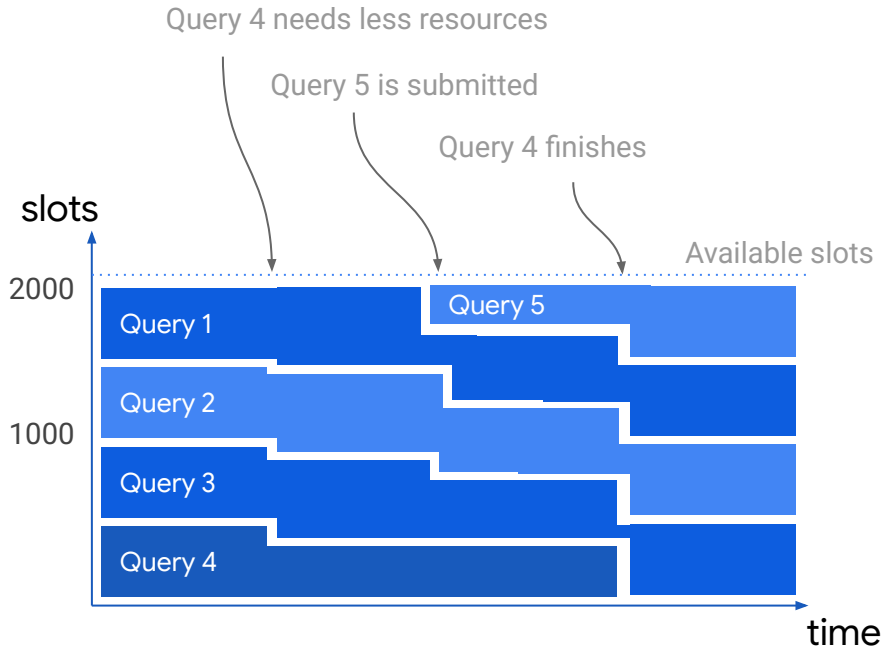
Stage 2: GROUP BY, SORT, LIMIT

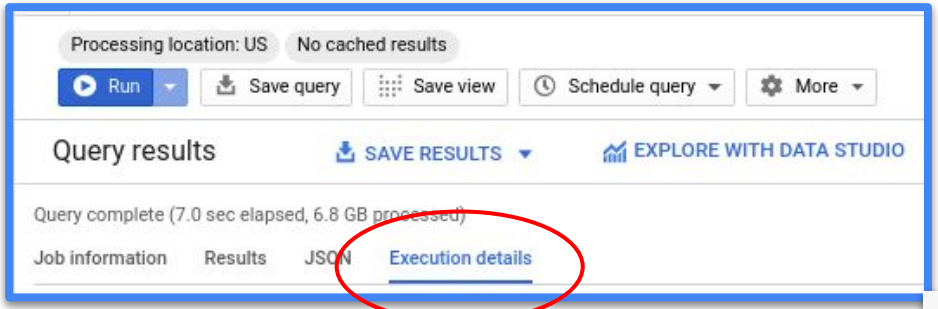
Shuffle with dynamic partitioning

Stage 1: Partial GROUP BY

Dynamic Scheduling in BigQuery

- **Dynamic central scheduler allocates slots**
- **Handles machine failure**
- **Fair resource distribution between queries**





1 2 3 4

Worker timing

Stages		Wait	Read	Compute	Write		Rows
S00: Input	Avg:	54 ms	80 ms	54 ms	2 ms	Input:	58,937,715
	Max:	111 ms	160 ms	67 ms	3 ms	Output:	19

READ
```
$1:start_station_name
FROM bigquery-public-data.new_york_citibike.citibike_trips
WHERE like($1, '%Broadway%')
```

AGGREGATE
```
$20 := COUNT_STAR()
```

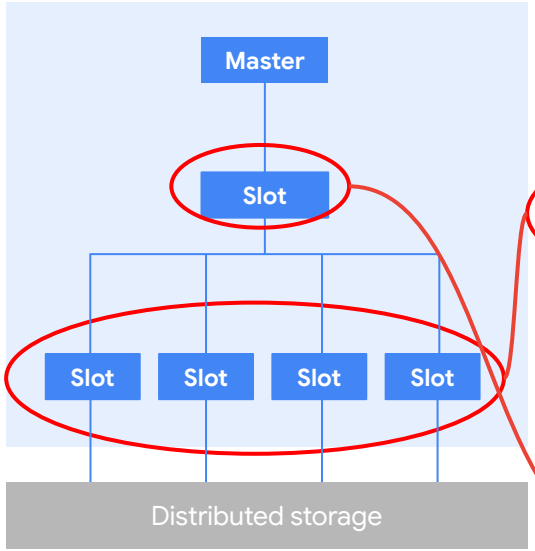
WRITE
```
$20
TO __stage00_output
```

S01: Output	Avg:	96 ms	0 ms	7 ms	3 ms	Input:	19
	Max:	96 ms	0 ms	7 ms	3 ms	Output:	1

READ
```
$20
FROM __stage00_output
```

AGGREGATE
```
$10 := SUM_OF_COUNTS($20)
```

WRITE
```
$10
TO __stage01_output
```



Worker timing

Stages		Wait	Read	Compute	Write		Rows
S00: Input	Avg:	54 ms	80 ms	54 ms	2 ms	Input:	58,937,715
	Max:	111 ms	160 ms	67 ms	3 ms	Output:	19

READ
```
$1:start_station_name
FROM bigquery-public-data.new_york_citibike.citibike_trips
WHERE like($1, '%Broadway%')
```

AGGREGATE
```
$20 := COUNT_STAR()
```

WRITE
```
$20
TO __stage00_output
```

S01: Output	Avg:	96 ms	0 ms	7 ms	3 ms	Input:	19
	Max:	96 ms	0 ms	7 ms	3 ms	Output:	1

READ
```
$20
FROM __stage00_output
```

AGGREGATE
```
$10 := SUM_OF_COUNTS($20)
```

WRITE
```
$10
TO __stage01_output
```

- **Focus on stages dominating resources**
- **Pinpoint data skew** (large diff in average and max time spent)
- **Reduce CPU tasks** (consider approx functions)

```
job = client.get_job(
            "bquxjob_7ae65493_179f5db3ab5"
            , location="US")
```

```
SELECT
 *
FROM
`region-us`.INFORMATION_SCHEMA.JOBS_TIMELINE_BY_PROJECT
WHERE
 job_id = "bquxjob_7ae65493_179f5db3ab5"
```

Going Deep -How to Optimise Queries

Optimization: Necessary columns only

Original code

```
SELECT
  *
FROM
`dataset.table`
```

Optimized

```
SELECT
  * EXCEPT (dim1, dim2)
FROM
`dataset.table`
```

Don't do "`select * limit 10`" to look at data, use the **preview** feature instead.

Optimization: Auto-pruning with Partitioning & Clustering

Table info

Table ID	bigquery-public-data:wikipedia.pageviews_2021
Table size	1.1 TB
Long-term storage size	497.28 GB
Number of rows	24,870,308,515
Created	Dec 31, 2020, 6:00:43 PM UTC-8
Last modified	Jun 11, 2021, 7:02:21 AM UTC-7
Table expiration	NEVER
Data location	US
Description	Wikipedia pageviews from http://dumps.wikimedia.you
Table Type	Partitioned
Partitioned by	DAY
Partitioned on field	datehour
Partition expiration	
Partition filter	Required
Clustered by	wiki title

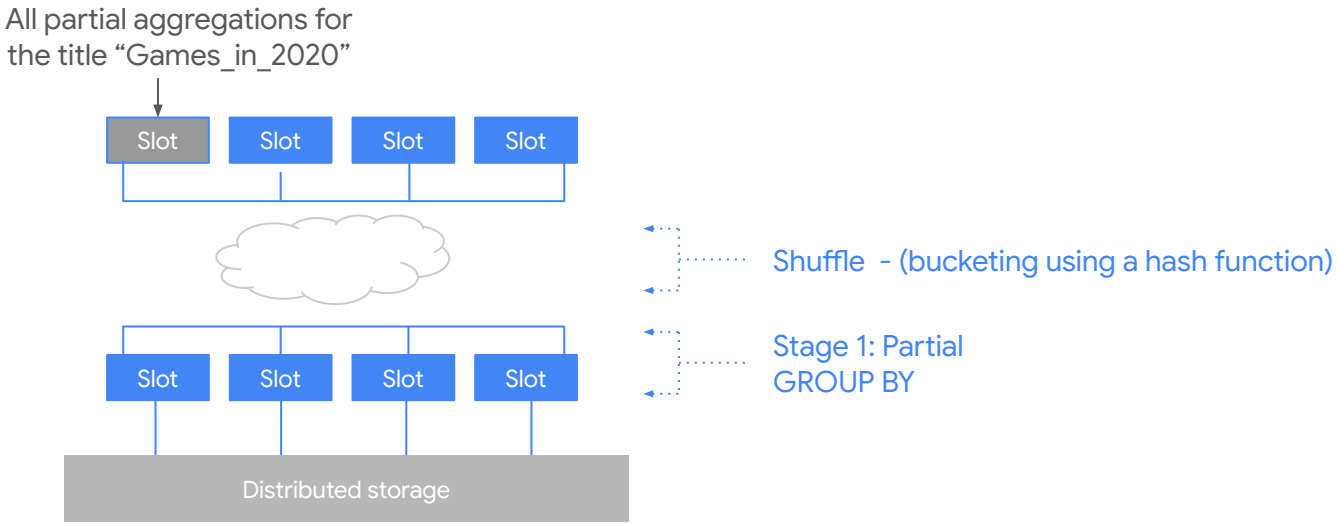
Original code

```
SELECT
  ...
FROM
  `wikipedia.pageviews_2021`
```

Optimized

```
SELECT
  ...
FROM
  `wikipedia.pageviews_2021`
WHERE
  DATE(datehour) = "2021-06-11"
  AND wiki="simple"
```

```sql
SELECT title, SUM(views)
FROM `wikipedia.pageviews_2021`
WHERE DATE(datehour) = "2021-06-11"
GROUP BY title
ORDER BY SUM(views) DESC
LIMIT 1000
```



```
SELECT title, SUM(views)
FROM `wikipedia.pageviews_2021`
WHERE DATE(datehour) = "2021-06-11"
GROUP BY title
ORDER BY SUM(views) DESC
LIMIT 1000
```

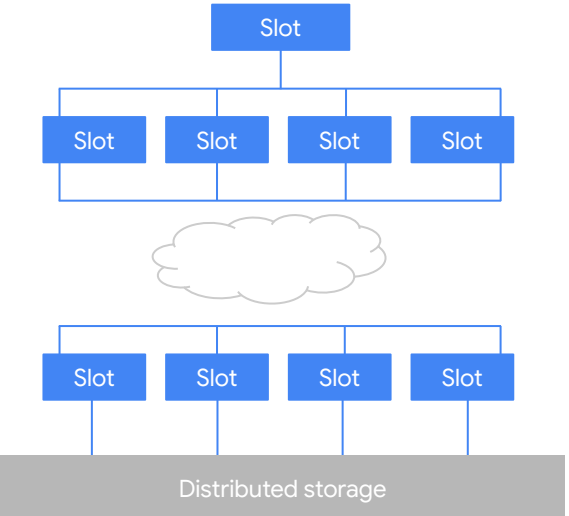
Worker timing

Stages		Wait	Read	Compute	Write	Rows
S00: Input	Avg:	236 ms	55 ms	723 ms	2 ms	Input: 81,888,146
	Max:	301 ms	205 ms	1571 ms	7 ms	Output: 2,635
S01: Sort+	Avg:	7 ms	0 ms	5 ms	1 ms	Input: 2,635
	Max:	8 ms	0 ms	7 ms	1 ms	Output: 868
S02: Output	Avg:	8 ms	0 ms	10 ms	10 ms	Input: 868
	Max:	8 ms	0 ms	10 ms	10 ms	Output: 868

READ
```
$2:title, $3:views, $1:datehour
FROM bigquery-public-data.wikipedia.pageviews_2021
WHERE and(equal(date($1), 18789), like($2, '%Deaths%'))
```

AGGREGATE
```
GROUP BY $30 := $2
$20 := SUM($3)
```

WRITE
```
$30, $20
TO __stage00_output
BY HASH($30)
```



Stage 3: SORT, LIMIT (1 slot)

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

Shuffle

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

Optimization: Late aggregation

Original code

```
SELECT
  t1.dim1,
  SUM(t1.m1)
  SUM(t2.m2)
FROM (SELECT
    dim1,
    SUM(metric1) m1
  FROM `dataset.table1` GROUP BY 1) t1
JOIN (SELECT
    dim1,
    SUM(metric2) m2
  FROM `dataset.table2` GROUP BY 1) t2
ON t1.dim1 = t2.dim1
GROUP BY 1;
```

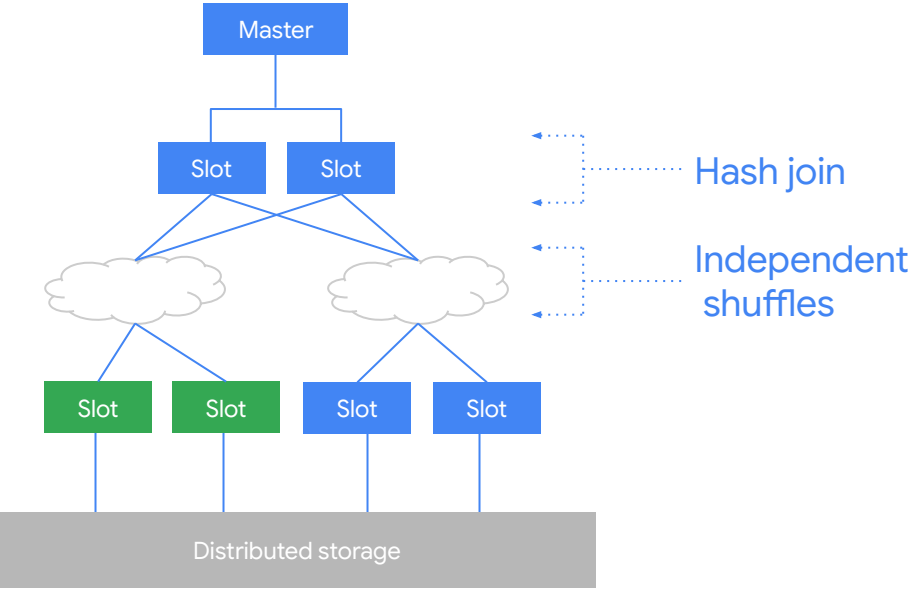
Optimized

```
SELECT
  t1.dim1,
  SUM(t1.m1)
  SUM(t2.m2)
FROM (SELECT
    dim1,
    metric1 m1
  FROM `dataset.table1`) t1
JOIN (SELECT
    dim1,
    metric2 m2
  FROM `dataset.table2`) t2
ON t1.dim1 = t2.dim1
GROUP BY 1;
```

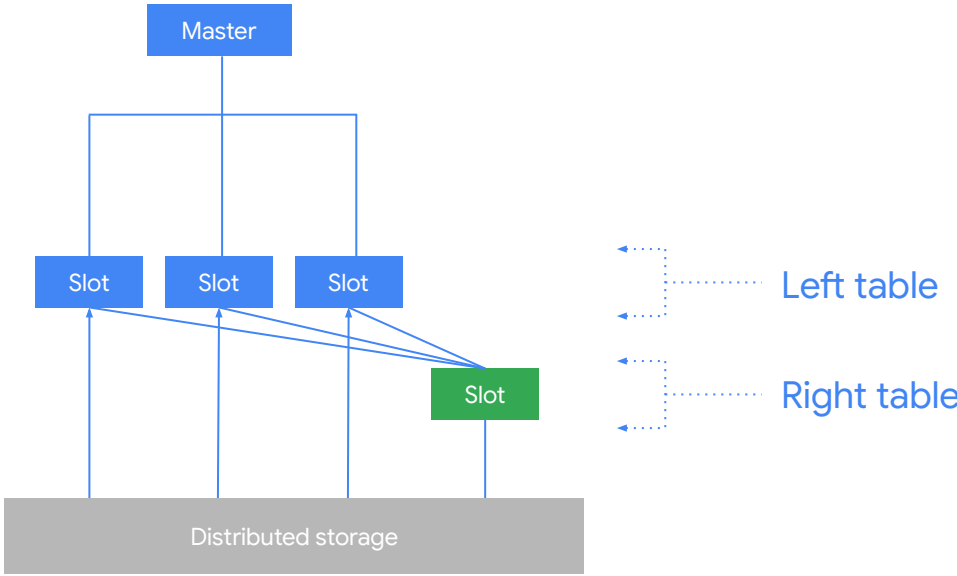
Avoid multiple group by executions, by moving them up.

Unless If a table can be reduced drastically by aggregating in preparation for being joined, then aggregate it early.

Large JOIN (shuffle)



Small JOIN (broadcast)



Optimization: JOIN pattern - largest table first

Original code

```
SELECT
  t1.dim1,
  SUM(t1.metric1),
  SUM(t2.metric2)
FROM
  `dataset.small_table` t1
JOIN
  `dataset.large_table` t2
ON
  t1.dim1 = t2.dim1
WHERE t1.dim1 = 'abc'
GROUP BY 1;
```

Optimized

```
SELECT
  t1.dim1,
  SUM(t1.metric1),
  SUM(t2.metric2)
FROM
  `dataset.large_table` t2
JOIN
  `dataset.small_table` t1
ON
  t1.dim1 = t2.dim1
WHERE t1.dim1 = 'abc'
GROUP BY 1;
```

Optimization: Filter before JOINs

Filter both tables before the join to reduce amount of data

Original code

```
SELECT
  t1.dim1,
  SUM(t1.metric1)
FROM
  `dataset.table1` t1
LEFT JOIN
  `dataset.table2` t2
ON
  t1.dim1 = t2.dim1
WHERE t2.dim2 = 'abc'
GROUP BY 1;
```

Optimized

```
SELECT
  t1.dim1,
  SUM(t1.metric1)
FROM
  `dataset.table1` t1
LEFT JOIN
  `dataset.table2` t2
ON
  t1.dim1 = t2.dim1
WHERE t2.dim2 = 'abc' AND t1.dim2 = 'abc'
GROUP BY 1;
```

WHERE clause: Expression order.

Original code

```
SELECT
  text
FROM
  `stackoverflow.comments`
WHERE
  text LIKE '%java%'
  AND user_display_name = 'anon'
```

Optimized

```
SELECT
  text
FROM
  `stackoverflow.comments`
WHERE
  user_display_name = 'anon'
  AND text LIKE '%java%'
```

The first part of your where clause should always contain the filter that will eliminate the most data.

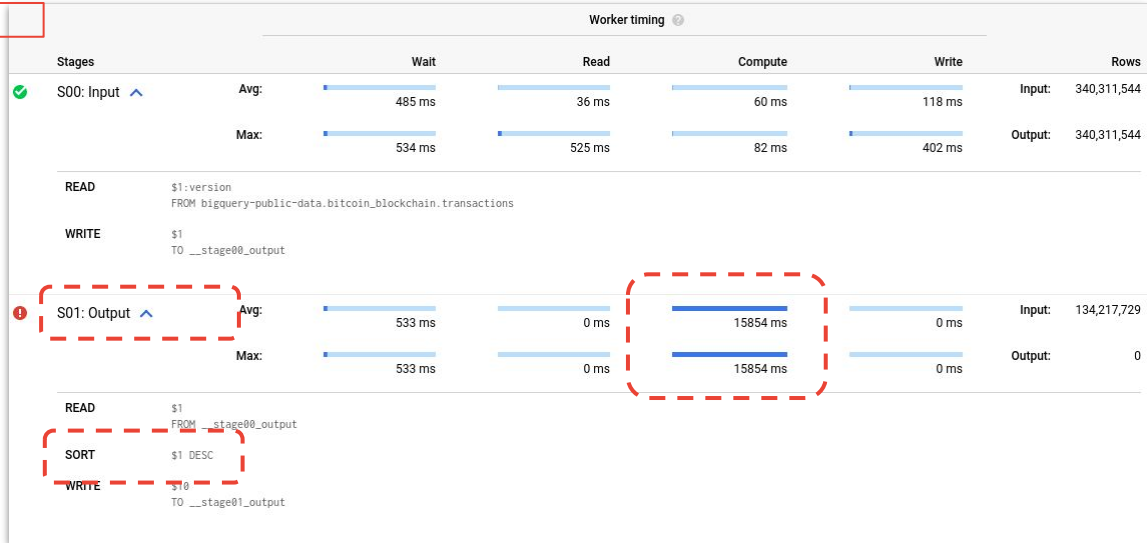
Optimization: ORDER BY with LIMIT

Original code

```
SELECT
  t.dim1,
  t.dim2,
  t.metric1
FROM
`dataset.table` t
ORDER BY t.metric1 DESC
```

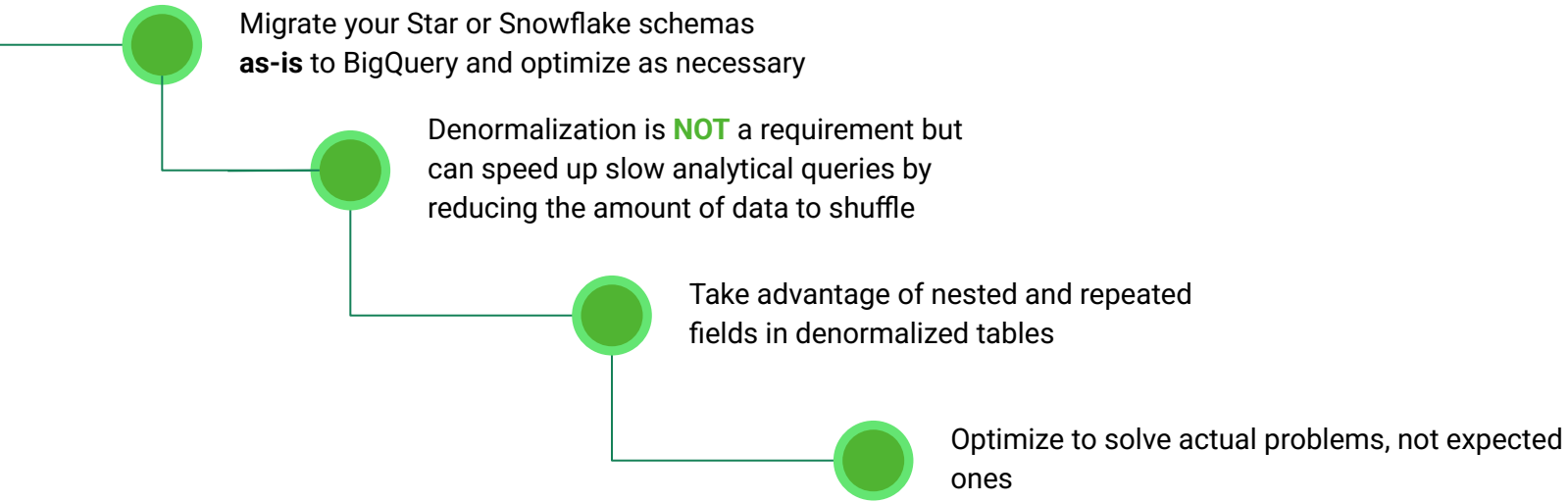
Optimized

```
SELECT
  t.dim1,
  t.dim2,
  t.metric1
FROM
  `dataset.table` t
ORDER BY t.metric1 DESC
LIMIT 1000
```



Schema design and data organisation

Schema design in a nutshell



Nested and repeated columns

Orders		
Customer	Order_id	Date
Bob	1234	2020/01/02
Lisa	4321	2020/01/12
John	4567	2020/02/01

Order_items		
Order_id	Product	Quantity
1234	Jeans	2
1234	Shirt	4
4321	Skirt	1
4567	Boots	1
4567	Jeans	3
3567	Hat	1

Orders				
Customer	Order_id	Date	Item.Product	Item.Quantity
Bob	1234	2020/01/02	Jeans	2
			Shirt	4
Lisa	4321	2020/01/12	Skirt	1
John	4567	2020/02/01	Boots	1
			Jeans	3
			Hat	1

More natural representation of hierarchical data

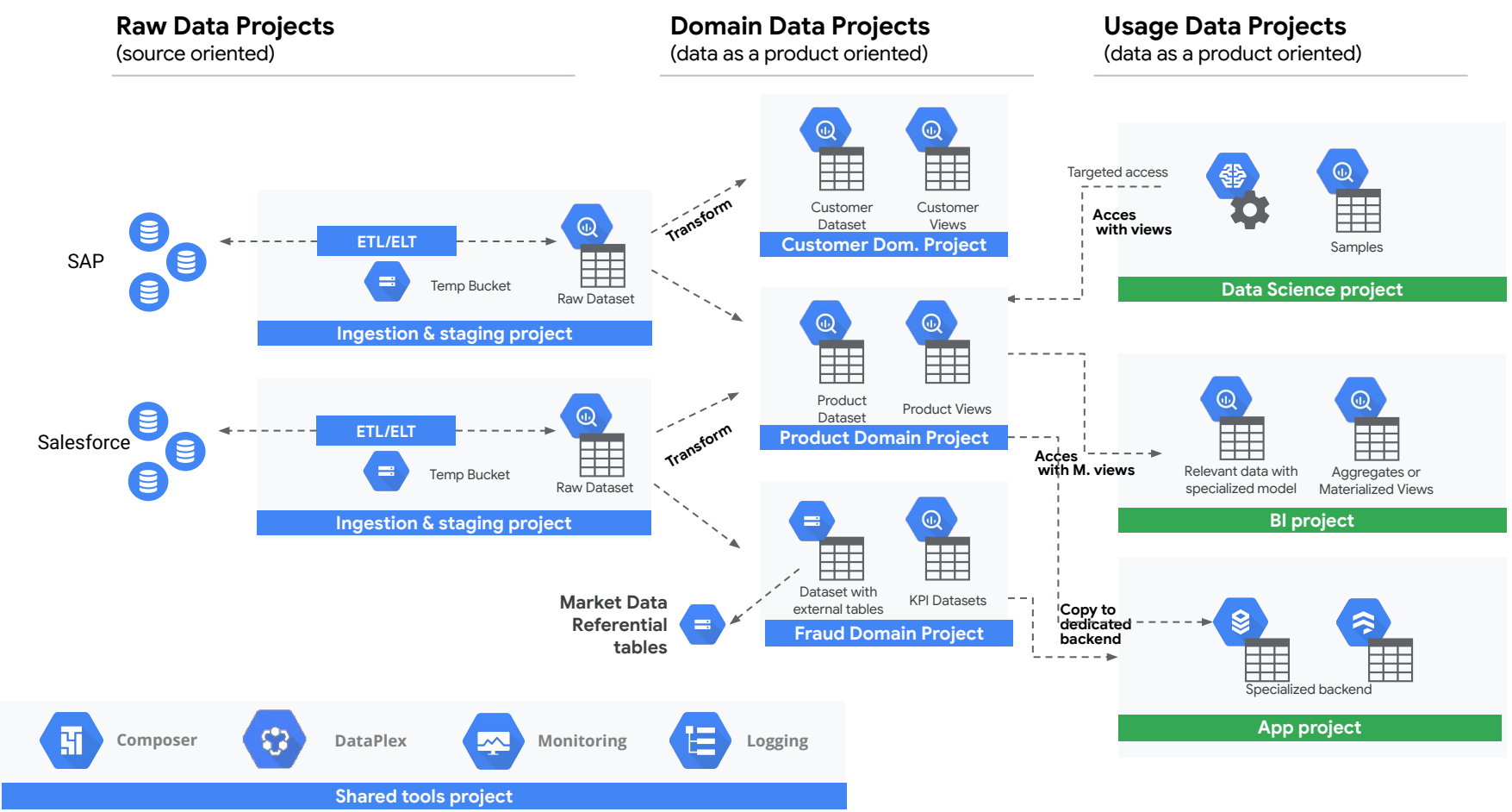
Can be accessed via dot notation: name vs. `Item[0].Quantity`

Leverage Nested and Repeated fields for:

- Tightly-coupled or Immutable relationships
  - Session w/ Events
  - Order w/ Line Items
- Infrequently changing data
  - Country, Region, Date, etc

Google Cloud

Store and organize your data on BigQuery (example)



Thank You

Summary

- Storage
  - Partitioning
  - Clustering
  - Search indexes
  - Materialized views
  - Bi Engine
- Compute
  - Slots
  - Query execution
- SQL best practices
- Q&A

Q&A