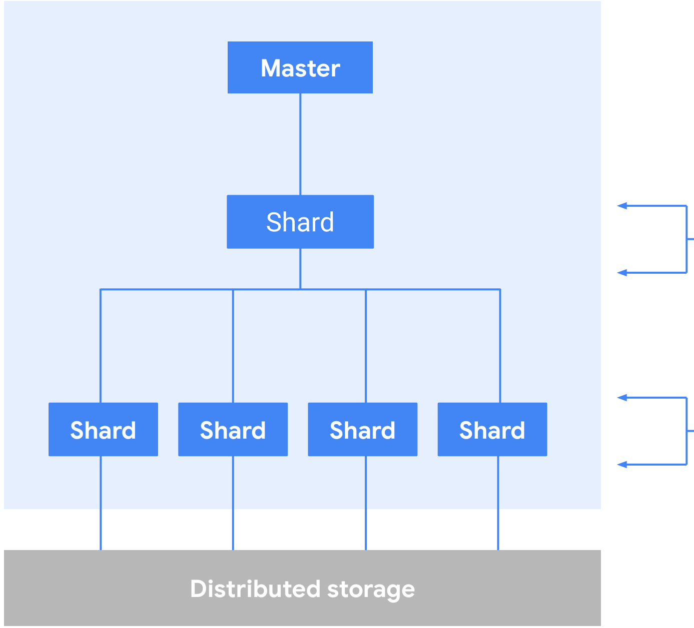
Query Processing and Optimization

Simple query execution

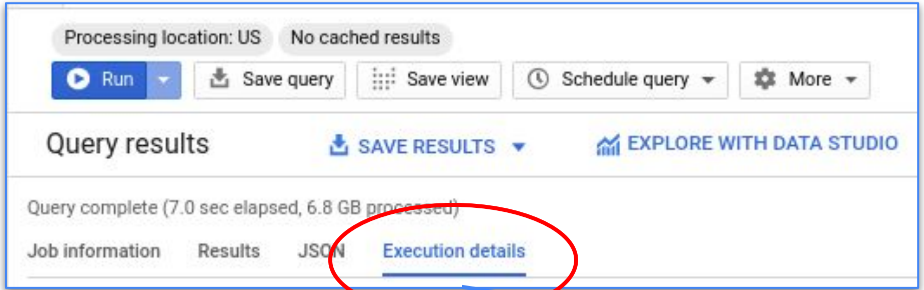



```
SELECT COUNT(*)
FROM `new_york.citibike_trips`
WHERE start_station_name LIKE
"%Broadway%"
```

Stage 2: Sum (1 slot)

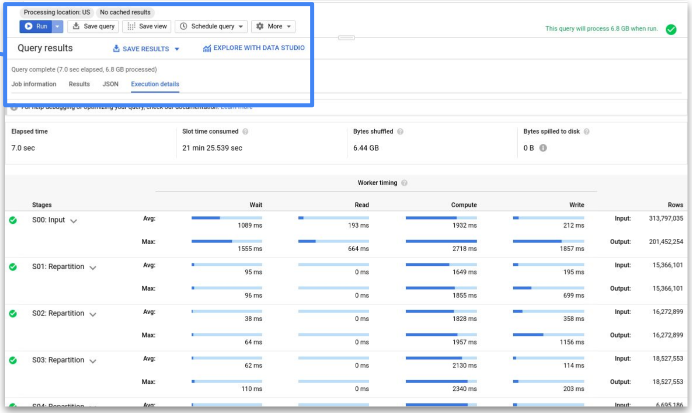
Stage 1: Filter, count (220 slots)

Viewing the query plan

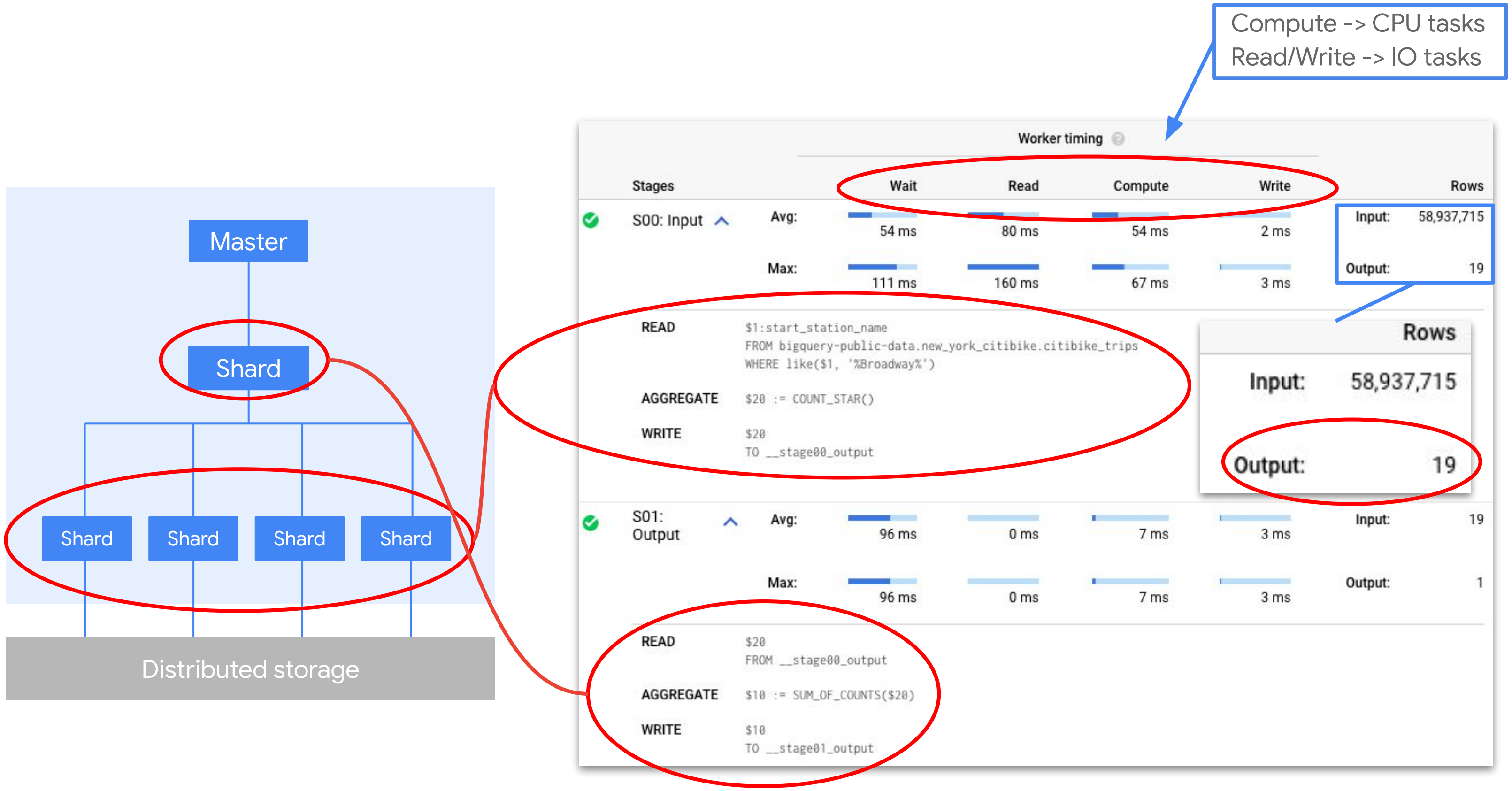


Example Query

SELECT COUNT(*) FROM \`bigquery-public-data.new_york.citibike_trips\` WHERE start_station_name LIKE "%Broadway%"



Simple query execution - Query plan



Query cache

Hash of

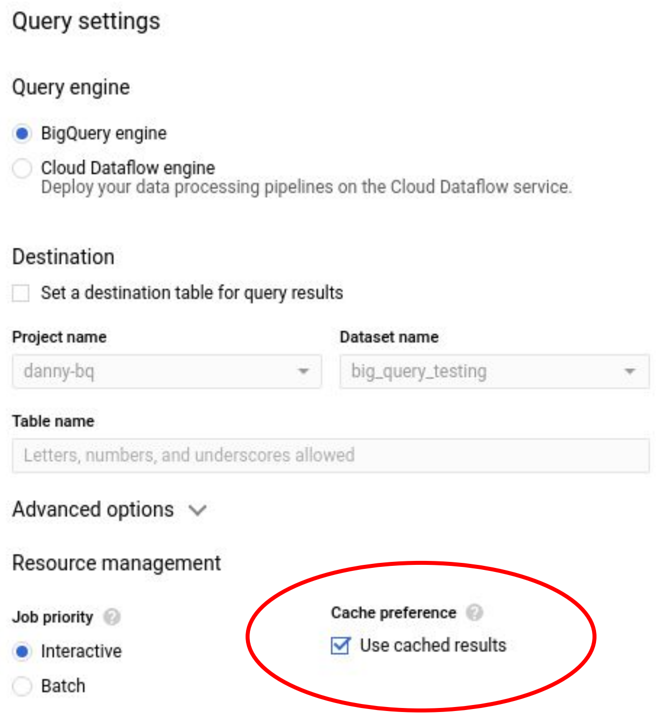
- Data modification times
- Tables used

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



Interpreting the query plan

- **Significant difference between avg and max time?**
  - Probably data skew—use APPROX_TOP_COUNT to check
  - Filter early to workaround
- **Most time spent reading from intermediate stages**
  - Consider filtering earlier in the query
- **Most time spent on CPU tasks**
  - Consider approximate functions, inspect UDF usage, filter earlier

How do you optimize queries?

Less work ⟶ Faster query

What is **work** for a query?

- **I/O** — How many bytes did you read?
- **Shuffle** — How many bytes did you pass to the next stage?
- **Grouping** — How many bytes do you pass to each group?
- **Materialization** — How many bytes did you write?
- **CPU work** — User-defined functions (UDFs), functions

SELECTs

Optimization: Necessary columns only

Original code

```
select
  *
from
`dataset.table`
```

Optimized

```
select
  * EXCEPT (dim1, dim2)
from
`dataset.table`
```

Reasoning

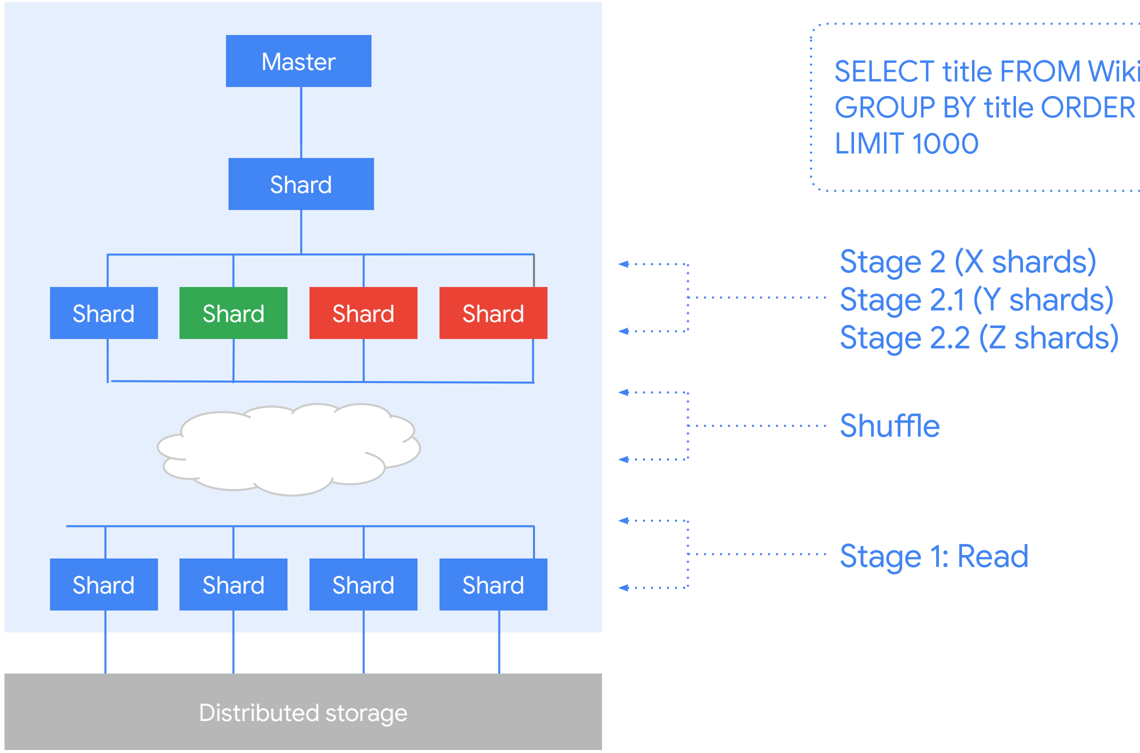
Only select the columns necessary, especially in inner queries. **SELECT** * is cost inefficient and may also hurt performance.

If the number of columns to return is large, consider using **SELECT** * **EXCEPT** to exclude unneeded columns.

In some use cases, **SELECT** * **EXCEPT** may be necessary.

Aggregation

Repartitioning



SELECT title FROM Wiki10B
GROUP BY title ORDER BY title
LIMIT 1000

Repartitioning in query plan

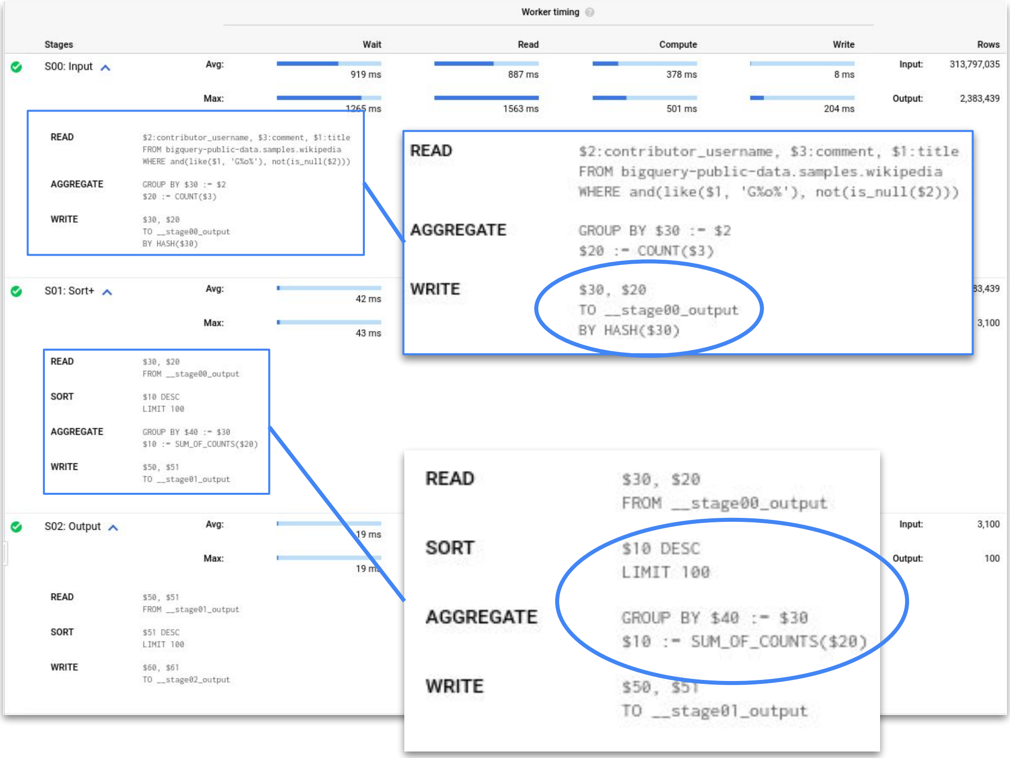
```
SELECT title
FROM wikipedia
GROUP BY title
ORDER BY title
LIMIT 1000
```

BigQuery dynamically repartitions data to improve distribution throughout slot workers

Stages		Wait	Read	Compute	Write	Rows	
		Worker timing					
S00: Input	Avg:	953 ms	125 ms	2030 ms	153 ms	Input:	313,797,035
	Max:	1372 ms	588 ms	2995 ms	839 ms	Output:	201,452,254
S01: Repartition	Avg:	1 ms	0 ms	1409 ms	67 ms	Input:	10,226,939
	Max:	1 ms	0 ms	1612 ms	164 ms	Output:	10,226,939
S02: Repartition	Avg:	0 ms	0 ms	1736 ms	78 ms	Input:	12,308,020
	Max:	1 ms	0 ms	1879 ms	142 ms	Output:	12,308,020
S03: Repartition	Avg:	1 ms	0 ms	1790 ms	96 ms	Input:	15,269,120
	Max:	1 ms	0 ms	1893 ms	189 ms	Output:	15,269,120
S04: Repartition	Avg:	1 ms	0 ms	2143 ms	126 ms	Input:	12,185,996
	Max:	1 ms	0 ms	2494 ms	198 ms	Output:	12,185,996
S05: Sort+	Avg:	4 ms	0 ms	480 ms	9 ms	Input:	201,452,254
	Max:	19 ms	0 ms	770 ms	80 ms	Output:	310,000
S06: Output	Avg:	7 ms	0 ms	86 ms	9 ms	Input:	310,000
	Max:	7 ms	0 ms	86 ms	9 ms	Output:	1,000

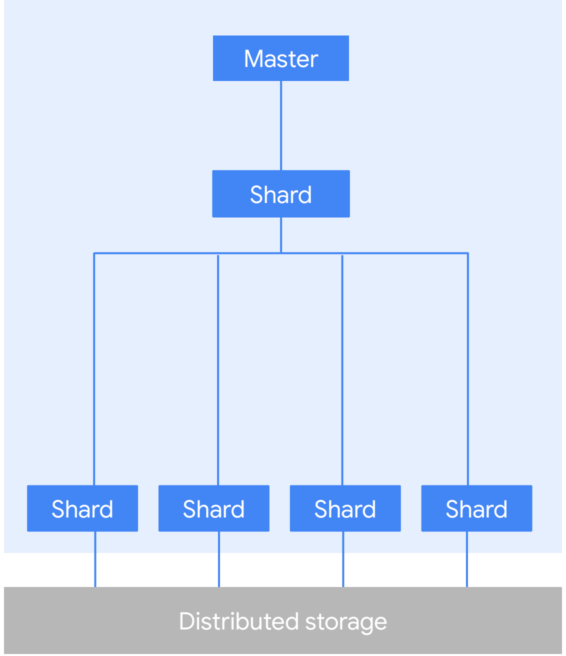
Aggregation with shuffle query plan

```
SELECT

contributor_username,
    COUNT(comment) as
comments
FROM
`bigquery-public-data.
samples.wikipedia`
WHERE title LIKE
"G%o%" AND
contributor_username
IS NOT NULL
GROUP BY 1
ORDER BY 2 DESC
LIMIT 100
```



Shuffle aggregation execution

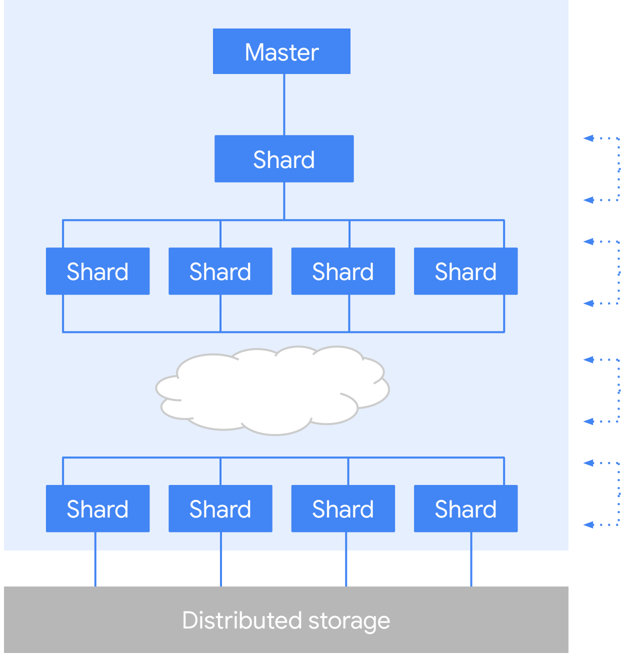
Simple select query



VS

Aggregation query

```
SELECT title
FROM Wiki10B
GROUP BY title
ORDER BY title
LIMIT 1000
```



Stage 3: SORT, LIMIT (1 slot)

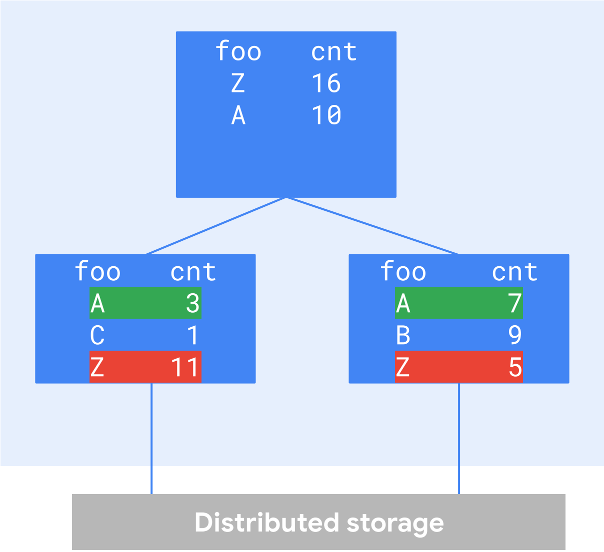
Stage 2: GROUP BY, SORT, LIMIT (289 slots)

Shuffle

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

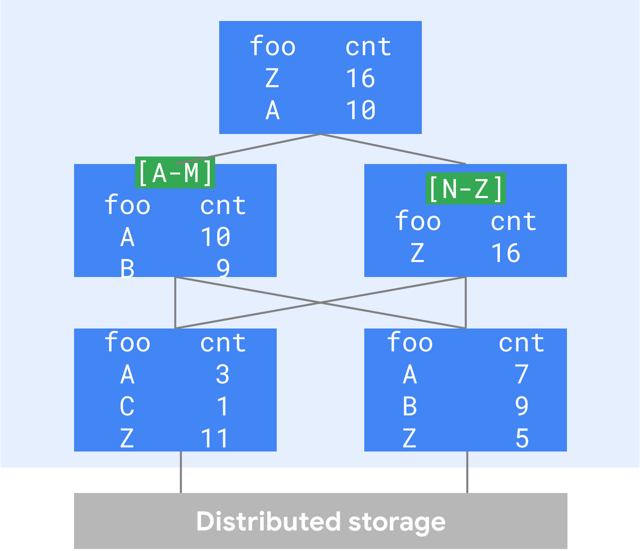
Aggregation with high cardinality

Without shuffle (non-BQ)



- Can't discard "B" or "C" until after all previous stages are complete.

With Shuffle (BQ)




```
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 *N* from each node in middle tier

Auto-pruning with partitioning and clustering

Partitioned table

Table info

Table ID	fh-bigquery:wikipedia_v2.pageviews_2017
Table size	2.2 TB
Long-term storage size	2.2 TB
Number of rows	54,489,325,868
Created	Feb 27, 2018, 1:54:41 AM
Table expiration	Never
Last modified	Feb 27, 2018, 4:47:50 AM
Data location	US
Table type	Partitioned
Partitioned by	Day
Partitioned on field	datehour
Partition filter	Required

Partitioned and clustered table

Table info

Table ID	fh-bigquery:wikipedia_v3.pageviews_2017
Table size	2.2 TB
Long-term storage size	2.2 TB
Number of rows	54,489,325,868
Created	Aug 1, 2018, 1:24:57 AM
Table expiration	Never
Last modified	Aug 2, 2018, 8:50:32 PM
Data location	US
Table type	Partitioned
Partitioned by	Day
Partitioned on field	datehour
Partition filter	Required
Clustered by	wiki, title

Auto-pruning with partitioning and clustering

Partitioned table by datehour

```
SELECT *
FROM `fh-bigquery.wikipedia_v2.pageviews_2017`
WHERE DATE(datehour) BETWEEN '2017-06-01' AND
'2017-06-30'
LIMIT 1
```

1.7 sec elapsed, 180 GB processed

Partitioned table by datehour
Clustered table by wiki, title

```
SELECT *
FROM `fh-bigquery.wikipedia_v3.pageviews_2017`
WHERE DATE(datehour) BETWEEN '2017-06-01' AND
'2017-06-30'
LIMIT 1
```

1.8 sec elapsed, 112 MB processed

Partitioning and clustering caveat

Partitioned table by date-month (fake)
Clustered table by name

```
SELECT name, state, ARRAY_AGG(STRUCT(date,temp) ORDER
BY temp DESC LIMIT 5) top_hot, MAX(date) active_until
FROM `fh-bigquery.weather_gsod.all` WHERE name LIKE
'SAN FRANC%' AND date > '1980-01-01' GROUP BY 1,2
ORDER BY active_until DESC
```

1.5 secs elapsed, **62.8MB processed**

Partitioned table by date-month (actual)
Clustered table by name

```
SELECT name, state, ARRAY_AGG(STRUCT(date,temp) ORDER
BY temp DESC LIMIT 5) top_hot, MAX(date) active_until
FROM `fh-bigquery.weather_gsod.all` WHERE name LIKE
'SAN FRANC%' AND date > '1980-01-01' GROUP BY 1,2 ORDER
BY active_until DESC
```

2.3 secs elapsed, **3.1 GB processed**

Clustering without partitions is much more efficient on tables that don't have a lot of GB per day!

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;
```

Reasoning

Aggregate as late and as seldom as possible, because aggregation is very costly.

BUT *if a table can be reduced drastically by aggregation in preparation for being joined, then aggregate it early.*

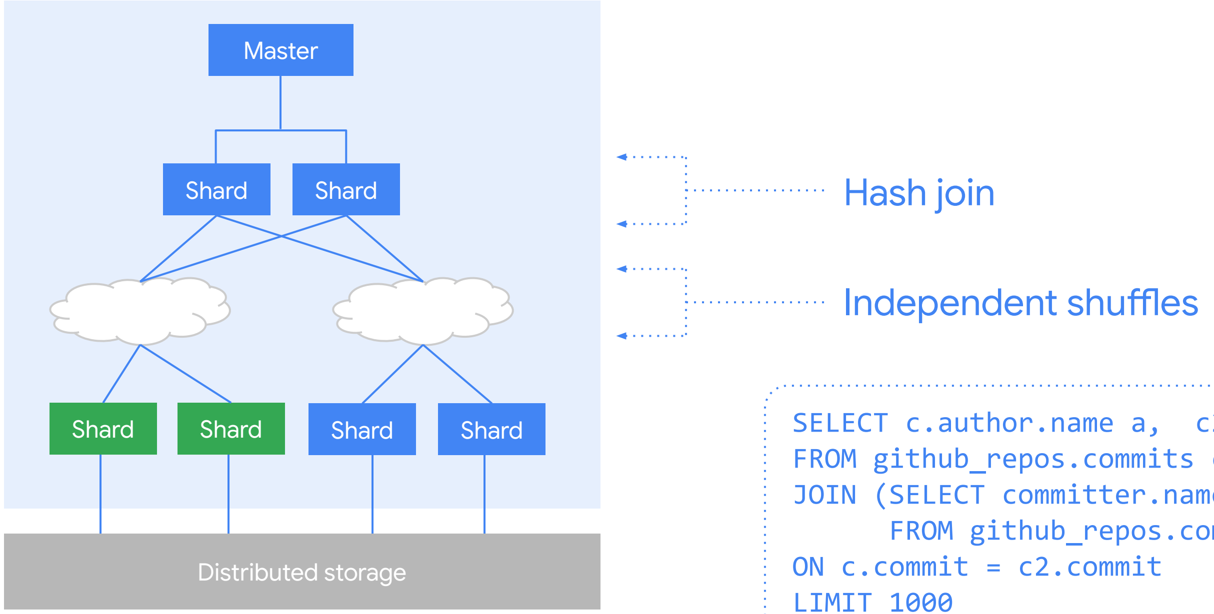
Caution: With JOINS, this only works if the two tables are already aggregated to the same level (i.e., if there is only one row for every join key value).

Nest repeated data

- **Customers often default to "flat" denormalization even if it is not the most beneficial**
  - Requires a GROUP BY to analyze data
- **Example: Orders table with a row for each line item**
  - `{order_id1, item_id1}, {order_id1, item_id2}, …`
- **If you model one order per row and nest line items in a nested field, GROUP BY no longer required**
  - `{order_id1, [ {item_id1}, {item_id2} ] }`

JOINs

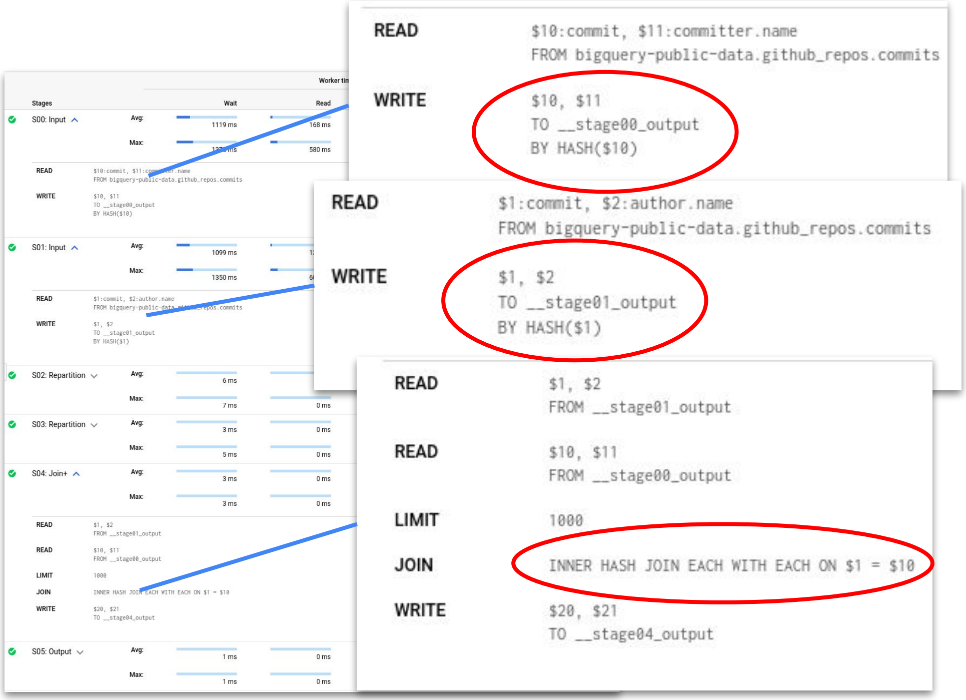
Large JOIN (shuffle)



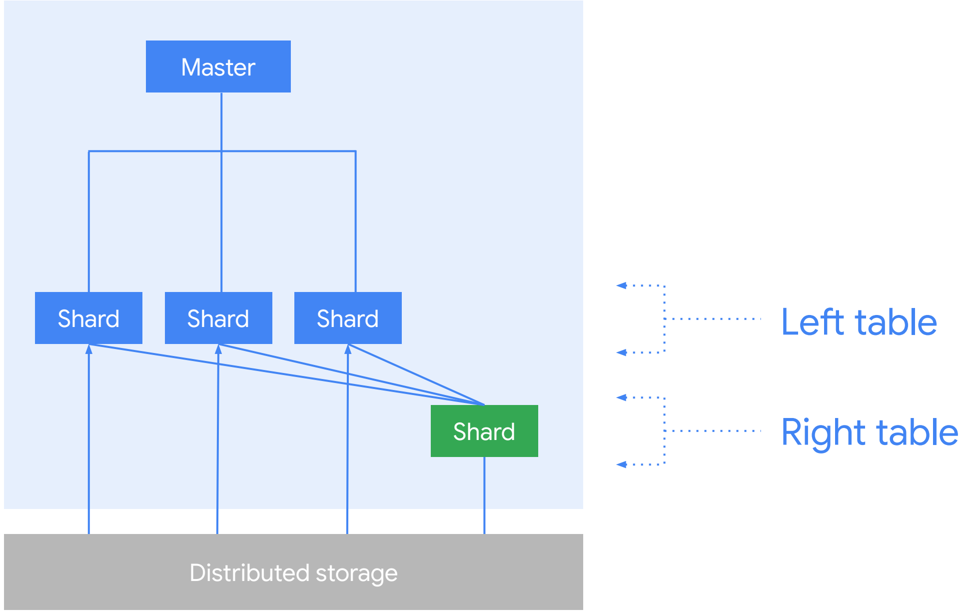

```
SELECT c.author.name a,  c2.a m
FROM github_repos.commits c
JOIN (SELECT committer.name a, commit
      FROM github_repos.commits) c2
ON c.commit = c2.commit
LIMIT 1000
```

Shuffle JOIN query plan

```
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
```



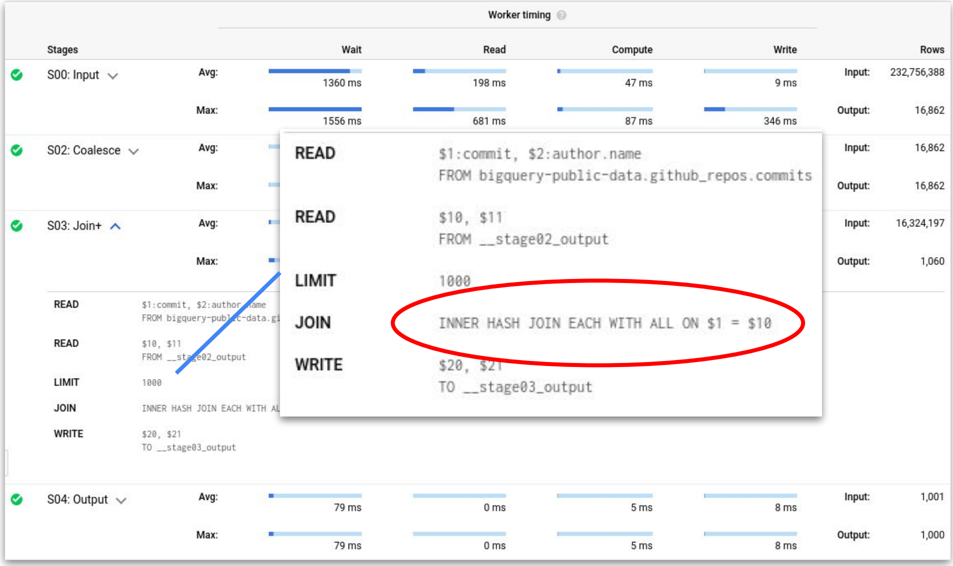
Small JOIN (broadcast)




```
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
```

Broadcast JOIN query plan

```
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
```



JOIN optimization with clustered tables

Subquery titles (“Animated_Google” ... ”A_Google_A_Day”) are used to filter table `pageviews_2017` **BEFORE** joining.

```
SELECT
  tbl2017.*
FROM
`wikipedia_v3.pageviews_2017` tbl2017
JOIN(SELECT * FROM
      `wikipedia_vt.just_latest_rows`
      WHERE
        REGEXP_CONTAINS(title, "Google"))
USING(title)
WHERE
  DATE(tbl2017.datehour)
  BETWEEN '2017-06-01' AND '2017-06-30'
```

Query complete (0.8 sec elapsed, 21.1 MB processed)

Job information | Results | JSON | Execution details

Row	datehour	wiki	title	views
1	2018-10-22 16:00:00 UTC	en	Animated_Google	1
2	2018-10-23 14:00:00 UTC	en	Alphabet_(Google)	1
3	2018-10-24 23:00:00 UTC	en	Actions_on_Google	1
4	2018-10-24 23:00:00 UTC	en	Authors_Guild,_Inc._v._Google,_Inc.	2
5	2018-10-22 15:00:00 UTC	en	Acquisitions_by_Google	1
6	2018-10-23 02:00:00 UTC	en	Adblock_Plus_for_Google_Chrome	1
7	2018-10-23 15:00:00 UTC	en	Accomplished_Googlebombs	1
8	2018-10-24 16:00:00 UTC	en	Are_You_Smart_Enough_to_Work_at_Google?	1
9	2018-10-24 21:00:00 UTC	en	Authors_Guild_v._Google	1
10	2018-10-24 21:00:00 UTC	en	A_Google_A_Day	1

Clustered table JOIN optimization requirements

```
SELECT
  tbl2017.*
FROM
`wikipedia_v3.pageviews_2017` tbl2017
JOIN(SELECT * FROM
     `wikipedia_vt.just_latest_rows`
     WHERE
       REGEXP_CONTAINS(title, "Google"))
USING(title)
WHERE
  DATE(tbl2017.datehour)
  BETWEEN '2017-06-01' AND '2017-06-30'
```

Left table `pageviews_2017` **must be clustered**

Subquery result size must qualify for **broadcast join** (e.g. JOIN EACH WITH **ALL**)

Clustered table JOIN optimization in query plan

Worker timing

Stages		Wait	Read	Compute	Write		Rows
S00: Input	Avg:	377 ms	28 ms	6 ms	5 ms	Input:	544,471
	Max:	466 ms	123 ms	205 ms	67 ms	Output:	10
S01: Output	Avg:	7 ms	0 ms	5 ms	3 ms	Input:	10
	Max:	7 ms	0 ms	5 ms	3 ms	Output:	1
S03: Output	Avg:	10 ms	353 ms	923 ms	4 ms	Input:	2,720,514,669
	Max:	29 ms					

S00:

```
READ    $10:title
        FROM fh-bigquery.wikipedia_vt.just_latest_rows
        WHERE regexp_contains($10, 'Google')
WRITE   $10
        TO __stage00_output
        BY HASH($10)
```

S03:

```
READ    $1:datehour, $2:wiki, $3:title, $4:views
        FROM fh-bigquery.wikipedia_v3.pageviews_2017
        WHERE and(between(date($1), 17318, 17347), between($3, 'A_
READ    $10
        FROM __stage00_output
JOIN    INNER HASH JOIN EACH WITH ALL ON $3 = $10
WRITE   $20, $21, $22, $23
        TO __stage03_output
```

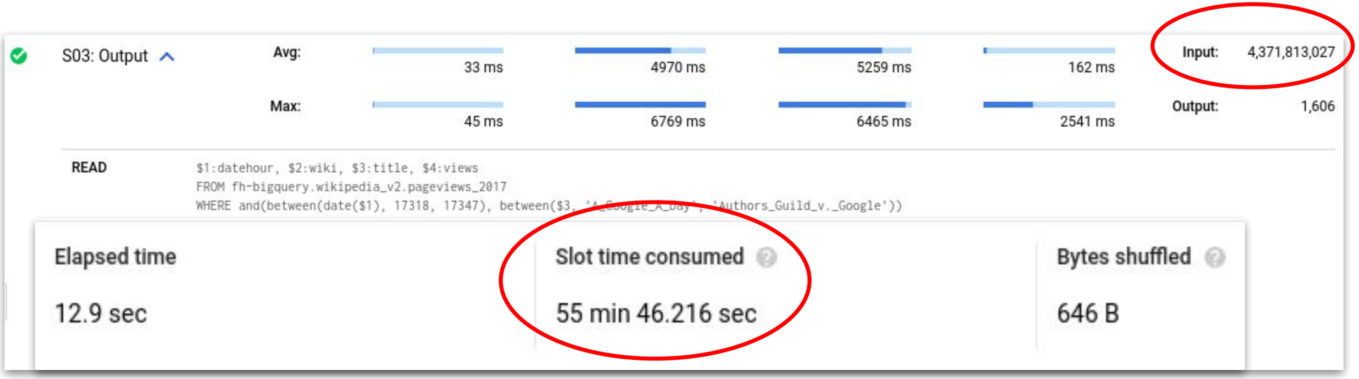


```
READ    $10:title
        FROM fh-bigquery.wikipedia_vt.just_latest_rows
        WHERE regexp_contains($10, 'Google')
```

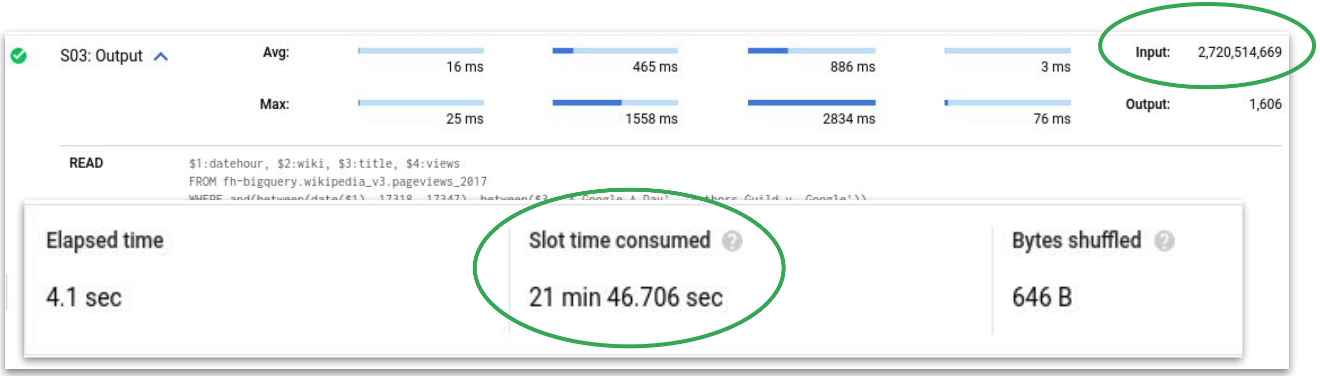

The right join table range of values is used to filter the left table before joining

```
READ    $1:datehour, $2:wiki, $3:title, $4:views
        FROM fh-bigquery.wikipedia_v3.pageviews_2017
        WHERE and(between(date($1), 17318, 17347), between($3, 'A_Google_A_Day', 'Authors_Guild_v._Google'))

READ    $10
        FROM __stage00_output

JOIN    INNER HASH JOIN EACH WITH ALL ON $3 = $10

WRITE   $20, $21, $22, $23
        TO __stage03_output
```

Clustered table JOIN optimization in query plan



Partitioned but **not clustered** table results in joining **more data**, consuming **more slots**



Partitioned **and clustered** table results in joining **less data**, consuming **less slots**

Optimization: JOIN pattern

Original code

```
select
  t1.dim1,
  sum(t1.metric1),
  sum(t2.metric2)
from
  small_table t1
join
  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
  large_table t2
join
  small_table t1
on
  t1.dim1 = t2.dim1
where t1.dim1 = ‘abc’
group by 1;
```

Reasoning

When you create a query by using a JOIN, consider the order in which you are merging the data. The standard SQL query optimizer can determine which table should be on which side of the join, but it is still recommended to order your joined tables appropriately.

The best practice is to manually place the **largest table first**, followed by the smallest, and then by decreasing size. Only under specific table conditions does BigQuery automatically reorder/optimize based on table size.

Optimization: Filter before JOINs

Original code

```
select
  t1.dim1,
  sum(t1.metric1),
  sum(t2.metric3)
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),
  sum(t2.metric3)
from
  `dataset.table1` t1
left join
  `dataset.table2` t2
on
  t1.dim1 = t2.dim1
where t1.dim2 = 'abc' AND t2.dim2 = 'abc'
group by 1;
```

Reasoning

WHERE clauses should be executed as soon as possible, especially within joins, so the tables to be joined are as small as possible.

WHERE clauses may not always be necessary, as standard SQL will do its best to push down filters. Review the explanation plan to see if filtering is happening as early as possible, and either fix the condition or use a subquery to filter in advance.

Join explosions

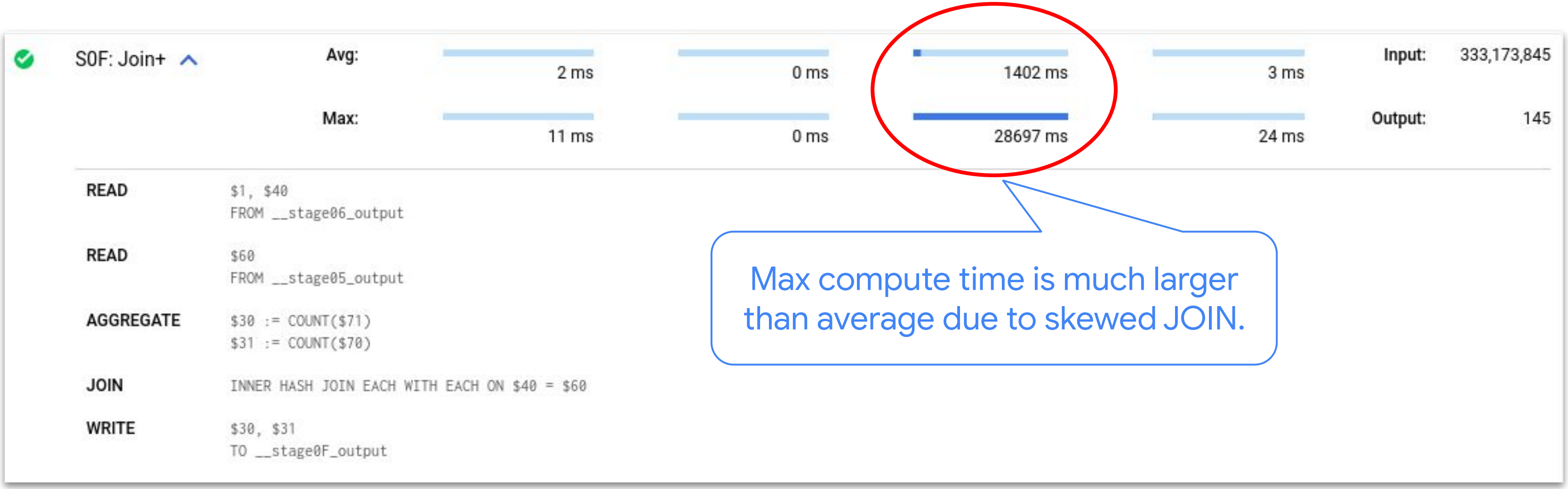
- Caused by JOIN with non-unique key on both sides
- SQL relational algebra gives cartesian product of rows which have the same join key
    - Worst case: Number of output rows is number of rows in left table multiplied by number of rows in right table
    - In extreme cases, query will not finish
- If job finishes then query explanation will show output rows versus input rows
- Confirm diagnosis by modifying query to print number of rows on each side of the JOIN grouped by the JOIN key
- Workaround is to use GROUP BY to pre-aggregate

Skewed JOINs

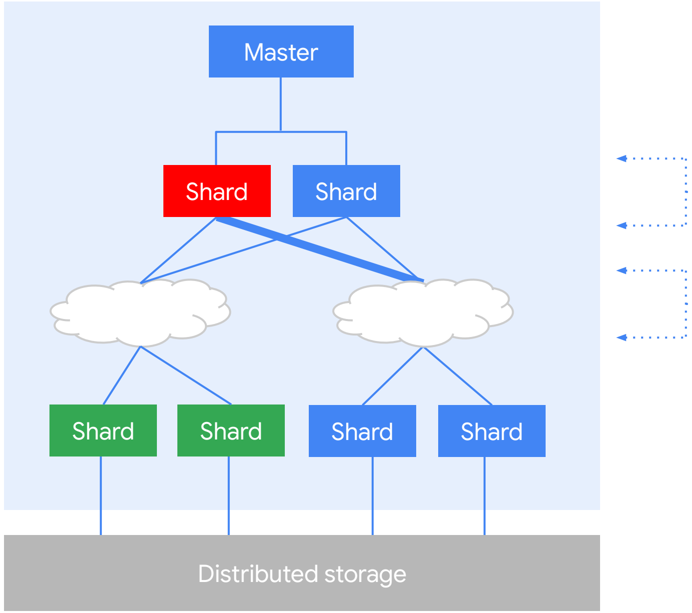
Skewed JOIN query

```
SELECT COUNT(l.title), COUNT(l.ot)
FROM (
  SELECT
    IF(title like "%c%", "c", title) as title,
    title as ot
  FROM `bigquery-public-data.samples.wikipedia`) l
JOIN (
  SELECT title
  FROM `bigquery-public-data.samples.wikipedia`
  GROUP BY title) r
USING(title)
```

Skewed JOIN query plan



Skewed JOIN



One shard gets too much data

Independent shuffles

Can’t just apply the redispatch trick because you’d need to reshuffle both sides

Skewed JOINs

- **Dremel shuffles data on each side of the join**
  - All data with the same join key goes to the same shard
  - Data can overload the shard
- **Typically result from data skew**
- **Workarounds**
  - Pre-filter rows from query with the unbalanced key
  - Potentially split into two queries

Filtering and ordering
WHERE / ORDER BY

WHERE clause: Expression order matters!

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 expression:
`user_display_name = 'anon'`
filters out much more data
than the expression:
`text LIKE '%java%'`

Reasoning

BigQuery assumes that the user has provided the best order of expressions in the WHERE clause, and does not attempt to reorder expressions. Expressions in your WHERE clauses should be ordered with the most selective expression first.

The optimized example is faster because it doesn't execute the expensive LIKE expression on the entire column content, but rather only on the content from user, 'anon'.

WHERE clause reordering: Proof in the query plan

Elapsed time	Slot time consumed	Bytes shuffled	Bytes spilled to disk
2.9 sec	2 min 48.212 sec	0 B	0 B

Worker timing

Stages		Wait	Read	Compute	Write	Rows	
S00: Output	Avg:	200 ms	1314 ms	812 ms	4 ms	Input:	75,437,848
	Max:	439 ms	1707 ms	1022 ms	28 ms	Output:	24

```
WHERE
  text LIKE
'%java%'
  AND
  user_display_name
=
  'anon'
```

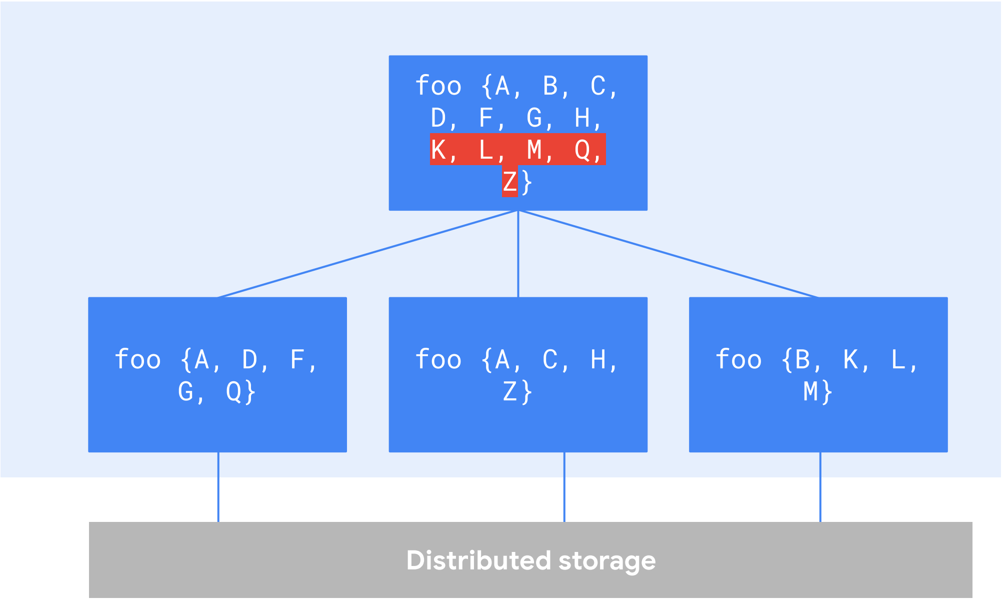
Elapsed time	Slot time consumed	Bytes shuffled	Bytes spilled to disk
1.0 sec	21.537 sec	0 B	0 B

Worker timing

Stages		Wait	Read	Compute	Write	Rows	
S00: Output	Avg:	212 ms	170 ms	21 ms	4 ms	Input:	75,437,848
	Max:	431 ms	359 ms	54 ms	27 ms	Output:	24

```
WHERE
  user_display_name
=
  'anon'
  AND
  text LIKE
'%java%'
```

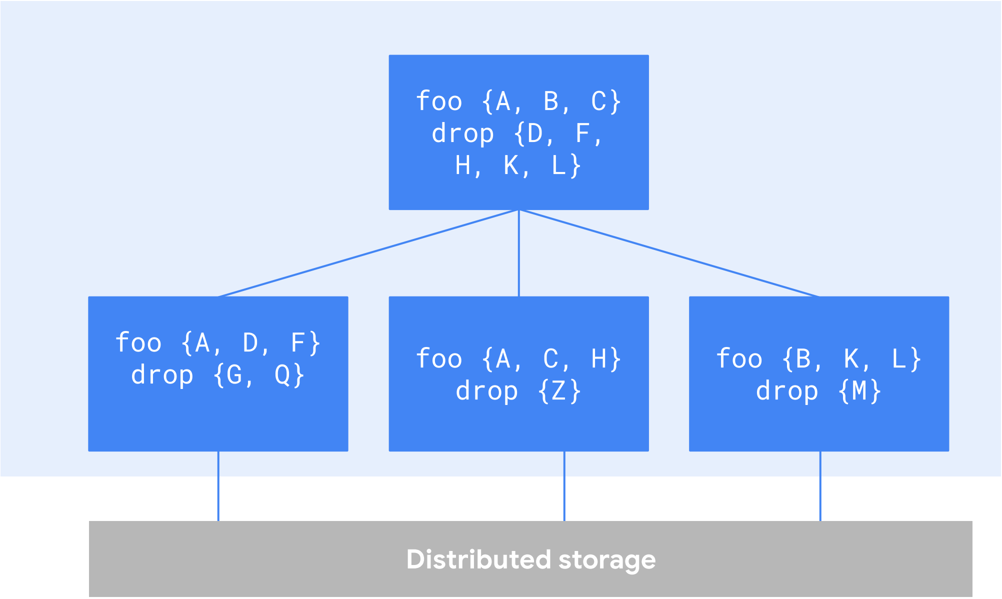
Large ORDER BYs



SELECT foo
FROM table
ORDER BY foo

Master node needs to sort and store all values

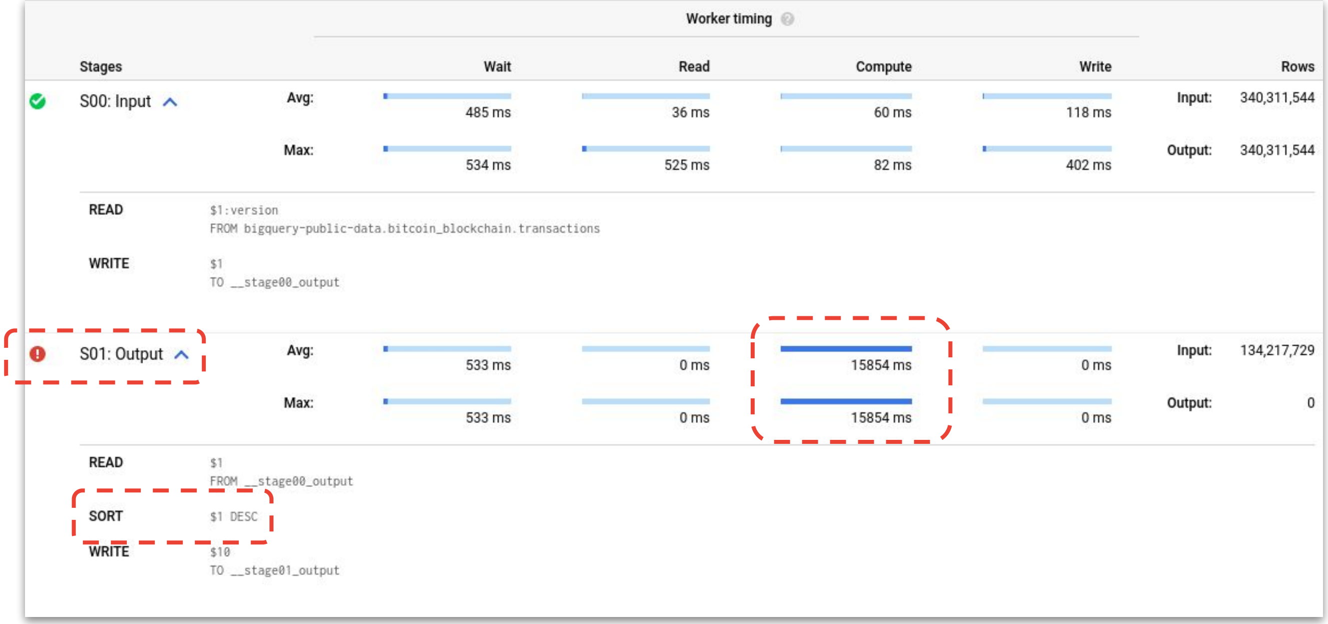
ORDER BY and LIMIT



SELECT foo
FROM table
ORDER BY foo
LIMIT 3

Can drop values over the limit at each node

Overloaded ORDER BY query plan




```
SELECT
  version
FROM
bitcoin_blockchai
n.transactions
ORDER BY
  version DESC
```

ORDER BY with LIMIT query plan

Worker timing

Stages		Wait	Read	Compute	Write	Rows	
S00: Input	Avg:	271 ms	44 ms	17 ms	8 ms	Input:	340,311,544
	Max:	312 ms	475 ms	28 ms	98 ms	Output:	1,667,000

READ	$1:version FROM bigquery-public-data.bitcoin_blockchain.transactions
SORT	$1 DESC LIMIT 1000
WRITE	$10 TO __stage00_output

Stages		Wait	Read	Compute	Write	Rows	
S01: Output	Avg:	312 ms	0 ms	177 ms	128 ms	Input:	1,667,000
	Max:	312 ms	0 ms	177 ms	128 ms	Output:	1,000

READ	$10 FROM __stage00_output
SORT	$10 DESC LIMIT 1000
WRITE	$20 TO __stage01_output

```
SELECT
  version
FROM
bitcoin_blockchain.transactions
ORDER BY
  version DESC
LIMIT 1000
```

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
```

Reasoning

Writing results for a query with an **ORDER BY** clause can result in Resources Exceeded errors. Because the final sorting must be done on a single slot, if you are attempting to order a very large result set, the final sorting can overwhelm the slot that is processing the data.

If you are sorting a very large number of values use a **LIMIT** clause.

Optimization: Latest record

Original code

```
select
  * except(rn)
from (
  select *,
    row_number() over(
      partition by id
      order by created_at desc) rn
  from
    `dataset.table` t
)
where rn = 1
order by created_at
```

Optimized

```
select
  event.*
from (
  select array_agg(
    t order by t.created_at desc limit 1
  )[offset(0)] event
  from
    `dataset.table` t
  group by
    id
)
order by created_at
```

Reasoning

Using the ROW_NUMBER() function can fail with Resources Exceeded errors as data volume grows if there are too many elements to ORDER BY in a single partition.

Using ARRAY_AGG() in standard SQL allows the query to run more efficiently because the ORDER BY is allowed to drop everything except the top record on each GROUP BY.

Working with arrays

Example: Flattening arrays

Row	band	members
1	The Beatles	John;Paul;George;Ringo
2	The Three Stooges	Moe;Larry;Curly

```
WITH sample_data AS (
SELECT
 'The Beatles' AS band, 'John;Paul;George;Ringo' AS members
 UNION ALL
 SELECT 'The Three Stooges' AS band, 'Moe;Larry;Curly' AS members
)
SELECT band, member
FROM sample_data
CROSS JOIN UNNEST( SPLIT(sample_data.members, ';') ) member
```

Correlated CROSS JOIN: nested array in each row is flattened and combined only with columns from same row

Row	band	member
1	The Beatles	John
2	The Beatles	Paul
3	The Beatles	George
4	The Beatles	Ringo
5	The Three Stooges	Moe
6	The Three Stooges	Larry
7	The Three Stooges	Curly

Best practices for functions

Optimization: String comparison

Original Code

```
select
  dim1
from
  `dataset.table`
where
  regexp_contains(dim1, ‘.*test.*’)
```

Optimized

```
select
  dim1
from
  `dataset.table`
where
  dim1 like ‘%test%’
```

Reasoning

REGEXP_CONTAINS > LIKE
where > means more functionality,
but also slower execution time.
Prefer LIKE when the full power of
regex is not needed (e.g. wildcard
matching).

Optimization: Approximate functions

Original code

```
select
  dim1,
  count(distinct dim2)
from
  `dataset.table`
group by 1;
```

Optimized

```
select
  dim1,
  approx_count_distinct(dim2)
from
  `dataset.table`
group by 1;
```

Reasoning

If the SQL aggregation function you're using has an equivalent approximation function, the approximation function will yield faster query performance.

Approximate functions produce a result which is generally **within 1%** of the exact number.

Optimization: SQL UDFs > JavaScript UDFs

Original code

```
create temporary function
  multiply(x INT64, y INT64)
returns INT64
language js
as “””
  return x * y;
“””;

select multiply(2, 2) as result;
```

Optimized

```
create temporary function
  multiply(x INT64, y INT64)
as
  (x * y);

select multiply(2, 2) as result;
```

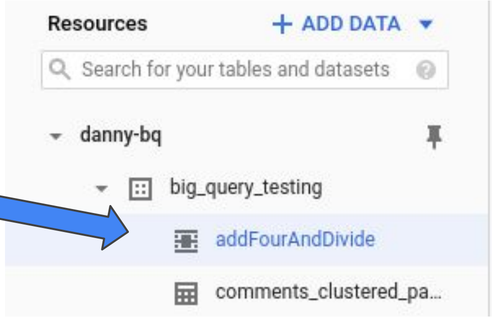
Reasoning

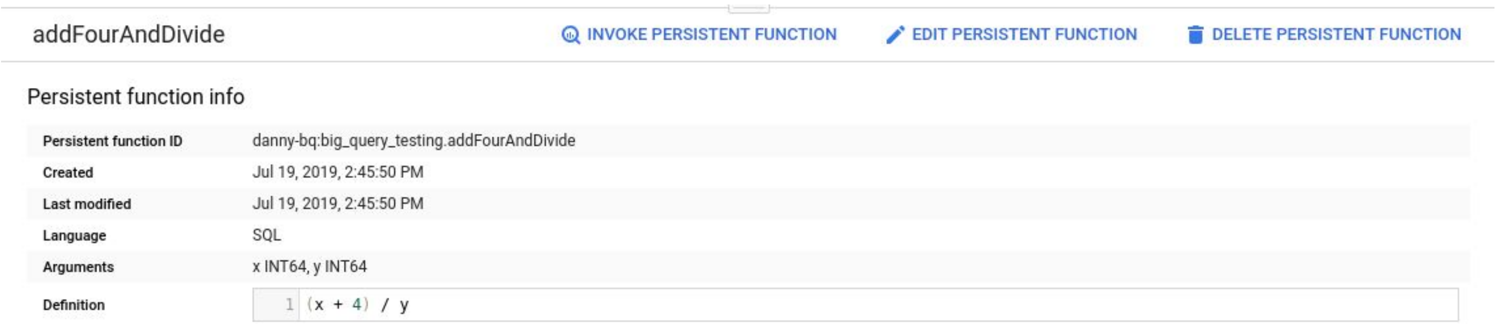
JavaScript UDFs are a performance killer because they have to spin up a V8 subprocess evaluate.

Prefer SQL UDFs where possible.

Optimization: Persistent UDFs

```
CREATE OR REPLACE FUNCTION
your_dataset.addFourAndDivide(x INT64, y INT64) AS (
        (x + 4) / y
);
```





Optimization: Persistent UDFs

Original code

```
CREATE TEMP FUNCTION addFourAndDivide(x
INT64, y INT64) AS ((x + 4) / y);

WITH numbers AS
  (SELECT 1 as val
  UNION ALL
  SELECT 3 as val
  UNION ALL
  SELECT 4 as val
  UNION ALL
  SELECT 5 as val)
SELECT val, addFourAndDivide(val, 2) AS
result
FROM numbers;
```

Optimized

```
# Replaced with persistent function
# and invoked below

WITH numbers AS
  (SELECT 1 as val
  UNION ALL
  SELECT 3 as val
  UNION ALL
  SELECT 4 as val
  UNION ALL
  SELECT 5 as val)
SELECT val,
`your_project.your_dataset.addFourAndDivide`
(val, 2) AS result
FROM numbers;
```

Reasoning

Create persistent user-defined SQL and JavaScript functions in a centralized BigQuery dataset which can be invoked across queries and in logical views.

Create org-wide libraries of business logic within shared datasets.

Scripting and stored procedures

- Execute multiple statements in one request
- Declare, assign, and use variables
- Control execution with conditions and loops
- Caveats
  - Statements are committed independently of each other
  - Cloud SDK version >= 267.0.0

```
DECLARE primes ARRAY<INT64> DEFAULT [2];
DECLARE n INT64 DEFAULT 3;
DECLARE max INT64 DEFAULT 30;
WHILE n <= max DO
  BEGIN
    DECLARE n_is_prime BOOL DEFAULT TRUE;

    -- Test all prime numbers from 2 to SQRT(n), inclusive.
    DECLARE i INT64 DEFAULT 0;
    WHILE i < ARRAY_LENGTH(primes) DO
      BEGIN
        DECLARE prime INT64 DEFAULT primes[OFFSET(i)];
        IF MOD(n, prime) = 0 THEN
          -- Found a prime < n, which divides evenly into n, so n is not prime.
          SET n_is_prime = FALSE;
          BREAK;
        END IF;
        IF prime * prime >= n THEN
          -- <primes> is kept sorted in increasing order, so once we find a
          -- single value >= SQRT(n), we can stop.
          BREAK;
        END IF;
      END;
      SET i = i + 1;
    END WHILE;

    -- If n is prime, then add it to the list of known primes.
    IF n_is_prime THEN
      SET primes = ARRAY_CONCAT(primes, [n]);
    END IF;
  END;
  SET n = n + 1;
END WHILE;

-- Display all the primes.
SELECT prime FROM UNNEST(primes) AS prime ORDER BY prime;
```

Optimization: Necessary columns only

Original code

```
select
  *
from
`dataset.table`
```

Optimized

```
select
  * EXCEPT (dim1, dim2)
from
`dataset.table`
```

Reasoning

Only select the columns necessary, especially in inner queries. **SELECT** * is cost inefficient and may also hurt performance.

If the number of columns to return is large, consider using **SELECT** * **EXCEPT** to exclude unneeded columns.

In some use cases, **SELECT** * **EXCEPT** may be necessary.