Bigquery Best Practise

X Normal Forms

- First Normal Form (1NF)
  - **Eliminate repeating columns** in individual tables
  - Have **consistent types**
  - Create a separate table for each set of related data
  - **No duplicate rows**

Name	Item	Shipping Address	Supplier	Supplier Phone	Price
James Smith	Chromebook	36 Park Av, Miami	Google	(000) 123 456	300
Michael Brown	Macbook	21 Church St, New York	Apple	(999) 987 654	1000
Robert Jones	Chromebook, iPad	47 South Av, LA	2	2	800
James Smith	Macbook	21 Church St, New York	Apple	(999) 987 654	1000

Key

Cust ID	Name	Item	Shipping Address	Supplier	Supplier Phone	Price
JS1	James Smith	Chromebook	36 Park Av, Miami	Google	(000) 123 456	300
MB1	Michael Brown	Macbook	21 Church St, New York	Apple	(999) 987 654	1000
RJ1	Robert Jones	Chromebook	47 South Av, LA	Google	(000) 123 456	300
RJ1	Robert Jones	iPad	47 South Av, LA	Apple	(999) 987 654	500
JS2	James Smith	Macbook	21 Church St, New York	Apple	(999) 987 654	1000

X Normal Forms

- Second Normal Form (2NF)
  - Be in 1NF
  - **All attributes dependent on key**

Key

Cust ID	Name	Item	Shipping Address	Supplier	Supplier Phone	Price
JS1	James Smith	Chromebook	36 Park Av, Miami	Google	(000) 123 456	300
MB1	Michael Brown	Macbook	21 Church St, New York	Apple	(999) 987 654	1000
RJ1	Robert Jones	Chromebook	47 South Av, LA	Google	(000) 123 456	300
RJ1	Robert Jones	iPad	47 South Av, LA	Apple	(999) 987 654	500
JS2	James Smith	Macbook	21 Church St, New York	Apple	(999) 987 654	1000

Second Normal Form

Key

Cust ID	Name	Item	Shipping Address	Supplier	Supplier Phone	Price
JS1	James Smith	Chromebook	36 Park Av, Miami	Google	(000) 123 456	300
MB1	Michael Brown	Macbook	21 Church St, New York	Apple	(999) 987 654	1000
RJ1	Robert Jones	Chromebook	47 South Av, LA	Google	(000) 123 456	300
RJ1	Robert Jones	iPad	47 South Av, LA	Apple	(999) 987 654	500
JS2	James Smith	Macbook	21 Church St, New York	Apple	(999) 987 654	1000

Primary Key

Cust ID	Name	Shipping Address
JS1	James Smith	36 Park Av, Miami
MB1	Michael Brown	21 Church St, New York
RJ1	Robert Jones	47 South Av, LA
JS2	James Smith	21 Church St, New York

Primary Key

Item	Supplier	Supplier Phone	Price
Chromebook	Google	(000) 123 456	300
Macbook	Apple	(999) 987 654	1000
iPad	Apple	(999) 987 654	500

Primary Key Primary Key

Cust ID	Item
JS1	Chromebook
MB1	Macbook
RJ1	Chromebook
RJ1	iPad
JS2	Macbook

X Normal Forms

- Third Normal Form (3NF)
  - Be in 2NF
  - **No transitive functional dependencies**
    - All fields can be determined only by the key in the table and no other column

Cust ID (Primary Key)	Name	Shipping Address
JS1	James Smith	36 Park Av, Miami
MB1	Michael Brown	21 Church St, New York
RJ1	Robert Jones	47 South Av, LA
JS2	James Smith	21 Church St, New York

Item (Primary Key)	Supplier	Supplier Phone	Price
Chromebook	Google	(000) 123 456	300
Macbook	Apple	(999) 987 654	1000
iPad	Apple	(999) 987 654	500

Cust ID (Primary Key)	Item (Primary Key)
JS1	Chromebook
MB1	Macbook
RJ1	Chromebook
RJ1	iPad
JS2	Macbook

Third Normal Form

Primary Key

Cust ID	Name	Shipping Address
JS1	James Smith	36 Park Av, Miami
MB1	Michael Brown	21 Church St, New York
RJ1	Robert Jones	47 South Av, LA
JS2	James Smith	21 Church St, New York

Primary Key

Item	Supplier	Supplier Phone	Price
Chromebook	Google	(000) 123 456	300
Macbook	Apple	(999) 987 654	1000
iPad	Apple	(999) 987 654	500

Primary Key Primary Key

Cust ID	Item
JS1	Chromebook
MB1	Macbook
RJ1	Chromebook
RJ1	iPad
JS2	Macbook

Primary Key

Cust ID	Name	Shipping Address
JS1	James Smith	36 Park Av, Miami
MB1	Michael Brown	21 Church St, New York
RJ1	Robert Jones	47 South Av, LA
JS2	James Smith	21 Church St, New York

Primary Key Foreign Key

Item	Supplier	Price
Chromebook	Google	300
Macbook	Apple	1000
iPad	Apple	500

Primary Key Primary Key

Cust ID	Item
JS1	Chromebook
MB1	Macbook
RJ1	Chromebook
RJ1	iPad
JS2	Macbook

Primary Key

Supplier	Supplier Phone
Google	(000) 123 456
Apple	(999) 987 654

X Normal Form

Advantages

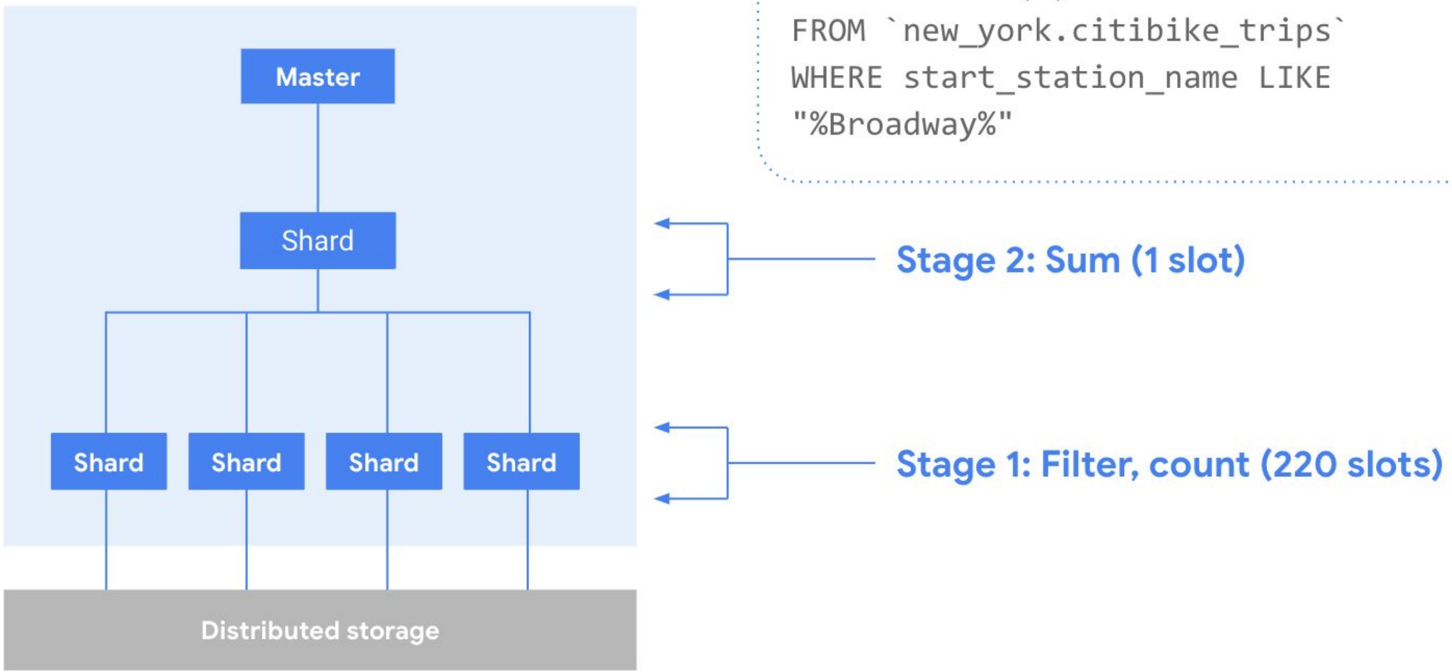
- Reduced data redundancy
- Increased data quality
- Capturing complete business requirements
- Reduce modification anomalies

A normalized data model can be easier to modify and maintain.

Disadvantages

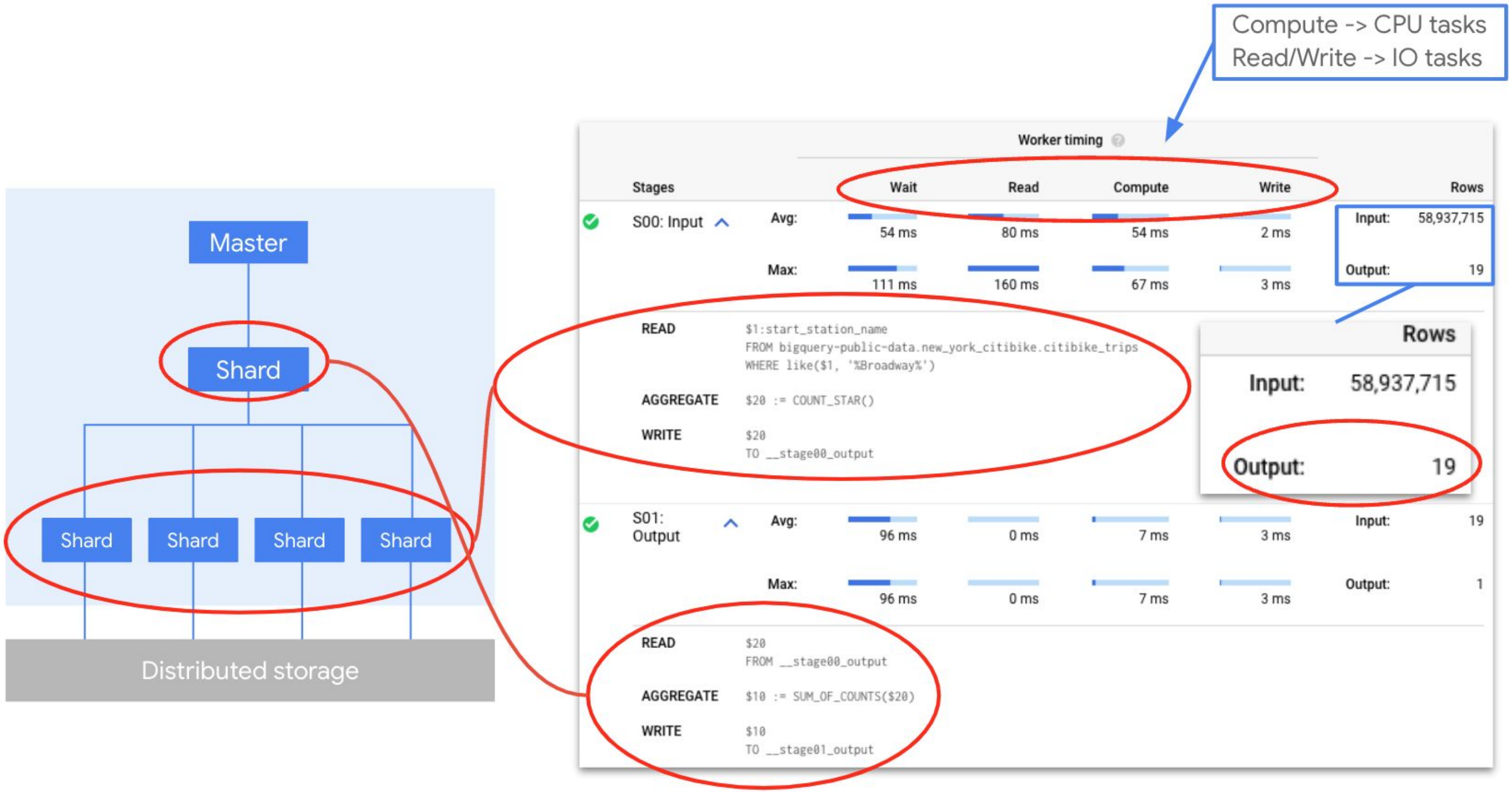
- Many number tables
  - Many joins
  - More time to retrieve data
  - Slow down the database performance

Simple query execution

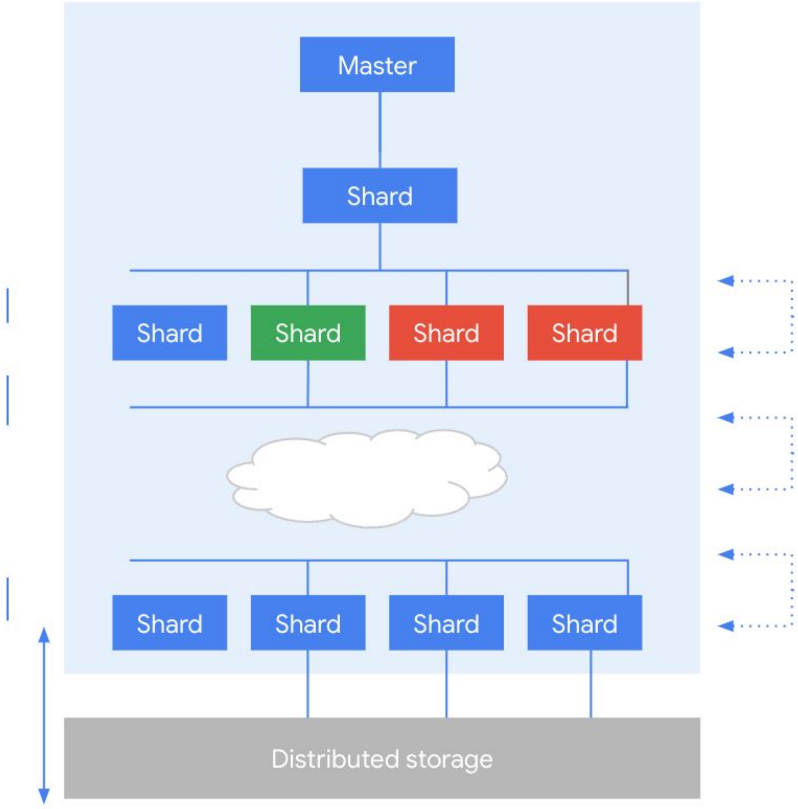



```
SELECT COUNT(*)
FROM `new_york.citibike_trips`
WHERE start_station_name LIKE
"%Broadway%"
```

Simple query execution - Query plan



Repartitioning



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

Stage 2 (X shards)
Stage 2.1 (Y shards)
Stage 2.2 (Z shards)

Shuffle

Stage 1: Read

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).

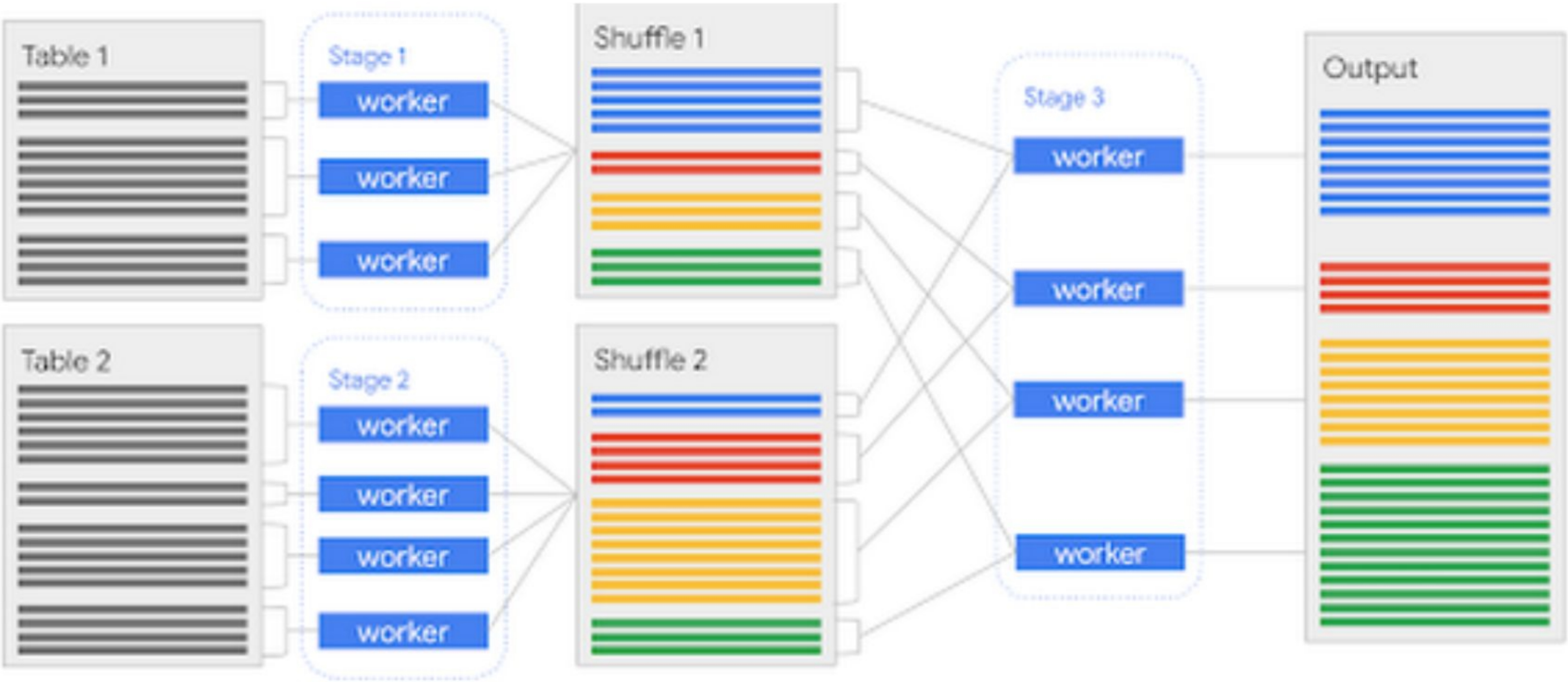
Nest repeated data

- **Customers often default to “flat” denormalization even if it is not the most beneficial**
  - Requires a GROUP BY to analyze data

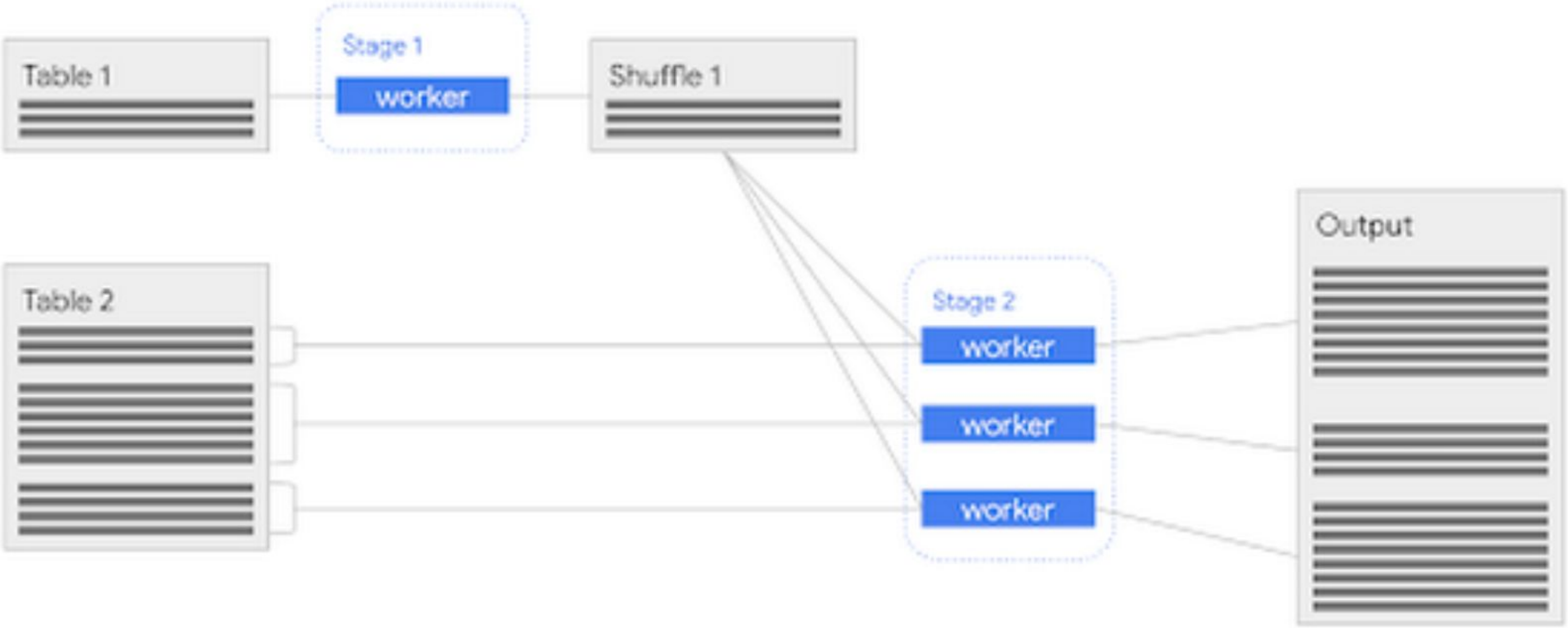
- **Example: Orders table with a row for each line item**
  - `{order_id1, item_id1}, {order_id1, item_id2}, …`

- **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} ] }`

Large JOIN (shuffle)



Small JOIN (broadcast)



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)
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 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.

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%'
```

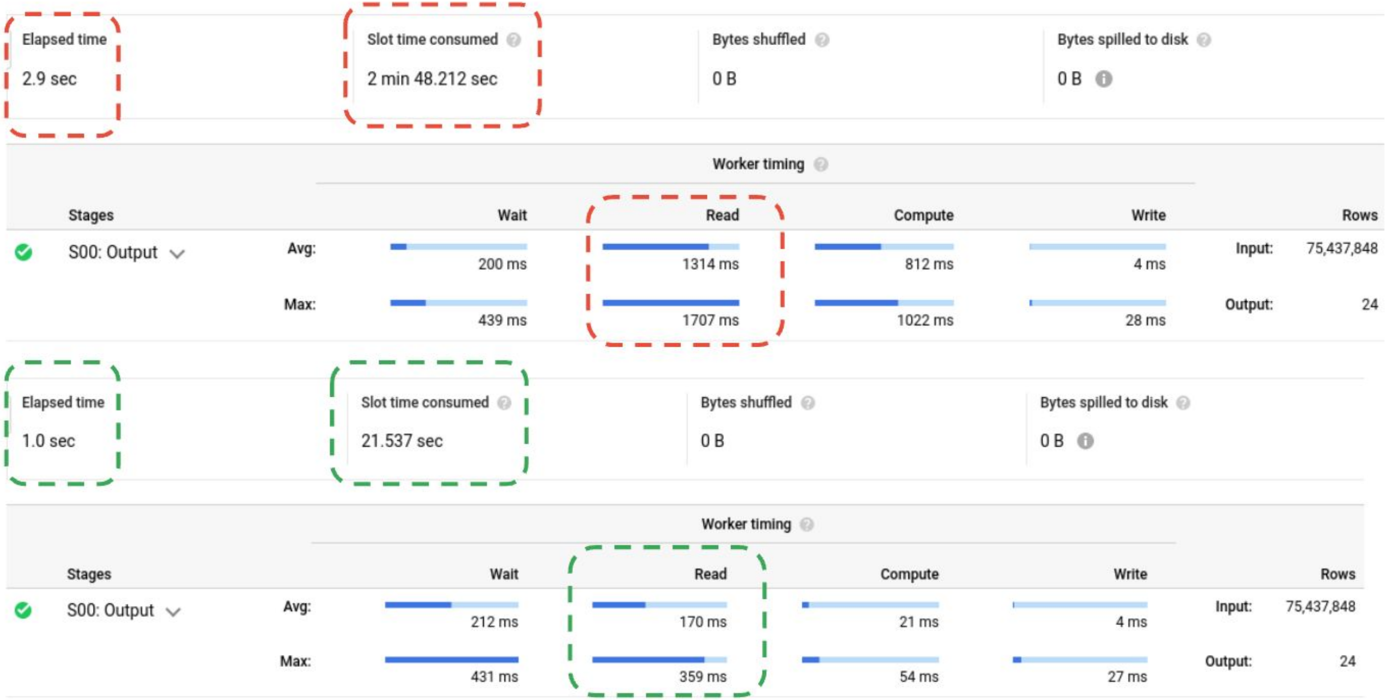
The expression:
`user_display_name = 'anon'`
filters out much more data
than the expression:
`text LIKE '%java%'`

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'.

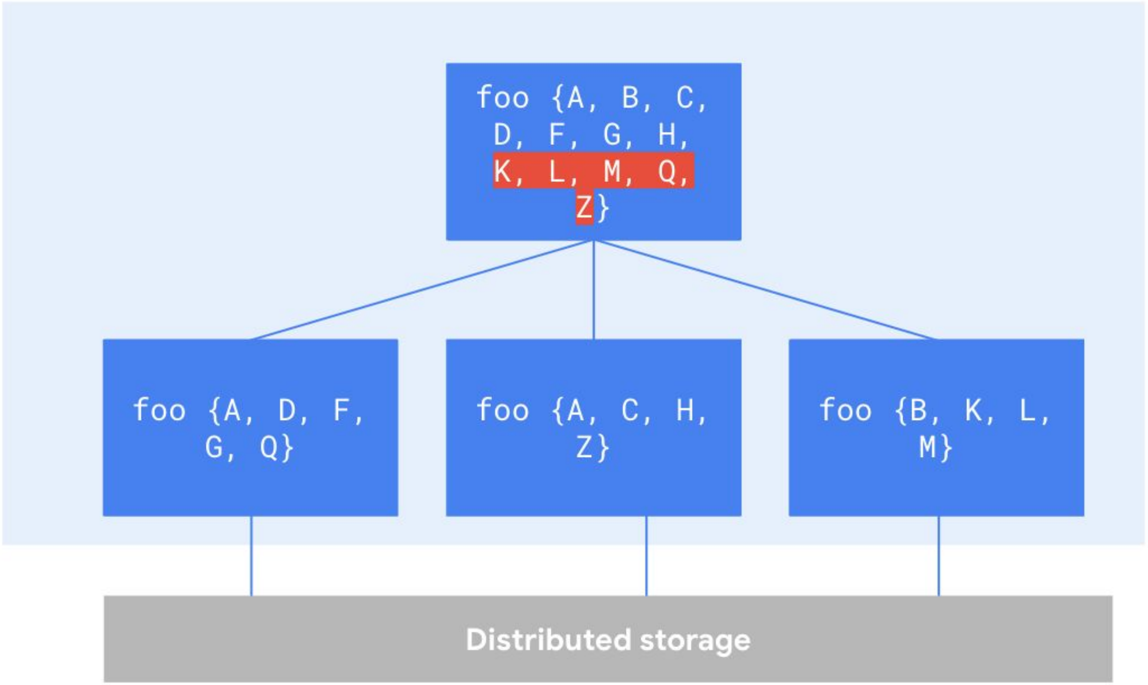
WHERE clause reordering: Proof in the query plan




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

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

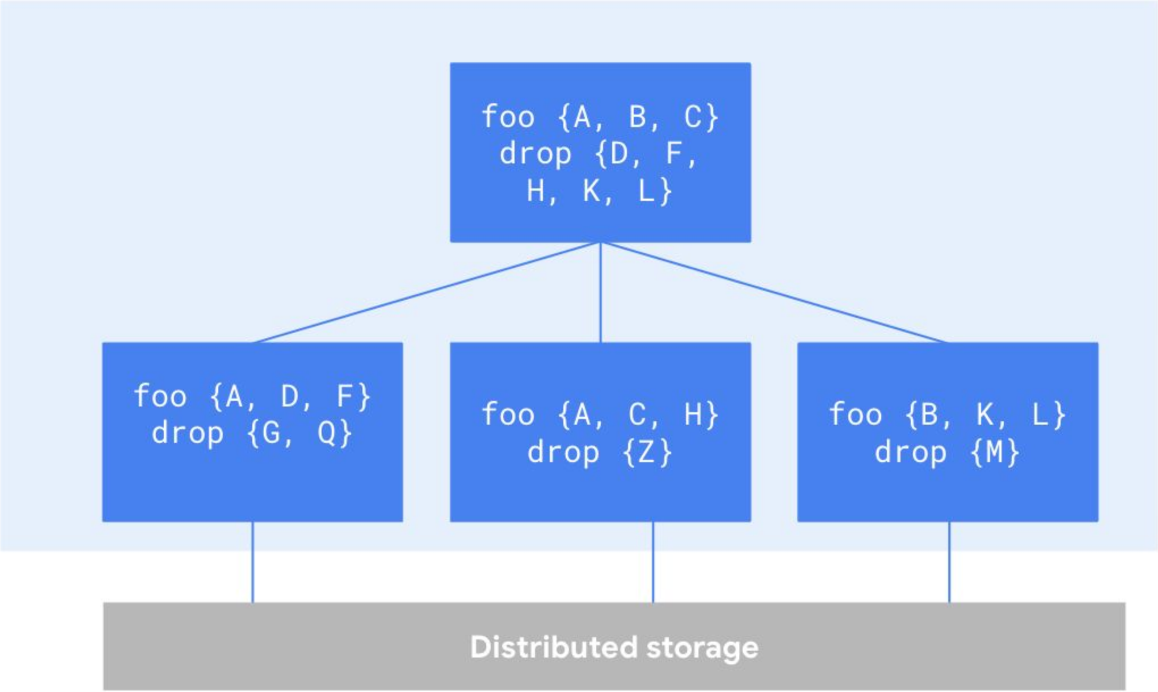
Large ORDER BYs



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

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.

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.