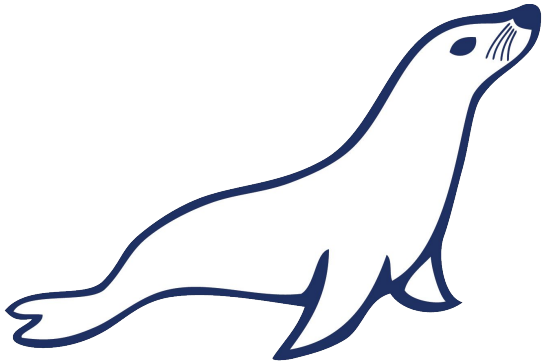


MariaDB®
FOUNDATION

MariaDB 10.3 Optimizer and Beyond

Vicențiu Ciorbaru
Software Engineer @ MariaDB Foundation
vicentiu@mariadb.org

Agenda

- Histograms
  - Current state of Histograms in MariaDB
  - Plans for 10.4
- Optimizations for derived tables
  - Condition pushdown into non-mergeable derived tables and views
  - Condition pushdown into in subqueries (10.4)
- Window Function optimisations
  - Condition pushdown through window functions' PARTITION BY

Condition Selectivity

- Query optimizer needs to decide on a plan to execute the query
- Goal is to get the shortest running time
  - Chose access method
    - Index Access, Hash Join, BKA, etc.
  - Choose correct join order to minimize the cost of reading rows
- Usually, minimizing rows read minimizes execution time
  - Sometimes reading more rows is advantageous, if table / index is all in memory
- **Use a cost model to estimate how long an execution plan would take**

Condition Selectivity

- Query optimizer needs to decide on a plan to execute the query
- Goal is to get the shortest running time
  - Chose access method
    - Index Access, Hash Join, BKA, etc.
  - Choose correct join order to minimize the cost of reading rows
- Usually, minimizing rows read minimizes execution time
  - Sometimes reading more rows is advantageous, if table / index is all in memory
- **Use a cost model to estimate how long an execution plan would take**
- For each condition in the where clause (and having) we compute
  - **Condition selectivity**
    How many rows of the table is this condition going to accept?
    10%, 20%, 90% ?

Condition Selectivity

- Query optimizer needs to decide on a plan to execute the query
- Goal is to get the shortest running time
  - Chose access method
    - Index Access, Hash Join, BKA, etc.
  - Choose correct join order to minimize the cost of reading rows
- Usually, minimizing rows read minimizes execution time
  - Sometimes reading more rows is advantageous, if table / index is all in memory
- **Use a cost model to estimate how long an execution plan would take**
- For each WHERE clause (and having) we compute
  - **Condition selectivity**
    How many rows of the table is this condition going to accept?
    10%, 20%, 90% ?

Getting the estimates right is important!

Condition Selectivity

- **Suppose we have query with 10 tables: T1, T2, T3, … T10**

- Query optimizer will:
  - **Estimate** the number of rows that it will read from each table
  - Based on the conditions in the where (and having) clauses

- Assume estimates have an average error coefficient e
  - Total number of estimated rows read is:

    (e * #T1) * (e * #T2) * (e * #T3) * … * (e * #T10)

    Where #T1..#T10 is the actual number of rows read for each table

- The estimation error is **amplified**, the more tables there are in a join
  - If we under/over estimate by a factor of 2 final error factor is 1024!
  - If error is only 1.5 (off by 50%), final error factor is ~60

Condition Selectivity

- **How does optimizer produce estimates?**

- Condition analysis:
  - Is it possible to satisfy conditions?

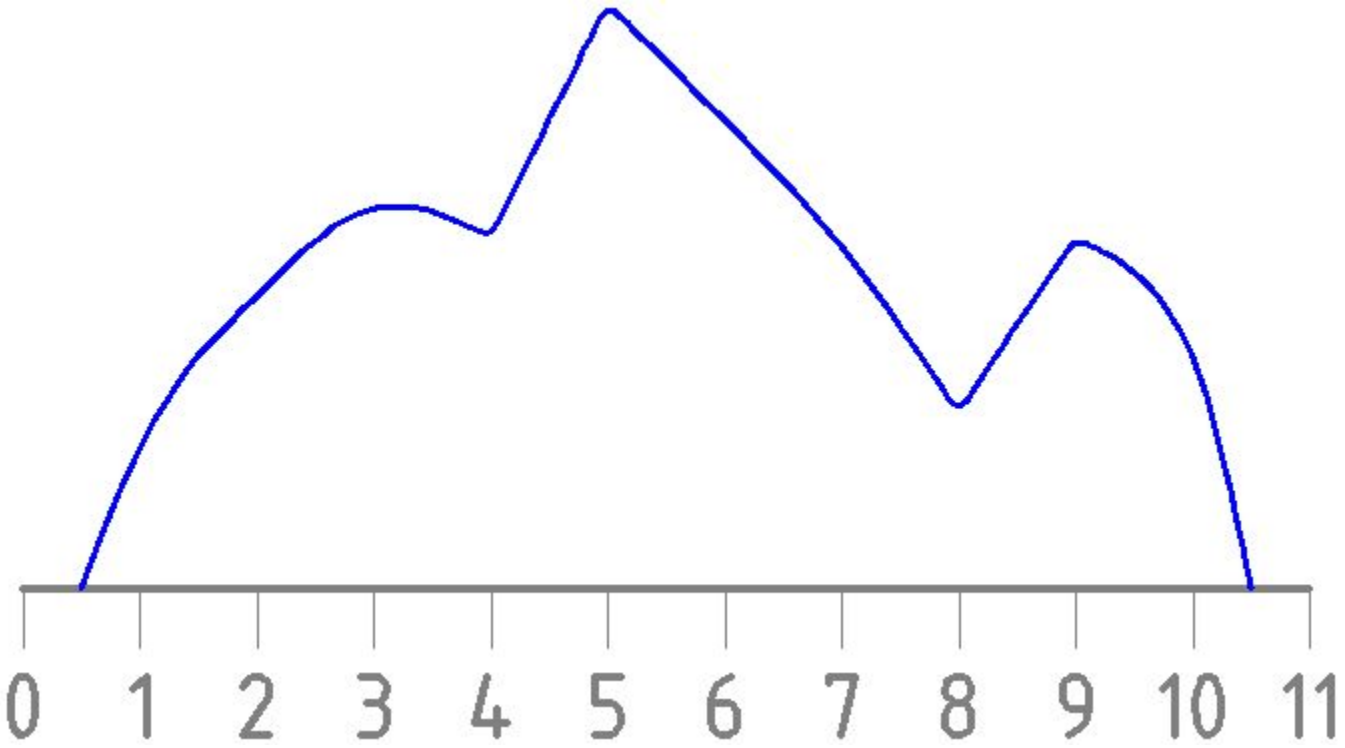
    t1.a > 10 and t1.a < 5
  - Equality condition on a distinct column?

- Index dives to get number of rows in a range
- Guesstimates (MySQL)

- Histograms for non-indexed columns

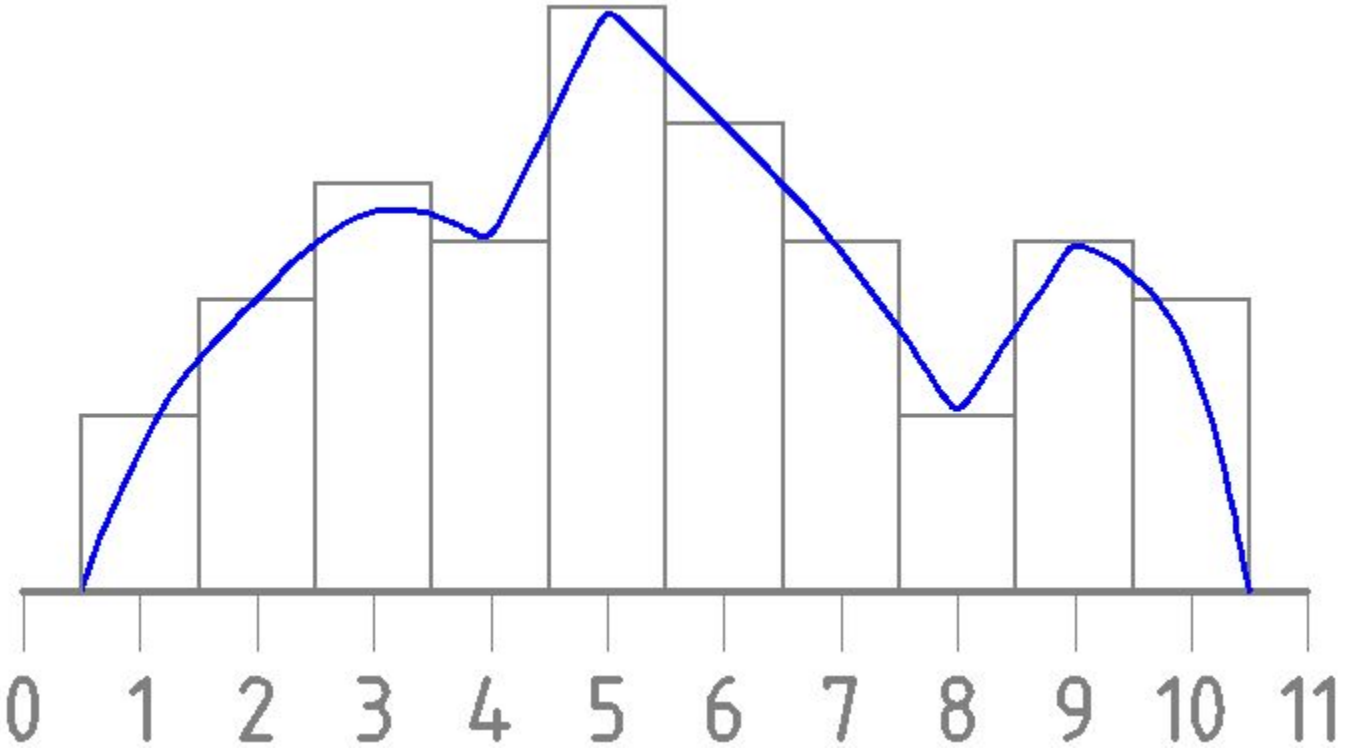
Histograms

- Estimate a distribution



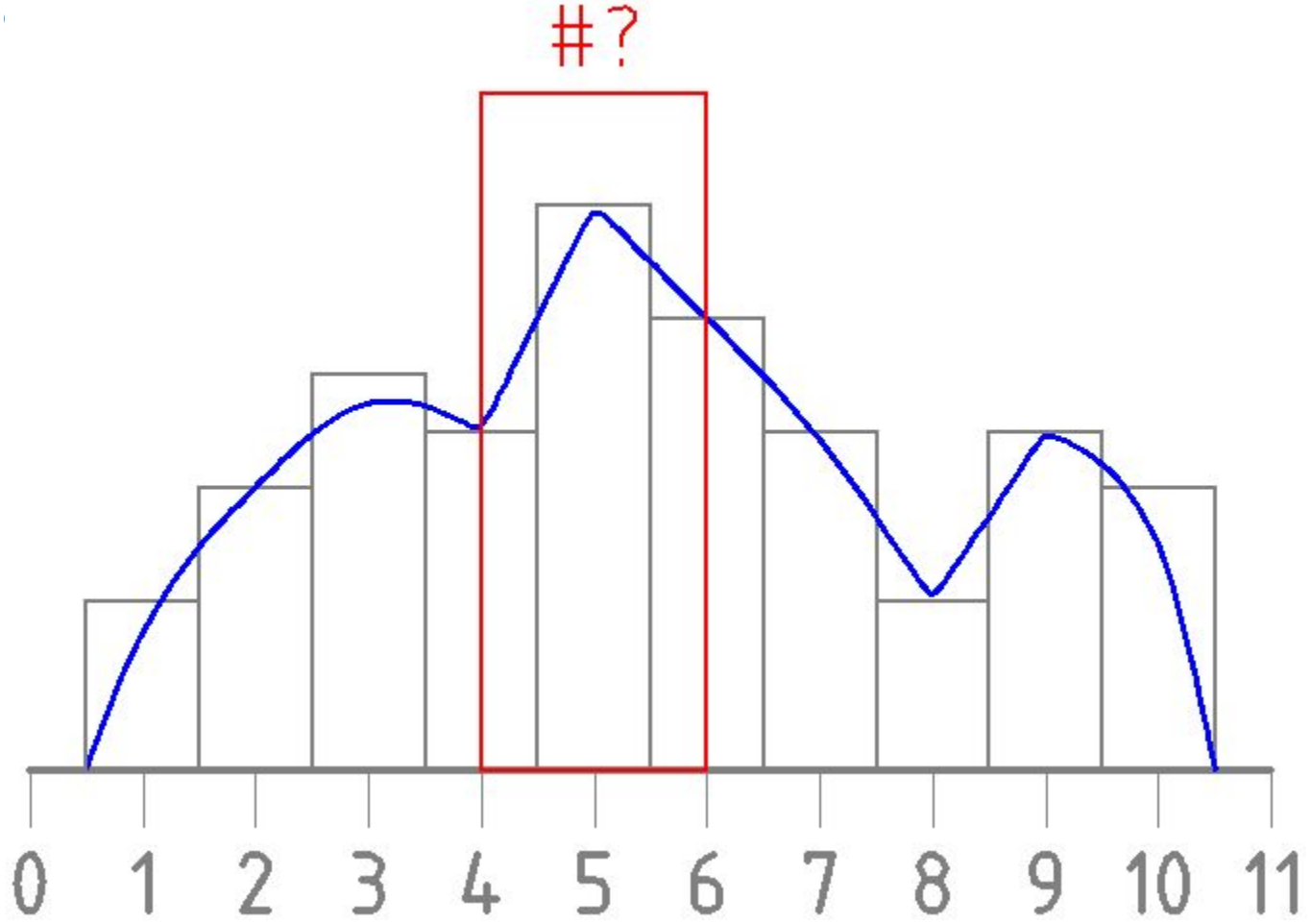
Histograms

- Estimate a distribution
  - Equal-width histogram
    - *Not uniform information*
    - *Many values in one bucket (5), fewer in others.*



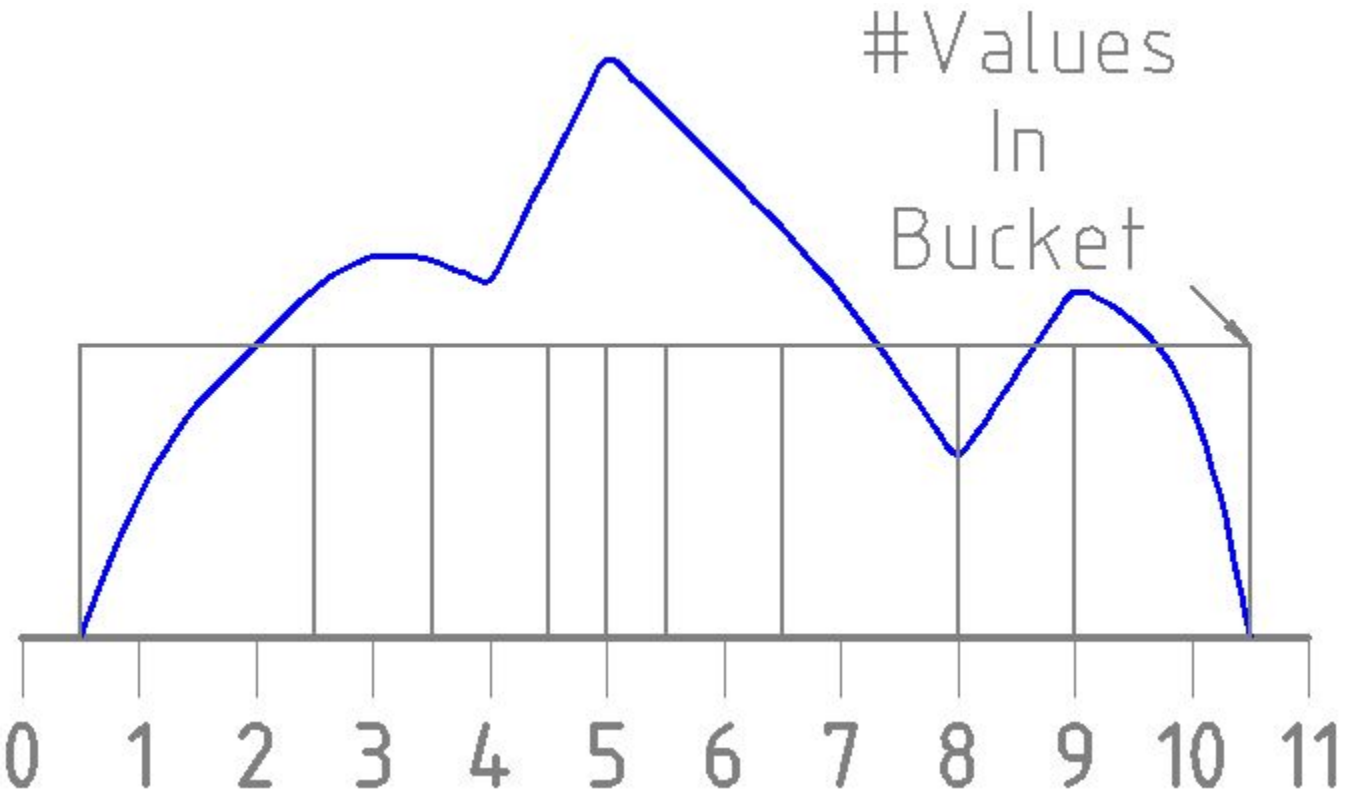
Histograms

- Estimate a distribution



Histograms

- Estimate a distribution
  - Equal-height histogram
    - *All bins have same # of values*
    - *More bins where there are more values.*



Histograms in MariaDB

- MariaDB histograms are collected by doing a full table scan
  - Support for sampling is under development for 10.4
    - 2018 GSOC project by Teodor Niculescu
- Needs to be done manually using ANALYZE TABLE … PERSISTENT
- Stored inside
  - mysql.table_stats, mysql.column_stats, mysql.index_stats
  - As a binary value (max 255 bytes), single / double precision
- Special function to decode: decode_histogram()
- Can be manually updated
  - One can run data collection on a slave, then propagate results
- Not enabled by default, needs a few switches turned on to work
  - Will be enabled by default in 10.4

Histograms in MySQL (8.0)

- MySQL histograms are collected by doing a full table scan
  - Needs to be done manually using ANALYZE TABLE … UPDATE HISTOGRAM
- Can collect all data or perform sampling by skipping rows, based on max memory allocation
- Comparable performance to full table scan.

- Stored inside data dictionary
  - Can be viewed through INFORMATION_SCHEMA.column_statistics
- Stored as Equi-Width (Singleton) or Equi-Height
  - Visible as JSON

- Can not be manually updated
- No obvious easy way to share statistics

- Enabled by default, will be used when available

Histograms in Postgres

- PostgreSQL histograms are collected by doing a true random read
  - Can be collected manually with ANALYZE
  - Also collected automatically when VACUUM runs
- Stores equal-height and most common values at the same time
  - Equal-height histogram doesn't cover MCV
- Can be manually updated
  - One could import histograms from slave instances
  - VACUUM auto-collection seems to cover the use case

Histograms in Postgres

- Histograms are useful for range conditions
  - Equi-width or equi-height:
    COLUMN > constant
- Most Common Values (Singleton):
  - COLUMN = constant

- Problematic when multiple columns are involved:
  - t1.COL1 > 100 AND t1.COL2 > 1000

- Most optimizers assume column values are independent
  - $P(A \cap B) = P(A) * P(B)$ vs $P(A \cap B) = P(A) * P(B \mid A)$

- PostgreSQL 10 has added support for multi-variable distributions.
  - MySQL assumes independent values.
- MariaDB doesn't handle multi-variable case well either.

Histograms Performance

Sample database world:

```
select city.name
from city
where (city.population > 10 mil or
       city.population < 10 thousand)
```

	MariaDB	MySQL	PostgreSQL
Estimated Rows Filtered	1.95%	1.09%	1.05%
Actual Rows Filtered	1.05 %		

Histograms Performance

- Table with 2 columns A and B
  - t1.a always equals t1.b
  - 10 distinct values, each value occurs with 10% probability

```
select t1.A, t1.B
from t1
where t1.A = t1.B and t1.A = 5
```

	MariaDB	MySQL	PostgreSQL
Estimated Rows Filtered	1%	1%	10%
Actual Rows Filtered	10 %		

Histograms Performance

- Both MySQL and MariaDB don't fare well with correlated column conditions
- MySQL is a bit more precise than MariaDB
- Harder to exchange histograms between MySQL instances
- Performance is no better than full-table-scan for MariaDB/MySQL so far.

Query Optimizations

Query rewriting can lead to significantly better query plans!

Background about optimizations

- A derived table is a table in the FROM clause, defined as a subquery.

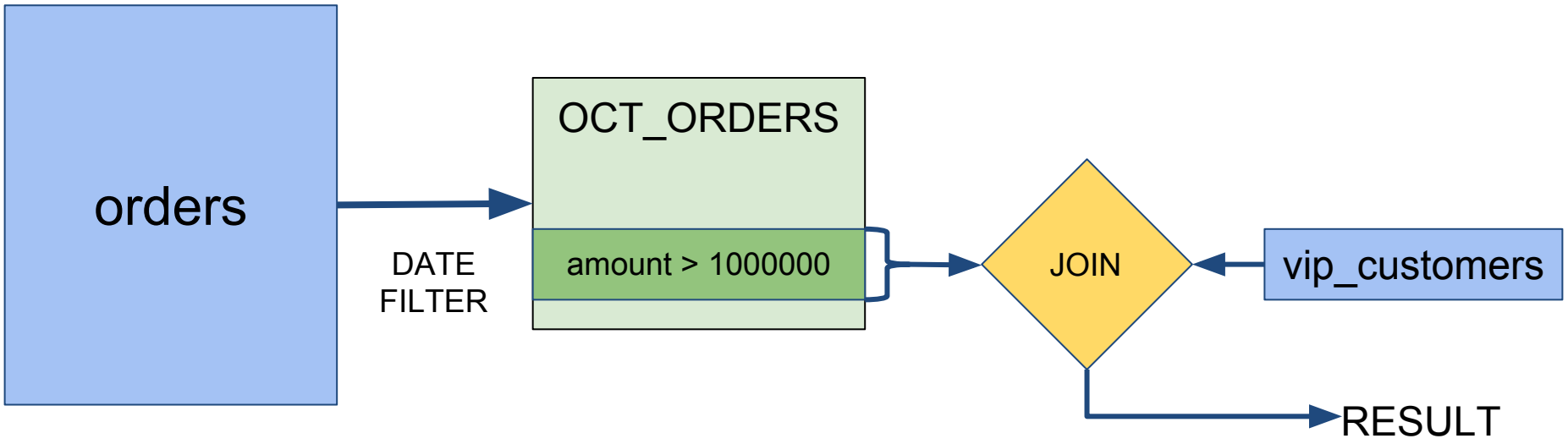
```
SELECT * FROM (SELECT a from t1) der_t1;
```

VIP Customers and their orders

```
select *
from vip_customers,
     (select *
      from orders
      where order_date
               between '2017-10-01' and '2017-10-31') as
     OCT_ORDERS
where OCT_ORDERS.amount > 1000000 and
      OCT_ORDERS.customer_id = vip_customers.customer_id;
```

Naive Execution

```
select *
from vip_customers,
     (select *
      from orders
      where order_date
              between '2017-10-01' and '2017-10-31') as
     OCT_ORDERS
where OCT_ORDERS.amount > 1000000 and
      OCT_ORDERS.customer_id = vip_customers.customer_id;
```



Derived Table Merge

```
select *
from
  vip_customers vc,
  (select *
   from orders
   where
    order_date between
     '2017-10-01' and '2017-10-31'
  ) as OCT_ORDERS
where
  OCT_ORDERS.amount > 1M and
  OCT_ORDERS.customer_id =
          vc.customer_id;
```

```
select *
from
  vip_customers vc,
  orders
where
  OCT_ORDERS.amount > 1M and
  OCT_ORDERS.customer_id =
          vc.customer_id and
  order_date between
   '2017-10-01' and '2017-10-31';
```

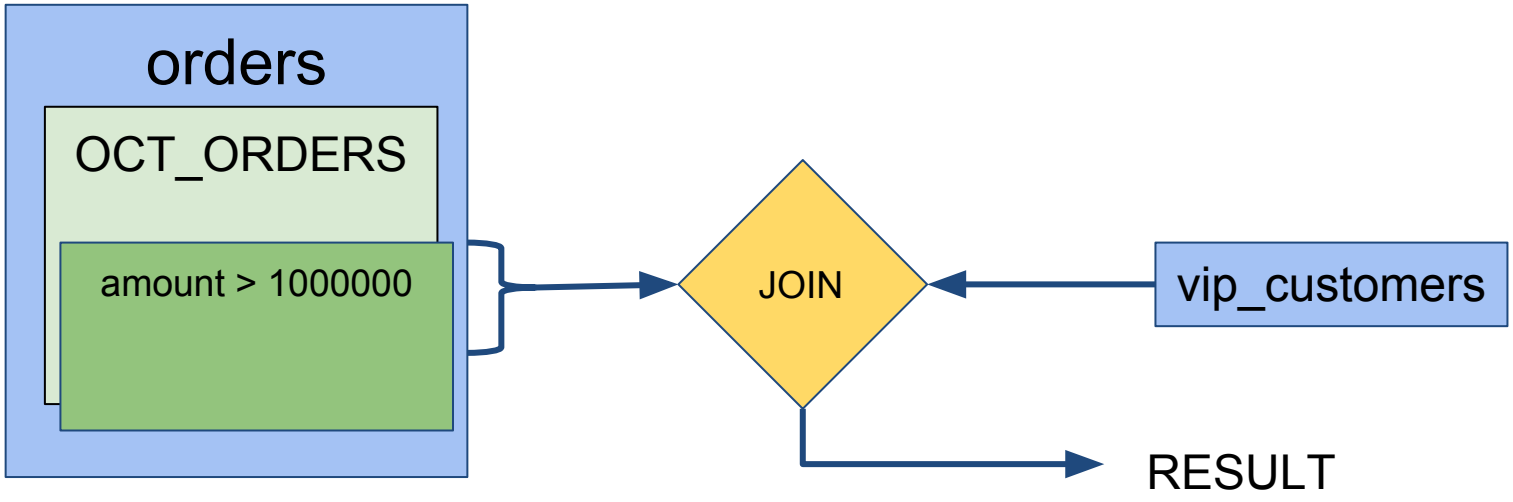
Explain shows the table being merged

```
select *
from vip_customers,
     (select *
      from orders
      where order_date
               between '2017-10-01' and '2017-10-31') as
     OCT_ORDERS
where OCT_ORDERS.amount > 1000000 and
      OCT_ORDERS.customer_id = vip_customers.customer_id;

16649 rows in set (7.64 sec)
```

```
+----+-------------+---------------+------+..+---------+-------------+
|id  | select_type | table         | type |..| rows    | Extra       |
+----+-------------+---------------+------+..+---------+-------------+
|  1 | SIMPLE      | vip_customers | ALL  |..|     101 |             |
|  1 | SIMPLE      | orders        | ALL  |..| 1000000 | Using where;|
+----+-------------+---------------+------+..+---------+-------------+
```

Execution after merge

```
select *
from
   vip_customers vc,
   orders
where
   orders.amount > 1M and
   orders.customer_id = vc.customer_id and
   order_date between '2017-10-01' and '2017-10-31';
```

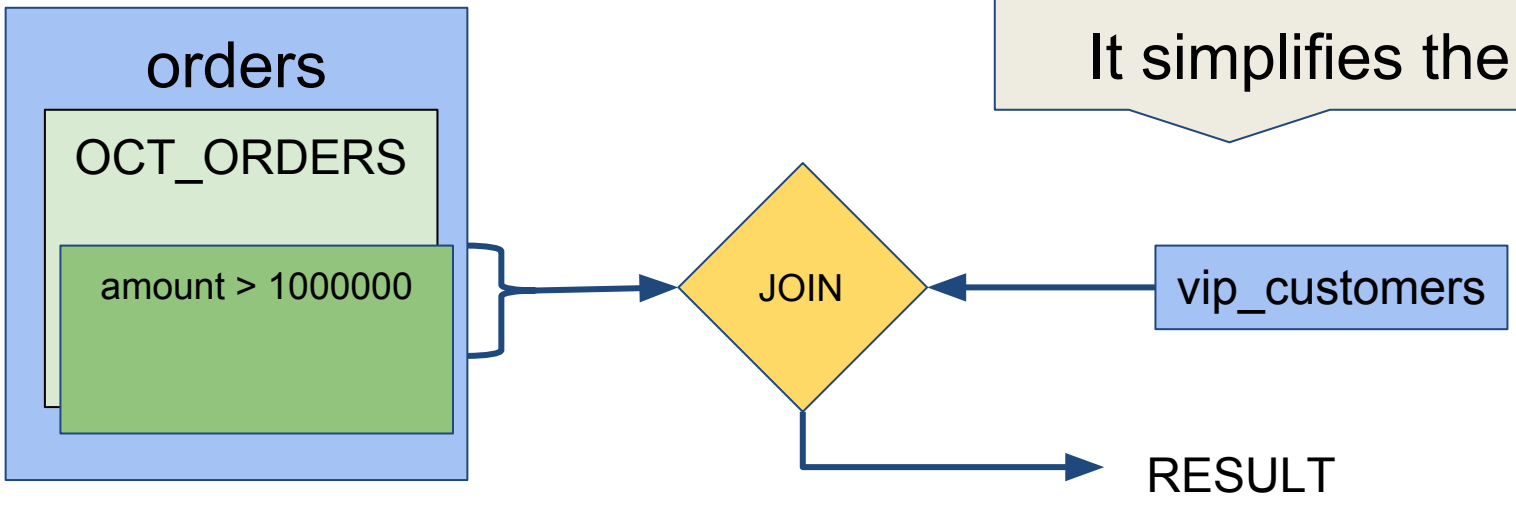


Execution after merge

```
select *
from
   vip_customers vc,
   orders
where
   orders.amount > 1M and
   orders.customer_id = vc.customer_id and
   order_date between '2017-10-01' and
```

Merging is good!

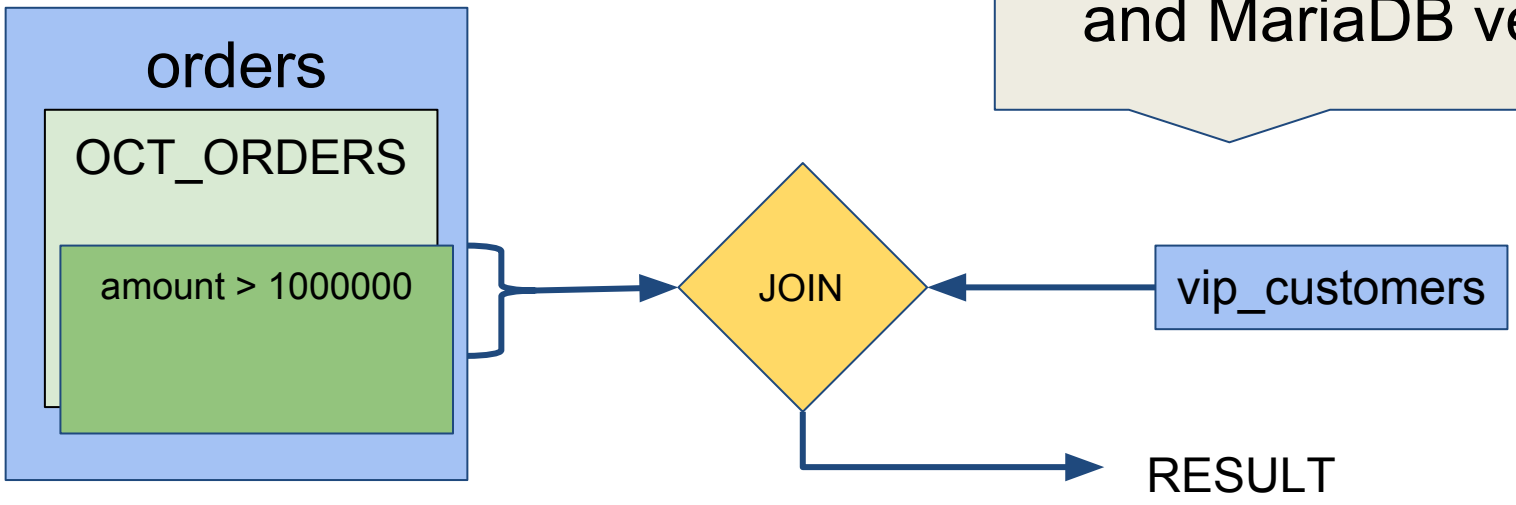
It simplifies the query.



Execution after merge

```
select *
from
   vip_customers vc,
   orders
where
   orders.amount > 1M and
   orders.customer_id = vc.customer_id and
   order_date between '2017-10-01' and
```

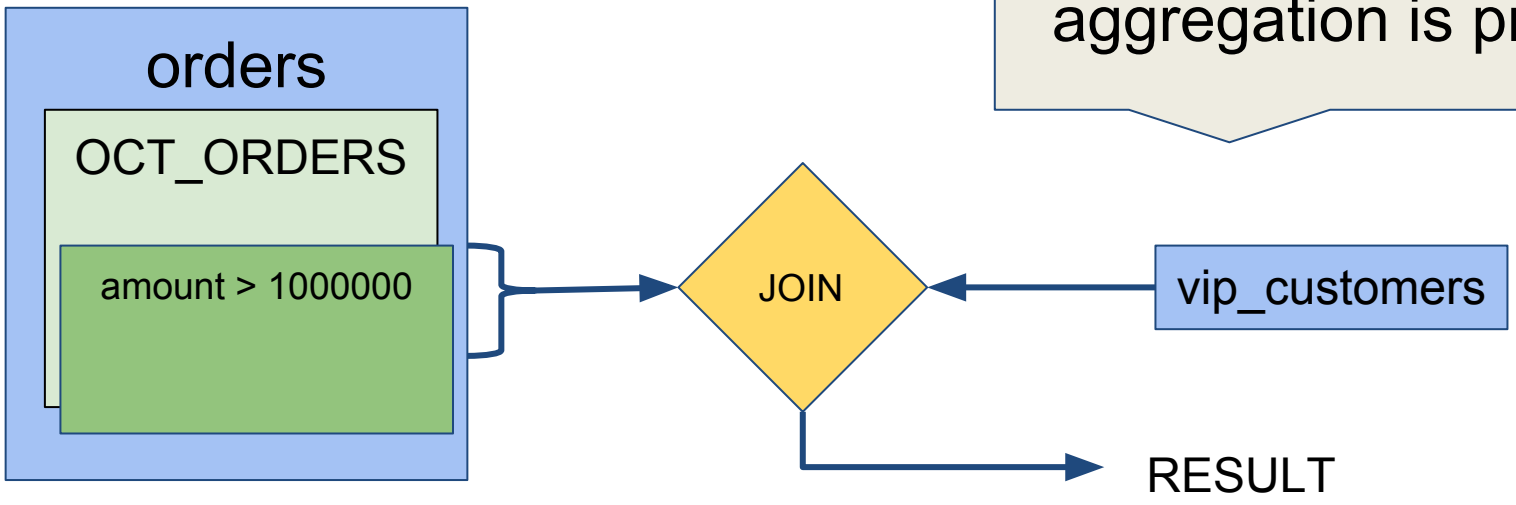
Works in all stable MySQL and MariaDB versions



Execution after merge

```
select *
from
  vip_customers vc,
  orders
where
  orders.amount > 1M and
  orders.customer_id = vc.customer_id and
  order_date between '2017-10-01' and
```

Can not be used when aggregation is present :(



Condition pushdown

```
create view OCT_TOTALS as
select customer_id, SUM(amount) as TOTAL_AMT
from orders
where order_date between '2017-10-01' and '2017-10-31'
group by
  customer_id
```

```
select *
from OCT_TOTALS
where customer_id=1
```

Condition pushdown

```
create view OCT_TOTALS as
select customer_id, SUM(amount) as TOTAL_AMT
from orders
where order_date between '2017-10-01' and '2017-10-31'
group by
  customer_id
```

```
select *
from OCT_TOTALS
where customer_id=1
```

There are a lot of customers and we only want a total for one.

Condition pushdown

```
create view OCT_TOTALS as
select customer_id, SUM(amount) as TOTAL_AMT
from orders
where order_date between '2017-10-01' and '2017-10-31'
group by
  customer_id
```

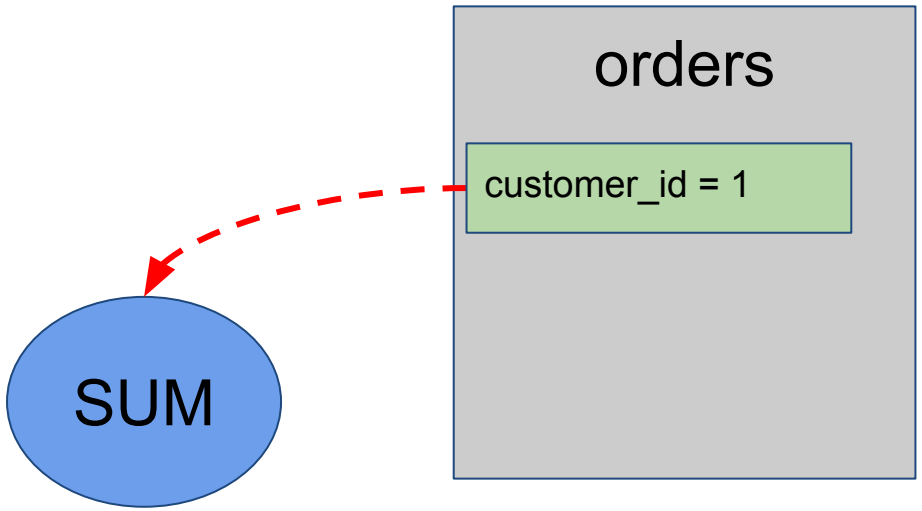
```
select *
from OCT_TOTALS
where customer_id=1
```

We can push the condition to the where clause!

Condition pushdown

```
create view OCT_TOTALS as
select customer_id, SUM(amount) as TOTAL_AMT
from orders
where order_date between '2017-10-01' and '2017-10-31'
group by
  customer_id
```

```
select *
from OCT_TOTALS
where customer_id=1
```



Condition pushdown

```
create view OCT_TOTALS as
select customer_id, SUM(amount) as TOTAL_AMT
from orders
where order_date between '2017-10-01' and '2017-10-31'
group by
  customer_id
```

```
select *
from OCT_TOTALS
where customer_id=1
```

All this is available in MariaDB 10.2

Condition pushdown

```
create view OCT_TOTALS as
select customer_id, SUM(amount) as TOTAL_AMT
from orders
where order_date between '2017-10-01' and '2017-10-31'
group by
  customer_id
```

```
select *
from OCT_TOTALS
where customer_id=1
```

This tactic works with window functions too!

Condition pushdown through Partition By

```
create view top_three_orders as
select * from (
 select customer_id, amount,
        rank() over (partition by customer_id
                     order by amount desc) as order_rank
from orders) as ordered_orders
where order_rank < 3
```

Condition pushdown through Partition By

```
create view top_three_orders as
select * from (
 select customer_id, amount,
        rank() over (partition by customer_id
                     order by amount desc) as order_rank
from orders) as ordered_orders
where order_rank < 3
```

```
+-------------+--------+------------+
| customer_id | amount | order_rank |
+-------------+--------+------------+
|           1 |  10000 |          1 |
|           1 |   9500 |          2 |
|           1 |    400 |          3 |
|           2 |   3200 |          1 |
|           2 |   1000 |          2 |
|           2 |    400 |          3 |
.....................................
```

Condition pushdown through Partition By

```
create view top_three_orders as
select * from (
 select customer_id, amount,
        rank() over (partition by customer_id
                     order by amount desc) as order_rank
from orders) as ordered_orders
where order_rank < 3
```

```
+-------------+--------+------------+
| customer_id | amount | order_rank |
+-------------+--------+------------+
|           1 |  10000 |          1 |
|           1 |   9500 |          2 |
|           1 |    400 |          3 |
|           2 |   3200 |          1 |
|           2 |   1000 |          2 |
|           2 |    400 |          3 |
.....................................
```

```
select * from top_three_orders where customer_id=1
```

Condition pushdown through PARTITION BY

```
create view top_three_orders as
select * from (
 select customer_id, amount,
        rank() over (partition by customer_id
                     order by amount desc) as order_rank
from orders) as ordered_orders
where order_rank < 3
```

```
+-------------+--------+------------+
| customer_id | amount | order_rank |
+-------------+--------+------------+
|           1 |  10000 |          1 |
|           1 |   9500 |          2 |
|           1 |    400 |          3 |
|           2 |   3200 |          1 |
|           2 |   1000 |          2 |
|           2 |    400 |          3 |
.....................................
```

```
select * from top_three_orders where customer_id=1
```

Condition pushdown through PARTITION BY

MariaDB 10.2, MySQL 8.0, MariaDB 10.3 Comparison

MariaDB 10.2, MySQL 8.0

- Compute top_three_orders for **all** customers
- Select rows with customer_id=1

MariaDB 10.3 (and e.g. PostgreSQL)

- Only compute top_three_orders for customer_id=1
- This can be much faster!
- Can make use of index(customer_id)

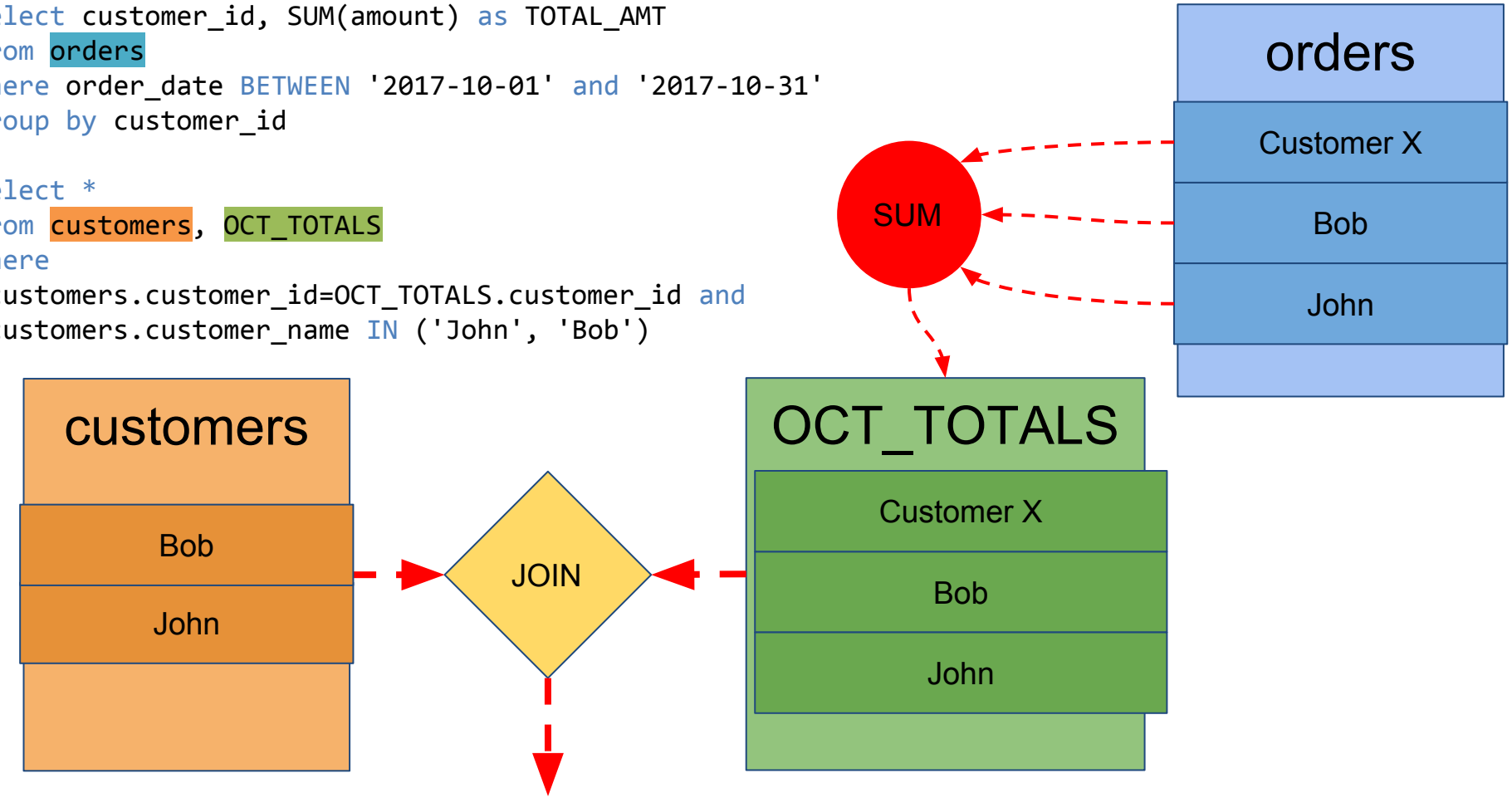
Split grouping for derived

```
create view OCT_TOTALS as
select customer_id, SUM(amount) as TOTAL_AMT
from orders
where
   order_date BETWEEN '2017-10-01' and '2017-10-31'
group by customer_id
```

```
select *
from customers, OCT_TOTALS
where customers.customer_id=OCT_TOTALS.customer_id and
      customers.customer_name IN ('John', 'Bob')
```

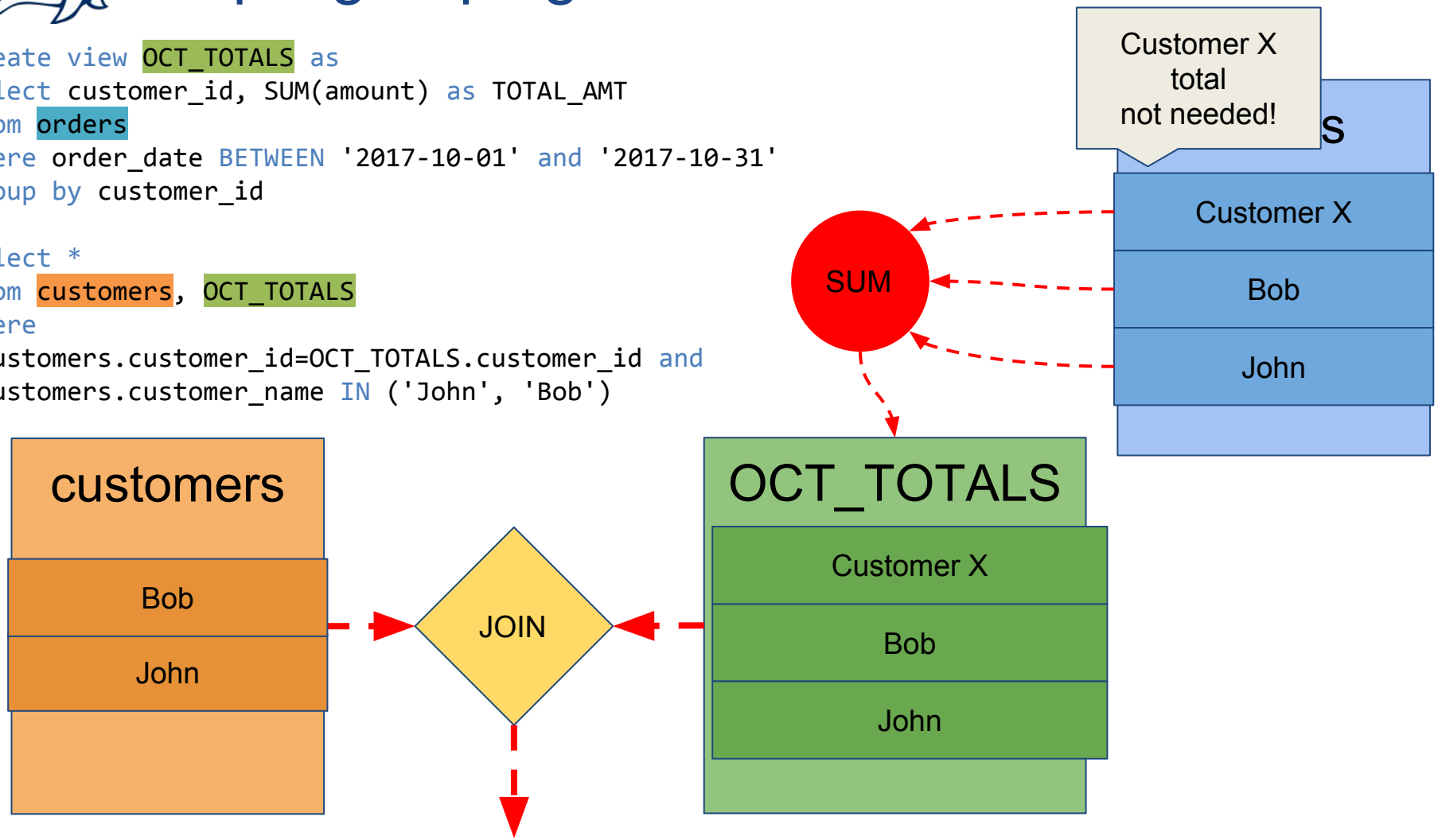
Split grouping for derived

```
create view OCT_TOTALS as
select customer_id, SUM(amount) as TOTAL_AMT
from orders
where order_date BETWEEN '2017-10-01' and '2017-10-31'
group by customer_id

select *
from customers, OCT_TOTALS
where
 customers.customer_id=OCT_TOTALS.customer_id and
 customers.customer_name IN ('John', 'Bob')
```



Split grouping for derived

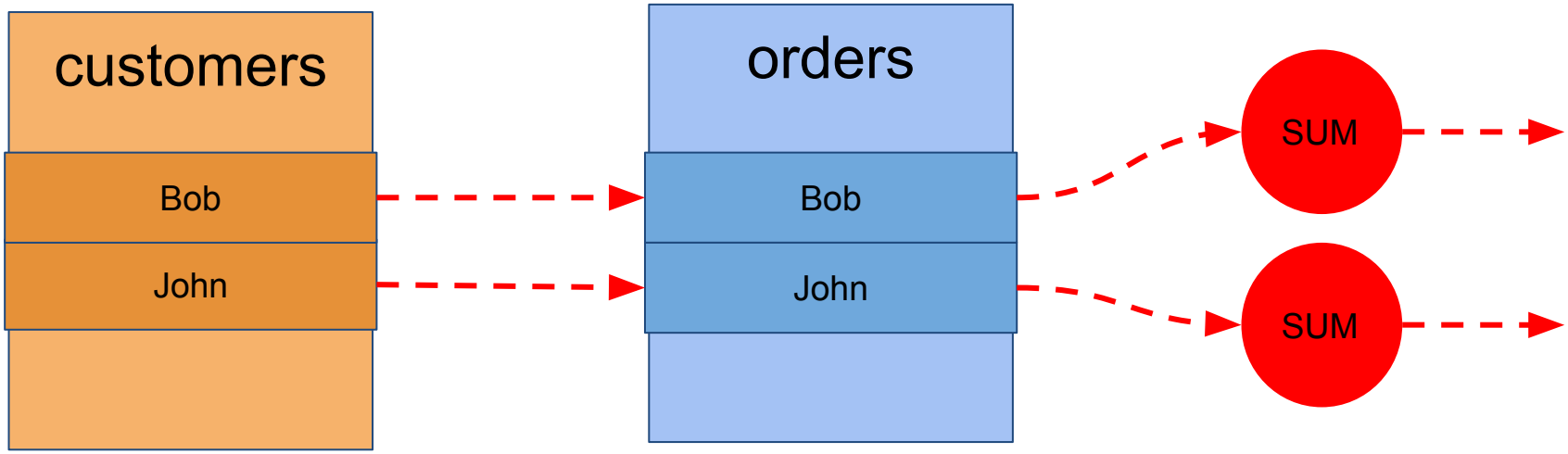
```
create view OCT_TOTALS as
select customer_id, SUM(amount) as TOTAL_AMT
from orders
where order_date BETWEEN '2017-10-01' and '2017-10-31'
group by customer_id

select *
from customers, OCT_TOTALS
where
 customers.customer_id=OCT_TOTALS.customer_id and
 customers.customer_name IN ('John', 'Bob')
```



Split grouping execution

Figure out which orders we need to aggregate first!

Aggregate each group individually.



Split grouping execution requirements

- Can be used when doing join from customer to orders
- Must have equalities for GROUP BY columns: OCT_TOTALS.customer_id=customer.customer_id
  - This allows to select one group
- The underlying table (orders) must have an index on the GROUP BY column (customer_id)
  - This allows to use ref access

Condition pushdown into in-subqueries

- Feature implemented for 10.4
- Shortcut subquery execution by adding more specific conditions from the outer query to in subqueries.

Condition pushdown into in-subqueries

```
SELECT product, release_year, years_waranty
FROM products
WHERE release_year >= 2010 and release_year <= 2015 and
      years_waranty > 1 and
      (release_year, years_waranty) in (
          SELECT competitor_release_year, max(competitor_years_waranty)
          FROM competitor_products
          GROUP BY competitor_release_year
     )
```

Condition pushdown into in-subqueries

```
SELECT product, release_year, years_waranty
FROM products
WHERE release_year >= 2010 and release_year <= 2015 and
      years_waranty > 1 and
      (release_year, years_waranty) in (
          SELECT competitor_release_year, max(competitor_years_waranty)
          FROM competitor_products
          GROUP BY competitor_release_year
     )
```

Condition pushdown into in-subqueries

```
SELECT product, release_year, years_waranty
FROM products
WHERE release_year >= 2010 and release_year <= 2015 and
      years_waranty > 1 and
      (release_year, years_waranty) in (
          SELECT competitor_release_year, max(competitor_years_waranty)
          FROM competitor_products
          GROUP BY competitor_release_year
     )
```

Condition pushdown into in-subqueries

```
SELECT product, release_year, years_waranty
FROM products
WHERE release_year >= 2010 and release_year <= 2015 and
      years_waranty > 1 and
      (release_year, years_waranty) in (
          SELECT competitor_release_year, max(competitor_years_waranty)
          FROM competitor_products
          GROUP BY competitor_release_year
     )
```

Condition pushdown into in-subqueries

```
SELECT product, release_year, years_waranty
FROM products
WHERE release_year >= 2010 and release_year <= 2015 and
      years_waranty > 1 and
      (release_year, years_waranty) in (
          SELECT competitor_release_year, max(competitor_years_waranty)
          FROM competitor_products
          GROUP BY competitor_release_year
      )
```

```
SELECT product, release_year, years_waranty
FROM products
WHERE
      (release_year, years_waranty) in (
          SELECT competitor_release_year, max(competitor_years_waranty)
          FROM competitor_products
          WHERE competitor_release_year >= 2010 and
                competitor_release_year <= 2015
          GROUP BY competitor_release_year
          HAVING years_waranty > 1
      )
```

Condition pushdown into in-subqueries

- Restrictions:
  - Where condition can be pushed only if the corresponding column is within GROUP BY in the subquery
  - Where condition maps to aggregate function within subquery.
- Provides speedup when subquery was expensive

Conclusions

- MariaDB 10.2: **Condition pushdown for derived tables** optimization
  - Push a condition into derived table
  - Used when derived table cannot be merged
  - Biggest effect is for subqueries with GROUP BY
- MariaDB 10.3: **Condition Pushdown through Window functions' partition by**
- MariaDB 10.3: **Split grouping for derived** optimization
- MariaDB 10.4: **Condition Pushdown into in-subqueries**.

Thank You!

Contact me at:
vicentiu@mariadb.org

Blog:
mariadb.org/blog