Altinity WEBINAR

CLICKHOUSE MATERIALIZED VIEWS: THE MAGIC CONTINUES

-2

with Robert Hodges

Introduction to Presenter

Robert Hodges - Altinity CEO

30+ years on DBMS plus virtualization and security.

ClickHouse is DBMS #20

Altinity

www.altinity.com

Leading software and services provider for ClickHouse

Major committer and community sponsor in US and Western Europe

Introduction to ClickHouse

Understands SQL

Runs on bare metal to cloud

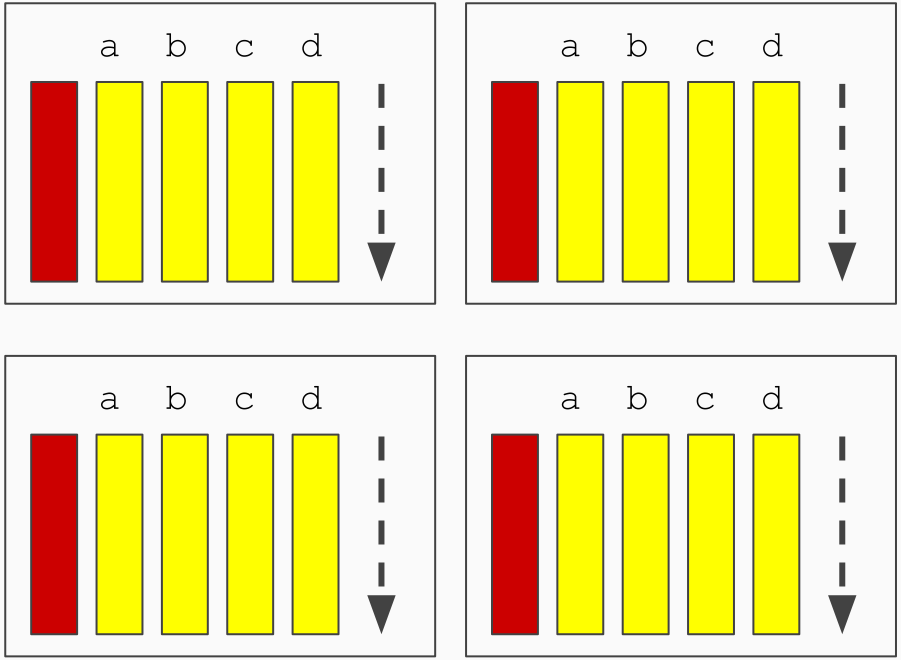
Stores data in columns

Parallel and vectorized execution

Scales to many petabytes

Is Open source (Apache 2.0)

Is WAY fast!



Materialized views for basic aggregates

Computing aggregates on large datasets

Taxi & Limousine Commission

Our favorite dataset!

1.31 billion rows of data from taxi and limousine rides in New York City

Question:

What is the average number of taxi passengers by month?

ClickHouse can solve it with a simple query

```
SELECT
   toYYYYMM(pickup_date) AS month,
   sum(passenger_count) AS passenger_count
FROM nyc_taxi_rides.tripdata
GROUP BY month
ORDER BY month ASC
LIMIT 10
. . .
10 rows in set. Elapsed: 2.105 sec. Processed 1.31
billion rows, 3.93 GB (622.85 million rows/s., 1.87
GB/s.)
```

Can we make it faster?

We can add a materialized view

```
CREATE MATERIALIZED VIEW passenger_daily_mv
ENGINE = SummingMergeTree()
PARTITION BY tuple()
ORDER BY pickup_date
POPULATE
AS SELECT
  pickup_date,
  sum(passenger_count) AS passenger_count
  FROM nyc_taxi_rides.tripdata GROUP BY pickup_date
```

How to sum

Auto-load after create

Engine

How to derive data

Now we can query the materialized view

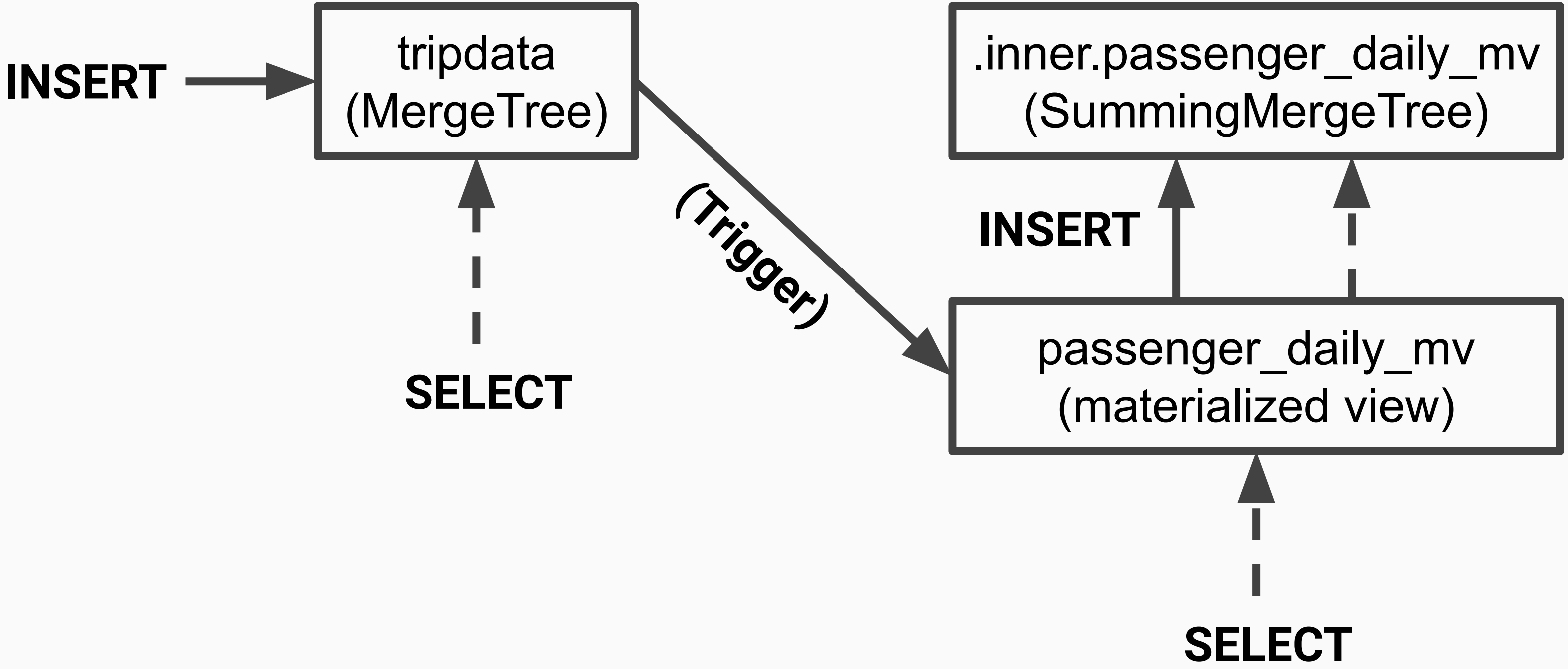
```
SELECT
   toYYYYMM(pickup_date) AS month,
   sum(passenger_count) AS passenger_count
FROM passenger_daily_mv
GROUP BY month
ORDER BY month ASC
LIMIT 10
. . .
10 rows in set. Elapsed: 0.002 sec. Processed 2.92
thousand rows, 90.66 KB (1.53 million rows/s., 47.36
MB/s.)
```

Materialized view is 1000 times faster!

What's going on under the covers?

Compressed size: ~43GB
Uncompressed size: ~104GB

Compressed size: ~36KB
Uncompressed size: ~47KB



Materialized views really are just triggers

```
-- Source table
CREATE TABLE number_table (
    value Int32
) ENGINE = Null

-- Transform data
CREATE MATERIALIZED VIEW square_mv
ENGINE = Log() AS
SELECT value * value AS square
FROM number_table
```

Source and target tables can be any engine type

The view “transforms” data

```
INSERT INTO number_table VALUES (1), (4)
```

```
SELECT * FROM square_mv
```

```
┌─square─┐
│      1 │
│      4 │
└────────┘
```

What is going on in the target table?

```
CREATE TABLE default.`.inner.passenger_daily_mv`
(
    `pickup_date` Date,
    `passenger_count` UInt64
)
ENGINE = SummingMergeTree()
PARTITION BY tuple()
ORDER BY pickup_date
```

Non-key number sums automatically

Aggregating table engine type

Sort order controls grouping

Materialized views for complex aggregates

Let’s look at more complex aggregates

```
SELECT min(fare_amount), avg(fare_amount),
  max(fare_amount), sum(fare_amount), count()
FROM tripdata
WHERE (fare_amount > 0) AND (fare_amount < 500.)

...

1 rows in set. Elapsed: 4.640 sec. Processed 1.31
billion rows, 5.24 GB (282.54 million rows/s., 1.13
GB/s.)
```

Can we make all aggregates faster?

Let’s create a view for multiple aggregates

Agg. order

Auto-load after create

```
CREATE MATERIALIZED VIEW tripdata_agg_mv
ENGINE = SummingMergeTree
PARTITION BY toYYYYMM(pickup_date)
ORDER BY (pickup_location_id, dropoff_location_id)
POPULATE AS SELECT
    pickup_date, pickup_location_id,
    dropoff_location_id,
    minState(fare_amount) as fare_amount_min,
    avgState(fare_amount) as fare_amount_avg,
    maxState(fare_amount) as fare_amount_max,
    sumState(fare_amount) as fare_amount_sum,
    countState() as fare_amount_count
FROM tripdata
WHERE (fare_amount > 0) AND (fare_amount < 500.)
GROUP BY
    pickup_date, pickup_location_id, dropoff_location_id
```

Engine

How to derive data

Filter on data

Digression: How aggregation works

Source value → `fare_amount`

Partial aggregate → `avgState(fare_amount)`

Merged aggregate → `avgMerge(fare_amount_avg)`

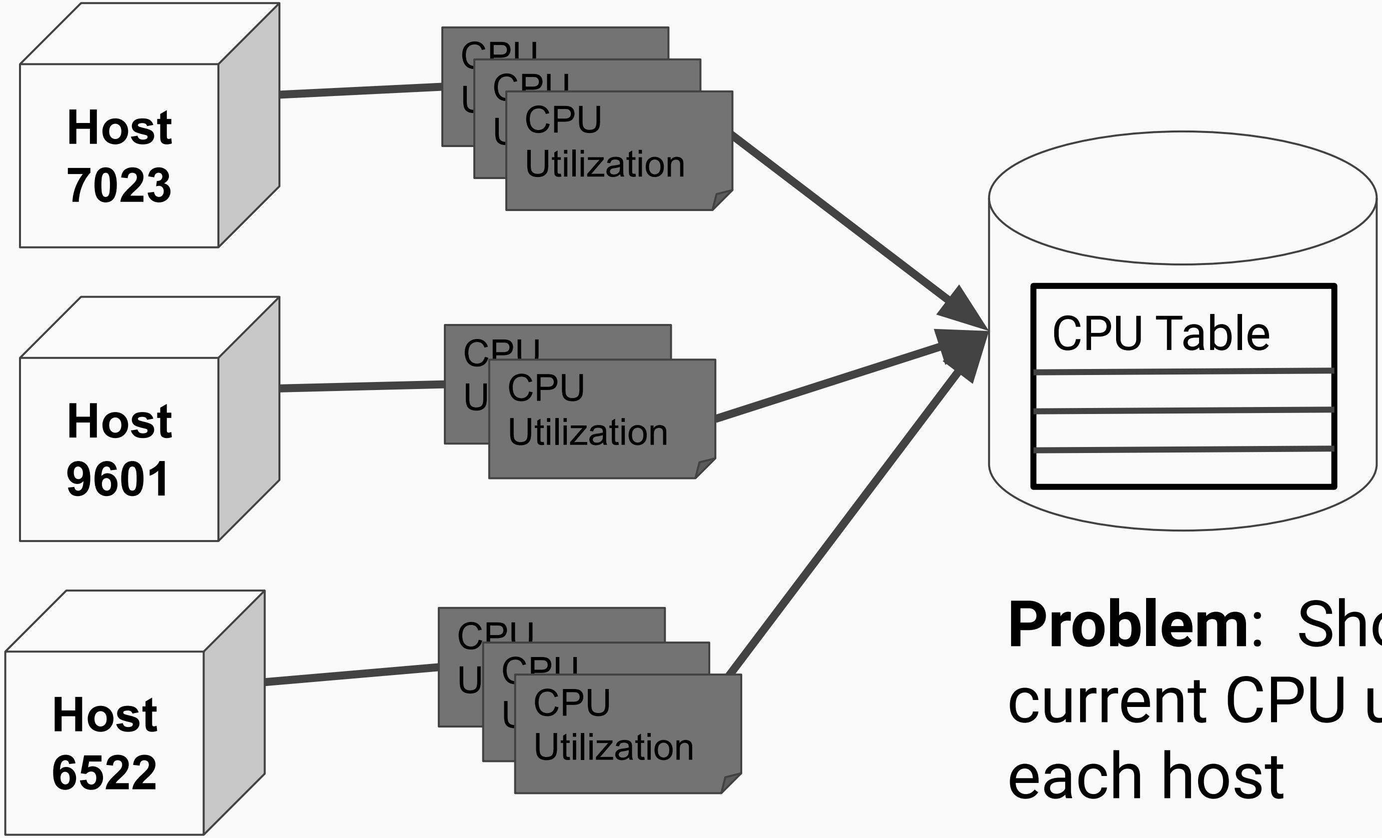
Now let's select from the materialized view

```
SELECT minMerge(fare_amount_min) AS fare_amount_min,
   avgMerge(fare_amount_avg) AS fare_amount_avg,
   maxMerge(fare_amount_max) AS fare_amount_max,
   sumMerge(fare_amount_sum) AS fare_amount_sum,
   countMerge(fare_amount_count) AS fare_amount_count
FROM tripdata_agg_mv

...

1 rows in set. Elapsed: 0.017 sec. Processed 208.86
thousand rows, 23.88 MB (12.54 million rows/s., 1.43
GB/s.)
```

Materialized view is 274 times faster

Materialized views for “last point queries”

Last point problems are common in time series



Problem: Show the current CPU utilization for each host

ClickHouse can solve this using a subquery

```
SELECT t.hostname, tags_id, 100 - usage_idle usage
FROM (
    SELECT tags_id, usage_idle
    FROM cpu
    WHERE (tags_id, created_at) IN
      (SELECT tags_id, max(created_at)
       FROM cpu GROUP BY tags_id)
) AS c
INNER JOIN tags AS t ON c.tags_id = t.id
ORDER BY
    usage DESC,
    t.hostname ASC
LIMIT 10
```

TABLE SCAN!

USE INDEX

OPTIMIZED JOIN COST

SQL queries work but are inefficient

OUTPUT:

hostname	tags_id	usage
host_1002	9003	100
host_1116	9117	100
host_1141	9142	100
host_1163	9164	100
host_1210	9211	100
host_1216	9217	100
host_1234	9235	100
host_1308	9309	100
host_1419	9420	100
host_1491	9492	100

Using direct query on table:

10 rows in set. **Elapsed: 0.566 sec.** Processed 32.87 million rows, 263.13 MB (53.19 million rows/s., 425.81 MB/s.)

Can we bring last point performance closer to real-time?

Create an explicit target for aggregate data

```
CREATE TABLE cpu_last_point_idle_agg (
  created_date AggregateFunction(argMax, Date, DateTime),
  max_created_at AggregateFunction(max, DateTime),
  time AggregateFunction(argMax, String, DateTime),
  tags_id UInt32,
  usage_idle AggregateFunction(argMax, Float64, DateTime)
)
ENGINE = AggregatingMergeTree()
PARTITION BY tuple()
ORDER BY tags_id
```

TIP: ‘tuple()’ means don’t partition and don’t sort data

TIP: This is a new way to create materialized views in ClickHouse

argMaxState links columns with aggregates

MV table

```
CREATE MATERIALIZED VIEW cpu_last_point_idle_mv
TO cpu_last_point_idle_agg
AS SELECT
    argMaxState(created_date, created_at) AS created_date,
    maxState(created_at) AS max_created_at,
    argMaxState(time, created_at) AS time,
    tags_id,
    argMaxState(usage_idle, created_at) AS usage_idle
FROM cpu
GROUP BY tags_id
```

Derive data

Understanding how argMaxState works

(Same row)

Source values

`created_at` ---------→ `usage_idle`

(Pick usage_idle value from any row with matching created_at)

Partial aggregates

`maxState(created_at)` ----→ `argMaxState(usage_idle, created_at)`

(Pick usage_idle from aggregate with matching created_at)

Merged aggregates

`avgMerge(created_at)` ----→ `avgMaxMerge(usage_idle)`

Load materialized view using a SELECT

```
INSERT INTO cpu_last_point_idle_mv
SELECT
    argMaxState(created_date, created_at) AS created_date,
    maxState(created_at) AS max_created_at,
    argMaxState(time, created_at) AS time,
    Tags_id,
    argMaxState(usage_idle, created_at) AS usage_idle
FROM cpu
GROUP BY tags_id
```

POPULATE keyword is not supported

Let’s hide the merge details with a view

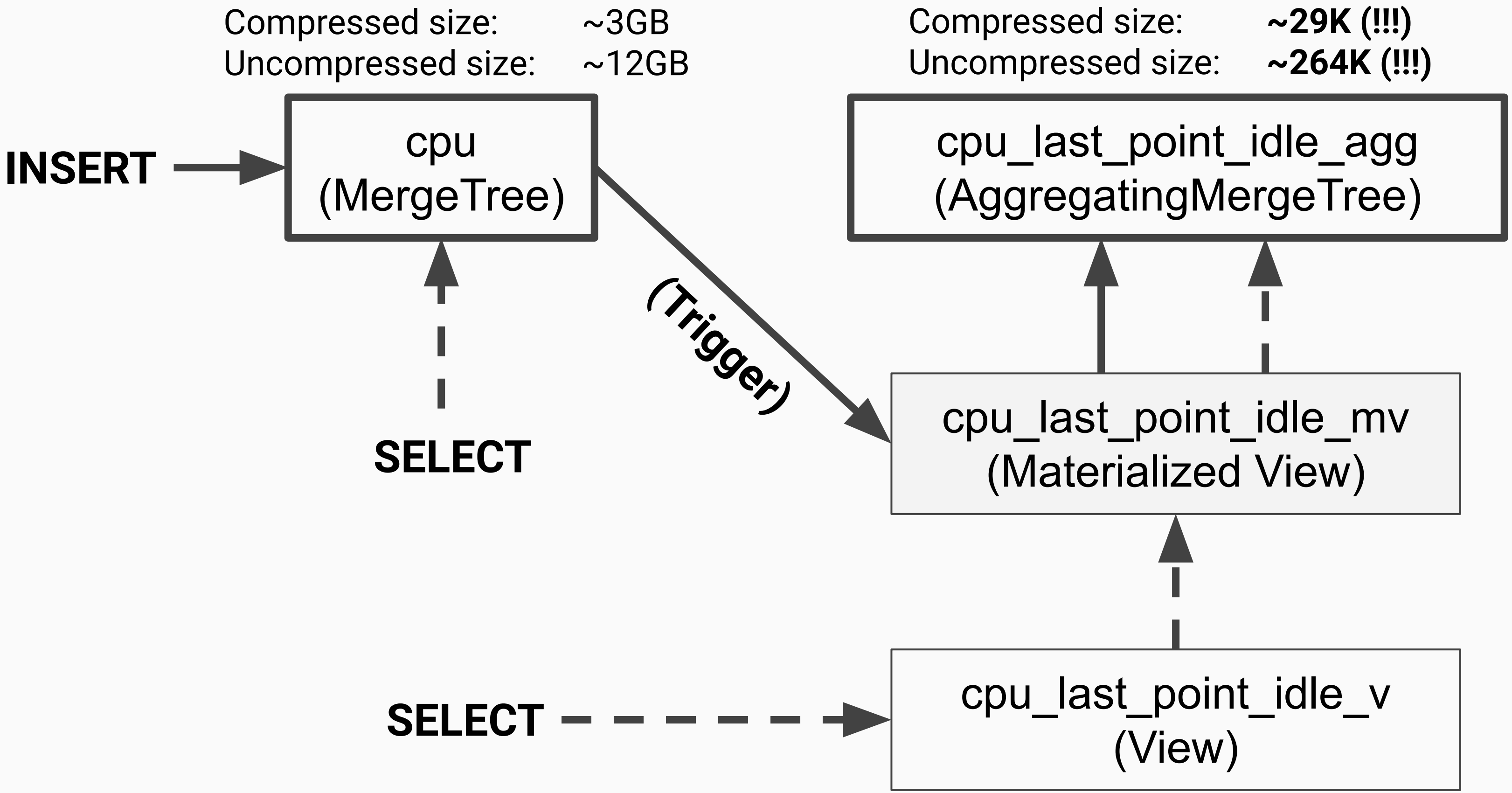
```
CREATE VIEW cpu_last_point_idle_v AS
SELECT
   argMaxMerge(created_date) AS created_date,
   maxMerge(max_created_at) AS created_at,
   argMaxMerge(time) AS time,
   tags_id,
   argMaxMerge(usage_idle) AS usage_idle
FROM cpu_last_point_idle_mv
GROUP BY tags_id
```

...Select again from the covering view

```
SELECT t.hostname, tags_id, 100 - usage_idle usage
FROM cpu_last_point_idle_v AS b
INNER JOIN tags AS t ON b.tags_id = t.id
ORDER BY usage DESC, t.hostname ASC
LIMIT 10

...

10 rows in set. Elapsed: 0.005 sec. Processed 14.00
thousand rows, 391.65 KB (2.97 million rows/s., 82.97
MB/s.)
```

Last point view is 113 times faster

Another view under the covers



Using views to remember unique visitors

It’s common to drop old data in large tables

```
CREATE TABLE traffic (
  datetime DateTime,
  date Date,
  request_id UInt64,
  cust_id UInt32,
  sku UInt32
) ENGINE = MergeTree
PARTITION BY toYYYYMM(datetime)
ORDER BY (cust_id, date)
TTL datetime + INTERVAL 90 DAY
```

ClickHouse can aggregate unique values

```
-- Find which hours have most unique visits on
-- SKU #100
SELECT
  toStartOfHour(datetime) as hour,
  uniq(cust_id) as uniqs
FROM purchases
WHERE sku = 100
GROUP BY hour
ORDER BY uniqs DESC, hour ASC
LIMIT 10
```

How can we remember deleted visit data?

Create a target table for unique visit data

```
CREATE TABLE traffic_uniqs_agg (
  hour DateTime,
  cust_id_uniqs AggregateFunction(uniq, UInt32),
  sku UInt32
)
ENGINE = AggregatingMergeTree()
PARTITION BY toYYYYMM(hour)
ORDER BY (sku, hour)
```

Aggregate for unique values

Engine

Agg. order

Need to partition

Use uniqState to collect unique customers

```
CREATE MATERIALIZED VIEW traffic_uniqs_mv
TO traffic_uniqs_agg
AS SELECT
  toStartOfHour(datetime) as hour,
  uniqState(cust_id) as cust_id_uniqs,
  Sku
FROM traffic
GROUP BY hour, sku
```

Now we can remember visitors indefinitely!

```
SELECT
  hour,
  uniqMerge(cust_id_uniqs) as uniqs
FROM traffic_uniqs_mv
WHERE sku = 100
GROUP BY hour
ORDER BY uniqs DESC, hour ASC
LIMIT 10
```

Check view size regularly!

```
SELECT
    table, count(*) AS columns,
    sum(data_compressed_bytes) AS tc,
    sum(data_uncompressed_bytes) AS tu
FROM system.columns
WHERE database = currentDatabase()
GROUP BY table
```

Uniq hash tables can be quite large

table	columns	tc	tu
traffic_uniqs_agg	3	52620883	90630442
traffic	5	626152255	1463621588
traffic_uniqs_mv	3	0	0

More tricks with materialized views

Chaining mat views to create pipelines

```
CREATE TABLE t1 ( value Int32 ) ENGINE = Log();
CREATE TABLE t2 ( value Int32 ) ENGINE = Log();
CREATE TABLE t3 ( value Int32 ) ENGINE = Log();

CREATE MATERIALIZED VIEW t1_t2_mv TO t2
AS SELECT value from t1

CREATE MATERIALIZED VIEW t2_t3_mv TO t3
AS SELECT value from t2

INSERT INTO t1 VALUES (35)
SELECT * FROM t3
┌─value─┐
│    35 │
└───────┘
```

WARNING: Chained updates are not atomic

SimpleAggregateFunction simplifies aggregation

```
CREATE TABLE simple_agg (
    id UInt64,
    my_sum SimpleAggregateFunction(sum, Double),
    my_max SimpleAggregateFunction(max, Double),
    my_min SimpleAggregateFunction(min, Double)
) ENGINE = AggregatingMergeTree ORDER BY id

INSERT INTO simple_agg VALUES (1, 2, 3, 4);
INSERT INTO simple_agg VALUES (1, 2, 5, 1);

SELECT * from simple_agg;

SELECT * from simple_agg final;
```

No aggregation

Automatically aggregated by id

Creating a view that accepts only future data

```
CREATE MATERIALIZED VIEW sales_amount_mv
TO sales_amount_agg
AS SELECT
   toStartOfHour(datetime) as hour,
   sumState(amount) as amount_sum,
   sku
FROM sales
WHERE
   datetime >= toDateTime('2020-02-12 00:00:00')
GROUP BY hour, sku
```

How to derive data

Accept only future rows from source

Loading past data using a select

```
INSERT INTO sales_amount_agg
SELECT
   toStartOfHour(datetime) as hour,
   sumState(amount) as amount_sum,
   sku
FROM sales
WHERE
   datetime < toDateTime('2020-02-12 00:00:00')
GROUP BY hour, sku
```

Load to view target table

Accept only past rows from source

Add an aggregate to MV with inner table

```
-- Detach view from server.
DETACH TABLE sales_amount_inner_mv

-- Update base table
ALTER TABLE `.inner.sales_amount_inner_mv` ADD COLUMN
`amount_avg` AggregateFunction(avg, Float32)
AFTER amount_sum
```

TIP: Stop source INSERTs to avoid losing data <u>or</u> load data later

Check order carefully!

Add column to MV with inner table (cont.)

```
-- Re-attach view with modifications.
ATTACH MATERIALIZED VIEW sales_amount_inner_mv
ENGINE = AggregatingMergeTree()
PARTITION BY toYYYYMM(hour)
ORDER BY (sku, hour)
AS SELECT
  toStartOfHour(datetime) as hour,
  sumState(amount) as amount_sum,
  avgState(amount) as amount_avg,
  sku
FROM sales
GROUP BY hour, sku
```

Specify table

Empty until new data arrive

‘TO’ option views are easier to change

```
-- Drop view
DROP TABLE sales_amount_mv

-- Update target table
ALTER TABLE sales_amount_agg ADD COLUMN
`amount_avg` AggregateFunction(avg, Float32)
AFTER amount_sum

-- Recreate view
CREATE MATERIALIZED VIEW sales_amount_mv TO sales_amount_agg
AS SELECT toStartOfHour(datetime) as hour,
  sumState(amount) as amount_sum,
  avgState(amount) as amount_avg,
  sku
FROM sales GROUP BY hour, sku
```

Empty until new data arrive

Add a dimension to the view with TO table

```
-- Drop view
DROP TABLE sales_amount_mv

-- Update target table
ALTER TABLE sales_amount_agg
  ADD COLUMN cust_id UInt32 AFTER sku,
  MODIFY ORDER BY (sku, hour, cust_id)

-- Recreate view
CREATE MATERIALIZED VIEW sales_amount_mv TO sales_amount_agg
AS SELECT
  toStartOfHour(datetime) as hour,
  sumState(amount) as amount_sum,
  avgState(amount) as amount_avg,
  sku, cust_id
FROM sales GROUP BY hour, sku, cust_id
```

Empty until new data arrive

Final trick to look like a ClickHouse genius

"Use the source, Luke!"

Look at the ClickHouse tests:

https://github.com/ClickHouse/ClickHouse/tree/master/dbms/tests

Conclusion

Key takeaways

- Materialized Views are automatic triggers that transform data
    - Move data from source table to target
- They solve important problems within the database:
    - Speed up query by pre-computing aggregates
    - Make ‘last point’ queries extremely fast
    - Remember aggregates after source rows are dropped
- AggregateFunctions hold partially aggregated data
    - Functions like avgState(X) to put data into the view
    - Functions like avgMerge(X) to get results out
    - Functions like argMaxState(X, Y) link values across columns
- Two “styles” of materialized views
    - TO external table or using implicit ‘inner’ table as target

But wait, there’s even more!

- [Many] More use cases for views -- Any kind of transformation!
  - Reading from specialized table engines like KafkaEngine
  - Different sort orders
  - Using views as indexes
  - Down-sampling data for long-term storage
  - Creating build pipelines

Thank you!

Special Offer:

Contact us for a 1-hour consultation!

Contacts:
info@altinity.com

Visit us at:
https://www.altinity.com

Free Consultation:
https://blog.altinity.com/offer