CLICKHOUSE MATERIALIZED VIEWS

A SECRET WEAPON FOR HIGH PERFORMANCE ANALYTICS

Robert Hodges -- Percona Live 2018 Amsterdam
Introduction to Presenter

Robert Hodges - Altinity CEO

30+ years on DBMS plus virtualization and security.

ClickHouse is DBMS #20

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!
ClickHouse materialized views are triggers

- INSERT
- Source table
  - (Trigger)
  - Target table
- Target table
- Target table
You can even create recursive views
Why might a materialized view be useful?

SELECT
toYear(FlightDate) AS year,
sum_Cancelled_15 / count(*) AS cancelled,
sum(DepDel15) / count(*) AS delayed_15
FROM airline.ontime
GROUP BY year ORDER BY year ASC LIMIT 10

...  
10 rows in set. Elapsed: 0.894 sec. Processed 173.82 million rows, 1.74 GB (194.52 million rows/s., 1.95 GB/s.)

Can we make it faster?
Let’s precompute and store aggregates!

CREATE MATERIALIZED VIEW ontime_daily_cancelled_mv
ENGINE = SummingMergeTree
PARTITION BY tuple() ORDER BY (FlightDate, Carrier)
POPULATE
AS SELECT
    FlightDate, Carrier, count(*) AS flights,
    sum(Cancelled) / count(*) AS cancelled,
    sum(DepDel15) / count(*) AS delayed_15
FROM ontime
GROUP BY FlightDate, Carrier
Query performance impact is significant

SELECT
toYear(FlightDate) AS year,
sum(flights) AS flights,
sum(cancelled) AS cancelled,
sum(delayed_15) AS delayed_15
FROM airline.ontime_dailyCancelled_mv
GROUP BY year ORDER BY year ASC LIMIT 10

...  

10 rows in set. Elapsed: 0.007 sec. Processed 148.16 thousand rows, 3.85 MB (20.37 million rows/s., 529.50 MB/s.)
What's going on under the covers?

Materialized view is 6471x smaller than source table!

Compressed size: ~14.6GB
Uncompressed size: ~55.4GB

Compressed size: ~2.31 MB
Uncompressed size: ~4.10 MB
A brief study of ClickHouse table structures

CREATE TABLE ontime (  
    Year UInt16,  
    Quarter UInt8,  
    Month UInt8,  
    ...  
) ENGINE = MergeTree()  
PARTITION BY toYYYYMM(FlightDate)  
ORDER BY (Carrier, FlightDate)
Possible ways to transform tables

- ontime
- ontime_agg
- ontime_resorted
- ontime_kafka

(Materialized view)

Reduce data
And/Or
Change layout
And/Or
Change table type
Exercise: the famous ‘last point problem’

Problem: Show the current CPU utilization for each host
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
SQL queries work but are inefficient

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 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
argMaxState links columns with aggregates

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
Digression: How aggregation works

Source value → created_at

Partial aggregate → maxState(created_at)

Merged aggregate → maxMerge(max_created_at)
Selecting rows that match max value

Source values

Partial aggregates

Merged aggregates

\[
\text{created_at} \rightarrow \text{usage_idle}
\]

(Same row)

\[
\text{maxState(created_at)} \rightarrow \text{argMaxState(usage_idle, created_at)}
\]

(Pick usage_idle from any row with matching created_at)

\[
\text{avgMerge(created_at)} \rightarrow \text{avgMaxMerge(usage_idle)}
\]

(Pick usage_idle from aggregate with matching created_at)
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 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.)
Common uses for materialized views

- Precompute aggregates
- Fetch last point data
- Transform table on-disk indexing and sorting
  - Like a Vertica projection
- Keep aggregates after raw input is dropped
- Create data cleaning pipelines
- Read from Kafka queues
Thank you!

We’re hiring!

Presenter: rhodges@altinity.com

ClickHouse: https://github.com/ClickHouse/ClickHouse

Altinity: https://www.altinity.com
...to reduce the amount of data we read

Rows match PARTITION BY expression
Sort columns on ORDER BY clause
Skip indexes reduce data
Sparse index selects rows

Table
Part
Part
Part