Size Matters

Best Practices for Trillion Row Datasets in ClickHouse

Robert Hodges and Altinity Engineering

10 August 2022

Let's make some introductions

Robert Hodges

Database geek with 30+ years on DBMS systems. Day job: Altinity CEO

Altinity Engineering

Database geeks with centuries of experience in DBMS and applications

Altinity

ClickHouse support and services including Altinity.Cloud
Authors of Altinity Kubernetes Operator for ClickHouse
and other open source projects

Foundations

ClickHouse is a SQL Data Warehouse

Understands SQL

Runs on bare metal to cloud

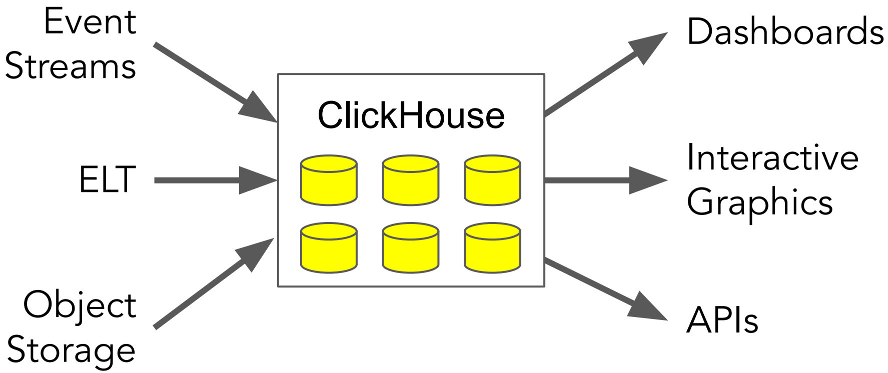
Shared nothing architecture

Stores data in columns

Parallel and vectorized execution

Scales to many petabytes

Is Open source (Apache 2.0)



It's a popular engine for real-time analytics

Seeing is believing

Demo Time!

Some definitions to guide discussion

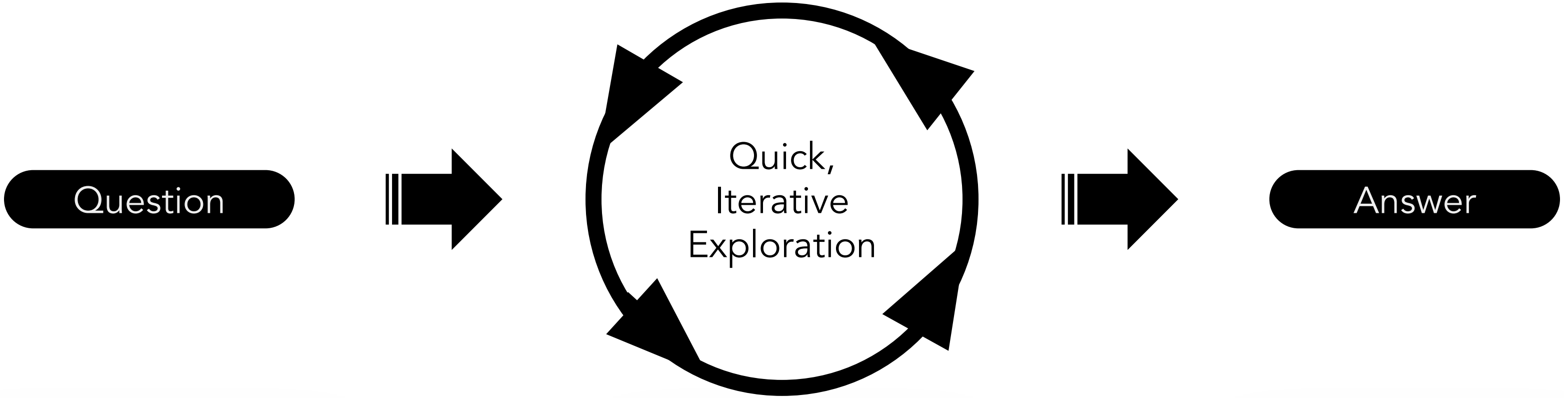
Consistent, sub-second response that scales linearly with resources

Deliver query results at costs that are low and predictable

Enabling Fast, Cost-Efficent End User Access to Trillion-Row Datasets

Market TICK data, DNS queries, weblogs, network flow logs, service logs, CDN telemetry, real-time ad bids, …

Why do we need fast access to source data?



Why do temperature sensors fail intermittently?

Slice and dice queries on detailed sensor data

Bad firmware upgrade

Principles for large datasets in ClickHouse

Reduce queries to a single scan

Reduce I/O

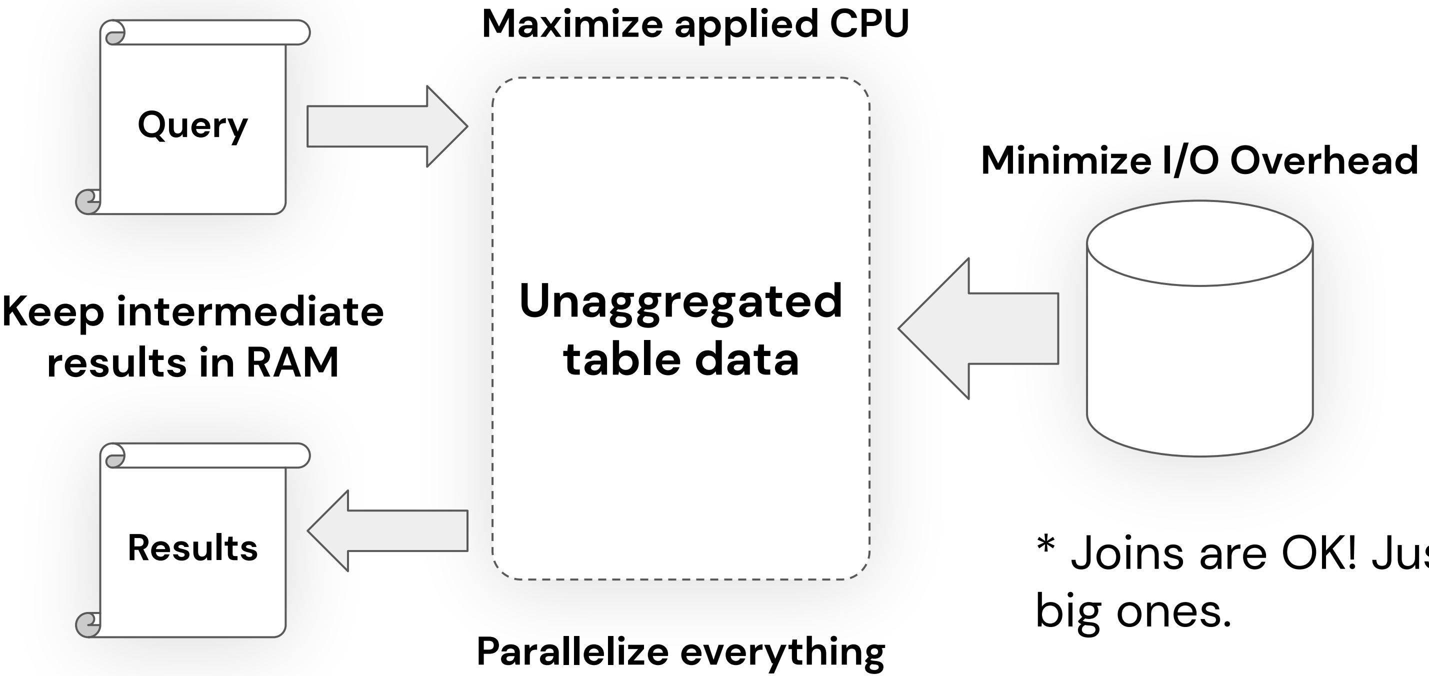
Parallelize query

Lean on aggregation (instead of joins)

Index information with materialized views

The key: One table* to rule them all

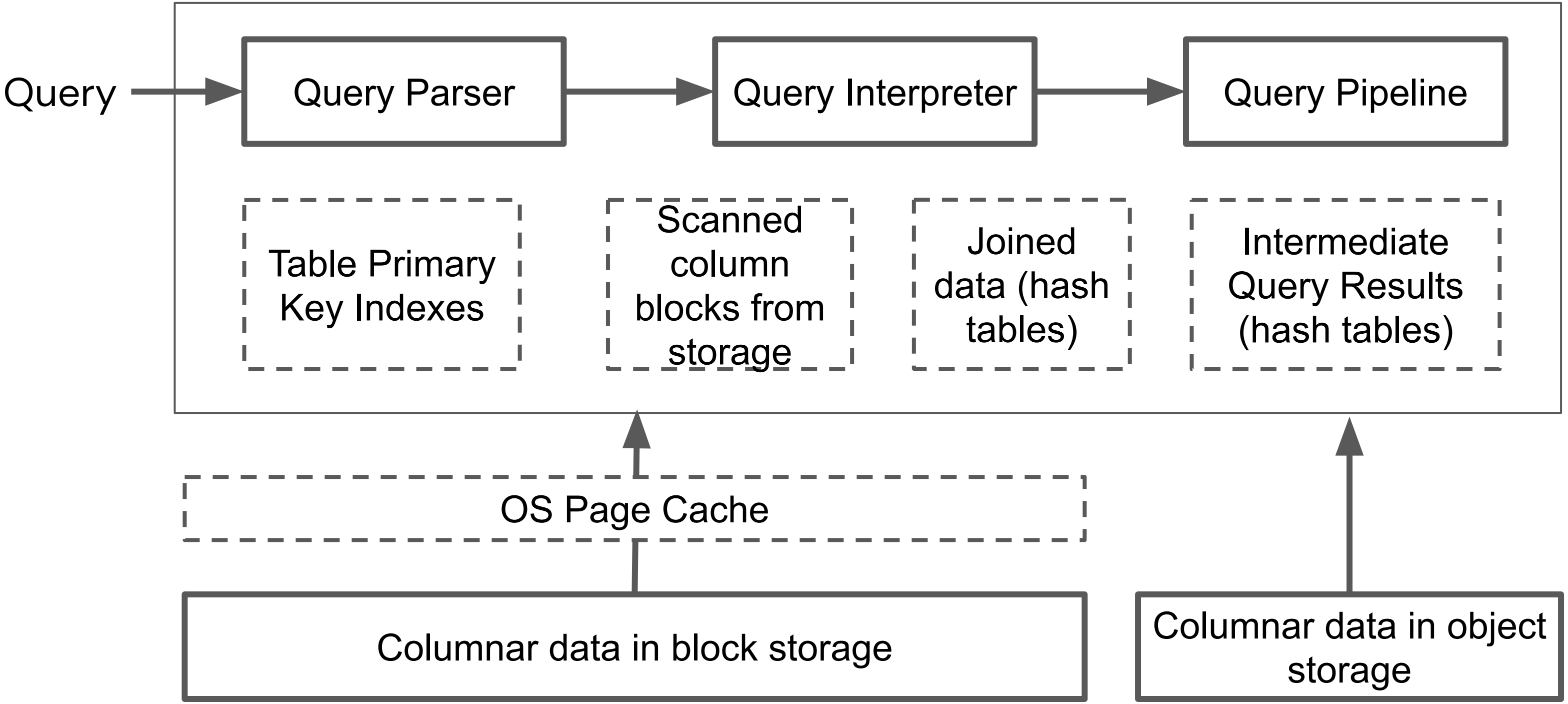
And make the scans *really fast*



* Joins are OK! Just not big ones.

Basic design for 1 trillion row tables

ClickHouse Server Architecture



Round up the usual performance suspects

Codecs

Data Types

Sharding

Read Replicas

Data Partitioning

Compression

Tiered Storage

In-RAM dictionaries

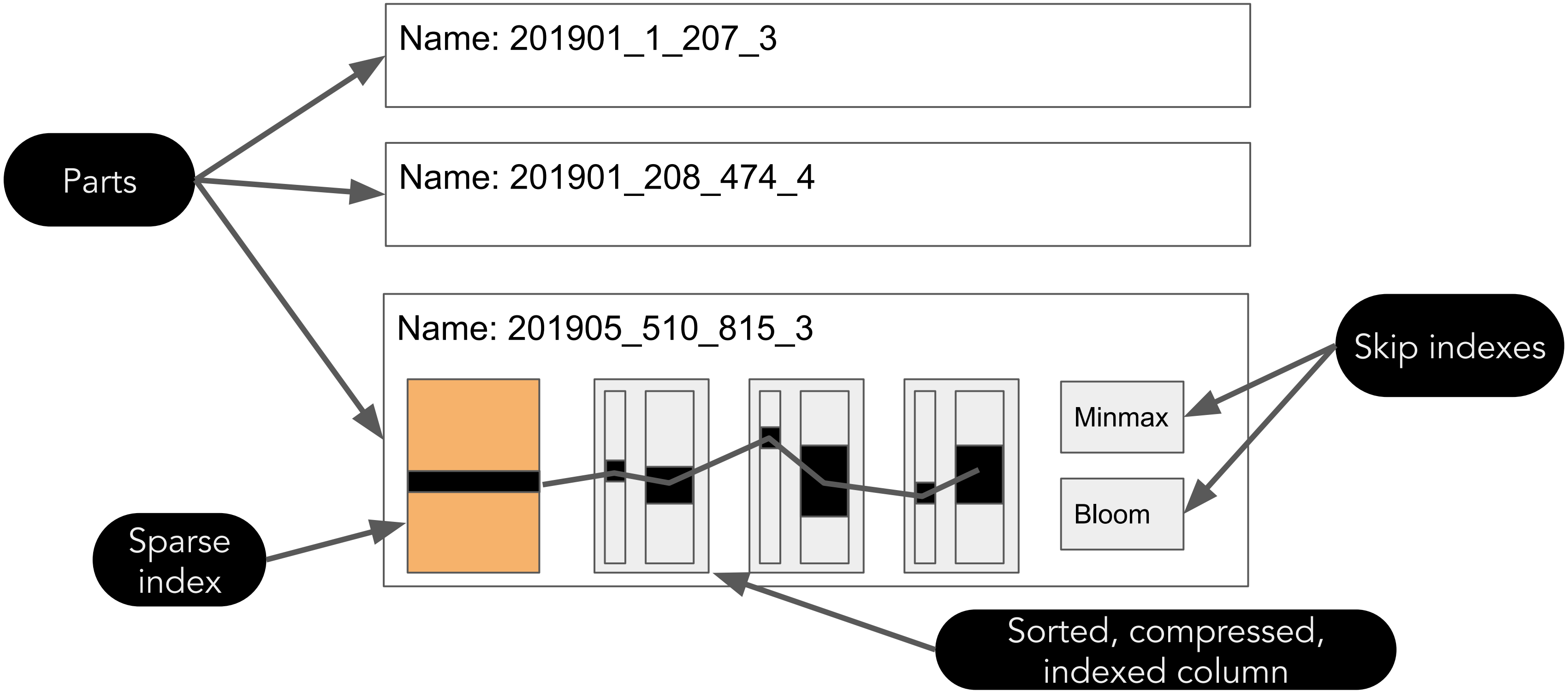
Skip Indexes

Projections

Distributed Query

Primary key index

MergeTree table organization in ClickHouse



Let’s start by making an experimental table!

```
CREATE TABLE IF NOT EXISTS readings_unopt (
    sensor_id Int64,
    sensor_type Int32,
    location String,
    time DateTime,
    date Date DEFAULT toDate(time),
    reading Float32
) Engine = MergeTree
PARTITION BY tuple()
ORDER BY tuple();
```

Sub-optimal datatypes!

No codecs!

No partitioning or ordering!

Here is a better experimental table with lower I/O cost

```
CREATE TABLE IF NOT EXISTS readings_zstd (
  sensor_id Int32 Codec(DoubleDelta, ZSTD(1)),
  sensor_type UInt16 Codec(ZSTD(1)),
  location LowCardinality(String) Codec(ZSTD(1)),
  time DateTime Codec(DoubleDelta, ZSTD(1)),
  date ALIAS toDate(time),
  temperature Decimal(5,2) Codec(T64, ZSTD(10))
)
Engine = MergeTree
PARTITION BY toYYYYMM(time)
ORDER BY (location, sensor_id, time);
```

Optimized data types

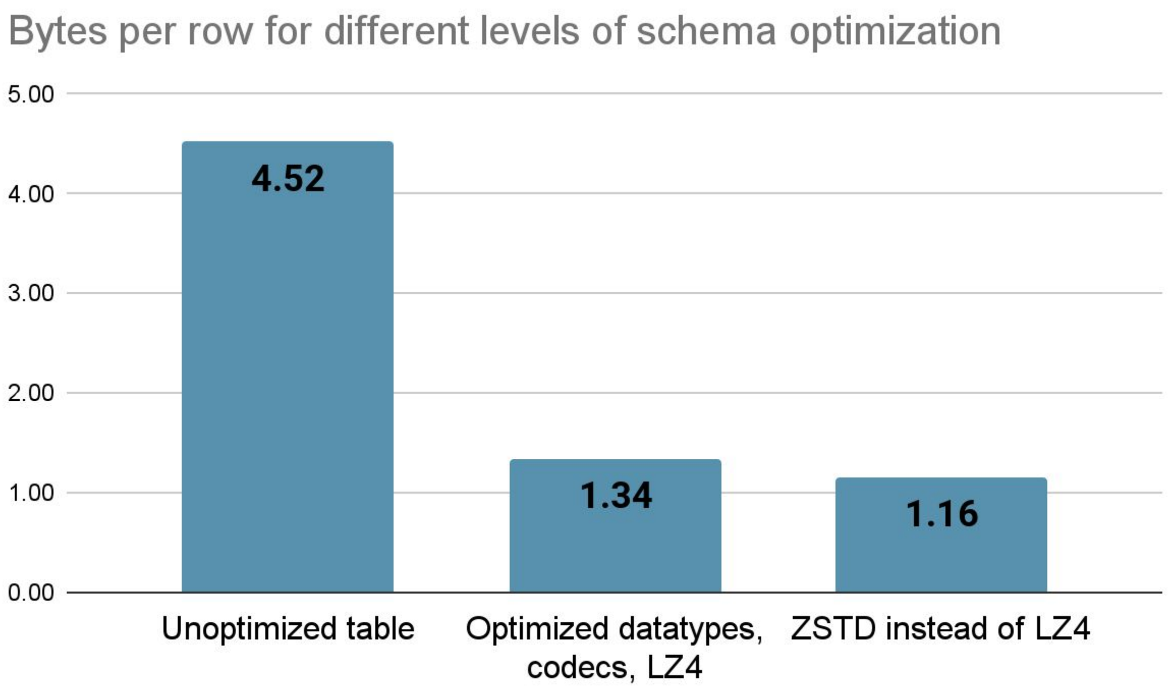
Codecs + ZSTD compression

ALIAS column

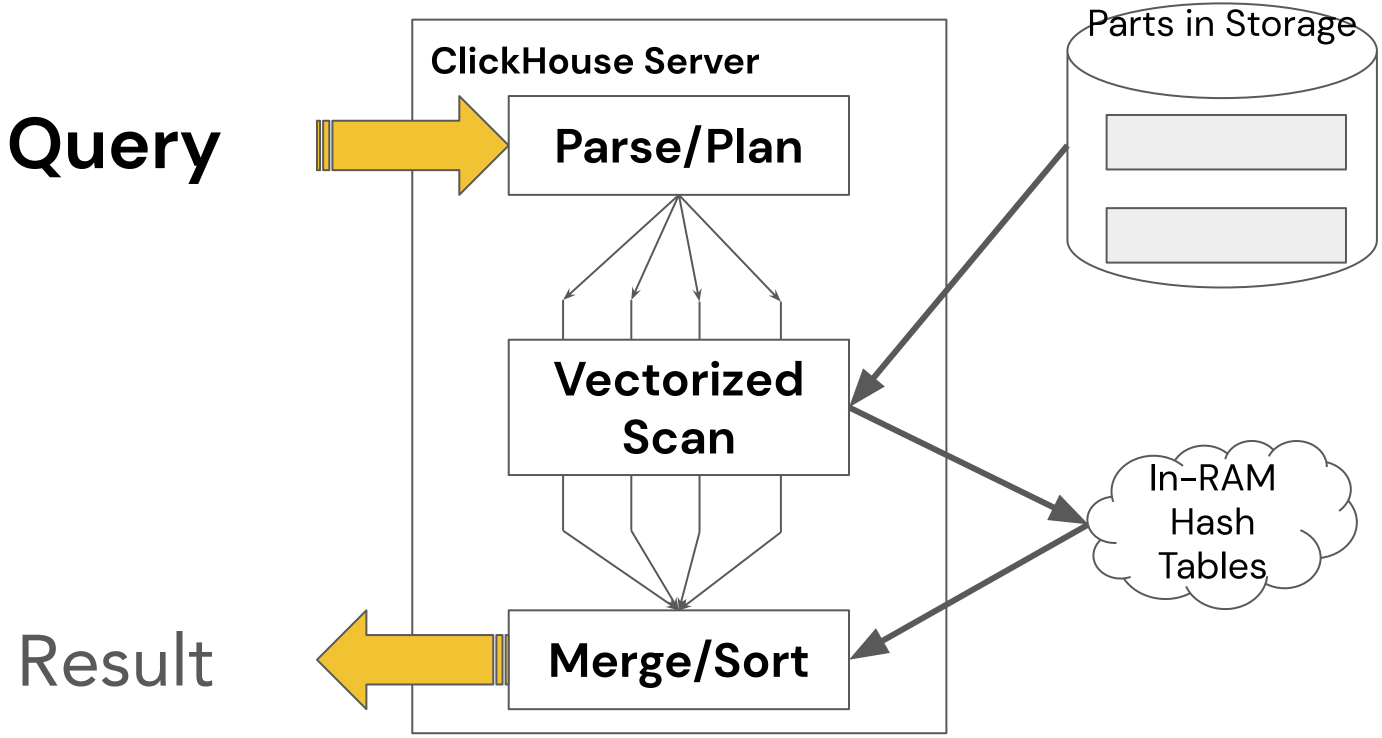
Time-based partitioning

Sorting by key columns + time

On-disk table size for different schemas

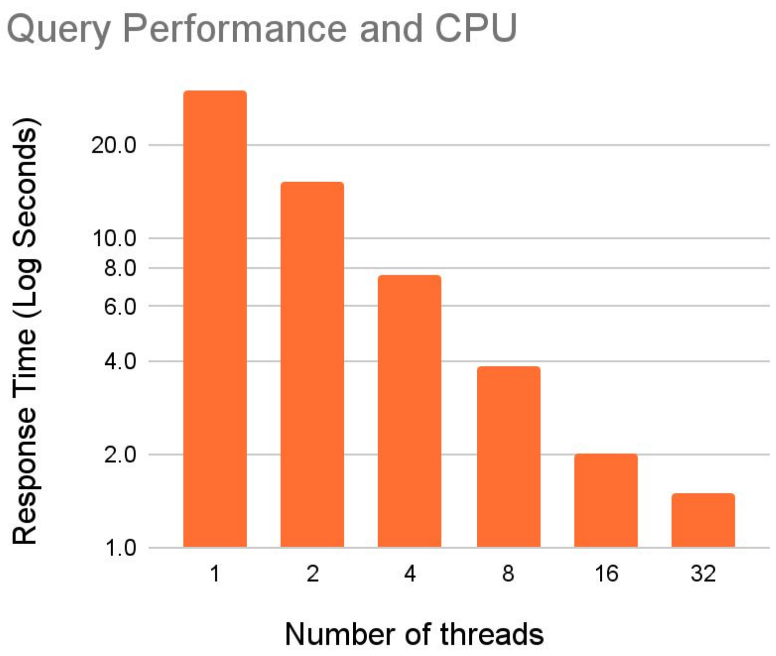


ClickHouse single node query model



Exploring linear local CPU scaling

```
-Query over 1.01 Billion rows
set max_threads = 16;
SELECT
    toYYYYMM(time) AS month,
    countIf(msg_type = 'reading') AS
readings,
    countIf(msg_type = 'restart') AS
restarts,
    min(temperature) AS min,
    round(avg(temperature)) AS avg,
    max(temperature) AS max
FROM test.readings_multi
WHERE sensor_id BETWEEN 0 and 10000
GROUP BY month ORDER BY month ASC;
```



Ingesting data into large tables

Pattern: multiple entities in a single table

Reading
- msg_type='reading'
- sensor_id
- time
- temperature

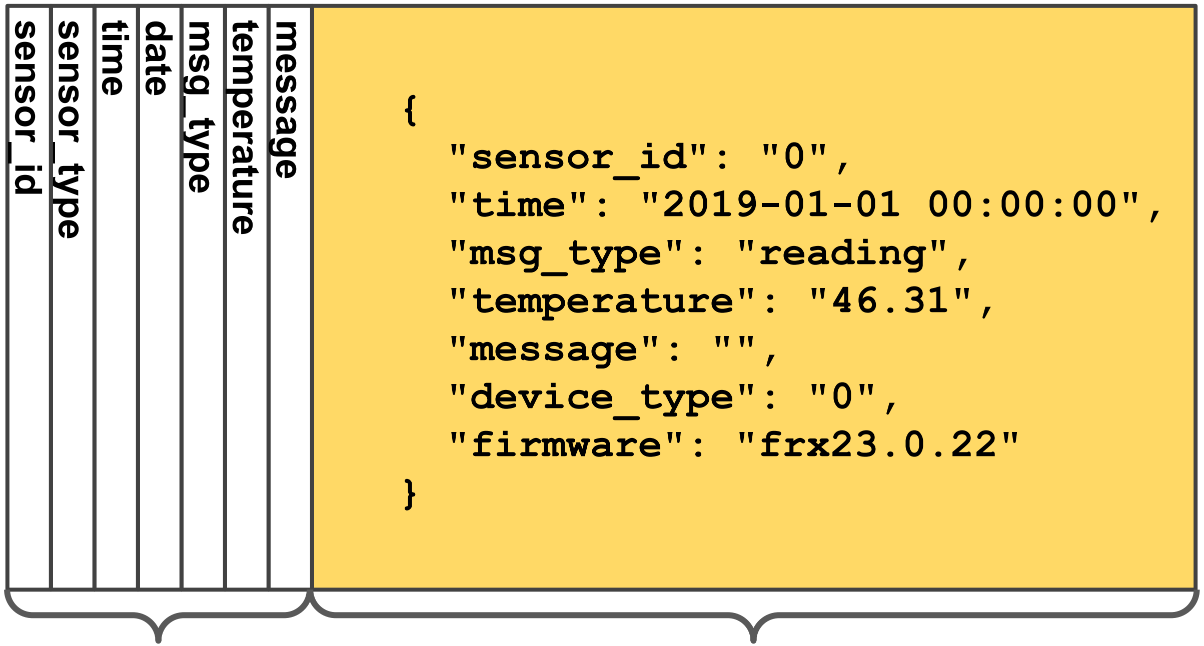
Restart
- msg_type='restart'
- sensor_id
- time

Error
- msg_type='err'
- sensor_id
- time
- message

Large table joins are an anti-pattern in low-latency apps

Many apps keep entity sources for future flexibility

Materialized columns



Source data

1.34 bytes/row **4.14 bytes/row, ~96% compression with ZSTD(1)**

Schema for a table based on multi-entity JSON

```
CREATE TABLE IF NOT EXISTS readings_multi_json (
    sensor_id Int32 Codec(DoubleDelta, LZ4),
    sensor_type UInt8,
    time DateTime Codec(DoubleDelta, LZ4),
    date ALIAS toDate(time),
    msg_type enum('reading'=1, 'restart'=2, 'err'=3),
    temperature Decimal(5,2) Codec(T64, LZ4),
    message String DEFAULT '',
    json String DEFAULT ''
) Engine = MergeTree
PARTITION BY toYYYYMM(time)
ORDER BY (msg_type, sensor_id, time);
```

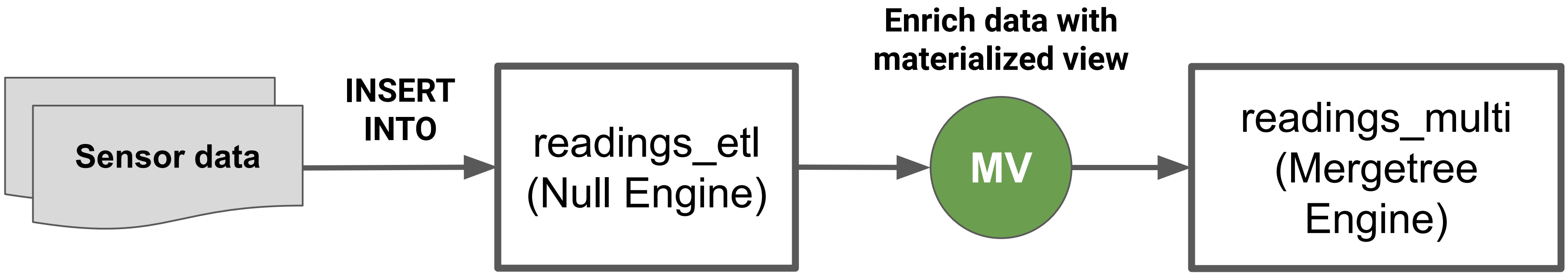
Codecs + LZ4 compression

Discriminator column

String column for JSON data

Sort by msg_type, sensor, time

Loading raw data into large systems



ClickHouse makes it easy to materialize columns

```
ALTER TABLE readings_multi_json
  ADD COLUMN IF NOT EXISTS firmware String
    DEFAULT JSONExtractString(json, 'firmware')
;

ALTER TABLE readings_multi_json
  UPDATE firmware = firmware WHERE 1=1
;
```

…But you can also index and query JSON directly

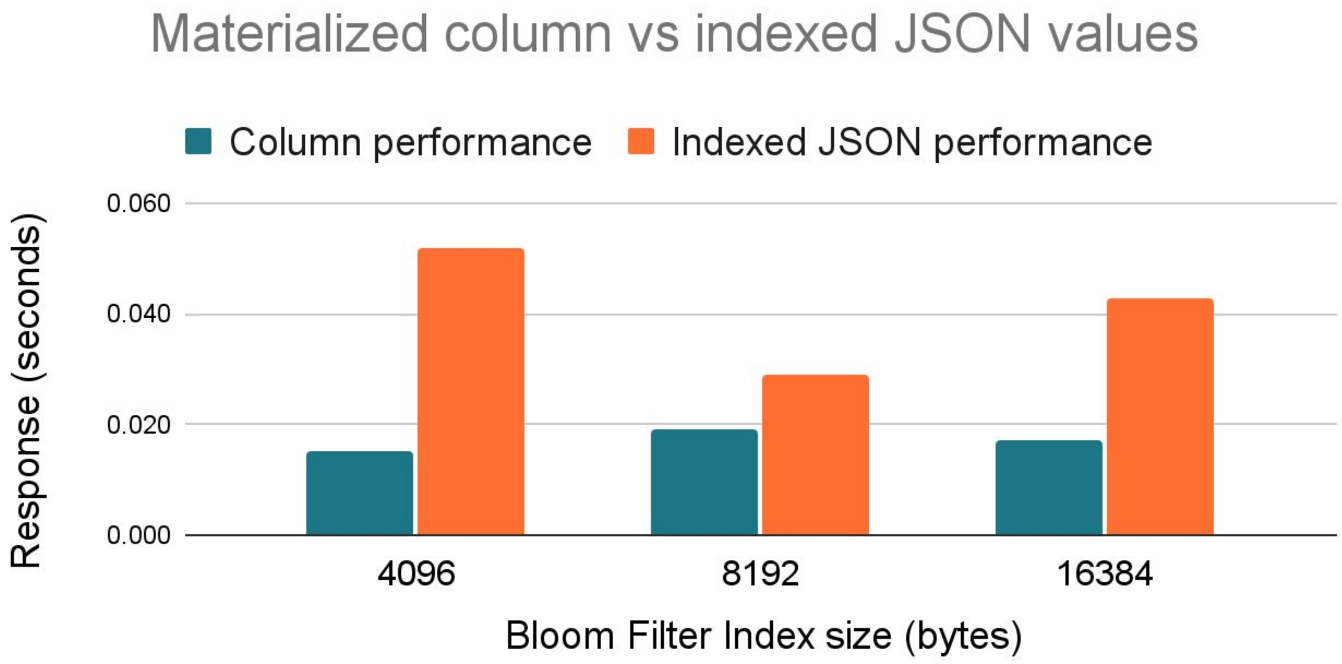
```
ALTER TABLE readings_multi_json
   ADD INDEX jsonbf json TYPE tokenbf_v1(8192, 3, 0)
   GRANULARITY 1;

ALTER TABLE readings_multi_json
    MATERIALIZE INDEX jsonbf;
```

Bloom filter tuning is complicated!

```
-- Count matches on column.
SELECT count()
FROM readings_multi_json
WHERE
  firmware = 'frx23ID0000x2532'
```

```
-- Count token matches in JSON.
SELECT count()
FROM readings_multi_json
WHERE
  hasToken(json,'frx23ID0000x2532')
```

Results are good if you have high cardinality values



Unique ClickHouse tricks for large datasets

How can we make queries fast on large data sets?

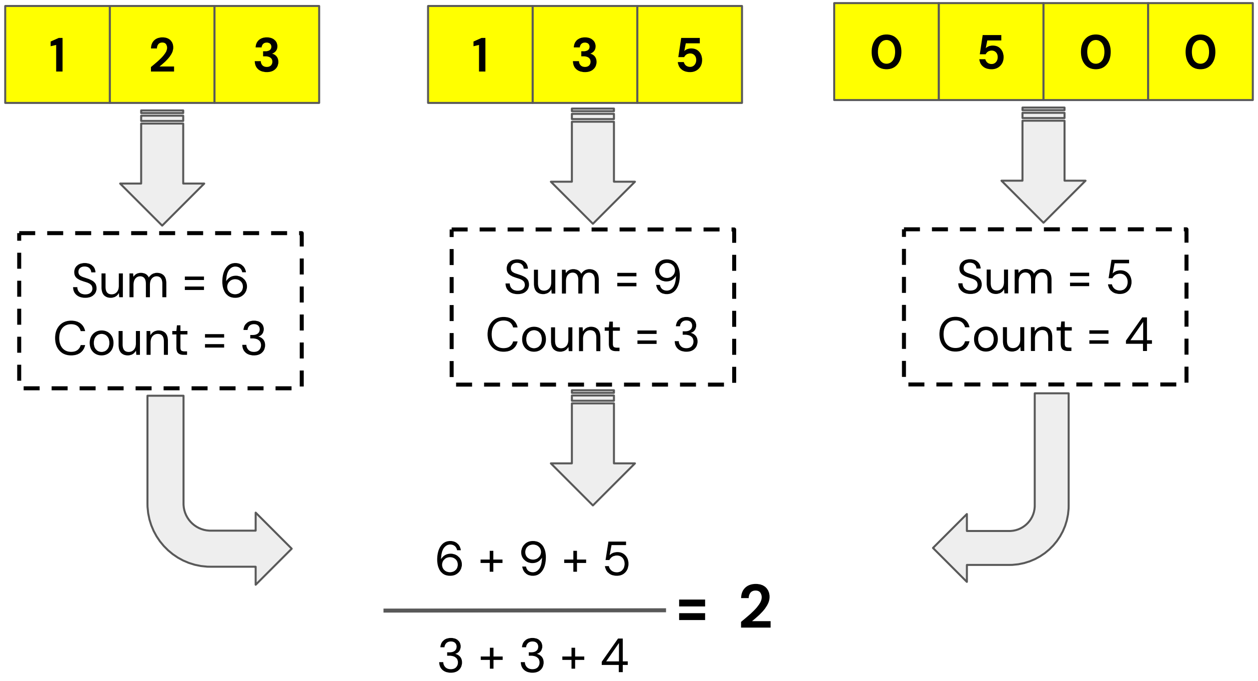
Create queries that work in a single scan without large-table joins

Hint: Aggregation runs in a single pass

No need to move data

Parallelizes!

Intermediate results are reusable



What about queries over all entities?

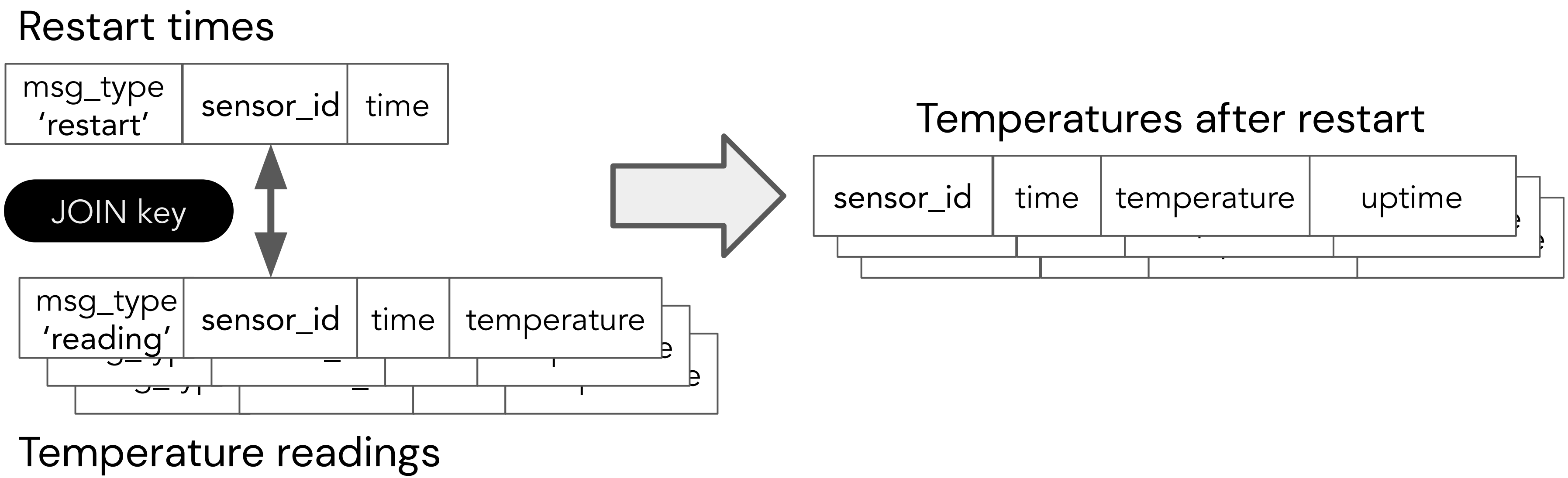
Use conditional aggregation!

```
SELECT toYYYYMM(time) AS month,
    countIf(msg_type = 'reading') AS readings,
    countIf(msg_type = 'restart') AS restarts,
    min(temperature) AS min,
    round(avg(temperature)) AS avg, max(temperature) AS max
FROM test.readings_multi WHERE sensor_id = 3
GROUP BY month ORDER BY month ASC
```

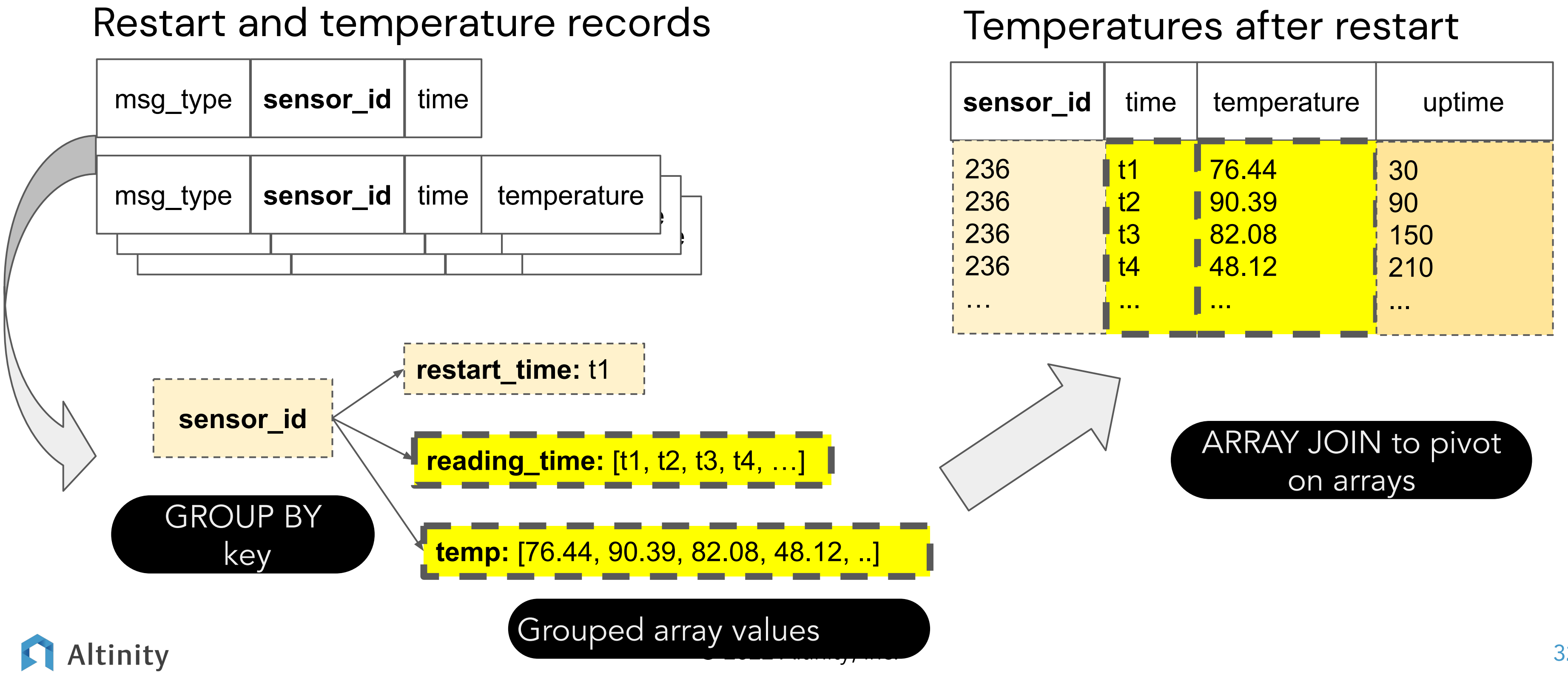
month	readings	restarts	min	avg	max
201901	44640	1	0	75	118.33
201902	40320	0	68.09	81	93.98
201903	15840	0	73.19	84	95.3

What about joins on distributed data?

Use case: join restarts with temperature readings



Aggregation can implement joins!



And here's the code…

```
SELECT sensor_id, reading_time, temp, reading_time,
  reading_time - restart_time AS uptime
FROM (
WITH toDateTime('2019-04-17 11:00:00') as start_of_range
SELECT sensor_id, groupArrayIf(time, msg_type = 'reading') AS
reading_time,
    groupArrayIf(temperature, msg_type = 'reading') AS temp,
    anyIf(time, msg_type = 'restart') AS restart_time
FROM test.readings_multi rm
WHERE (sensor_id = 2555)
  AND time BETWEEN start_of_range AND start_of_range + 600
GROUP BY sensor_id)
ARRAY JOIN reading_time, temp
```

Not everyone's cup of tea, but it works!!!

How about locating key events in tables?

When was the last restart on sensor 236?

```
SELECT message
FROM readings_multi
WHERE (msg_type, sensor_id, time) IN
   (SELECT msg_type, sensor_id, max(time)
    FROM readings_multi
    WHERE msg_type = 'restart'
      AND sensor_id = 236
    GROUP BY msg_type, sensor_id)
```

Expensive on large datasets!

Finding the last restart is an aggregation task!

sensor_id	time	msg_type
236	2019-01-10 20:00:13	restart

sensor_id	time	msg_type
236	2019-01-10 21:07:56	restart

Merge

GROUP BY key

sensor_id	time	msg_type
236	2019-01-10 21:07:56	restart

Max value

Matching row value

Use materialized views to “index” data

MergeTree Table

Block lands in source table

“Last point query”

```
SELECT
   sensor_id,
   max(time) AS time
FROM readings_multi
WHERE msg_type = 'restart'
GROUP BY sensor_id
```

Block(s) land in materialized view target table

AggregatingMergeTree Table

Finding the last restart on a sensor

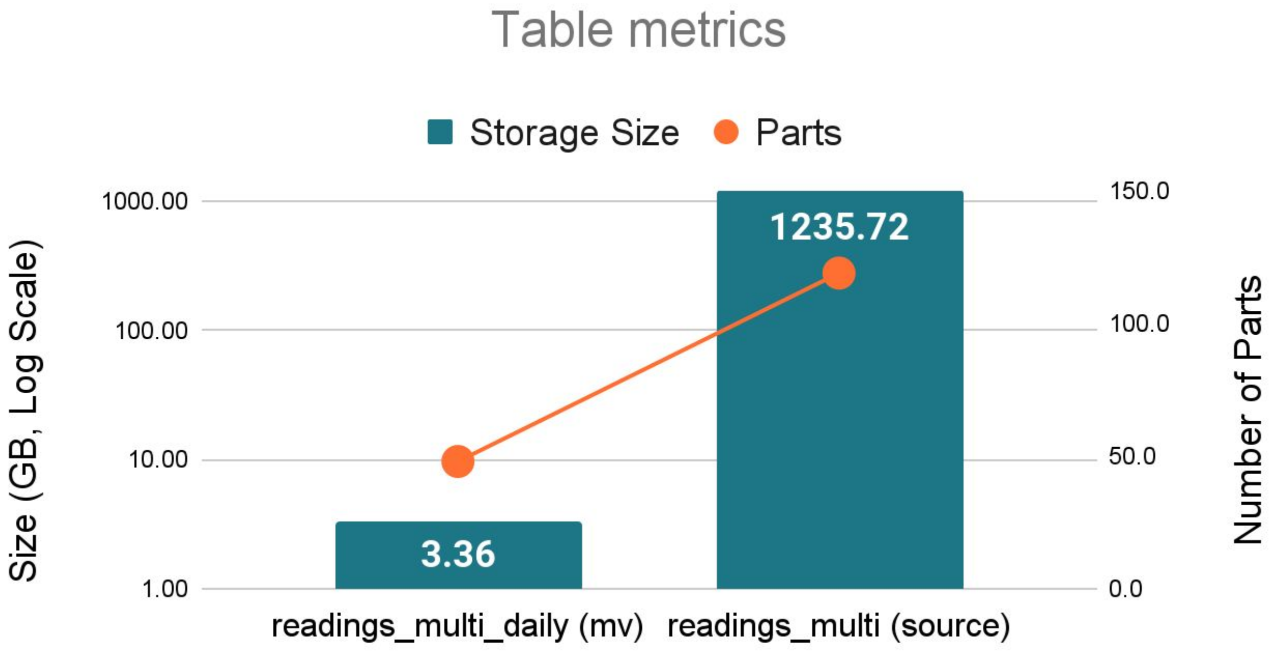
And here's code for the materialized view…

```
CREATE TABLE sensor_last_restart_agg (
  sensor_id Int32,
  time SimpleAggregateFunction(max, DateTime),
  msg_type AggregateFunction(argMax, String, DateTime)
)
ENGINE = AggregatingMergeTree()
PARTITION BY tuple() ORDER BY sensor_id

CREATE MATERIALIZED VIEW sensor_last_restart
TO sensor_last_restart_agg AS SELECT
  sensor_id, max(time) AS time,
  argMaxState(msg_type, time) AS msg_type
FROM readings_multi
WHERE msg_type = 'restart' GROUP BY sensor_id
```

SimpleAggregateFunction simplifies insert and query

tuple() is a dubious choice!

Comparison of source table to typical materialized view



Wrap-up

Learnings from large ClickHouse installations

Use a single large table to hold all entities

Make sound implementation choices to get baseline performance

Include source data for future flexibility without moving data

Aggregation is a secret ClickHouse power: use it to scan, join, index data

Your reward: Linear scaling, high cost-efficiency, and happy users

Other important techniques for big data

Sharding and replication

Tiered storage

Object storage

Approximate queries using sampling and approximate uniqs (lighter aggregation)

Where is the documentation?

ClickHouse official docs – https://clickhouse.com/docs/

Altinity Blog – https://altinity.com/blog/

Altinity Youtube Channel – https://www.youtube.com/channel/UCE3Y2lDKl_ZfjaCrh62onYA

Altinity Knowledge Base – https://kb.altinity.com/

Meetups, other blogs, and external resources. Use your powers of Search!

Thank you!
Questions?

https://altinity.com

Altinity.Cloud

Altinity Support

Altinity Stable Builds

We're hiring!