DATA+AI
SUMMIT 2022

Opening the Floodgates

Enabling Fast, Unmediated End User Access to Trillion-Row Datasets

Robert Hodges
CEO, Altinity

ORGANIZED BY databricks

Basic information about me

- Career focus: Databases from pre-relational to SQL to Analytic DBMS
- Most recent: ClickHouse, a SQL data warehouse released under Apache 2.0 license
- Academic background: Classics, History, and Languages (Human)
- Day job: CEO of Altinity, an enterprise provider for ClickHouse

Framing the 1 trillion row problem

Seeing is believing

Demo Time!

Some definitions to guide discussion

Consistent, sub-second response that scales linearly with resources

Direct access to source data

Enabling **Fast**, **Unmediated** 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?

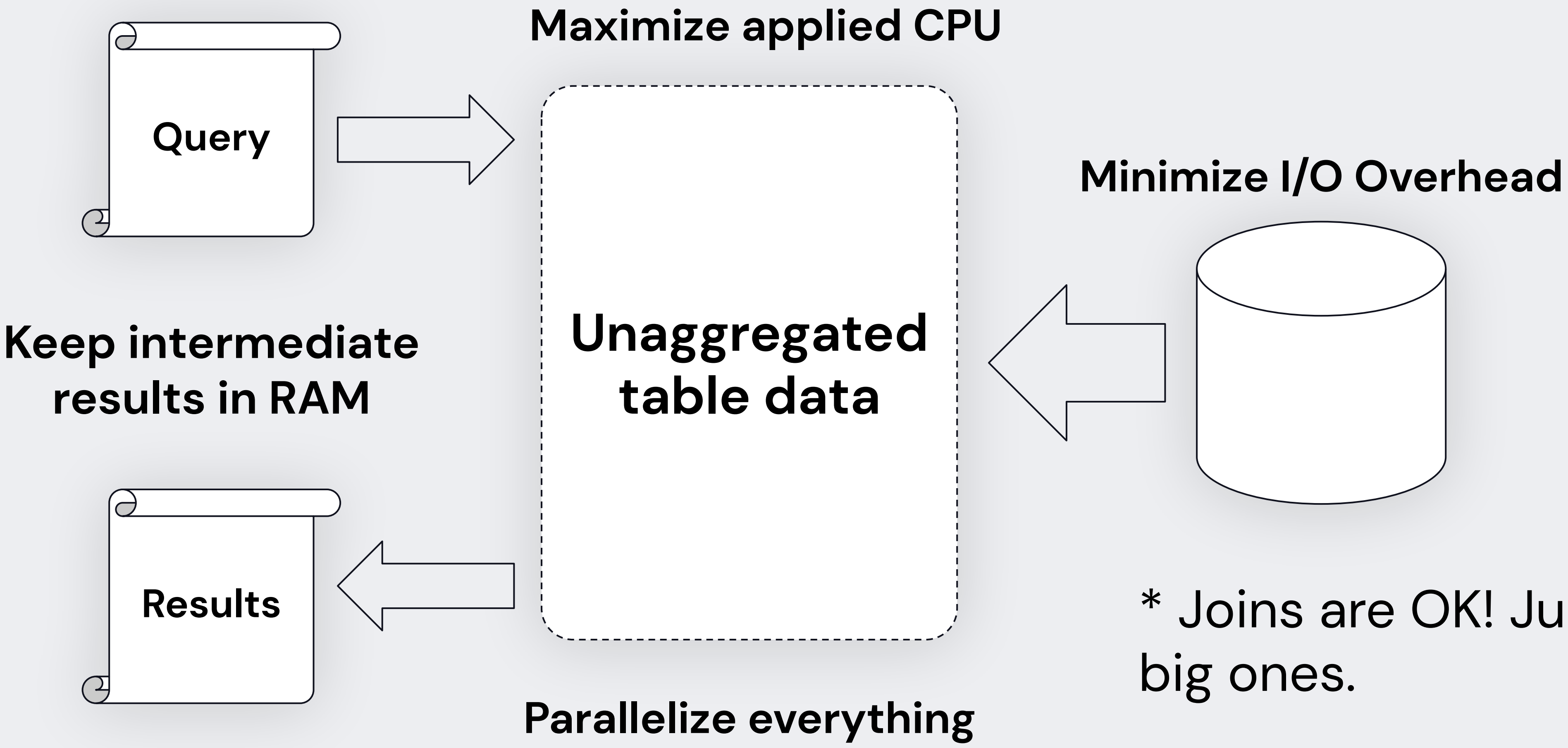
Why do temperature sensors fail intermittently?

Slice and dice queries on detailed sensor data

Bad firmware upgrade

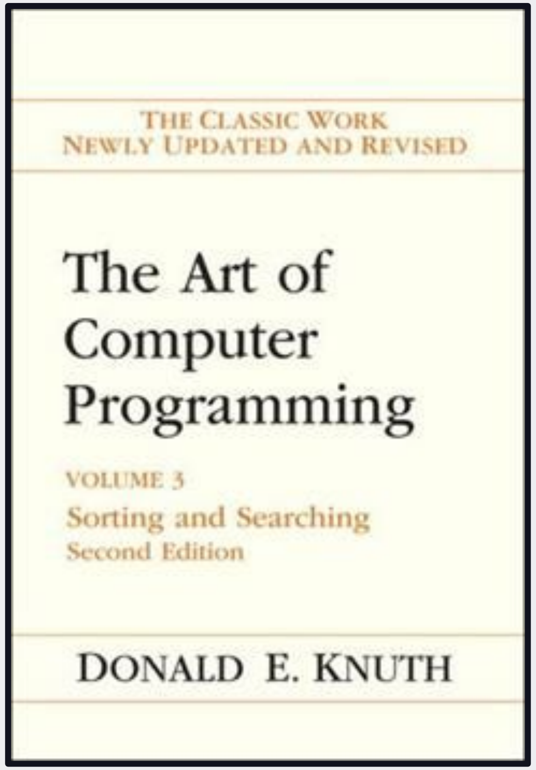
The solution: One table* to rule them all

And make the scans *really fast*



* Joins are OK! Just not big ones.

Read this before asking for your money back



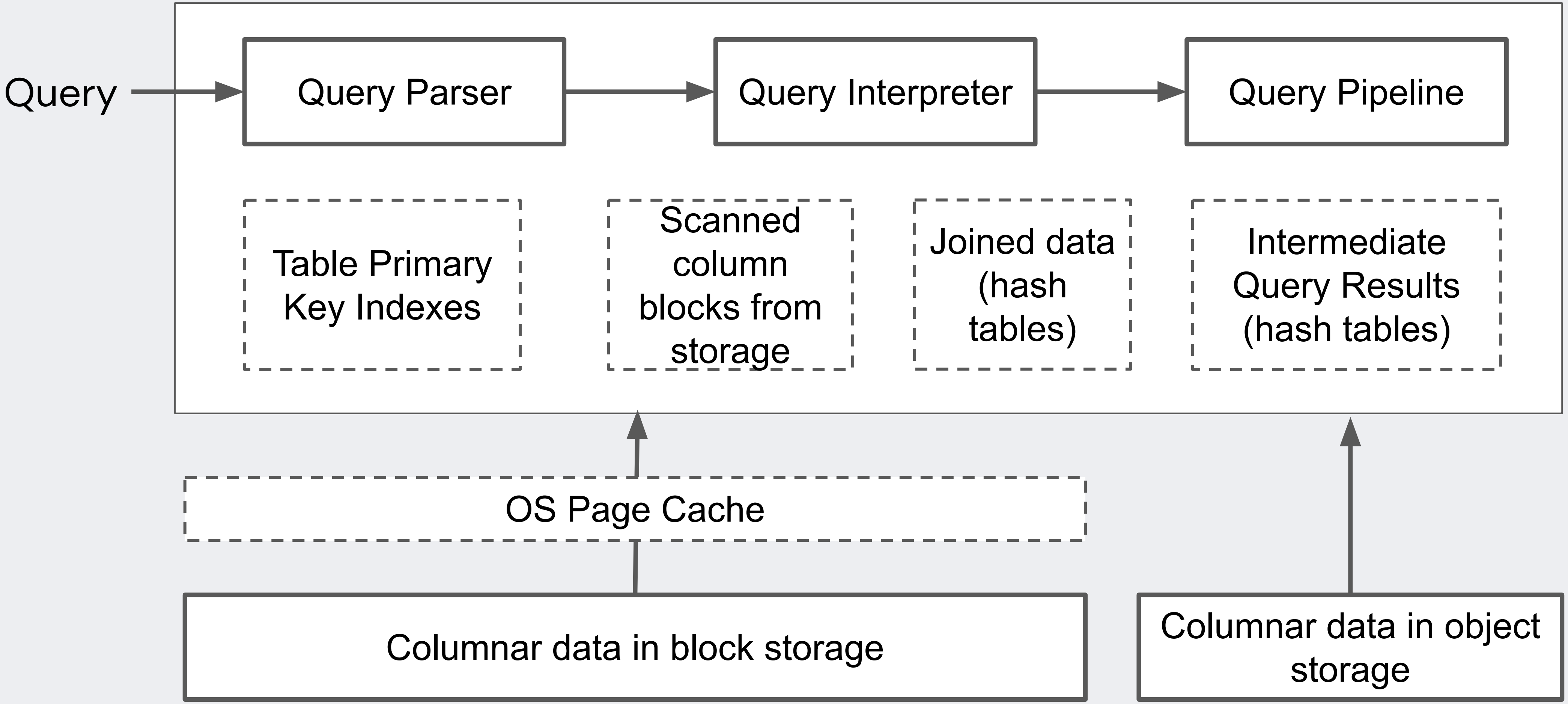
"BEGIN AT THE BEGINNING, and go on until you find the right key; then stop. . . We shall see that sequential searching involves some very interesting ideas, in spite of its simplicity.

AOCP, Volume 3, Page 396

Blocking and tackling

Storing and querying 1 trillion rows

ClickHouse Server Architecture



Round up the usual performance suspects

Codecs

Data Types

Sharding

Read Replicas

Data Partitioning

Compression

Tiered Storage

In-RAM dictionaries

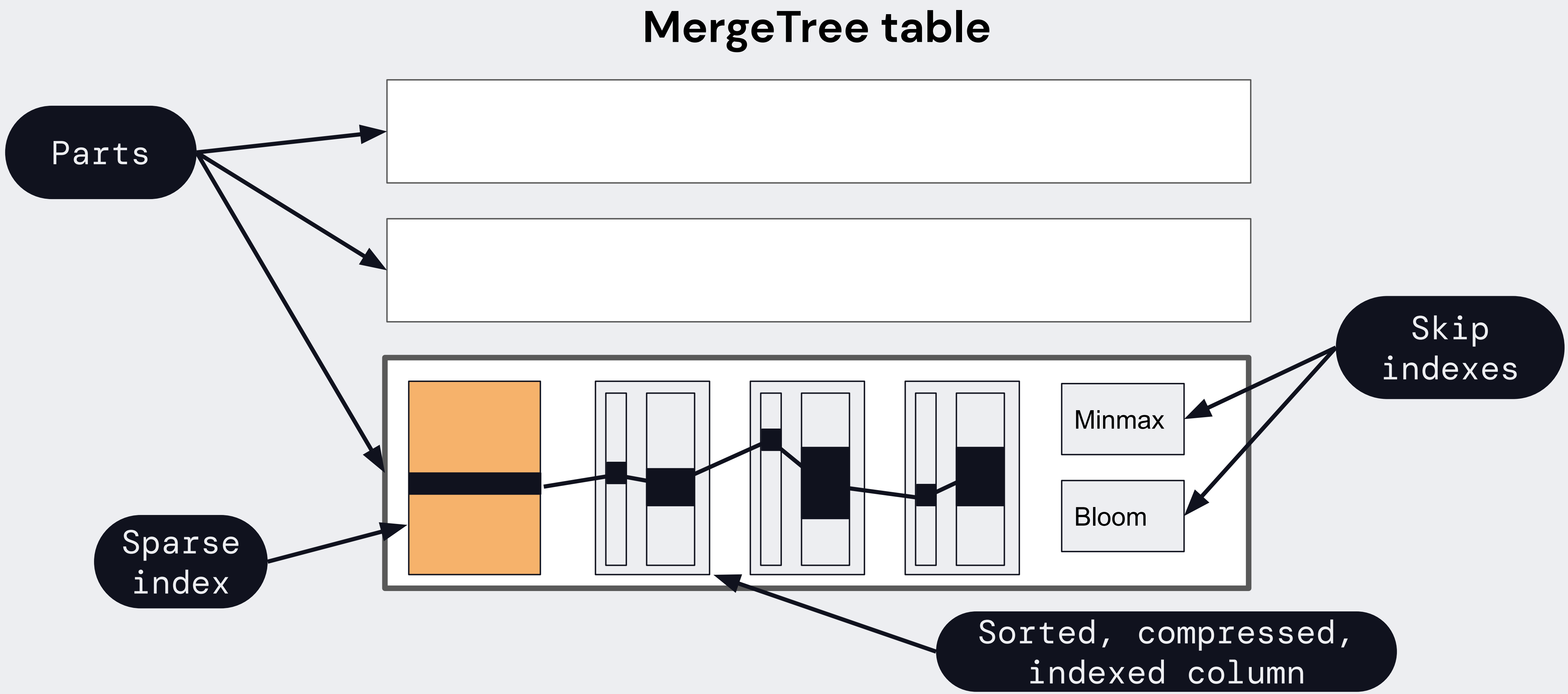
Skip Indexes

Projections

Distributed Query

Primary key index

Table organization in ClickHouse



Let’s start by making a 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 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);
```

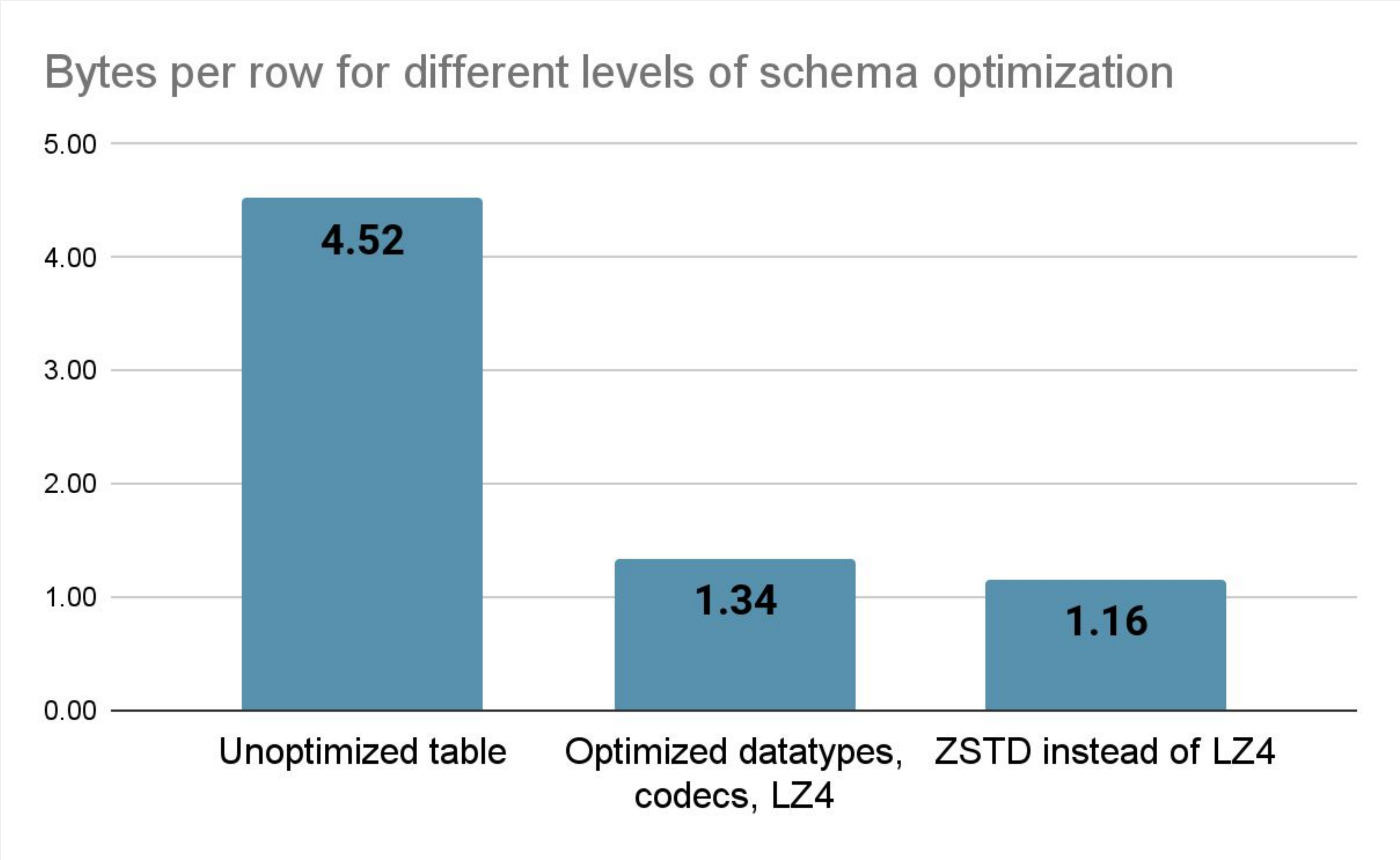
Optimized data types

Codecs + ZSTD compression

Time-based partitioning

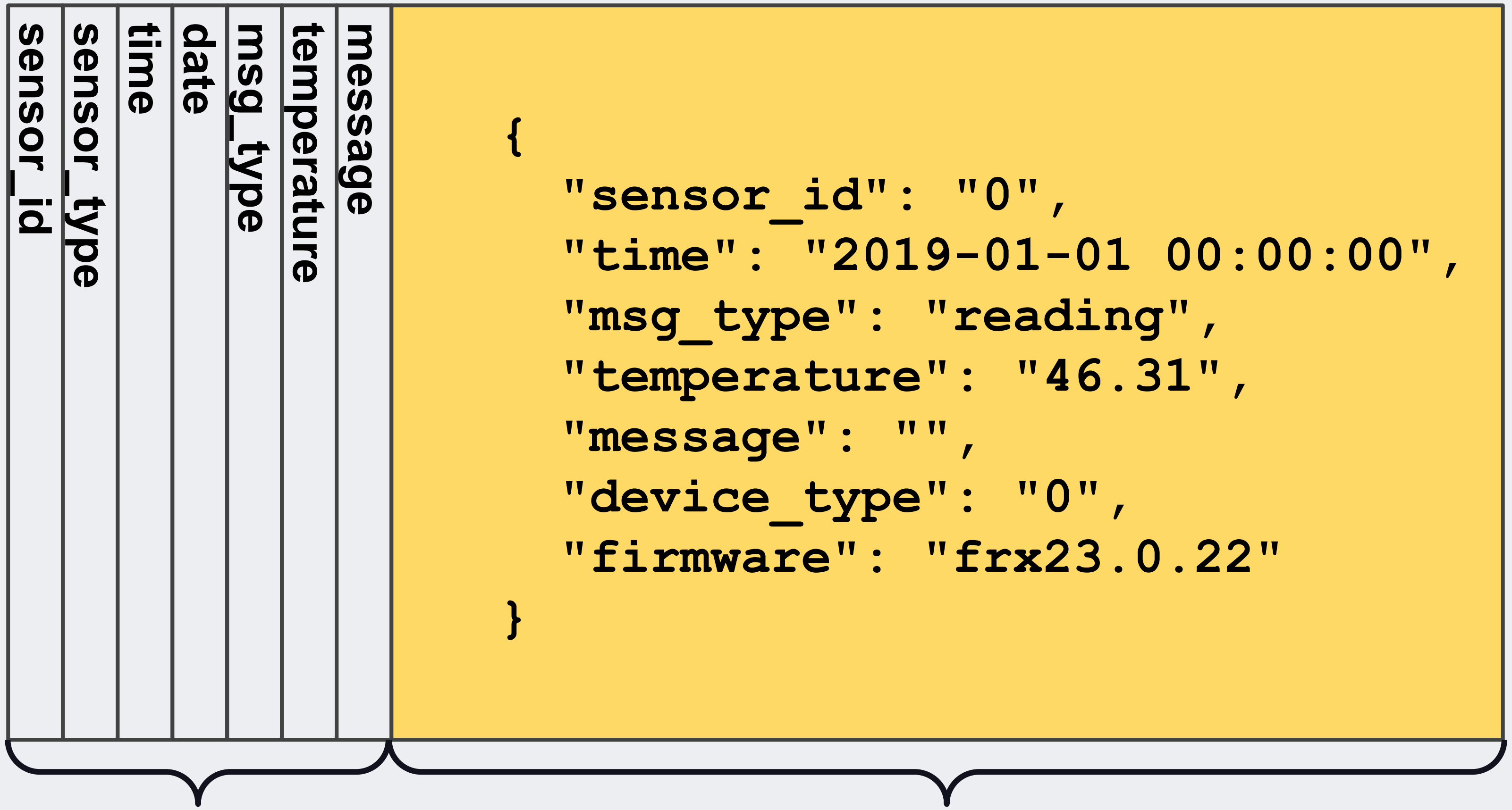
Sorting by key columns + time

On-disk table size for different schemas



Many apps keep entity sources in the row

Materialized columns

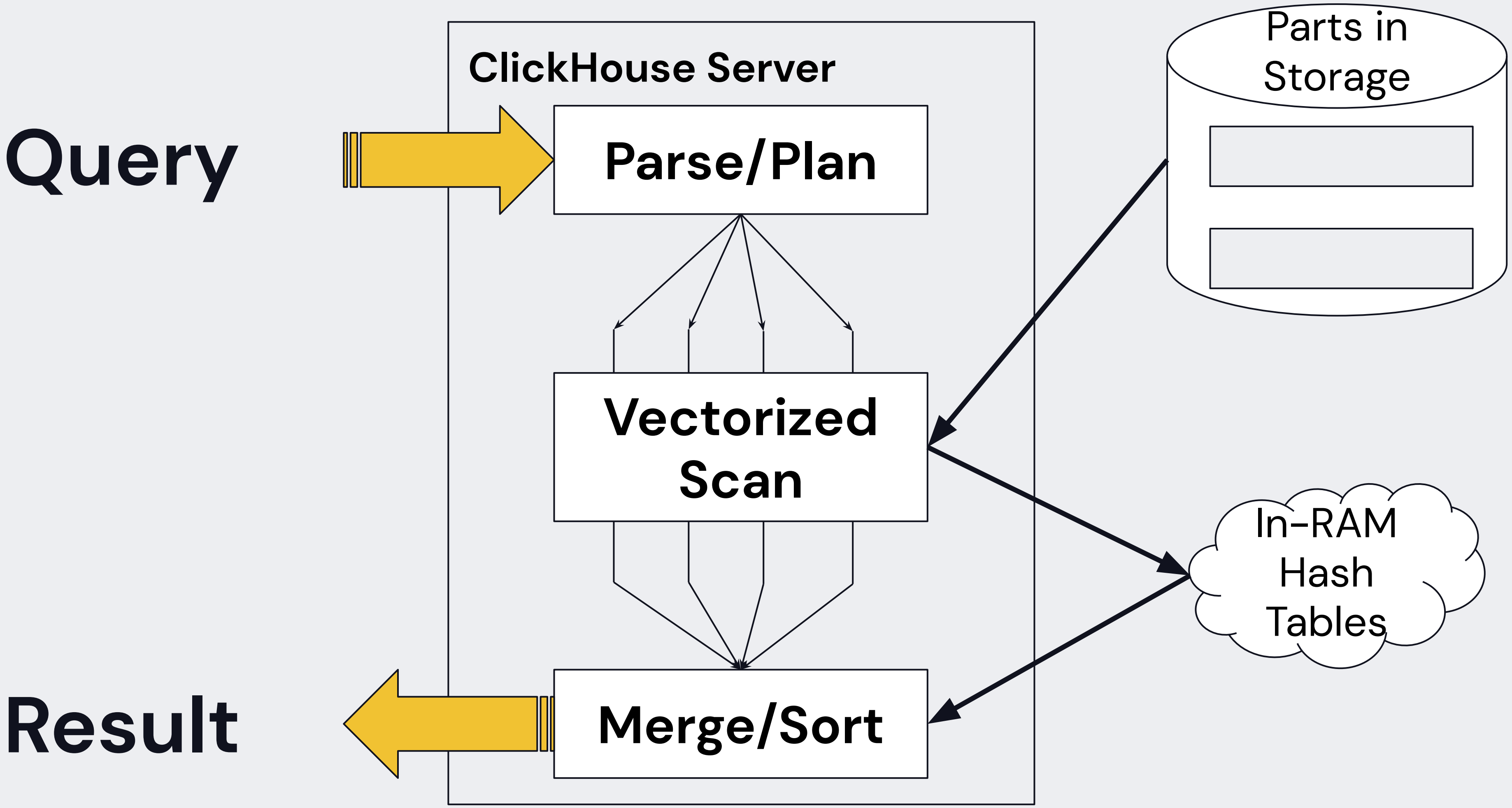


Source data

1.34 bytes/row

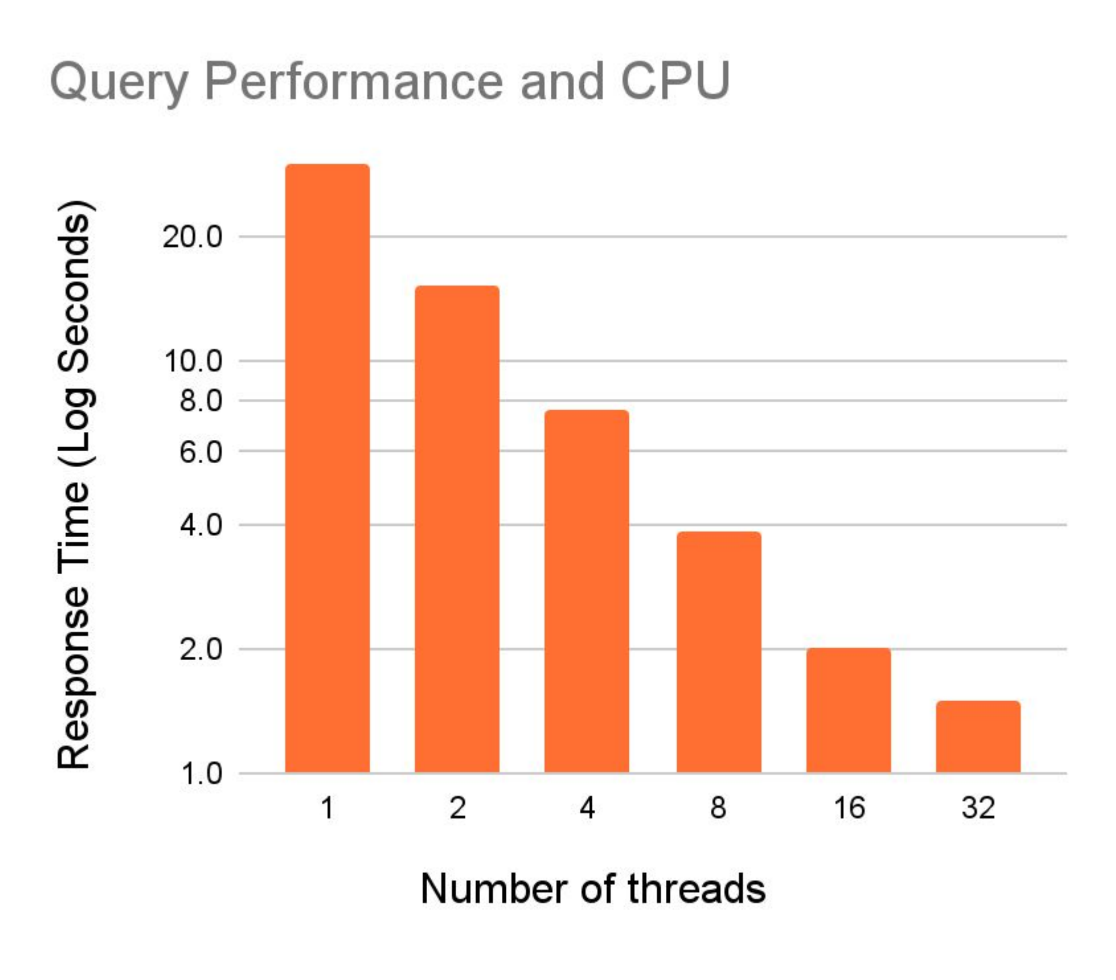
4.14 bytes/ row
~96% compression with ZSTD(1)

ClickHouse single node query model



Demonstration of 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;
```



Adding razzle-dazzle

Unique tricks for large datasets

Pattern: multiple entities in a single table

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

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

Restart
- msg_type='restart'
- sensor_id
- time

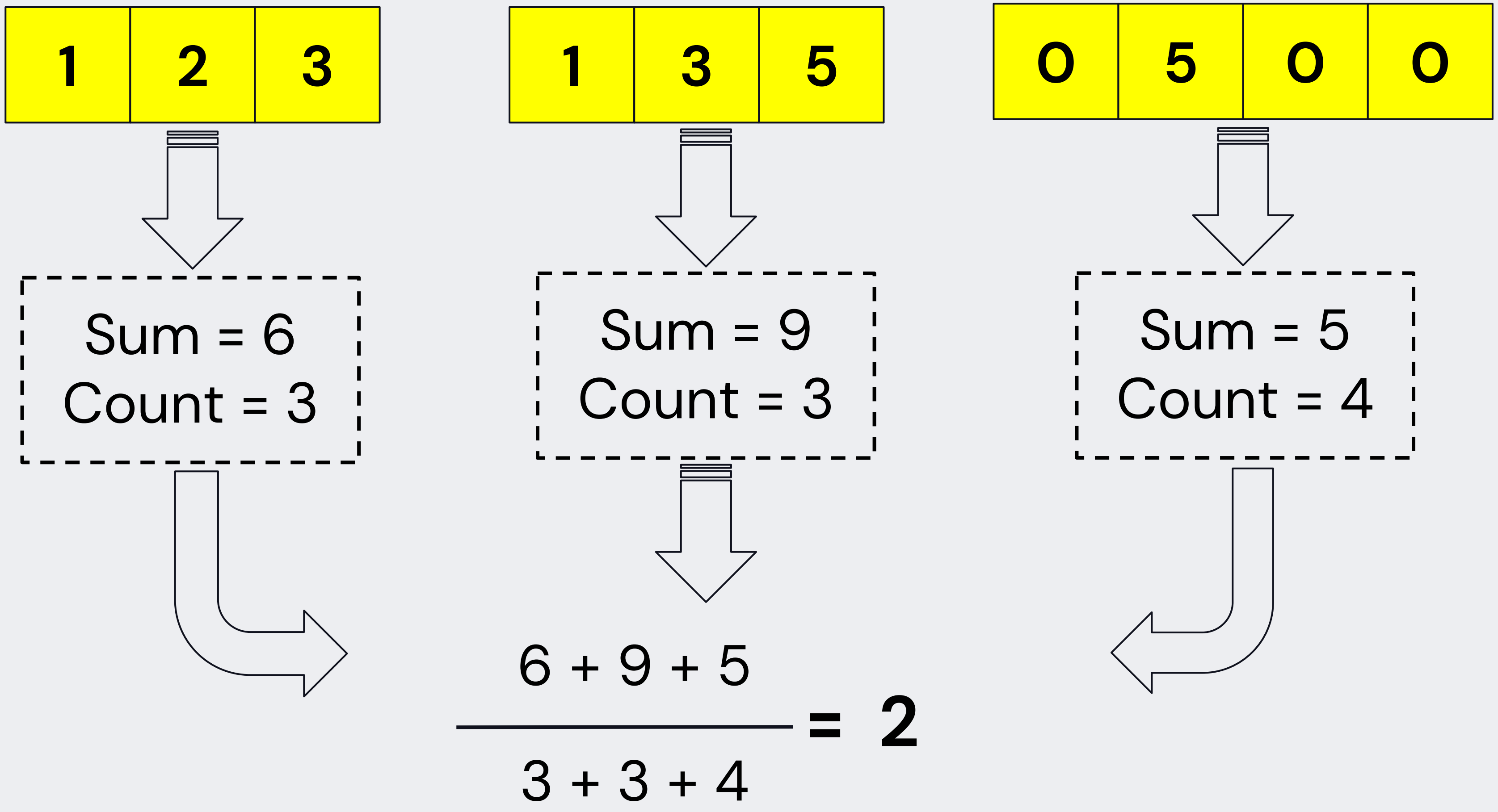
Error
- msg_type='err'
- sensor_id
- time
- message

Aggregation is the key technique to scale

No need to move data

Parallelizes!

Intermediate results are reusable



What about queries over all entities?

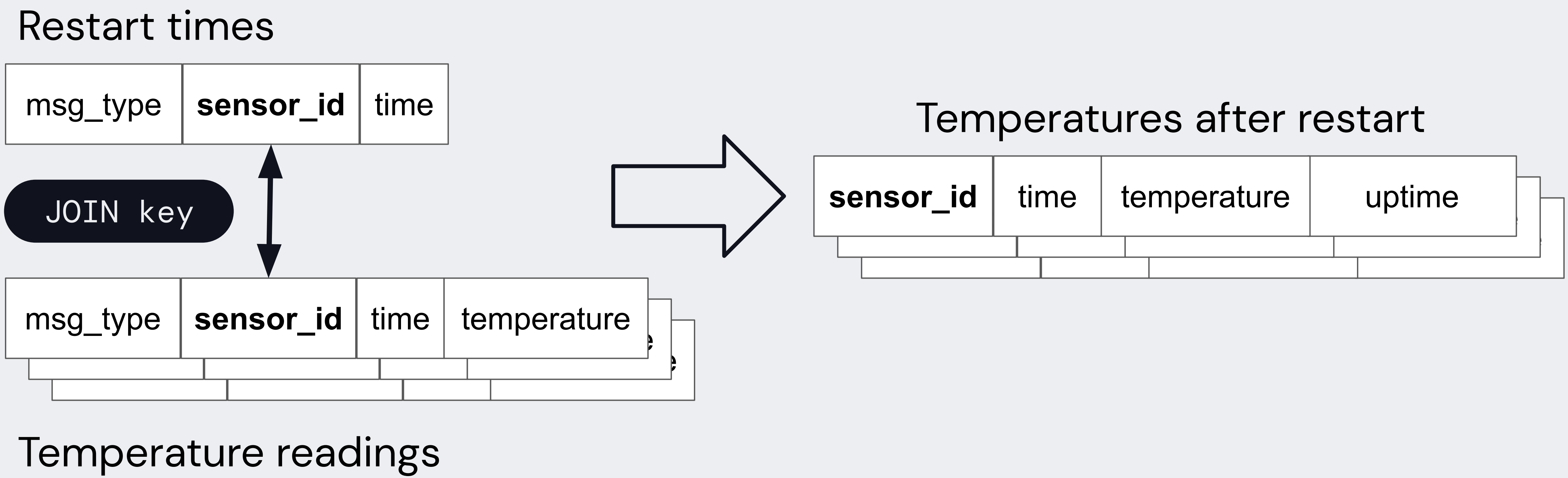
Use conditional aggregation to cover multiple types

```
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!

Restart and temperature records

msg_type	sensor_id	time

msg_type	sensor_id	time	temperature

GROUP BY key

sensor_id

restart_time: t1

reading_time: [t1, t2, t3, t4, …]

temp: [76.44, 90.39, 82.08, 48.12, ..]

Grouped array values

Temperatures after restart

sensor_id	time	temperature	uptime
236	t1	76.44	30
236	t2	90.39	90
236	t3	82.08	150
236	t4	48.12	210
…	...	...	...

ARRAY JOIN to pivot on arrays

And here's the code!

(Possibly not for everyone, but it works.)

```
SELECT sensor_id, reading_time, temp, 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
```

How about locating key events in tables?

What was the last error 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 = 'err'
     AND sensor_id = 236
   GROUP BY msg_type, sensor_id)
```

Expensive on large datasets!

Finding the last error is an aggregation task!

sensor_id	time	err
236	2019-01-10 20:00:13	Segfault

sensor_id	time	err
236	2019-01-10 21:07:56	OOM

Merge

`GROUP BY key`

sensor_id	time	err
236	2019-01-10 21:07:56	OOM

`Max value`

`Matching row value`

Use materialized views to “index” data

Finding the last error on a sensor

MergeTree Table

Block lands in source table

“Last point query”

```
SELECT
  sensor_id,
  max(time) AS last_time,
  argMaxState(message, time)
    AS last_message
FROM readings_multi rm
WHERE msg_type = 'err'
GROUP BY sensor_id
```

Block(s) land in materialized view target table

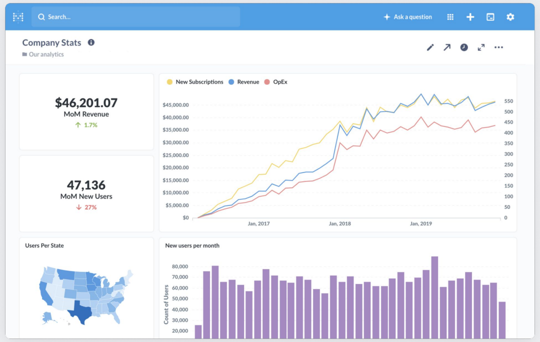
AggregatingMergeTree Table

Opening up the gate

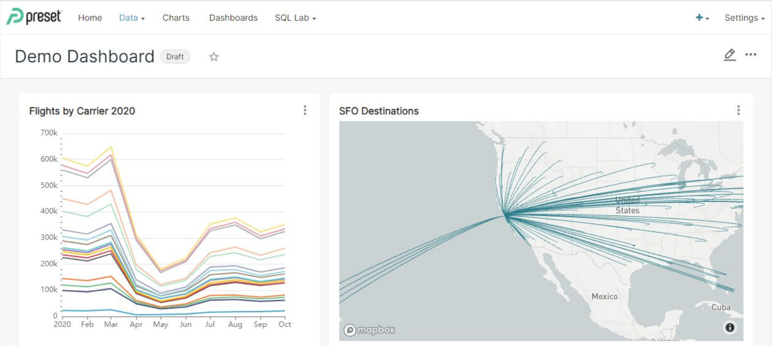
End user access patterns

Traditional approaches to end user access

Custom UIs – MUX.com video analytics*



Data Exploration Tools – Tableau, Metabase**

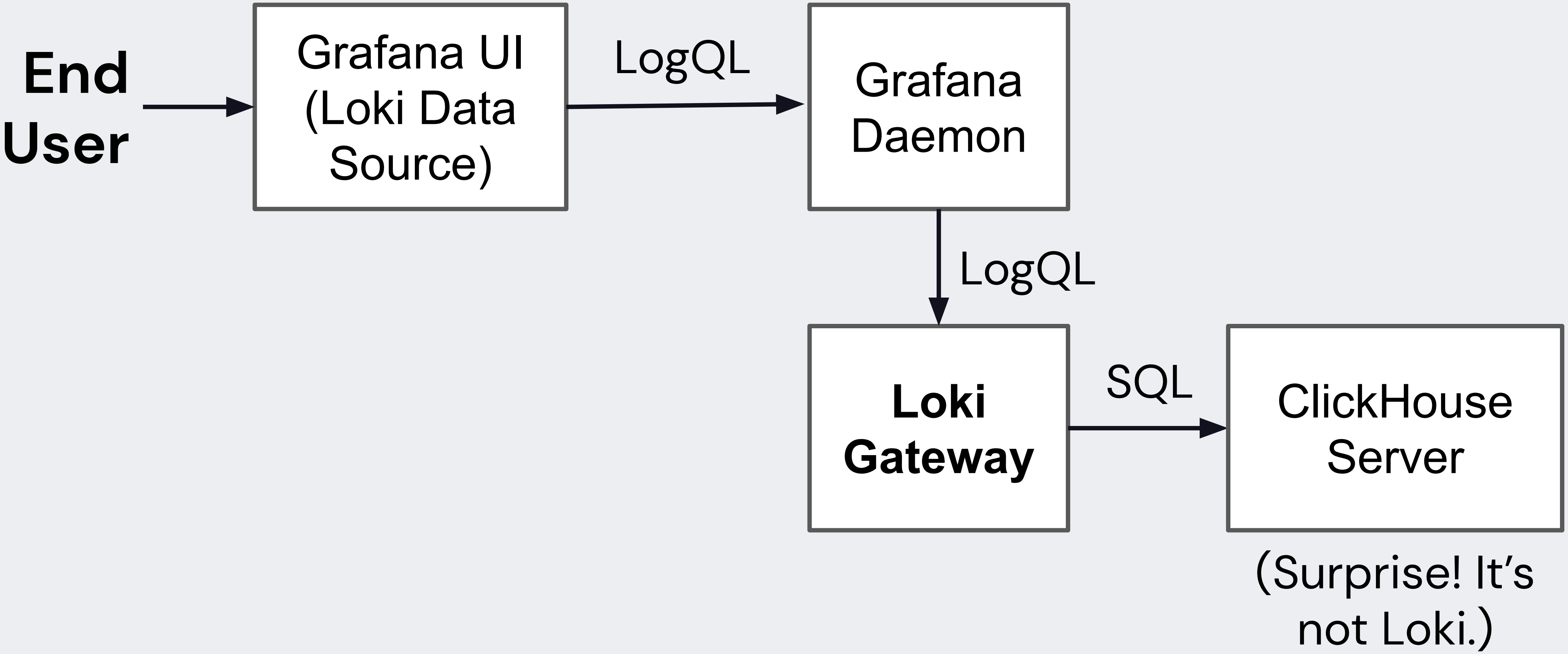


Dashboards – Superset, Grafana

*
https://www.mux.com/blog/from-russia-with-love-how-clickhouse-saved-our-data)
** https://www.metabase.com/product/

Leverage existing query/visualization tools

Aka subverting dominant UI paradigms



Wrap Up and Acknowledgements

Learnings from large ClickHouse installations

Use a single large table to hold all entities

Make sound implementation choices to get baseline performance

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

Build gateways from LogQL, PromQL, etc. to leverage powerful search UIs

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

Appreciations

- Arnaud Adant
- Mikhail Filimonov
- Anurag Gupta
- Lorenzo Mangani
- Alexey Milovidov
- Alexander Zaitsev
- The entire Altinity team

Thank you!

rhodges at altinity.com

https://altinity.com

Robert Hodges
Altinity CEO