Five Ways to Make your ClickHouse® Slow (and How to Avoid Them)

Robert Hodges - Altinity CEO
Mikhail Filimonov - ClickHouse Architect

18 February 2026

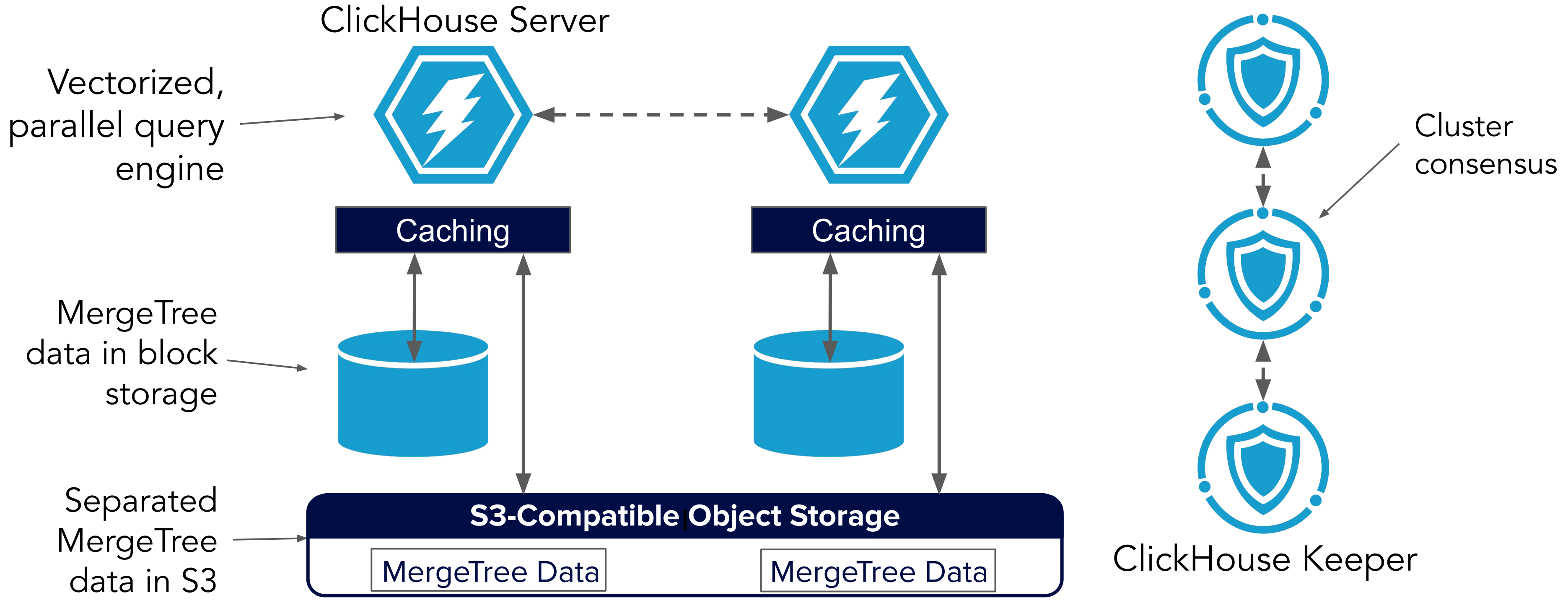
ALTINITY®

Run Open Source ClickHouse® Better

Altinity.Cloud

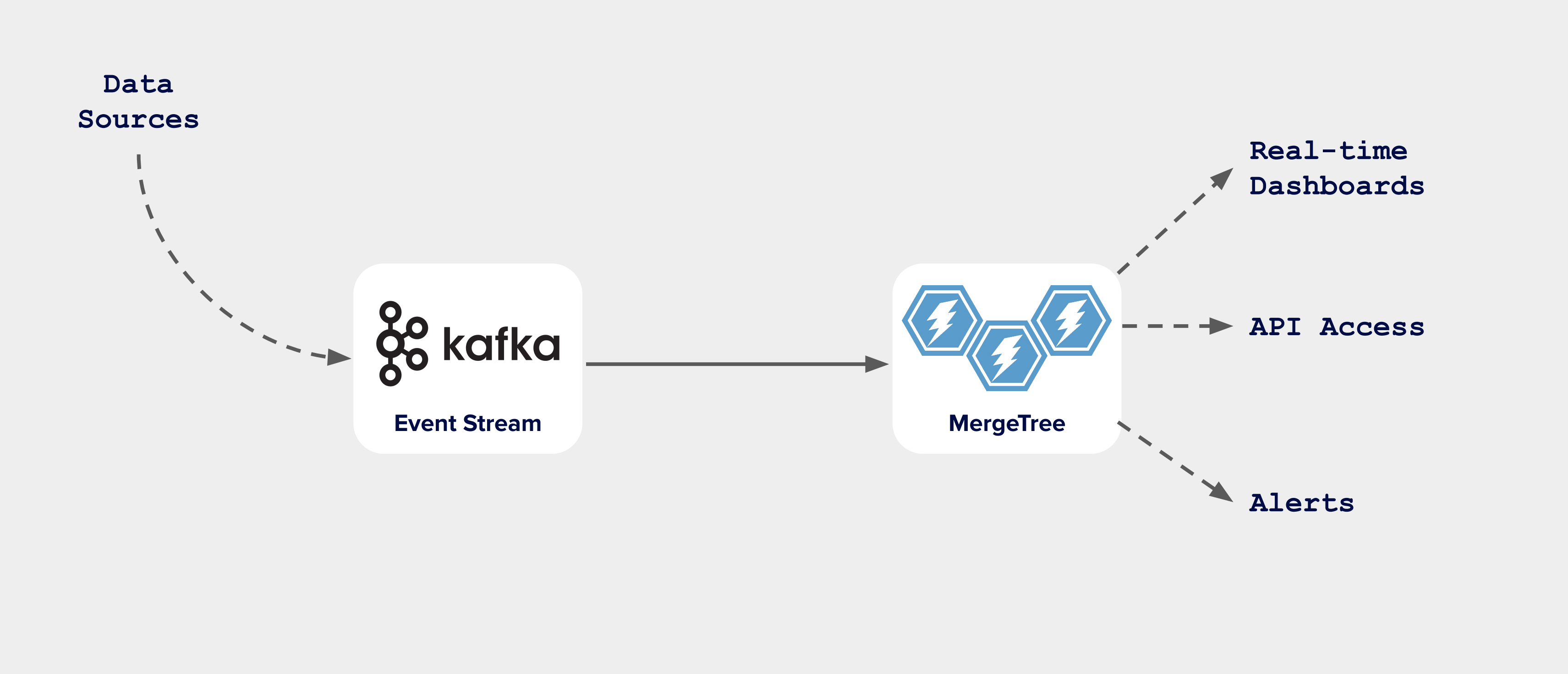
Enterprise Support

ClickHouse shared nothing architecture



ClickHouse Server
Vectorized, parallel query engine
Caching
Caching
MergeTree data in block storage
Separated MergeTree data in S3
S3-Compatible Object Storage
MergeTree Data
MergeTree Data
Cluster consensus
ClickHouse Keeper

Traditional real-time event pipeline with ClickHouse



Data Sources

Event Stream

MergeTree

Real-time Dashboards

API Access

Alerts

ClickHouse works great for almost any real-time analytics

```
SELECT Carrier, toYear(FlightDate) AS Year,
    (sum(Cancelled) / count(*)) * 100. AS cancelled_pct
FROM default.ontime_ref
GROUP BY Carrier, Year HAVING cancelled_pct > 1.
ORDER BY cancelled_pct DESC LIMIT 10

   ┌─Carrier─┬─Year─┬──────cancelled_pct─┐
1. │ G4      │ 2020 │ 16.73318604034276 │
2. │ EA      │ 1989 │ 10.32150096638536 │
3. │ WN      │ 2020 │  9.28430765359388 │
. . .
10 rows in set. Elapsed: 0.825 sec. Processed 196.51 million
rows, 982.57 MB (756.93 million rows/s., 1.19 GB/s.)
```

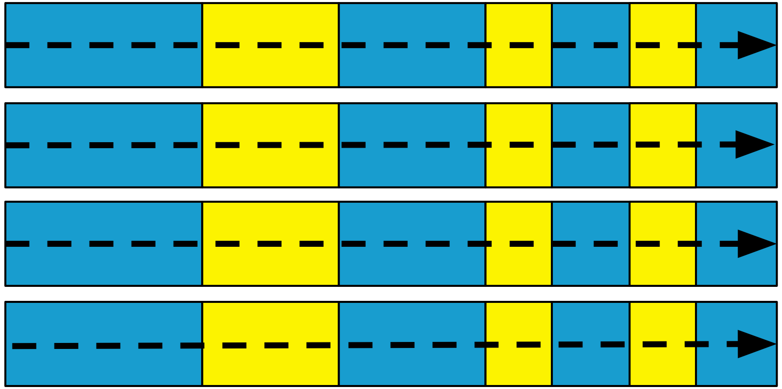
But…we did everything possible to make it slow!!

- Underpowered AWS m7g.xlarge Graviton with 4 vCPUs & 14GB RAM
- Slowest EBS storage speed: 125 MiB/sec
- Force cold reads with **SETTINGS min_bytes_to_use_direct_io = 1**

The secret of ClickHouse's success: compressed columns

PostgreSQL, MySQL

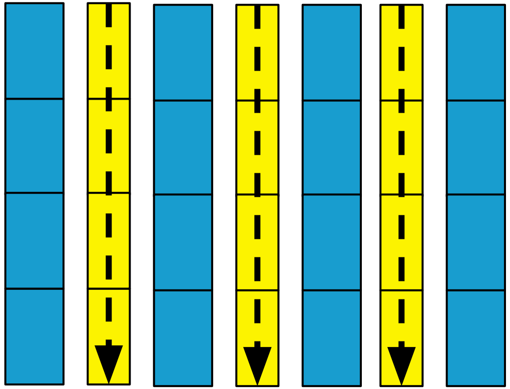
Read all columns in row



Rows minimally or not compressed

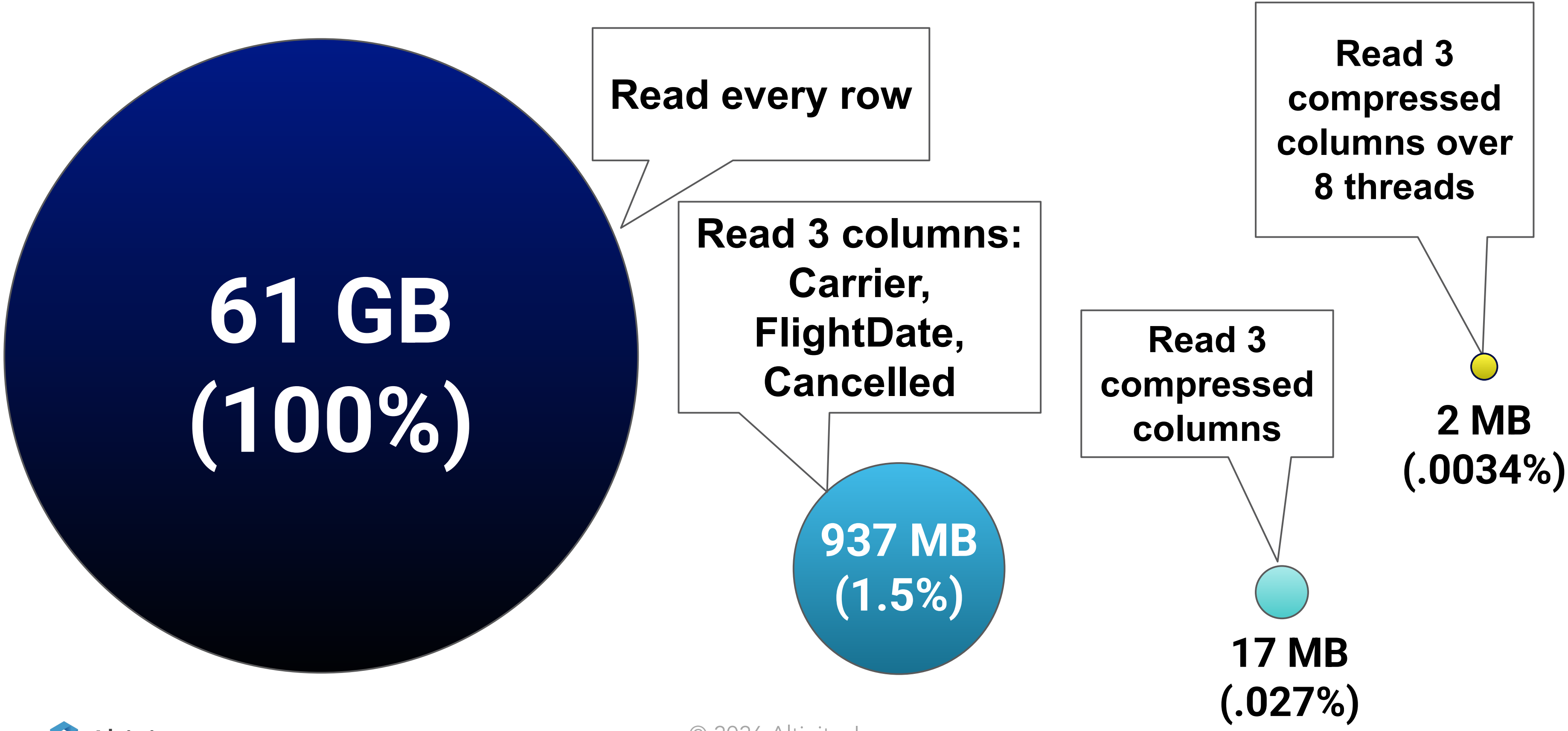
ClickHouse

Read only selected columns



Columns highly compressed

Visualizing effect of columns and compression



Read every row

61 GB (100%)

Read 3 columns: Carrier, FlightDate, Cancelled

937 MB (1.5%)

Read 3 compressed columns

17 MB (.027%)

Read 3 compressed columns over 8 threads

2 MB (.0034%)

Altinity

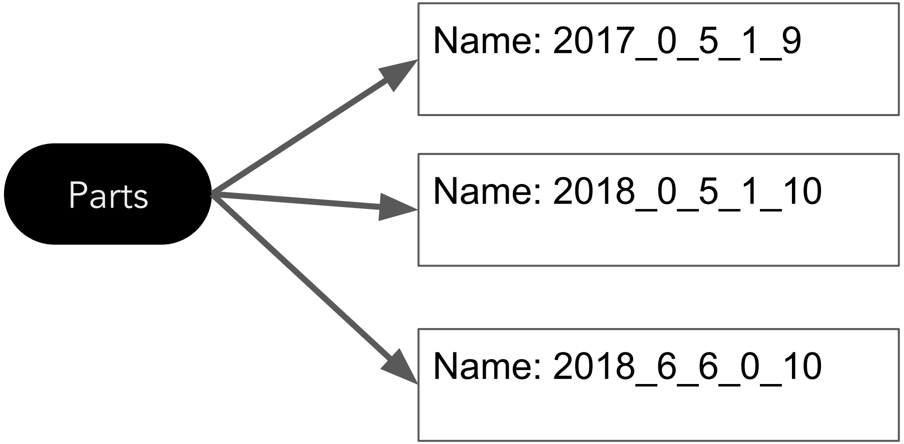
So…What could possibly go wrong?

Problem #1

Bad table design

Best practice: partition by time

```
CREATE TABLE default.ontime_ref( . . .)
ENGINE = MergeTree
PARTITION BY Year ORDER BY (Carrier, FlightDate)
```



Parts

Name: 2017_0_5_1_9

Name: 2018_0_5_1_10

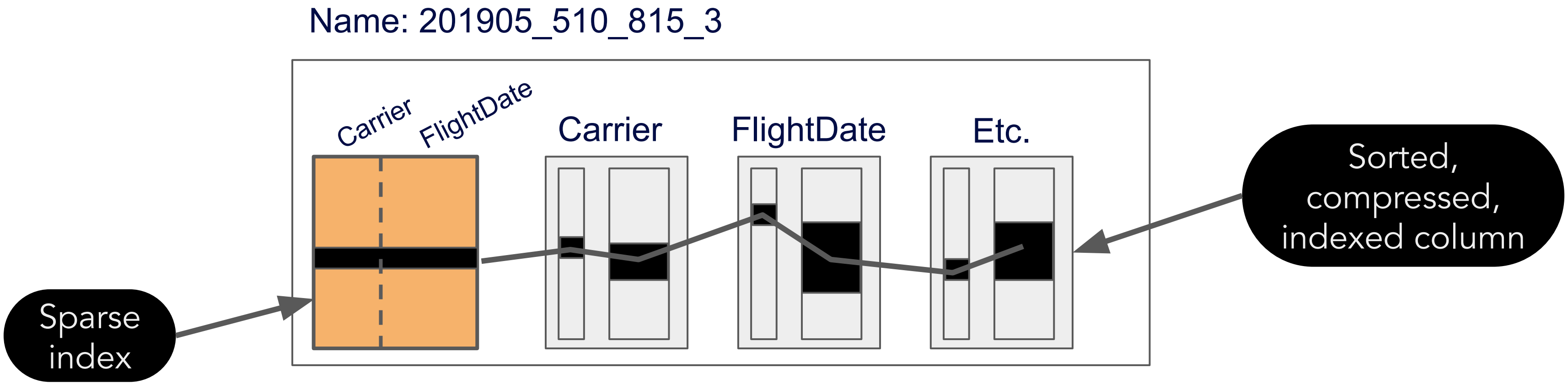
Name: 2018_6_6_0_10

Rule of thumb:

Choose partitions that result in ~1000 parts or less

Order by increasing cardinality, with tenant first

```
CREATE TABLE default.ontime_ref( . . .)
ENGINE = MergeTree
PARTITION BY Year ORDER BY (Carrier, FlightDate)
```



But what if we made a different choice of schema?

```
CREATE TABLE test.ontime_bad_partitioning
AS default.ontime_ref
ENGINE = MergeTree
PARTITION BY (Carrier, toYYYYMM(FlightDate))
ORDER BY (Carrier, FlightDate)

INSERT INTO test.ontime_bad_partitioning
SELECT *
FROM default.ontime_ref
SETTINGS max_threads = 1, max_insert_threads = 1
```

Pro tip: Reduce threads to avoid running out of memory

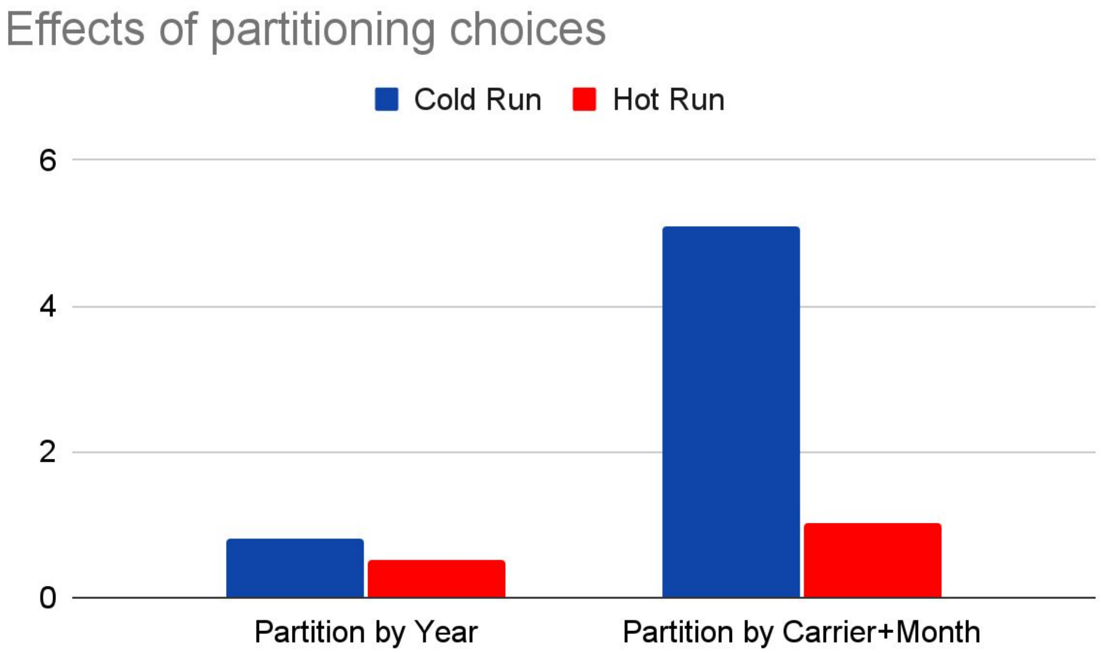
We can now make ClickHouse really slow!

```
SELECT Carrier, toYear(FlightDate) AS Year,
    (sum(Cancelled) / count(*)) * 100. AS cancelled_pct
FROM test.ontime_bad_partitioning
GROUP BY Carrier, Year HAVING cancelled_pct > 1.
ORDER BY cancelled_pct DESC LIMIT 10
[SETTINGS min_bytes_to_use_direct_io = 1]
```

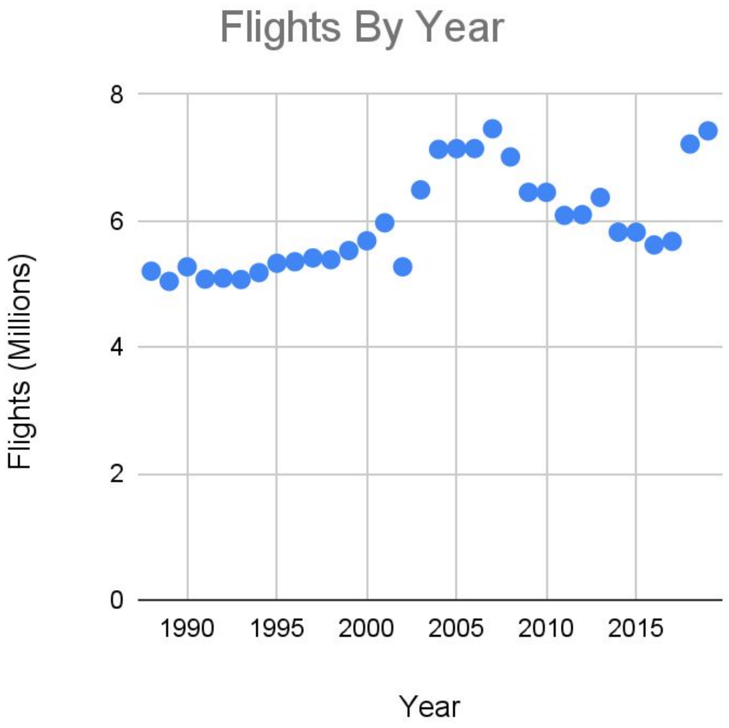
Force direct I/O

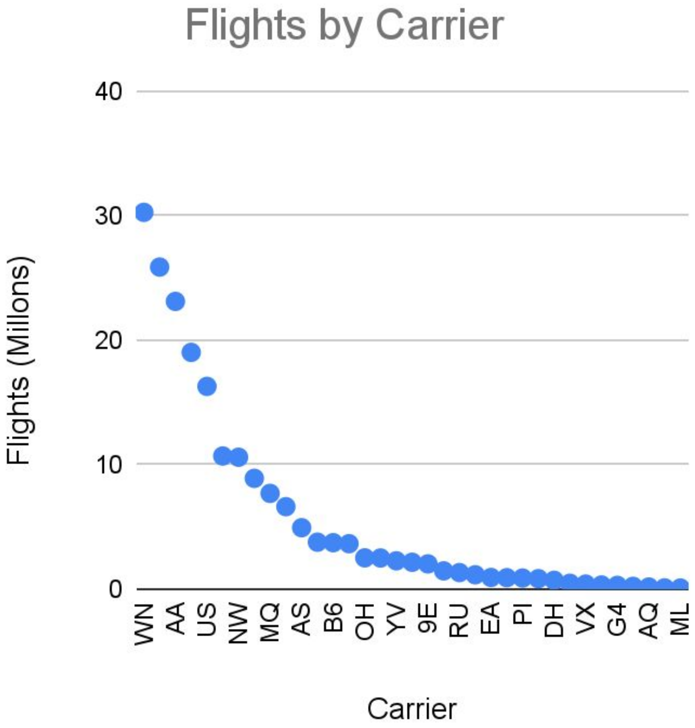
```
   ┌─Carrier─┬─Year─┬──────cancelled_pct─┐
1. │ G4      │ 2020 │ 16.733186040434276 │
. . .
10 rows in set. Elapsed: 5.092 sec. Processed 196.51 million
rows, 982.57 MB (38.59 million rows/s., 160.74 MB/s.)
```

Bad partitioning == bad performance!!



Why it’s better to partition by year?





Cheat sheet for schema design

Hard to change!

1. Time-based column in PARTITION BY
2. Put tenants using ORDER BY, then add columns in order of cardinality
3. Use appropriate datatypes (e.g., Int32, not String)
4. Use codecs like Delta or LowCardinality
5. Use ZSTD compression instead of default LZ4 to really squeeze space
    a. Use it *if* you hit I/O limits but have free CPU capacity

Measure compression with amazing system tables!

```
SELECT
    count(),
    formatReadableSize(sum(data_compressed_bytes),
    formatReadableSize(sum(data_uncompressed_bytes)
FROM system.columns
WHERE (database = 'default') AND (`table` = 'ontime_ref')
AND (name IN ('Carrier', 'FlightDate', 'Cancelled'))
```

Other great tables: system.parts and system.tables

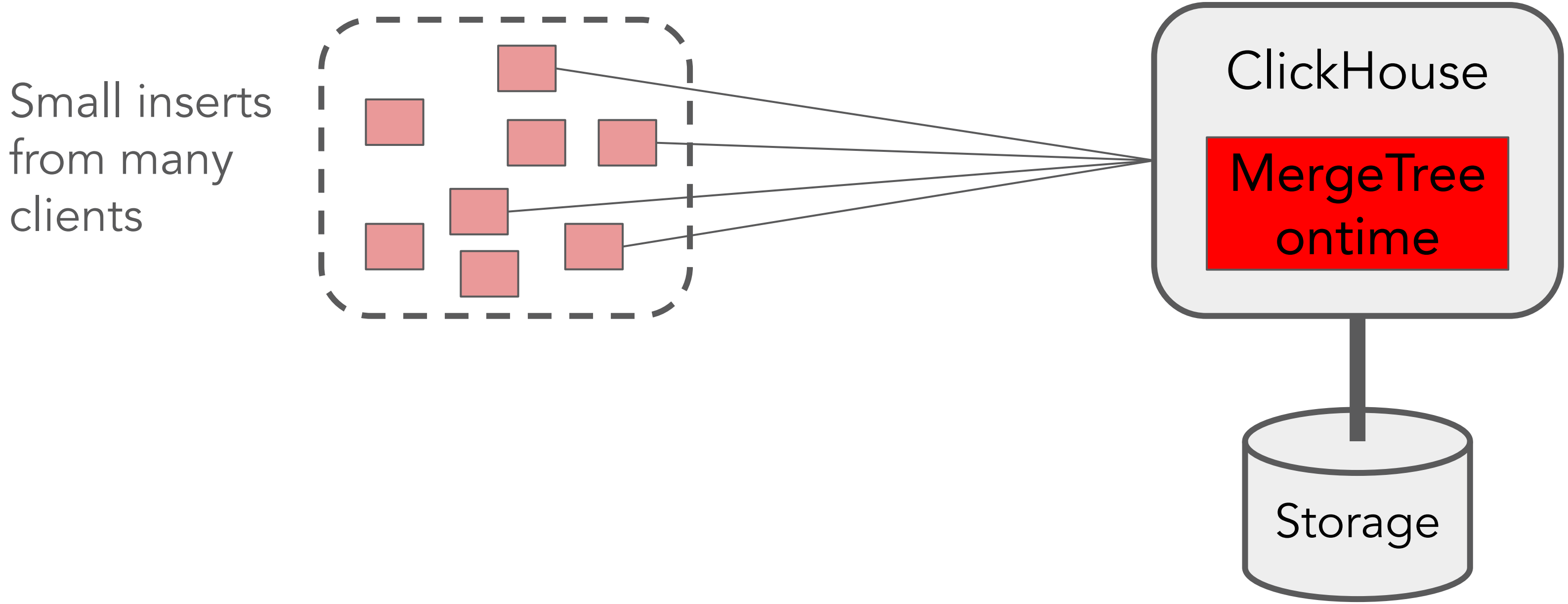
Problem #2

Too many tiny inserts

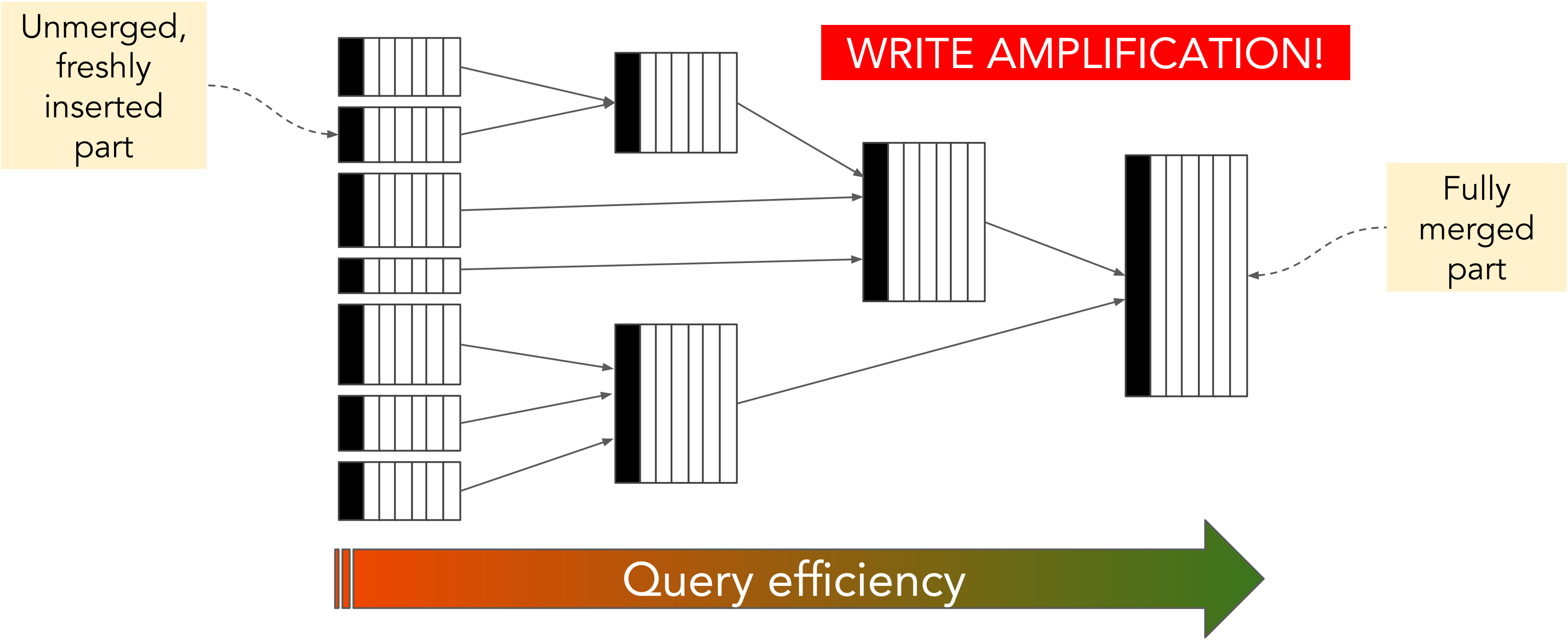
If your inserts look like this, you are doing it wrong!

```
INSERT INTO default.ontime_ref VALUES
(2017,4,12,12,2,'2017-12-12','UA\0\0\0\0\0',
19977,'UA',...),
(2017,4,12,12,2,'2017-12-12','UA\0\0\0\0\0',
19977,'UA',...)
```

Small inserts can crush your ClickHouse server



Lots of small parts == slow queries and high merge load

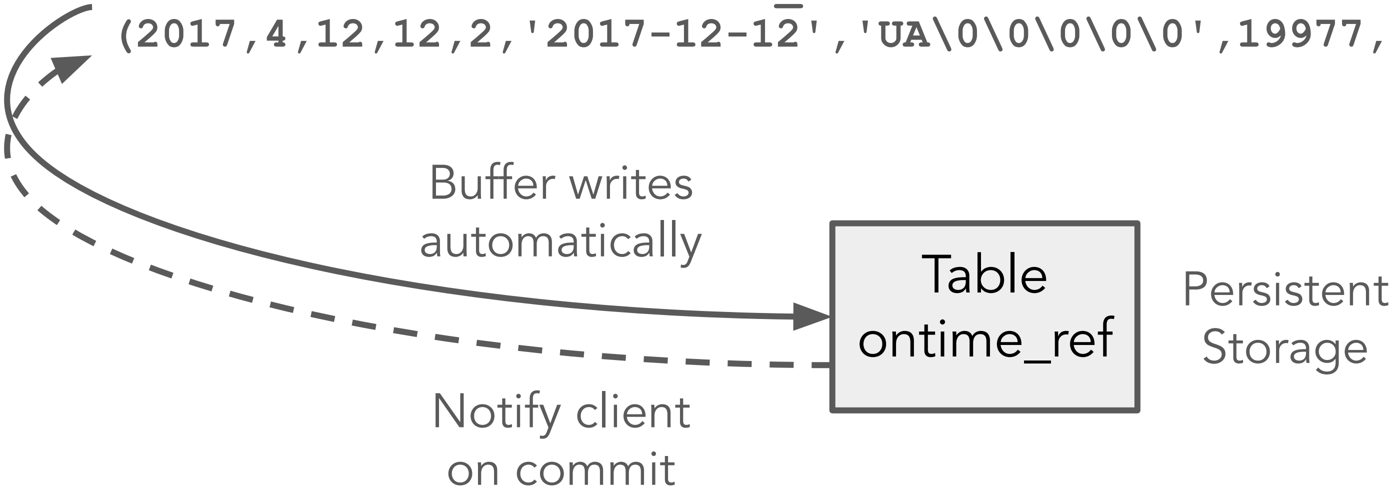


Fix #1: Use big batches in your application

```
#!/bin/bash
INSERT='INSERT+INTO+ontime+Format+CSVWithNames'
cat test.csv | curl -X POST --data-binary @- \
      "http://localhost:8123/?query=${INSERT}"
```

Fix #2: Enable async inserts

```
INSERT INTO default.ontime_ref VALUES
(2017,4,12,12,2,'2017-12-12','UA\0\0\0\0\0',19977,'UA',...),
```



Buffer writes automatically

Table ontime_ref

Persistent Storage

Notify client on commit

https://kb.altinity.com/altinity-kb-queries-and-syntax/async-inserts/

Enable async inserts using property settings

```
CREATE SETTINGS PROFILE IF NOT EXISTS `async_profile`
ON CLUSTER '{cluster}'
SETTINGS
  async_insert = 1,
  wait_for_async_insert=1,
  async_insert_busy_timeout_ms = 10000,
  async_insert_use_adaptive_busy_timeout = 0
;

CREATE USER IF NOT EXISTS async ON CLUSTER '{cluster}'
  IDENTIFIED WITH sha256_password BY 'topsecret' HOST ANY
  SETTINGS PROFILE `async_profile`
;
```

Use async insert and wait for answer

Wait this long

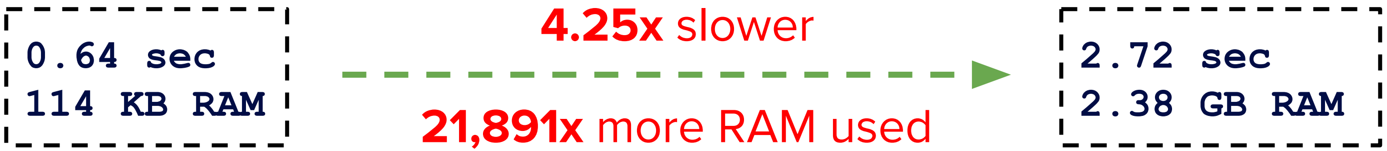
Don't let ClickHouse set automatic values

User with settings

Problem #3

Bad queries

Small differences in queries make big differences in response



```
0.64 sec
114 KB RAM
```

4.25x slower

21,891x more RAM used

```
2.72 sec
2.38 GB RAM
```

```
SELECT Carrier,
  avg(DepDelay)AS Delay
FROM ontime_ref
GROUP BY Carrier
ORDER BY Delay DESC
LIMIT 50
```

Simple aggregate, short GROUP BY key with few values

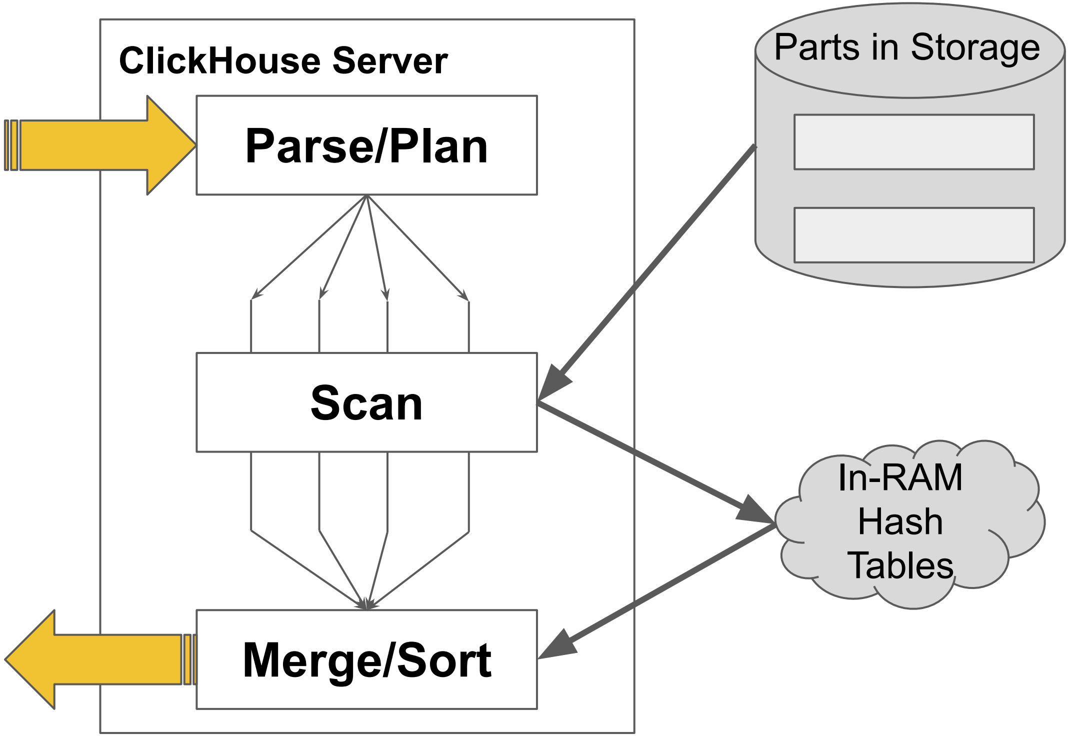
```
SELECT Carrier, FlightDate,
  avg(DepDelay) AS Delay,
  uniqExact(TailNum) AS Aircraft
FROM ontime_ref
GROUP BY Carrier, FlightDate
ORDER BY Delay DESC
LIMIT 50
```

More complex aggregates, longer GROUP BY with more values

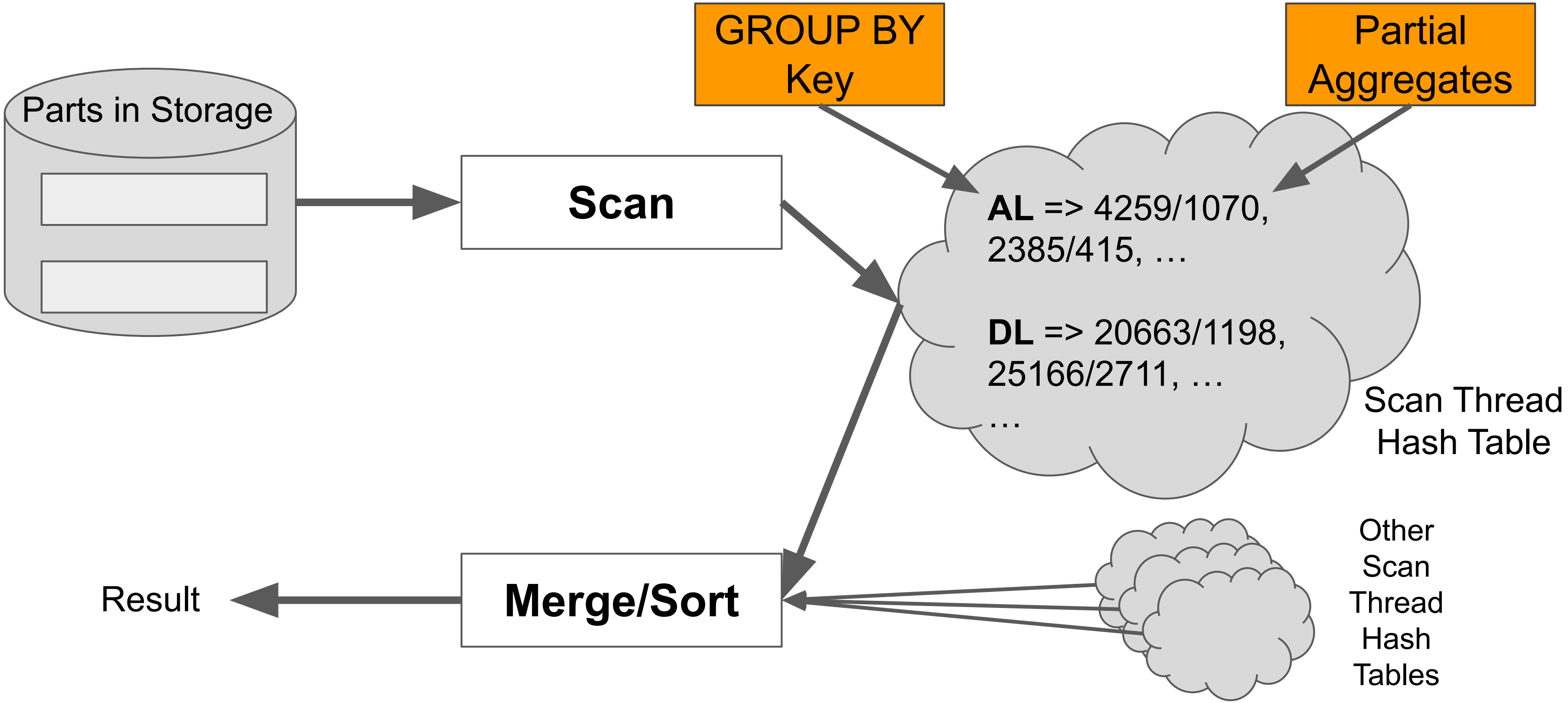
How does ClickHouse process a query with aggregates?

```
SELECT Carrier,
  avg(DepDelay)AS Delay
FROM ontime
GROUP BY Carrier
ORDER BY Delay DESC
```

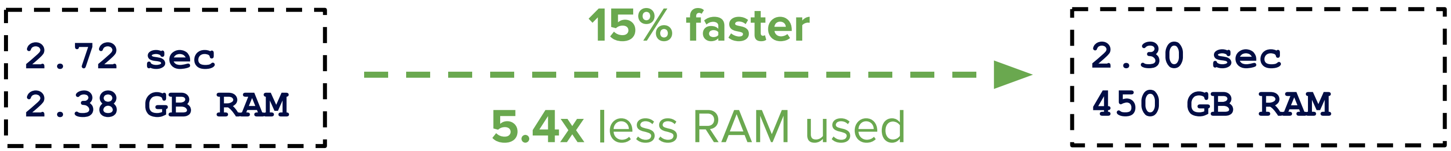
```
┌─Carrier─┬─────────────Delay─┐
│ B6      │ 12.058290698785067 │
│ EV      │ 12.035012037703922 │
│ NK      │ 10.437692933474269 │
. . .
```



How does a ClickHouse thread do aggregation?



We can optimize queries by choosing better aggregates



```
2.72 sec
2.38 GB RAM
```

15% faster

5.4x less RAM used

```
2.30 sec
450 GB RAM
```

```
SELECT Carrier, FlightDate,
  avg(DepDelay) AS Delay,
  uniqExact(TailNum) AS Aircraft
FROM ontime
GROUP BY Carrier, FlightDate
ORDER BY Delay DESC
LIMIT 50
```

uniqExact stores each unique value in a hash table that grows

```
SELECT Carrier, FlightDate,
  avg(DepDelay) AS Delay,
  uniqHLL12(TailNum) AS Aircraft
FROM ontime
GROUP BY Carrier, FlightDate
ORDER BY Delay DESC
LIMIT 50
```

uniqHLL12 uses fixed size HyperLogLog structure

Here’s some magic with optimizing joins

```
SELECT o.Dest, any(a.Name) AS AirportName,
  count(Dest) AS Flights
FROM ontime_ref o
JOIN default.airports a ON a.IATA = o.Dest
GROUP BY Dest ORDER BY Flights
DESC LIMIT 10
```

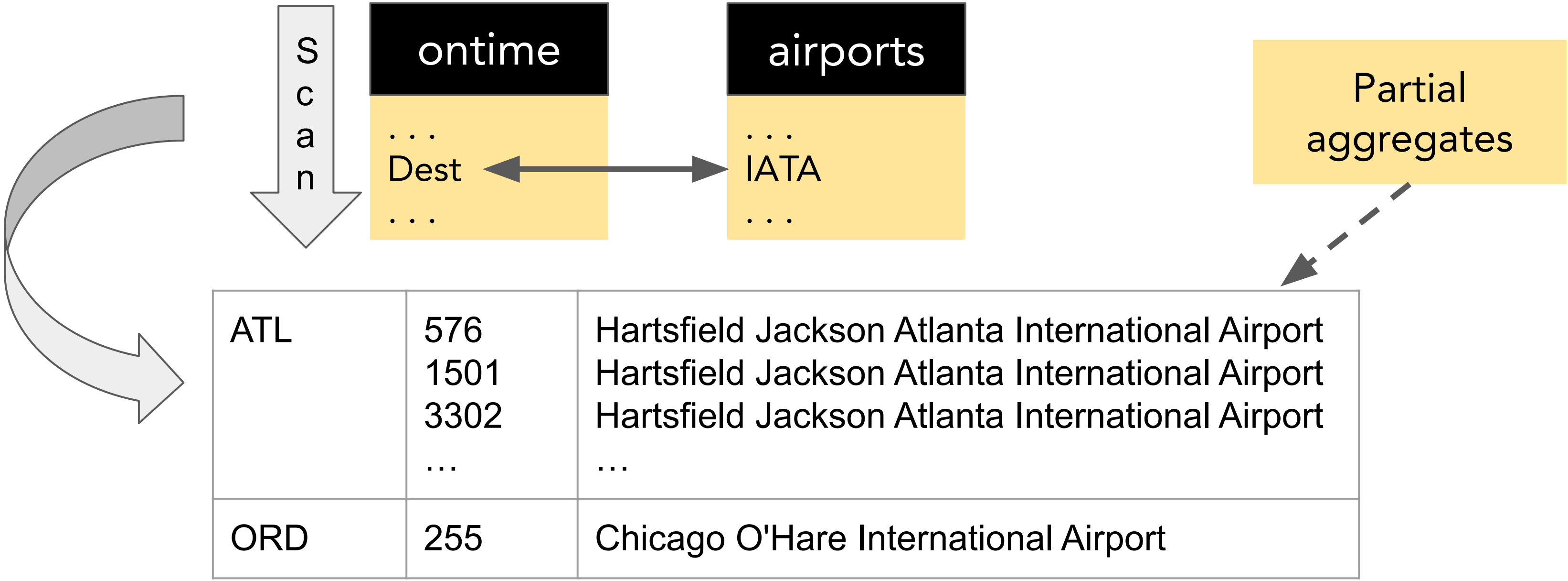
2.685 sec
39.18 MB RAM

```
SELECT o.Dest, a.Name AS AirportName, o.Flights
FROM (
  SELECT Dest, count(Dest) AS Flights
  FROM ontime_ref GROUP BY Dest ) AS o
JOIN default.airports a ON a.IATA = o.Dest
ORDER BY Flights DESC LIMIT 10
```

0.524 sec
1.02 MB RAM

Let’s look more deeply at what’s happening in the scan

```
SELECT . . . FROM ontime o JOIN airports a ON a.IATA = o.Dest
```



ATL	576 1501 3302 …	Hartsfield Jackson Atlanta International Airport Hartsfield Jackson Atlanta International Airport Hartsfield Jackson Atlanta International Airport …
ORD	255	Chicago O'Hare International Airport

Where did those awesome query stats come from?

```
SELECT
    event_time,
    type,
    is_initial_query,
    query_duration_ms / 1000 AS duration,
    read_rows,
    read_bytes,
    result_rows,
    formatReadableSize(memory_usage) AS memory,
    query
FROM system.query_log
WHERE (user = 'default') AND (type = 'QueryFinish')
ORDER BY event_time DESC
LIMIT 50
```

Fixing queries efficiently

#1: Run against real data (and plenty of it

#2: Isolate slow queries and optimize them

#3: Rinse and repeat

Problem #4

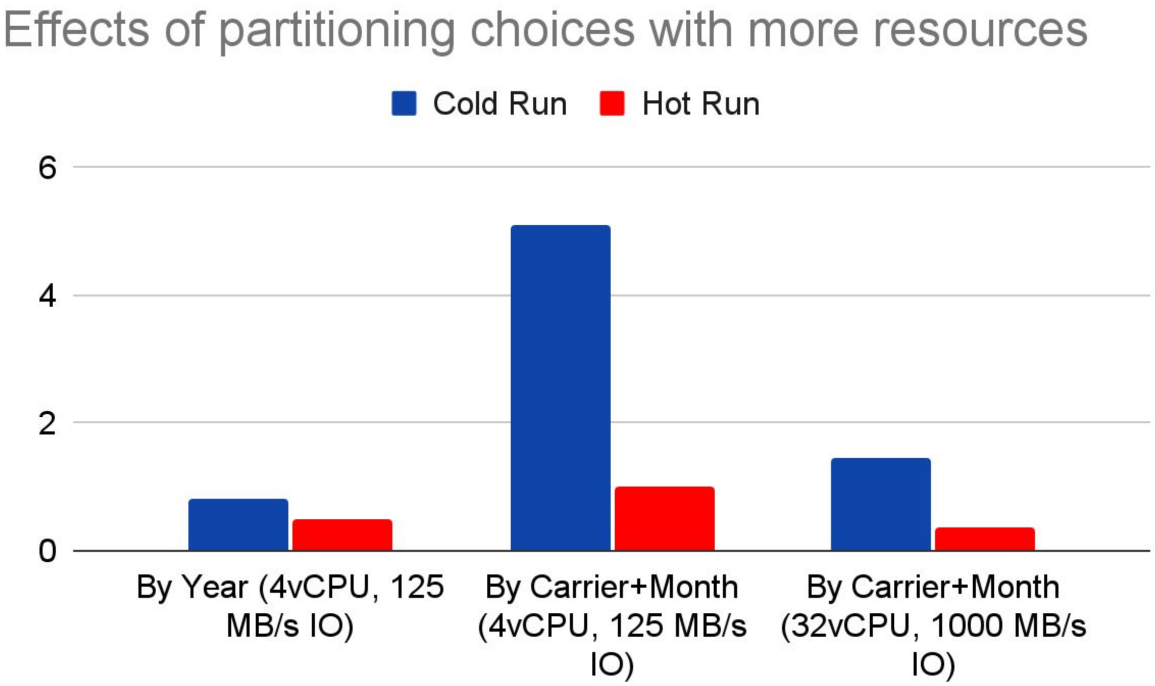
Insufficient resources

Is your query still too slow? Throw money at it!

```
SELECT Carrier, toYear(FlightDate) AS Year,
    (sum(Cancelled) / count(*)) * 100. AS cancelled_pct
FROM test.ontime_bad_partitioning
GROUP BY Carrier, Year HAVING cancelled_pct > 1.
ORDER BY cancelled_pct DESC LIMIT 10
[SETTINGS min_bytes_to_use_direct_io = 1]

   ┌─Carrier─┬─Year─┬──────cancelled_pct─┐
1. │ G4      │ 2020 │ 16.733186040434276 │
. . .
10 rows in set. Elapsed: 5.092 sec. Processed 196.51 million
rows, 982.57 MB (38.59 million rows/s., 160.74 MB/s.)
```

Better hardware can make slow queries fast...



But ClickHouse servers don’t just run queries…

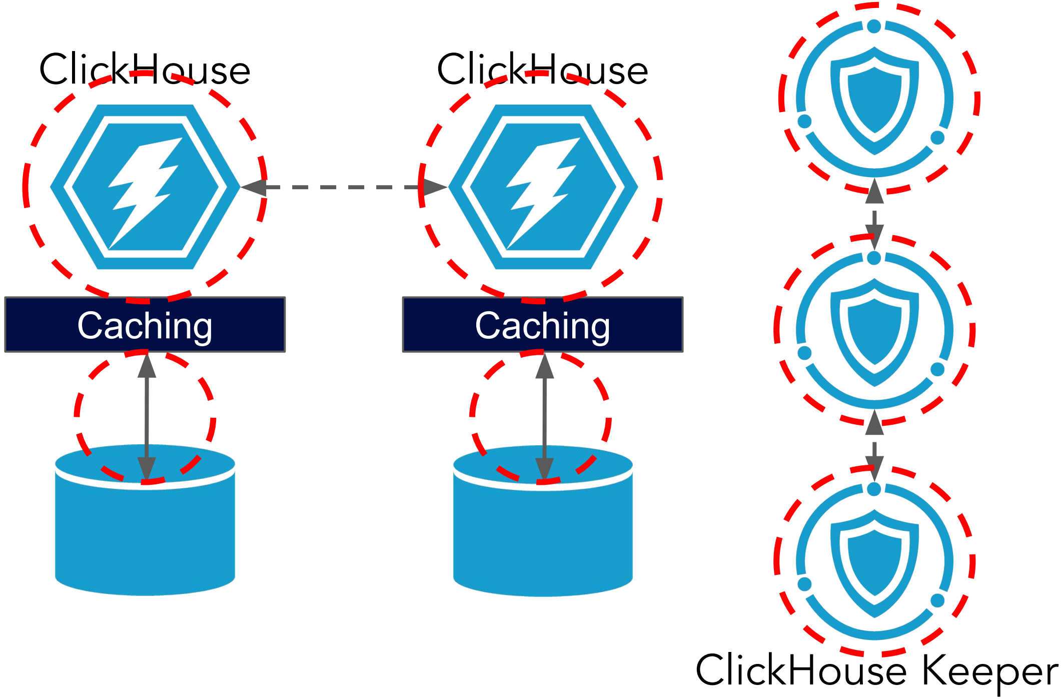
4-Course Menu

Query

Insert

Merge

Update



Common issues with resource management

#1: Testing real workloads (**large & concurrent**)

#2: Detecting trouble: CPU, IOWait, RAM, Network

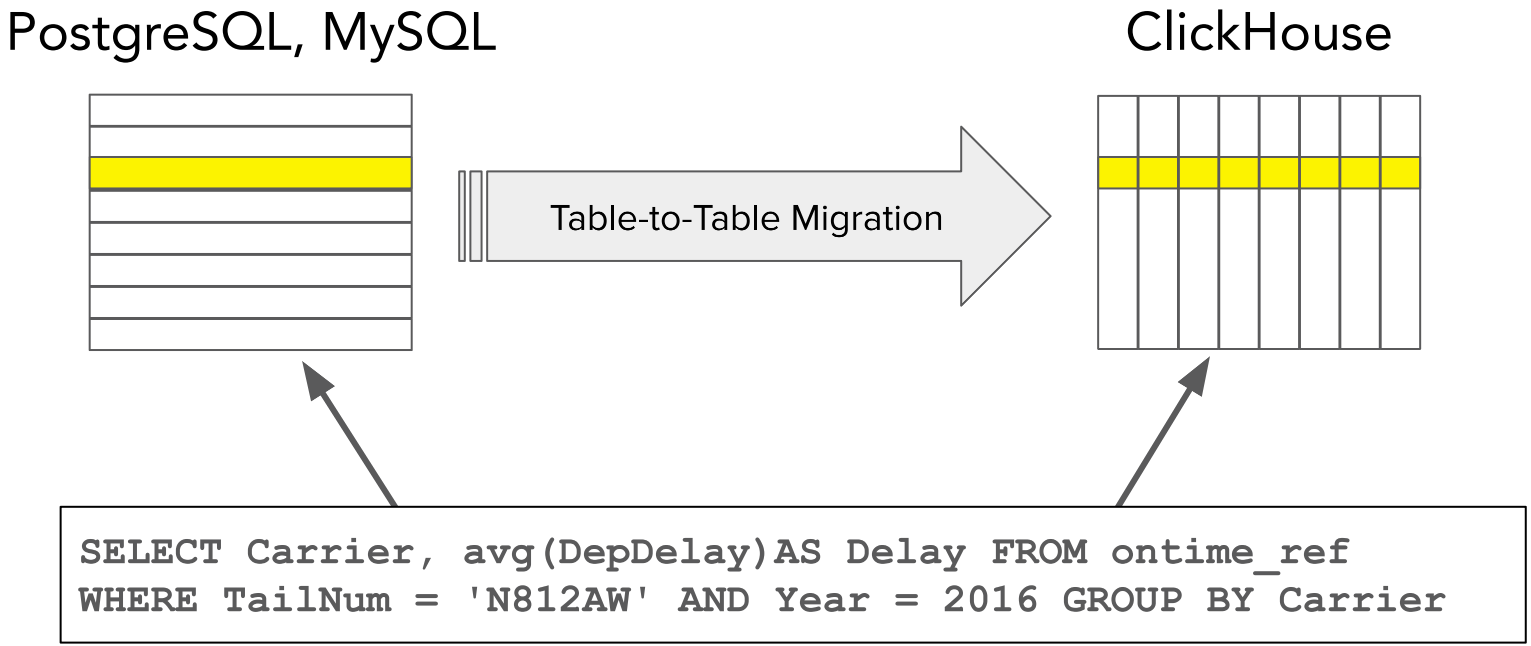
#3: Scaling quickly when trouble hits

#4: Fixing apps that overuse resources

Problem #5

Migrations from non-compatible databases to ClickHouse

Some migrations to ClickHouse just work



```
SELECT Carrier, avg(DepDelay)AS Delay FROM ontime_ref
WHERE TailNum = 'N812AW' AND Year = 2016 GROUP BY Carrier
```

Others are more “challenging”

Events with full text indexes

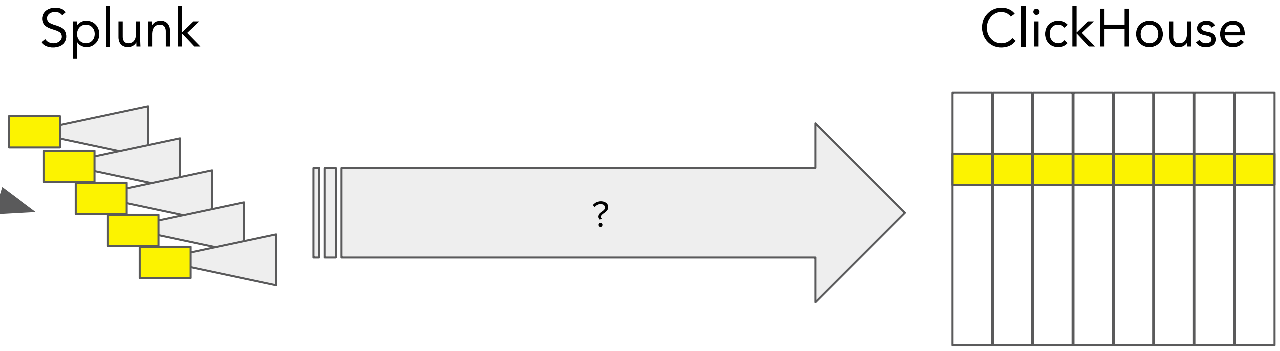
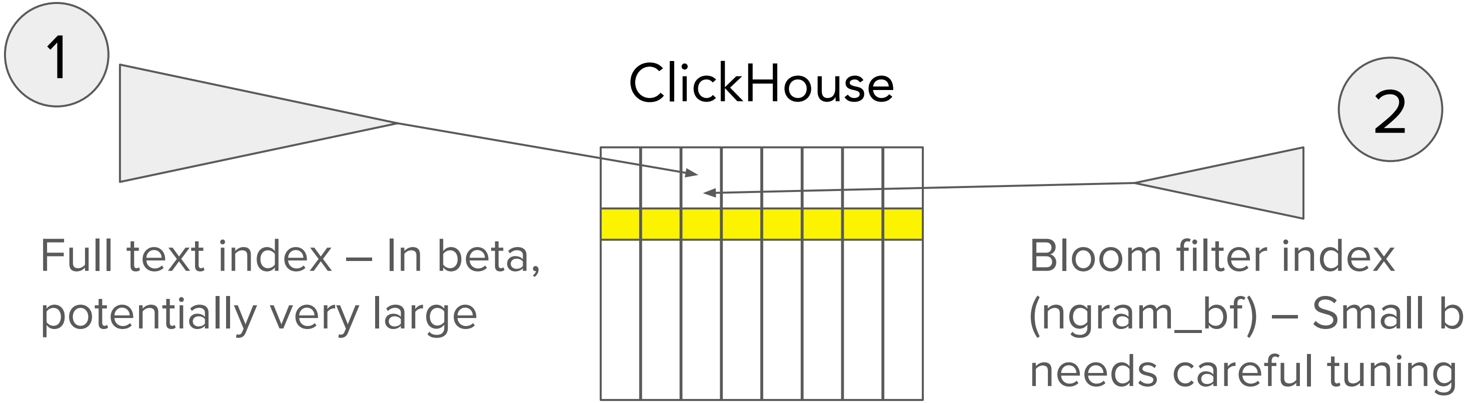


Table with skip indexes

This is full-text search

```
SELECT Carrier, TailNum, avg(DepDelay) AS Delay FROM ontime_ref
WHERE hasToken(TailNum, 'N812') OR hasToken(TailNum, 'N573')
GROUP BY Carrier, TailNum
```

We need to decide how to implement full text search



Full text index – In beta, potentially very large

Bloom filter index (ngram_bf) – Small but needs careful tuning

```
SELECT Carrier, TailNum, avg(DepDelay) AS Delay FROM ontime_ref
WHERE TailNum LIKE 'N812%' OR TailNum LIKE 'N573%'
GROUP BY Carrier, TailNum
```

3 LIKE operator – It's fast and ngram_bf index makes it faster

Migrating different database types to ClickHouse

#1: Test queries under realistic load

#2: Rethink slow query patterns

#3: It takes time to tune indexes and queries

Server log messages are your friend

```
SELECT Carrier, TailNum, avg(DepDelay) AS Delay FROM
rhodges_7273b.ontime_bloom_filter
WHERE TailNum LIKE '%N812%' OR TailNum LIKE '%N128%'
GROUP BY Carrier, TailNum ORDER BY Delay DESC LIMIT 10
SETTINGS send_logs_level='debug'

... 2026.02.18 07:00:54.615183 [ 31 ]
{caf33ae2-a6ee-424d-b4f0-0ac022edab32} <Debug> executeQuery: (from
[::ffff:10.129.59.185]:60032, user: admin) (query 1, line 1) SELECT …

... 2026.02.18 07:00:54.630439 [ 31 ]
{caf33ae2-a6ee-424d-b4f0-0ac022edab32} <Debug>
rhodges_7273b.ontime_bloom_filter (SelectExecutor): Index
`TailNum_Ngrambf` has dropped 14033/24089 granules, it took 21ms across 4
threads
```

Altinity

Avoiding the Five Performance Problems

- Tune your schema to reduce I/O
- Make inserts as big as possible
- Test queries on real data and fix the slow ones
- Test hardware on realistic workloads and increase it before you hit problems
- Design and test migrations from other databases carefully!

Don’t assume ClickHouse will be fast. Prove it!!!

Thank you! Questions?

Check out our TTL Guide!

Robert Hodges
CEO Altinity

https://altinity.com

We’re hiring!

OPEN SOURCE

My LinkedIn

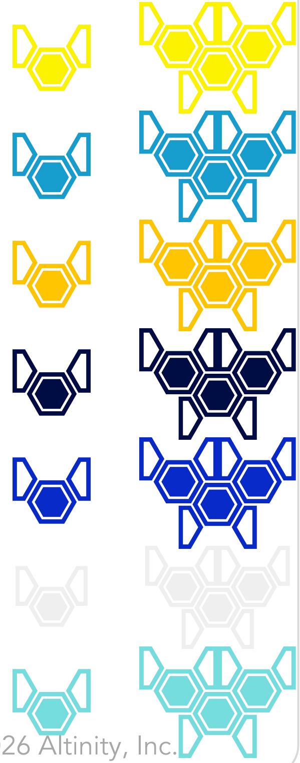
ALTINITY

Icons- Transparent

Clickhouse (Native) Cluster

Director Cluster

Swarm Cluster

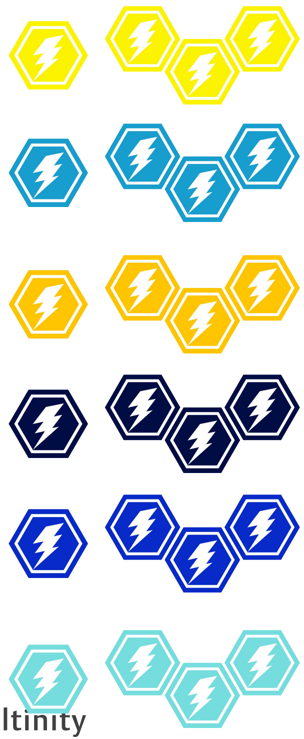


Keeper

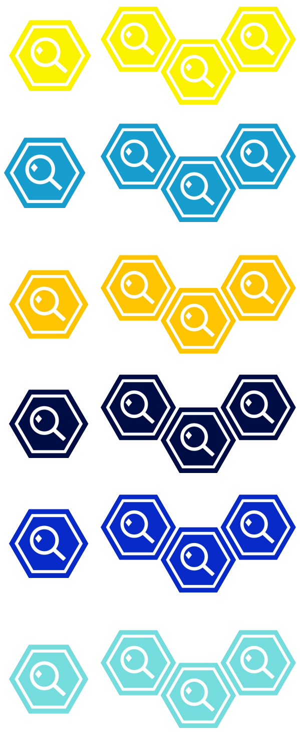
Other

Icons-Stackable on White Backgrounds

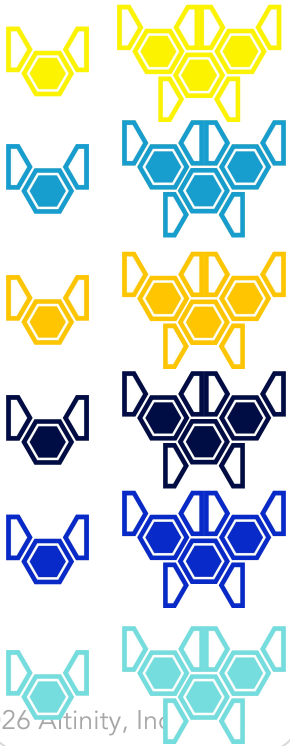
Clickhouse (Native) Cluster



Director Cluster



Swarm Cluster



Keeper

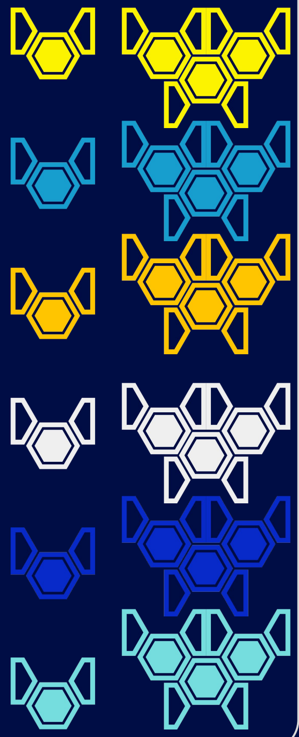
Other

Icons-Stackable on Dark Backgrounds

Clickhouse (Native) Cluster

Director Cluster

Swarm Cluster



Keeper

Other