A Day in the Life of a ClickHouse Query

Intro to ClickHouse Internals

Robert Hodges & Altinity Engineering

10 February 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

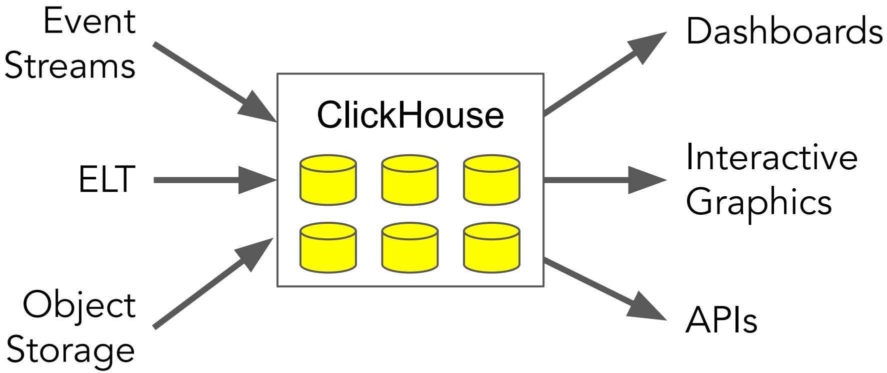
Shared nothing architecture

Stores data in columns

Parallel and vectorized execution

Scales to many petabytes

Is Open source (Apache 2.0)



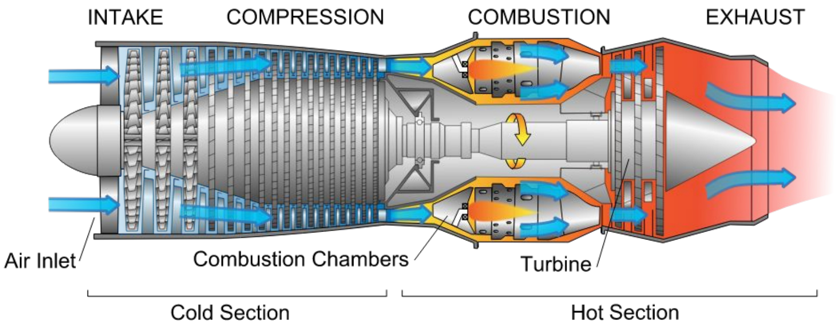
It's a popular engine for real-time analytics

If you understand the engine you can make it faster

ClickHouse has a simple execution model–there’s no magic

Any developer can understand how it works

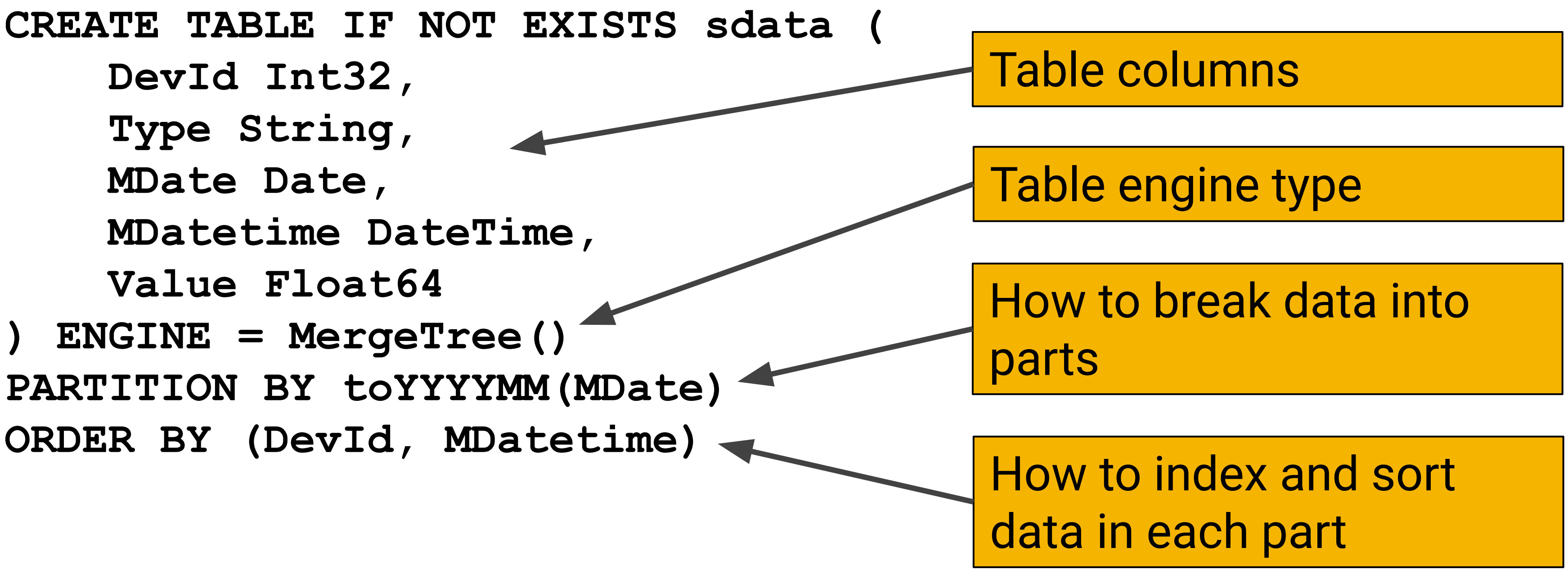
Knowledge leads to faster and more efficient queries



(Another fast engine!)

What happens when you insert data?

Let’s create a table!



```
CREATE TABLE IF NOT EXISTS sdata (
    DevId Int32,
    Type String,
    MDate Date,
    MDatetime DateTime,
    Value Float64
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(MDate)
ORDER BY (DevId, MDatetime)
```

Table columns

Table engine type

How to break data into parts

How to index and sort data in each part

Let’s now insert some data…

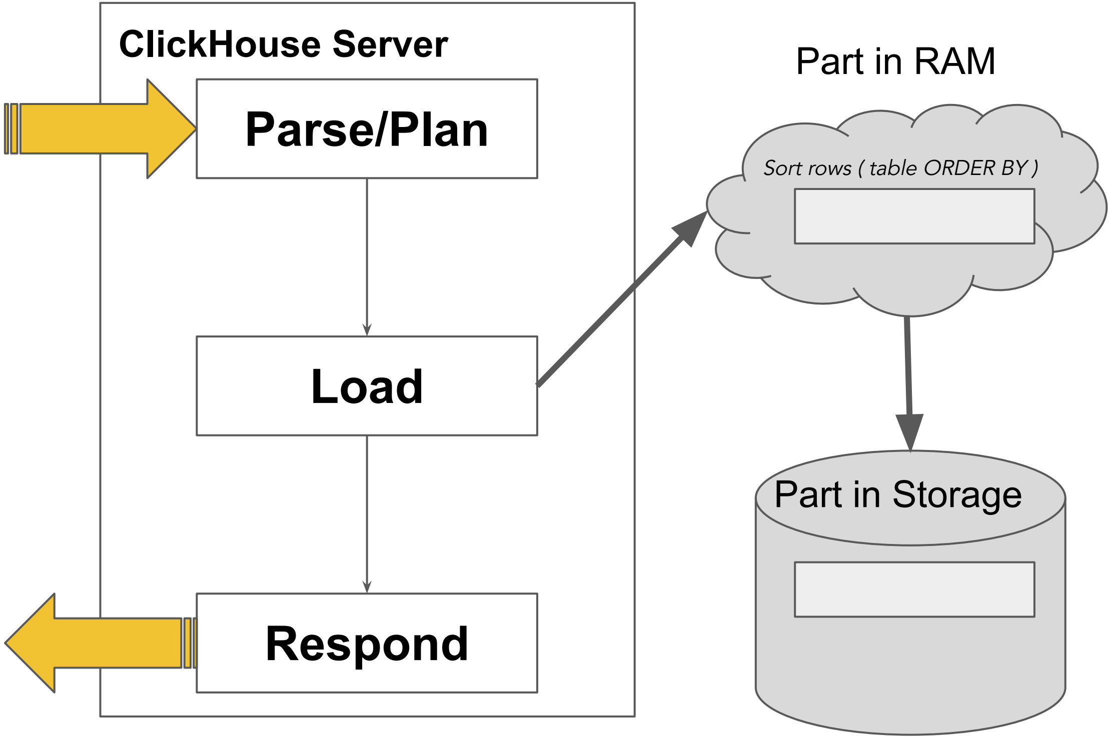
```
INSERT INTO sdata VALUES
(15, 'TEMP', '2018-01-01', '2018-01-01 23:29:55', 18.0),
(15, 'TEMP', '2018-01-01', '2018-01-01 23:30:56', 18.7)
```

(This is an example. Most people don’t insert data this way!)

How does ClickHouse process an insert?

```
INSERT INTO sdata
VALUES
(15, 'TEMP', . . .),
(15, 'TEMP', . . .)
```

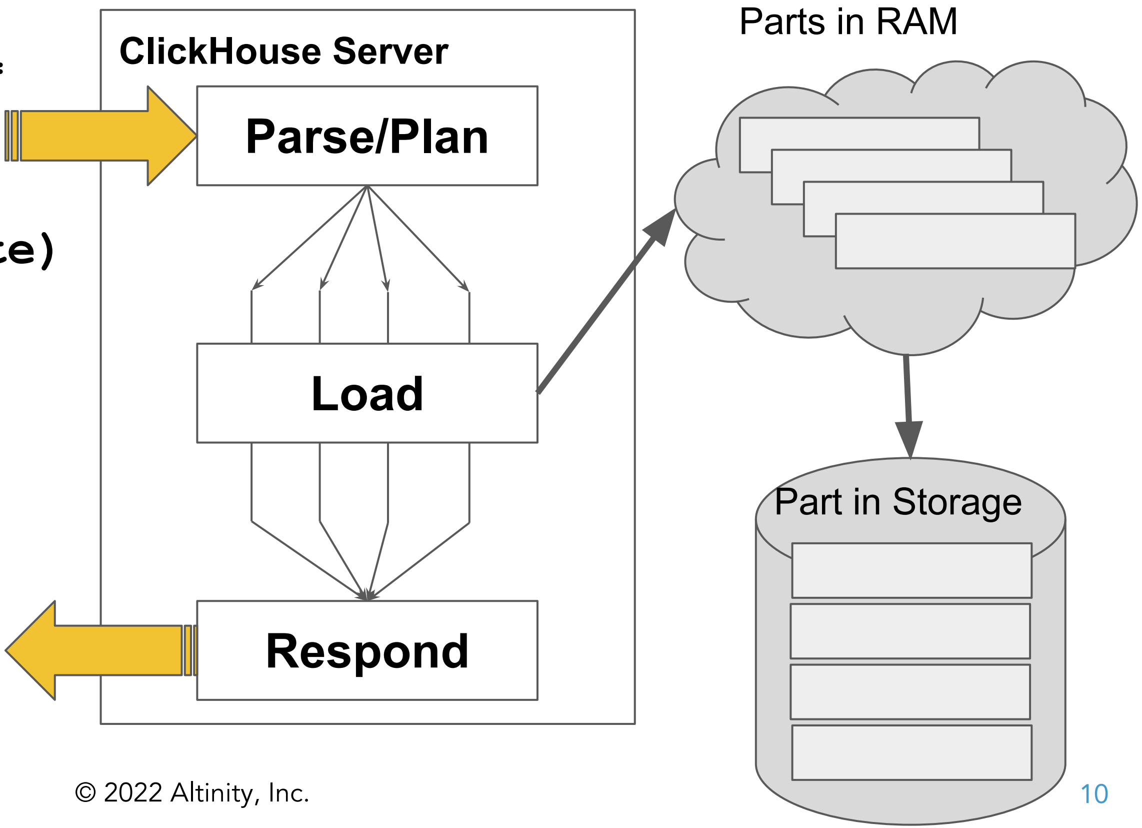
```
2 rows in set. Elapsed: 0.271 sec.
```



How can we make this more efficient? Parallelize!

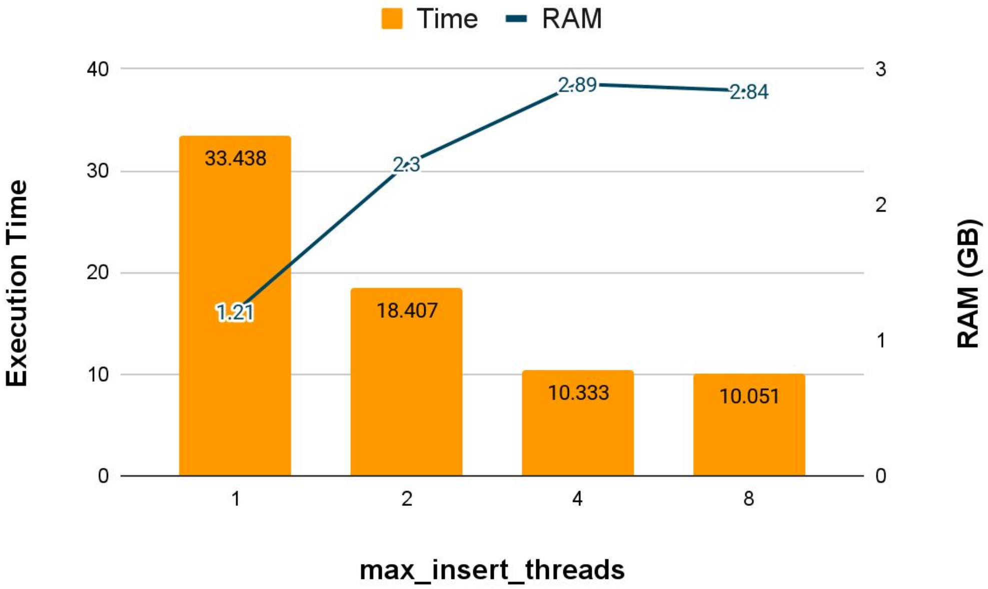
```
set max_insert_threads=4
insert into ontime_test
select * from ontime
  where toYear(FlightDate)
  between 2000 and 2001
```

```
2 rows in set. Elapsed: 0.271 sec.
```



Parallelism affects speed and memory usage

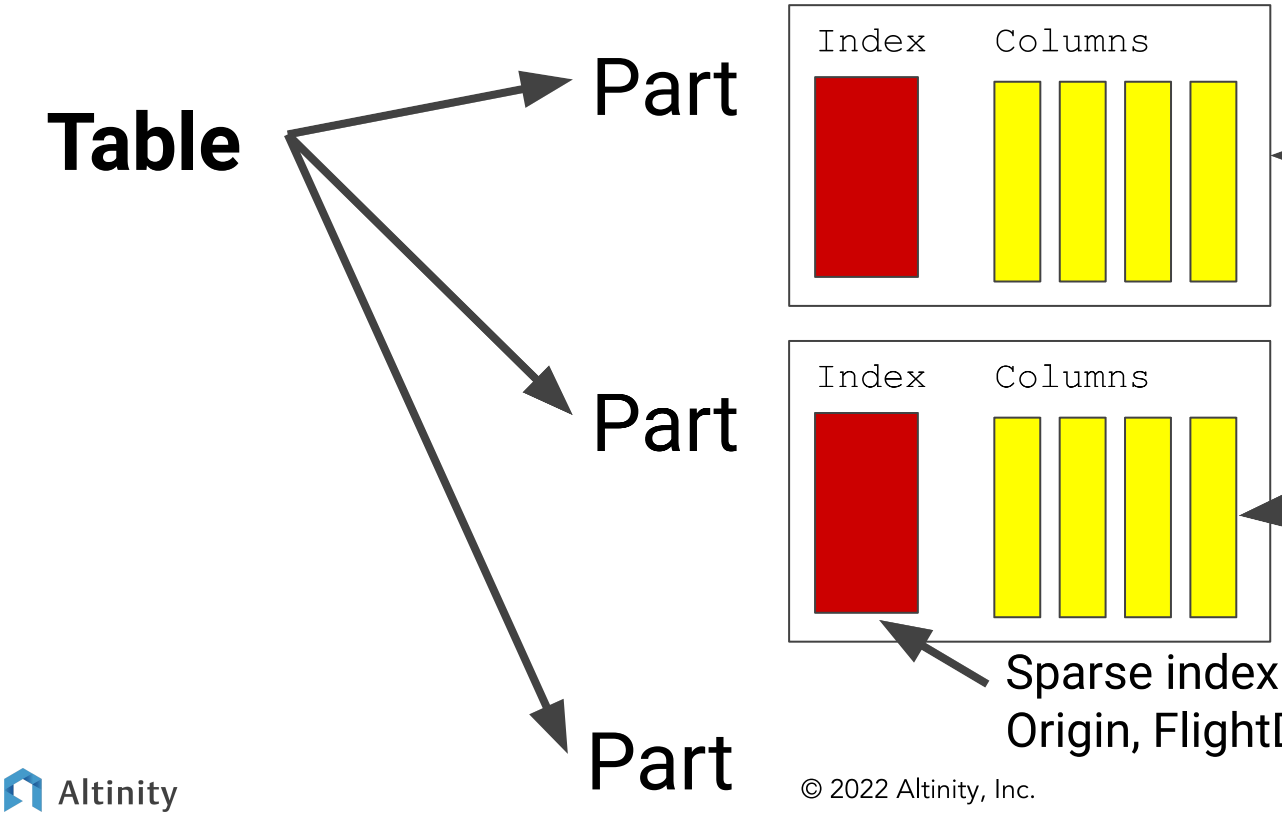
```
insert into ontime_test
select * from ontime
  where toYear(FlightDate)
  between 2000 and 2001



set max_insert_threads=1
. . .
set max_insert_threads=2
. . .
set max_insert_threads=4
```



OK, where did those awesome 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
```

What's going on down there when you INSERT?

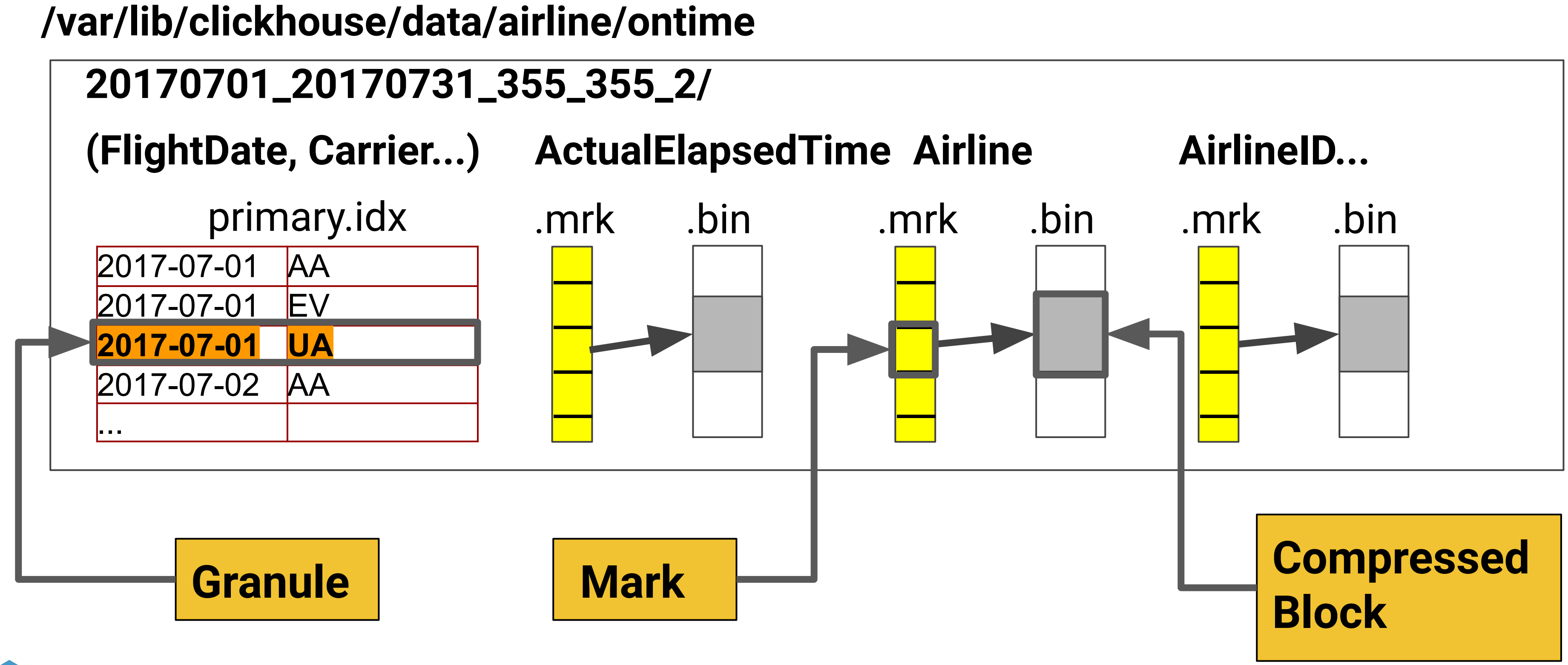


Rows in the part all belong to same Year

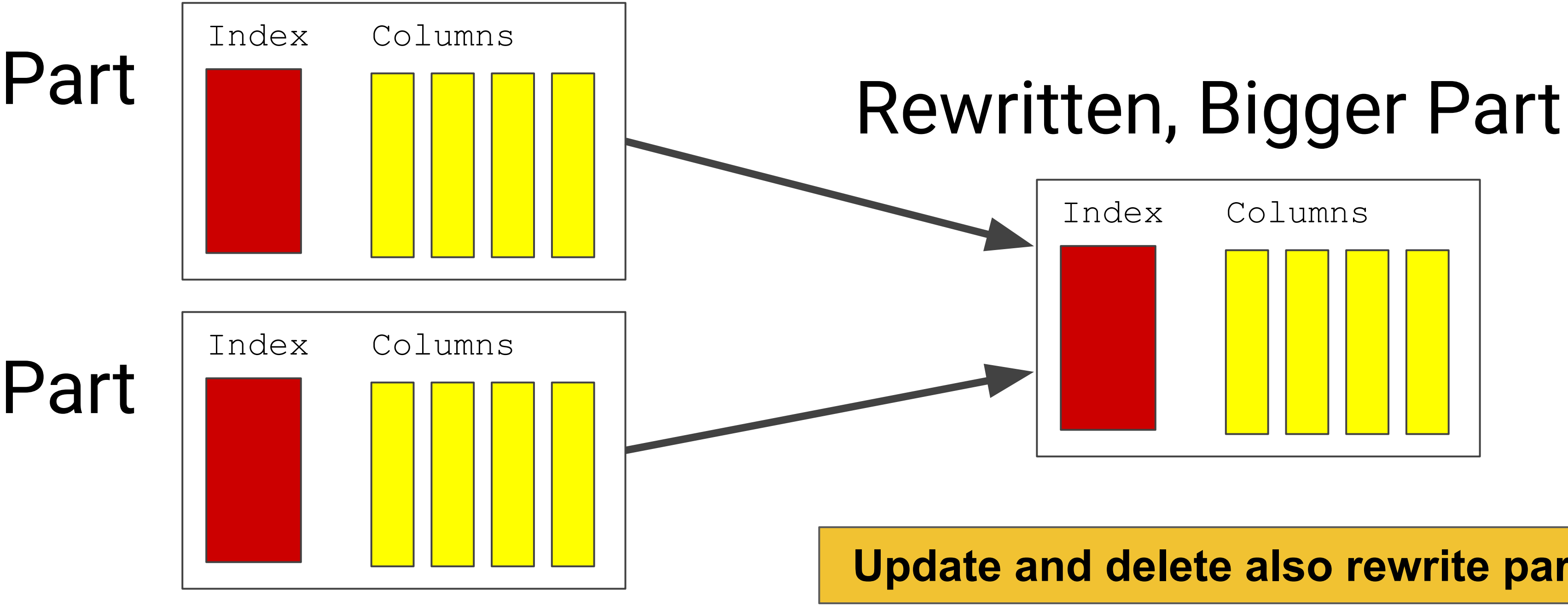
Columns sorted by Carrier, Origin, FlightDate

Sparse index finds rows by Carrier, Origin, FlightDate

Understanding what’s in a MergeTree part



Why MergeTree? Because it merges!



Update and delete also rewrite parts

© 2022 Altinity, Inc.

Bigger parts are more efficient!

- Pick a PARTITION BY that gives nice, fat partitions (1-300GB, < 1000 total parts per table)
  - Can’t decide? Partition by month.
- Insert <u>large</u> blocks of data to avoid lots of merges afterwards
  - ClickHouse is fine with tens of millions of rows!
- The simplest way to make blocks bigger is to batch input data
  - Avoid different partition keys in the same block
  - ClickHouse has parameters like max_insert_block_size but defaults are OK
  - Look at logs and actual part sizes to see if you need to do more

How can I see how big table parts are?

```
SELECT
    table, partition, name,
    marks, rows, data_compressed_bytes,
    data_uncompressed_bytes, bytes_on_disk
FROM system.parts
WHERE active
  AND level=0
  AND database = 'default'
  AND table = 'ontime_test'
ORDER BY table DESC, partition ASC, name ASC
```

Part is in use; can also omit

Part has not been merged

Tips to optimized INSERT

Making INSERT faster

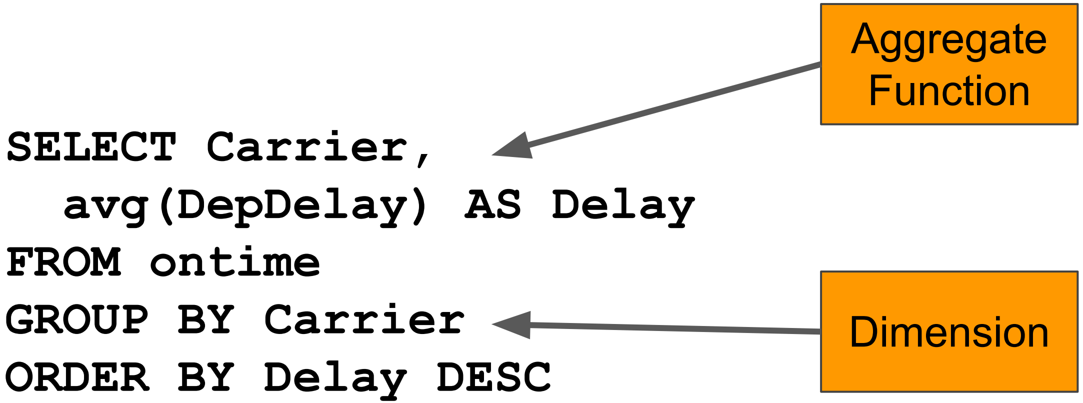
- Increase **`max_insert_threads`** (parallel creation of parts)
- Enable **`input_format_parallel_parsing`** to parallelize input parsing
    - Works for TSV/CSV/Values data
- Write bigger blocks (less merging afterwards)

Making INSERT less memory intensive

- Decrease **`max_insert_threads`** (reduces parts simultaneously in memory)
- Disable **`input_format_parallel_parsing`**
- Write smaller blocks (less memory required at INSERT time)

How do basic queries work?

Aggregation is a key feature of analytic queries



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

Aggregate Function

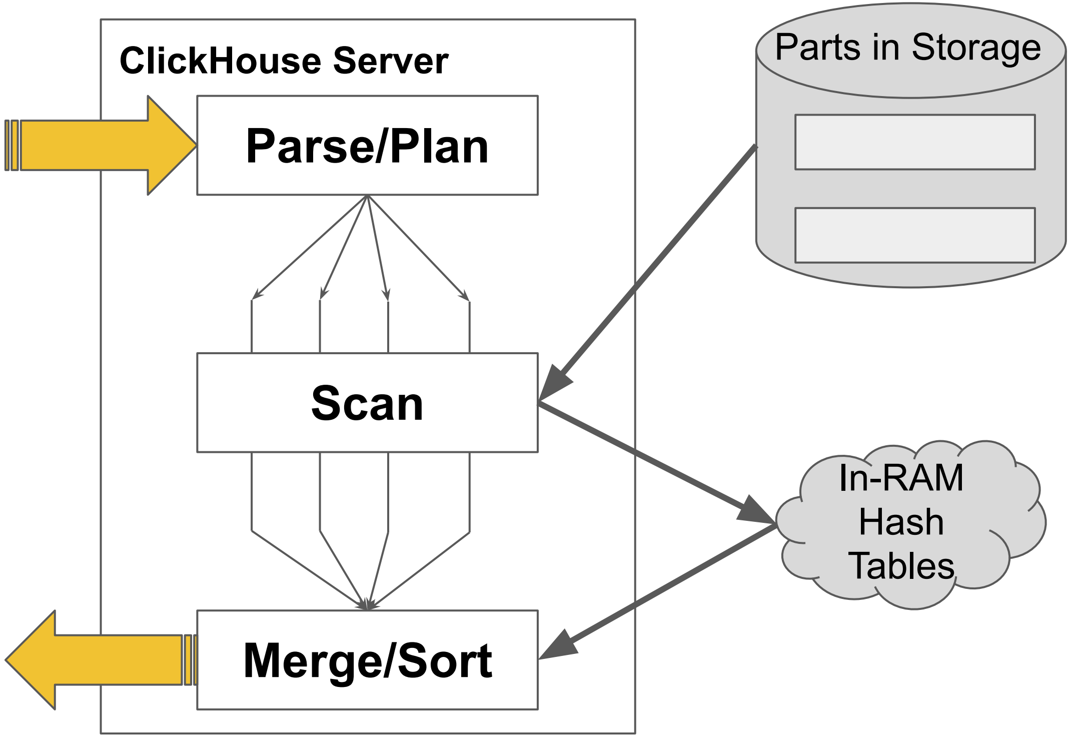
Dimension

Aggregates group measurements for one more more dimensions

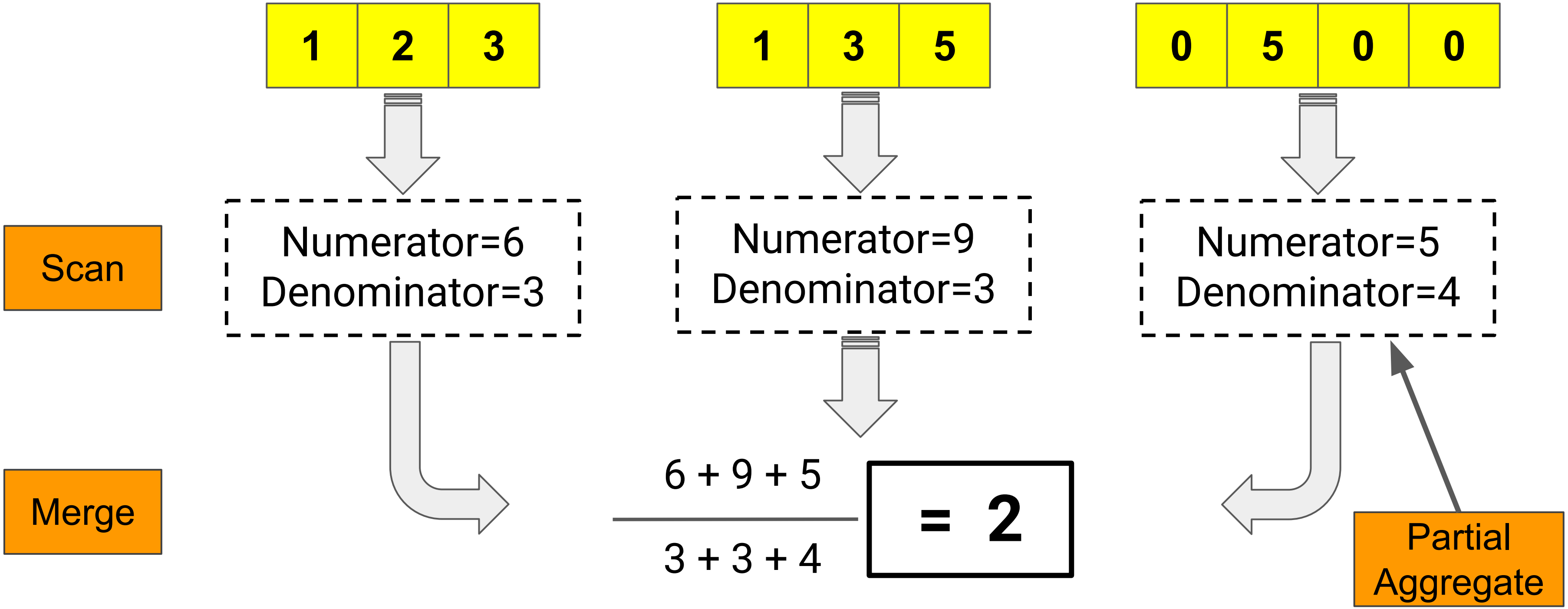
How does ClickHouse process a query with aggregates?

```
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 can you compute an average in parallel?



1	2	3

1	3	5

0	5	0	0

Scan

Numerator=6
Denominator=3

Numerator=9
Denominator=3

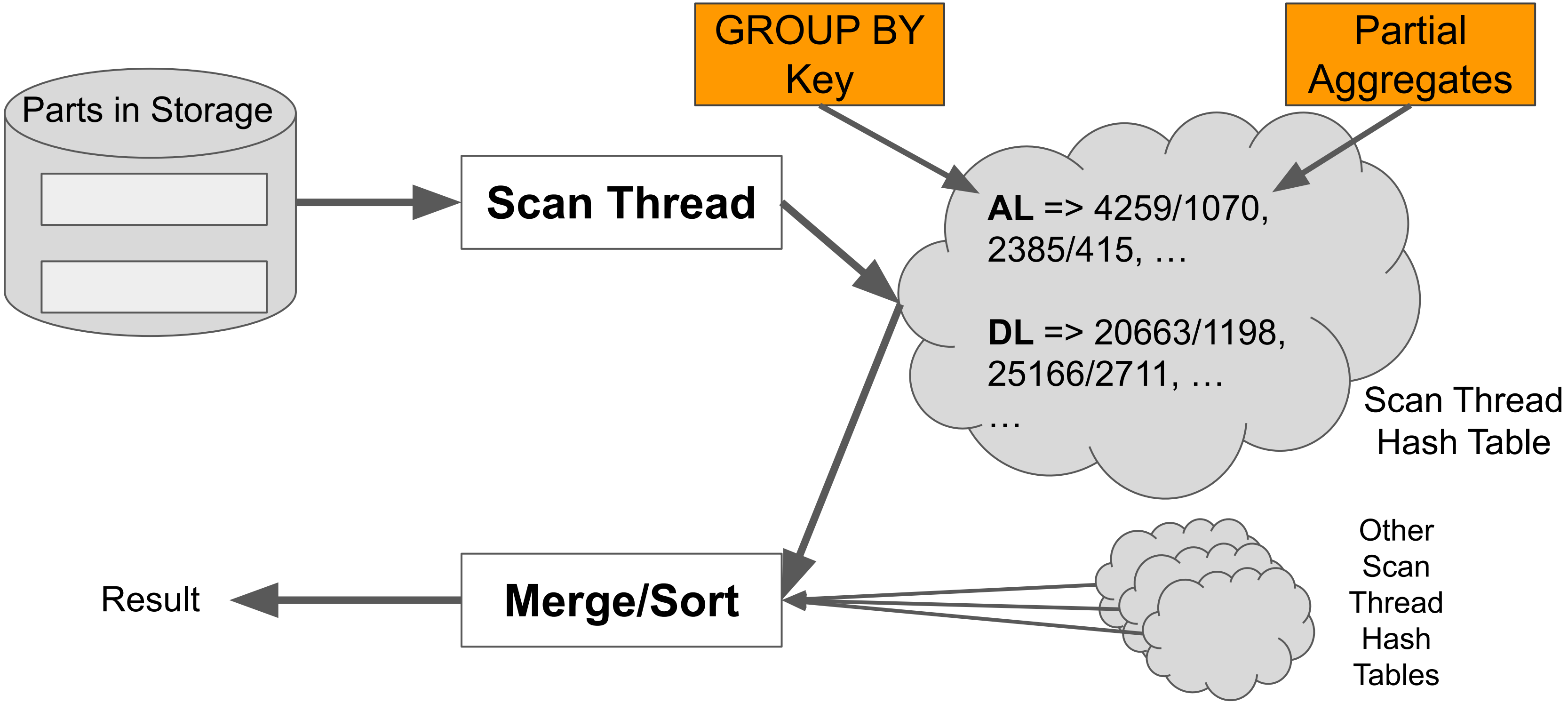
Numerator=5
Denominator=4

Merge

$$\frac{6 + 9 + 5}{3 + 3 + 4} \boxed{= 2}$$

Partial Aggregate

How does a ClickHouse thread do aggregation?


GROUP BY Key
Partial Aggregates
Parts in Storage
Scan Thread
AL => 4259/1070, 2385/415, …
DL => 20663/1198, 25166/2711, …
…
Scan Thread Hash Table
Other Scan Thread Hash Tables
Result
Merge/Sort

We can now understand aggregation performance drivers

```
0.84 sec
1.6 KB RAM
```

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

Simple aggregate, short GROUP BY key with few values

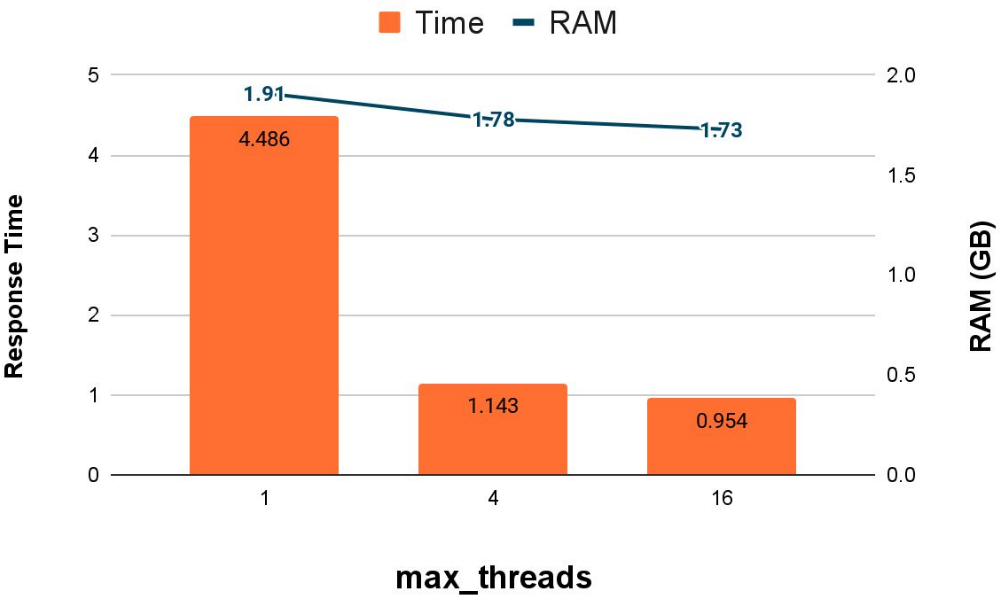
```
3.4 sec
2.4 GB RAM
```

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

More complex aggregates, longer GROUP BY with more values

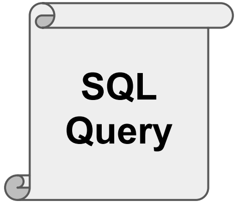
Parallelism affects speed and memory usage

```
SELECT Origin, FlightDate,
  avg(DepDelay) AS Delay,
  uniqExact(TailNum) AS Aircraft
FROM ontime
WHERE Carrier='WN'
GROUP BY Origin, FlightDate
ORDER BY Delay DESC
LIMIT 5

SET max_threads = 1
. . .
SET max_threads = 4
. . .
SET max_threads = 16
```



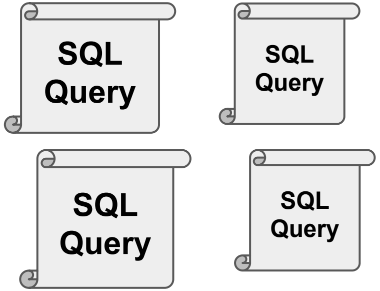
This is a good time to mention ClickHouse memory limits

Single query limit



max_memory_usage
(Default=10Gb)

All queries for a user



max_memory_usage_for_user
(Default=Unlimited)

All memory on server

ClickHouse
Process

max_server _memory_usage
(Default=90% of available RAM)

© 2022 Altinity, Inc.

Tips to make aggregation queries faster

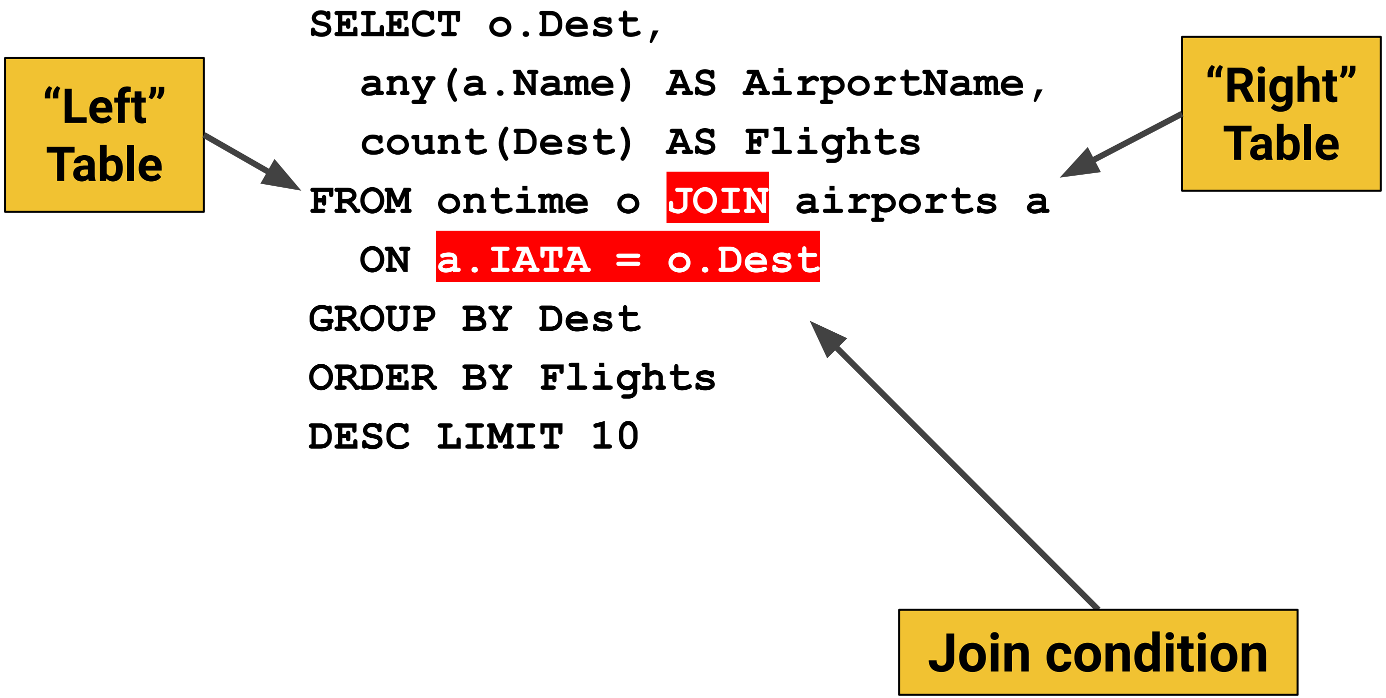
- Remove/exchange “heavy” aggregation functions
- Reduce the number of values in GROUP BY
- Increase max_threads (parallelism)
- Reduce I/O
  - Filter out unnecessary rows
  - Improve compression of data in storage

Tips to reduce memory usage in aggregation queries

- Remove/exchange “heavy” aggregation functions
- Reduce number of values in GROUP BY
- Change max_threads value
- Dump aggregates to external storage
    - SET max_bytes_before_external_group_by > 0
- Filter out unnecessary rows

How do joins work?

JOIN combines data between tables



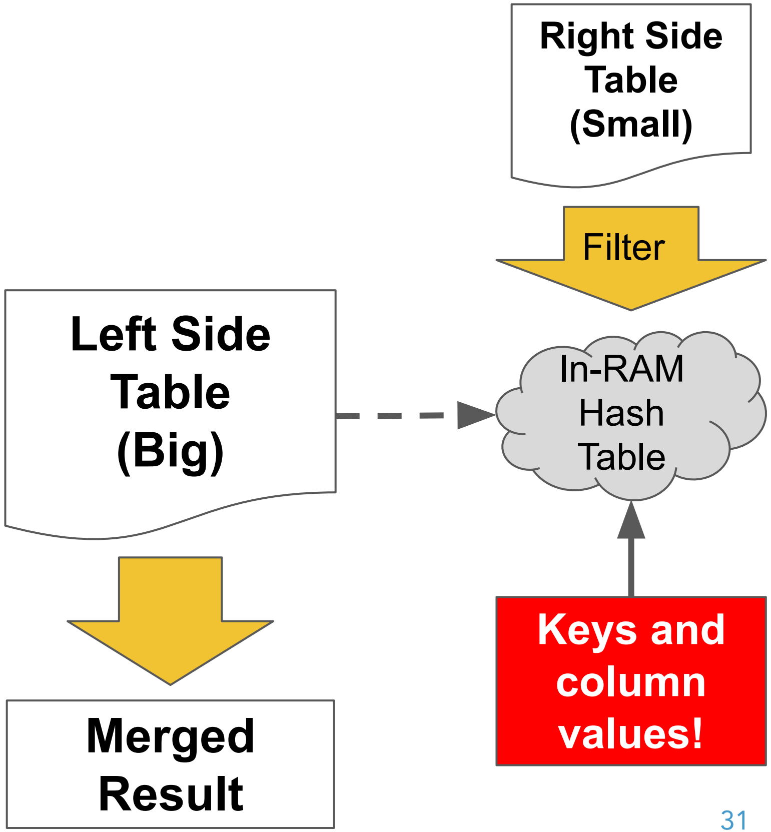
```
SELECT o.Dest,
  any(a.Name) AS AirportName,
  count(Dest) AS Flights
FROM ontime o JOIN airports a
  ON a.IATA = o.Dest
GROUP BY Dest
ORDER BY Flights
DESC LIMIT 10
```

How does ClickHouse process a query with a join?

Load the right side table

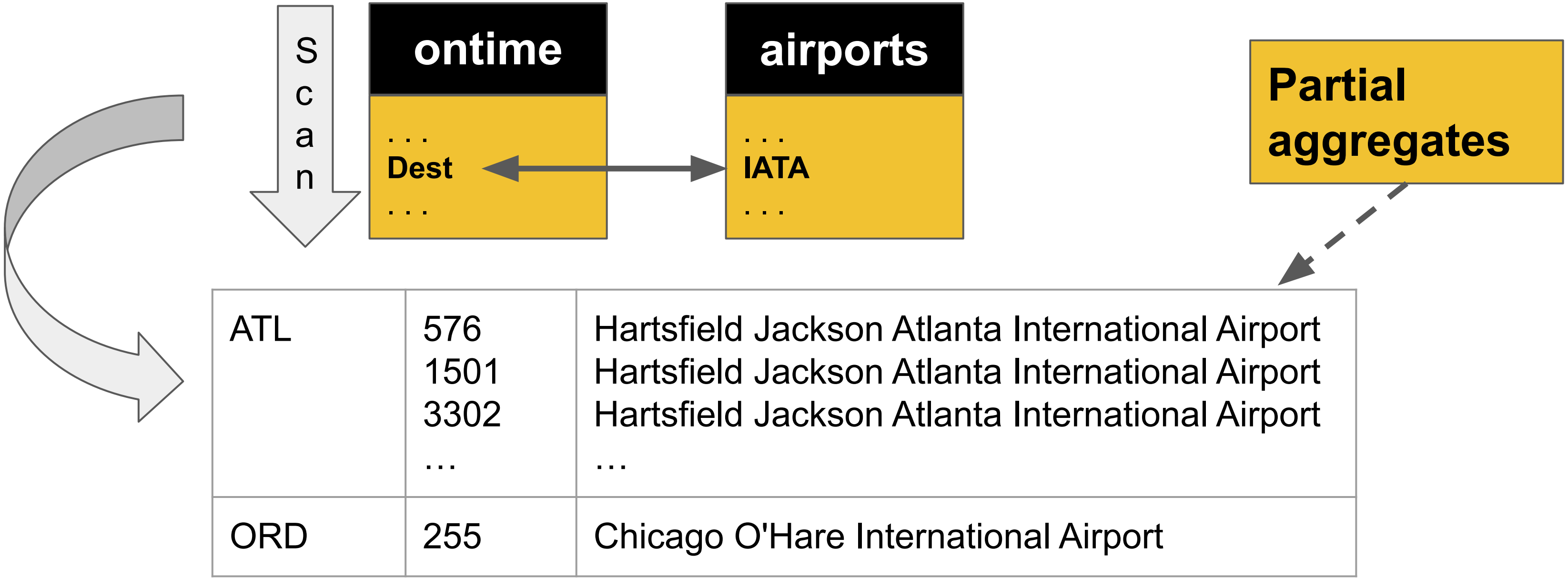
Scan left side table in parallel, computing aggregates and adding joined data

Merge and sort results



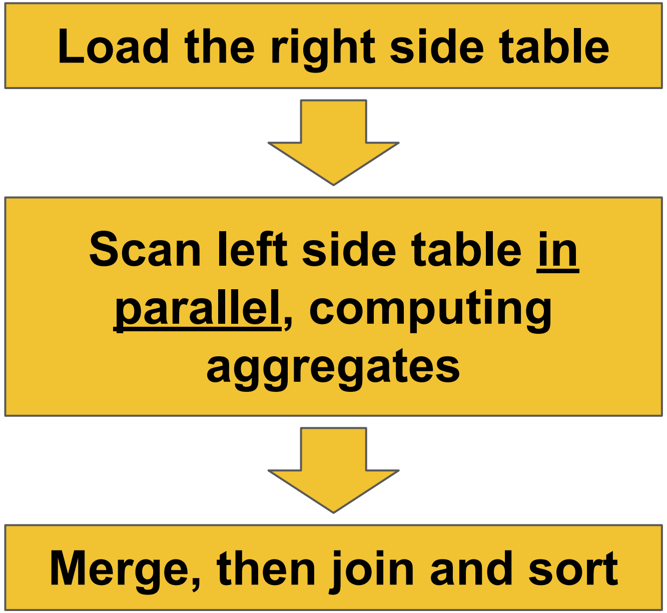
Let’s look more deeply at what’s happening in the scan

```
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

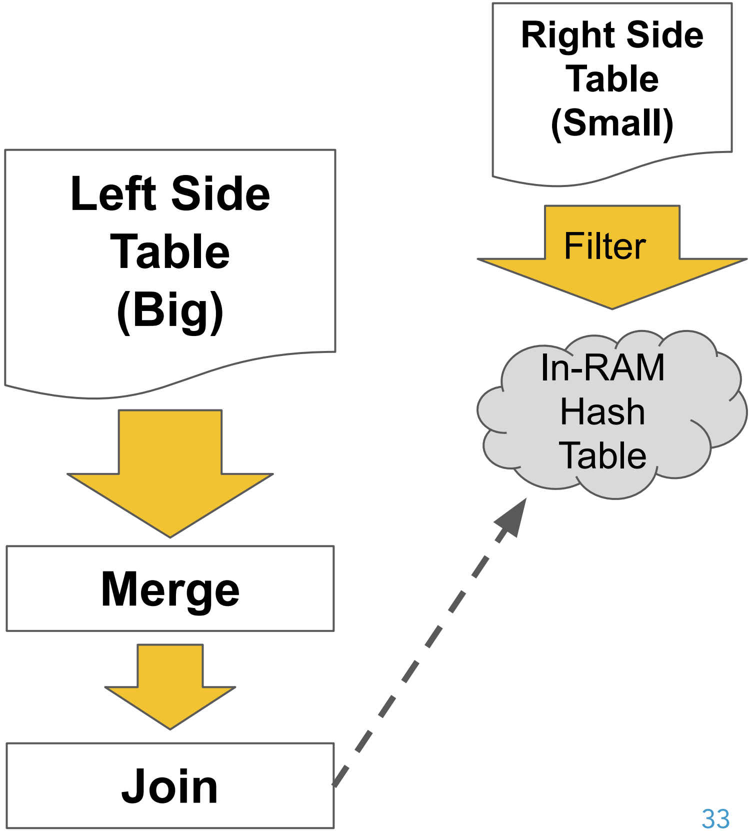
It would be more efficient to join after aggregating



Load the right side table

Scan left side table in parallel, computing aggregates

Merge, then join and sort



You can do exactly that with a subquery

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

2.71 sec
19.9 MB RAM

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

0.663 sec
1.58 KB RAM

Altinity

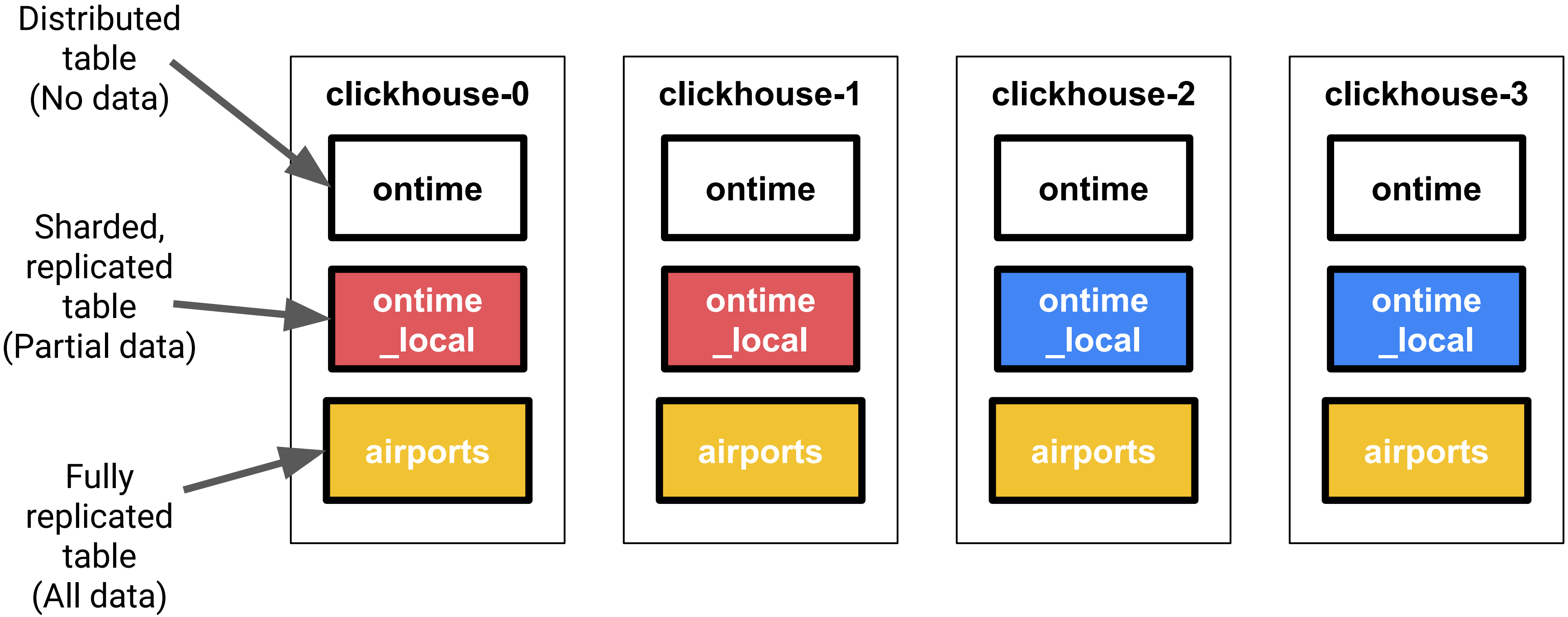
Simple ways to keep JOINs fast and efficient

- Keep the right side table(s) overall size small
- Minimize the columns joined from the right side
- Add filter conditions to the right side table to reduce rows
- JOIN after aggregation if possible
- Use a Dictionary instead of a JOIN
  - Dictionaries are just loaded once and can be shared across queries

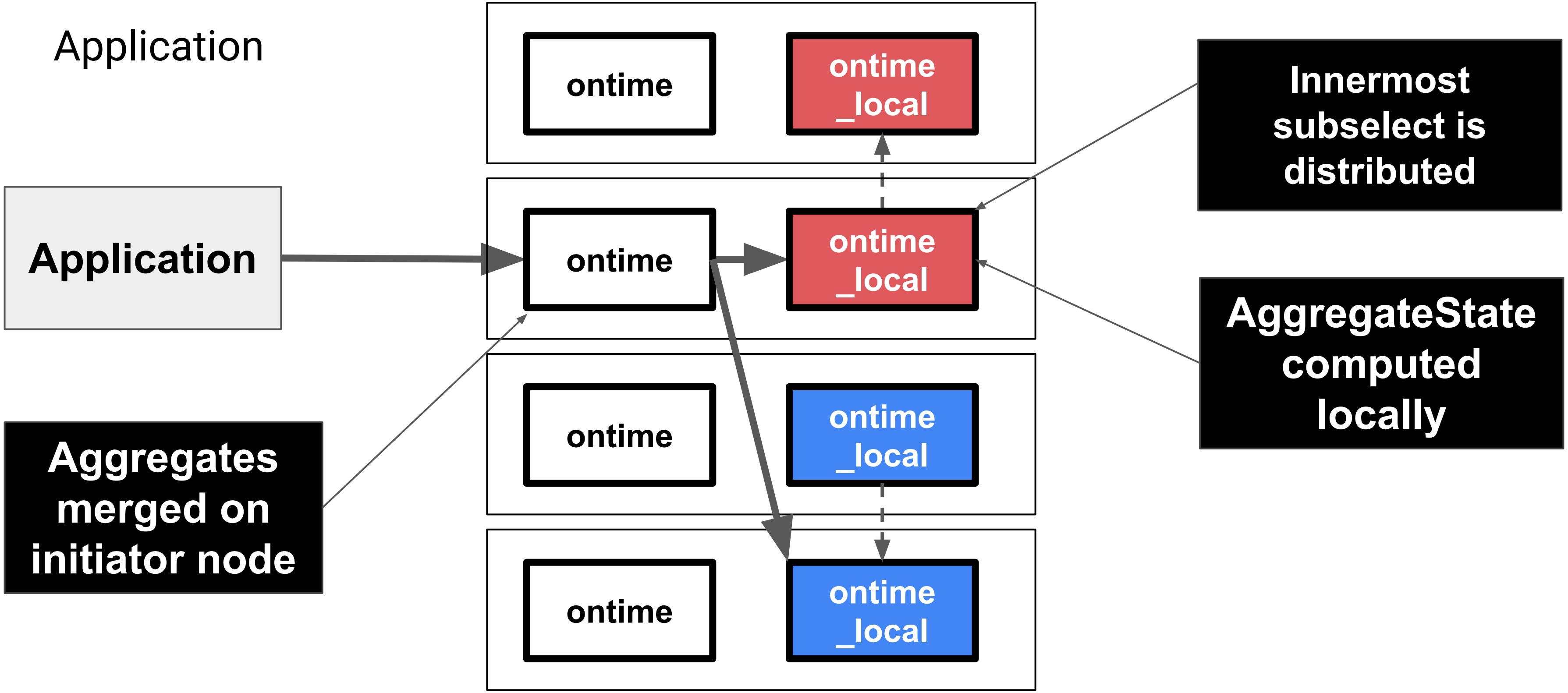
Pro tip: The SQL **`IN`** operator is also a join under the covers.

How does a distributed query work?

Example of a distributed data set with shards and replicas


Distributed table (No data)
Sharded, replicated table (Partial data)
Fully replicated table (All data)
clickhouse-0
clickhouse-1
clickhouse-2
clickhouse-3
ontime
ontime_local
airports

Distributed send subqueries to multiple nodes



Innermost subselect is distributed

AggregateState computed locally

Aggregates merged on initiator node

© 2022 Altinity, Inc.

Queries are pushed to all shards

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

↓

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

Altinity

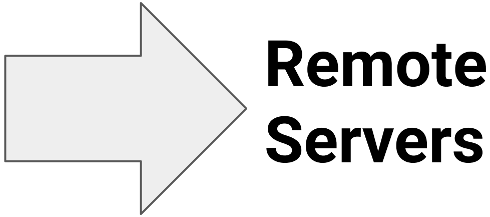
ClickHouse pushes down JOINs by default

```
SELECT o.Dest d, a.Name n, count(*) c, avg(o.ArrDelayMinutes) ad
  FROM default.ontime o
    JOIN default.airports a ON (a.IATA = o.Dest)
    GROUP BY d, n HAVING c > 100000 ORDER BY d DESC
      LIMIT 10
```

```
SELECT Dest AS d, Name AS n, count() AS c, avg(ArrDelayMinutes) AS
ad
  FROM default.ontime_local AS o
    ALL INNER JOIN default.airports AS a ON a.IATA = o.Dest
      GROUP BY d, n HAVING c > 100000 ORDER BY d DESC LIMIT 10
```

…Unless the left side “table” is a subquery

```
SELECT d, Name n, c AS flights, ad
FROM
(
  SELECT Dest d, count(*) c, avg(ArrDelayMinutes) ad
    FROM default.ontime
      GROUP BY d HAVING c > 100000
        ORDER BY ad DESC
) AS o
LEFT JOIN airports ON airports.IATA = o.d
LIMIT 10
```



Remote Servers

Altinity

It’s more complex when multiple tables are distributed

```
select foo from T1 where a in (select a from T2)
```

distributed_product_mode=?

local

```
select foo
from T1_local
where a in (
   select a
   from T2_local)
```

(Subquery runs on local table)

allow

```
select foo
from T1_local
where a in (
   select a
   from T2)
```

(Subquery runs on distributed table)

global

```
create temporary table
tmp Engine = Set
AS select a from T2;

select foo from
T1_local where a in
tmp;
```

(Subquery runs on initiator; broadcast to local temp table)

Tips to make distributed queries more efficient

- Think about where your data are located
- Move WHERE and heavy grouping work to left hand side of join
- Use a subquery to order joins after the remote scan
- Use the query_log to see what actually executes on the remote node(s)

Where to learn more

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!

References for this talk

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

ClickHouse Source Code – https://github.com/ClickHouse/ClickHouse

Talks and Blog Articles -

- ClickHouse Deep Dive, Alexey Milovidov
- Про JOIN'ы (в ClickHouse) - Artyem Zuikov
- Модификаторы DISTINCT и ORDER BY для всех агрегатных функций - Sofia Sergeevna Borzenkova
- ClickHouse Kernel Analysis - Storage Structure and Query Acceleration of MergeTree - Alibaba Cloud

Thank you!
Questions?

https://altinity.com