



Altinity WEBINAR

# MORE SECRETS OF CLICKHOUSE QUERY PERFORMANCE

with Robert Hodges



# Brief Intros



Robert Hodges - Altinity CEO

30+ years on DBMS plus  
virtualization and security.

ClickHouse is DBMS #20



# Altinity

[www.altinity.com](http://www.altinity.com)

Leading software and services  
provider for ClickHouse

Major committer and community  
sponsor in US and Western Europe

# Goals of the talk

- Understand single node MergeTree structure
- Improve response by tuning queries
- Get much bigger gains by changing data layout
- Increase storage performance with new multi-disk volumes

## Non-Goals:

- Boost performance of sharded/replicated clusters
- Teach advanced ClickHouse performance management

# ClickHouse & MergeTree Intro

# Introduction to ClickHouse

Understands SQL

Runs on bare metal to cloud

Shared nothing architecture

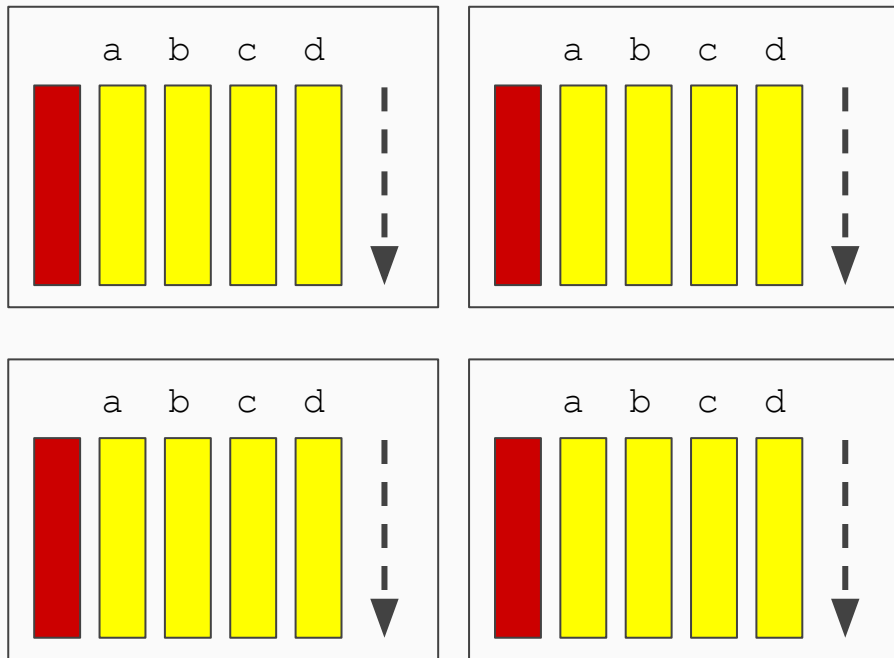
Stores data in columns

Parallel and vectorized execution

Scales to many petabytes

Is Open source (Apache 2.0)

**And it's really fast!**



# Introducing the MergeTree table engine

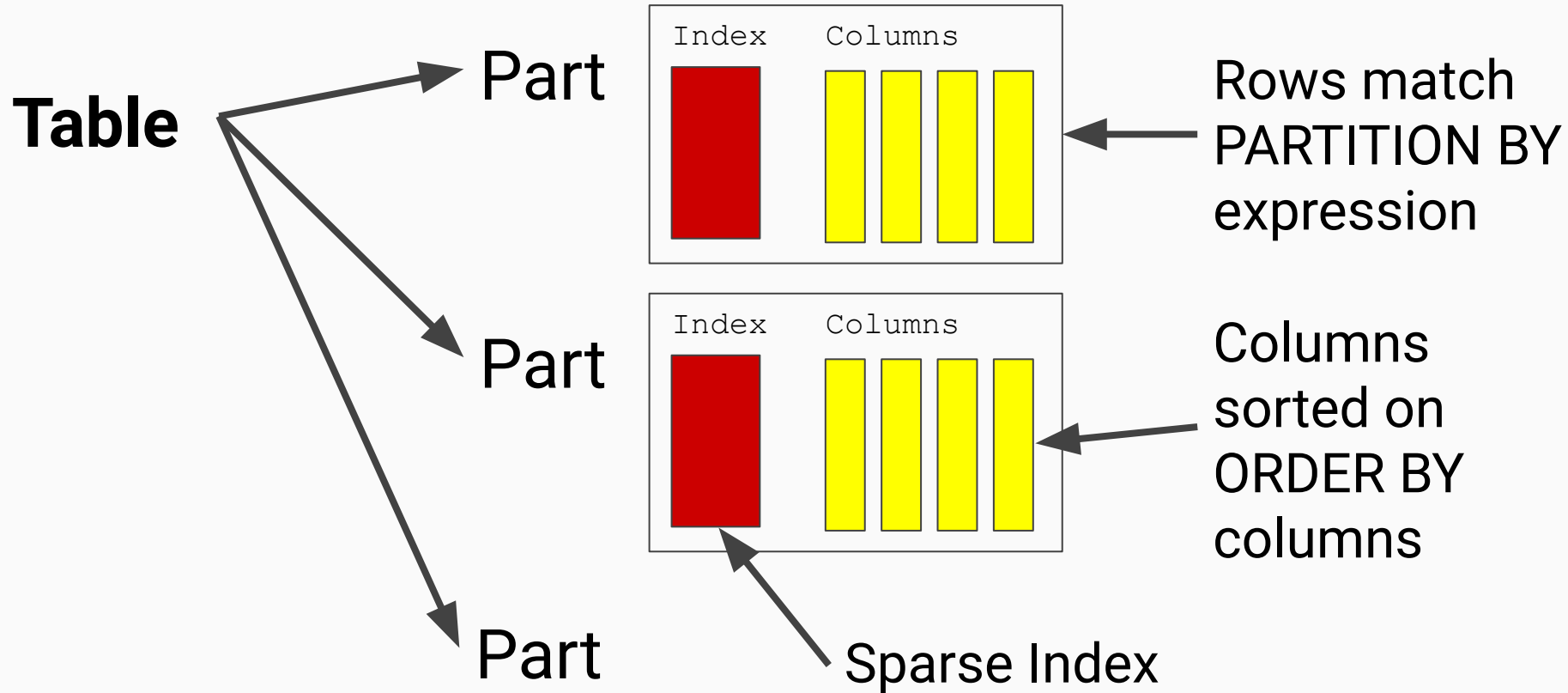
```
CREATE TABLE ontime (  
    Year UInt16,  
    Quarter UInt8,  
    Month UInt8,  
    ...  
) ENGINE = MergeTree()  
PARTITION BY toYYYYMM(FlightDate)  
ORDER BY (Carrier, FlightDate)
```

Table engine type

How to break data  
into parts

How to index and  
sort data in each part

# Basic MergeTree data layout



# MergeTree layout within a single part

`/var/lib/clickhouse/data/airline/ontime_reordered`

`20170701_20170731_355_355_2/`

`(FlightDate, Carrier...)`

`ActualElapsedTime Airline`

`AirlineID...`

`primary.idx`

2017-01-01	AA
2017-01-01	EV
2018-01-01	UA
2018-01-02	AA
...	

`.mrk`

`.bin`

`.mrk`

`.bin`

`.mrk`

`.bin`



**Granule**

**Mark**

**Compressed  
Block**



# Basic Query Tuning

# ClickHouse performance tuning is different...

## The bad news...

- No query optimizer
- No EXPLAIN PLAN
- May need to move [a lot of] data for performance

## The good news...

- No query optimizer!
- System log is great
- System tables are too
- Performance drivers are simple: I/O and CPU
- Constantly improving

# Your friend: the ClickHouse query log

```
clickhouse-client --send_logs_level=trace
```

**Return messages to  
clickhouse-client**

```
select * from system.text_log
```

**Must enable in  
config.xml**

```
sudo less \  
/var/log/clickhouse-server/clickhouse-server.log
```

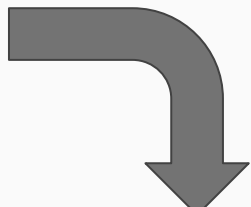
**View all log  
messages on server**

# Use system log to find out query details

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

**Query pipeline in log**

**8 parallel threads  
to read table**



(Log messages)  
Limit  
Expression  
MergeSorting  
PartialSorting  
Expression  
ParallelAggregating  
Expression × 8  
MergeTreeThread

# Speed up query executing by adding threads

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

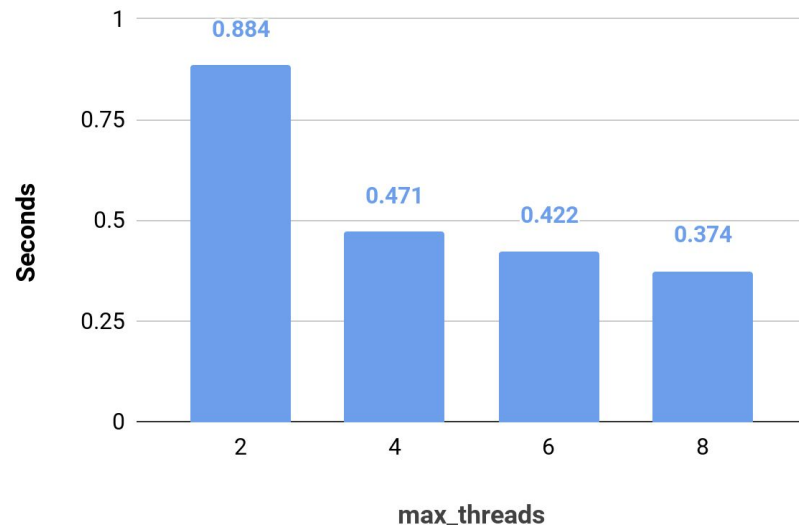
```
SET max_threads = 2
```

```
SET max_threads = 4
```

```
. . .
```

**max\_threads defaults to half the number of physical CPU cores**

Query Response



# Speed up queries by reducing reads

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

(Log messages)

Selected **355 parts by date**,  
355 parts by key,  
**21393 marks to read from 355  
ranges**

```
SELECT toYear(FlightDate) year,  
       sum(Cancelled)/count(*) cancelled,  
       sum(DepDel15)/count(*) delayed_15  
FROM airline.ontime  
WHERE year =  
toYear(toDate('2016-01-01'))  
GROUP BY year ORDER BY year LIMIT 10
```

(Log messages)

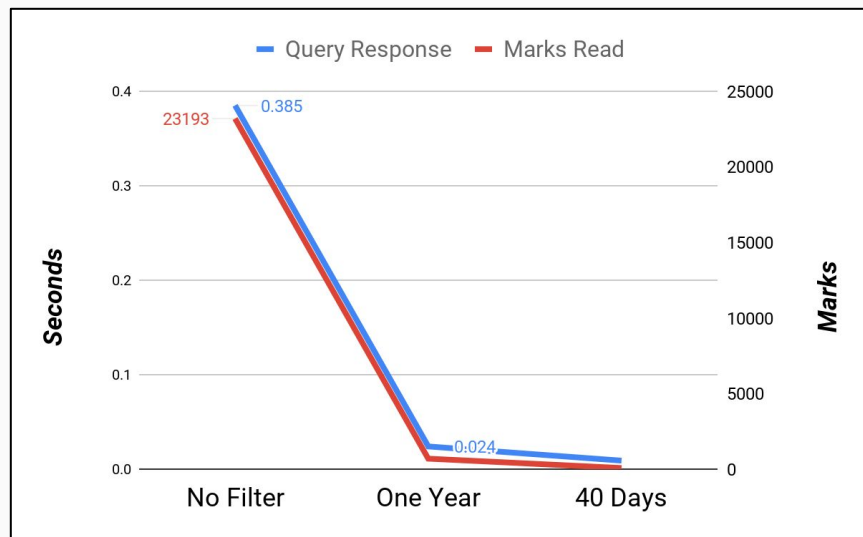
Selected **12 parts by date**,  
12 parts by key,  
**692 marks to read from 12  
ranges**

# Query execution tends to scale with I/O

```
SELECT
  FlightDate,
  count(*) AS total_flights,
  sum(Cancelled) / count(*) AS cancelled,
  sum(DepDel15) / count(*) AS delayed_15
FROM airline.ontime
WHERE (FlightDate >= toDate('2016-01-01'))
      AND (FlightDate <= toDate('2016-02-10'))
GROUP BY FlightDate
```

(Log messages)

Selected **2 parts by date**,  
2 parts by key,  
**73 marks to read from 2 ranges**



# Use PREWHERE to help filter unindexed data

```
SELECT
  Year, count(*) AS total_flights,
  count(distinct Dest) as destinations,
  count(distinct Carrier) as carriers,
  sum(Cancelled) / count(*) AS cancelled,
  sum(DepDel15) / count(*) AS delayed_15
FROM airline.ontime [PRE]WHERE Dest = 'BLI' GROUP BY Year
```

(PREWHERE Log messages)

Elapsed: **0.591 sec.**

Processed 173.82 million rows,  
**2.09 GB** (294.34 million rows/s.,  
3.53 GB/s.)

(WHERE Log messages)

Elapsed: **0.816 sec.**

Processed 173.82 million rows,  
**5.74 GB** (213.03 million rows/s.,  
7.03 GB/s.)



# But PREWHERE can kick in automatically

```
SET optimize_move_to_prewhere = 1
```

This is the default value

```
SELECT
```

```
  Year, count(*) AS total_flights,  
  count(distinct Dest) as destinations,  
  count(distinct Carrier) as carriers,  
  sum(Cancelled) / count(*) AS cancelled,  
  sum(DepDel15) / count(*) AS delayed_15
```

```
FROM airline.ontime
```

```
WHERE Dest = 'BLI' GROUP BY Year
```

(Log messages)

InterpreterSelectQuery:

MergeTreeWhereOptimizer: condition

"Dest = 'BLI'" moved to PREWHERE

# Restructure joins to reduce data scanning

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

**3.878  
seconds**

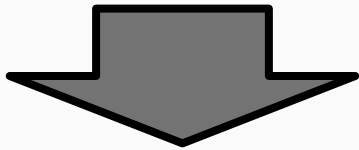


```
SELECT dest, Name n, c AS flights, ad FROM (
  SELECT Dest dest, count(*) c, avg(ArrDelayMinutes) ad
  FROM ontime
    GROUP BY dest HAVING c > 100000
    ORDER BY ad DESC
) LEFT JOIN airports ON airports.IATA = dest LIMIT 10
```

**1.177  
seconds**

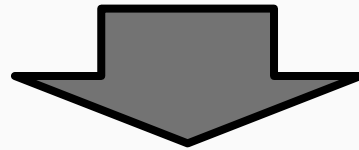
# The log tells the story

Join during  
MergeTree scan



```
(Log messages)
ParallelAggregatingBlockInputStream
: Total aggregated. 173790727 rows
  (from 10199.035 MiB) in 3.844 sec.
  (45214666.568 rows/sec., 2653.455
MiB/sec.)
```

Join after  
MergeTree scan



```
(Log messages)
ParallelAggregatingBlockInputStream
: Total aggregated. 173815409 rows
  (from 2652.213 MiB) in 1.142 sec.
  (152149486.717 rows/sec., 2321.617
MiB/sec.)
```

# More ways to find out about queries

```
SET log_queries = 1
```

Run a query

```
SELECT version()
```

```
SET log_queries = 0
```

```
SELECT * FROM system.query_log  
WHERE query='SELECT version()'
```

**Start query logging**

**Stop query logging**

```
SHOW PROCESSLIST
```

**Show currently  
executing queries**

# Optimizing Data Layout

# Restructure data for big performance gains

- Ensure optimal number of parts
- Optimize primary key index and ordering to reduce data size and index selectivity
- Use skip indexes to avoid unnecessary I/O
- Use encodings to reduce data size before compression
- Use materialized views to transform data outside of source table
- Plus many other tricks

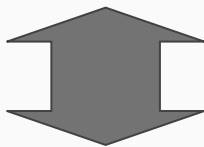
# How do partition keys affect performance?

```
CREATE TABLE ontime ...
```

```
ENGINE=MergeTree()
```

```
PARTITION BY
```

```
toYYYYMM(FlightDate)
```



```
CREATE TABLE ontime_many_parts
```

```
...
```

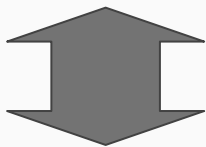
```
ENGINE=MergeTree()
```

```
PARTITION BY FlightDate
```

Is there a  
practical  
difference?

# Keep parts in the hundreds, not thousands

```
CREATE TABLE ontime ...  
ENGINE=MergeTree()  
PARTITION BY  
toYYYYMM(FlightDate)
```



```
CREATE TABLE ontime_many_parts  
...  
ENGINE=MergeTree()  
PARTITION BY FlightDate
```

SELECT count() performance

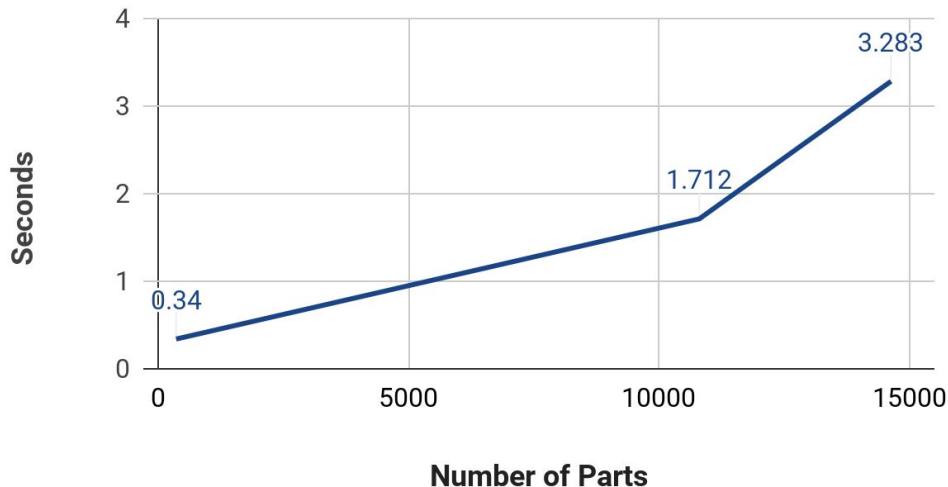


Table	Rows	Parts
ontime	174M	355
ontime_many_parts (after OPTIMIZE)	174M	10,085
ontime_many_parts (before OPTIMIZE)	174M	14,635



# Think about primary key index structure

```
CREATE TABLE ontime_reordered (  
  Year UInt16,  
  Quarter` UInt8,  
  . . .)  
ENGINE = MergeTree()  
PARTITION BY toYYYYMM(FlightDate)  
ORDER BY (Carrier, Origin, FlightDate)  
SETTINGS index_granularity=8192
```

Hashing large values  
allows index to fit in  
memory more easily

Large granularity  
makes index smaller

Small granularity can make  
skip indexes more selective

# Table ORDER BY is key to performance

```
CREATE TABLE ontime_reordered (  
    Year UInt16,  
    Quarter` UInt8,  
    . . .)  
ENGINE = MergeTree()  
PARTITION BY toYYYYMM(FlightDate)  
ORDER BY (Carrier, Origin, FlightDate)  
SETTINGS index_granularity=8192
```

**Choose order to make  
dependent non-key  
values less random**

## **Benefits:**

- Higher compression
- Better index selectivity
- Better PREWHERE performance

# Skip indexes cut down on I/O

```
SET allow_experimental_data_skipping_indices=1;
```

Default value



```
ALTER TABLE ontime ADD INDEX
```

```
dest_name Dest TYPE ngrambf_v1(3, 512, 2, 0) GRANULARITY 1
```

```
ALTER TABLE ontime ADD INDEX
```

```
cname Carrier TYPE set(0) GRANULARITY 1
```

```
OPTIMIZE TABLE ontime FINAL
```

```
-- OR, in current releases
```

```
ALTER TABLE ontime
```

```
UPDATE Dest=Dest, Carrier=Carrier
```

```
WHERE 1=1
```

# Indexes & PREWHERE remove granules

**Apply PREWHERE  
on Dest predicate**

```
(Log messages)
InterpreterSelectQuery: MergeTreeWhereOptimizer:
condition "Dest = 'PPG'" moved to PREWHERE
. . .
(SelectExecutor): Index `dest_name` has dropped 55
granules.
(SelectExecutor): Index `dest_name` has dropped 52
granules.
```

**Use index to remove  
granules from scan**

# Effectiveness depends on data distribution

```
SELECT
  Year, count(*) AS flights,
  sum(Cancelled) / flights AS cancelled,
  sum(DepDel15) / flights AS delayed_15
FROM airline.ontime WHERE [Column] = [Value] GROUP BY Year
```

Column	Value	Index	Count	Rows Processed	Query Response
Dest	PPG	ngrambf_v1	525	4.30M	0.053
Dest	ATL	ngrambf_v1	9,360,581	166.81M	0.622
Carrier	ML	set	70,622	3.39M	0.090
Carrier	WN	set	25,918,402	166.24M	0.566


# Current index types

Name	What it tracks
minmax	High and low range of data; good for numbers with strong locality like timestamps
set	Unique values
ngrambf_v1	Presence of character ngrams, works with =, LIKE, search predicates; good for long strings
tokenbf_v1	Like ngram but for whitespace-separated strings; good for searching on tags
bloomfilter	Presence of value in column

# Encodings improve compression efficiency

```
CREATE TABLE test_codecs ( a String,  
    a_lc LowCardinality(String) DEFAULT a,  
    b UInt32,  
    b_delta UInt32 DEFAULT b Codec(Delta) ,  
    b_delta_lz4 UInt32 DEFAULT b Codec(Delta, LZ4) ,  
    b_dd UInt32 DEFAULT b Codec(DoubleDelta) ,  
    b_dd_lz4 UInt32 DEFAULT b Codec(DoubleDelta, LZ4)  
)  
Engine = MergeTree  
PARTITION BY tuple() ORDER BY tuple();
```

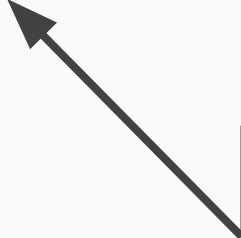
**Values with  
dictionary  
encoding**



**Differences  
between  
values**

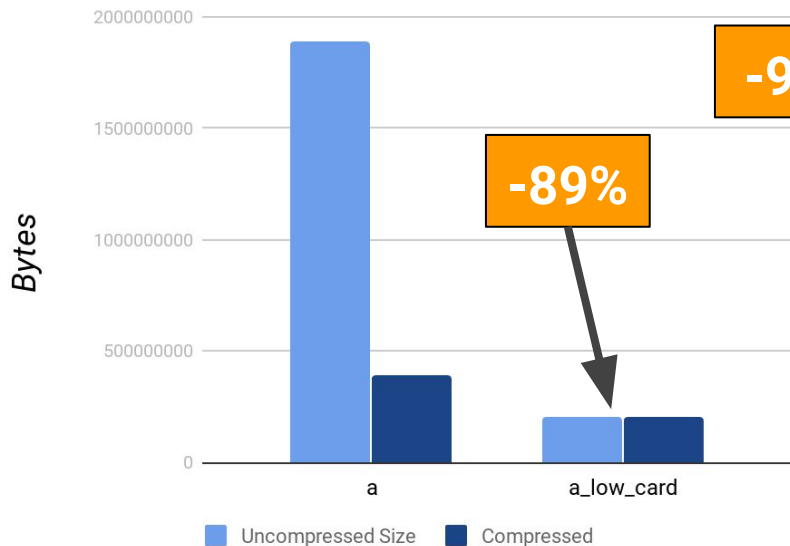


**Differences  
between change  
of value**

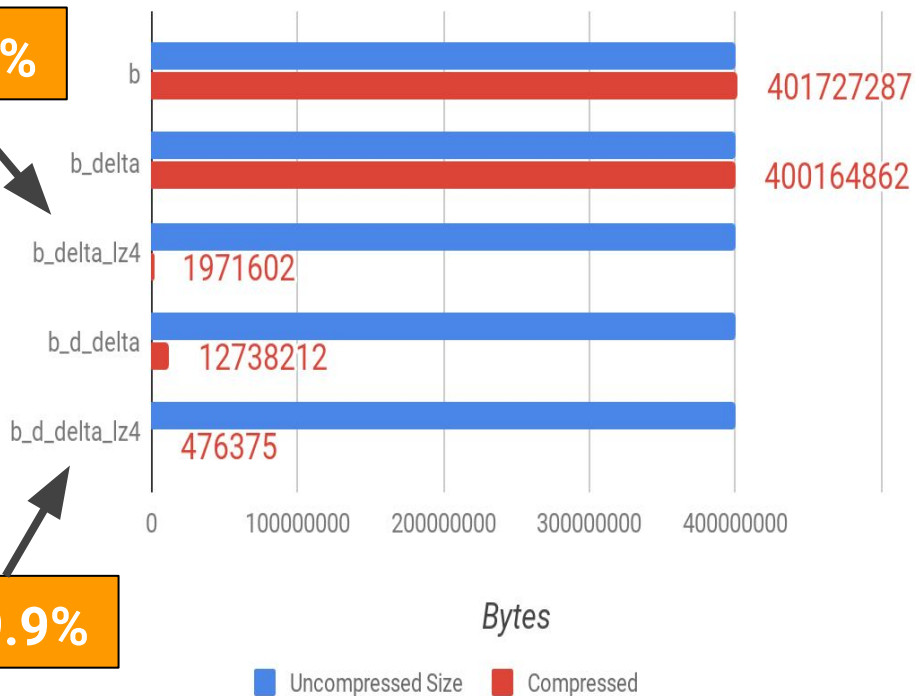


# Effect on storage size is dramatic

## Low Cardinality Encoding



## Numeric Encodings





# Queries are faster due to less I/O

```
SELECT a AS a, count(*) AS c FROM test_codec  
GROUP BY a ORDER BY c ASC LIMIT 10
```

. . .

10 rows in set. Elapsed: **0.681 sec**. Processed 100.00 million  
rows, **2.69 GB** (146.81 million rows/s., 3.95 GB/s.)



```
SELECT a_lc AS a, count(*) AS c FROM test_codec  
GROUP BY a ORDER BY c ASC LIMIT 10
```

. . .

10 rows in set. Elapsed: **0.148 sec**. Processed 100.00 million  
rows, **241.16 MB** (675.55 million rows/s., 1.63 GB/s.)

# Overview of encodings

Name	Best for
LowCardinality	Strings with fewer than 10K values
Delta	Time series
Double Delta	Increasing counters
Gorilla	Gauge data (bounces around mean)
T64	Integers other than random hashes

**Compression may vary across ZSTD and LZ4**

# TIP: use system.columns to check data size

```
SELECT table,  
       formatReadableSize(sum(data_compressed_bytes)) tc,  
       formatReadableSize(sum(data_uncompressed_bytes)) tu,  
       sum(data_compressed_bytes) / sum(data_uncompressed_bytes) as ratio  
FROM system.columns  
WHERE database = currentDatabase()  
GROUP BY table ORDER BY table
```

# Use mat views to boost performance further

```
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
```

Returns cancelled/late flights where Carrier = 'WN' in 0.007 seconds

# More things to think about

Use smaller datatypes wherever possible

Use ZSTD compression (slower but better ratio)

Use dictionaries instead of joins

Use sampling when approximate answers are acceptable

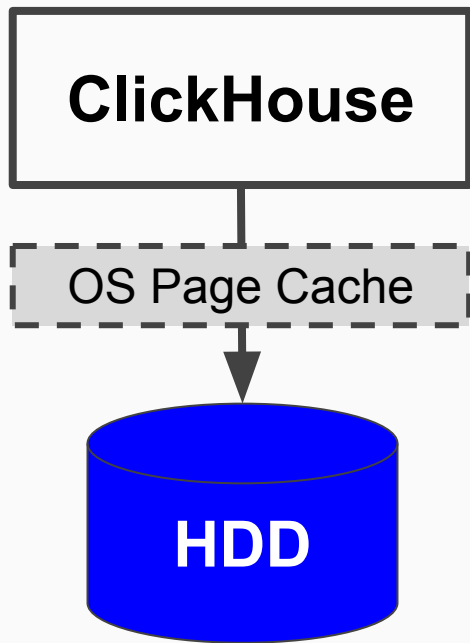
Shard/replicate data across a cluster for large datasets

**Check out “Further Resources”  
slide for more information**

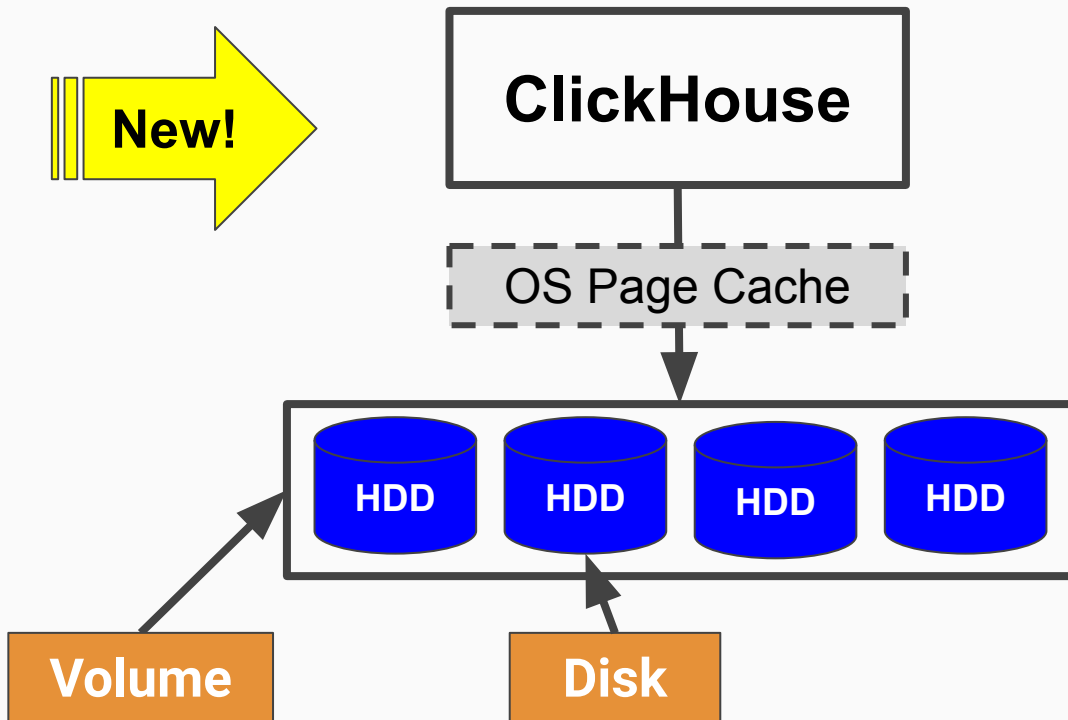
# Thinking about Storage and Memory

# ClickHouse now has flexible storage policies

Default Disk Storage



JBOD Storage



# How do you apply storage policies?

```
CREATE TABLE.tripdata
(
  `pickup_date` Date DEFAULT toDate(tpep_pickup_datetime),
  ...
)
ENGINE = MergeTree()
PARTITION BY toYYYYMM(pickup_date)
ORDER BY (pickup_location_id, dropoff_location_id, vendor_id)
SETTINGS storage_policy = 'ebs_jbod_4',
         index_granularity = 8192
```



Policy

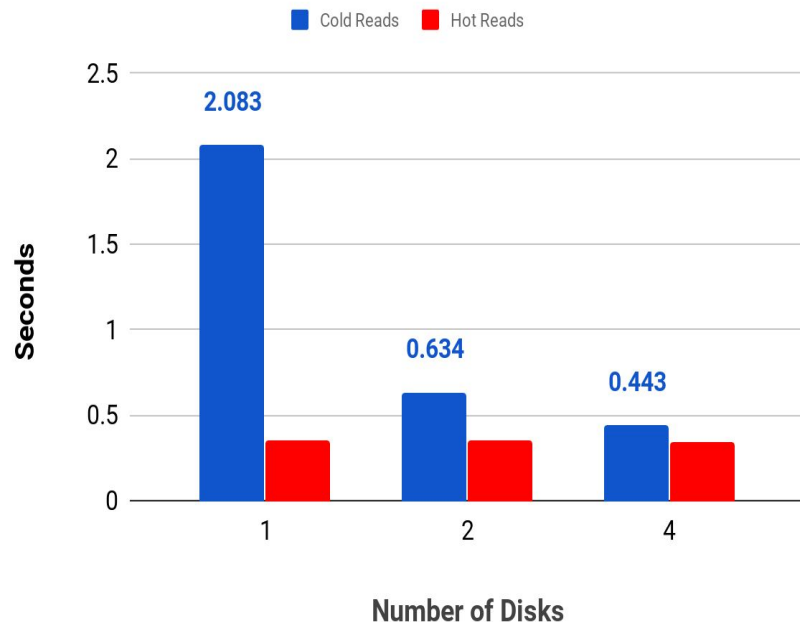


# Effect of storage policies on response

```
-- Cold query
set min_bytes_to_use_direct_io=1
SELECT avg(passenger_count)
FROM tripdata
```

```
-- Hot query
set min_bytes_to_use_direct_io=0
SELECT avg(passenger_count)
FROM tripdata
. . .
SELECT avg(passenger_count)
FROM tripdata
```

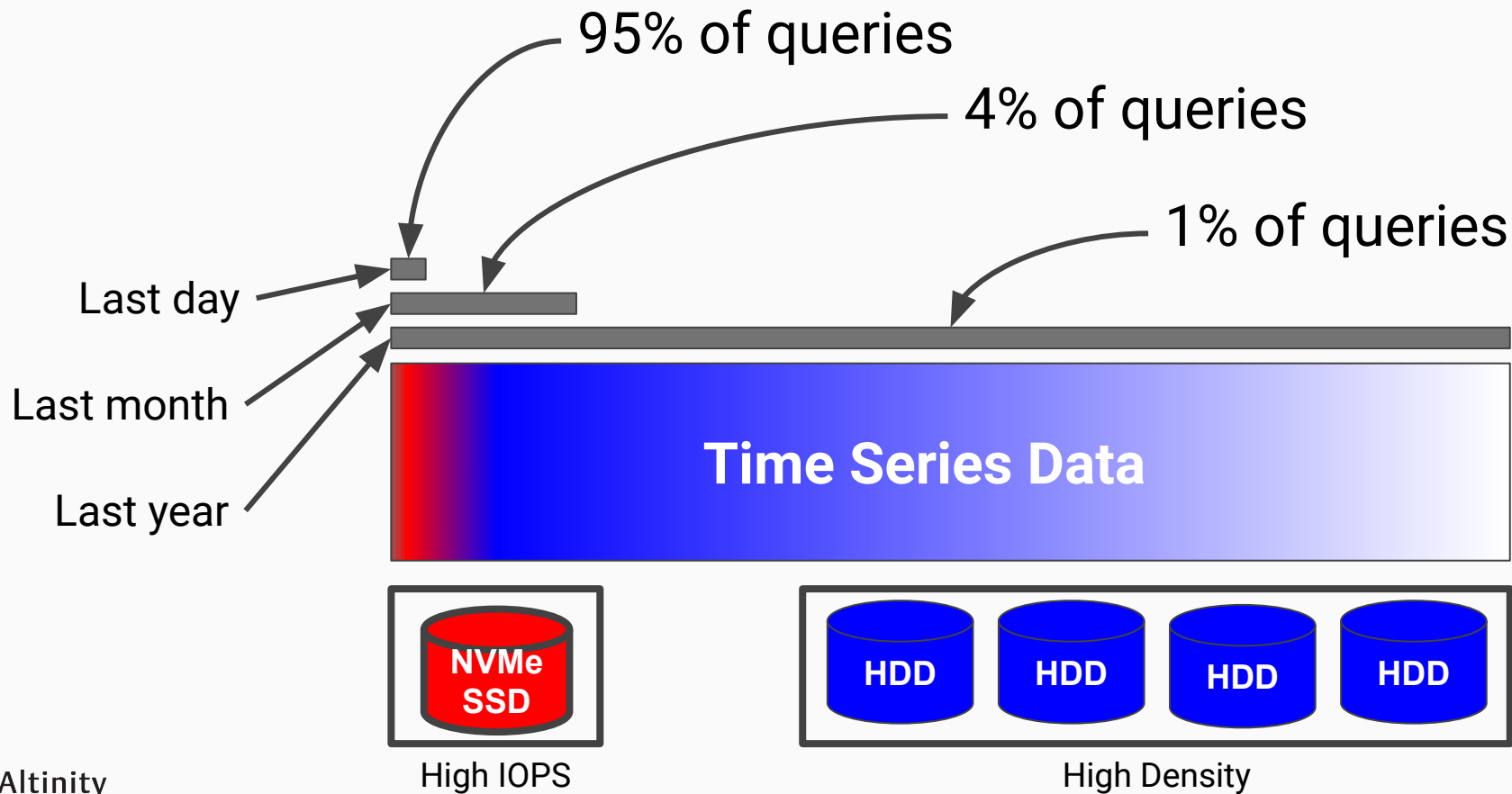
Query Response on AWS EBS



# TIP: system.parts tracks content across disks

```
SELECT
  database, table, disk_name,
  count(*) AS parts,
  uniq(partition) AS partitions,
  sum(marks) AS marks,
  sum(rows) AS rows,
  formatReadableSize(sum(data_compressed_bytes)) AS compressed,
  formatReadableSize(sum(data_uncompressed_bytes)) AS uncompressed,
  round(sum(data_compressed_bytes) / sum(data_uncompressed_bytes) * 100.0, 2)
  AS percentage
FROM system.parts
WHERE active and database = currentDatabase()
GROUP BY database, table, disk_name
ORDER BY database ASC, table ASC, disk_name ASC
```

# Tiered storage is another new option



# And don't forget all the great OS utilities!

- top and htop -- CPU and memory
- dstat -- I/O and network consumption
- iostat -- I/O by device
- iotop -- I/O by process
- iftop -- Network consumption by host
- perf top -- CPU utilization by system function

For a full description see [Performance Analysis of ClickHouse Queries](#) by Alexey Milovidov

# Wrap-up

# Takeaways on ClickHouse Performance

- ClickHouse performance drivers are CPU and I/O
- The system query log is key to understanding performance
- Query optimization can improve response substantially
- Restructure tables and add indexes/mat views for biggest gains
- In recent versions you can now optimize storage, too!

# Further resources

- [Altinity Blog](#), especially:
  - [Amplifying ClickHouse Capacity with Multi-Volume Storage](#)
- [Altinity Webinars](#), especially:
  - [ClickHouse Materialized Views: The Magic Continues](#)
  - [Strength in Numbers: Introduction to ClickHouse Cluster Performance](#)
- [ClickHouse documentation](#)
- [Performance Analysis of ClickHouse Queries](#) by Alexey Milovidov
- ClickHouse Telegram Channel
- ClickHouse Slack Channel

# Thank you!

## Special Offer:

Contact us for a  
1-hour consultation!

Contacts:

[info@altinity.com](mailto:info@altinity.com)

Visit us at:

<https://www.altinity.com>

Free Consultation:

<https://blog.altinity.com/offer>