



# Altinity

## A Fast Intro to Fast Query with ClickHouse

Robert Hodges, Altinity CEO



Altinity

[www.altinity.com](http://www.altinity.com)  
[info@altinity.com](mailto:info@altinity.com)

## Altinity Background

- Premier provider of software and services for ClickHouse
- Incorporated in UK with distributed team in US/Canada/Europe
- Main US/Europe sponsor of ClickHouse community
- Offerings:
  - Enterprise support for ClickHouse and ecosystem projects
  - Software (Kubernetes, cluster manager, tools & utilities)
  - POCs/Training

The shape of data has changed

Business insights are hidden in massive pools of automatically collected information



Applications that rule the digital era have a common success factor

The ability to discover and apply  
business-critical insights  
from petabyte datasets in real time

# Let's consider a concrete example

Web properties track clickstreams to:

- Calculate clickthrough/buy rates
- Guide ad placement
- Optimize eCommerce services

Constraints:

- Run on commodity hardware
- Simple to operate
- Fast interactive query
- Avoid encumbering licenses

# Existing analytic databases do not meet requirements fully



Google BigQuery

Cloud-native data warehouses cannot operate on-prem, limiting range of solutions



Microsoft®  
SQL Server®

Legacy SQL databases are expensive to run, scale poorly on commodity hardware, and adapt slowly



amazon  
EMR

cloudera



databricks®

Hadoop/Spark ecosystem solutions are resource intensive with slow response and complex pipelines



elasticsearch



TIME SCALE

Specialized solutions limit query domain and are complex/resource-inefficient for general use

# ClickHouse fills the gaps and does much more besides

Understands SQL

Runs on bare metal to cloud

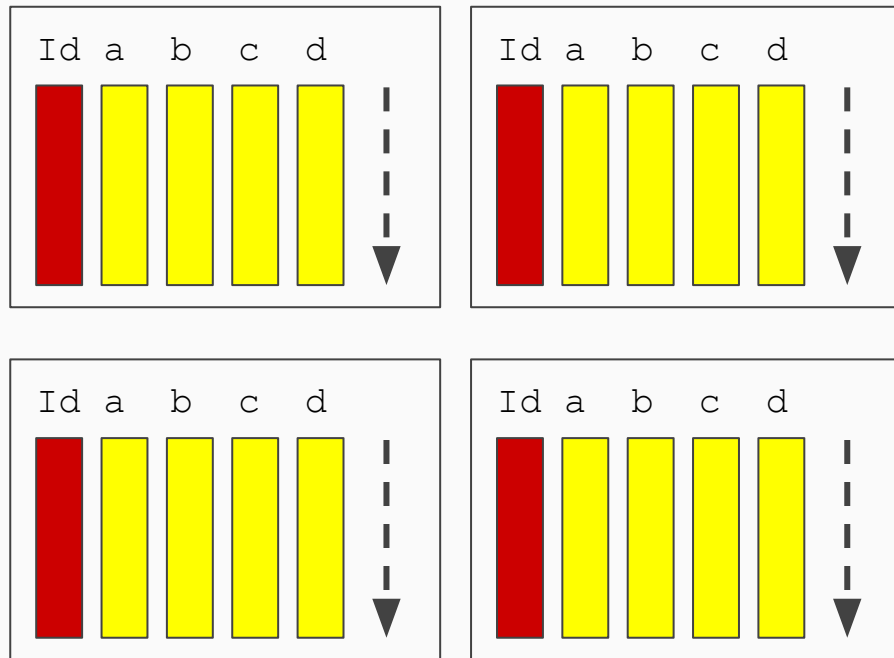
Stores data in columns

Parallel and vectorized execution

Scales to many petabytes

Is Open source (Apache 2.0)

Is WAY fast!



# What does “WAY fast” mean?

```
SELECT Dest d, count(*) c, avg(ArrDelayMinutes) ad
FROM ontime GROUP BY d HAVING c > 100000
ORDER BY ad DESC limit 5
```

d	c	ad
EWR	3660570	17.637564095209218
SFO	4056003	16.029478528492213
JFK	2198078	15.33669824273752
LGA	3133582	14.533851994299177
ORD	9108159	14.431460737565077

**5 rows in set. Elapsed: 1.182 sec. Processed 173.82 million rows, 2.78 GB (147.02 million rows/s., 2.35 GB/s.)**

(Amazon md5.2xlarge: Xeon(R) Platinum 8175M, 8vCPU, 30GB RAM, NVMe SSD)



## What are the main ClickHouse use patterns?

- Fast, scalable data warehouse for online services (SaaS and in-house apps)
- Built-in data warehouse for installed analytic applications
- Exploration -- throw in a bunch of data and go crazy!

# Getting started is easy with Docker image

```
$ docker run -d --name ch-s yandex/clickhouse-server
```

```
$ docker exec -it ch-s clickhouse client
```

```
...
```

```
11e99303c78e :) select version()
```

```
SELECT version()
```

version()
19.3.3

```
1 rows in set. Elapsed: 0.001 sec.
```

Or install recommended Altinity stable version packages

```
$ sudo apt -y install clickhouse-client=18.16.1 \  
clickhouse-server=18.16.1 \  
clickhouse-common-static=18.16.1
```

...

```
$ sudo systemctl start clickhouse-server
```

...


```
11e99303c78e :) select version()
```

```
SELECT version()
```


version()
18.16.1

```
1 rows in set. Elapsed: 0.001 sec.
```

# Examples of table creation and data insertion



```
CREATE TABLE sdata (  
    DevId Int32,  
    Type String,  
    MDate Date,  
    MDatetime DateTime,  
    Value Float64  
) ENGINE = MergeTree() PARTITION BY toYYYYMM(MDate)  
ORDER BY (DevId, MDatetime)
```




```
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)
```

```
INSERT INTO sdata VALUES  
(15, 'TEMP', '2018-01-01', '2018-01-01 23:31:53', 18.1),  
(2, 'TEMP', '2018-01-01', '2018-01-01 23:31:55', 7.9)
```

# Loading data from CSV files

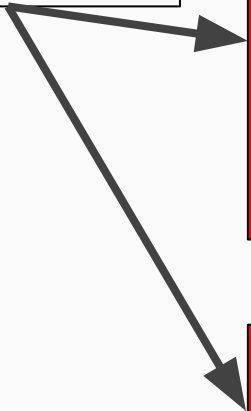
```
cat > sdata.csv <<END
DevId,Type,MDate,MDatetime,Value
59,"TEMP","2018-02-01","2018-02-01 01:10:13",19.5
59,"TEMP","2018-02-01","2018-02-01 02:10:01",18.8
59,"TEMP","2018-02-01","2018-02-01 03:09:58",18.6
59,"TEMP","2018-02-01","2018-02-01 04:10:05",15.1
59,"TEMP","2018-02-01","2018-02-01 05:10:31",12.2
59,"TEMP","2018-02-01","2018-02-01 06:10:02",11.8
59,"TEMP","2018-02-01","2018-02-01 07:09:55",10.9
END
```



```
cat sdata.csv |clickhouse-client --database foo
--query='INSERT INTO sdata FORMAT CSVWithNames'
```

# Select results can be surprising!

```
SELECT *  
FROM sdata  
WHERE  
DevId < 20
```



Result right after INSERT:

DevId	Type	MDate	MDatetime	Value
15	TEMP	2018-01-01	2018-01-01 23:29:55	18
15	TEMP	2018-01-01	2018-01-01 23:30:56	18.7

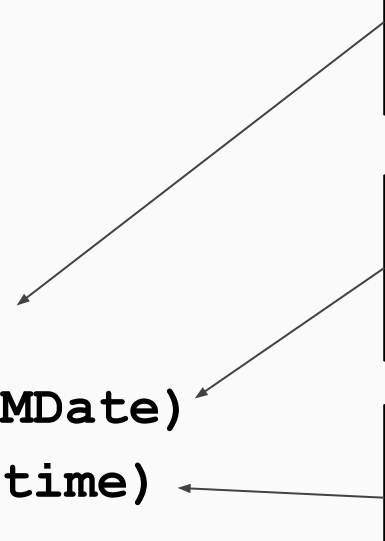
DevId	Type	MDate	MDatetime	Value
2	TEMP	2018-01-01	2018-01-01 23:31:55	7.9
15	TEMP	2018-01-01	2018-01-01 23:31:53	18.1

Result somewhat later:

DevId	Type	MDate	MDatetime	Value
2	TEMP	2018-01-01	2018-01-01 23:31:55	7.9
15	TEMP	2018-01-01	2018-01-01 23:29:55	18
15	TEMP	2018-01-01	2018-01-01 23:30:56	18.7
15	TEMP	2018-01-01	2018-01-01 23:31:53	18.1

# Time for some research into table engines

```
CREATE TABLE sdata (  
    DevId Int32,  
    Type String,  
    MDate Date,  
    MDatetime DateTime,  
    Value Float64  
) ENGINE = MergeTree()  
PARTITION BY toYYYYMM(MDate)  
ORDER BY (DevId, MDatetime)
```



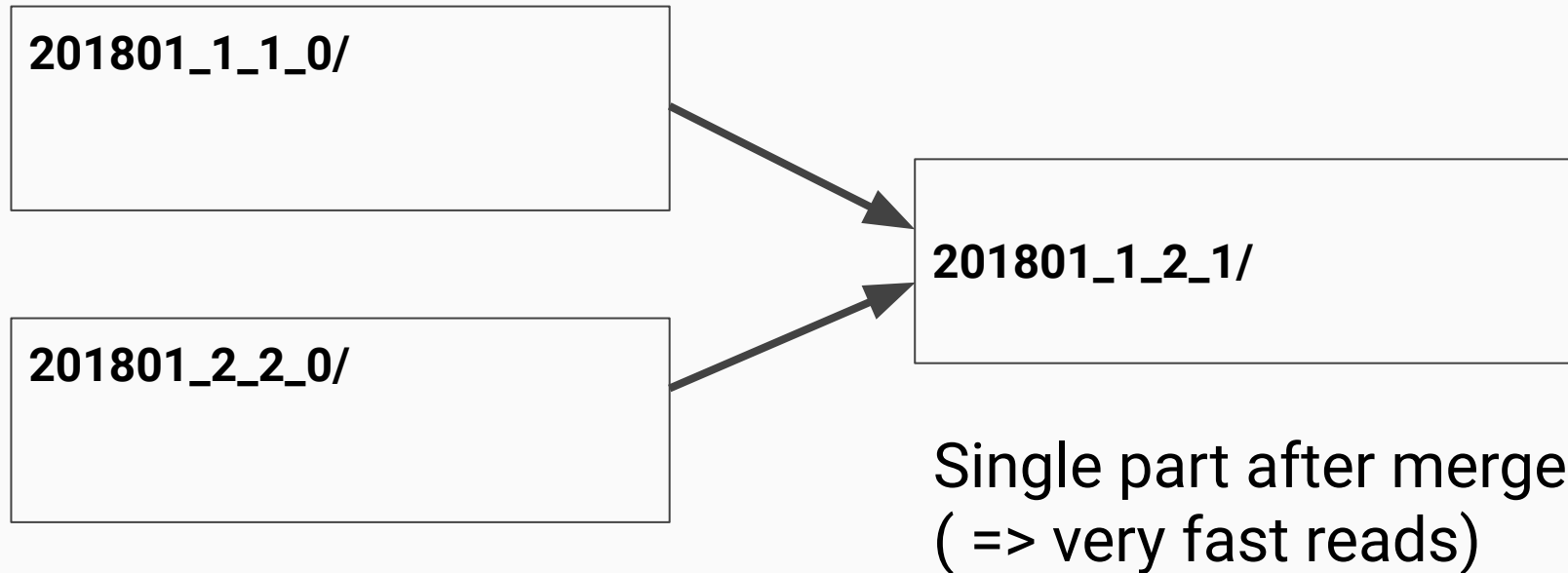
How to manage data  
and handle queries

How to break table  
into parts

How to index and  
sort data in each part

# MergeTree writes parts quickly and merges them offline

**/var/lib/clickhouse/data/default/sdata**



Multiple parts after initial  
insertion ( => very fast writes)



# Rows are indexed and sorted inside each part

**/var/lib/clickhouse/data/default/sdata**

**201801\_1\_2\_1/**

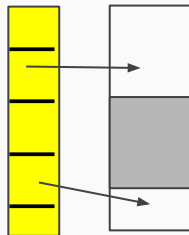
**(DevId, MDateTime)**

primary.idx

...	...
956	2018-01-01 15:22:37
575	2018-01-01 23:31:53
1300	2018-01-02 05:14:47
...	...

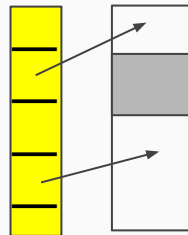
**DevId**

.mrk .bin



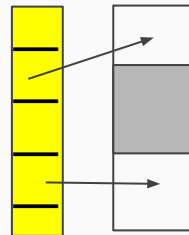
**Type**

.mrk .bin



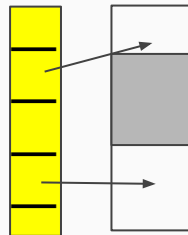
**MDate**

.mrk .bin



**MDatetime...**

.mrk .bin



**201802\_1\_1\_0/**

**(DevId, MDateTime)**

primary.idx

**DevId**

.mrk .bin

**Type**

.mrk .bin

**MDate**

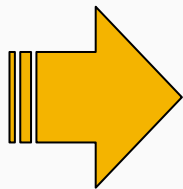
.mrk .bin

**MDatetime...**

.mrk .bin

# Now we can follow how query works on a single server

```
SELECT DevId, Type, avg(Value)
FROM sdata
WHERE MDate = '2018-01-01'
GROUP BY DevId, Type
```



## ClickHouse

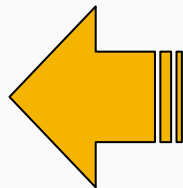
Identify parts to search



Query in parallel

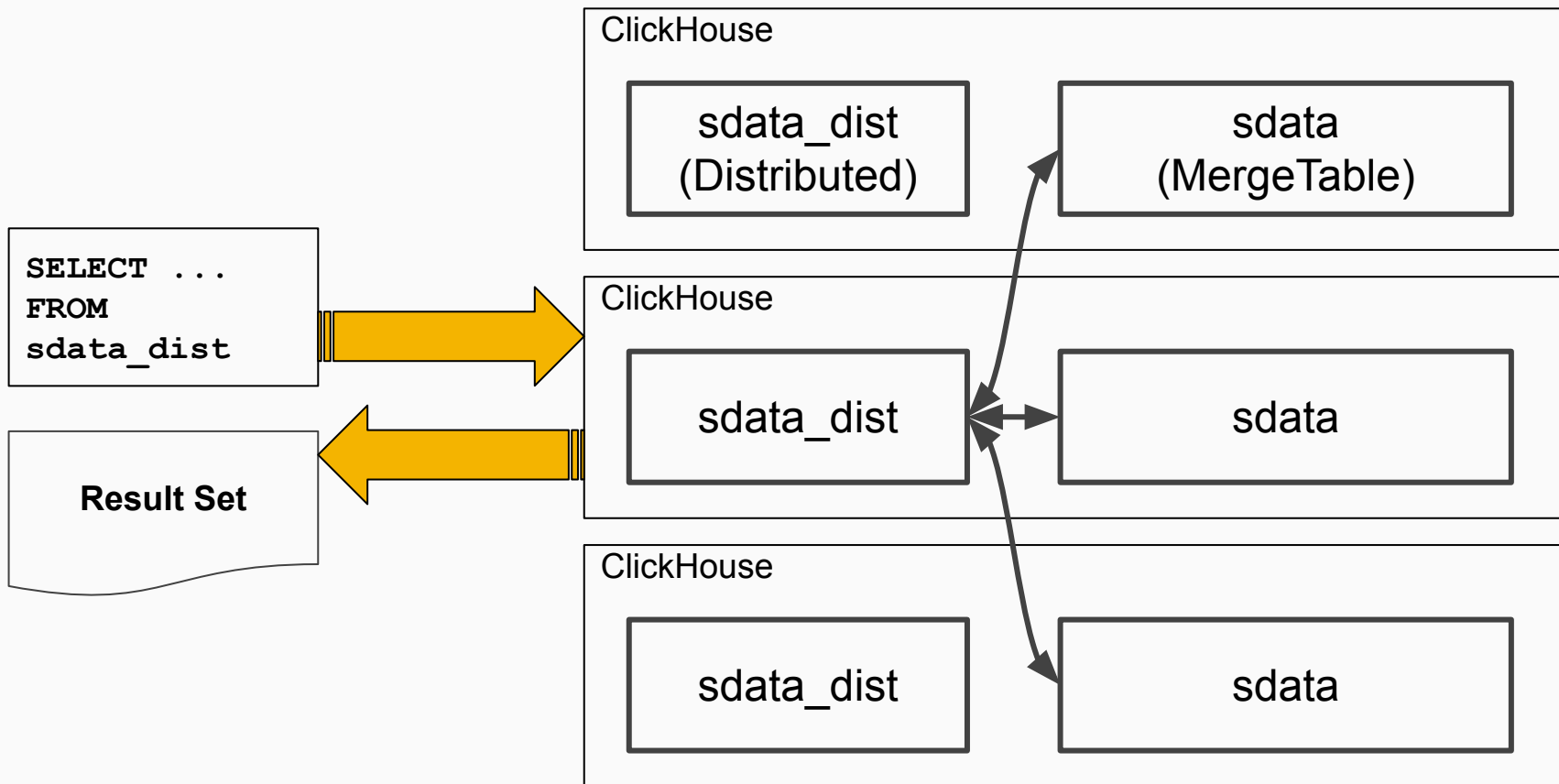


Aggregate results

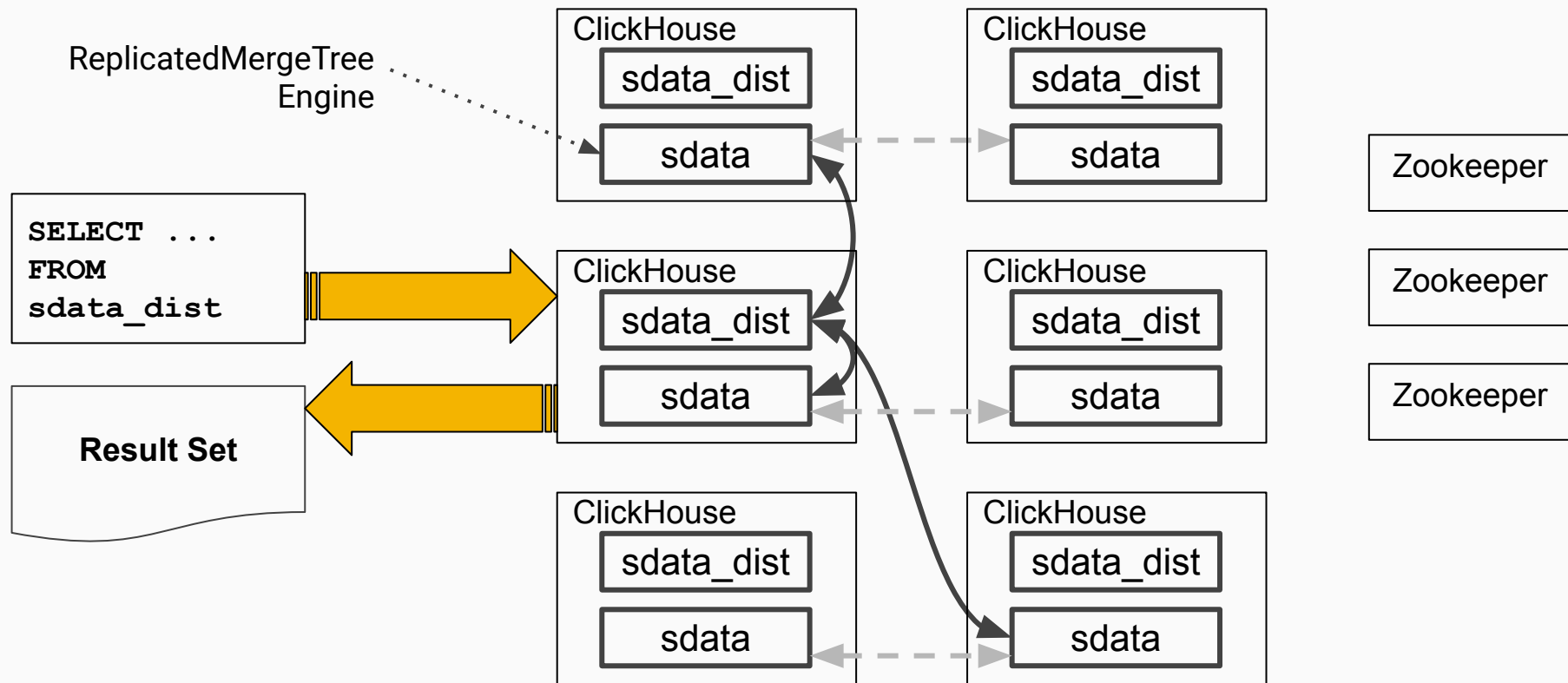


Result Set

# Clickhouse distributed engine spreads queries across shards



# ReplicatedMergeTree engine spreads over shards and replicas



# With basic engine knowledge you can now tune queries

```
SELECT Dest, count(*) c, avg(DepDelayMinutes)
FROM ontime
GROUP BY Dest HAVING c > 100000
ORDER BY c DESC limit 5
```

**Scans 355 table parts  
in parallel; does not  
use index**



```
SELECT Dest, count(*) c, avg(DepDelayMinutes)
FROM ontime
WHERE toYear(FlightDate) =
      toYear(toDate('2016-01-01'))
GROUP BY Dest HAVING c > 100000
ORDER BY c DESC limit 5
```

**Scans 12 parts (3%  
of data) because  
FlightDate is  
partition key**

**Hint: clickhouse-server.log has the query plan**

# You can also optimize joins

```
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 ad DESC
```



```
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
```

**Joins on data  
before GROUP BY,  
increased amount  
to scan**

**Subquery  
minimizes data  
scanned in  
parallel; joins on  
GROUP BY results**

# ClickHouse has a wealth of features to help queries go fast

Dictionaries

Materialized Views

Arrays

Specialized functions and SQL  
extensions

Lots more table engines

...And a nice set of supporting ecosystem tools

Client libraries: JDBC, ODBC, Python, Golang, ...

Kafka table engine to ingest from Kafka queues

Visualization tools: Grafana, Tableau, Tabix, SuperSet

Data science stack integration: Pandas, Jupyter Notebooks

Kubernetes ClickHouse operator



## Where to get more information

- ClickHouse Docs: <https://clickhouse.yandex/docs/en/>
- Altinity Blog: <https://www.altinity.com/blog>
- Meetups and conference presentations
  - 2 April – Madrid, Spain ClickHouse Meetup
  - 28-30 May -- Austin, TX Percona Live 2019
  - San Francisco ClickHouse Meetup

Questions?

Thank you!

Contacts:

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

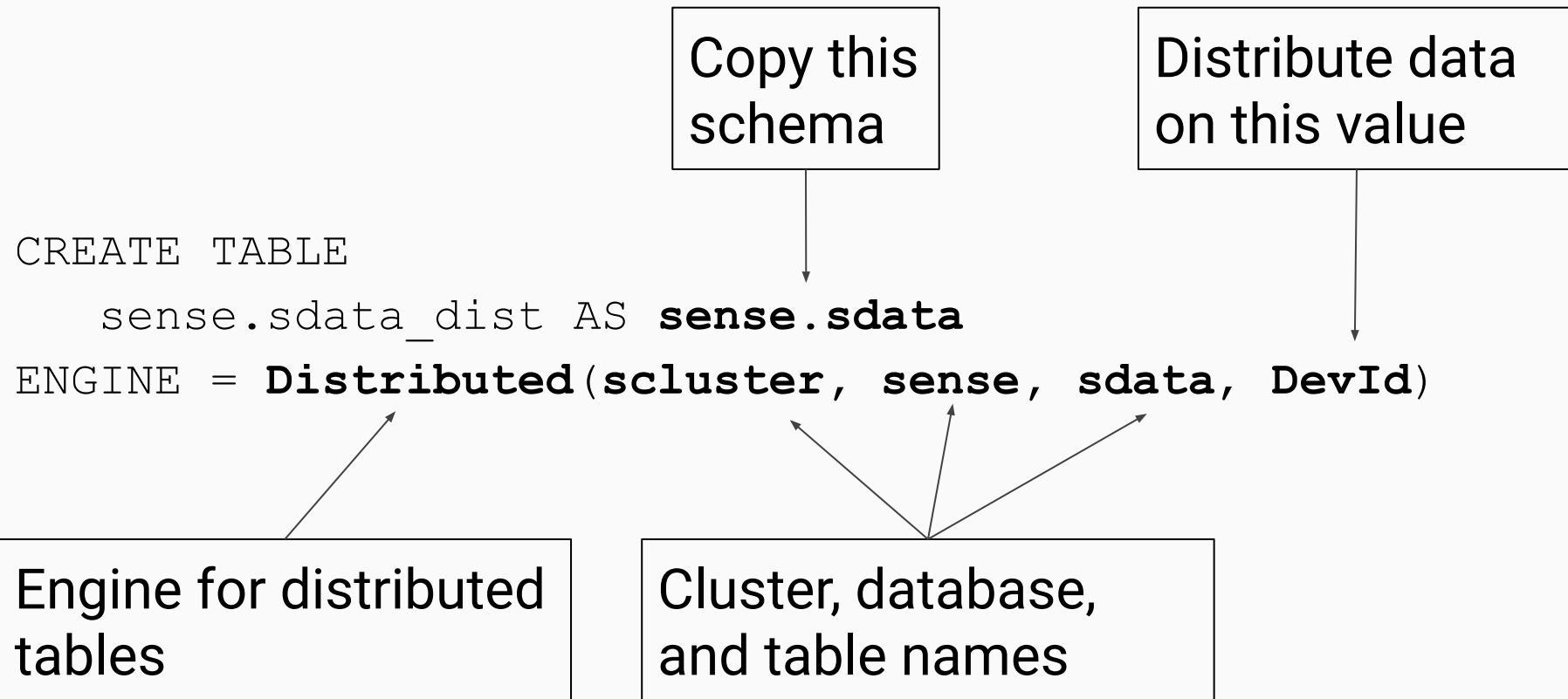
Visit us at:

<https://www.altinity.com>

Read Our Blog:

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

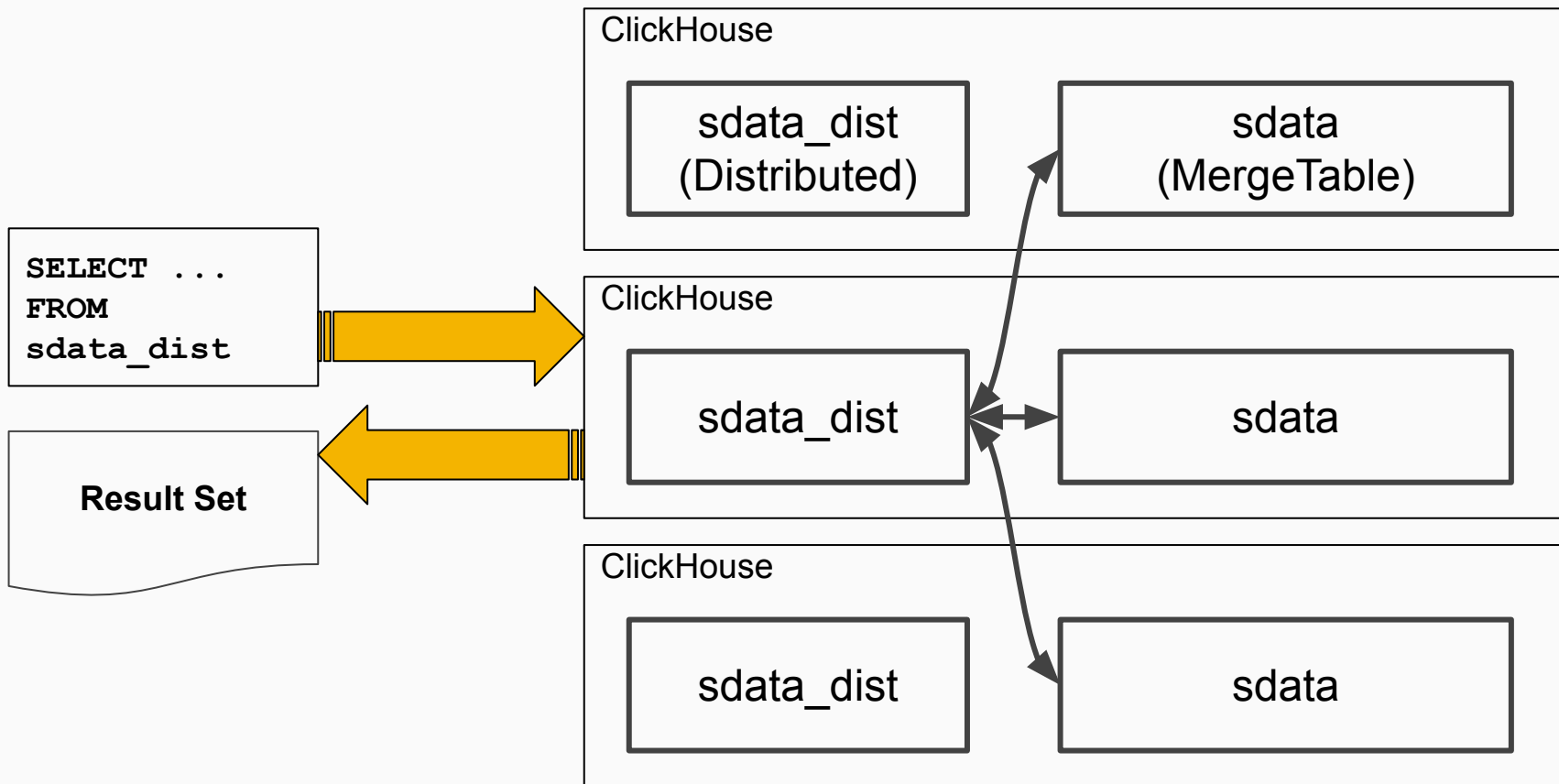
# What if you have a lot of data?



# What's a Cluster? (Hint: it's a file.)

```
<yandex>
  <remote_servers>
    <scluster>
      <shard>
        <internal_replication>true</internal_replication>
        <replica>
          <host>ch-d7dc980i1-0.d7dc980i1s</host>
          <port>9000</port>
        </replica>
      </shard>
      <shard>
        <internal_replication>true</internal_replication>
        <replica>
          <host>ch-d7dc980i2-0.d7dc980i2s</host>
          <port>9000</port>
        </replica>
      </shard>
      ...
    </scluster>
  </remote_servers>
</yandex>
```

# Distributed engine spreads queries across shards



# What if you have a lot of data and you don't want to lose it?

```
CREATE TABLE sense.sdata (  
  DevId Int32, Type String,  
  MDate Date,  
  MDatetime DateTime,  
  Value Float64  
) engine=ReplicatedMergeTree(  
  '/clickhouse/{installation}/{scluster}/tables/{scluster-  
shard}/sense/sdata',  
  '{replica}', MDate, (DevId, MDatetime), 8192);
```

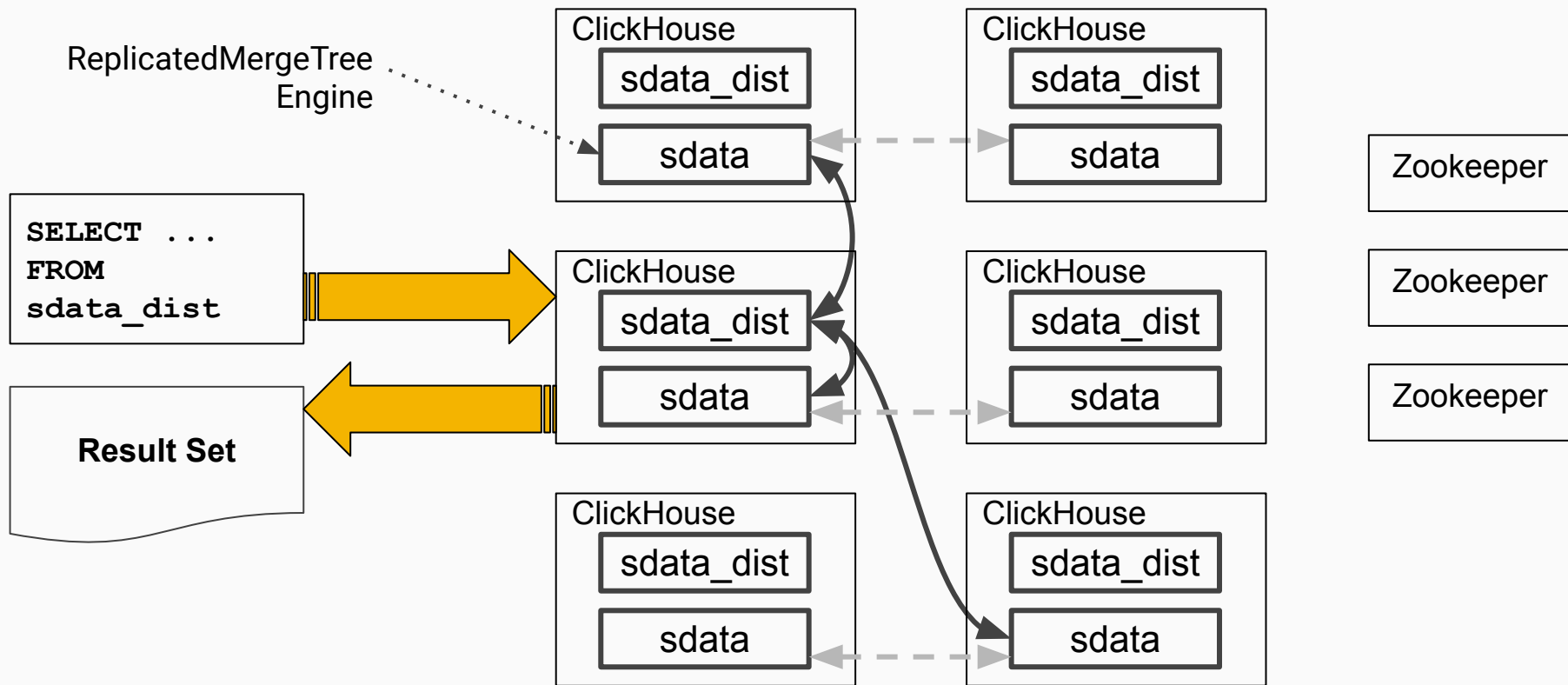
Engine for replicated  
tables

Cluster identification

Replica #

MergeTree parameters

# Now we distribute across shards and replicas



# With basic engine knowledge you can now tune queries

```
SELECT Dest, count(*) c, avg(DepDelayMinutes)
FROM ontime
GROUP BY Dest HAVING c > 100000
ORDER BY c DESC limit 5
```



```
SELECT Dest, count(*) c, avg(DepDelayMinutes)
FROM ontime
WHERE toYear(FlightDate) =
      toYear(toDate('2016-01-01'))
GROUP BY Dest HAVING c > 100000
ORDER BY c DESC limit 5
```

**Scans 355 table parts  
in parallel; does not  
use index**

**Scans 12 parts (3%  
of data) because  
FlightDate is  
partition key**

**Hint: clickhouse-server.log has the query plan**