ClickHouse Data Warehouse 101

The First Billion Rows

Alexander Zaitsev and Robert Hodges
About Us

Robert Hodges - Altinity CEO
30+ years on DBMS plus virtualization and security.
Previously at VMware and Continuent

Alexander Zaitsev - Altinity CTO
Expert in data warehouse with petabyte-scale deployments.
Altinity Founder; Previously at LifeStreet (Ad Tech business)
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
ClickHouse Overview
ClickHouse is a powerful data warehouse that handles many use cases

- 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!
Tables are split into indexed, sorted parts for fast queries.

<table>
<thead>
<tr>
<th>Table</th>
<th>Part</th>
<th>Part</th>
<th>Part</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Index</td>
<td>Columns</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Indexed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sorted</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Compressed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
If one server is not enough -- ClickHouse can scale out easily

```
SELECT ...
FROM tripdata_dist
```

Result Set

```
tripdata_dist (Distributed)
```

```
tripdata (MergeTable)
```

```
tripdata_dist
```

```
tripdata
```

```
tripdata_dist
```

```
tripdata
```
Getting Started: Data Loading
Installation: Use packages on Linux host

```
$ sudo apt -y install clickhouse-client=19.6.2 \
  clickhouse-server=19.6.2 \
  clickhouse-common-static=19.6.2
...
$ sudo systemctl start clickhouse-server
...
$ clickhouse-client
11e99303c78e :) select version()
...
  version()
  19.6.2.11
```
Decision tree for ClickHouse basic schema design

1. Fields are fixed?
   - Yes
     - Types are known?
       - Yes
         - Use scalar columns with specific type
       - No
         - Use scalar columns with String type
   - No
     - Use array columns to store key value pairs

2. Select partition key and sort order
Tabular data structure typically gives the best results

```sql
CREATE TABLE tripdata (  
  `pickup_date` Date DEFAULT toDate(tpep_pickup_datetime),  
  `id` UInt64,  
  `vendor_id` String,  
  `tpep_pickup_datetime` DateTime,  
  `tpep_dropoff_datetime` DateTime,  
  ...  
) ENGINE = MergeTree  
PARTITION BY toYYYYMM(pickup_date)  
ORDER BY (pickup_location_id, dropoff_location_id, vendor_id)
```
Use clickhouse-client to load data quickly from files

CSV Input Data

```
"Pickup_date","id","vendor_id","tpep_pickup_datetime"
"2016-01-02",0,"1","2016-01-02 04:03:29","2016-01-02...
"2016-01-29",0,"1","2016-01-29 12:00:51","2016-01-29...
"2016-01-09",0,"1","2016-01-09 17:22:05","2016-01-09...
```

Reading CSV Input with Headers

```
clickhouse-client --database=nyc_taxi_rides --query='INSERT INTO tripdata FORMAT CSVWithNames' < data.csv
```

Reading Gzipped CSV Input with Headers

```
gzip -d -c | clickhouse-client --database=nyc_taxi_rides --query='INSERT INTO tripdata FORMAT CSVWithNames'
```
Wouldn’t it be nice to run in parallel over a lot of input files?

Altinity Datasets project does exactly that!

- Dump existing schema definitions and data to files
- Load files back into a database
- Data dump/load commands run in parallel

See https://github.com/Altinity/altinity-datasets
How long does it take to load 1.3B rows?

$ time ad-cli dataset load nyc_taxi_rides --repo_path=/data1/sample-data
Creating database if it does not exist: nyc_timed
Executing DDL: /data1/sample-data/nyc_taxi_rides/ddl/taxi_zones.sql
. . .
Loading data: table=tripdata, file=data-200901.csv.gz
. . .
Operation summary: succeeded=193, failed=0

real    11m4.827s
user    63m32.854s
sys     2m41.235s

(Amazon md5.2xlarge: Xeon(R) Platinum 8175M, 8vCPU, 30GB RAM, NVMe SSD)
Do we really have 1B+ table?

:) select count() from tripdata;

SELECT count()
FROM tripdata

<table>
<thead>
<tr>
<th>count()</th>
</tr>
</thead>
<tbody>
<tr>
<td>1310903963</td>
</tr>
</tbody>
</table>

1 rows in set. Elapsed: 0.324 sec. Processed 1.31 billion rows, 1.31 GB (4.05 billion rows/s., 4.05 GB/s.)

1,310,903,963/11m4s = 1,974,253 rows/sec!!!
Getting Started on Queries
Let’s try to predict maximum performance

```
SELECT avg(number)
FROM
(
    SELECT number
    FROM system.numbers
    LIMIT 1310903963
)
```

```
├─avg(number)─┐
│   655451981 │
└─────────────┘
```

1 rows in set. Elapsed: 3.420 sec. Processed 1.31 billion rows, 10.49 GB (383.29 million rows/s., 3.07 GB/s.)
Now we try with the real data

```
SELECT avg(passenger_count)
FROM tripdata

<table>
<thead>
<tr>
<th>avg(passenger_count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6817462943317076</td>
</tr>
</tbody>
</table>

1 rows in set. Elapsed: ?
```

Guess how fast?
Now we try with the real data

```
SELECT avg(passenger_count)
FROM tripdata
```

```
│ avg(passenger_count) │
│----------------------|
│ 1.6817462943317076   |
```

1 rows in set. Elapsed: 1.084 sec. Processed 1.31 billion rows, 1.31 GB (1.21 billion rows/s., 1.21 GB/s.)

Even faster!!!!

Data type and cardinality matters
What if we add a filter

SELECT avg(passenger_count)
FROM tripdata
WHERE toYear(pickup_date) = 2016

<table>
<thead>
<tr>
<th>avg(passenger_count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6571129913837774</td>
</tr>
</tbody>
</table>

1 rows in set. Elapsed: 0.162 sec. Processed 131.17 million rows, 393.50 MB (811.05 million rows/s., 2.43 GB/s.)
What if we add a group by

```sql
SELECT
  pickup_location_id AS location_id,
  avg(passenger_count),
  count()
FROM tripdata
WHERE toYear(pickup_date) = 2016
GROUP BY location_id LIMIT 10
...
10 rows in set. Elapsed: 0.251 sec. Processed 131.17 million rows, 655.83 MB
(522.62 million rows/s., 2.61 GB/s.)
```
SELECT 
  zone,
  avg(passenger_count),
  count()
FROM tripdata
INNER JOIN taxi_zones ON taxi_zones.location_id = pickup_location_id
WHERE toYear(pickup_date) = 2016
GROUP BY zone
LIMIT 10

10 rows in set. Elapsed: 0.803 sec. Processed 131.17 million rows, 655.83 MB (163.29 million rows/s., 816.44 MB/s.)
Yes, ClickHouse is FAST!

This is the first time a free, CPU-based database has managed to out-perform a GPU-based database in my benchmarks. That GPU database has since undergone two revisions but nonetheless, the performance ClickHouse has found on a single node is very impressive.

https://tech.marksblogg.com/benchmarks.html
Optimization Techniques

How to make ClickHouse even faster
You can optimize

Server settings

Schema

Column storage

Queries
You can optimize

```
SELECT avg(passenger_count)
FROM tripdata
SETTINGS max_threads = 1
```

...  

1 rows in set. **Elapsed: 4.855 sec.** Processed 1.31 billion rows, 1.31 GB (270.04 million rows/s., 270.04 MB/s.)

```
SELECT avg(passenger_count)
FROM tripdata
SETTINGS max_threads = 8
```

...  

1 rows in set. **Elapsed: 1.092 sec.** Processed 1.31 billion rows, 1.31 GB (1.20 billion rows/s., 1.20 GB/s.)

Default is a half of available cores -- good enough
Schema optimizations

Data types

Index

Dictionaries

Arrays

Materialized Views and aggregating engines
Data Types matter!

CREATE MATERIALIZED VIEW tripdata_mv
ENGINE = SummingMergeTree
PARTITION BY toYYYYMM(pickup_date)
ORDER BY (pickup_location_id, dropoff_location_id, vendor_id) AS
SELECT
    pickup_date,
    vendor_id,
    pickup_location_id,
    dropoff_location_id,
    sum(passenger_count) AS passenger_count_sum,
    sum(trip_distance) AS trip_distance_sum,
    sum(fare_amount) AS fare_amount_sum,
    sum(tip_amount) AS tip_amount_sum,
    sum(tolls_amount) AS tolls_amount_sum,
    sum(total_amount) AS total_amount_sum,
    count() AS trips_count
FROM tripdata
GROUP BY
    pickup_date,
    vendor_id,
    pickup_location_id,
    dropoff_location_id
MaterializedView with SummingMergeTree

INSERT INTO tripdata_mv SELECT
    pickup_date,
    vendor_id,
    pickup_location_id,
    dropoff_location_id,
    passenger_count,
    trip_distance,
    fare_amount,
    tip_amount,
    tolls_amount,
    total_amount,
    1
FROM tripdata;

Ok.

0 rows in set. Elapsed: 303.664 sec. Processed 1.31 billion rows,
50.57 GB (4.32 million rows/s., 166.54 MB/s.)

Note, no group by!

SummingMergeTree automatically aggregates data in the background
SELECT count()
FROM tripdata_mv

<table>
<thead>
<tr>
<th>count()</th>
</tr>
</thead>
<tbody>
<tr>
<td>20742525</td>
</tr>
</tbody>
</table>

1 rows in set. Elapsed: 0.015 sec. Processed 20.74 million rows, 41.49 MB (1.39 billion rows/s., 2.78 GB/s.)

SELECT zone,
       sum(passenger_count_sum)/sum(trips_count),
       sum(trips_count)
FROM tripdata_mv
INNER JOIN taxi_zones ON taxi_zones.location_id = pickup_location_id
WHERE toYear(pickup_date) = 2016
GROUP BY zone
LIMIT 10

10 rows in set. Elapsed: 0.036 sec. Processed 3.23 million rows, 64.57 MB (89.14 million rows/s., 1.78 GB/s.)
Realtime Aggregation with Materialized Views

INSERTS → Raw Data → Summing MergeTree
             ↙             ↘
             ↘             ↘
             Summing MergeTree
             Summing MergeTree
             Summing MergeTree
Column storage optimizations

Compression

LowCardinality

Column encodings
LowCardinality example. Another 1B rows.

```sql
:) create table test_lc (
    a String, a_lc LowCardinality(String) DEFAULT a) Engine = MergeTree
PARTITION BY tuple() ORDER BY tuple();

:) INSERT INTO test_lc (a) SELECT
    concat('openconfig-interfaces:interfaces/interface/subinterfaces/subinterface/state/index', toString(rand() % 1000))
FROM system.numbers LIMIT 1000000000;

<table>
<thead>
<tr>
<th>table</th>
<th>name</th>
<th>type</th>
<th>compressed</th>
<th>uncompressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_lc</td>
<td>a</td>
<td>String</td>
<td>4663631515</td>
<td>84889975226</td>
</tr>
<tr>
<td>test_lc</td>
<td>a_lc</td>
<td>LowCardinality(String)</td>
<td>2010472937</td>
<td>2002717299</td>
</tr>
</tbody>
</table>
```
LowCardinality example. Another 1B rows

:) select a, count(*) from test_lc group by a order by count(*) desc limit 10;

<table>
<thead>
<tr>
<th>a</th>
<th>count()</th>
</tr>
</thead>
<tbody>
<tr>
<td>openconfig-interfaces:interfaces/...</td>
<td>1002761</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>openconfig-interfaces:interfaces/...</td>
<td>1002203</td>
</tr>
</tbody>
</table>

10 rows in set. Elapsed: 11.627 sec. Processed 1.00 billion rows, 92.89 GB (86.00 million rows/s., 7.99 GB/s.)

Faster

:) select a_lc a, count(*) from test_lc group by a order by count(*) desc limit 10;

...  

10 rows in set. Elapsed: 1.569 sec. Processed 1.00 billion rows, 3.42 GB (637.50 million rows/s., 2.18 GB/s.)
create table test_array ( 
  s String,
  a Array(LowCardinality(String)) default arrayDistinct(splitByChar(',', s)) 
) Engine = MergeTree PARTITION BY tuple() ORDER BY tuple();

INSERT INTO test_array (s)
WITH ['Percona', 'Live', 'Altinity', 'ClickHouse', 'MySQL', 'Oracle', 'Austin', 'Texas', 'PostgreSQL', 'MongoDB'] AS keywords
SELECT concat(keywords[((rand(1) % 10) + 1)], ',',
  keywords[((rand(2) % 10) + 1)], ',',
  keywords[((rand(3) % 10) + 1)], ',',
  keywords[((rand(4) % 10) + 1)])
FROM system.numbers LIMIT 1000000000;
### Array example. Another 1B rows

Data sample:

<table>
<thead>
<tr>
<th>s</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Texas,ClickHouse,Live,MySQL</td>
<td>['Texas','ClickHouse','Live','MySQL']</td>
</tr>
<tr>
<td>Texas,Oracle,Altinity,PostgreSQL</td>
<td>['Texas','PostgreSQL','Oracle','Altinity']</td>
</tr>
<tr>
<td>Percona,MySQL,MySQL,Austin</td>
<td>['MySQL','Percona','Austin']</td>
</tr>
<tr>
<td>PostgreSQL,Austin,PostgreSQL,Percona</td>
<td>['PostgreSQL','Percona','Austin']</td>
</tr>
<tr>
<td>Altinity,Percona,Percona,Percona</td>
<td>['Altinity','Percona']</td>
</tr>
</tbody>
</table>

Storage:

<table>
<thead>
<tr>
<th>table</th>
<th>name</th>
<th>type</th>
<th>comp</th>
<th>uncomp</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_array</td>
<td>s</td>
<td>String</td>
<td>11239860686</td>
<td>31200058000</td>
</tr>
<tr>
<td>test_array</td>
<td>a</td>
<td>Array(LowCardinality(String))</td>
<td>4275679420</td>
<td>11440948123</td>
</tr>
</tbody>
</table>

Array efficiently models 1-to-N relationship
Array example. Another 1B rows

): select count() from test_array where s like 'ClickHouse';

<table>
<thead>
<tr>
<th>count()</th>
</tr>
</thead>
<tbody>
<tr>
<td>343877409</td>
</tr>
</tbody>
</table>

1 rows in set. Elapsed: 7.363 sec. Processed 1.00 billion rows, 39.20 GB (135.81 million rows/s., 5.32 GB/s.)

): select count() from test_array where has(a,'ClickHouse');

<table>
<thead>
<tr>
<th>count()</th>
</tr>
</thead>
<tbody>
<tr>
<td>343877409</td>
</tr>
</tbody>
</table>

1 rows in set. Elapsed: 8.428 sec. Processed 1.00 billion rows, 11.44 GB (118.66 million rows/s., 1.36 GB/s.)

Well, ‘like’ is very efficient, but we reduced I/O a lot.

* has() will be optimized by dev team
SELECT
    zone,
    avg(passenger_count),
    count()
FROM tripdata
INNER JOIN taxi_zones ON taxi_zones.location_id = pickup_location_id
WHERE toYear(pickup_date) = 2016
GROUP BY zone
LIMIT 10

10 rows in set. Elapsed: 0.803 sec. Processed 131.17 million rows, 655.83 MB (163.29 million rows/s., 816.44 MB/s.)

Can we do it any faster?
SELECT
    zone,
    sum(pc_sum) / sum(pc_cnt) AS pc_avg,
    sum(pc_cnt)
FROM
    (SELECT
        pickup_location_id,
        sum(passenger_count) AS pc_sum,
        count() AS pc_cnt
    FROM tripdata
    WHERE toYear(pickup_date) = 2016
    GROUP BY pickup_location_id
    )
INNER JOIN taxi_zones ON taxi_zones.location_id = pickup_location_id
GROUP BY zone LIMIT 10

10 rows in set. Elapsed: 0.248 sec. Processed 131.17 million rows, 655.83 MB (529.19 million rows/s., 2.65 GB/s.)
ClickHouse Integrations
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
Integrations with MySQL

MySQL External Dictionaries (pull data from MySQL to CH)

MySQL Table Engine and Table Function (query/insert)

Binary Log Replication

ProxySQL supports ClickHouse

ClickHouse supports MySQL wire protocol (in June release)
..and with PostgreSQL

ODBC External Dictionaries (pull data from PostgreSQL to CH)

ODBC Table Engine and Table Function (query/insert)

Logical Replication: https://github.com/mkabilov/pg2ch

Foreign Data Wrapper: https://github.com/Percona-Lab/clickhousedb_fdw
ClickHouse Operator -- an easy way to manage ClickHouse DWH in Kubernetes

[Diagram]

- ClickHouse Installation YAML file
- Kubernetes API
- Monitoring Healthchecks
- ClickHouse cluster resources

https://github.com/Altinity/clickhouse-operator
Where to get more information

- ClickHouse Docs: https://clickhouse.yandex/docs/en/
- Altinity Blog: https://www.altinity.com/blog
- Meetups and presentations: https://www.altinity.com/presentations
  - 2 April -- Madrid, Spain ClickHouse Meetup
  - 7 May -- Limassol, Cyprus ClickHouse Meetup
  - 28-30 May -- Austin, TX Percona Live 2019
  - 4 June -- San Francisco ClickHouse Meetup
  - 8 June -- Beijing ClickHouse Meetup
  - September -- ClickHouse Paris Meetup
Questions?

Thank you!

Contacts:
info@altinity.com

Visit us at:
https://www.altinity.com

Read Our Blog:
https://www.altinity.com/blog