New Tips and Tricks Every ClickHouse Developer Should Know

Robert Hodges & Altinity Engineering
Let’s make some introductions - Us…

**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

ClickHouse support and services including Altinity.Cloud
Authors of Altinity Kubernetes Operator for ClickHouse and other open source projects
And ClickHouse, a real-time analytic database

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 the core engine for low-latency analytics
ClickHouse tips and tricks from 2019

- Use send_logs_level = 'trace' to see what ClickHouse is doing
- Use encodings to reduce data size
- Use materialized views to find last point data
- Use arrays to store key-value pairs
- Use materialized columns to precompute values
- Use dictionaries instead of joins for dimension data
- Use MySQL database engine instead of dictionaries
- Use TTLs to delete data
- Use replication instead of backups

From: *Tips and Tricks Every ClickHouse User Should Know*  
*(YouTube Video, 2019)*
Read S3 data fast with wildcards and named collections
Selecting data from S3

S3 Object Storage → s3() Function → Result Set

Parquet File

SELECT ... FROM s3(collection, url = '...bucket...', ...)
Define a named collection to specific url + credentials

/etc/clickhouse-server/config.d/s3.xml

<clickhouse>
  <named_collections>
    <s3_data>
      <url>https://s3.us-east-1.amazonaws.com/bucket/ORDERS/*.parquet</url>
      <access_key_id>AK...UA</access_key_id>
      <secret_access_key>dy...qu</secret_access_key>
      <format>Parquet</format>
    </s3_data>
  </named_collections>
</clickhouse>
Create a table on ClickHouse

```sql
set max_threads = 2;
SELECT min(O_TOTALPRICE),
    avg(O_TOTALPRICE),
    max(O_TOTALPRICE)
FROM s3(s3_data);
set max_threads = 4;
. . .
set max_threads = 8;
. . .
set max_threads = 16;
. . .
set max_threads = 32;
```
More ways to go fast: parallel inserts

On a single host

-- Define parallelization
SET max_insert_threads=16
SET max_threads = 16

INSERT INTO default.ORDERS
SELECT * FROM s3(s3_data)

Across a cluster

-- Propagate max_insert_threads & max_threads to other shards using profile.

INSERT INTO default.ORDERS
SELECT * FROM s3Cluster(...)
Reduce storage size using codecs and compression
Before: Unoptimized table for sensor readings

CREATE TABLE IF NOT EXISTS readings_unopt (  
  sensor_id Int64,
  sensor_type Int32,
  location String,
  time DateTime,
  date Date DEFAULT toDate(time),
  reading Float32
) Engine = MergeTree
PARTITION BY tuple()
ORDER BY tuple();
Codecs reduce data before compression

- **N862AA**
  - Value: 6 bytes
- **LowCardinality encoding**
  - Turns each string into an integer
  - Aka Dictionary Encoding
  - 2 bytes
- **Smaller Value**: 351
  - 2 bytes
- **ZSTD(1) compression**
  - Something very small!
After: Apply codecs and compression together!

CREATE TABLE IF NOT EXISTS readings_zstd (  
sensor_id Int32 Codec(DoubleDelta, ZSTD(1)),  
sensor_type UInt16 Codec(ZSTD(1)),  
location LowCardinality(String) Codec(ZSTD(1)),  
time DateTime Codec(DoubleDelta, ZSTD(1)),  
date ALIAS toDate(time),  
temperature Decimal(5,2) Codec(T64, ZSTD(10))  
)  
Engine = MergeTree  
PARTITION BY toYYYYMM(time)  
ORDER BY (location, sensor_id, time);
On-disk table size for different schemas

Bytes per row for different levels of schema optimization

- Unoptimized table: 4.52
- Optimized datatypes, codecs, LZ4: 1.34
- ZSTD instead of LZ4: 1.16
## Quick Comparison of Codecs

<table>
<thead>
<tr>
<th>Name</th>
<th>Best for</th>
</tr>
</thead>
<tbody>
<tr>
<td>LowCardinality</td>
<td>Strings with fewer than 10K values</td>
</tr>
<tr>
<td>Delta</td>
<td>Time series</td>
</tr>
<tr>
<td>Double Delta</td>
<td>Increasing counters</td>
</tr>
<tr>
<td>Gorilla</td>
<td>Gauge data (bounces around mean)</td>
</tr>
<tr>
<td>T64</td>
<td>Integers other than random hashes</td>
</tr>
<tr>
<td>FPC</td>
<td>Floating point sequences</td>
</tr>
</tbody>
</table>
Repeat time column in ORDER BY to speed up time-based queries
Time is a common component of table order

CREATE TABLE web_events_1x (  
`time` DateTime,  
`user_id` UInt32,  
`session_id` UInt64,  
`event_type` UInt16,  
`str_value` String,  
`float_value` Float32  
) ENGINE = MergeTree  
PARTITION BY toYYYYMM(time)  
ORDER BY (user_id, session_id, time)  

Partition by month to get <= 1000 parts total
Following the “rules” can lead to slow queries!

WITH toStartOfDay(toDateTime('2019-02-05 01:00:00')) AS day
SELECT
    avg(float_value), avg(length(str_value))
FROM web_events_1x
WHERE day = toStartOfDay(time)

[host] . . . Selected 1/2 parts by partition key, 1 parts by primary key, **28544/28544 marks by primary key**, 28544 marks to read from 1 ranges

1 row in set.  **Elapsed: 1.322 sec.** Processed 233.83 million rows, **2.70 GB** (176.88 million rows/s., 2.04 GB/s.)
CREATE TABLE web_events_2x (  `time` DateTime,  `user_id` UInt32,  `session_id` UInt64,  `event_type` UInt16,  `str_value` String,  `float_value` Float32 ) ENGINE = MergeTree PARTITION BY toYYYYMM(time) PRIMARY KEY (user_id, toStartOfDay(time), session_id) ORDER BY (user_id, toStartOfDay(time), session_id, time)
What goes on under the covers in MergeTree tables

Parts

Sparse index

Skip indexes

Name: 201901_1_207_3

Name: 201901_208_474_4

Name: 201905_510_815_3

Sorted, compressed, indexed column

Minmax

Bloom
Now we can query efficiently by day in monthly partitions

WITH toStartOfDay(toDateTime('2019-02-05 01:00:00')) AS day
SELECT
    avg(float_value), avg(length(str_value))
FROM web_events_1x
WHERE day = toStartOfDay(time)

[host] . . . Selected 1/1 parts by partition key, 1 parts by primary key, 4690/61036 marks by primary key, 4690 marks to read from 999 ranges

1 row in set. Elapsed: 0.646 sec. Processed 38.42 million rows, 986.50 MB (59.43 million rows/s., 1.53 GB/s.)
Handle mutable data with
ReplacingMergeTree
CREATE TABLE sakila.film (
    `film_id` UInt16,
    `title` String,
    
    `_version` UInt64 DEFAULT 0,
    `_sign` Int8 DEFAULT 1
)
ENGINE = ReplacingMergeTree( version)
ORDER BY language_id, studio_id, film_id

Pro tip: Use PRIMARY KEY to prefix a long ORDER BY

Other cols go to left

Row key goes on right (if you have one)
How ReplacingMergeTree works

<table>
<thead>
<tr>
<th>_version</th>
<th>_sign</th>
<th>film_id</th>
<th>language_id</th>
<th>studio_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>+1</td>
<td>1001</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>1001</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>+1</td>
<td>1001</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>-1</td>
<td>1001</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Eventually consistent replacement of rows

De-duplicate on these columns

(Other data columns)

INSERT

UPDATE

DELETE
Adding a row to RMT table

```
INSERT INTO sakila.film VALUES
(1001,'Blade Runner','Best. Sci-fi. Film. Ever.',
'1982',1,NULL,6,'0.99',117,'20.99','PG',
'Deleted Scenes,Behind the Scenes',now(),
0,1)
```

```
SELECT title, release_year
FROM film WHERE film_id = 1001
```

<table>
<thead>
<tr>
<th>title</th>
<th>release_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blade Runner</td>
<td>1982</td>
</tr>
</tbody>
</table>
Updating a row in the RMT table

INSERT INTO sakila.film VALUES
(1001,'Blade Runner','Best. Sci-fi. Film. Ever.',...,3,-1),
(1001,'Blade Runner - Director''s Cut','Best. Sci-fi. Film. Ever.',...,3,1)

SELECT title, release_year
FROM film WHERE film_id = 1001

<table>
<thead>
<tr>
<th>title</th>
<th>release_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blade Runner - Director's Cut</td>
<td>1982</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>title</th>
<th>release_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blade Runner</td>
<td>1982</td>
</tr>
</tbody>
</table>
Rows are replaced when merges occur

Part

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1001</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>+1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>1001</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>+1</td>
<td>1001</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Merged Part

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1001</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>-1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>+1</td>
<td>1001</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Pro tip: never assume rows will merge fully
FINAL keyword merges data dynamically

```
SELECT film_id, title 
FROM sakila.film
    FINAL
WHERE film_id = 1001
```

<table>
<thead>
<tr>
<th>title</th>
<th>release_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blade Runner - Director's Cut</td>
<td>1982</td>
</tr>
</tbody>
</table>

Adds initial scan to merge rows
Row policies prevent deleted rows from showing up

```
CREATE ROW POLICY sakila_film_rp ON sakila.film
    FOR SELECT USING sign != -1 TO ALL

SELECT title, release_year, _version, _sign
FROM sakila.film FINAL
WHERE film_id = 1001

Ok.

0 rows in set. Elapsed: 0.005 sec.
```
Store RBAC model in ZooKeeper

(Or ClickHouse Keeper!)
SQL RBAC is clumsy to manage across multiple servers

CREATE USER example IDENTIFIED WITH SHA256_PASSWORD BY 'secret';

ClickHouse
/var/lib/clickhouse/access

???
ClickHouse
/var/lib/clickhouse/access
Old school approach to propagate RBAC commands

```sql
CREATE USER IF NOT EXISTS example
ON CLUSTER '{cluster}'
IDENTIFIED WITH
    SHA256_PASSWORD BY 'secret';
```

- Execute on cluster hosts
- Persnickety syntax to make call idempotent
- Repeat every time a new server is added
Configuration to enable RBAC replication

/etc/clickhouse-server/users.d/zk_rbac.xml

<clickhouse>
  <user_directories replace="replace"/>
    <users_xml>
      <path>/etc/clickhouse-server/users.xml</path>
    </users_xml>
  <replicated>
    <zookeeper_path>/clickhouse/access/</zookeeper_path>
  </replicated>
</user_directories>
</clickhouse>
Now all changes will propagate automatically!

CREATE USER example IDENTIFIED WITH SHA256_PASSWORD BY 'secret';
How to see current RBAC settings

SELECT name, value
FROM system.zookeeper
WHERE path = '/clickhouse/access/uuid/
FORMAT Vertical

Row 1:

name: ee48286d-e012-674d-4629-c395b84db6b0
value: ATTACH USER example IDENTIFIED WITH sha256_hash
       BY 'F9...06' SALT '6E...BF';
Reduce data size and cost with TTL clauses
TTLs started as a way to “time out” rows

CREATE TABLE default.web_events_with_ttl_2 (  `time` DateTime,
  . . .
  `float_value` Float32
)
ENGINE = MergeTree
PARTITION BY toYYYYMM(time)
ORDER BY (user_id, toStartOfDay(time), session_id, time)
TTL time + INTERVAL 12 MONTH DELETE
Now TTLs can move, aggregate, and recompress data

```
CREATE TABLE default.web_events_with_ttl_2 (  
`time` DateTime,
  . . .
`float_value` Float32
)
ENGINE = MergeTree
PARTITION BY toYYYYMM(time)
ORDER BY (user_id, toStartOfDay(time), session_id, time)
TTL time + INTERVAL 1 MONTH RECOMPRESS CODEC (ZSTD(1)),
  time + INTERVAL 6 MONTH RECOMPRESS CODEC (ZSTD(10)),
  time + INTERVAL 12 MONTH DELETE
```
Let’s prove it works!

```sql
SELECT partition, name, rows,
    data_compressed_bytes AS compressed,
    data_uncompressed_bytes AS uncompressed
FROM system.parts
WHERE (table = 'web_events_with_ttl_2') AND active
ORDER BY name DESC
```

<table>
<thead>
<tr>
<th>partition</th>
<th>name</th>
<th>rows</th>
<th>compressed</th>
<th>uncompressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>202304</td>
<td>202304_1_1_0</td>
<td>50000</td>
<td>613930</td>
<td>1388890</td>
</tr>
<tr>
<td>202302</td>
<td>202302_2_2_1</td>
<td>50000</td>
<td>327461</td>
<td>1388890</td>
</tr>
<tr>
<td>202208</td>
<td>202208_3_3_1</td>
<td>50000</td>
<td>264054</td>
<td>1388890</td>
</tr>
</tbody>
</table>

Use aggregation to simulate joins
Basic big data design: One table or many?

Transaction Header
- msg_type='xheader'
- xact_id
- start_time
- initiator_host

Transaction State
- msg_type='xstate'
- xact_id
- start_time
- end_time
- state
- host

ClickHouse bias for big data: Put entities in a single table
How do we normally join master detail records?

Transaction header

<table>
<thead>
<tr>
<th>msg_type</th>
<th>xact_id</th>
<th>time</th>
</tr>
</thead>
</table>

JOIN key

History of transaction state

<table>
<thead>
<tr>
<th>xact_id</th>
<th>t_time</th>
<th>t_state</th>
<th>start_time</th>
</tr>
</thead>
</table>

Transaction state changes

Large table joins are an anti-pattern in low-latency apps!
Aggregation can implement master/detail joins!

Transaction header and state changes

<table>
<thead>
<tr>
<th>msg_type</th>
<th>xact_id</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>msg_type</th>
<th>xact_id</th>
<th>time</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

History of transaction state

<table>
<thead>
<tr>
<th>xact_id</th>
<th>t_time</th>
<th>t_state</th>
<th>start_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>236</td>
<td>t1</td>
<td>init</td>
<td>t0</td>
</tr>
<tr>
<td>236</td>
<td>t2</td>
<td>compute</td>
<td>t0</td>
</tr>
<tr>
<td>236</td>
<td>t3</td>
<td>compute</td>
<td>t0</td>
</tr>
<tr>
<td>...</td>
<td>t4</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

GROUP BY

key

xact_id

start_time: t0

t_time: [t1, t2, t3, ...]

t_state: [init, compute, compute, ..]

Grouped array values

ARRAY JOIN to pivot on arrays
And here’s the code...

```
SELECT xact_id, t_time, t_state, start_time
FROM
(
    SELECT xact_id, groupArrayIf(time, msg_type = 'xstate') AS t_time,
         groupArrayIf(state, msg_type = 'xstate') AS t_state,
         anyIf(time, msg_type = 'xheader') AS start_time
    FROM transaction
    GROUP BY xact_id
)
ARRAY JOIN t_time, t_state
```
And the output...

<table>
<thead>
<tr>
<th>xact_id</th>
<th>t_time</th>
<th>t_state</th>
<th>start_time</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>2023-04-10 23:13:10.000</td>
<td>init</td>
<td>2023-04-10 23:13:06.000</td>
</tr>
<tr>
<td>17</td>
<td>2023-04-10 23:13:11.000</td>
<td>compute</td>
<td>2023-04-10 23:13:06.000</td>
</tr>
<tr>
<td>17</td>
<td>2023-04-10 23:13:24.000</td>
<td>compute</td>
<td>2023-04-10 23:13:06.000</td>
</tr>
<tr>
<td>14</td>
<td>2023-04-10 23:13:06.000</td>
<td>init</td>
<td>2023-04-10 23:13:05.000</td>
</tr>
<tr>
<td>14</td>
<td>2023-04-10 23:13:08.000</td>
<td>compute</td>
<td>2023-04-10 23:13:05.000</td>
</tr>
<tr>
<td>14</td>
<td>2023-04-10 23:13:22.000</td>
<td>compute</td>
<td>2023-04-10 23:13:05.000</td>
</tr>
</tbody>
</table>
Q: Why does this work??  A: ClickHouse query model
Conclusion
Handy list of tips and tricks!

- Read S3 fast with max_threads and wildcards
- Reduce storage size using codecs and compression
- Use multiple time values in ORDER BY to query subsets of parts efficiently
- Handle rapidly changing data with ReplacingMergeTree
- Store RBAC model in ZooKeeper (or ClickHouse Keeper)
- Recompress data to reduce size over time with TTLs
- Use aggregation instead of joins on large datasets
More information!

- Altinity YouTube channel
  - Tips and Tricks Every ClickHouse User Should Know
  - Adventures with the ClickHouse ReplacingMergeTree Engine
  - A Day in the Life of a ClickHouse Query
- Altinity Blog – https://altinity.com/blog
Thank you! Questions?

Altinity.Cloud
Altinity Support
Altinity Stable Builds

https://altinity.com