Tackling Big Telco Data With ClickHouse

- Jonathan Abrams @ NexPath Networks
  jon@nexpath.net

Who am I?

- Started out in telecom back in 2001
- Along with switching, background in software development and database work
- Work with the switching, operations and financial aspects of customers' businesses
- Successfully migrated many customers to more open, financially viable solutions

What is ClickHouse?

- ClickHouse is an open-source, high-performance column-store database with a SQL front end
- Developed by Yandix
- Column-store storage layout is key to ClickHouse’s performance

What ClickHouse is good at

- Storing lots of data
- Aggregate (OLAP) queries on large tables
- (Semi) Structured Data
- Time Series Data

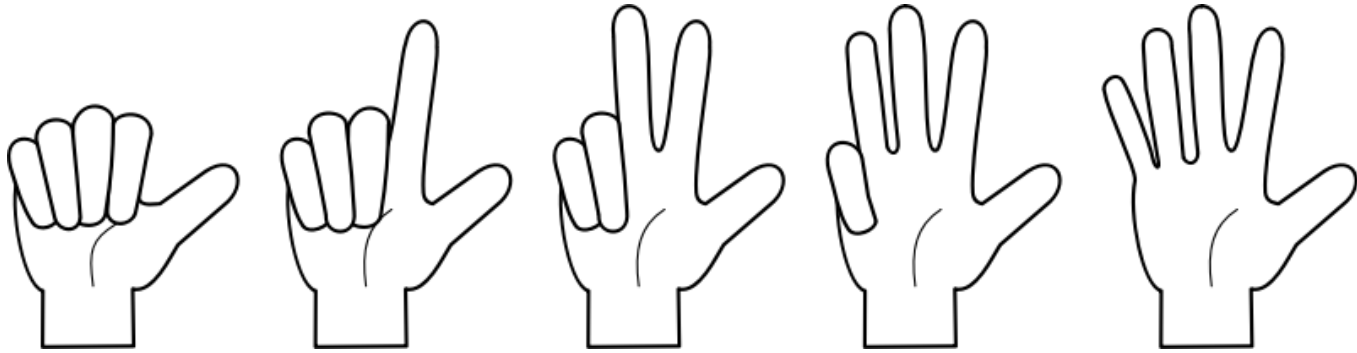
Even More Good

- Fast queries on reasonably powerful hardware
- Decent performance on platter storage
- Very flexible interacting with outside data sources
- Replication and distributed queries

What is not so great

- OLTP
- No ACID
- Key-Value Store

Usage Scenarios



- CDR/Event record storage for reporting and analysis
- CDR/Event record archival
- Streaming and tabulating event
- SIP Capture Storage

Tools and Integrations

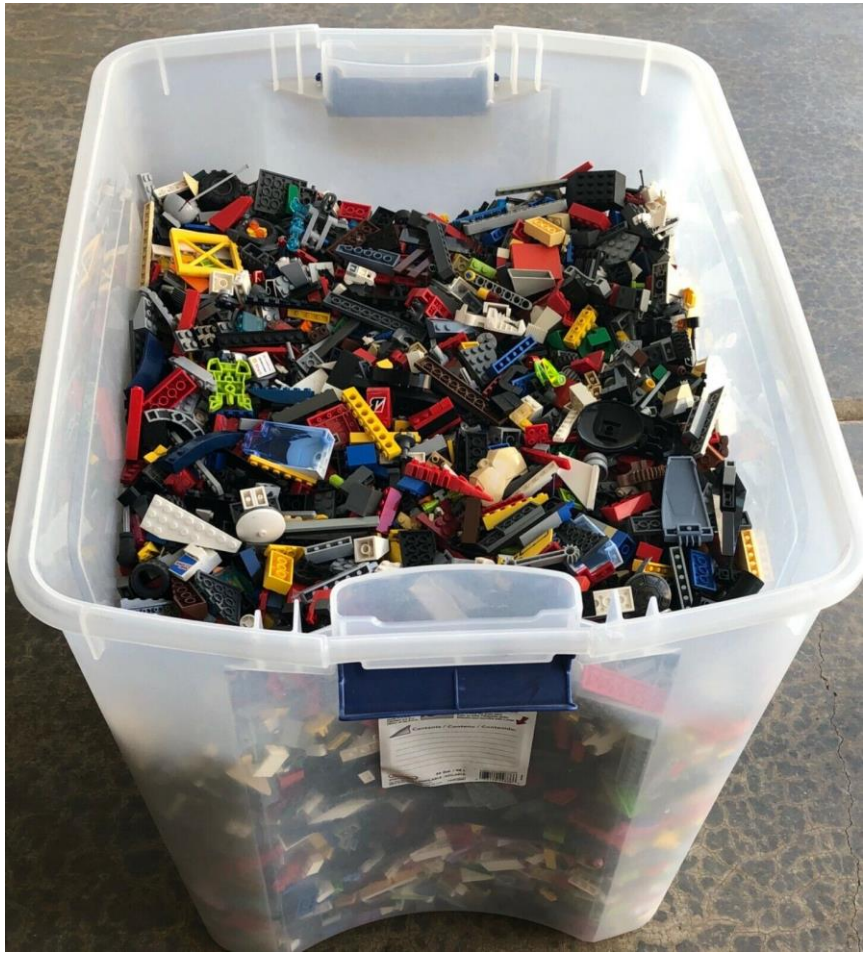
- Query Interfaces
  - Clickhouse cli client
  - ClickHouse HTTP Server
  - JetBrains DataGrip (commercial)
- Visualization
  - Grafana
  - Apache SuperSet
  - Looker
  - Redash
  - Tabix
  - A lot more, and the list keeps growing

Storage

- ClickHouse supports many table engine types
- For local/native storage, MergeTree is the workhorse table engine
- Other specialized MergeTree implementations exist for specific purposes
- ClickHouse also has table engines to support external storage in S3 or HDFS.

What you can store

- Data Types
  - Integer: (U)Int8, (U)Int16, (U)Int32, (U)Int64, Int128, (U)Int256
  - Floating Point - Float32, Float64
  - Fixed Point - Decimal32, Decimal64
  - Boolean
  - Strings - String, FixedString
  - Date - Date, DateTime, DateTime64
- Arrays, Tuples, Maps, and Enums
- Nulls are supported



Encodings and Compression

- Each column in a ClickHouse table can have different encoding and compression schemes.

- Encodings and Compression can be used together for further storage efficiency gains.

- No need to normalize your data

Column Encodings

- Delta and DoubleDelta
  - encodes deltas
- Gorilla
  - encodes delta from a mean
- T64
  - auto sizing Int

LowCardinality()

- Auto-Enum column
- Not only does this increase storage efficiency but provides a performance boost as well.
- Compression can be enabled on top this, further reducing storage.

Column Compression

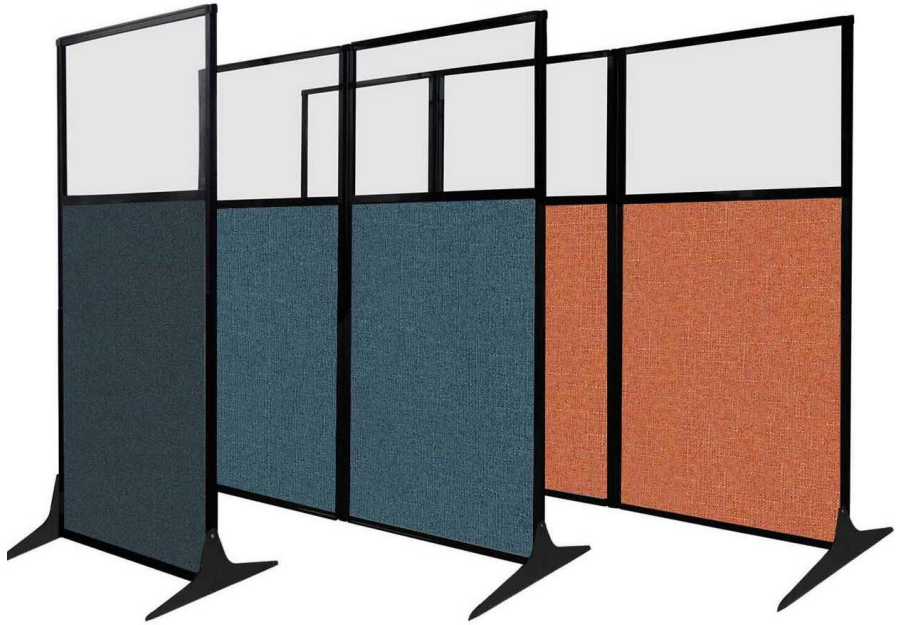
- LZ4, LZ4HC and ZSTD column compression are supported out of the box
- LZ4 is the default works well for most data after encoding
- Overall table compression will be similar to that of gzipped flat files on most CDR related datasets I’ve seen

Common Telco Data
Column Compression Ratios

name	type	compression_codec	compressed	uncompressed	compress_ratio	bytes_per_row
orig_call_id	String	CODEC(ZSTD(1))	68.76 GiB	251.40 GiB	27.35%	11.00
called	String	CODEC(ZSTD(1))	23.24 GiB	75.04 GiB	30.97%	3.72
calling	String	CODEC(ZSTD(1))	21.66 GiB	75.04 GiB	28.86%	3.46
cust_rate	Decimal	CODEC(LZ4)	19.66 GiB	56.28 GiB	34.93%	3.14
src_ip	LC(String)	CODEC(LZ4)	6.28 GiB	6.27 GiB	100.16%	1.00
cust_rounded_dur	UInt32	CODEC(T64, LZ4)	2.26 GiB	25.01 GiB	9.05%	0.36
vendor_rate	Decimal	CODEC(LZ4)	15.04 GiB	56.28 GiB	26.72%	2.40
cld_lrn	LC(String)	CODEC(LZ4)	11.12 GiB	13.08 GiB	85.01%	1.78
cld_ocn	LC(String)	CODEC(LZ4)	8.26 GiB	12.52 GiB	65.97%	1.32
cust_carrier_id	UInt64	CODEC(T64, LZ4)	6.93 GiB	50.03 GiB	13.84%	1.11
cld_lata	LC(String)	CODEC(LZ4)	6.02 GiB	6.27 GiB	96.13%	0.96
cld_state	LC(String)	CODEC(LZ4)	5.75 GiB	6.27 GiB	91.71%	0.92
cust_cost	Decimal(16,8)	CODEC(LZ4)	5.42 GiB	56.28 GiB	9.64%	0.87
vendor_cost	Decimal(16,8)	CODEC(LZ4)	5.27 GiB	56.28 GiB	9.36%	0.84
orig_carrier_name	LC(String)	CODEC(LZ4)	4.98 GiB	6.27 GiB	79.40%	0.80
pdd_ms	UInt32	CODEC(T64, LZ4)	4.43 GiB	25.01 GiB	17.70%	0.71
release_reason	LC(String)	CODEC(LZ4)	4.11 GiB	6.27 GiB	65.66%	0.66
cld_category	LC(String)	CODEC(LZ4)	3.65 GiB	6.27 GiB	58.24%	0.58
bill_dur	UInt32	CODEC(LZ4)	3.60 GiB	25.01 GiB	14.38%	0.58
rate_ts	UInt64	CODEC(DoubleDelta, LZ4)	1.11 GiB	50.03 GiB	2.23%	0.18
end_time	DateTime64	Primary Key	330.13 MiB	50.03 GiB	0.64%	0.05
call_dir	UInt8	CODEC(T64, LZ4)	318.88 MiB	6.25 GiB	4.98%	0.05
short_flag1	UInt8	CODEC(T64, LZ4)	286.67 MiB	6.25 GiB	4.48%	0.04

Table Partitioning

- MergeTree tables can have Partition Key specified to enable table partitioning.
- You can drop, truncate, detach, and optimize individual partition parts.
- Helper functions such as toYYYYMMDD(), toYYYYMM(), and toYYYY() make partitioning by date simple



Primary Keys

- MergeTree tables can have a non-unique "primary key" based on columns or expressions

- The PRIMARY KEY is a skip index that can be used to significantly speed up queries with WHERE clauses

- Very space efficient, fits in memory

OpenSIPs acc table in ClickHouse

```
CREATE TABLE opensips.acc (
    method LowCardinality(String),
    from_tag String CODEC (ZSTD),
    to_tag String CODEC (ZSTD),
    callid String CODEC (ZSTD),
    sip_code LowCardinality(String),
    sip_reason LowCardinality(String),
    time DateTime CODEC (DoubleDelta, LZ4),
    duration UInt32 CODEC (T64, LZ4),
    ms_duration UInt32 CODEC (T64, LZ4),
    setuptime DateTime CODEC(DoubleDelta, LZ4),
    created Nullable(DateTime) CODEC(DoubleDelta, LZ4)
) ENGINE = MergeTree() PRIMARY KEY (time) ORDER BY (time)
    PARTITION BY toYYYYMM(time);
```

Skip Indexes

- Secondary data skipping indexes allow you to further speed up queries containing WHERE clauses.
- Skip indexes don’t point to individual rows but give ClickHouse hints to what data might exist in a block of column data.
- Skip indexes are defined on a per column basis

Skip Index Types

- minmax – stores the minimum and maximum values of a column or even expression
- set – stores a list of unique values in a column
- bloom_filter – General purpose bloom filter that can be used on most column data types.
- tokenbf_v1 – Stores a bloom filter for tokens/strings separated by a delimiter.

Skip Index Types, continued

- ngrambf_v1 – n-gram bloom filter
  - Stores a n-gram bloom filter for n-grams/chunks of Strings, such as “Str” and “ings”.
  - Speeds up queries on String columns in WHERE clauses with string operators such as equals, like, in, startsWith or endsWith.
  - You specify the size of the n-gram bloom filter when you create the index.

Integrating ClickHouse With Outside Data Sources

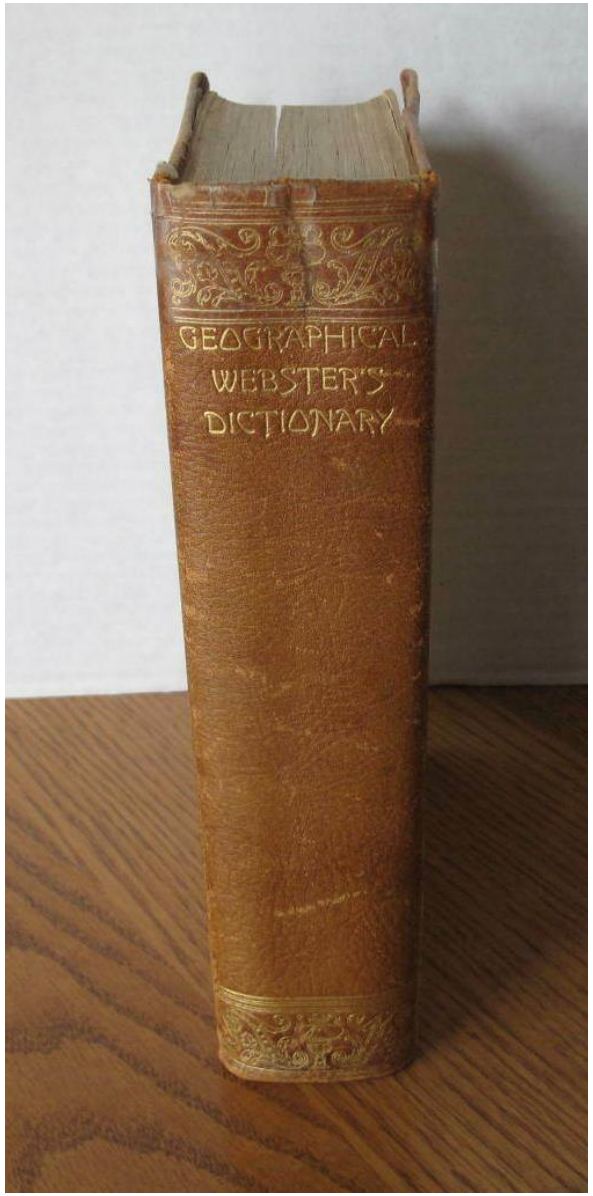
- External Dictionaries
- Proxy tables
- Directly from SQL in an ad-hoc fashion
- Streaming from Kafka and RabbitMQ topics/queues
- As a replication client to Postgres or MySQL

Dictionaries

- Dictionaries can be created from tables, text sources, or external database table engines
- These allow quick and easy lookups of meta data in SQL queries
- Dictionaries can be auto-refreshed at specified time intervals

```
CREATE DICTIONARY customer_data(
    id UInt32,
    name String )
PRIMARY KEY id
SOURCE(MYSQL(
   port 3306
   host 'localhost'
   user 'user'
   password 'password'
   db 'rar'
   table 'customer'
)) LAYOUT(HASHED()), LIFETIME(300);
```

```
SELECT
    dictGetString(customer_data, 'name', cust_id) "customer_name",
    calling,
    called,
    duration
FROM cdrs
WHERE call_time >= '2021-05-01 00:00:00'
```



Proxy Table Engines

- You can define proxy tables with the external table engines
- These proxy tables can be queried like normal tables from SQL within ClickHouse


```
CREATE TABLE customers
(
    `customer_id` UInt32,
    `customer_name` String,
    `datecreated` Date,
    `status` UInt32,
    `balance` Float64
)
ENGINE = MySQL('127.0.0.1:3307', 'db_name', 'customers', 'user', 'password')
```

Ad-Hoc External Queries

- You can query external tables directly from SQL queries by using a function for the table name in the SELECT FROM.
- Makes querying and joining data across multiple data sources very simple

```
SELECT b.customer_name, SUM(duration)/60 "minutes", COUNT(DURATION) "attempts",
  SUM(IF(duration>=0 or sip_code='200', 1, 0)) "completes" FROM acc a
LEFT OUTER JOIN (SELECT customer_name, customer_ip FROM
  mysql('127.0.0.1', 'db_name', 'customers', 'user', 'pass')) b ← mysql() table engine
ON a.src_ip = b.customer_ip
GROUP BY b.customer_name;
```

Kafka and RabbitMQ Streaming

- Kafka and RabbitMQ table engines allow ClickHouse to become a topic/queue consumer

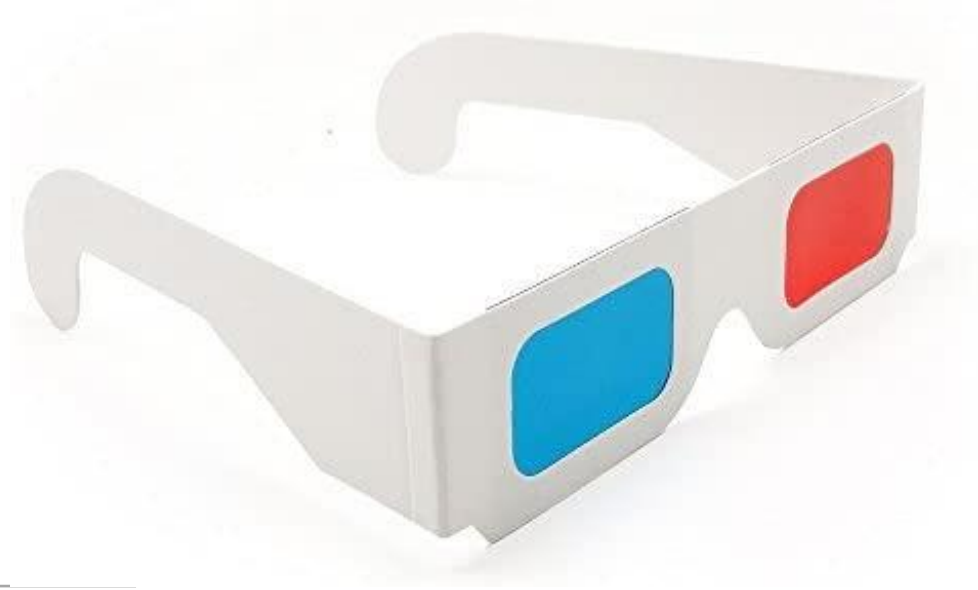
- MATERIALIZED VIEWs can be used to automatically pull the data and insert it into tables

```
CREATE TABLE kafka_event_stream (
timestamp DateTime64,
server_ip String,
event_type String,
status String,
response_ms UInt32
) ENGINE = Kafka('127.0.0.1:9092', 'event_topic', 'ch_1', 'CSV')
```

Materialized Views

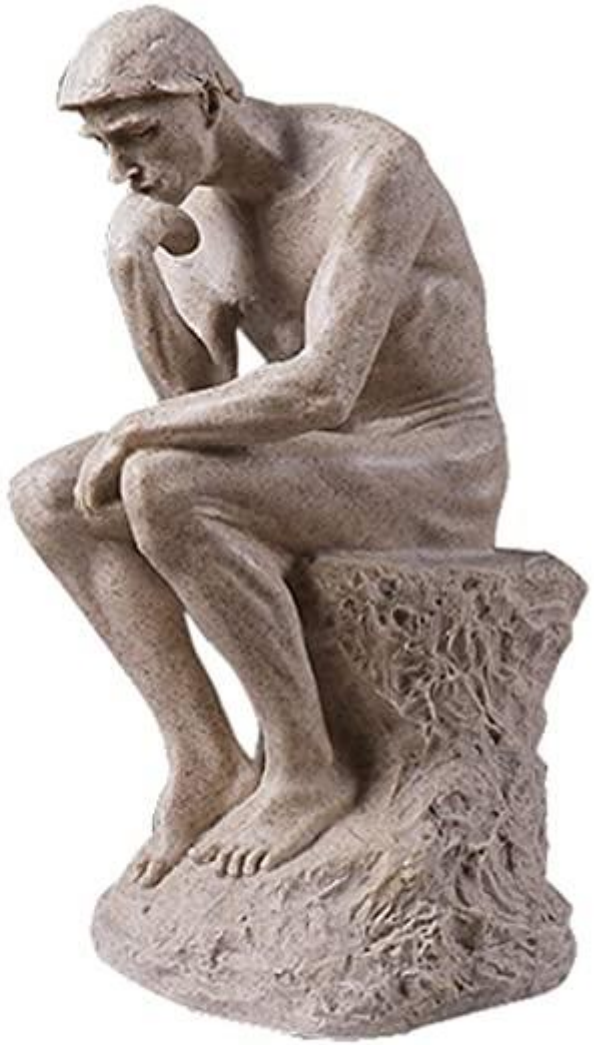
- Materialized views can be used to summarize/transform data from one table into another on an ongoing basis
- Can be combined with Kafka/RabbitMQ streams to insert data into tables as it becomes available
- SummingMergeTree tables will automatically aggregate columns, grouping by the order keys

```
CREATE MATERIALIZED VIEW mv_event_summary
TO event_summary_by_hour
AS SELECT
    toStartOfHour(timestamp) event_hour,
    server_ip,
    event_type,
    count(event_type) attempts,
    sum(if(status='reject',1,0)) AS rejects
FROM kafka_event_stream
GROUP BY event_hour, server_ip, event_type
```



Other Interesting Query Features

- Json column data function
- Statistical Functions
- Window Functions
- Arrays, Tuples, and Maps
- CatBoost Integration



FULL OUTER JOINs

- Joins 2 record sets and show rows where data exists in both, or just one dataset.
- A normal INNER JOIN will only show rows where data exists in both record sets,
- LEFT/RIGHT OUTER JOIN will only show rows that exists in both record sets or the LEFT/RIGHT record set.

ASOF JOINs

- ASOF JOINs allow you to match 2 record sets on keys that might not be exact matches
- It will join on the closest match

```
SELECT a.call_date, a.orig_number, a.term_number, b.bill_dur vendor_dur, a.bill_dur-
vendor_dur “dur_diff”
   FROM (SELECT call_date, orig_number, term_number, bill_dur FROM my_cdrs
   WHERE call_date BETWEEN '2021-01-25 00:00:00' AND '2021-01-25 01:00:00') a
ASOF JOIN
   (SELECT call_date, orig_number, term_number, bill_dur FROM vendor_cdrs
   WHERE call_date BETWEEN '2021-01-25 00:00:00' AND '2021-01-25 01:00:00') b
ON a.orig_number = b.orig_number AND a.term_number = b.term_number
AND a.call_date <= b.call_date ← this is the inexact match
```

Bulk-loading Data

- Data can be bulk-loaded locally or remotely via the clickhouse cli utility
  - zcat acc.csv.gz | clickhouse client --host=127.0.0.1 --query= "INSERT INTO cdrs.acc FORMAT CSV"
- Textual formats
  - CSV, TSV, JSON.
- Binary formats
  - CapnProto, Protobuf, Avro, Parquet, Arrow, or ORC

Long-term data maintenance

- The easiest way to purge data is to drop partitions. Partitions can also be detached, and the data moved to a different location for archival.
- MergeTree tables can have a TTL clause defined to automatically drop rows after a definable time period.
  - **ALTER TABLE cdrs.acc TTL req_date + toIntervalDay(14)**
- UPDATEs
  - **ALTER TABLE cdrs.acc UPDATE rated = 0 WHERE time < ‘2021-03-01 00:00:00’**
- DELETEs
  - **ALTER TABLE cdrs.acc DELETE WHERE time < ‘2021-03-01 00:00:00’**

The Future

- Still under heavy development
- Seeing more and more integrations with other software packages, open-source and commercial
- I expect ClickHouse to commoditize the column-store like MySQL/PostGres did for the RDMS