



■ Using ClickHouse for Market Data

Christoph Wurm



Real-time data feeds

Market security <i>(stock, bond, coin, etc.)</i>	open, price, price, volume, ... , price, price, close, volume
Device	temp, temp, pressure, temp, temp, pressure, temp, flow rate, temp, ...
Hardware/ Software	cpu, memory, cpu, memory, sockets, cpu, memory, users, ...



Real-time data feeds

Market security <i>(stock, bond, coin, etc.)</i>	open, price, price, volume, ... , price, price, close, volume
Device	temp, temp, pressure, temp, temp, pressure, temp, flow rate, temp, ...
Hardware/ Software	cpu, memory, cpu, memory, sockets, cpu, memory, users, ...



Table

```
CREATE TABLE ticks
(
    timestamp DateTime64,
    symbol     String,
    open       Float64,
    volume     Float64,
    price      Float64
)
ENGINE = MergeTree
ORDER BY (symbol, timestamp)
```



Let's take a look

```
SELECT count() FROM ticks
```

```
count()  
234010000
```

```
SELECT countDistinct(symbol) FROM ticks
```

```
uniqExact(symbol)  
10000
```

- 10,000 symbols times 23,400 seconds in a US trading day (6.5 hours)
- Plus 10,000 open prices



Table

```
CREATE TABLE ticks
(
  timestamp DateTime64,
  symbol     LowCardinality(String),
  open       Nullable(Float64),
  volume     Nullable(Float64),
  price      Nullable(Float64)
)
ENGINE = MergeTree
ORDER BY (symbol, timestamp)
```



Let's take a look

```
INSERT INTO ticks ... FROM file('...')
```

0 rows in set. Elapsed: 100.609 sec. Processed 234.01 million rows,
10.56 GB (2.33 million rows/s., 104.97 MB/s.)

```
SELECT formatReadableSize(total_bytes) FROM system.tables WHERE name =  
'ticks'
```

```
formatReadableSize(total_bytes)  
2.87 GiB
```

Let's optimize

```
CREATE TABLE ticks
(
    timestamp DateTime64 CODEC(Delta, Default),
    symbol LowCardinality(String),
    open Nullable(Float64) CODEC(Delta, Default),
    volume Nullable(Float64) CODEC(Delta, Default),
    price Nullable(Float64) CODEC(Delta, Default)
)
ENGINE = MergeTree
ORDER BY (symbol, timestamp)
```



Let's take a look

- Without codecs:

```
SELECT formatReadableSize(total_bytes) FROM system.tables WHERE name =  
'ticks'
```

```
formatReadableSize(total_bytes)  
2.87 GiB
```

- With codecs:

```
SELECT formatReadableSize(total_bytes) FROM system.tables WHERE name =  
'ticks'
```

```
formatReadableSize(total_bytes)  
1.79 GiB
```



First Query

```
SELECT
    symbol,
    argMax(open, timestamp) as open,
    argMax(volume, timestamp) as volume,
    argMax(price, timestamp) as price
FROM ticks
GROUP BY symbol
```

```
10000 rows in set. Elapsed: 0.985 sec. Processed 234.01 million rows,
9.83 GB (237.67 million rows/s., 9.98 GB/s.)
```

Let's optimize even more

```
CREATE TABLE ticks
(
    timestamp DateTime64 CODEC(Delta, Default),
    symbol LowCardinality(String),
    open Float64 DEFAULT -1 CODEC(Delta, Default),
    volume Float64 DEFAULT -1 CODEC(Delta, Default),
    price Float64 DEFAULT -1 CODEC(Delta, Default)
)
ENGINE = MergeTree
ORDER BY (symbol, timestamp)
```



Faster query

```
SELECT
    symbol,
    argMaxIf(open, timestamp, open >= 0) as open,
    argMaxIf(volume, timestamp, volume >= 0) as volume,
    argMaxIf(price, timestamp, price >= 0) as price
FROM ticks
GROUP BY symbol
```

10000 rows in set. Elapsed: 0.763 sec. Processed 234.01 million rows,
8.82 GB (306.71 million rows/s., 11.56 GB/s.)



Benchmark 10

```
clickhouse benchmark --concurrency 10 --iterations 10 -q "SELECT ... "
```

```
localhost:9000, queries 10, QPS: 2.211, RPS: 517328195.408, MiB/s: 18587.041, result RPS:  
22107.098, result MiB/s: 0.822.
```

0.000%	2.629 sec.
10.000%	4.029 sec.
20.000%	4.034 sec.
30.000%	4.050 sec.
40.000%	5.034 sec.
50.000%	5.072 sec.
60.000%	5.072 sec.
70.000%	5.090 sec.
80.000%	5.098 sec.
90.000%	5.099 sec.
95.000%	5.100 sec.
99.000%	5.100 sec.
99.900%	5.100 sec.
99.990%	5.100 sec.



Benchmark 100

```
clickhouse benchmark -c 100 -i 100 -q "SELECT ... "
```

```
localhost:9000, queries 100, QPS: 1.912, RPS: 447454621.459, MiB/s: 16076.559, result RPS:  
19121.175, result MiB/s: 0.711.
```

0.000%	47.508 sec.
10.000%	50.339 sec.
20.000%	51.177 sec.
30.000%	51.315 sec.
40.000%	51.373 sec.
50.000%	52.648 sec.
60.000%	53.113 sec.
70.000%	53.558 sec.
80.000%	54.040 sec.
90.000%	54.176 sec.
95.000%	54.236 sec.
99.000%	54.259 sec.
99.900%	54.359 sec.
99.990%	54.359 sec.



Benchmark 1000, oops...

```
clickhouse benchmark -c 1000 -i 1000 -q "SELECT . . . "
```

```
. (CANNOT_SCHEDULE_TASK) (version 22.7.1.375 (official build))
```

```
An error occurred while processing the query 'SELECT
```

```
symbol,
```

```
argMax(open, timestamp) as open,
```

```
argMax(volume, timestamp) as volume,
```

```
argMax(price, timestamp) as price
```

```
FROM ticks
```

```
GROUP BY symbol
```

```
': Code: 439. DB::Exception: Received from localhost:9000. DB::Exception: Cannot schedule  
a task: cannot allocate thread (threads=3086, jobs=3086). Stack trace:
```

```
<Empty trace>
```

Materialized View to the rescue

```
CREATE TABLE ticks_5min
(
  start  DateTime64 CODEC(Delta, Default),
  symbol LowCardinality(String),
  open   AggregateFunction(argMax, Float64, DateTime64),
  volume AggregateFunction(argMax, Float64, DateTime64),
  price  AggregateFunction(argMax, Float64, DateTime64)
)
ENGINE = AggregatingMergeTree
ORDER BY (symbol, start)
```


Materialized View to the rescue

```
CREATE MATERIALIZED VIEW ticks_mv TO ticks_5min
AS SELECT
    toStartOfFiveMinute(timestamp) AS start,
    symbol,
    argMaxStateIf(open, timestamp, open >= 0) as open,
    argMaxStateIf(volume, timestamp, volume >= 0) as volume,
    argMaxStateIf(price, timestamp, price >= 0) as price
FROM ticks
GROUP BY symbol, start
```

Let's take a look

```
SELECT count() FROM ticks_5min
```

```
count()  
790000
```

```
SELECT countDistinct(symbol) FROM ticks_5min
```

```
uniqExact(symbol)  
10000
```

- 79 5-minute intervals in the US trading day

Let's take a look

■ Raw data

```
SELECT formatReadableSize(total_bytes) FROM system.tables WHERE name =  
'ticks'
```

```
formatReadableSize(total_bytes)  
1.79 GiB
```

■ Materialized View

```
SELECT formatReadableSize(total_bytes) FROM system.tables WHERE name =  
'ticks_5min'
```

```
formatReadableSize(total_bytes)  
7.48 MiB
```



Super fast query

```
SELECT
    symbol,
    argMaxMerge(open) as open,
    argMaxMerge(volume) as volume,
    argMaxMerge(price) as price
FROM ticks_5min
GROUP BY symbol
```

10000 rows in set. Elapsed: 0.190 sec. Processed 885.17 thousand rows, 198.24 MB (4.67 million rows/s., 1.05 GB/s.)



Benchmark 10

```
clickhouse benchmark -c 10 -i 10 -q "SELECT . . . "
```

```
localhost:9000, queries 10, QPS: 102.920, RPS: 81306502.874, MiB/s: 17296.626, result RPS:  
1029196.239, result MiB/s: 38.280.
```

0.000%	0.097 sec.
10.000%	0.097 sec.
20.000%	0.097 sec.
30.000%	0.097 sec.
40.000%	0.097 sec.
50.000%	0.097 sec.
60.000%	0.097 sec.
70.000%	0.097 sec.
80.000%	0.097 sec.
90.000%	0.097 sec.
95.000%	0.097 sec.
99.000%	0.097 sec.
99.900%	0.097 sec.
99.990%	0.097 sec.



Benchmark 100

```
clickhouse benchmark -c 100 -i 100 -q "SELECT . . . "
```

```
localhost:9000, queries 100, QPS: 111.483, RPS: 88071799.609, MiB/s: 18735.832, result  
RPS: 1114832.906, result MiB/s: 41.465.
```

0.000%	0.217 sec.
10.000%	0.650 sec.
20.000%	0.814 sec.
30.000%	0.923 sec.
40.000%	0.947 sec.
50.000%	0.969 sec.
60.000%	0.975 sec.
70.000%	0.994 sec.
80.000%	1.005 sec.
90.000%	1.009 sec.
95.000%	1.013 sec.
99.000%	1.015 sec.
99.900%	1.016 sec.
99.990%	1.016 sec.



Benchmark 1000, works!

```
clickhouse benchmark -c 1000 -i 1000 -q "SELECT . . . "
```

```
localhost:9000, queries 1000, QPS: 155.534, RPS: 122871906.714, MiB/s: 26138.984, result  
RPS: 1555340.591, result MiB/s: 57.850.
```

0.000%	0.785 sec.
10.000%	2.701 sec.
20.000%	3.511 sec.
30.000%	4.873 sec.
40.000%	5.648 sec.
50.000%	6.638 sec.
60.000%	7.294 sec.
70.000%	8.341 sec.
80.000%	9.403 sec.
90.000%	9.692 sec.
95.000%	9.981 sec.
99.000%	10.444 sec.
99.900%	10.605 sec.
99.990%	10.611 sec.



Benchmark 10000, oh well...

```
clickhouse benchmark -c 10000 -i 10000 -q "SELECT ... "
```

```
DB::NetException: Timeout exceeded while reading from socket ([::1]:9000, 300000 ms)
```



Combined query

```
WITH '2022-04-05 21:57:30'::DateTime64 AS point_in_time
SELECT
    symbol,
    argMaxIf(open, timestamp_outer, open >= 0) as open,
    argMaxIf(volume, timestamp_outer, volume >= 0) as volume,
    argMaxIf(price, timestamp_outer, price >= 0) as price
FROM (
    SELECT
        max(start) AS timestamp_outer,
        symbol,
        argMaxMerge(open) as open,
        argMaxMerge(volume) as volume,
        argMaxMerge(price) as price
    FROM ticks_5min
    WHERE start <= toStartOfFiveMinute(point_in_time)
    GROUP BY symbol
```



Combined query

```
UNION ALL
SELECT
    max(timestamp) AS timestamp_outer,
    symbol,
    argMaxOrNullIf(open, timestamp, open >= 0) as open,
    argMaxOrNullIf(volume, timestamp, volume >= 0) as volume,
    argMaxOrNullIf(price, timestamp, price >= 0) as price
FROM ticks
WHERE timestamp BETWEEN toStartOfFiveMinute(point_in_time) AND
point_in_time
GROUP BY symbol
)
GROUP BY symbol
```

So slow...

10000 rows in set. Elapsed: **1.990 sec**. Processed 85.70 million rows,
2.25 GB (43.06 million rows/s., 1.13 GB/s.)

■ It's slower!



Why is it slower?

```
EXPLAIN indexes = 1 SELECT ...
```

```
Condition: (start in (-Inf, '1649192100'])
```

```
Parts: 1/1
```

```
Granules: 97/97
```

```
...
```

```
Condition: and((timestamp in (-Inf, '1649192340']), (timestamp in  
['1649192100', +Inf)))
```

```
Parts: 1/1
```

```
Granules: 10365/28566
```



Let's optimize one more time

```
CREATE TABLE ticks
(
    timestamp DateTime64 CODEC(Delta, Default),
    symbol      LowCardinality(String),
    open        Float64 DEFAULT -1 CODEC(Delta, Default),
    volume      Float64 DEFAULT -1 CODEC(Delta, Default),
    price       Float64 DEFAULT -1 CODEC(Delta, Default)
)
ENGINE = MergeTree
ORDER BY timestamp
```



There we go

- Inserts are faster:

```
0 rows in set. Elapsed: 59.372 sec. Processed 234.01 million rows, 10.56 GB (3.94 million rows/s., 177.88 MB/s.)
```

- Queries are faster:

```
10000 rows in set. Elapsed: 0.282 sec. Processed 2.31 million rows, 188.82 MB (8.17 million rows/s., 669.13 MB/s.)
```



And this is why it's fast

```
EXPLAIN indexes = 1 SELECT ...
```

```
Condition: (start in (-Inf, '1649192100'])
```

```
Parts: 1/1
```

```
Granules: 97/97
```

```
...
```

```
Condition: and((timestamp in (-Inf, '1649192340']), (timestamp in  
['1649192100', +Inf)))
```

```
Parts: 1/1
```

```
Granules: 185/28566
```



Benchmark 10

```
clickhouse benchmark -c 10 -i 10 -q "SELECT . . . "
```

```
localhost:9000, queries 10, QPS: 9.963, RPS: 24245771.070, MiB/s: 2217.666, result RPS:  
99634.928, result MiB/s: 3.947.
```

0.000%	0.972 sec.
10.000%	0.996 sec.
20.000%	1.003 sec.
30.000%	1.004 sec.
40.000%	1.004 sec.
50.000%	1.005 sec.
60.000%	1.005 sec.
70.000%	1.008 sec.
80.000%	1.011 sec.
90.000%	1.011 sec.
95.000%	1.021 sec.
99.000%	1.021 sec.
99.900%	1.021 sec.
99.990%	1.021 sec.



Benchmark 100

```
clickhouse benchmark -c 100 -i 100 -q "SELECT . . . "
```

```
localhost:9000, queries 100, QPS: 4.095, RPS: 9965198.357, MiB/s: 911.478, result RPS:  
40950.721, result MiB/s: 1.618.
```

0.000%	24.058 sec.
10.000%	24.273 sec.
20.000%	24.405 sec.
30.000%	24.423 sec.
40.000%	24.429 sec.
50.000%	24.450 sec.
60.000%	24.467 sec.
70.000%	24.481 sec.
80.000%	24.488 sec.
90.000%	24.493 sec.
95.000%	24.497 sec.
99.000%	24.505 sec.
99.900%	24.514 sec.
99.990%	24.514 sec.



Benchmark 1000

```
clickhouse benchmark -c 1000 -i 1000 -q "SELECT . . . "
```

```
localhost:9000, queries 1000, QPS: 6.996, RPS: 17024708.664, MiB/s: 1557.184, result RPS:  
69960.886, result MiB/s: 2.778.
```

0.000%	18.574 sec.
10.000%	59.784 sec.
20.000%	61.625 sec.
30.000%	120.383 sec.
40.000%	122.938 sec.
50.000%	163.713 sec.
60.000%	195.064 sec.
70.000%	197.533 sec.
80.000%	206.188 sec.
90.000%	215.800 sec.
95.000%	227.546 sec.
99.000%	227.726 sec.
99.900%	227.768 sec.
99.990%	227.771 sec.



Takeaways

- Use ClickHouse for real-time data feeds
 - ◇ Financial, sensors, metrics, etc.
- Use Materialized Views
 - ◇ On their own, or in combination with their source tables
- Mind your sorting key
 - ◇ It matters A LOT - for queries *and* inserts
 - ◇ Experiment!
- Avoid Nullable if possible
 - ◇ Use sentinel values
- Measure every step
 - ◇ clickhouse-benchmark is your friend
 - ◇ Test concurrent queries
- Implement exponential backoff in your client



The background features a dark blue field on the right and a series of diagonal stripes in varying shades of blue on the left. The stripes are oriented from the top-left towards the bottom-right.

Questions?