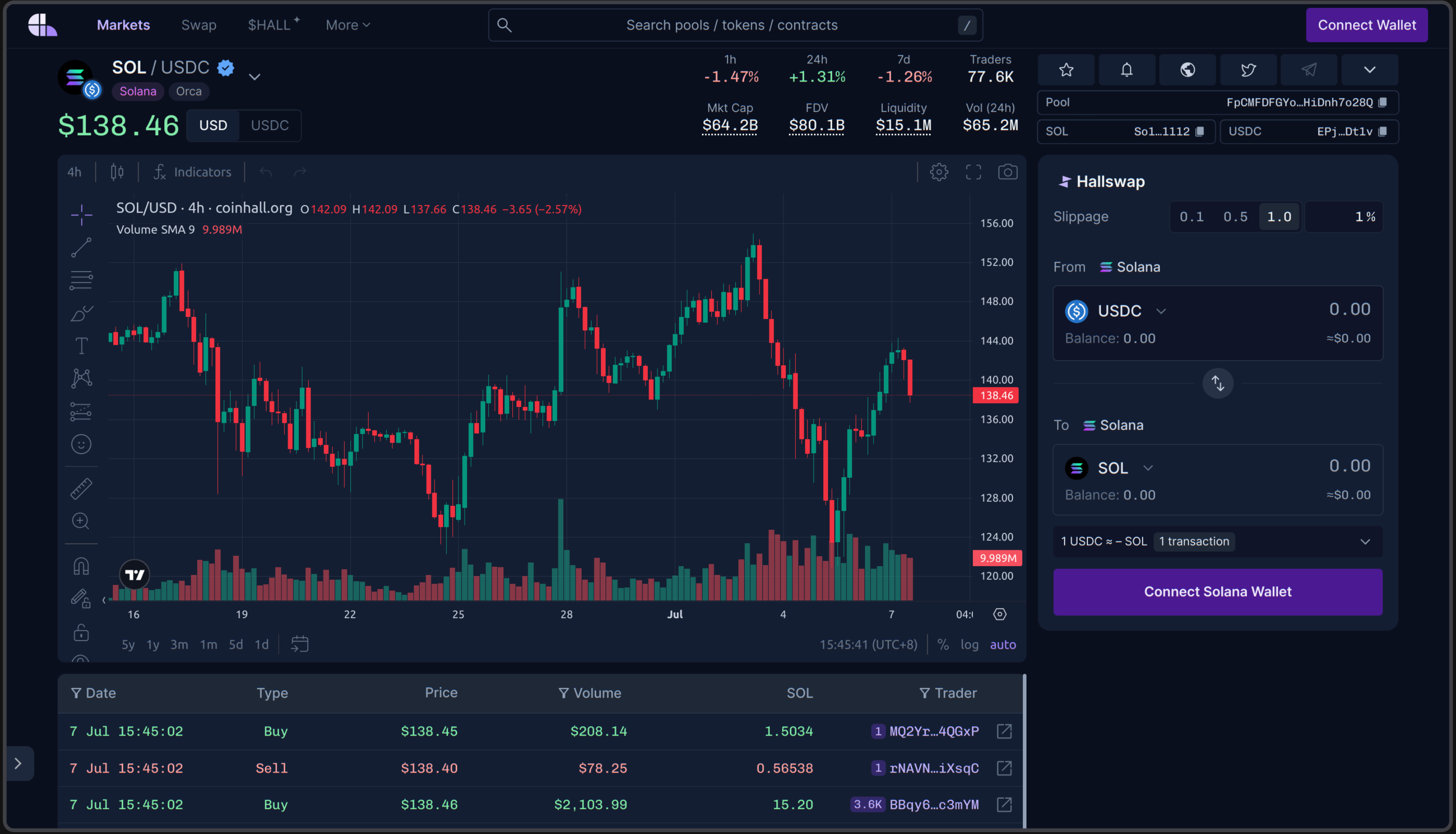
ClickHouse

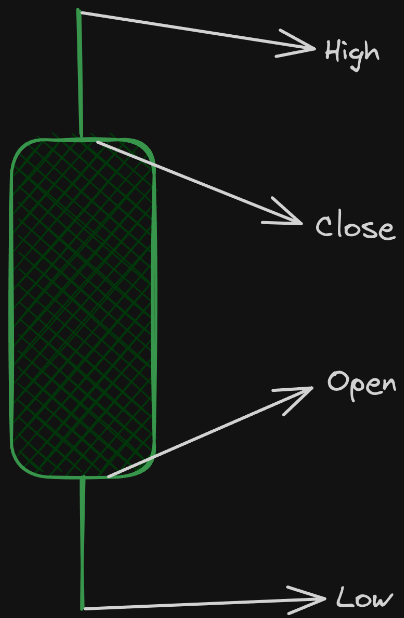
Powering Coinhall’s Real-Time Blockchain Data Platform

Coinhall - Trading Terminal for Decentralised Exchanges



Candlestick Crash Course

Visualising the open, high, low, and closing prices (OHLC) of an asset within a period



Candlestick Crash Course

Visualising the open, high, low, and closing prices (OHLC) of an asset within a period

Example `prices` table:

tstamp	price	quantity
1	2.23	4
3	2.14	2
4	2.71	7
6	2.69	1
9	2.42	2
10	2.92	5
12	2.38	1
15	2.72	8
17	2.13	6
19	2.61	3

> Each row is also known as a "tick"

Example query using ClickHouse SQL:

```sql
SELECT
  ts,
  argMin(price, tstamp) AS open,
  max(price) AS high,
  min(price) AS low,
  argMax(price, tstamp) AS close,
  sum(price * quantity) AS volume
FROM prices
GROUP BY floor(tstamp / 10) * 10 AS ts
--       toStartOfInterval(tstamp, INTERVAL 1 hour)
ORDER BY ts;
```

Results

ts	open	high	low	close	volume
0	2.23	2.71	2.14	2.42	39.70
10	2.92	2.92	2.13	2.61	59.35

Coinhall’s Data Journey

August 2021

Google
Big Query

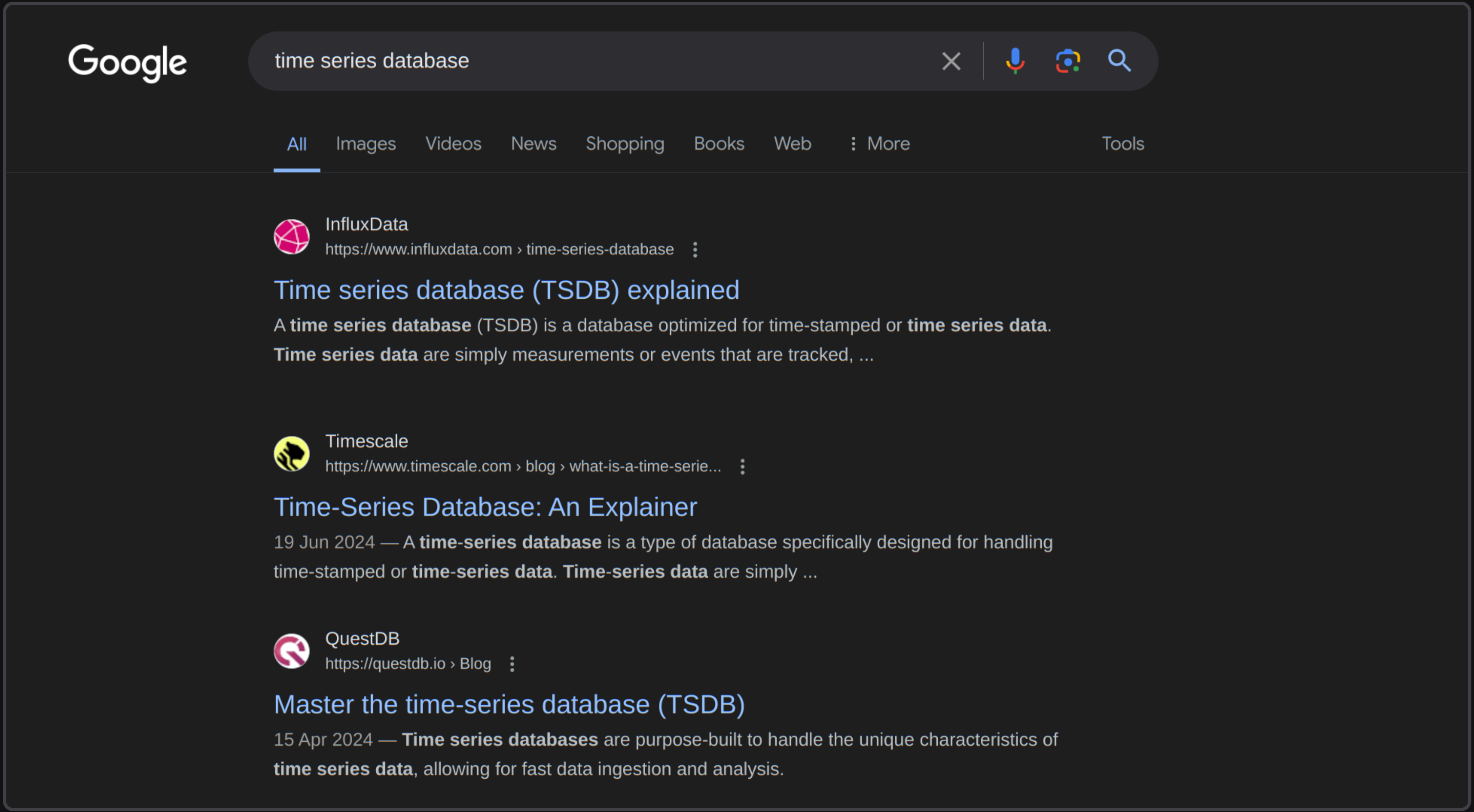
Too "slow"

The most trivial of queries take on average ~2s to run

Too expensive

$8.40 SGD per TB scanned, billed at a minimum of 10 MB per query

$$
\begin{aligned}
\text{Cost of 1M queries} &= 10^6 \times 10 \text{ MB} \\
&= 10 \text{ TB} \\
&= 10 \times \$8.40 \\
&= \$84 \\
\text{2M queries per day} &= 2 \times 30 \times \$84 \\
&= \$5040 \text{ per month}
\end{aligned}
$$


Google
time series database
All
Images
Videos
News
Shopping
Books
Web
More
Tools
InfluxData
https://www.influxdata.com › time-series-database
Time series database (TSDB) explained
A time series database (TSDB) is a database optimized for time-stamped or time series data. Time series data are simply measurements or events that are tracked, ...
Timescale
https://www.timescale.com › blog › what-is-a-time-serie...
Time-Series Database: An Explainer
19 Jun 2024 — A time-series database is a type of database specifically designed for handling time-stamped or time-series data. Time-series data are simply ...
QuestDB
https://questdb.io › Blog
Master the time-series database (TSDB)
15 Apr 2024 — Time series databases are purpose-built to handle the unique characteristics of time series data, allowing for fast data ingestion and analysis.

October 2021

QuestDB

Too limited

Lack of developer resources and features

Support for SQL DELETE #10

Open **bluestreak01** opened this issue on Nov 8, 2021 · 6 comments

A (Pretty Subjective) Comparison

	Cloud Spanner	BigQuery	QuestDB	Apache Druid	Rockset	Snowflake	ClickHouse	SingleStore
Type	OLTP	OLAP	OLAP	OLAP	OLAP	OLAP	OLAP	HTAP
Has Managed	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
SQL Support	Yes	Partial	Partial	Partial	Yes	Yes	Yes	Yes
Year Launched	2017	2010	2019	2011	2016	2014	2016	2011
Popularity	4.11	50.03	1.28	2.73	0.51	103.12	13.18	7.03
Tuning Difficulty*	Low	Medium	Low	Medium	Low	Medium	High	Medium
Developer Experience*	Good	Good	Bad	Normal	Normal	Normal	Bad	Good
Sub-second Aggregations*	No	No	Yes	Yes	Yes	Yes	Yes	Yes
Sub-second Joins*	No	No	Yes	No	Yes	Yes	Yes	Yes
Normal Views	Yes	Yes	No	No	Yes	Yes	Yes	Yes
Materialised Views	No	Batch only	No	Experimental	Near real-time	Unsuitable	Real-time	No
Costs	Very High	Very High	Low	High	Medium	Very High	Low	High

Data as of June 2022

* Inferred from online benchmarks and other resources; may be inaccurate

A (Pretty Naive) Benchmark

	Rockset (Managed)	SingleStore (Managed)	Snowflake (Managed)	ClickHouse (Doublecloud)	ClickHouse (Self-Hosted)
Specs	4 vCPU 32 GB RAM	2 vCPU 16 GB RAM	8 vCPU 16 GB RAM	4 vCPU 16 GB RAM	4 vCPU 16 GB RAM
Cost Per Month	$950	$641	$1,950	$368	$50
Candlestick	400ms	480ms → 50ms	600ms → 300ms	240ms → 40ms	20ms
Latest Prices	850ms	180ms → 45ms	600ms → 260ms	100ms → 15ms	8ms
ASOF JOIN	>30s (timeout)	Can't figure out	20mins	330ms → 80ms	50ms

Data as of June 2022
All tables contain 1M rows

July 2022

ClickHouse

Pitfalls & Optimisations

A database is only as fast as its slowest query

Optimisation: RTFM

https://clickhouse.com/docs/en/optimize/sparse-primary-indexes

Guides / Best Practices / **Sparse Primary Indexes**

A Practical Introduction to Primary Indexes in ClickHouse

Introduction

In this guide we are going to do a deep dive into ClickHouse indexing. We will illustrate and discuss in detail:

- how indexing in ClickHouse is different from traditional relational database management systems
- how ClickHouse is building and using a table's sparse primary index
- what some of the best practices are for indexing in ClickHouse

You can optionally execute all ClickHouse SQL statements and queries given in this guide by yourself on your own machine. For installation of ClickHouse and getting started instructions, see the Quick Start.

> **note**
> This guide is focusing on ClickHouse sparse primary indexes.
>
> For ClickHouse secondary data skipping indexes, see the Tutorial.

Data Set

Throughout this guide we will use a sample anonymized web traffic data set.

- We will use a subset of 8.87 million rows (events) from the sample data set.
- The uncompressed data size is 8.87 million events and about 700 MB. This compresses to 200 mb when stored in ClickHouse.
- In our subset, each row contains three columns that indicate an internet user (`UserID` column) who clicked on a URL (`URL` column) at a specific time (`EventTime` column).

With these three columns we can already formulate some typical web analytics queries such as:

The `LIMIT BY` Clause

To select the first `n` rows for each distinct value (eg. latest prices of each asset)

asset_id	tstamp	price	quantity
a	1	2.23	4
a	3	2.14	2
a	4	2.71	7
a	6	2.69	1
a	9	2.42	2
b	1	2.92	5
b	2	2.38	1
b	5	2.72	8
b	7	2.13	6
b	9	2.61	3

```sql
-- Table definition
CREATE TABLE prices (
  `asset_id` String,
  `tstamp`   UInt64,
  `price`    Float64,
  `quantity` UInt64,
)
ORDER BY (asset_id, tstamp);

-- Query
SELECT * FROM prices
ORDER BY tstamp DESC
LIMIT 1 BY asset_id;
```

Results

asset_id	tstamp	price	quantity
a	9	2.42	2
b	9	2.61	3

limit by always scanning full table #37567

Open harryhan1989 opened this issue on May 26, 2022 · 0 comments

Optimisation: `AggregatingMergeTree` + Materialized View

```sql
CREATE TABLE prices_latest (
  `asset_id` String,
  `tstamp`   AggregateFunction(max, UInt64),
  `price`    AggregateFunction(argMax, Float64, UInt64),
)
ENGINE = AggregatingMergeTree ORDER BY asset_id;

CREATE MATERIALIZED VIEW prices_latest_mv
TO prices_latest
AS SELECT
  asset_id,
  maxState(prices.tstamp) AS tstamp,
  argMaxState(prices.price, prices.tstamp) AS price
FROM prices
GROUP BY asset_id;
```

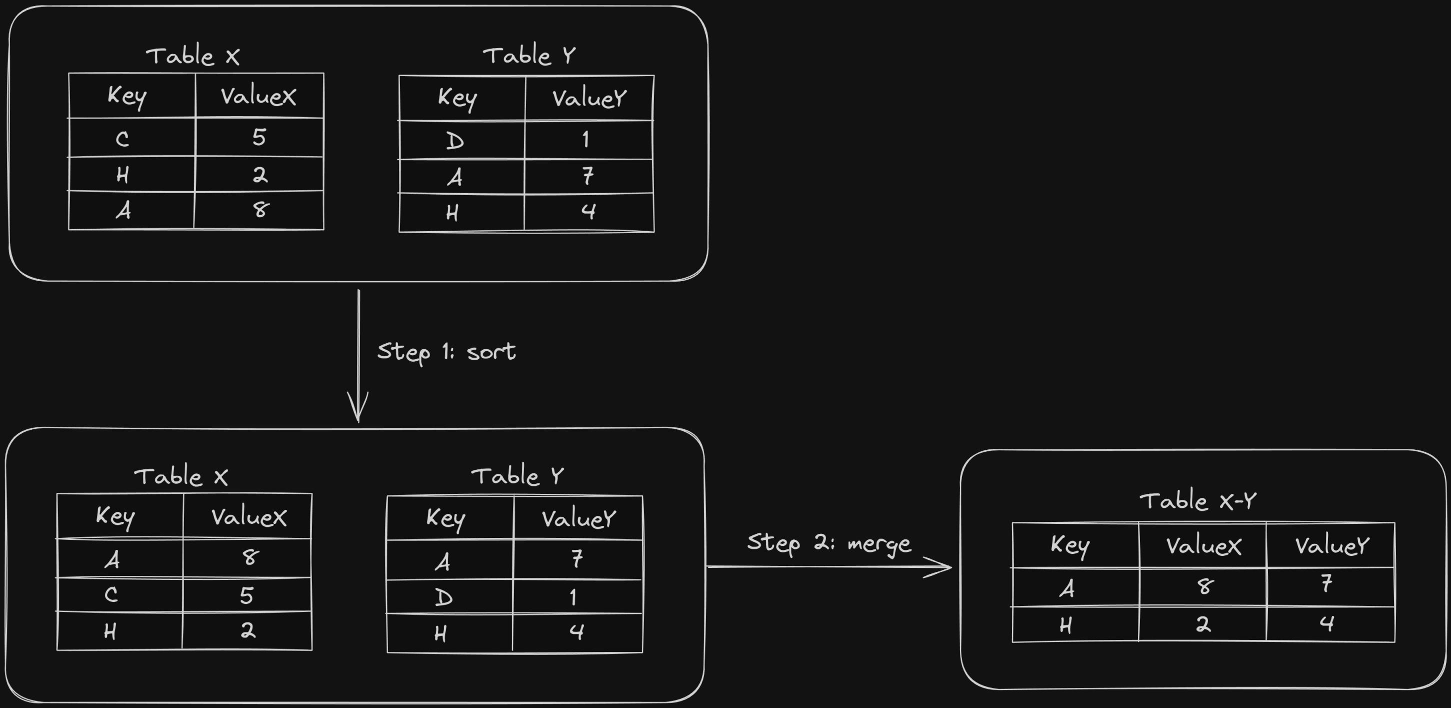
```sql
SELECT
  asset_id,
  maxMerge(tstamp) AS tstamp,
  argMaxMerge(price) AS price
FROM prices_latest
GROUP BY asset_id;
```

Results

```
┌─asset_id─┬─tstamp─┬─price─┐
│ a        │      9 │  2.42 │
│ b        │      9 │  2.61 │
└──────────┴────────┴───────┘
```

Sort-Merge Joins

Join algorithm where tables are individually sorted, then merged



Real merge JOIN support. #34236

Open **alexey-milovidov** opened this issue on Feb 2, 2022 · 17 comments

alexey-milovidov commented on Feb 2, 2022 Member ···

Use case

JOIN two large tables,
especially when the JOIN key is similar to the order key of these tables.

Describe the solution you'd like

If the data from either or both sides of JOIN can be transformed to the order by any of permutations of the JOIN key by `FinishSortingTransform`, we should apply this transform. If it cannot, we should order data similar to external sorting.

Then we JOIN two sorted streams of data.

The choice of this algorithm should be tuned by `join_algorithm` setting.
`join_algorithm = 'merge_in_order'` - use merge join algorithm if data can be finish-sorted from the table's primary key or subquery's ORDER BY clause;
`join_algorithm = 'merge'` - always use merge join algorithm, even if full sorting is needed.

Additional context

We already have "partial merge join" algorithm but it works different.

Integration into `join_algorithm = 'auto'` is out of scope of this task and will be the next step if this will play out.

👍 11

Optimisation: just don’t join?

Thank You!