Accelerating QuestDB: Lessons from A 6x Query Performance Boost

Jaromir Hamala

Core Engineer at QuestDB

Javier Ramirez

Database Advocate at QuestDB

QuestDB

The database is the bottleneck

- Every developer in the 90s

Common use case for a time-series database

Real-time dashboards on recent data.

Real-time decision making (i.e. payment fraud).

Historical queries aggregated by time chunks.

timestamp ↓	asset	counter	quantity	consideration	new
2024-08-08 14:48:31.221	BTC	USD	0.0300	-1724	x
2024-08-08 14:48:31.221	BTC	USD	0.0200	-1148	x
2024-08-08 14:48:30.914	ETH	USD	0.0218	-53.4	x
2024-08-08 14:48:30.913	ETH	DAI	0.00290	-7.12	x
2024-08-08 14:48:30.909	ETH	USD	0.0104	-25.5	x
2024-08-08 14:48:30.909	ETH	USD	0.00712	-17.5	x
2024-08-08 14:48:30.909	ETH	USD	0.00427	-10.5	x
2024-08-08 14:48:30.909	ETH	USD	0.0232	-57.0	x

Meet QuestDB: OSS time-series database

- https://github.com/questdb/questdb (Apache License 2.0)
- High-speed ingestion: InfluxDB line protocol over TCP or HTTP
- Columnar storage format (native or Parquet), partitioned and ordered by time
- Written in Java (90%) and C++/Rust (10%)
- Uses in-house replacement of Java's standard library
- Zero GC, SIMD, parallel SQL execution, SQL JIT compiler
- SQL with time-series extensions: PGWire, HTTP API

Time-series databases high level overview

- Time Series Databases specialise in very fast ingestion, very fast queries over nascent data, and powerful time-based analytical queries.

- They focus on nascent data, deleting, downsampling, or slowing-down older data.

Time-series SQL extensions

```sql
SELECT pickup_datetime, count() FROM trips
WHERE pickup_datetime in '2016-06-13;1M;1y;3'
SAMPLE BY 1w;

select timestamp, avg(price) from
(read_parquet('trades.parquet')
timestamp(timestamp)) sample by 15m;

SELECT * FROM trades
WHERE symbol in ('BTC-USDT', 'ETH-USDT')
LATEST ON timestamp PARTITION BY symbol;
```

```sql
SELECT
  timestamp, symbol, side, sum(amount) as volume
FROM trades
WHERE side = 'sell' AND timestamp IN today()
SAMPLE BY 1m FILL(NULL);

SELECT pickup_datetime, fare_amount, timestamp, tempF,
windDir FROM trips ASOF JOIN weather WHERE
pickup_datetime in '2018-06-01';
```

QuestDB in action: quick showcase

https://dashboard.demo.questdb.io/d/fb13b4ab-b1c9-4a54-a920-b60c5fb0363f/public-dashboard-questdb-io-use-cases-crypto?orgId=1&refresh=750ms

https://demo.questdb.io

https://github.com/questdb/time-series-streaming-analytics-template


YES,
Ratings & Reviews
5.0
out of 5
BUT
(1)
based on 1 review
_yes_but
Gudim.

What makes a decent analytical database?

- SQL
- Columnar storage format
- All HW resources (CPU & RAM) are available for faster query execution
- Complex queries with GROUP BY / JOIN / filter over large volumes of data, not necessarily accessed over time

How do you improve analytical DB capabilities?

- ClickBench - https://github.com/ClickHouse/ClickBench
  - Results accepted by ClickHouse: https://benchmark.clickhouse.com
- db-benchmark - https://github.com/duckdblabs/db-benchmark
  - Results accepted by DuckDB: https://duckdblabs.github.io/db-benchmark
- TPC benchmarks - https://www.tpc.org
- TSBS - https://github.com/timescale/tsbs
  - Time-series specific, not maintained

ClickBench

- Created by ClickHouse team in 2022
- Single table with 105 columns and 99M rows (Yandex search events)
- Includes data import, e.g. in CSV, but the main focus is on queries
- 43 queries with complex GROUP BY, WHERE, and ORDER BY clauses
- Only a few of the queries make use of time (QuestDB was already fast there)
- Run on different machines, but most popular are AWS EC2 instances with EBS volumes

https://github.com/ClickHouse/ClickBench/tree/main

Some sample queries

```
SELECT COUNT(*) FROM hits;
SELECT COUNT(*) FROM hits WHERE AdvEngineID <> 0;
SELECT count_distinct(UserID) FROM hits;
SELECT count_distinct(SearchPhrase) FROM hits;
SELECT UserID FROM hits WHERE UserID = 435090932899640449;

SELECT ClientIP, ClientIP - 1, ClientIP - 2, ClientIP - 3, COUNT(*) AS c FROM hits GROUP
BY ClientIP, ClientIP - 1, ClientIP - 2, ClientIP - 3 ORDER BY c DESC LIMIT 10;

SELECT TraficSourceID, SearchEngineID, AdvEngineID, CASE WHEN (SearchEngineID = 0 AND
AdvEngineID = 0) THEN Referer ELSE '' END AS Src, URL AS Dst, COUNT(*) AS PageViews FROM
hits WHERE CounterID = 62 AND EventTime >= '2013-07-01T00:00:00Z' AND EventTime <=
'2013-07-31T23:59:59Z' AND IsRefresh = 0 GROUP BY TraficSourceID, SearchEngineID,
AdvEngineID, Src, Dst ORDER BY PageViews DESC LIMIT 1000, 1010;

SELECT TraficSourceID, SearchEngineID, AdvEngineID, CASE WHEN (SearchEngineID = 0 AND
AdvEngineID = 0) THEN Referer ELSE '' END AS Src, URL AS Dst, COUNT(*) AS PageViews FROM
hits WHERE CounterID = 62 AND EventTime >= '2013-07-01T00:00:00Z' AND EventTime <=
'2013-07-31T23:59:59Z' AND IsRefresh = 0 GROUP BY TraficSourceID, SearchEngineID,
AdvEngineID, Src, Dst ORDER BY PageViews DESC LIMIT 1000, 1010;
```

Some sample queries

```
SELECT SUM(ResolutionWidth), SUM(ResolutionWidth + 1), SUM(ResolutionWidth + 2), SUM(ResolutionWidth + 3),
SUM(ResolutionWidth + 4), SUM(ResolutionWidth + 5), SUM(ResolutionWidth + 6), SUM(ResolutionWidth + 7),
SUM(ResolutionWidth + 8), SUM(ResolutionWidth + 9), SUM(ResolutionWidth + 10), SUM(ResolutionWidth + 11),
SUM(ResolutionWidth + 12), SUM(ResolutionWidth + 13), SUM(ResolutionWidth + 14), SUM(ResolutionWidth + 15),
SUM(ResolutionWidth + 16), SUM(ResolutionWidth + 17), SUM(ResolutionWidth + 18), SUM(ResolutionWidth + 19),
SUM(ResolutionWidth + 20), SUM(ResolutionWidth + 21), SUM(ResolutionWidth + 22), SUM(ResolutionWidth + 23),
SUM(ResolutionWidth + 24), SUM(ResolutionWidth + 25), SUM(ResolutionWidth + 26), SUM(ResolutionWidth + 27),
SUM(ResolutionWidth + 28), SUM(ResolutionWidth + 29), SUM(ResolutionWidth + 30), SUM(ResolutionWidth + 31),
SUM(ResolutionWidth + 32), SUM(ResolutionWidth + 33), SUM(ResolutionWidth + 34), SUM(ResolutionWidth + 35),
SUM(ResolutionWidth + 36), SUM(ResolutionWidth + 37), SUM(ResolutionWidth + 38), SUM(ResolutionWidth + 39),
SUM(ResolutionWidth + 40), SUM(ResolutionWidth + 41), SUM(ResolutionWidth + 42), SUM(ResolutionWidth + 43),
SUM(ResolutionWidth + 44), SUM(ResolutionWidth + 45), SUM(ResolutionWidth + 46), SUM(ResolutionWidth + 47),
SUM(ResolutionWidth + 48), SUM(ResolutionWidth + 49), SUM(ResolutionWidth + 50), SUM(ResolutionWidth + 51),
SUM(ResolutionWidth + 52), SUM(ResolutionWidth + 53), SUM(ResolutionWidth + 54), SUM(ResolutionWidth + 55),
SUM(ResolutionWidth + 56), SUM(ResolutionWidth + 57), SUM(ResolutionWidth + 58), SUM(ResolutionWidth + 59),
SUM(ResolutionWidth + 60), SUM(ResolutionWidth + 61), SUM(ResolutionWidth + 62), SUM(ResolutionWidth + 63),
SUM(ResolutionWidth + 64), SUM(ResolutionWidth + 65), SUM(ResolutionWidth + 66), SUM(ResolutionWidth + 67),
SUM(ResolutionWidth + 68), SUM(ResolutionWidth + 69), SUM(ResolutionWidth + 70), SUM(ResolutionWidth + 71),
SUM(ResolutionWidth + 72), SUM(ResolutionWidth + 73), SUM(ResolutionWidth + 74), SUM(ResolutionWidth + 75),
SUM(ResolutionWidth + 76), SUM(ResolutionWidth + 77), SUM(ResolutionWidth + 78), SUM(ResolutionWidth + 79),
SUM(ResolutionWidth + 80), SUM(ResolutionWidth + 81), SUM(ResolutionWidth + 82), SUM(ResolutionWidth + 83),
SUM(ResolutionWidth + 84), SUM(ResolutionWidth + 85), SUM(ResolutionWidth + 86), SUM(ResolutionWidth + 87),
SUM(ResolutionWidth + 88), SUM(ResolutionWidth + 89) FROM hits;
```

https://tinyurl.com/clickbench-2022-10

ClickBench — a Benchmark For Analytical DBMS

Methodology | Reproduce and Validate the Results | Add a System | Report Mistake | Hardware Benchmark

System: All | Athena (partitioned) | Athena (single) | Aurora for MySQL | Aurora for PostgreSQL | ByteHouse | Citus | clickhouse-local (partitioned) | clickhouse-local (single) | ClickHouse | ClickHouse (zstd) | CrateDB | Databend | datafusion | Druid | DuckDB | Elasticsearch | Elasticsearch (tuned) | Greenplum | HeavyAI | Infobright | MariaDB ColumnStore | MariaDB | MonetDB | MongoDB | MySQL (MyISAM) | MySQL | Pinot | PostgreSQL | QuestDB (partitioned) | QuestDB | Redshift | SingleStore | Snowflake | SQLite | StarRocks (tuned) | StarRocks | TimescaleDB (compression) | TimescaleDB

Type: All | stateless | managed | Java | column-oriented | C++ | MySQL compatible | row-oriented | C | PostgreSQL compatible | ClickHouse derivative | embedded | Rust | search | document | time-series

Machine: All | serverless | 16acu | L | M | S | XS | c6a.4xlarge, 500gb gp2 | c6a.metal, 500gb gp2 | f16s v2 | c6a.4xlarge, 1500gb gp2 | ra3.16xlarge | ra3.4xlarge | ra3.xlplus | S24 | S2 | 2XL | 3XL | 4XL | XL

Cluster size: All | 1 | 2 | 4 | 8 | 12 | 16 | 32 | 64 | 128 | serverless | undefined

Metric: Cold Run | Hot Run | Load Time | Storage Size

System & Machine	Relative time (lower is better)
ClickHouse (c6a.metal, 500gb gp2):	×1.30
StarRocks (c6a.metal, 500gb gp2):	×1.69
StarRocks (c6a.4xlarge, 500gb gp2):	×3.34
ClickHouse (c6a.4xlarge, 500gb gp2):	×3.54
ByteHouse (L):	×3.77
Redshift (4×ra3.4xlarge):	×3.86
Snowflake (8×L):	×4.50
SingleStore (2×S2)†:	×5.30
ByteHouse (M):	×5.88
Snowflake (4×M):	×6.16
Redshift (4×ra3.xlplus):	×6.92
ByteHouse (S):	×7.53
Snowflake (2×S):	×8.62
MonetDB (c6a.4xlarge, 500gb gp2):	×8.81
SingleStore (c6a.4xlarge, 500gb gp2)†:	×9.40
ByteHouse (XS):	×12.55
Snowflake (XS):	×12.55
DuckDB (c6a.4xlarge, 500gb gp2)†:	×22.44
Pinot (c6a.4xlarge, 500gb gp2)†:	×22.54
Greenplum (c6a.4xlarge, 500gb gp2):	×26.33
datafusion (f16s v2):	×31.84
QuestDB (c6a.4xlarge, 500gb gp2):	×33.89
Databend (c6a.4xlarge, 500gb gp2):	×33.93
MariaDB ColumnStore (c6a.4xlarge, 500gb gp2)†:	×46.38
CrateDB (c6a.4xlarge, 500gb gp2)†:	×47.81
Elasticsearch (c6a.4xlarge, 1500gb gp2):	×57.81
TimescaleDB (compression) (c6a.4xlarge, 500gb gp2):	×68.01
Druid (c6a.4xlarge, 500gb gp2)†:	×117.77
HeavyAI (c6a.4xlarge, 500gb gp2)†:	×127.44
Citus (c6a.4xlarge, 500gb gp2):	×173.17

https://tinyurl.com/clickbench-2023-08-18

https://tinyurl.com/clickbench-2024-09

System: All, AlloyDB, AlloyDB (tuned), Athena (partitioned), Athena (single), Aurora for MySQL, Aurora for PostgreSQL, ByConity, ByteHouse, chDB (DataFrame), chDB (Parquet, partitioned), chDB, Citus, ClickHouse Cloud (aws), ClickHouse Cloud (aws) Parallel Replicas ON, ClickHouse Cloud (Azure), ClickHouse Cloud (Azure) Parallel Replica ON, ClickHouse Cloud (Azure) Parallel Replicas ON, ClickHouse Cloud (gcp), ClickHouse Cloud (gcp) Parallel Replicas ON, ClickHouse (data lake, partitioned), ClickHouse (data lake, single), ClickHouse (Parquet, partitioned), ClickHouse (Parquet, single), ClickHouse (web), ClickHouse, ClickHouse (tuned), ClickHouse (tuned, memory), Cloudberry, CrateDB, Crunchy Bridge for Analytics (Parquet), Databend, DataFusion (Parquet, partitioned), DataFusion (Parquet, single), Apache Doris, Druid, DuckDB (DataFrame), DuckDB (Parquet, partitioned), DuckDB, Elasticsearch, Elasticsearch (tuned), GlareDB, Greenplum, HeavyAI, Hydra, Infobright, Kinetica, MariaDB ColumnStore, MariaDB, MonetDB, MongoDB, Motherduck, MySQL (MyISAM), MySQL, Oxla, Pandas (DataFrame), ParadeDB (Parquet, partitioned), ParadeDB (Parquet, single), Pinot, Polars (DataFrame), PostgreSQL (tuned), PostgreSQL, QuestDB (partitioned), QuestDB, Redshift, SingleStore, Snowflake, SQLite, StarRocks, Tablespace, Tembo OLAP (columnar), TimescaleDB (compression), TimescaleDB, Umbra

Type: All, C, column-oriented, PostgreSQL compatible, managed, gcp, stateless, Java, C++, MySQL compatible, row-oriented, ClickHouse derivative, embedded, serverless, dataframe, aws, parallel replicas, Azure, analytical, Rust, search, document, somewhat PostgreSQL compatible, time-series

Machine: All, 16 vCPU 128GB, 8 vCPU 64GB, serverless, 16acu, c6a.4xlarge, 500gb gp2, L, M, S, XS, c6a.metal, 500gb gp2, 192GB, 24GB, 360GB, 48GB, 720GB, 96GB, 1430GB, dev, 708GB, c5n.4xlarge, 500gb gp2, Analytics-256GB (64 vCores, 256 GB), c5.4xlarge, 500gb gp2, c6a.4xlarge, 1500gb gp2, cloud, dc2.8xlarge, ra3.16xlarge, ra3.4xlarge, ra3.xlplus, S2, S24, 2XL, 3XL, 4XL, XL, L1 - 16CPU 32GB, c6a.4xlarge, 500gb gp3

Cluster size: All, 1, 2, 4, 8, 16, 32, 64, 128, serverless, dedicated

Metric: Cold Run, Hot Run, Load Time, Storage Size

System & Machine	Relative time (lower is better)
Umbra (c6a.metal, 500gb gp2):	×1.57
Apache Doris (c6a.metal, 500gb gp2):	×2.10
ClickHouse (c6a.metal, 500gb gp2):	×2.15
StarRocks (c6a.metal, 500gb gp2):	×2.32
Umbra (c6a.4xlarge, 500gb gp2):	×2.34
Databend (c6a.metal, 500gb gp2):	×2.40
DuckDB (c6a.metal, 500gb gp2):	×2.60
QuestDB (partitioned) (c6a.metal, 500gb gp2)†:	×3.18
SingleStore (S24)†:	×3.90
DuckDB (c6a.4xlarge, 500gb gp2):	×4.29
ClickHouse (c6a.4xlarge, 500gb gp2):	×4.34
Databend (c6a.4xlarge, 500gb gp2):	×4.45
chDB (DataFrame) (c6a.metal, 500gb gp2):	×4.81
Databend (c5.4xlarge, 500gb gp2):	×4.82
DuckDB (c5.4xlarge, 500gb gp2):	×5.07
Apache Doris (c6a.4xlarge, 500gb gp2):	×5.59
StarRocks (c6a.4xlarge, 500gb gp2):	×5.63
Snowflake (32×2XL):	×5.91
chDB (c6a.metal, 500gb gp2):	×6.09
Tablespace (L1 - 16CPU 32GB):	×6.75
QuestDB (c6a.4xlarge, 500gb gp2):	×7.15
ByConity (c6a.4xlarge, 500gb gp2):	×7.56
ByteHouse (8×L):	×7.64

https://tinyurl.com/clickbench-2025-01-29

System: All, AlloyDB, AlloyDB (tuned), Athena (partitioned), Athena (single), Aurora for MySQL, Aurora for PostgreSQL, ByConity, ByteHouse, chDB (DataFrame), chDB (Parquet, partitioned), chDB, Citus, ClickHouse Cloud (aws), ClickHouse Cloud (azure), ClickHouse Cloud (gcp), ClickHouse (data lake, partitioned), ClickHouse (data lake, single), ClickHouse (Parquet, partitioned), ClickHouse (Parquet, single), ClickHouse (web), ClickHouse, ClickHouse (tuned), ClickHouse (tuned, memory), Cloudberry, CrateDB, Crunchy Bridge for Analytics (Parquet), Databend, DataFusion (Parquet, partitioned), DataFusion (Parquet, single), Apache Doris, Drill, Druid, DuckDB (DataFrame), DuckDB (memory), DuckDB (Parquet, partitioned), DuckDB, Elasticsearch, Elasticsearch (tuned), GlareDB, Greenplum, HeavyAI, Hydra, Salesforce Hyper (Parquet), Salesforce Hyper, Infobright, Kinetica, MariaDB ColumnStore, MariaDB, MonetDB, MongoDB, MotherDuck, MySQL (MyISAM), MySQL, OctoSQL, Opteryx, Oxla, Pandas (DataFrame), ParadeDB (Parquet, partitioned), ParadeDB (Parquet, single), pg_duckdb (MotherDuck enabled), pg_duckdb, PostgreSQL with pg_mooncake, Pinot, Polars (DataFrame), Polars (Parquet), PostgreSQL (tuned), PostgreSQL, QuestDB, Redshift, SelectDB, SingleStore, Snowflake, Spark, SQLite, StarRocks, Tablespace, Tembo OLAP (columnar), Timescale Cloud, TimescaleDB (no columnstore), TimescaleDB, Tinybird (Free Trial), Umbra, VictoriaLogs

Type: All, C, column-oriented, PostgreSQL compatible, managed, gcp, stateless, Java, C++, MySQL compatible, row-oriented, ClickHouse derivative, embedded, serverless, dataframe, aws, azure, analytical, Rust, search, document, Go, somewhat PostgreSQL compatible, DataFrame, parquet, time-series

Machine: All, 16 vCPU 128GB, 8 vCPU 64GB, serverless, 16acu, c6a.4xlarge, 500gb gp2, L, M, S, XS, c6a.metal, 500gb gp2, 12GiB, 8GiB, 120GiB, 16GiB, 236GiB, 32GiB, 64GiB, c5n.4xlarge, 500gb gp2, Analytics-256GB (64 vCores, 256 GB), c5.4xlarge, 500gb gp2, c6a.4xlarge, 1500gb gp2, cloud, dc2.8xlarge, ra3.16xlarge, ra3.4xlarge, ra3.xlplus, S2, S24, 2XL, 3XL, 4XL, XL, L1 - 16CPU 32GB, c6a.4xlarge, 500gb gp3, 16 vCPU 64GB, 4 vCPU 16GB, 8 vCPU 32GB

Cluster size: All, 1, 2, 4, 8, 16, 32, 64, 128, serverless, 1, 2, 3, undefined

Metric: Cold Run, Hot Run, Load Time, Storage Size

System & Machine	Relative time (lower is better)
Umbra (c6a.metal, 500gb gp2):	×1.51
Salesforce Hyper (c6a.metal, 500gb gp2):	×1.58
ClickHouse (c6a.metal, 500gb gp2):	×2.39
Umbra (c6a.4xlarge, 500gb gp2):	×2.43
SelectDB (c6a.metal, 500gb gp2):	×2.46
Apache Doris (c6a.metal, 500gb gp2):	×2.49
DuckDB (c6a.metal, 500gb gp2):	×2.73
StarRocks (c6a.metal, 500gb gp2):	×2.74
Databend (c6a.metal, 500gb gp2):	×2.84
chDB (c6a.metal, 500gb gp2):	×2.97
QuestDB (c6a.metal, 500gb gp2)†:	×3.04
Salesforce Hyper (Parquet) (c6a.metal, 500gb gp2):	×3.19
Salesforce Hyper (c6a.4xlarge, 500gb gp2):	×3.65
SingleStore (S24)†:	×4.62
DuckDB (c6a.4xlarge, 500gb gp2):	×4.89
Databend (c6a.4xlarge, 500gb gp2):	×5.26
ClickHouse (c6a.4xlarge, 500gb gp2):	×5.32
chDB (c6a.4xlarge, 500gb gp2):	×5.53
Databend (c5.4xlarge, 500gb gp2):	×5.71
QuestDB (c6a.4xlarge, 500gb gp2):	×6.35
Apache Doris (c6a.4xlarge, 500gb gp2):	×6.62

https://tinyurl.com/clickbench-2025-01-29

Same hardware (c6a.metal 500gb gp2)

System: All | AlloyDB | AlloyDB (tuned) | Athena (partitioned) | Athena (single) | Aurora for MySQL | Aurora for PostgreSQL | ByConity | ByteHouse | chDB (DataFrame) | chDB (Parquet, partitioned) | chDB | Citus | ClickHouse Cloud (aws) | ClickHouse Cloud (azure) | ClickHouse Cloud (gcp) | ClickHouse (data lake, partitioned) | ClickHouse (data lake, single) | ClickHouse (Parquet, partitioned) | ClickHouse (Parquet, single) | ClickHouse (web) | ClickHouse | ClickHouse (tuned) | ClickHouse (tuned, memory) | Cloudberry | CrateDB | Crunchy Bridge for Analytics (Parquet) | Databend | DataFusion (Parquet, partitioned) | DataFusion (Parquet, single) | Apache Doris | Drill | Druid | DuckDB (DataFrame) | DuckDB (memory) | DuckDB (Parquet, partitioned) | DuckDB | Elasticsearch | Elasticsearch (tuned) | GlareDB | Greenplum | HeavyAI | Hydra | Salesforce Hyper (Parquet) | Salesforce Hyper | Infobright | Kinetica | MariaDB ColumnStore | MariaDB | MonetDB | MongoDB | MotherDuck | MySQL (MyISAM) | MySQL | OctoSQL | Opteryx | Oxla | Pandas (DataFrame) | ParadeDB (Parquet, partitioned) | ParadeDB (Parquet, single) | pg_duckdb (MotherDuck enabled) | pg_duckdb | PostgreSQL with pg_mooncake | Pinot | Polars (DataFrame) | Polars (Parquet) | PostgreSQL (tuned) | PostgreSQL | QuestDB | Redshift | SelectDB | SingleStore | Snowflake | Spark | SQLite | StarRocks | Tablespace | Tembo OLAP (columnar) | Timescale Cloud | TimescaleDB (no columnstore) | TimescaleDB | Tinybird (Free Trial) | Umbra | VictoriaLogs

Type: All | C | column-oriented | PostgreSQL compatible | managed | gcp | stateless | Java | C++ | MySQL compatible | row-oriented | ClickHouse derivative | embedded | serverless | dataframe | aws | azure | analytical | Rust | search | document | Go | somewhat PostgreSQL compatible | DataFrame | parquet | time-series

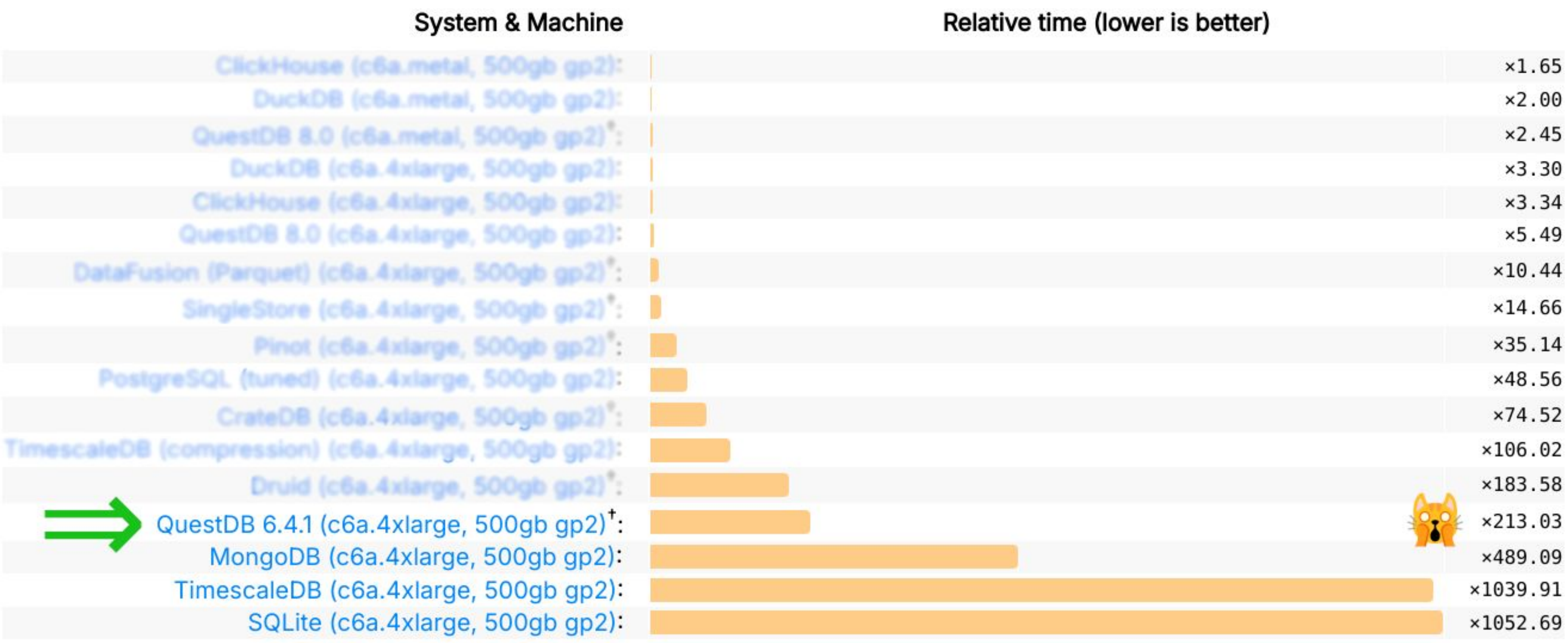
Machine: All | 16 vCPU 128GB | 8 vCPU 64GB | serverless | 16acu | c6a.4xlarge, 500gb gp2 | L | M | S | XS | c6a.metal, 500gb gp2 | 12GiB | 8GiB | 120GiB | 16GiB | 236GiB | 32GiB | 64GiB | c5n.4xlarge, 500gb gp2 | Analytics-256GB (64 vCores, 256 GB) | c5.4xlarge, 500gb gp2 | c6a.4xlarge, 1500gb gp2 | cloud | dc2.8xlarge | ra3.16xlarge | ra3.4xlarge | ra3.xlplus | S2 | S24 | 2XL | 3XL | 4XL | XL | L1 - 16CPU 32GB | c6a.4xlarge, 500gb gp3 | 16 vCPU 64GB | 4 vCPU 16GB | 8 vCPU 32GB

Cluster size: All | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | serverless | 1 | 2 | 3 | undefined

Metric: Cold Run | Hot Run | Load Time | Storage Size

System & Machine	Relative time (lower is better)
Umbra (c6a.metal, 500gb gp2):	×1.31
Salesforce Hyper (c6a.metal, 500gb gp2):	×1.37
ClickHouse (c6a.metal, 500gb gp2):	×2.07
SelectDB (c6a.metal, 500gb gp2):	×2.14
Apache Doris (c6a.metal, 500gb gp2):	×2.15
DuckDB (c6a.metal, 500gb gp2):	×2.37
StarRocks (c6a.metal, 500gb gp2):	×2.38
Databend (c6a.metal, 500gb gp2):	×2.46
chDB (c6a.metal, 500gb gp2):	×2.57
QuestDB (c6a.metal, 500gb gp2)†:	×2.64
Salesforce Hyper (Parquet) (c6a.metal, 500gb gp2):	×2.77
Polars (Parquet) (c6a.metal, 500gb gp2):	×14.14
Polars (DataFrame) (c6a.metal, 500gb gp2):	×14.39
GlareDB (c6a.metal, 500gb gp2):	×97.33
Pandas (DataFrame) (c6a.metal, 500gb gp2):	×260.04

QuestDB in ClickBench: how it started



The Journey, or There and Back Again

- ~2 years of calendar time
- Done along with major features: Write-Ahead-Log (WAL), replication, window functions, Parquet and JSON support, etc.
- ~80 patches, including community contributions
- A number of failed optimization attempts
- Even more plans for further steps

Trivial steps

- Added missing SQL functions, e.g. count_distinct() for integer column types or max()/min() on strings
- Reduced memory footprint of some SQL functions to avoid OOM crashes

```
SELECT RegionID, count_distinct(UserID) AS u
FROM hits
GROUP BY RegionID
ORDER BY u DESC
LIMIT 10;
```

QuestDB's JIT compiler

- SQL JIT compiler for filters (WHERE clauses)
- Backend is written in C++ with asmjit library, frontend is in Java
- Emits SIMD (AVX-2) instructions for a subset of filters
- JIT compiled (and Java) filter execution is multi-threaded

```sql
SELECT count(*)
FROM hits
WHERE AdvEngineID <> 0;
```

Predicate Compile Time:

SQL (Java) -> AST (Java) -> IR (Java) -> machine code (C++)

Predicate Execution Time:

```
public static native long callFunction(
        long fnAddress,
        long colsAddress,
        long colsSize,
        long varSizeIndexesAddress,
        long varsAddress,
        long varsSize,
        long rowsAddress,
        long rowsSize,
        long rowsStartOffset
);
```

```cpp
inline Ymm cmp_eq_double(Compiler &c, data_type_t type, const Ymm &lhs, const Ymm &rhs) {
    Ymm lhs_copy = c.newYmm();
    Ymm rhs_copy = c.newYmm();
    c.vmovapd(lhs_copy, lhs);
    c.vmovapd(rhs_copy, rhs);
    Ymm dst = c.newYmm();
    Ymm nans = mask_and(c, is_nan(c, type, lhs_copy), is_nan(c, type, rhs_copy));
    Mem sign_mask = vec_sign_mask(c, type);
    c.vsubpd(lhs_copy, lhs_copy, rhs_copy); //(lhs - rhs)
    c.vpand(lhs_copy, lhs_copy, sign_mask); // abs(lhs - rhs)
    double eps[4] = {DOUBLE_EPSILON, DOUBLE_EPSILON, DOUBLE_EPSILON, DOUBLE_EPSILON};
    Mem epsilon = c.newConst(ConstPool::kScopeLocal, &eps, 32);
    c.vcmppd(dst, lhs_copy, epsilon, Predicate::kCmpLT);
    c.vpor(dst, dst, nans);
    return dst;
}
```

JIT compiler improvements

- Expanded supported operators and types

```
SELECT URL, count(*) AS PageViews
FROM hits
WHERE CounterID = 62
  AND EventTime >= '2013-07-01T00:00:00Z'
  AND EventTime <= '2013-07-31T23:59:59Z'
  AND DontCountHits = 0
  AND IsRefresh = 0
  AND URL IS NOT NULL
GROUP BY URL
ORDER BY PageViews DESC
LIMIT 10;
```

SQL rewrites

```
SELECT count_distinct(SearchPhrase)
FROM hits;

-- gets rewritten into:
SELECT count(*)
FROM (
  SELECT SearchPhrase
  FROM hits
  WHERE SearchPhrase IS NOT NULL
  GROUP BY SearchPhrase
);
```

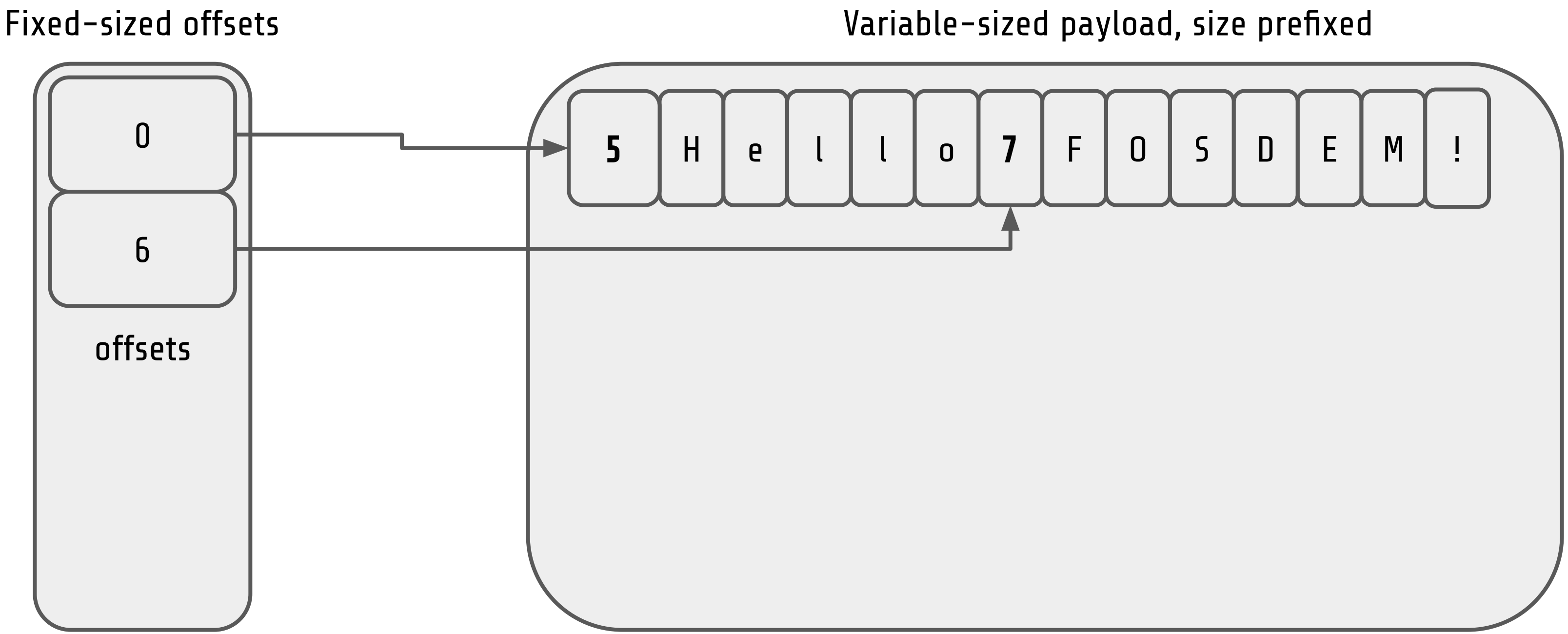
SQL function optimizations #1

```
-- uses SWAR-based LIKE operator implementation
SELECT count(*)
FROM hits
WHERE URL LIKE '%google%';
```

SQL function optimizations #2

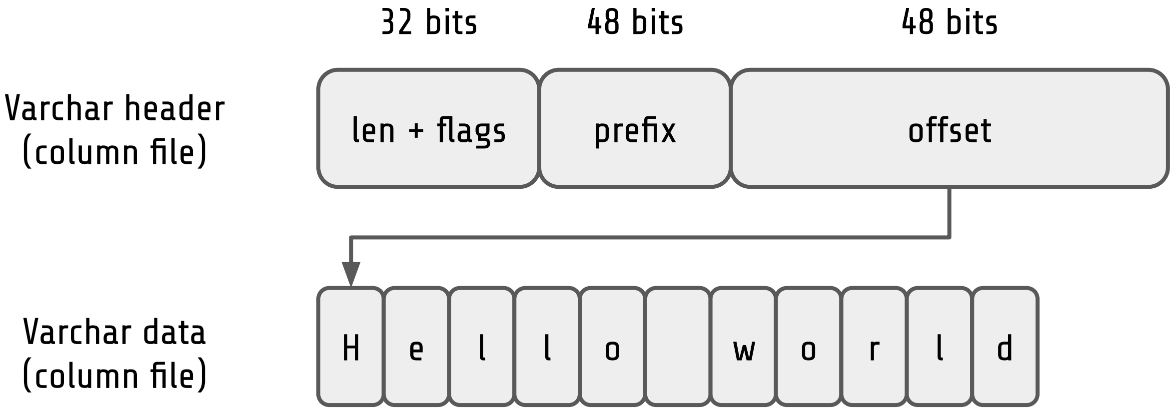
```
-- regexp_replace() uses Java regular expressions, but with a few fast paths
SELECT *
FROM (
  SELECT
    regexp_replace(Referer, '^https?://(?:www\.)?([^/]+)/.*$', '$1') AS k,
    avg(length(Referer)) AS l,
    count(*) AS c,
    min(Referer)
  FROM hits
  WHERE Referer IS NOT NULL
  GROUP BY k
)
WHERE c > 100000
ORDER BY l DESC
LIMIT 25;
```

Old STRING column type - UTF-16 encoded



New VARCHAR column type

- Introduced VARCHAR type (UTF-8) instead of old STRING type (UTF-16)
- Layout is similar to what Andy Pavlo calls "German Strings", but with some differences, including an ASCII bit flag



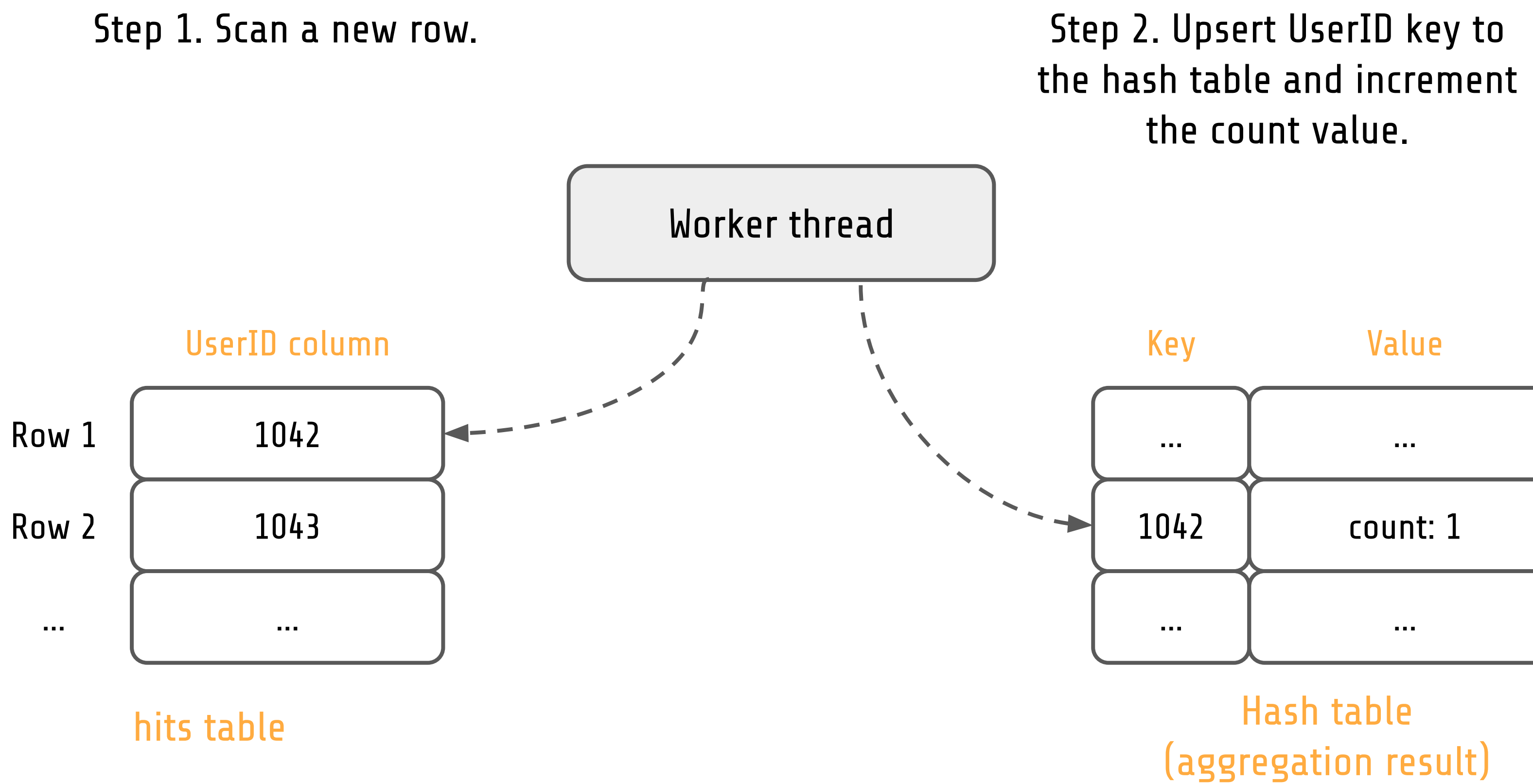
The elephant in the room

- Only a few GROUP BY queries ran parallel (and used SIMD)

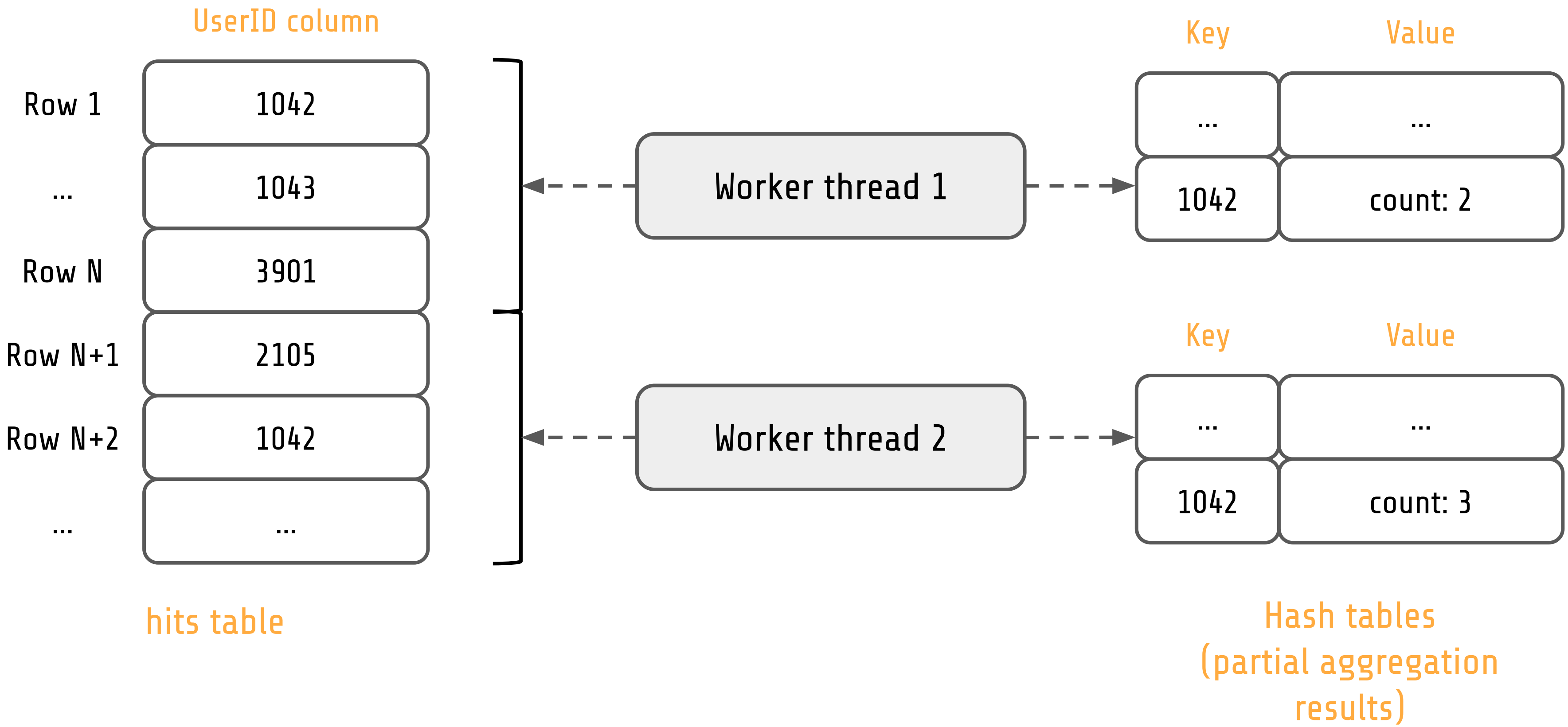
```
SELECT sum(AdvEngineID), count(*), avg(ResolutionWidth) FROM hits;

SELECT avg(UserID) FROM hits;

SELECT min(EventDate), max(EventDate) FROM hits;

SELECT sum(ResolutionWidth), sum(ResolutionWidth + 1), -- many more sums…
FROM hits;
```

How do you implement a GROUP BY?

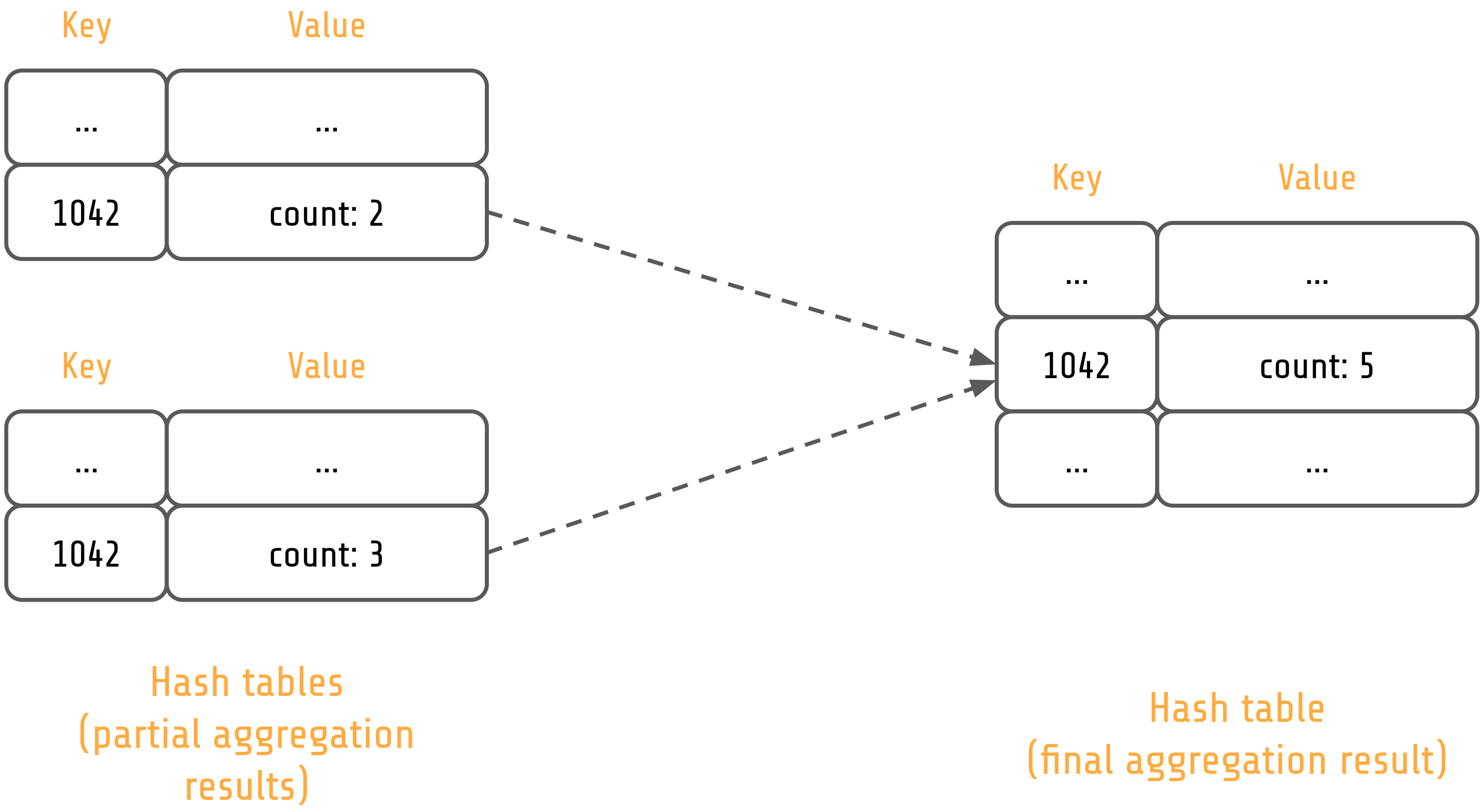
```sql
SELECT UserID, count(*) AS c
FROM hits
GROUP BY UserID;
```



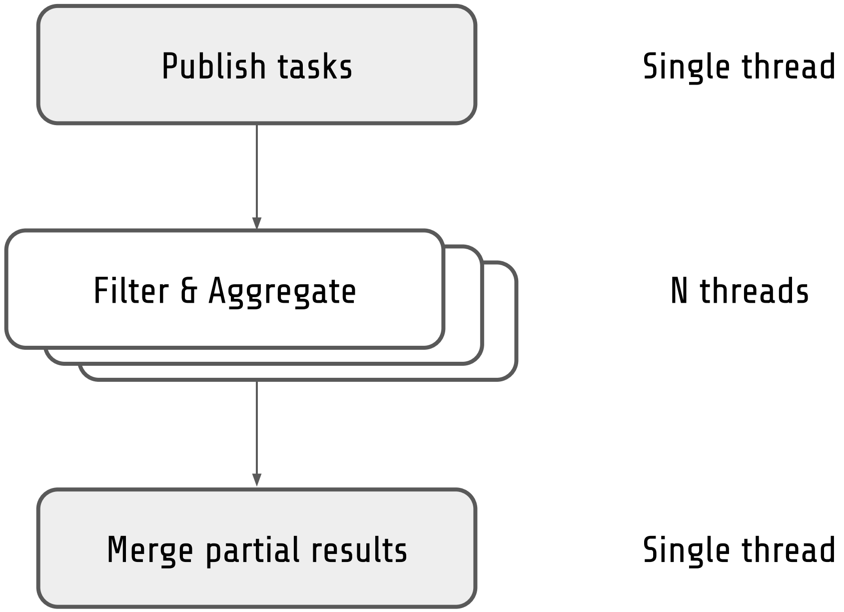
Single-threaded GROUP BY



Multi-threaded GROUP BY: aggregation



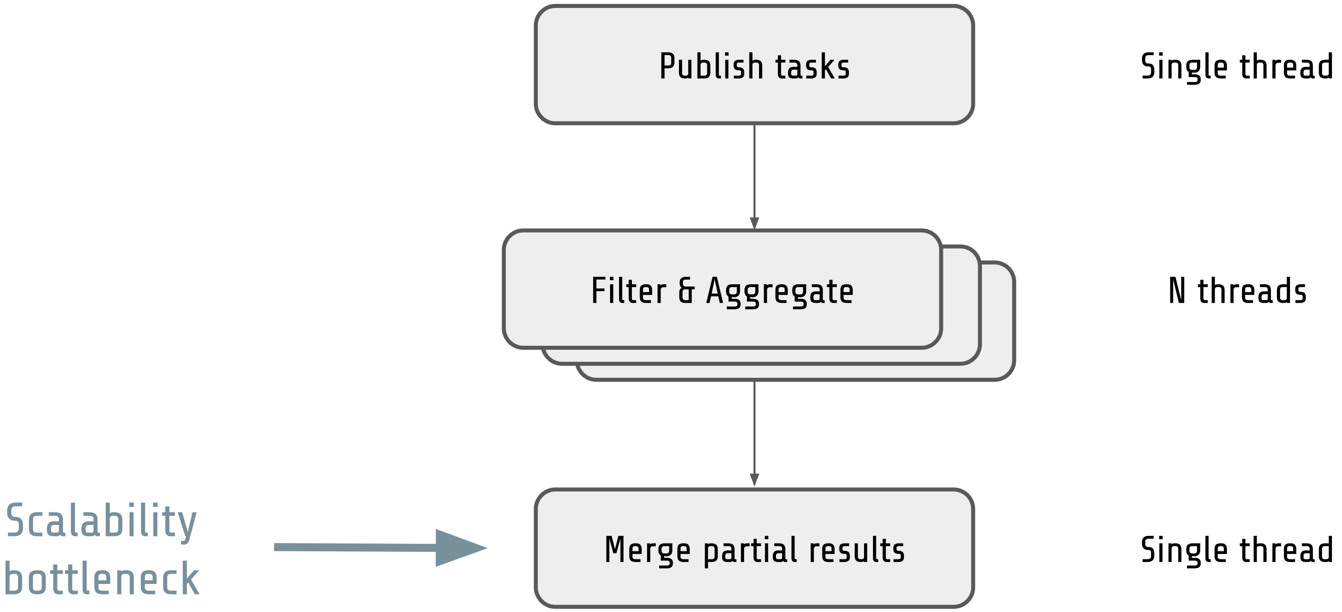
Multi-threaded GROUP BY: merge



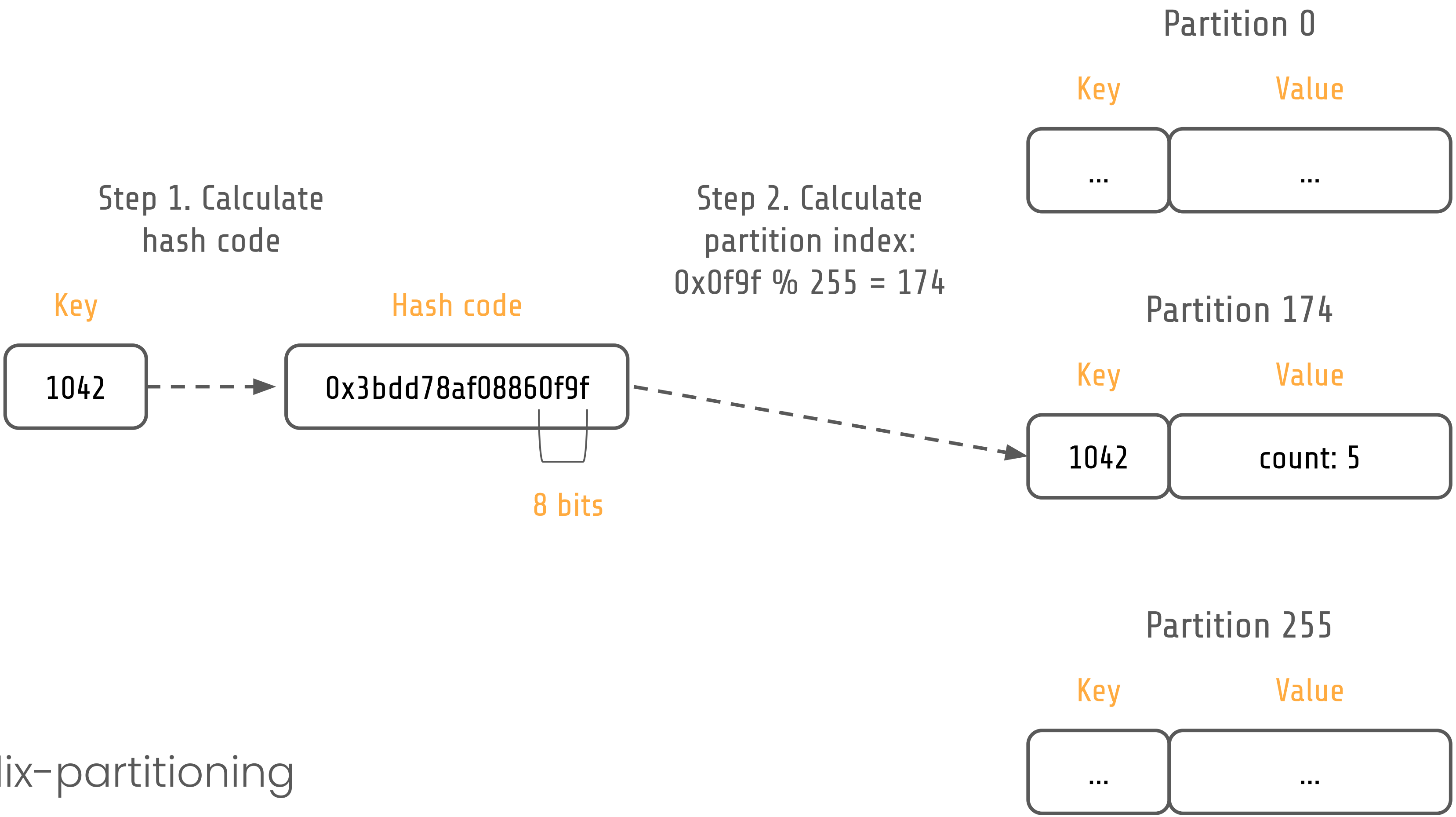
Multi-threaded GROUP BY pipeline

Parallel GROUP BY v1: any good?

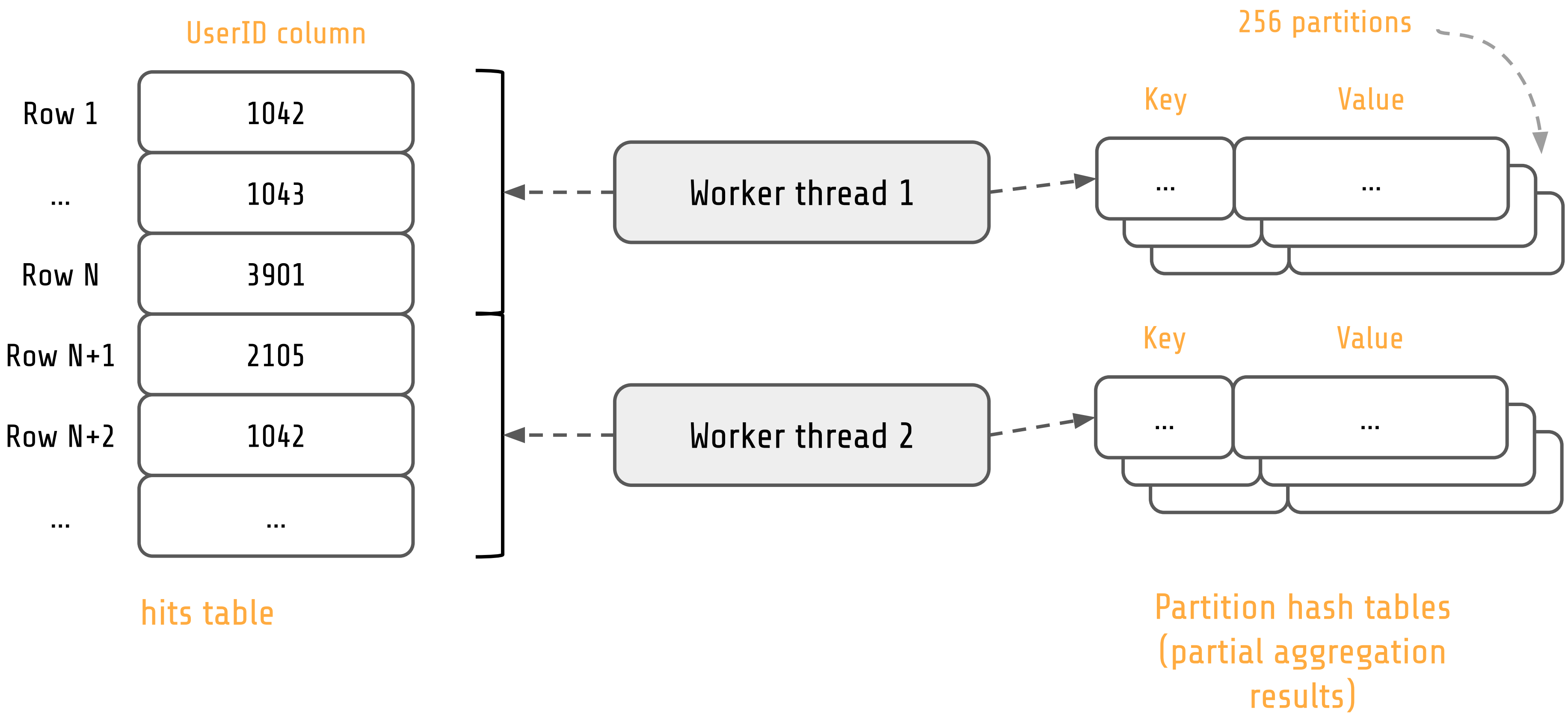
- Simple pipeline, easy to implement
- Scales nicely when there are not so many groups (distinct UserID values)
- Yet, high cardinality (>= 100K groups) is a problem



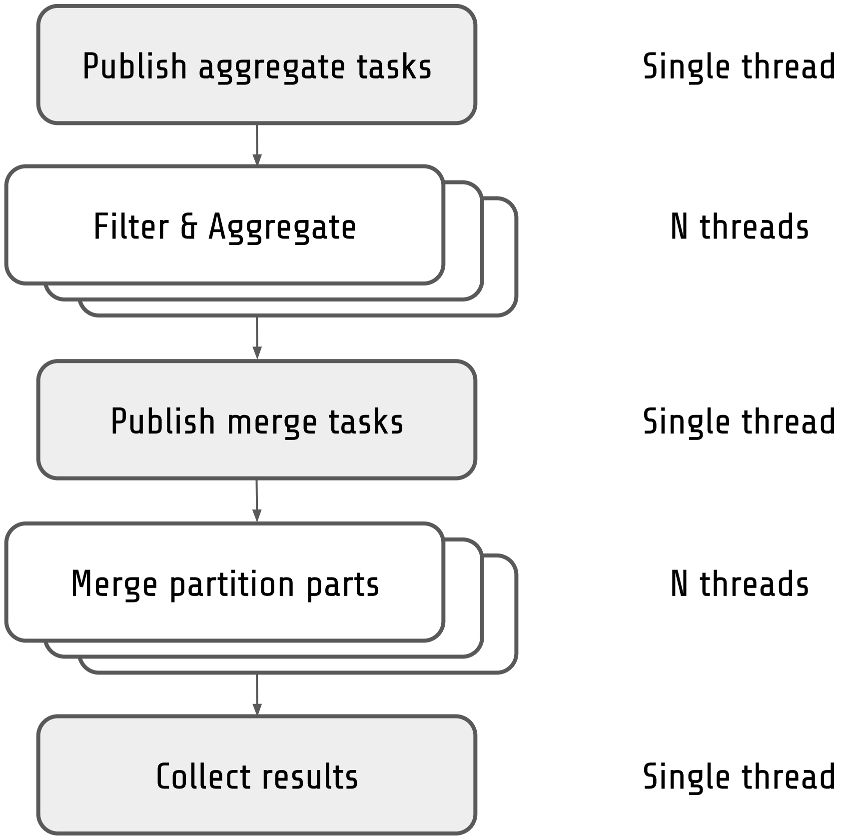
Multi-threaded GROUP BY pipeline: the cardinality problem



Radix-partitioning



High-cardinality multi-threaded GROUP BY



High-cardinality multi-threaded GROUP BY pipeline

Parallel GROUP BY v2

- More complex pipeline, a bit harder to implement
- Scales nicely for any cardinality
- Potentially parallel ORDER BY + LIMIT when the cardinality is high
- Used for multi-threaded GROUP BY and SAMPLE BY

```
SELECT RegionID, count_distinct(UserID) AS u FROM hits GROUP BY RegionID
ORDER BY u DESC LIMIT 10;
```

The more hash tables, the merrier

- Introduced a number of specialized hash tables
- All use open addressing with linear probing
- Some preserve insertion order

So far, we have:

- A "general purpose" hash table for variable-size keys
- Hash tables with small fixed-size keys (32-bit and 64-bit integers)
- A lookup table for 16-bit keys
- A hash table for single VARCHAR key

Lessons learned

- A fast time-series database must be a good analytical database
- Benchmarks made by 3rd-parties help when deciding what to optimize
- Improving query engine efficiency requires discipline
- As a nice side effect, we made SAMPLE BY run parallel
- We have lots of plans for the next steps.

https://github.com/questdb/questdb
https://questdb.io
https://demo.questdb.io
https://slack.questdb.io/
https://github.com/questdb/time-series-streaming-analytics-template

THANKS!

Jaromir Hamala

Core Engineer at QuestDB

Javier Ramirez

Database Advocate at QuestDB

@supercoco9
@supercoco9.bsky.social
@j@chaos.social
linkedin.com/in/ramirez

QuestDB