ClickHouse - Lightning Fast Analytics for Everyone

Robert Schulze
ClickHouse Inc.
robert@clickhouse.com

Tom Schreiber
ClickHouse Inc.
tom@clickhouse.com

Ilya Yatsishin
ClickHouse Inc.
iyatsishin@clickhouse.com

Ryadh Dahimene
ClickHouse Inc.
ryadh@clickhouse.com

Alexey Milovidov
ClickHouse Inc.
alexey@clickhouse.com

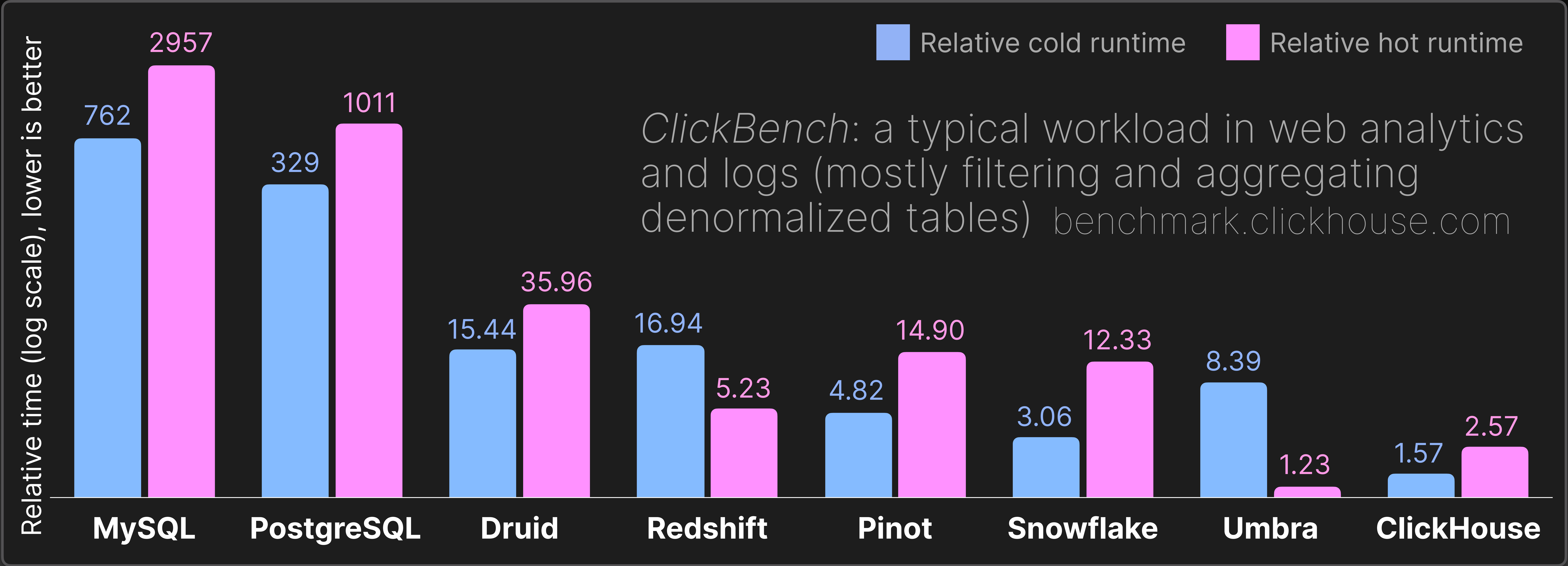
VLDB2024
GUANGZHOU

ClickHouse - Lightning Fast Analytics for Everyone

Open source column-oriented distributed OLAP database

Developed since 2009, built in C++ OSS (Apache 2.0) since 2016	Best for filter and aggregation queries Optimized for append-only workloads	Replication Sharding Eventually consistent	Business intelligence Logs, events, traces Real-time analytics

ClickHouse - Lightning Fast Analytics for Everyone

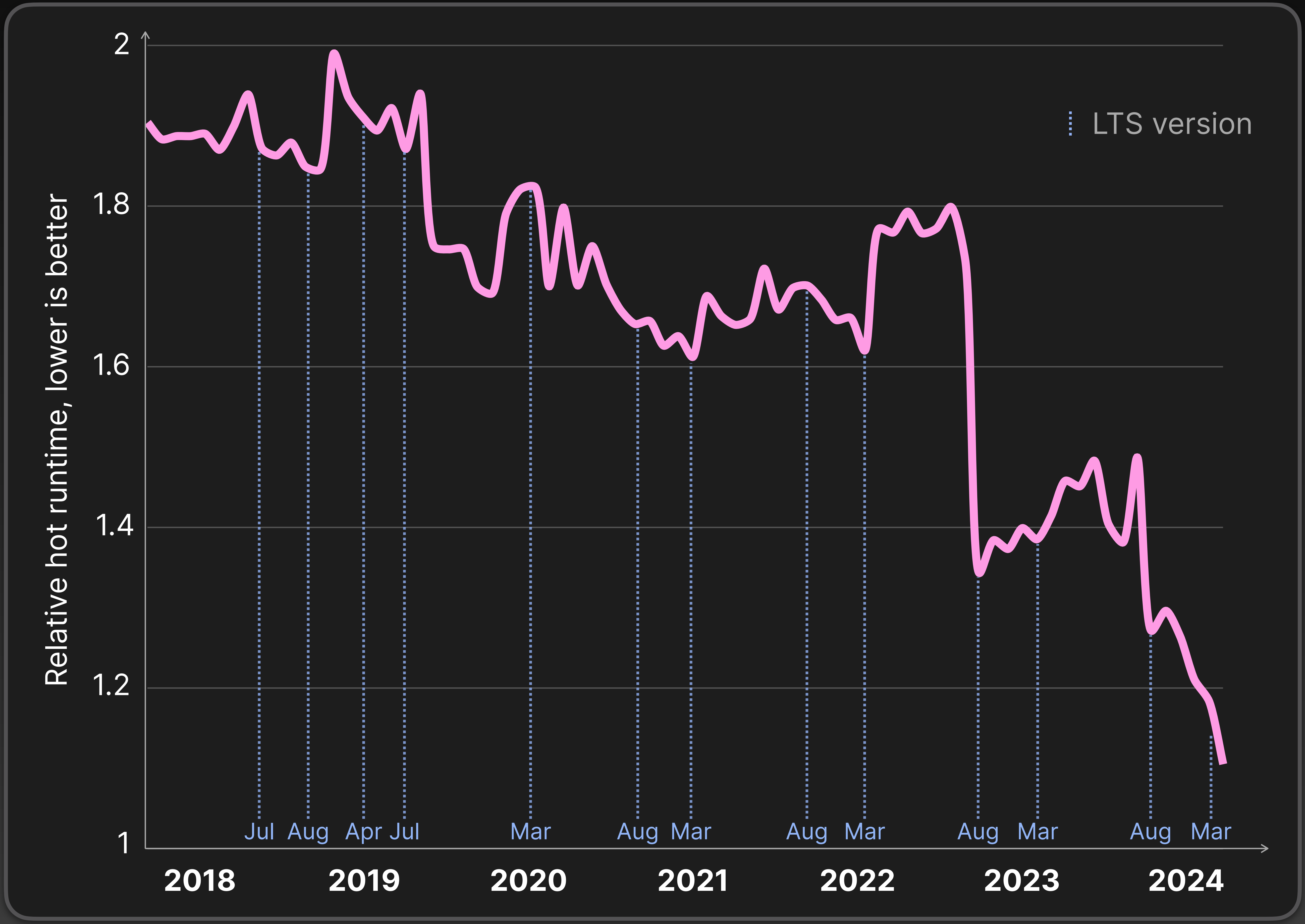


Total relative cold and hot runtimes for sequentially executing all ClickBench queries in databases frequently used for analytics.
Measurements taken on a single-node AWS EC2 c6a.4xlarge instance with 16 vCPUs, 32 GB RAM, and 5000 IOPS / 1000 MiB/s disk.
Comparable systems were used for Redshift (ra3.4xlarge, 12 vCPUs, 96 GB RAM) and Snowflake (warehouse size S: 2×8 vCPUs, 2×16 GB RAM).

ClickHouse has the best query performance amongst production-grade analytics databases.

ClickHouse - Lightning Fast Analytics for Everyone

Performance improvements by 1.72 × since 2018



- *VersionBench* benchmark is run when a new release is published to check its performance and identify regressions.
- Combination of four benchmarks:

	# Queries	# Rows
ClickBench	42	100 million
MgBench	15	200 million
Star Schema Benchmark (denormalized schema)	13	600 million
NYC Taxi Rides Benchmark	4	3.4 billion

Query performance is a top priority and continuously improved.

ClickHouse - Lightning Fast Analytics for Everyone

- **2009** — Start of development
- **2012** — Production launch
- **2016** — Open source
- **2017** — First integration engine (Kafka) (Section 5)
- **2018** — Mutations (Section 3)
- **2019** — New execution framework (Section 4)
- **2020** — Improved introspection tools (Section 6)
- **2021** — Async inserts, Projections (Section 3)
- **2022** — ClickHouse Cloud
- **2023** — Column statistics (section 3) CPU and IO scheduling (Section 4)
- **2024** — ... (Section 8)

The most popular analytics database on GitHub (**36k ⭐, 2k+ contributors**).

github.com/ClickHouse/ClickHouse

Runs on anything from Raspberry Pi to clusters with hundreds of nodes, largest known cluster is 4000 servers.

Used by hundreds of companies globally for production workloads.

clickhouse.com/docs/en/faq/general/who-is-using-clickhouse

ClickHouse is trusted by 50%+ of Fortunes Global Top 2000 companies.

Architecture

50+ integrations
with external systems

90+ file formats

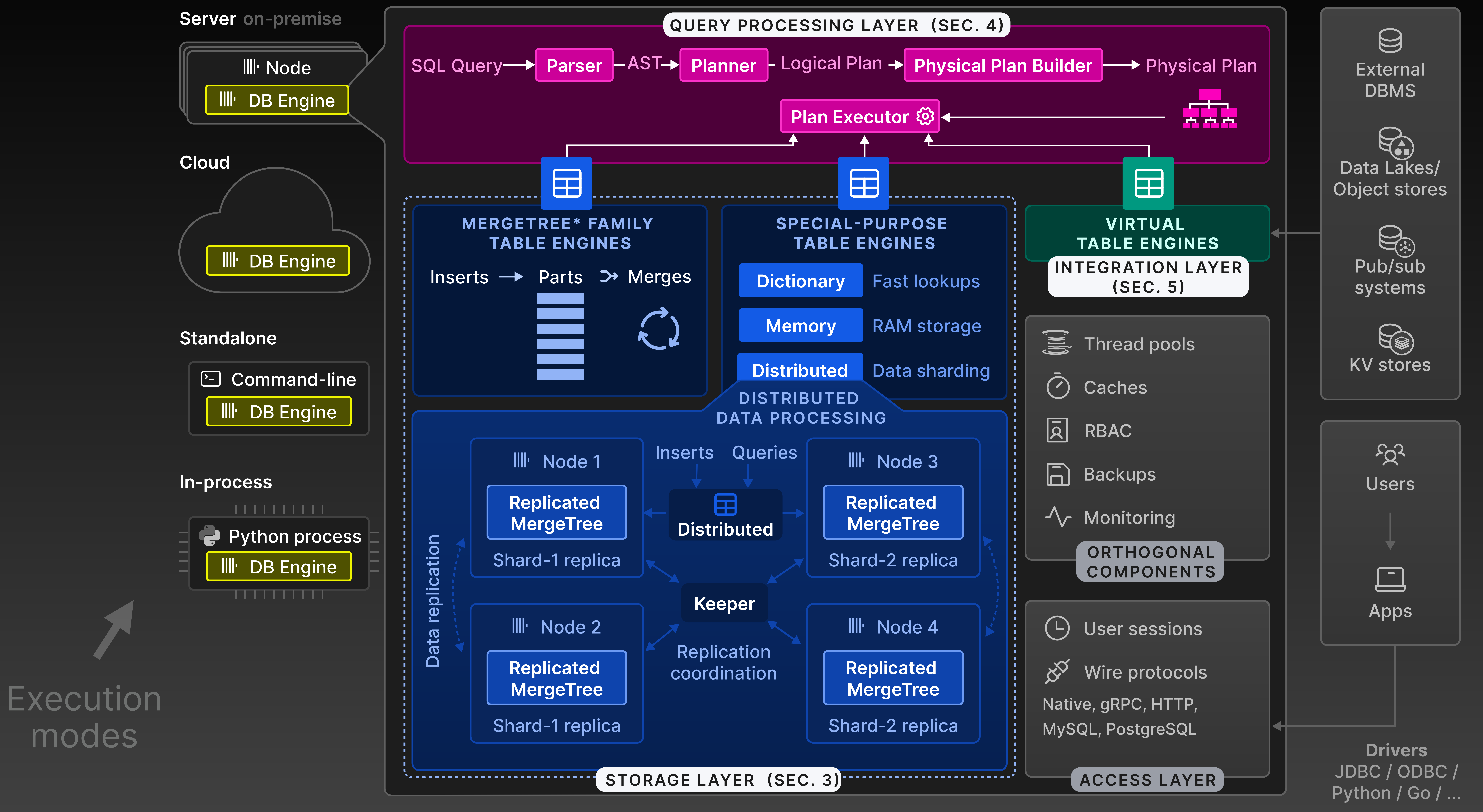
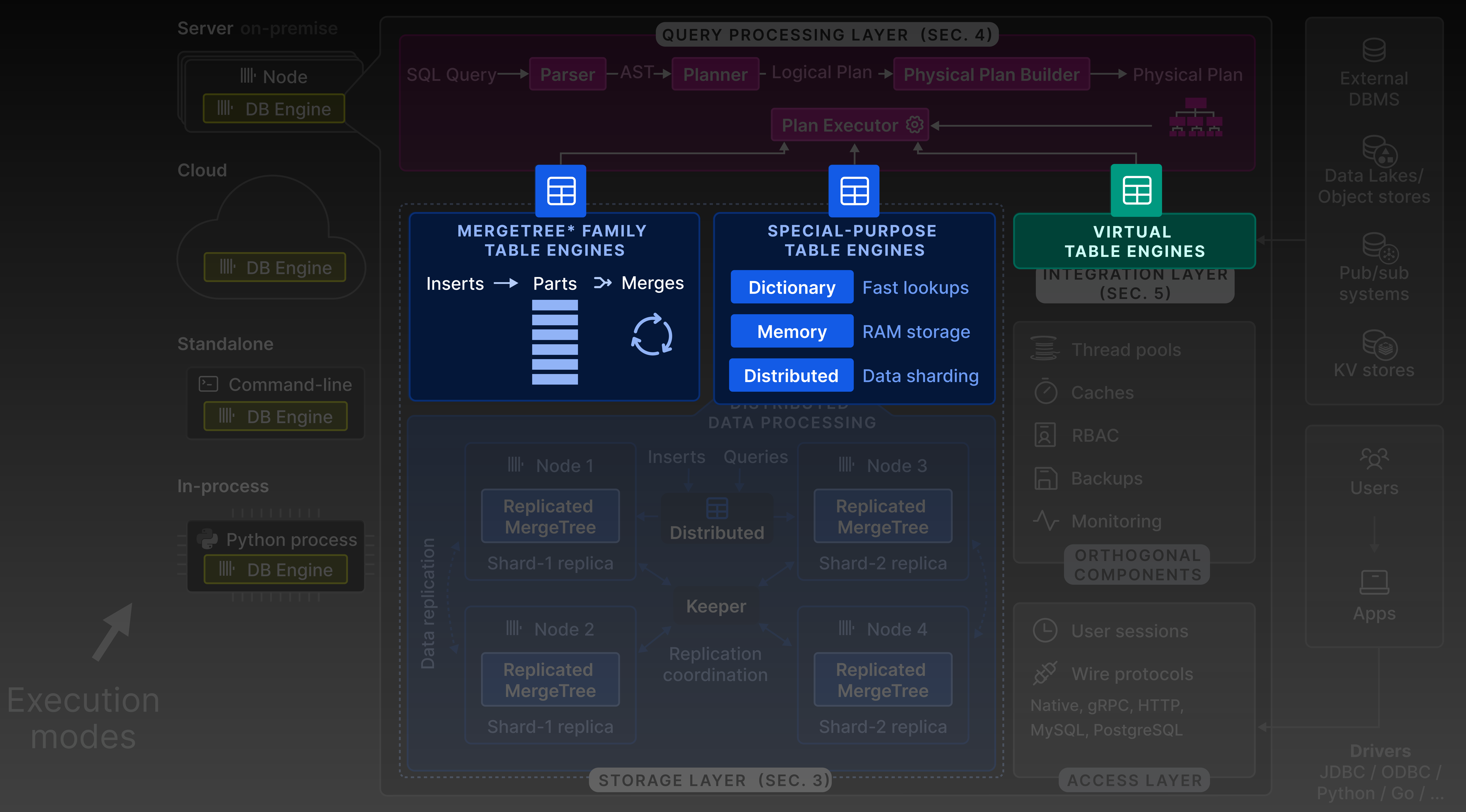
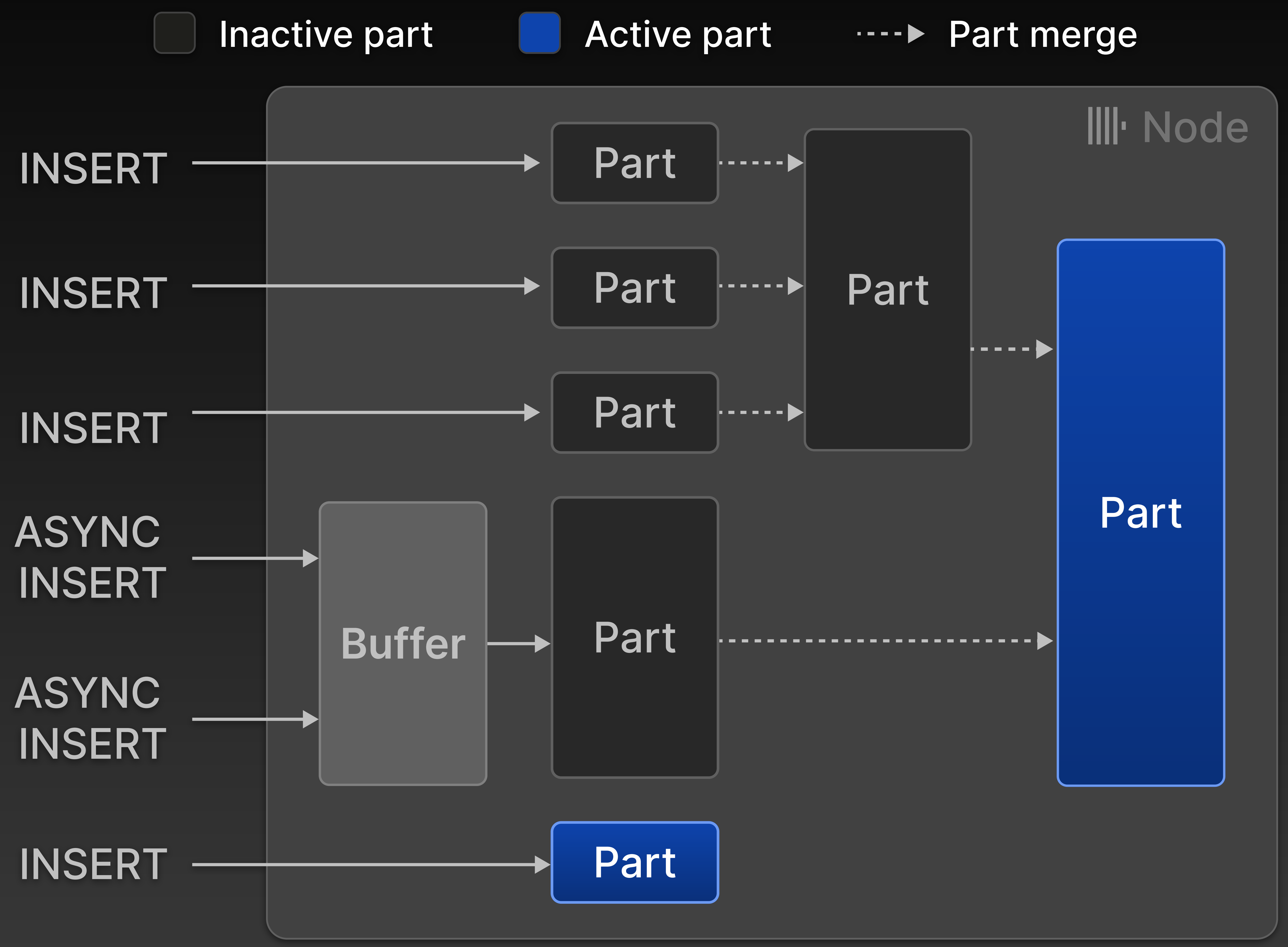


Table engines encapsulate the location and format of table data



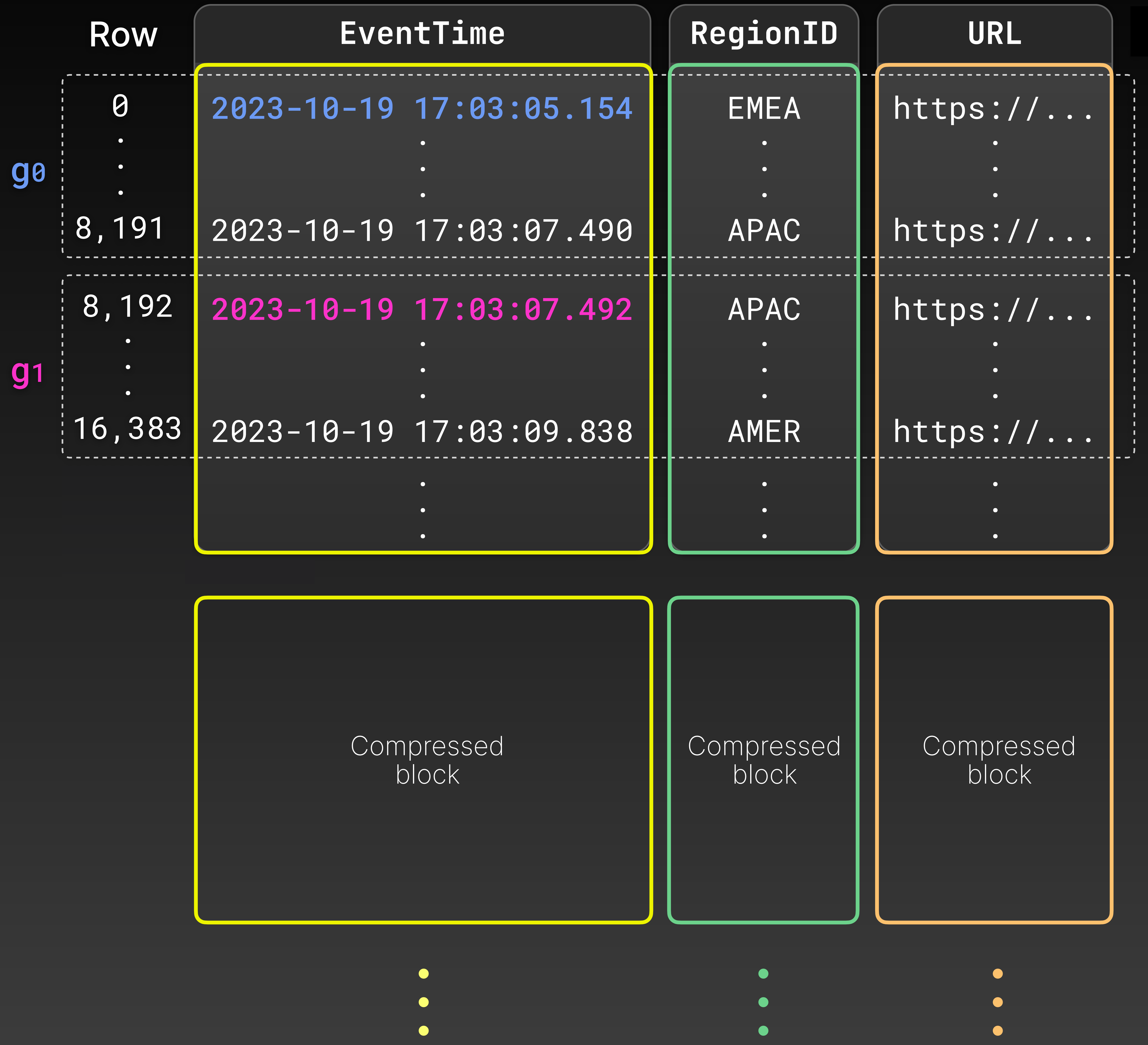
An LSM-Tree Inspired Storage Layer



- Like LSM trees, INSERTs create sorted and immutable *parts*.
- Parts are continuously merged by a background job.
- Unlike LSM trees, all parts are equal (i.e., no levels or notion of recency).
- INSERTs can be synchronous or asynchronous.

Ingestion rates are only limited by the speed of disk.

Column Layout and Compression



- Local (per-part) sorting defined by primary key:

```
CREATE TABLE page_hits
(
    EventTime  Date    CODEC(Delta, ZSTD),
    RegionId   String  CODEC(LZ4),
    URL        String  CODEC(AES),
    PRIMARY KEY (EventTime)
)
```

- Parts are further divided into *granules* g0, g1, ...
- Consecutive granules in a column form *blocks* which are encoded:
  - Generic codecs: LZ4, ZSTD, DEFLATE, ...
  - Logical codecs: Delta, GCD, ...
  - Specializec Codecs: Gorilla(FP), AES,...
- Codecs can be combined: CODEC(Delta, ZSTD)

High compression rates are in many use cases critical for cost efficiency and performance.

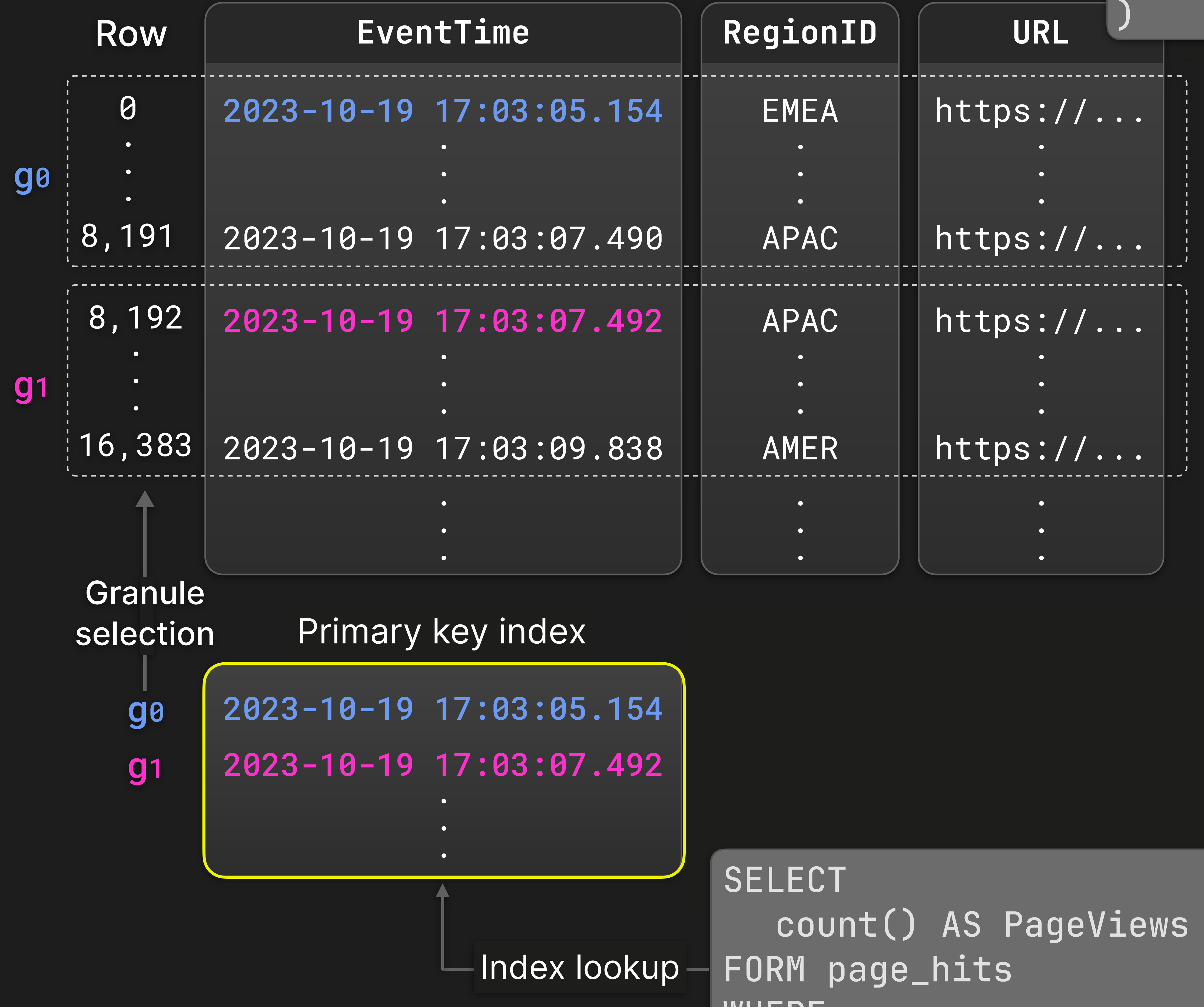
Data Pruning

Primary key indexes

Table projections

Skipping indexes

```
CREATE TABLE page_hits
(
   EventTime Date,
   RegionId  String,
   URL       String,
   PRIMARY KEY (EventTime)
)
```




```
SELECT
   count() AS PageViews
FORM page_hits
WHERE
   EventTime ≥ '2023-12-09'
```

- Define the local part sorting (clustered index).
- Also create a mapping from primary key column values to granules.
- The mapping is small enough to remain in DRAM at all times.

Quickly find granules containing rows that match a predicate on a prefix of the PK columns.

Data Pruning

Primary key indexes

Table projections

Skipping indexes

EventTime	RegionID	URL
2023-10-19 17:03:05.154	EMEA	https:// ...
2023-10-19 17:03:05.462	APAC	https:// ...
2023-10-19 17:03:05.875	AMER	https:// ...
2023-10-19 17:03:06.104	AMER	https:// ...
2023-10-19 17:03:07.550	APAC	https:// ...

```
ALTER TABLE page_hits ADD PROJECTION proj (
    SELECT *
    ORDER BY RegionID
);
ALTER TABLE page_hits MATERIALIZE PRJECTION prj;
```

EventTime	RegionID	URL
2023-10-19 17:03:05.875	AMER	https:// ...
2023-10-19 17:03:07.550	AMER	https:// ...
2023-10-19 17:03:06.104	APAC	https:// ...
2023-10-19 17:03:05.462	APAC	https:// ...
2023-10-19 17:03:05.154	EMEA	https:// ...

- Alternative table versions sorted by different primary keys.
- Works at the granularity of parts.
- Speed up queries on columns different than primary key columns.

```
SELECT
    count() AS PageViews
FORM page_hits
WHERE
    RegionID = 'AMER'
```

Powerful but increase space consumption and insert/merge overhead.

Data Pruning

Primary key indexes

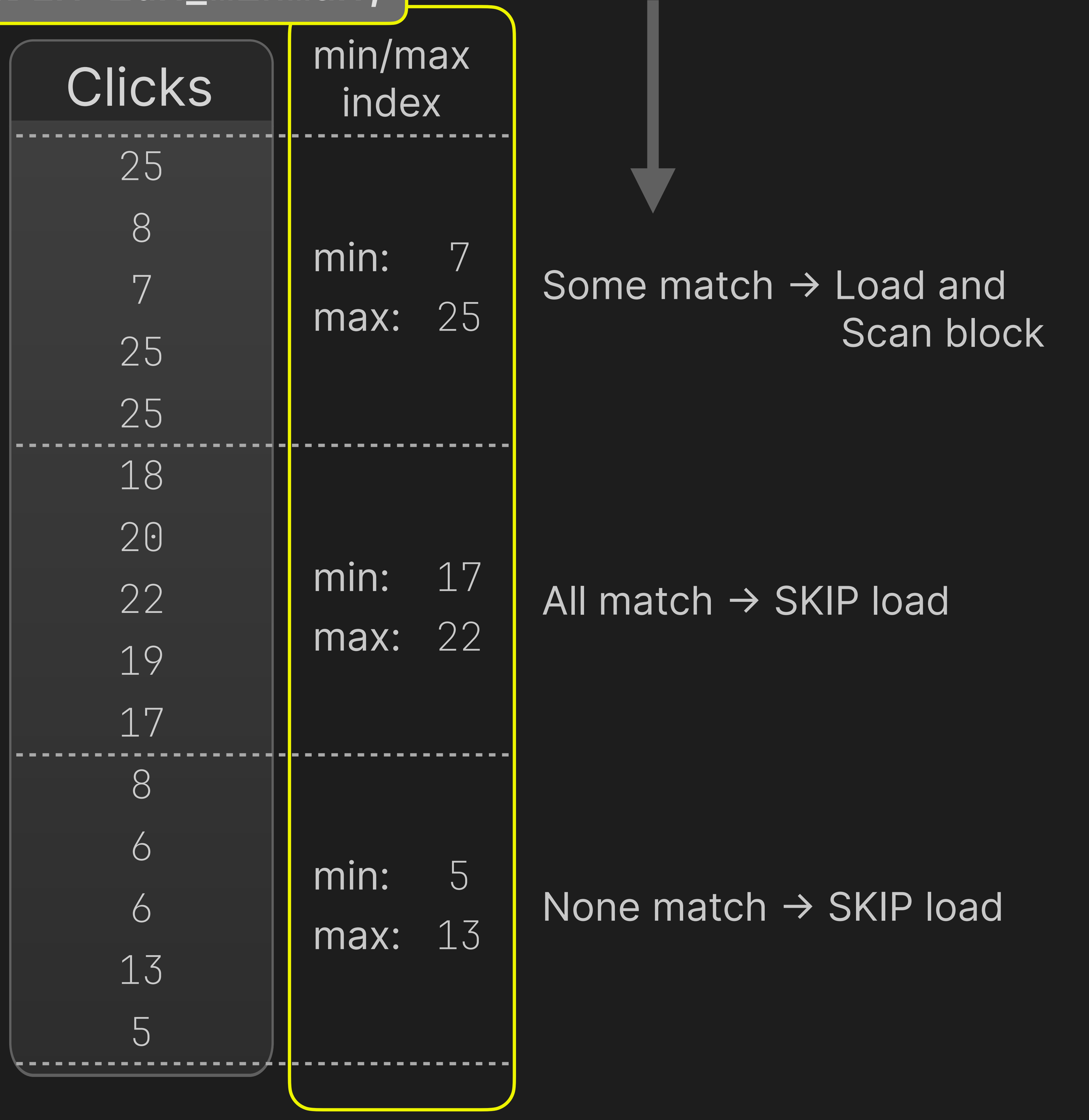
Table projections

Skipping indexes

```
ALTER TABLE T
ADD INDEX idx_minmax (Clicks) TYPE minmax;
ALTER TABLE T MATERIALIZE INDEX idx_minmax;
```

```
SELECT *
FROM T
WHERE
  Clicks BETWEEN 15 AND 30
```

- Store small amounts of metadata at the level of granules or multiple granules which allows to skip data during scans.
- Skipping index types:
  - Min/Max values
  - Unique values
  - Bloom filter
  - ...



Skipping indexes are a light-weight alternative to projections.

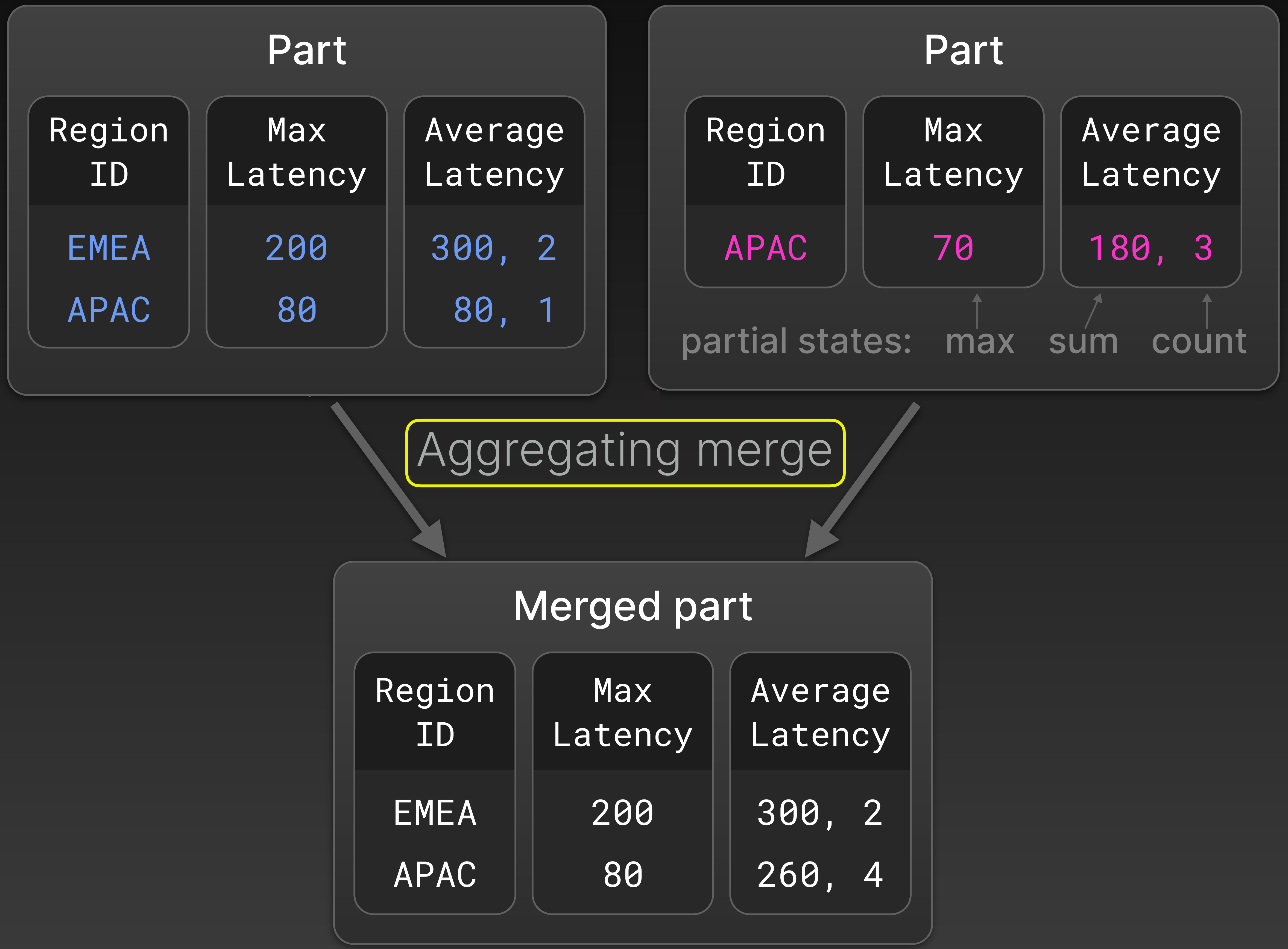
Merge-time Data Transformation

Merges optionally perform additional data transformations or maintenance.

Replacing merges
Retain the most recently inserted version of the same rows in multiple input parts.

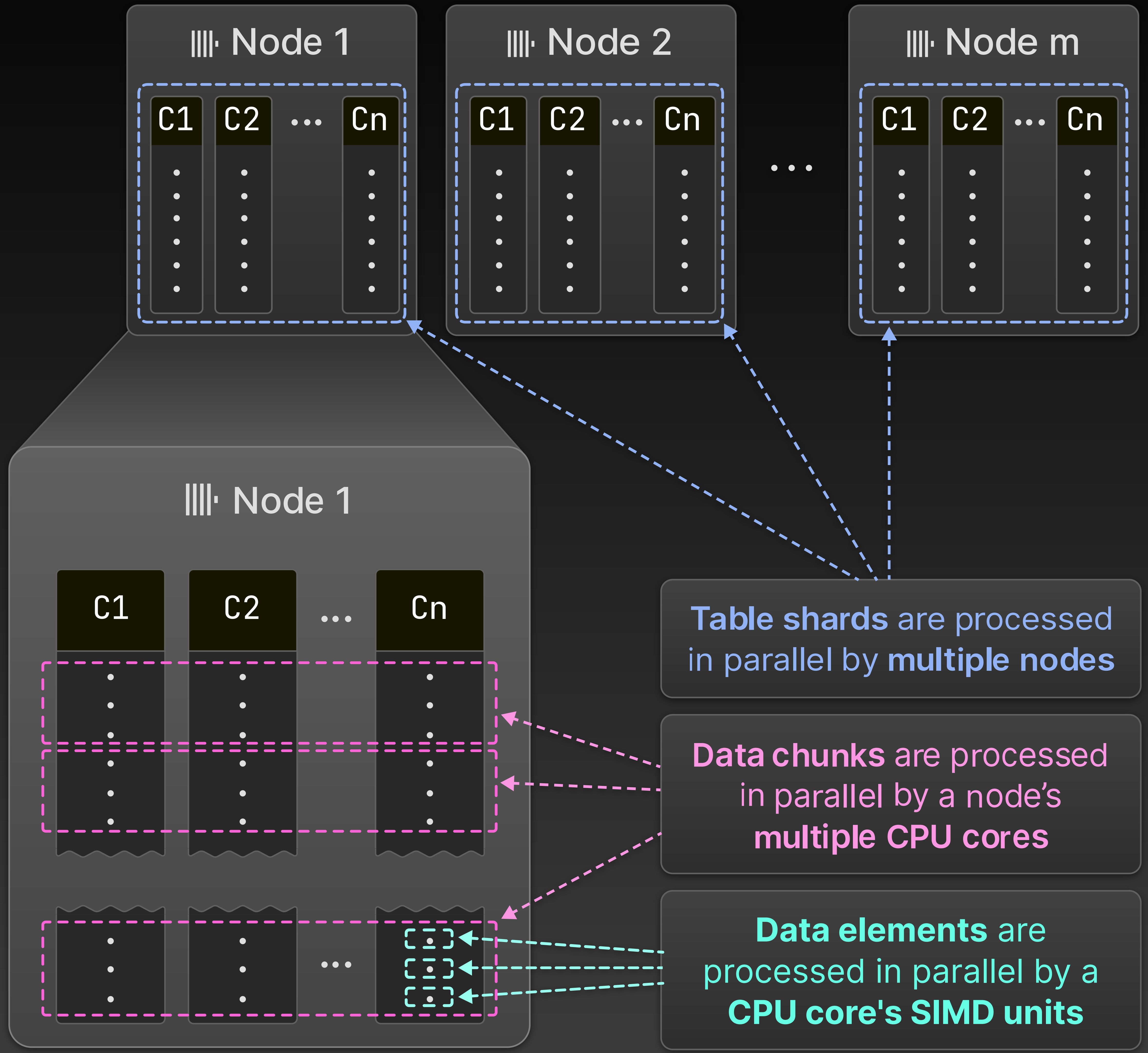
Aggregating merges
Combine aggregation states into new aggregation states.

TTL (time-to-live) merges
Compress, move, or, delete rows or parts.



Data transformations don't compromise the performance of parallel INSERTs and SELECTs.

State-of-the-art Vectorized Query Execution Engine



Query execution utilizes all server and cluster resources.

Parallelization Across SIMD ALUs

- Based on compiler auto-vectorization or manually written intrinsics.
- SQL expressions are compiled into *compute kernels*.
- The fastest kernel is selected at runtime based on the system capabilities (cpuid).

```
SELECT col1 + col2
FROM T
```

```
if (isArchSupported(TargetArch::AVX512))
      implAVX512BW(in1, in2);
else if (isArchSupported(TargetArch::AVX2))
      implAVX2(in1, in2, out);
else if (isArchSupported(TargetArch::SSE42))
      implSSE42(in1, in2, out);
else
      implGeneric(in1, in2, out);
```

Dispatch code based on cpuid

```
MULTITARGET_FUNCTION_AVX512F_AVX2_SSE42(
MULTITARGET_FUNCTION_HEADER(),
    impl,
MULTITARGET_FUNCTION_BODY((
    const double * in1, const double * in2
    double * out, size_t num_elements)
{
     for (size_t i = 0; i < (sz & ~0x7); i += 8)
     {
            const __m512d _in1 = _mm512_load_pd(&in1[i]);
            const __m512d _in2 = _mm512_load_pd(&in2[i]);
            const __m512d _out = _mm512_add_pd(_in1, _in2);
            out[i] = (double*)&_out;
     } /* tail handling */
}))
```

AVX-512 kernel, manually vectorized

```
MULTITARGET_FUNCTION_AVX2_SSE42(
MULTITARGET_FUNCTION_HEADER(),
     impl,
MULTITARGET_FUNCTION_BODY((
    const double * in1, const double * in2
    double * out, size_t num_elements)
{
    for (size_t i = 0; i < num_elements; ++i)
        *out[i] = *in1[i] + *in2[i];
}))
```

AVX2 kernel, compiler auto-vectorized

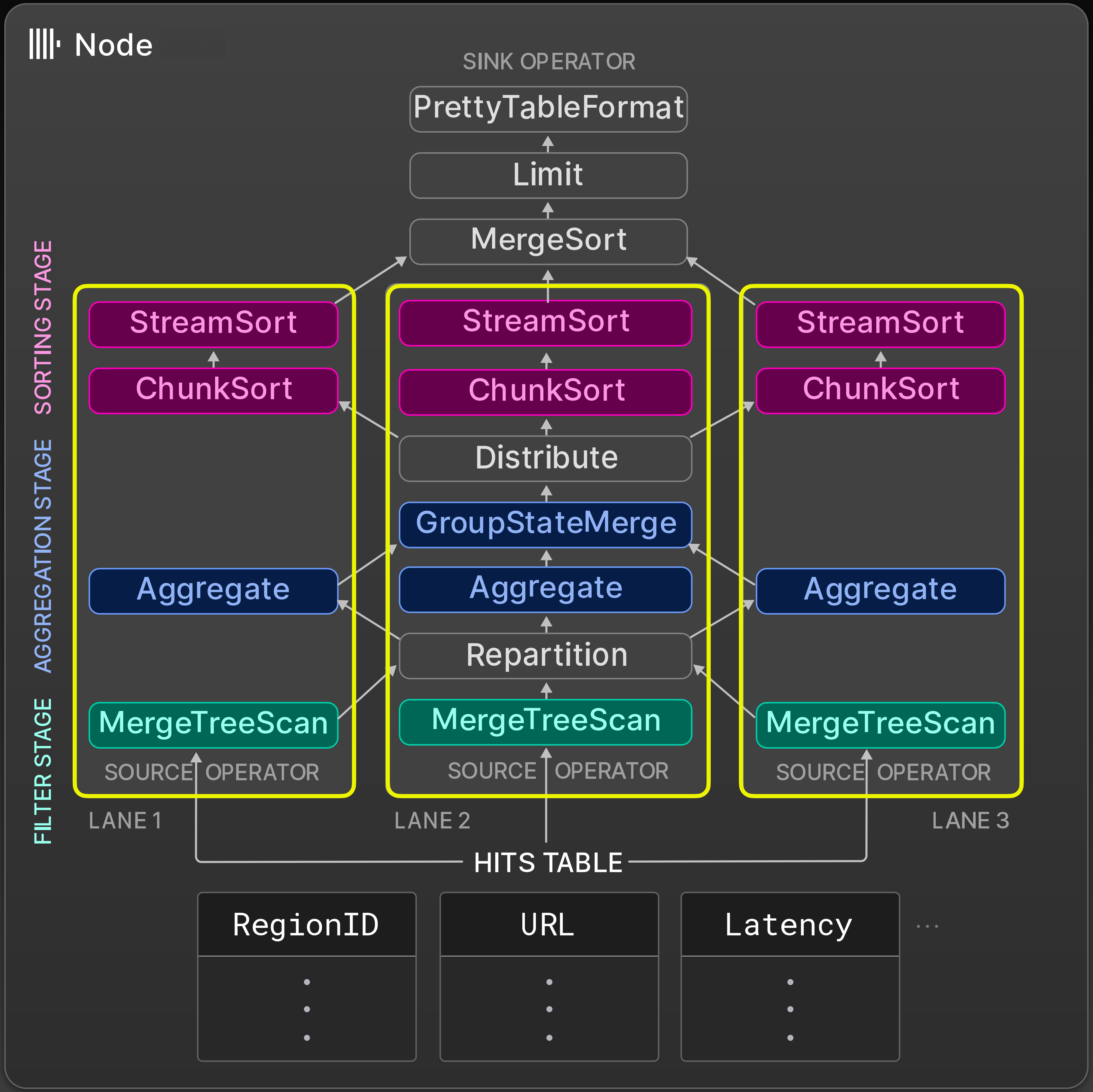
Remain compatible with legacy hardware while utilizing modern hardware fully.

Parallelization Across CPU Cores

filter / aggregation / sort

```
SELECT RegionID, avg(Latency) AS AvgLatency
FROM hits
WHERE URL = 'https://clickhouse.com'
GROUP BY RegionID
ORDER BY AvgLatency DESC
LIMIT 3
```

- Execution plan gets unfolded into N lanes (typically 1 lane per CPU core).
- Lanes decompose the data to be processed into non-overlapping ranges.
- Exchange operators (`Repartition`, `Distribute`) ensure lanes remain balanced.



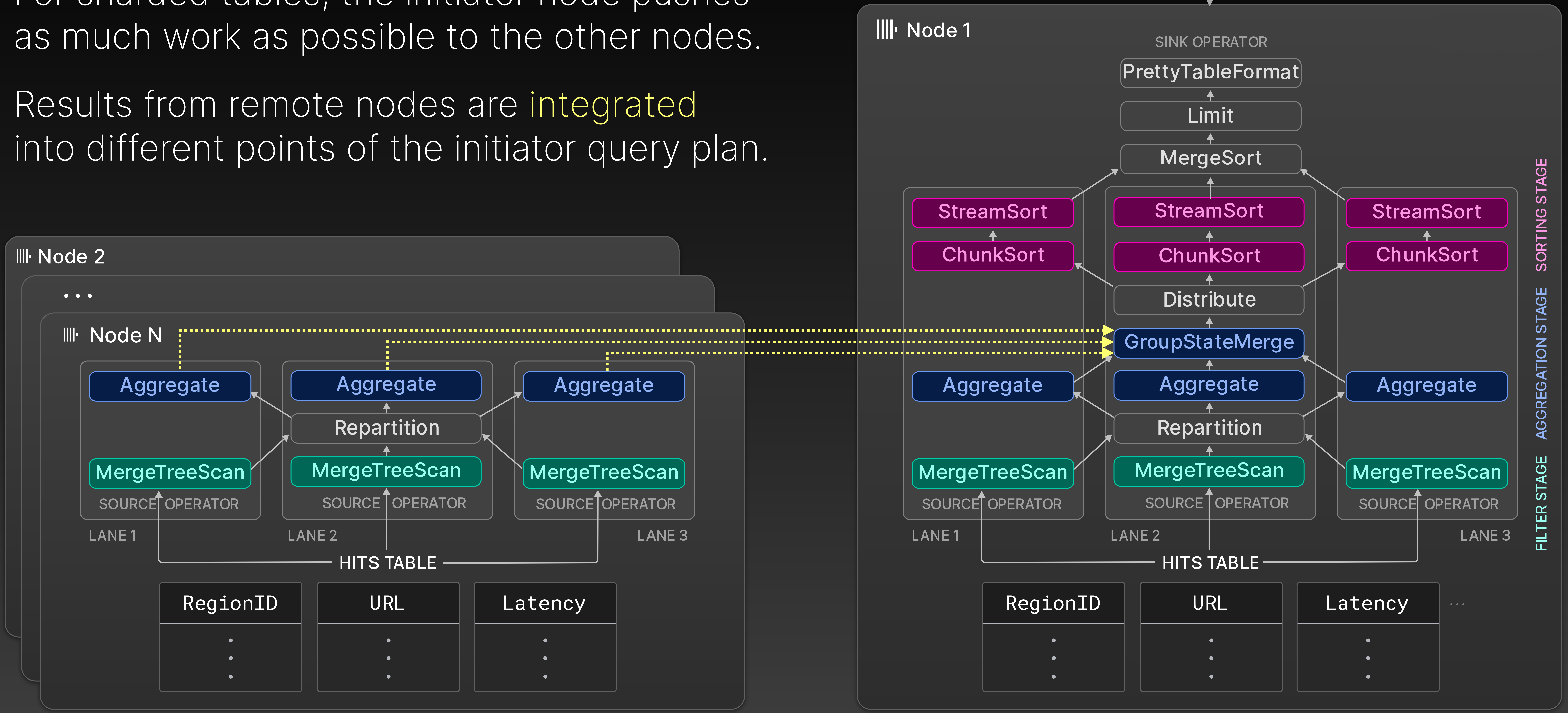
Enables vertical scaling by adding more CPU cores.

Parallelization Across Cluster Nodes

- For sharded tables, the initiator node pushes as much work as possible to the other nodes.
- Results from remote nodes are integrated into different points of the initiator query plan.

```
SELECT RegionID, avg(Latency) AS AvgLatency
FROM hits
WHERE URL = 'https://clickhouse.com'
GROUP BY RegionID
ORDER BY AvgLatency DESC
LIMIT 3
```

filter
aggregation
sort



Enables horizontal scaling by adding more cluster nodes.

What else is described in the paper?

Additional storage layer details

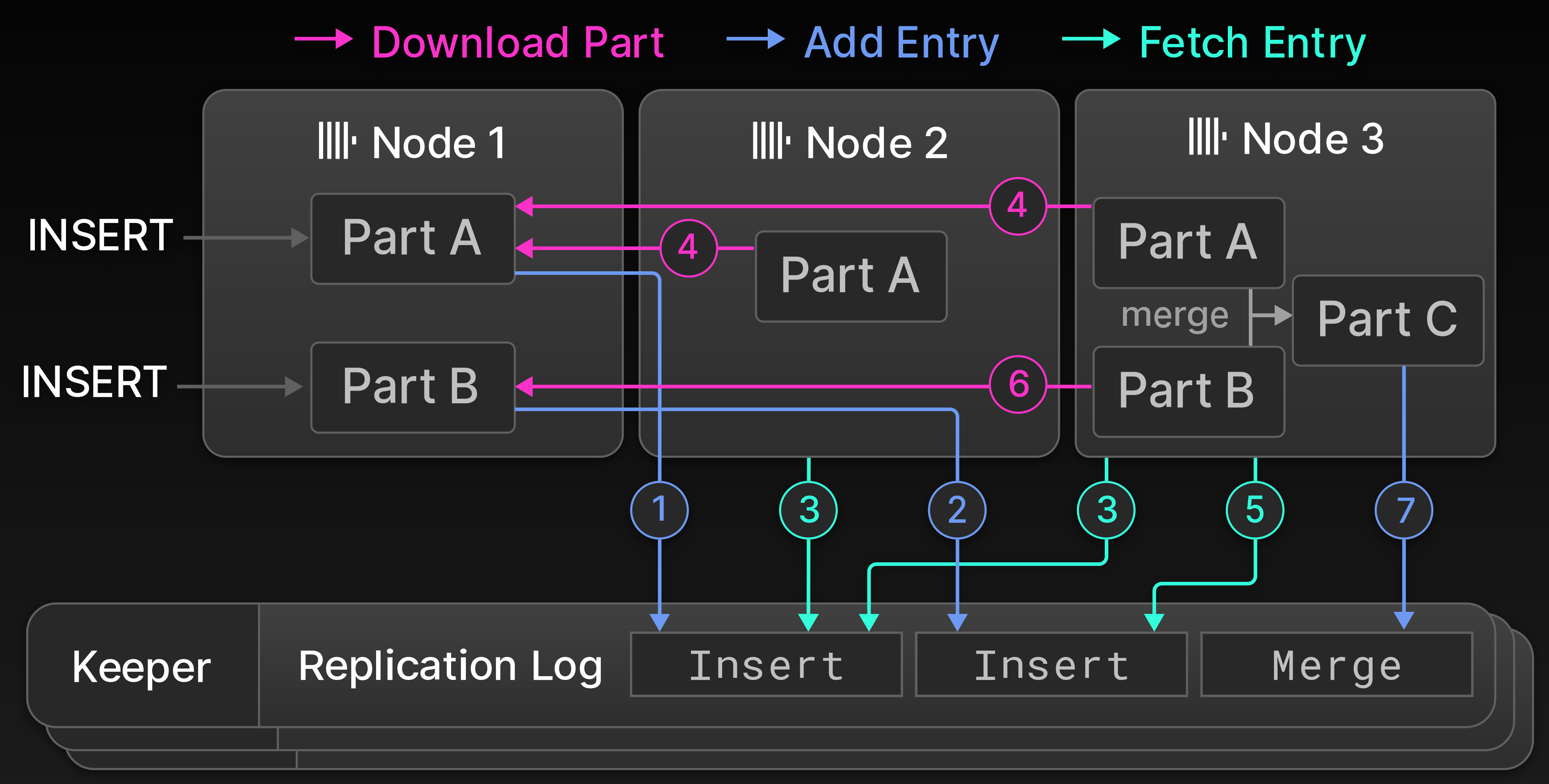
- Updates and deletes, idempotent inserts
- Data replication
- ACID compliance

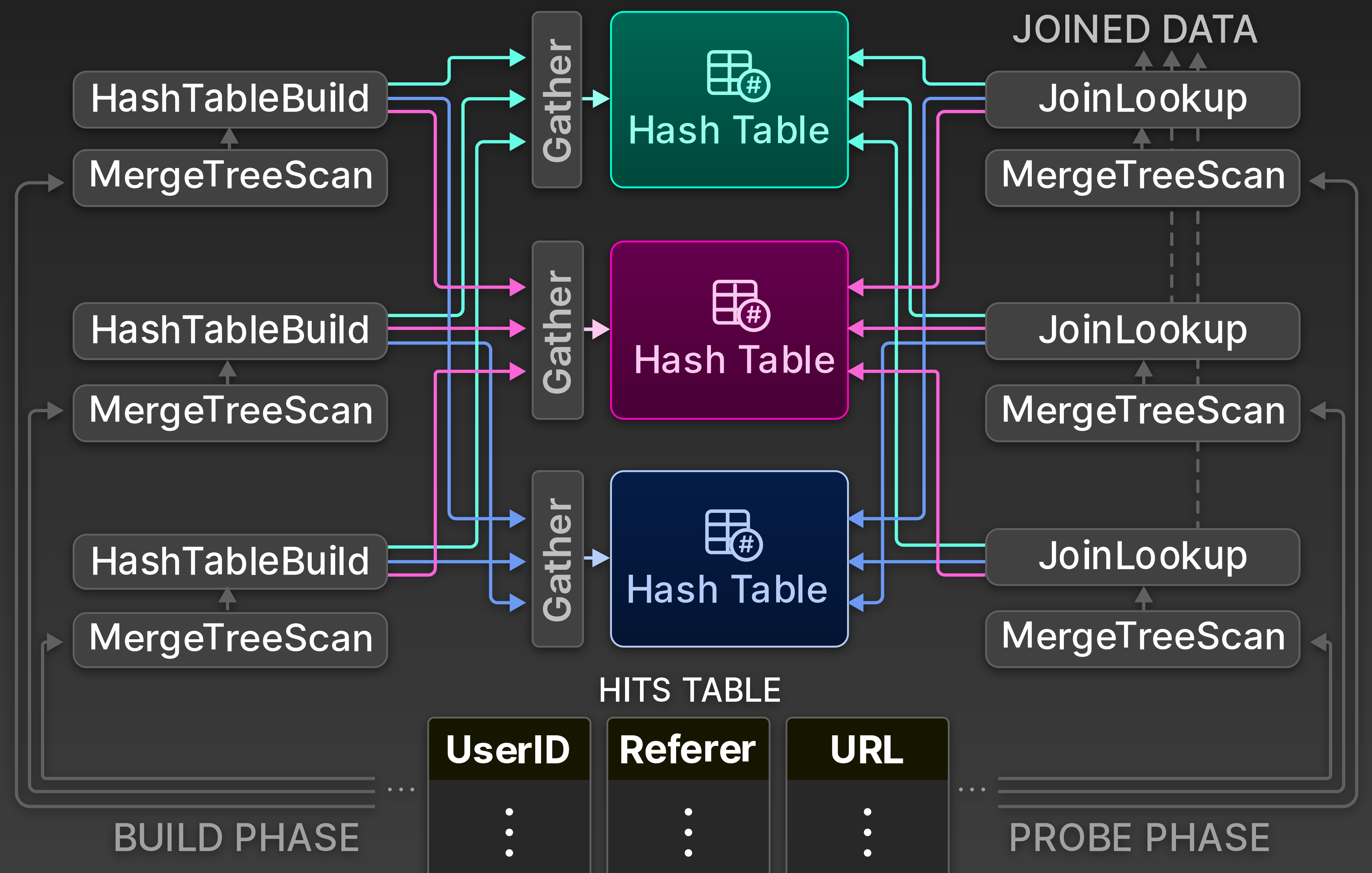
Low-level query optimizations

- JIT query compilation based on LLVM
- Hash table framework for aggregations and joins
- Parallel join execution

Integration layer

- Native support for 90+ file formats and 50+ integrations with external systems





Come and join us on GitHub in our mission to build the world's fastest analytics database 🚀

github.com/ClickHouse/ClickHouse

Backup slides

Benchmarks on Normalized Tables - **TPC-H**

	Q1	Q2	Q3	Q4	Q5	Q6	Q7-Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20- Q22
	1.86		4.13		7.01	0.39		3.59	0.83	1.53		1.00	1.04	0.48		2.18		
	2.20		2.10		1.90	0.23		4.30	1.30	0.88		0.65	0.77	1.90		3.40		

The results of eleven queries are excluded:

- Queries containing **correlated subqueries** (not supported as of ClickHouse v24.6)
- Queries requiring **extended plan-level optimizations for joins** (missing as of ClickHouse v24.6)

Hot runtimes of the TPC-H queries based on the parallel hash join algorithm described in Section 4.4. The fastest of five runs was recorded.
Measurements taken on a single-node AWS EC2 c6i.16xlarge instance with 64 vCPUs, 128 GB RAM, and 5000 IOPS / 1000 MiB/s disk.
Comparable size was used for Snowflake (warehouse size L, 8×8 vCPUs, 8×16 GB RAM)

- Queries over normalized tables are an emerging use case for ClickHouse
- Automatic subquery decorrelation and better plan optimizer support for joins are planned for 2024.
  https://github.com/ClickHouse/ClickHouse/issues/58392

One more thing...
There's a lot more to uncover

- Powerful SQL dialect with higher-order functions and lambda functions.
- 150+ built-in aggregate functions plus aggregate function combinators.
- 1300+ regular functions (mathematics, geo, machine learning, time series, etc.)
- Parallelized window functions and joins.
- JSON, maps and arrays plus 80+ array functions.