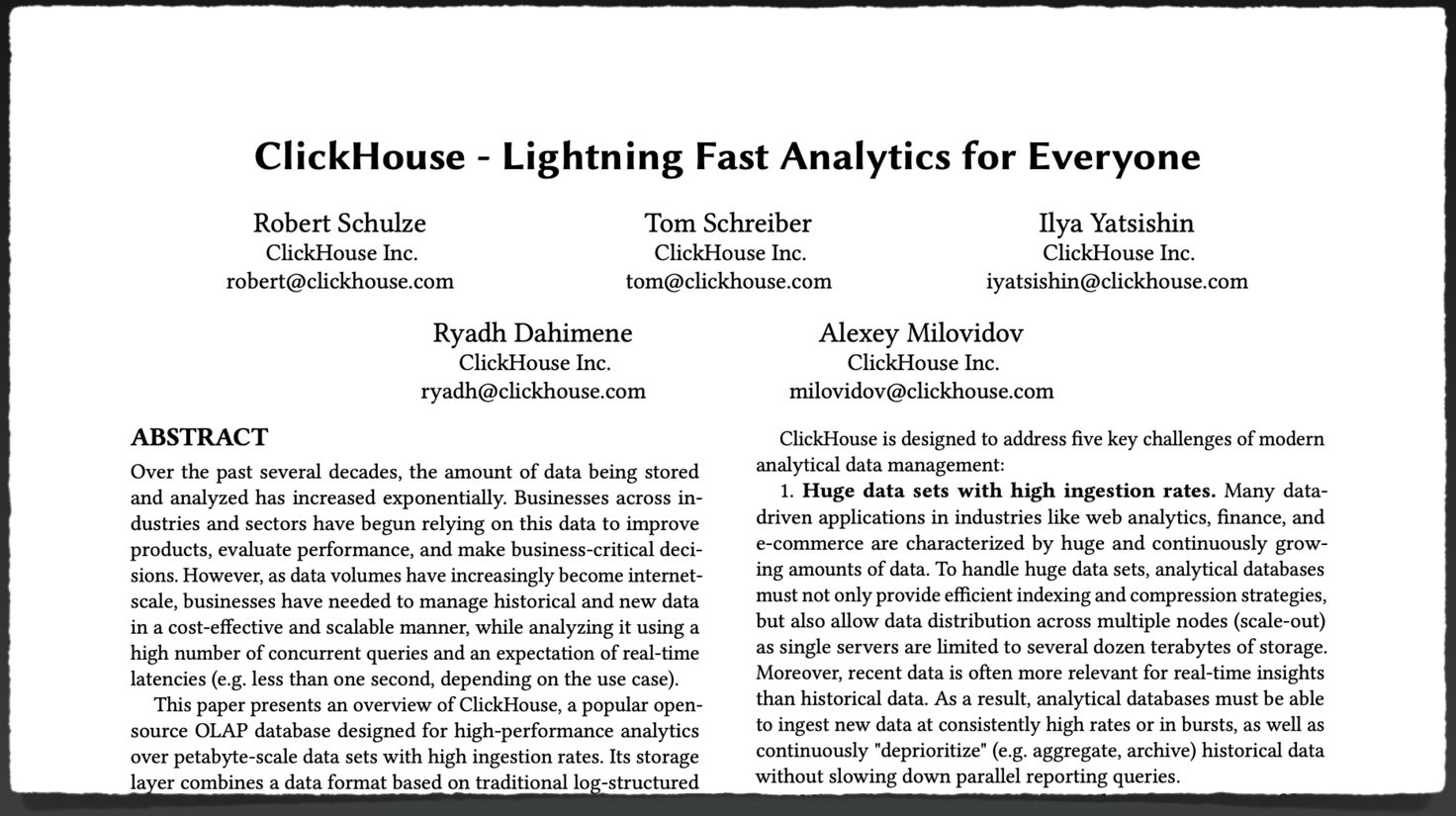
ClickHouse - the What, the Why, the How

Robert Schulze
ClickHouse Inc.
robert@clickhouse.com

Tom Schreiber
ClickHouse Inc.
tom@clickhouse.com



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.
milovidov@clickhouse.com

ABSTRACT

Over the past several decades, the amount of data being stored and analyzed has increased exponentially. Businesses across industries and sectors have begun relying on this data to improve products, evaluate performance, and make business-critical decisions. However, as data volumes have increasingly become internet-scale, businesses have needed to manage historical and new data in a cost-effective and scalable manner, while analyzing it using a high number of concurrent queries and an expectation of real-time latencies (e.g. less than one second, depending on the use case).

This paper presents an overview of ClickHouse, a popular open-source OLAP database designed for high-performance analytics over petabyte-scale data sets with high ingestion rates. Its storage layer combines a data format based on traditional log-structured

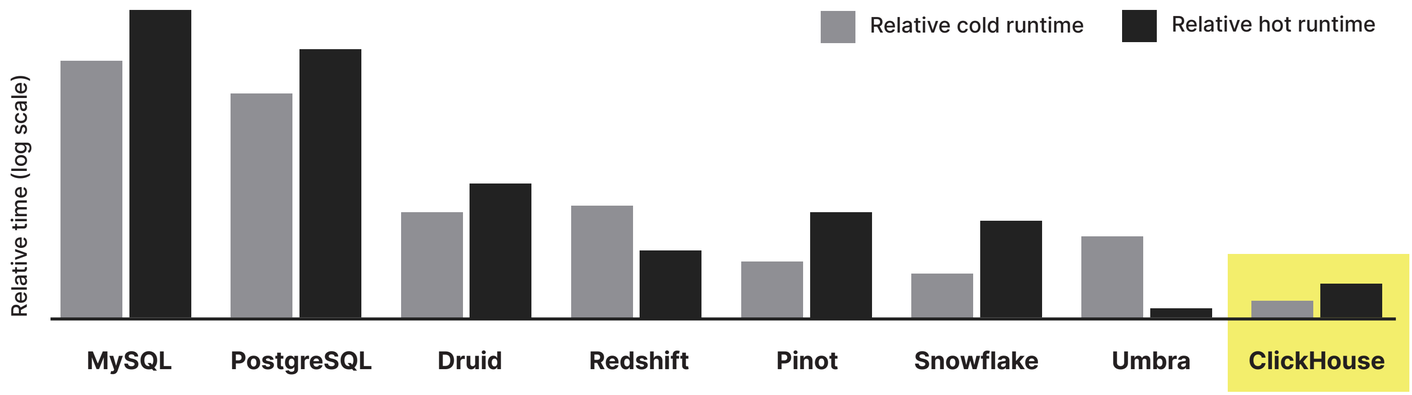
ClickHouse is designed to address five key challenges of modern analytical data management:

1. **Huge data sets with high ingestion rates.** Many data-driven applications in industries like web analytics, finance, and e-commerce are characterized by huge and continuously growing amounts of data. To handle huge data sets, analytical databases must not only provide efficient indexing and compression strategies, but also allow data distribution across multiple nodes (scale-out) as single servers are limited to several dozen terabytes of storage. Moreover, recent data is often more relevant for real-time insights than historical data. As a result, analytical databases must be able to ingest new data at consistently high rates or in bursts, as well as continuously "deprioritize" (e.g. aggregate, archive) historical data without slowing down parallel reporting queries.


DB Engine

DB Engine

Fastest analytics database



ClickHouse has the best query performance amongst production-grade analytics databases. Performance is a top priority and continuously improved.

benchmark.clickhouse.com

45+ commercial and research databases

Fast

the best
nce
n-grade
s.
p priority
nproved.
use.com

ClickBench — a Benchmark For Analytical DBMS

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

System: All | AlloyDB | Athena (partitioned) | Athena (single) | Aurora for MySQL | Aurora for PostgreSQL | ByConity | ByteHouse | 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 (Parquet, partitioned) | DuckDB | Elasticsearch | Elasticsearch (tuned) | GlareDB | Greenplum | HeavyAI | Hydra | Infobright | Kinetica | MariaDB ColumnStore | MariaDB | MonetDB | MongoDB | Motherduck | MySQL (MyISAM) | MySQL | Oxla | ParadeDB (Parquet, partitioned) | ParadeDB (Parquet, single) | Pinot | PostgreSQL (tuned) | PostgreSQL | QuestDB (partitioned) | QuestDB | Redshift | SelectDB | SingleStore | Snowflake | SQLite | StarRocks | Tablespace | Tembo OLAP (columnar) | TimescaleDB (compression) | TimescaleDB | Umbra

Type: parallel replicas | Azure | analytical | Rust | search | document | somewhat PostgreSQL compatible | time-series

Machine: All | 16 vCPU 128GB | 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 | 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 | XL | L1 - 16CPU 32GB | c6a.4xlarge, 500gb gp3

Cluster size: dedicated

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

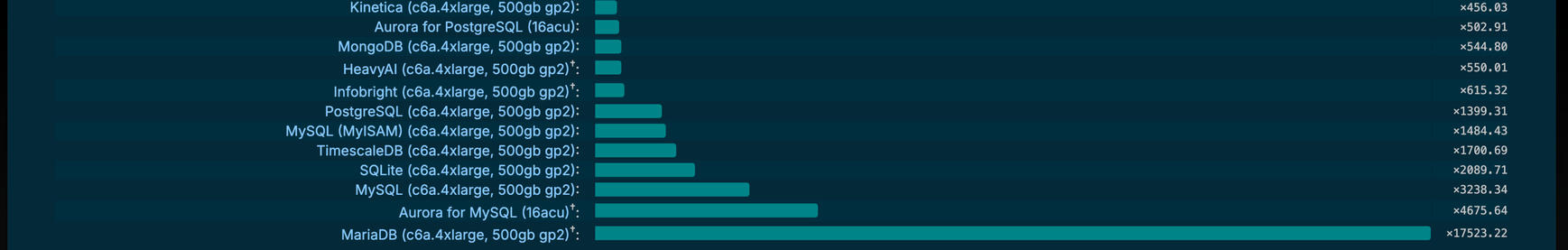
42 queries analyzing 100 million rows of event data

System & Machine	Relative time (lower is better)
ClickHouse (tuned, memory) (c6a.metal, 500gb gp2):	×1.44
ClickHouse Cloud (aws) (1430GB):	×3.75
ClickHouse Cloud (aws) (720GB):	×3.95
ClickHouse Cloud (gcp) (708GB):	×4.03
StarRocks (c6a.metal, 500gb gp2):	×4.36
Snowflake (64×3XL):	×5.13
Snowflake (32×2XL):	×5.19
ClickHouse Cloud (aws) (360GB):	×5.22
ClickHouse Cloud (Azure) (192GB):	×5.89
Snowflake (16×XL):	×6.05
Snowflake (128×4XL):	×6.08
ByteHouse (8×L):	×6.22
ClickHouse (web) (c6a.metal, 500gb gp2):	×6.37
ClickHouse Cloud (Azure) (360GB):	×6.51
SelectDB (c5.4xlarge, 500gb gp2):	×6.52
SelectDB (c6a.4xlarge, 500gb gp2):	×6.58
ClickHouse Cloud (Azure) Parallel Replicas ON (360GB):	×6.60
ClickHouse Cloud (aws) (192GB):	×6.69
ClickHouse Cloud (gcp) (360GB):	×6.80
ClickHouse (tuned) (c6a.metal, 500gb gp2):	×6.91
Apache Doris (c6a.4xlarge, 500gb gp2):	×7.08
Crunchy Bridge for Analytics (Parquet) (Analytics-256GB (64 vCores, 256 GB)):	×7.24
ClickHouse Cloud (Azure) Parallel Replica ON (192GB):	×7.32
ClickHouse Cloud (gcp) (192GB):	×7.54
ClickHouse (c6a.4xlarge, 500gb gp2):	×7.86
SingleStore (S24)†:	×7.96
Snowflake (8×L):	×8.13
ClickHouse (c6a.metal, 500gb gp2):	×8.43
ClickHouse Cloud (Azure) Parallel Replica ON (96GB):	×8.52
ByteHouse (4×M):	×8.89
ClickHouse Cloud (gcp) (96GB):	×9.15

Fast

ClickHouse (Parquet, partitioned) (c6a.metal, 500gb gp2): ×16.35
DataFusion (Parquet, single) (c6a.4xlarge, 500gb gp2)†: ×16.38
Databend (c6a.metal, 500gb gp2): ×16.51
ClickHouse (Parquet, single) (c6a.4xlarge, 500gb gp2): ×16.57
chDB (Parquet, partitioned) (c6a.4xlarge, 500gb gp2): ×17.11
SingleStore (S2)†: ×17.73
ClickHouse Cloud (Azure) Parallel Replica ON (24GB): ×17.83
ParadeDB (Parquet, single) (c6a.4xlarge, 500gb gp2): ×18.23
chDB (c6a.metal, 500gb gp2): ×18.31
chDB (Parquet, partitioned) (c6a.metal, 500gb gp2): ×18.48
ClickHouse (data lake, single) (c6a.4xlarge, 500gb gp2): ×19.36
chDB (c6a.4xlarge, 500gb gp2): ×19.71
ParadeDB (Parquet, partitioned) (c6a.4xlarge, 500gb gp2): ×19.86
Snowflake (XS): ×19.92
ClickHouse Cloud (gcp) (24GB): ×20.31
DuckDB (c5.4xlarge, 500gb gp2): ×20.83
ClickHouse Cloud (Azure) (24GB): ×21.63
DuckDB (c6a.4xlarge, 500gb gp2): ×22.84
ClickHouse (data lake, partitioned) (c6a.4xlarge, 500gb gp2): ×23.17
ClickHouse Cloud (aws) (24GB): ×23.27
ByteHouse (XS): ×23.31
MonetDB (c6a.4xlarge, 500gb gp2): ×25.30
Elasticsearch (tuned) (c6a.4xlarge, 1500gb gp2)†: ×26.11
SelectDB (c6a.metal, 500gb gp2): ×27.88
Cloudberry (c6a.4xlarge, 500gb gp2)†: ×35.56
Umbra (c6a.4xlarge, 500gb gp2): ×36.19
Greenplum (c6a.4xlarge, 500gb gp2): ×36.71
Athena (single) (serverless): ×37.02
Redshift (4×ra3.16xlarge): ×37.39
DuckDB (c6a.metal, 500gb gp2): ×37.66
GlareDB (c6a.4xlarge, 500gb gp2): ×38.77
Tembo OLAP (columnar) (c6a.4xlarge, 500gb gp3): ×38.82
Hydra (c6a.4xlarge, 500gb gp2): ×41.79
Athena (partitioned) (serverless): ×43.15
Pinot (c6a.4xlarge, 500gb gp2)†: ×43.48
SingleStore (c6a.4xlarge, 500gb gp2)†: ×45.33
AlloyDB (16 vCPU 128GB): ×51.16
Redshift (serverless): ×54.99
AlloyDB (8 vCPU 64GB): ×55.30
Redshift (2×dc2.8xlarge): ×58.40
Umbra (c6a.metal, 500gb gp2): ×58.64
MariaDB ColumnStore (c6a.4xlarge, 500gb gp2)†: ×68.98
Redshift (4×ra3.xlplus): ×69.31
Redshift (4×ra3.4xlarge): ×72.16
Elasticsearch (c6a.4xlarge, 1500gb gp2): ×75.26
PostgreSQL (tuned) (c6a.4xlarge, 500gb gp2): ×85.66
CrateDB (c6a.4xlarge, 500gb gp2)†: ×89.69
GlareDB (c6a.metal, 500gb gp2): ×90.87
AlloyDB (8 vCPU 64GB): ×109.93
Druid (c6a.4xlarge, 500gb gp2)†: ×187.11
Citus (c6a.4xlarge, 500gb gp2): ×279.66
TimescaleDB (compression) (c6a.4xlarge, 500gb gp2): ×434.95
Kinetica (c6a.4xlarge, 500gb gp2): ×456.03
Aurora for PostgreSQL (16acu): ×502.91
MongoDB (c6a.4xlarge, 500gb gp2): ×544.80
HeavyAI (c6a.4xlarge, 500gb gp2)†: ×550.01
Infobright (c6a.4xlarge, 500gb gp2)†: ×615.32
PostgreSQL (c6a.4xlarge, 500gb gp2): ×1399.31
MySQL (MyISAM) (c6a.4xlarge, 500gb gp2): ×1484.43
TimescaleDB (c6a.4xlarge, 500gb gp2): ×1700.69
SQLite (c6a.4xlarge, 500gb gp2): ×2089.71
MySQL (c6a.4xlarge, 500gb gp2): ×3238.34
Aurora for MySQL (16acu)†: ×4675.64
MariaDB (c6a.4xlarge, 500gb gp2)†: ×17523.22

the best
nce
n-grade
s.
p priority
nproved.
use.com



Detailed Comparison

	ClickHouse (tuned, memory) (c6a.metal, 500gb gp2)	ClickHouse Cloud (aws) (1430GB)	ClickHouse Cloud (aws) (720GB)	ClickHouse Cloud (gcp) (708GB)	StarRocks (c6a.metal, 500gb gp2)	Snowflake (64×3XL)	Snowflake (32×2XL)	ClickHouse Cloud (aws) (360GB)	ClickHouse Cloud (192GB)
Load time:	290s (×877.65)	225s (×682.20)	220s (×667.39)	333s (×1007.92)	433s (×1312.12)	2524s (×7648.48)	2524s (×7648.48)	217s (×657.36)	295s (×
Data size:	128.16 GiB (×13.84)	**9.26 GiB (×1.00)**	9.26 GiB (×1.00)	9.27 GiB (×1.00)	16.49 GiB (×1.78)	11.46 GiB (×1.24)	11.46 GiB (×1.24)	9.26 GiB (×1.00)	9.26 GiB
Q0.	0.019s (×2.84)	0.014s (×2.35)	0.004s (×1.37)	0.008s (×1.76)	0.040s (×4.89)	0.165s (×17.13)	0.177s (×18.30)	0.003s (×1.27)	0.006s
Q1.	**0.014s (×1.00)**	0.090s (×4.17)	0.118s (×5.33)	0.978s (×41.17)	0.120s (×5.42)	1.356s (×56.92)	0.903s (×38.04)	0.651s (×27.54)	0.547s
Q2.	**0.019s (×1.00)**	0.520s (×18.28)	0.248s (×8.90)	0.171s (×6.24)	0.660s (×23.10)	1.287s (×44.72)	0.458s (×16.14)	0.307s (×10.93)	0.287s
Q3.	**0.029s (×1.00)**	0.332s (×8.77)	0.175s (×4.74)	0.851s (×22.08)	2.080s (×53.59)	0.627s (×16.33)	0.881s (×22.85)	0.638s (×16.62)	1.496s
Q4.	0.142s (×1.38)	0.529s (×4.90)	0.222s (×2.11)	0.413s (×3.85)	**0.100s (×1.00)**	0.265s (×2.50)	0.404s (×3.76)	0.273s (×2.57)	1.815s
Q5.	**0.162s (×1.00)**	0.932s (×5.48)	0.405s (×2.41)	0.996s (×5.85)	2.190s (×12.79)	0.887s (×5.22)	0.481s (×2.85)	0.585s (×3.46)	1.135s
Q6.	0.017s (×1.80)	0.356s (×24.40)	0.164s (×11.60)	0.119s (×8.60)	0.040s (×3.33)	0.054s (×4.27)	0.056s (×4.40)	0.117s (×8.47)	0.032s
Q7.	0.030s (×1.48)	0.074s (×3.11)	0.058s (×2.52)	0.042s (×1.93)	0.060s (×2.59)	0.182s (×7.11)	0.183s (×7.15)	0.059s (×2.56)	0.024s
Q8.	**0.187s (×1.00)**	0.502s (×2.60)	0.522s (×2.70)	0.507s (×2.62)	0.990s (×5.08)	0.408s (×2.12)	0.444s (×2.30)	0.610s (×3.15)	0.703s
Q9.	**0.282s (×1.00)**	0.401s (×1.41)	0.552s (×1.92)	0.448s (×1.57)	0.730s (×2.53)	0.434s (×1.52)	0.408s (×1.43)	1.158s (×4.00)	0.703s
Q10.	**0.055s (×1.00)**	0.261s (×4.17)	0.245s (×3.92)	0.290s (×4.62)	0.130s (×2.15)	1.357s (×21.03)	0.345s (×5.46)	0.236s (×3.78)	0.305s
Q11.	**0.050s (×1.00)**	0.206s (×3.60)	0.196s (×3.43)	0.269s (×4.65)	1.040s (×17.50)	0.343s (×5.88)	0.406s (×6.93)	0.289s (×4.98)	0.296s
Q12.	**0.142s (×1.00)**	0.229s (×1.57)	0.514s (×3.45)	0.361s (×2.44)	0.170s (×1.18)	0.273s (×1.86)	0.521s (×3.49)	0.444s (×2.99)	0.576s
Q13.	**0.175s (×1.00)**	0.377s (×2.09)	0.662s (×3.63)	0.407s (×2.25)	0.230s (×1.30)	0.404s (×2.24)	0.466s (×2.57)	0.582s (×3.20)	0.639s
Q14.	**0.155s (×1.00)**	0.275s (×1.73)	0.398s (×2.47)	0.452s (×2.80)	0.540s (×3.33)	0.379s (×2.36)	0.447s (×2.77)	0.739s (×4.54)	0.549s
Q15.	0.131s (×1.41)	0.173s (×1.83)	0.236s (×2.46)	0.208s (×2.18)	**0.090s (×1.00)**	0.275s (×2.85)	0.327s (×3.37)	0.313s (×3.23)	0.334s
Q16.	0.317s (×1.02)	0.484s (×1.54)	0.665s (×2.11)	0.660s (×2.09)	**0.310s (×1.00)**	0.418s (×1.34)	0.462s (×1.48)	1.126s (×3.55)	1.121s
Q17.	0.139s (×9.65)	0.326s (×21.77)	0.474s (×31.35)	0.458s (×30.32)	0.120s (×8.42)	0.417s (×27.66)	0.489s (×32.32)	0.838s (×54.93)	0.693s
Q18.	0.660s (×1.17)	0.975s (×1.71)	1.462s (×2.56)	1.284s (×2.25)	1.270s (×2.23)	0.753s (×1.33)	0.731s (×1.29)	1.880s (×3.29)	2.795s
Q19.	0.032s (×3.50)	0.032s (×3.50)	0.032s (×3.50)	0.027s (×3.08)	0.010s (×1.67)	0.291s (×25.08)	0.151s (×13.42)	0.036s (×3.83)	0.397s
Q20.	**0.241s (×1.00)**	0.693s (×2.80)	0.762s (×3.08)	0.629s (×2.55)	12.090s (×48.21)	0.954s (×3.84)	0.832s (×3.35)	0.950s (×3.82)	7.958s
Q21.	0.183s (×1.50)	0.345s (×2.75)	0.553s (×4.37)	0.502s (×3.97)	0.170s (×1.40)	0.568s (×4.48)	0.289s (×2.32)	0.842s (×6.61)	0.683s
Q22.	**0.292s (×1.00)**	0.485s (×1.64)	0.928s (×3.11)	0.979s (×3.27)	10.910s (×36.16)	0.568s (×1.91)	0.591s (×1.99)	0.931s (×3.12)	5.801s
Q23.	0.182s (×1.36)	2.533s (×17.98)	4.070s (×28.85)	1.770s (×12.59)	28.250s (×199.84)	1.458s (×10.38)	2.661s (×18.89)	2.660s (×18.88)	21.765s (×
Q24.	0.042s (×1.30)	0.177s (×4.67)	0.206s (×5.40)	0.248s (×6.45)	**0.030s (×1.00)**	0.179s (×4.72)	0.190s (×5.00)	0.258s (×6.70)	0.262s
Q25.	0.035s (×3.00)	0.194s (×13.60)	0.128s (×9.20)	0.172s (×12.13)	0.060s (×4.67)	0.165s (×11.67)	0.181s (×12.73)	0.305s (×21.00)	0.164s
Q26.	0.042s (×3.33)	0.334s (×22.03)	0.177s (×11.97)	0.143s (×9.80)	0.020s (×1.92)	0.197s (×13.25)	0.220s (×14.73)	0.257s (×17.10)	0.245s
Q27.	**0.175s (×1.00)**	0.773s (×4.23)	0.528s (×2.91)	0.459s (×2.54)	0.630s (×3.46)	0.289s (×1.62)	0.368s (×2.04)	0.770s (×4.22)	0.645s
Q28.	**0.327s (×1.00)**	1.728s (×5.16)	4.037s (×12.01)	3.294s (×9.80)	8.770s (×26.05)	0.513s (×1.55)	0.677s (×2.04)	5.886s (×17.50)	5.475s
Q29.	0.036s (×2.80)	0.387s (×24.13)	0.663s (×40.91)	0.665s (×41.03)	0.120s (×7.90)	0.766s (×47.17)	0.877s (×53.92)	0.655s (×40.43)	0.045s
Q30.	**0.100s (×1.00)**	0.195s (×1.86)	0.286s (×2.69)	0.240s (×2.27)	1.330s (×12.18)	0.389s (×3.63)	0.415s (×3.86)	0.427s (×3.97)	1.094s
Q31.	**0.139s (×1.00)**	0.557s (×3.81)	0.395s (×2.72)	0.350s (×2.42)	3.460s (×23.29)	0.484s (×3.32)	1.265s (×8.56)	0.541s (×3.70)	0.644s
Q32.	1.053s (×2.22)	1.193s (×2.52)	1.704s (×3.59)	1.782s (×3.75)	0.970s (×2.05)	0.505s (×1.08)	0.786s (×1.67)	2.303s (×4.84)	4.399s
Q33.	**0.541s (×1.00)**	0.707s (×1.30)	1.307s (×2.39)	0.916s (×1.68)	0.950s (×1.74)	0.656s (×1.21)	0.905s (×1.66)	1.604s (×2.93)	1.851s
Q34.	**0.538s (×1.00)**	0.697s (×1.29)	1.276s (×2.35)	1.188s (×2.19)	0.960s (×1.77)	0.634s (×1.18)	0.864s (×1.59)	1.590s (×2.92)	6.335s
Q35.	0.200s (×1.40)	0.310s (×2.13)	0.351s (×2.41)	0.370s (×2.53)	**0.140s (×1.00)**	0.309s (×2.13)	0.352s (×2.41)	0.621s (×4.21)	0.316s
Q36.	0.066s (×1.41)	0.150s (×2.96)	0.134s (×2.67)	0.103s (×2.09)	0.070s (×1.48)	0.192s (×3.74)	0.201s (×3.91)	0.174s (×3.41)	0.152s
Q37.	0.054s (×1.78)	0.049s (×1.64)	0.070s (×2.22)	0.052s (×1.72)	0.050s (×1.67)	0.840s (×23.61)	0.143s (×4.25)	0.084s (×2.61)	0.051s
Q38.	0.054s (×2.13)	0.076s (×2.87)	0.086s (×3.20)	0.100s (×3.67)	0.040s (×1.67)	0.172s (×6.07)	0.290s (×10.00)	0.061s (×2.37)	0.065s
Q39.	**0.072s (×1.00)**	0.219s (×2.79)	0.171s (×2.21)	0.179s (×2.30)	0.100s (×1.34)	0.323s (×4.06)	0.310s (×3.90)	0.283s (×3.57)	0.180s
Q40.	0.052s (×1.88)	0.167s (×5.36)	0.063s (×2.21)	0.094s (×3.15)	0.470s (×14.55)	0.200s (×6.36)	0.212s (×6.73)	0.062s (×2.18)	0.032s
Q41.	0.046s (×2.15)	0.053s (×2.42)	0.059s (×2.65)	0.044s (×2.08)	0.250s (×10.00)	0.191s (×7.73)	0.223s (×8.96)	0.210s (×8.46)	0.029s
Q42.	0.030s (×2.00)	0.039s (×2.45)	0.061s (×3.55)	0.027s (×1.85)	0.040s (×2.50)	0.137s (×7.35)	0.172s (×9.10)	0.029s (×1.95)	0.011s

Fast

the best
nce
n-grade
$.
p priority
nproved.
use.com

ClickHouse/ClickBench: Click

https://github.com/ClickHouse/ClickBench

alloydb	Update README.md	8 months ago
athena	Remove bogus tag	2 years ago
aurora-mysql	impl	last year
aurora-postgresql	impl	last year
bigquery	Avoid too large cloud-init log	last year
brytlytdb	Add S3 Select	2 months ago
byconity	impl	last year
bytehouse	Remove undefined cluster size	2 months ago
chdb-parquet	Fix chDB (Parquet, partitioned) ...	2 months ago
chdb	Fix json tailing ,	2 months ago
citus	Avoid too large cloud-init log	last year
clickhouse-cloud	Added azure 360	3 months ago
clickhouse-datalake	ClickHouse 24.1	6 months ago
clickhouse-parquet	ClickHouse 24.1	6 months ago
clickhouse-web	ClickHouse 24.1	6 months ago
clickhouse	Update README.md	3 months ago
cloudberry	Update README.md	2 months ago
cratedb	Avoid too large cloud-init log	last year
crunchy-bridge-for-analyti	apply feedback	3 weeks ago

benchmark sql big-data analytics databases olap

Readme
View license
Activity
Custom properties
632 stars
25 watching
141 forks
Report repository

Contributors 54

+ 40 contributors

HTML 80.3% Shell 16.7%
JavaScript 2.6%
Python 0.4%

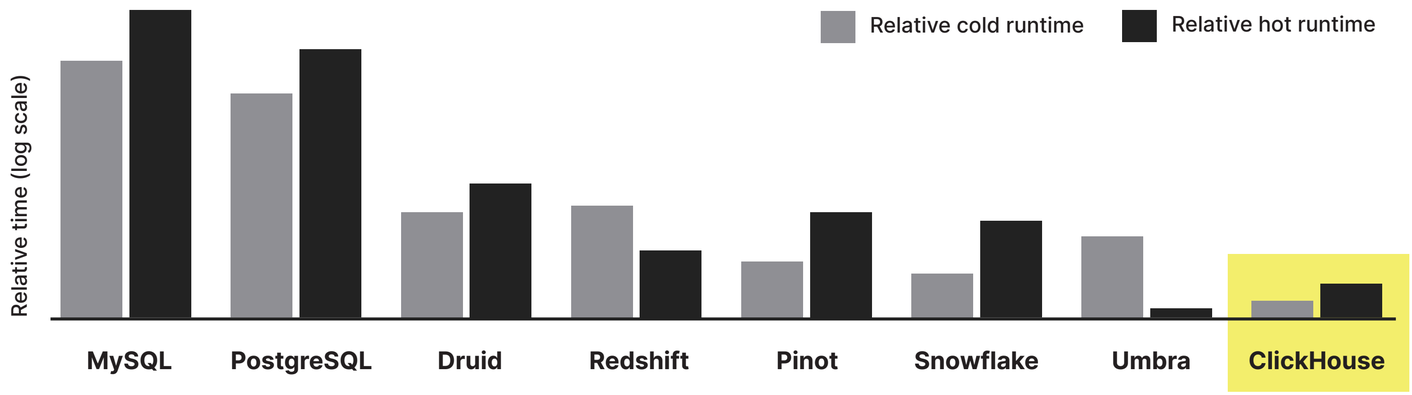
Kinetica (c6a.4xlarge, 500gb gp2):
Aurora for PostgreSQL (16acu):
MongoDB (c6a.4xlarge, 500gb gp2):
HeavyAI (c6a.4xlarge, 500gb gp2)†:
Infobright (c6a.4xlarge, 500gb gp2)†:
PostgreSQL (c6a.4xlarge, 500gb gp2):
MySQL (MyISAM) (c6a.4xlarge, 500gb gp2):
TimescaleDB (c6a.4xlarge, 500gb gp2):
SQLite (c6a.4xlarge, 500gb gp2):
MySQL (c6a.4xlarge, 500gb gp2):
Aurora for MySQL (16acu)†:
MariaDB (c6a.4xlarge, 500gb gp2)†:

Detailed Comparison

	ClickHouse (tuned, memory) (c6a.metal, 500gb gp2)	ClickHouse Cloud (aws) (1430GB)	ClickHouse Cloud (aws) (720GB)	ClickHouse Cloud (gcp) (708GB)	StarRocks (c6a.metal, 500gb
Load time:	290s (×877.65)	225s (×682.20)	220s (×667.39)	333s (×1007.92)	433s (×1312
Data size:	128.16 GiB (×13.84)	9.26 GiB (×1.00)	9.26 GiB (×1.00)	9.27 GiB (×1.00)	16.49 GiB (×1
Q0.	0.019s (×2.84)	0.014s (×2.35)	0.004s (×1.37)	0.008s (×1.76)	0.040s (×4
Q1.	0.014s (×1.00)	0.090s (×4.17)	0.118s (×5.33)	0.978s (×41.17)	0.120s (×5
Q2.	0.019s (×1.00)	0.520s (×18.28)	0.248s (×8.90)	0.171s (×6.24)	0.660s (×23
Q3.	0.029s (×1.00)	0.332s (×8.77)	0.175s (×4.74)	0.851s (×22.08)	2.080s (×53
Q4.	0.142s (×1.38)	0.529s (×4.90)	0.222s (×2.11)	0.413s (×3.85)	0.100s (×1
Q5.	0.162s (×1.00)	0.932s (×5.48)	0.405s (×2.41)	0.996s (×5.85)	2.190s (×12
Q6.	0.017s (×1.80)	0.356s (×24.40)	0.164s (×11.60)	0.119s (×8.60)	0.040s (×3
Q7.	0.030s (×1.48)	0.074s (×3.11)	0.058s (×2.52)	0.042s (×1.93)	0.060s (×2
Q8.	0.187s (×1.00)	0.502s (×2.60)	0.522s (×2.70)	0.507s (×2.62)	0.990s (×5
Q9.	0.282s (×1.00)	0.401s (×1.41)	0.552s (×1.92)	0.448s (×1.57)	0.730s (×2
Q10.	0.055s (×1.00)	0.261s (×4.17)	0.245s (×3.92)	0.290s (×4.62)	0.130s (×2
Q11.	0.050s (×1.00)	0.206s (×3.60)	0.196s (×3.43)	0.269s (×4.65)	1.040s (×17
Q12.	0.142s (×1.00)	0.229s (×1.57)	0.514s (×3.45)	0.361s (×2.44)	0.170s (×1
Q13.	0.175s (×1.00)	0.377s (×2.09)	0.662s (×3.63)	0.407s (×2.25)	0.230s (×1
Q14.	0.155s (×1.00)	0.275s (×1.73)	0.398s (×2.47)	0.452s (×2.80)	0.540s (×3
Q15.	0.131s (×1.41)	0.173s (×1.83)	0.236s (×2.46)	0.208s (×2.18)	0.090s (×1
Q16.	0.317s (×1.02)	0.484s (×1.54)	0.665s (×2.11)	0.660s (×2.09)	0.310s (×1
Q17.	0.139s (×9.65)	0.326s (×21.77)	0.474s (×31.35)	0.458s (×30.32)	0.120s (×8
Q18.	0.660s (×1.17)	0.975s (×1.71)	1.462s (×2.56)	1.284s (×2.25)	1.270s (×2
Q19.	0.032s (×3.50)	0.032s (×3.50)	0.032s (×3.50)	0.027s (×3.08)	0.010s (×1
Q20.	0.241s (×1.00)	0.693s (×2.80)	0.762s (×3.08)	0.629s (×2.55)	12.090s (×48
Q21.	0.183s (×1.50)	0.345s (×2.75)	0.553s (×4.37)	0.502s (×3.97)	0.170s (×1
Q22.	0.292s (×1.00)	0.485s (×1.64)	0.928s (×3.11)	0.979s (×3.27)	10.910s (×36
Q23.	0.182s (×1.36)	2.533s (×17.98)	4.070s (×28.85)	1.770s (×12.59)	28.250s (×199
Q24.	0.042s (×1.30)	0.177s (×4.67)	0.206s (×5.40)	0.248s (×6.45)	0.030s (×1
Q25.	0.035s (×3.00)	0.194s (×13.60)	0.128s (×9.20)	0.172s (×12.13)	0.060s (×4
Q26.	0.042s (×3.33)	0.334s (×22.03)	0.177s (×11.97)	0.143s (×9.80)	0.020s (×1
Q27.	0.175s (×1.00)	0.773s (×4.23)	0.528s (×2.91)	0.459s (×2.54)	0.630s (×3
Q28.	0.327s (×1.00)	1.728s (×5.16)	4.037s (×12.01)	3.294s (×9.80)	8.770s (×26
Q29.	0.036s (×2.80)	0.387s (×24.13)	0.663s (×40.91)	0.665s (×41.03)	0.120s (×7
Q30.	0.100s (×1.00)	0.195s (×1.86)	0.286s (×2.69)	0.240s (×2.27)	1.330s (×12
Q31.	0.139s (×1.00)	0.557s (×3.81)	0.395s (×2.72)	0.350s (×2.42)	3.460s (×23
Q32.	1.053s (×2.22)	1.193s (×2.52)	1.704s (×3.59)	1.782s (×3.75)	0.970s (×2
Q33.	0.541s (×1.00)	0.707s (×1.30)	1.307s (×2.39)	0.916s (×1.68)	0.950s (×1
Q34.	0.538s (×1.00)	0.697s (×1.29)	1.276s (×2.35)	1.188s (×2.19)	0.960s (×1
Q35.	0.200s (×1.40)	0.310s (×2.13)	0.351s (×2.41)	0.370s (×2.53)	0.140s (×1
Q36.	0.066s (×1.41)	0.150s (×2.96)	0.134s (×2.67)	0.103s (×2.09)	0.070s (×1
Q37.	0.054s (×1.78)	0.049s (×1.64)	0.070s (×2.22)	0.052s (×1.72)	0.050s (×1
Q38.	0.054s (×2.13)	0.076s (×2.87)	0.086s (×3.20)	0.100s (×3.67)	0.040s (×1
Q39.	0.072s (×1.00)	0.219s (×2.79)	0.171s (×2.21)	0.179s (×2.30)	0.100s (×1
Q40.	0.052s (×1.88)	0.167s (×5.36)	0.063s (×2.21)	0.094s (×3.15)	0.470s (×14
Q41.	0.046s (×2.15)	0.053s (×2.42)	0.059s (×2.65)	0.044s (×2.08)	0.250s (×10
Q42.	0.030s (×2.00)	0.039s (×2.45)	0.061s (×3.55)	0.027s (×1.85)	0.040s (×2

DB Engine

Fastest analytics database



ClickHouse has the best query performance amongst production-grade analytics databases. Performance is a top priority and continuously improved.

benchmark.clickhouse.com

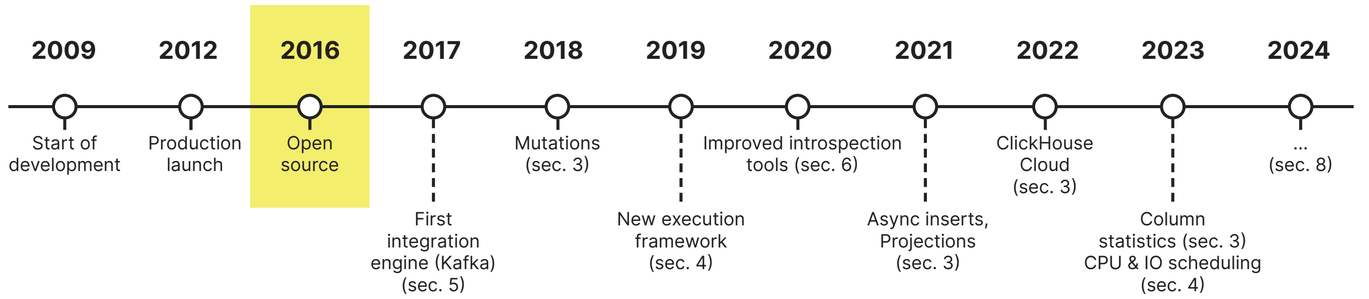
DB Engine

For everyone

Trusted by **50%+ of Fortunes Global Top 2000 companies.**

- The most popular OSS analytics database (Apache 2.0 license)
- 36k ★ and 2k+ contributors
- Runs on anything

github.com/ClickHouse/ClickHouse



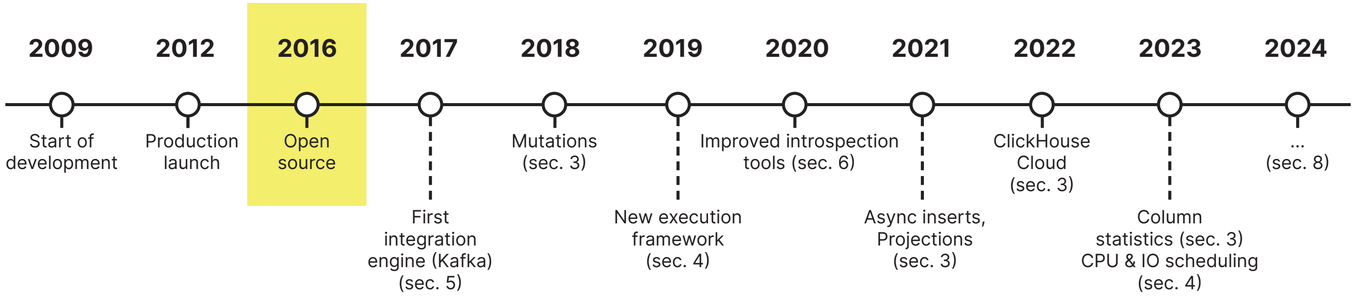
DB Engine

For everyone

Trusted by **50%+ of Fortunes Global Top 2000 companies.**

- The most popular OSS analytics database (Apache 2.0 license)
- 36k ★ and 2k+ contributors
- Runs on anything

github.com/ClickHouse/ClickHouse



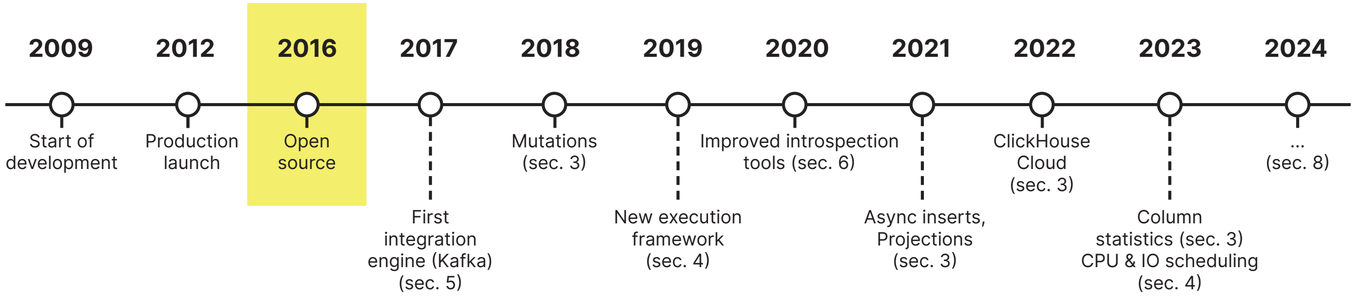
DB Engine

For everyone

Trusted by **50%+ of Fortunes Global Top 2000 companies.**

- The most popular OSS analytics database (Apache 2.0 license)
- 36k ★ and 2k+ contributors
- Runs on anything

github.com/ClickHouse/ClickHouse



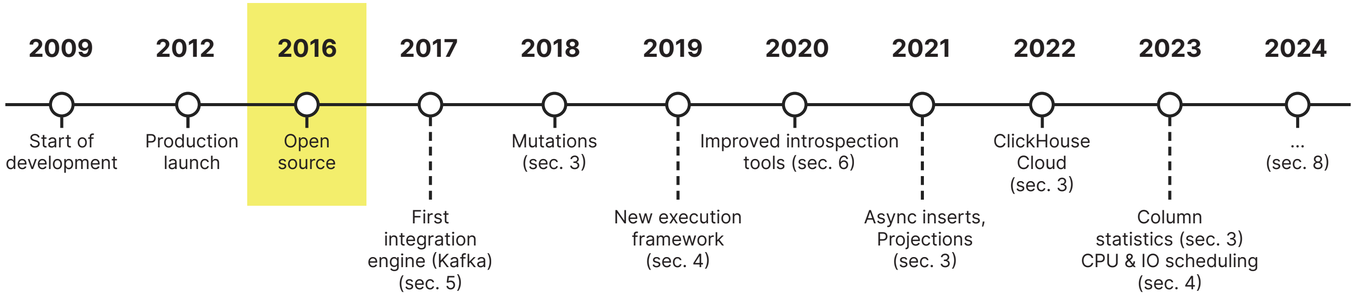
DB Engine

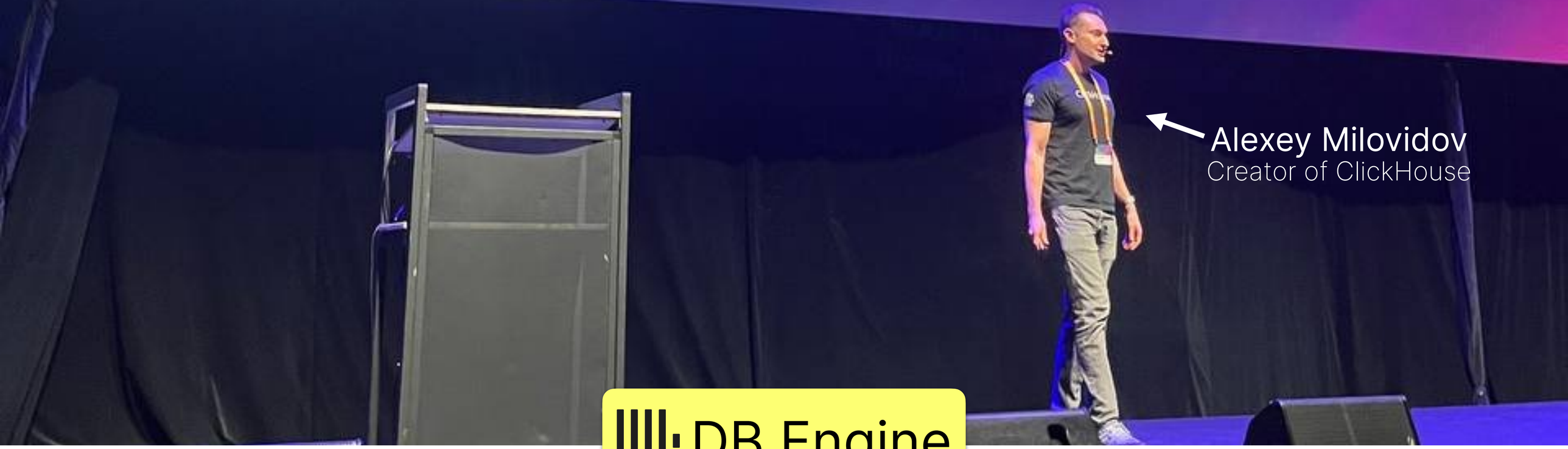
For everyone

Trusted by **50%+ of Fortunes Global Top 2000 companies.**

- The most popular OSS analytics database (Apache 2.0 license)
- 36k ★ and 2k+ contributors
- Runs on anything

github.com/ClickHouse/ClickHouse





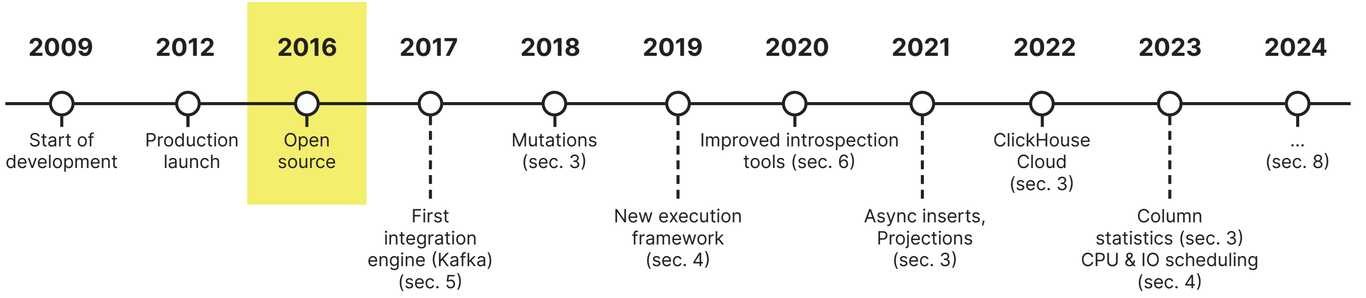
DB Engine

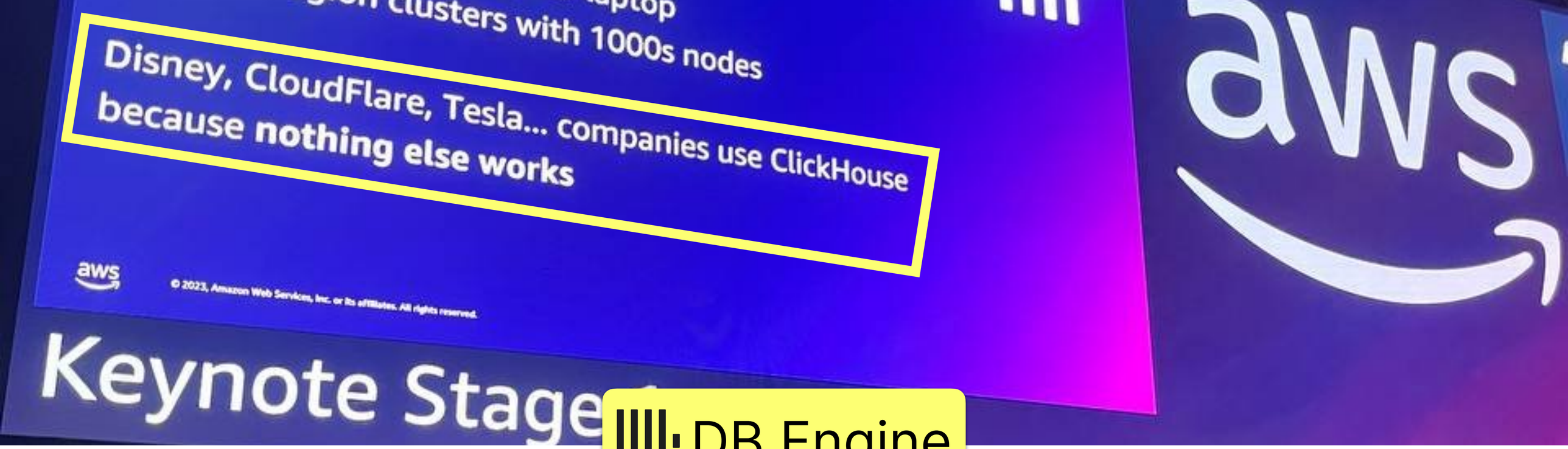
For everyone

Trusted by **50%+ of Fortunes Global Top 2000 companies.**

- The most popular OSS analytics database (Apache 2.0 license)
- 36k ★ and 2k+ contributors
- Runs on anything

github.com/ClickHouse/ClickHouse





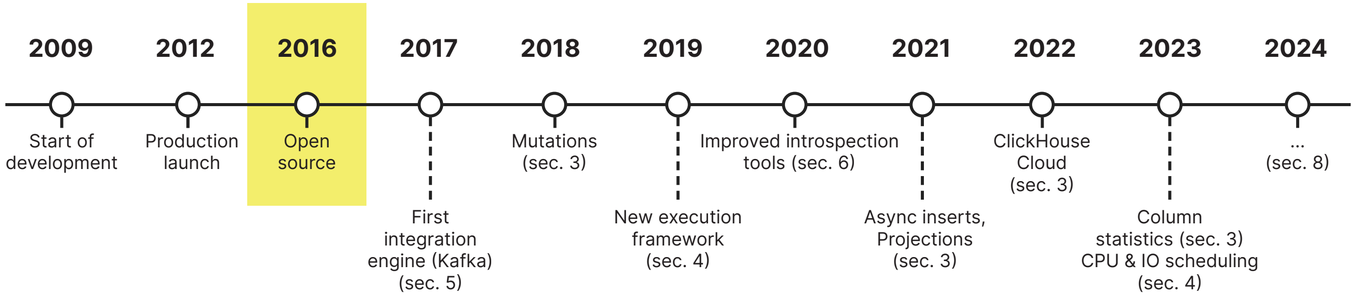
DB Engine

For everyone

Trusted by **50%+ of Fortunes Global Top 2000 companies.**

- The most popular OSS analytics database (Apache 2.0 license)
- 36k ★ and 2k+ contributors
- Runs on anything

github.com/ClickHouse/ClickHouse



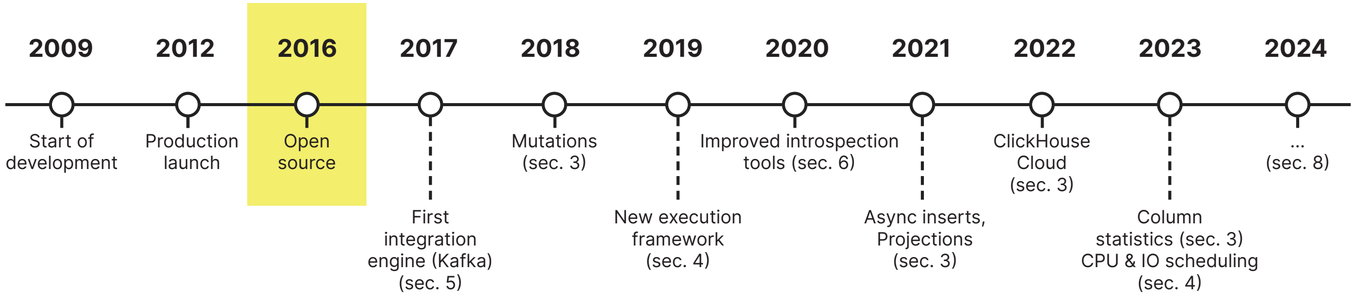
DB Engine

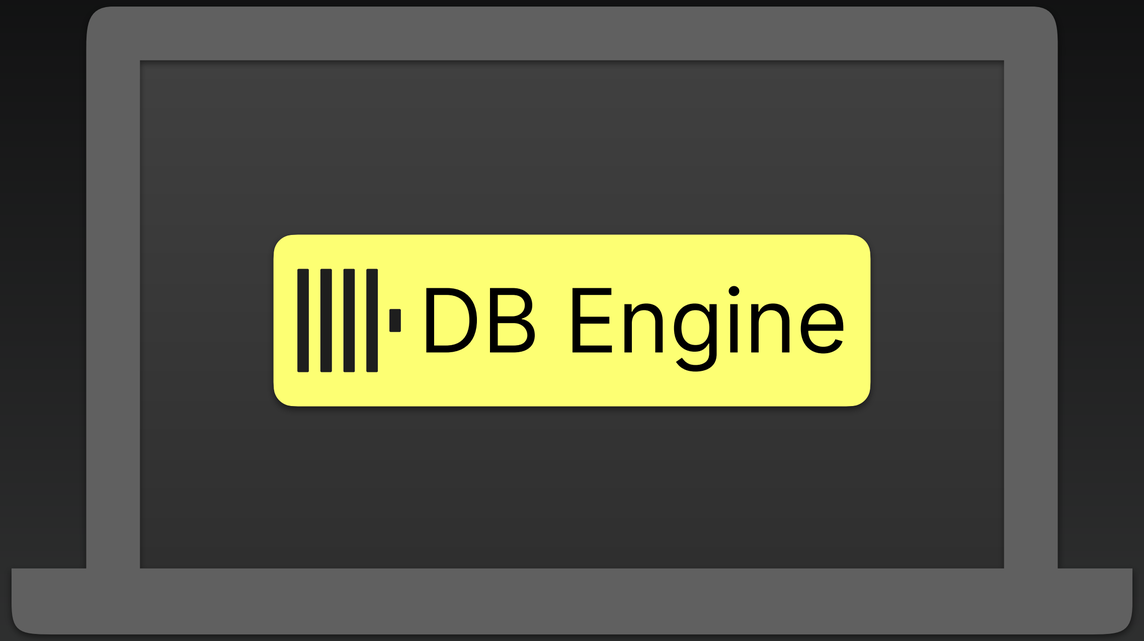
For everyone

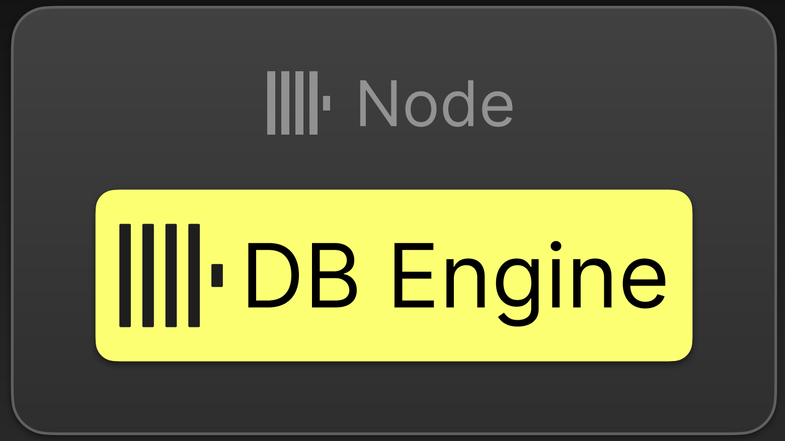
Trusted by **50%+ of Fortunes Global Top 2000 companies.**

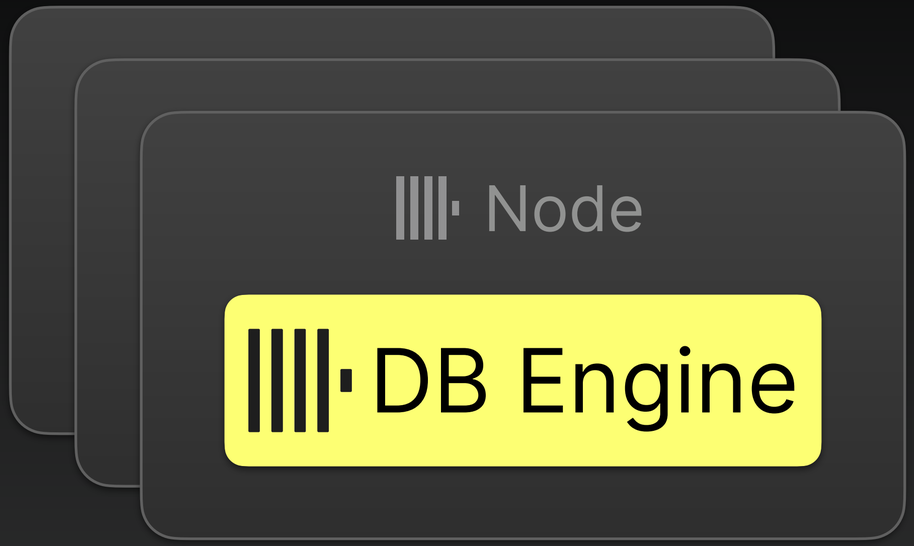
- The most popular OSS analytics database (Apache 2.0 license)
- 36k ★ and 2k+ contributors
- Runs on anything

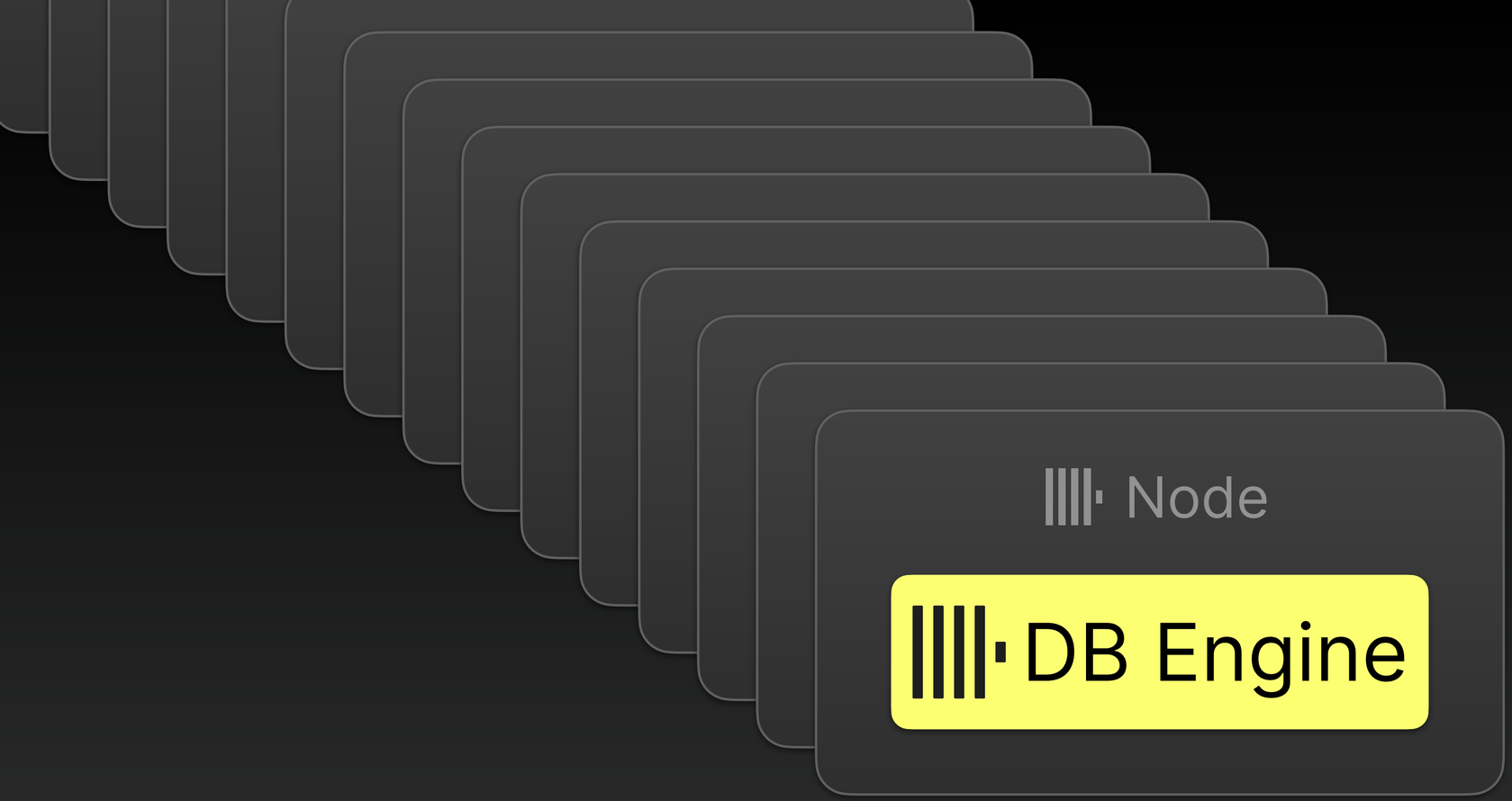
github.com/ClickHouse/ClickHouse

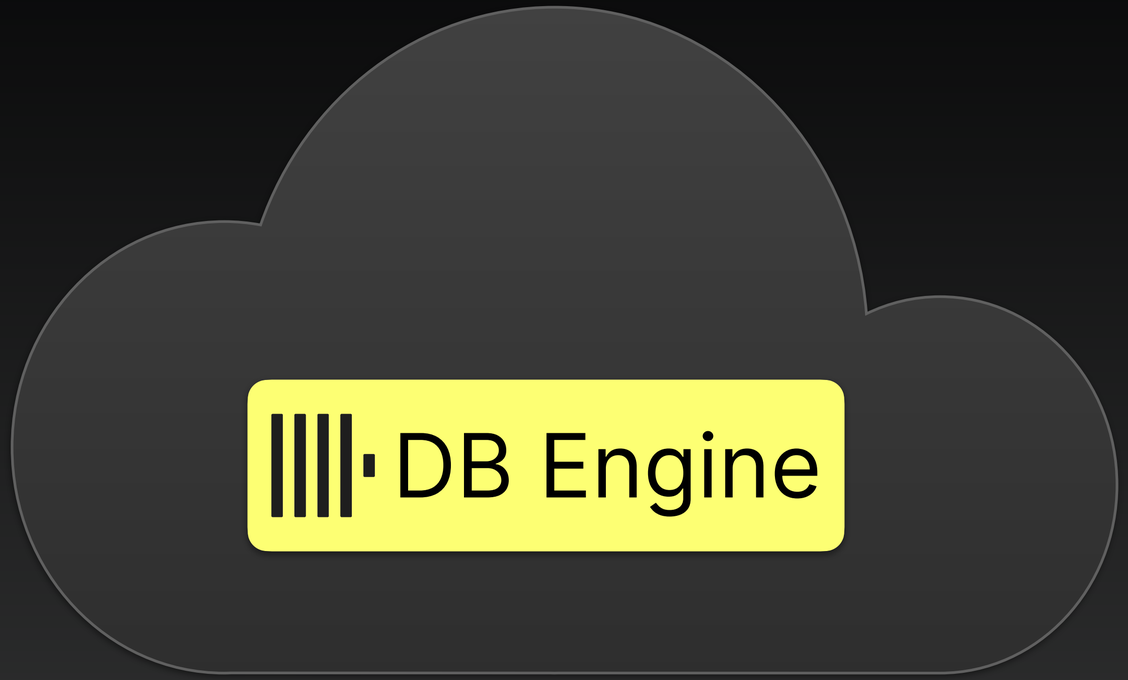


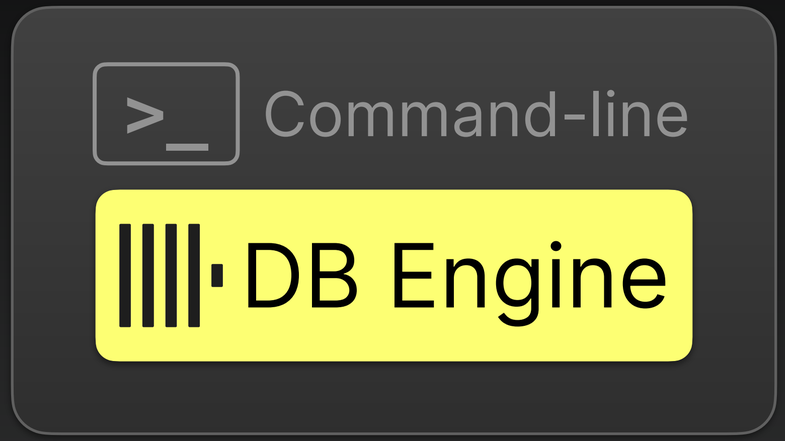

DB Engine

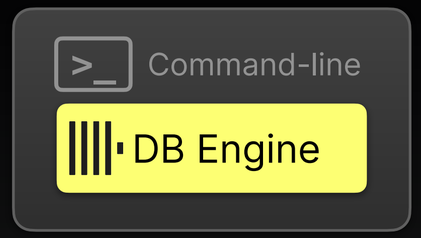

Node
DB Engine


Node
DB Engine


Node
DB Engine

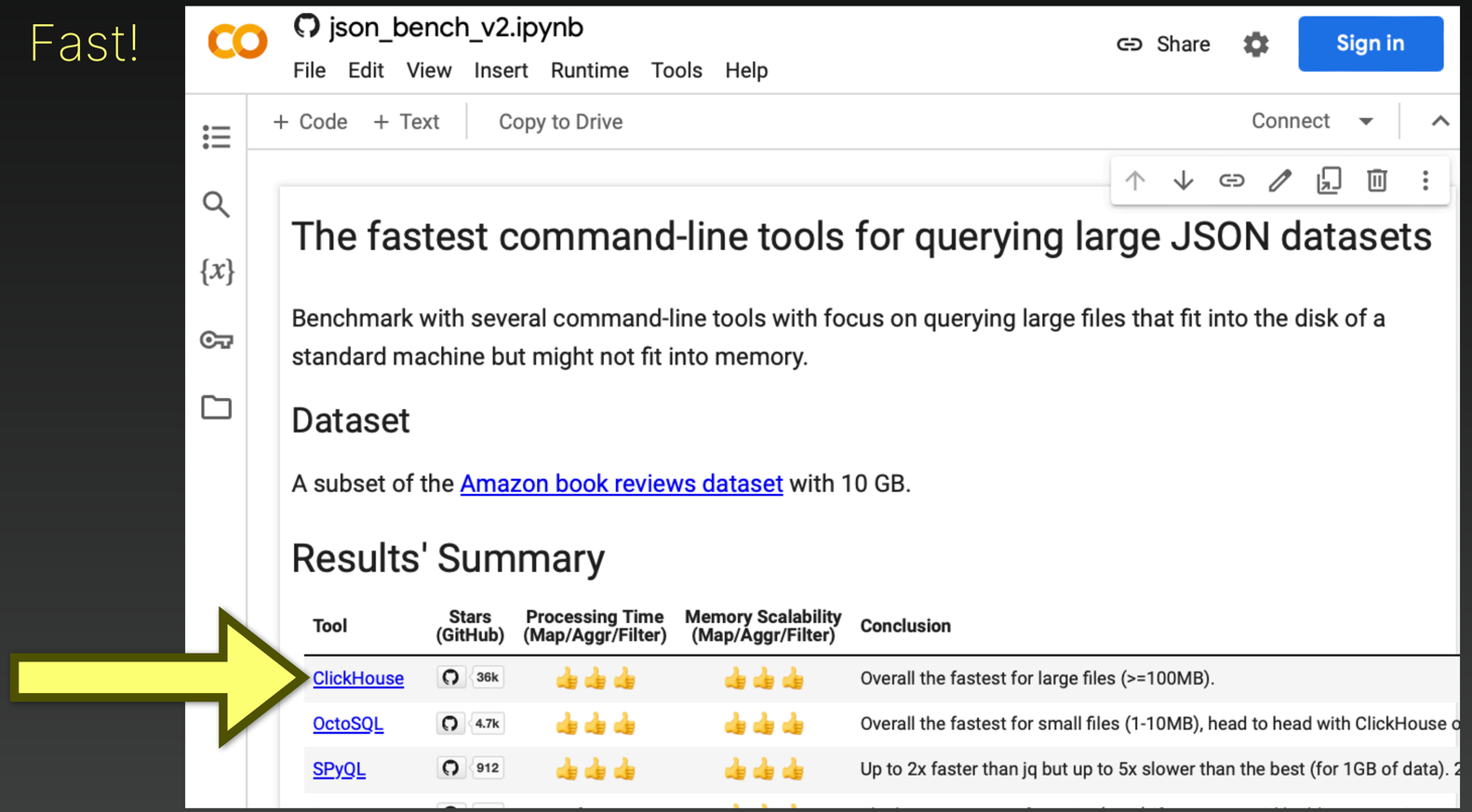

DB Engine


>_
Command-line
DB Engine



ClickHouse Local

- ClickHouse DB engine isolated as standalone command-line utility
- Serverless: No need to install/configure/start ClickHouse
- Fast!



Command-line

DB Engine

uk_price_paid.csv

uk_price_paid.csv

date	price	town	street	postcode
1995-01-04 00:00	96000	ABERYSTWYTH	NORTH ROAD	SY23 2EE
1995-01-04 00:00	15000	BRAINTREE	COGGESHALL ROAD	CM7 9EL
1995-01-04 00:00	230000	LONDON	ULLSWATER CRESCENT	SW15 3RQ
1995-01-04 00:00	31000	SWANSEA	PROSPECT PLACE	SA9 2GL

What are the top 3 districts in London
with the most sold properties?

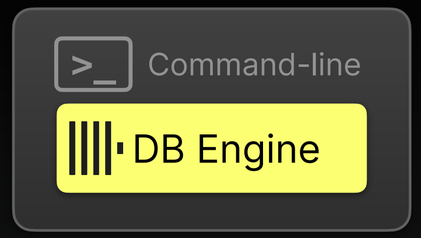
```
>_ cat sort uniq cut sed ...
```

😎

```
>_ ./clickhouse local -q "
    SELECT
      splitByChar(' ', postcode)[1] AS district,
      count() as properties
    FROM file('uk_price_paid.csv')
    WHERE town = 'LONDON'
    GROUP BY district
    ORDER BY properties DESC
    LIMIT 3"
```

50+ integrations
with external systems

90+ file formats



What are the top 3 districts in London
with the most sold properties?

```
./clickhouse local -q "
 SELECT
   splitByChar(' ', postcode)[1] AS district,
   count() as properties
 FROM file('uk_price_paid.csv')
 WHERE town = 'LONDON'
 GROUP BY district
 ORDER BY properties DESC
 LIMIT 3"
```

district	properties
E14	55765
SW11	49389
SW19	47222

Command-line

DB Engine

uk_price_paid.csv

uk_price_paid.csv

date	price	town	street	postcode
1995-01-04 00:00	96000	ABERYSTWYTH	NORTH ROAD	SY23 2EE
1995-01-04 00:00	15000	BRAINTREE	COGGESHALL ROAD	CM7 9EL
1995-01-04 00:00	230000	LONDON	ULLSWATER CRESCENT	SW15 3RQ
1995-01-04 00:00	31000	SWANSEA	PROSPECT PLACE	SA9 2GL

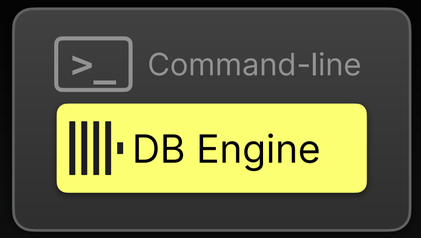
What are the top 3 districts in London
with the most sold properties?

```
./clickhouse local -q "
 SELECT
   splitByChar(' ', postcode)[1] AS district,
   count() as properties
 FROM file('uk_price_paid.csv')
 WHERE town = 'LONDON'
 GROUP BY district
 ORDER BY properties DESC
 LIMIT 3
 INTO OUTFILE 'top_3_districts.csv'
"
```

district	properties
E14	55765
SW11	49389
SW19	47222

50+ integrations
with external systems

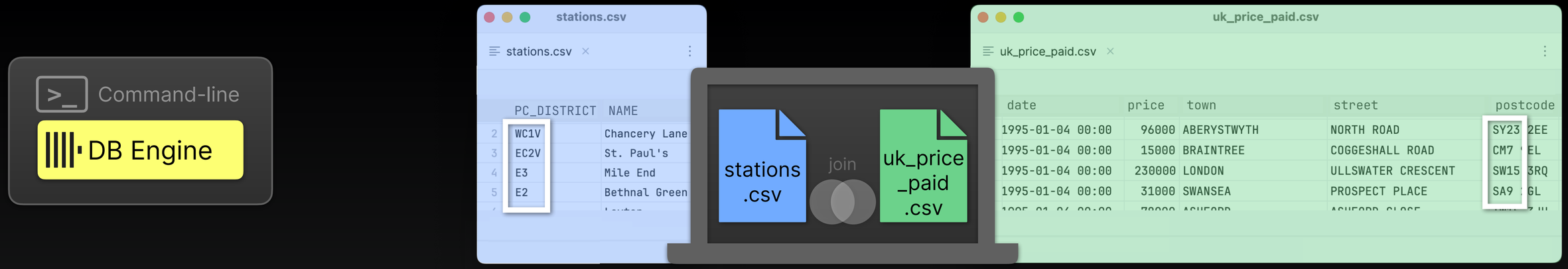
90+ file formats



What are the top 3 districts in London
with the most sold properties?

```
./clickhouse local -q "
 SELECT
   splitByChar(' ', postcode)[1] AS district,
   count() as properties
 FROM file('uk_price_paid.csv')
 WHERE town = 'LONDON'
 GROUP BY district
 ORDER BY properties DESC
 LIMIT 3"
```

district	properties
E14	55765
SW11	49389
SW19	47222

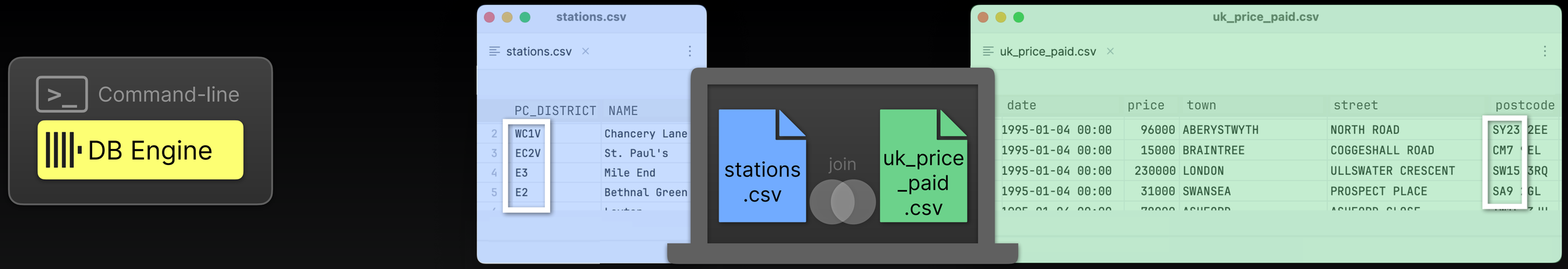


How many public transport stations are in the top 3 districts in London with the most sold properties?

What are the top 3 districts in London with the most sold properties?

```
./clickhouse local -q "
 SELECT
   splitByChar(' ', postcode)[1] AS district,
   count() as properties
 FROM file('uk_price_paid.csv')
 WHERE town = 'LONDON'
 GROUP BY district
 ORDER BY properties DESC
 LIMIT 3"
```

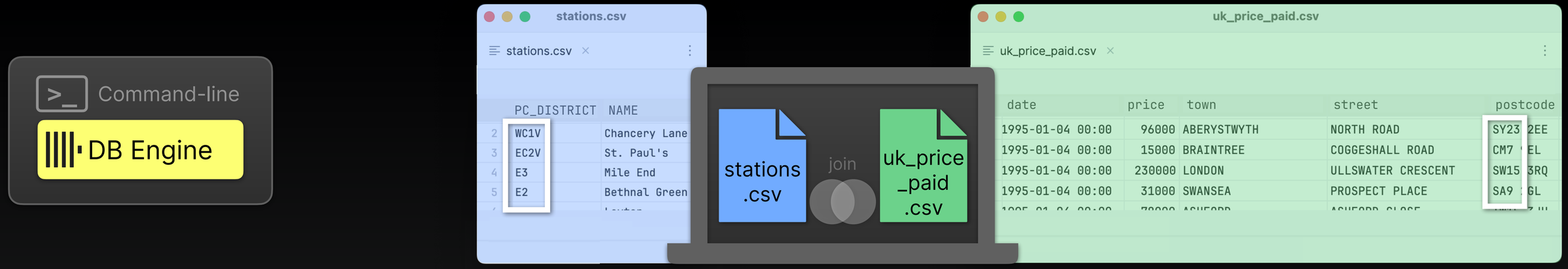
district	properties
E14	55765
SW11	49389
SW19	47222



How many public transport stations are in the top 3 districts in London with the most sold properties?

```
./clickhouse local -q "
SELECT ...
FROM file('stations.csv') AS stations
JOIN (
  SELECT
    splitByChar(' ', postcode)[1] AS district,
    count() as properties
  FROM file('uk_price_paid.csv')
  WHERE town = 'LONDON'
  GROUP BY district
  ORDER BY properties DESC
  LIMIT 3) AS properties
ON ..."
```

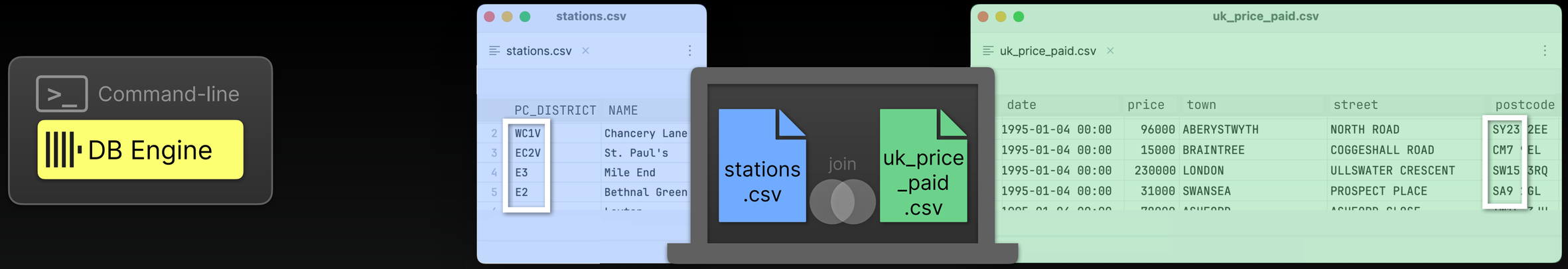
district	properties
E14	55765
SW11	49389
SW19	47222



How many public transport stations are in the top 3 districts in London with the most sold properties?

```
./clickhouse local -q "
SELECT
  district,
  any(properties) as properties,
  count() as stations
FROM file('stations.csv') AS stations
JOIN (
  SELECT
    splitByChar(' ', postcode)[1] AS district,
    count() as properties
  FROM file('uk_price_paid.csv')
  WHERE town = 'LONDON'
  GROUP BY district
  ORDER BY properties DESC
  LIMIT 3) AS properties
ON stations.PC_DISTRICT = properties.district
GROUP BY district"
```

district	properties
E14	55765
SW11	49389
SW19	47222



How many public transport stations are in the top 3 districts in London with the most sold properties?

```
./clickhouse local -q "
SELECT
  district,
  any(properties) as properties,
  count() as stations
FROM file('stations.csv') AS stations
JOIN (
  SELECT
    splitByChar(' ', postcode)[1] AS district,
    count() as properties
  FROM file('uk_price_paid.csv')
  WHERE town = 'LONDON'
  GROUP BY district
  ORDER BY properties DESC
  LIMIT 3) AS properties
ON stations.PC_DISTRICT = properties.district
GROUP BY district"
```

district	properties	stations
E14	55765	16
SW11	49389	1
SW19	47222	10

How many public transport stations are in the top 3 districts in London with the most sold properties?

```
./clickhouse local -q "
SELECT
  district,
  any(properties) as properties,
  count() as stations
FROM file('stations.csv') AS stations
JOIN (
  SELECT
    splitByChar(' ', postcode)[1] AS district,
    count() as properties
  FROM file('uk_price_paid.csv')
  WHERE town = 'LONDON'
  GROUP BY district
  ORDER BY properties DESC
  LIMIT 3) AS properties
ON stations.PC_DISTRICT = properties.district
GROUP BY district"
```

district	properties	stations
E14	55765	16
SW11	49389	1
SW19	47222	10

How many public transport stations are in the top 3 districts in London with the most sold properties?

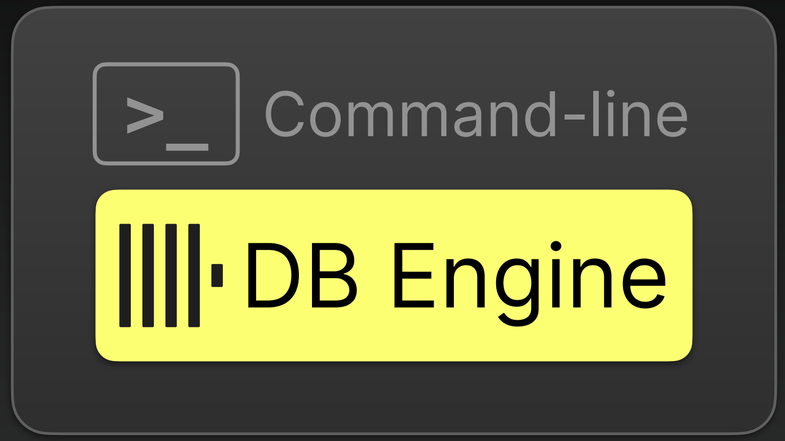
```
./clickhouse local -q "
SELECT
  district,
  any(properties) as properties,
  count() as stations
FROM file('stations.csv') AS stations
JOIN (
  SELECT
    splitByChar(' ', postcode)[1] AS district,
    count() as properties
  FROM file('uk_price_paid.csv')
  WHERE town = 'LONDON'
  GROUP BY district
  ORDER BY properties DESC
  LIMIT 3) AS properties
ON stations.PC_DISTRICT = properties.district
GROUP BY district"
```

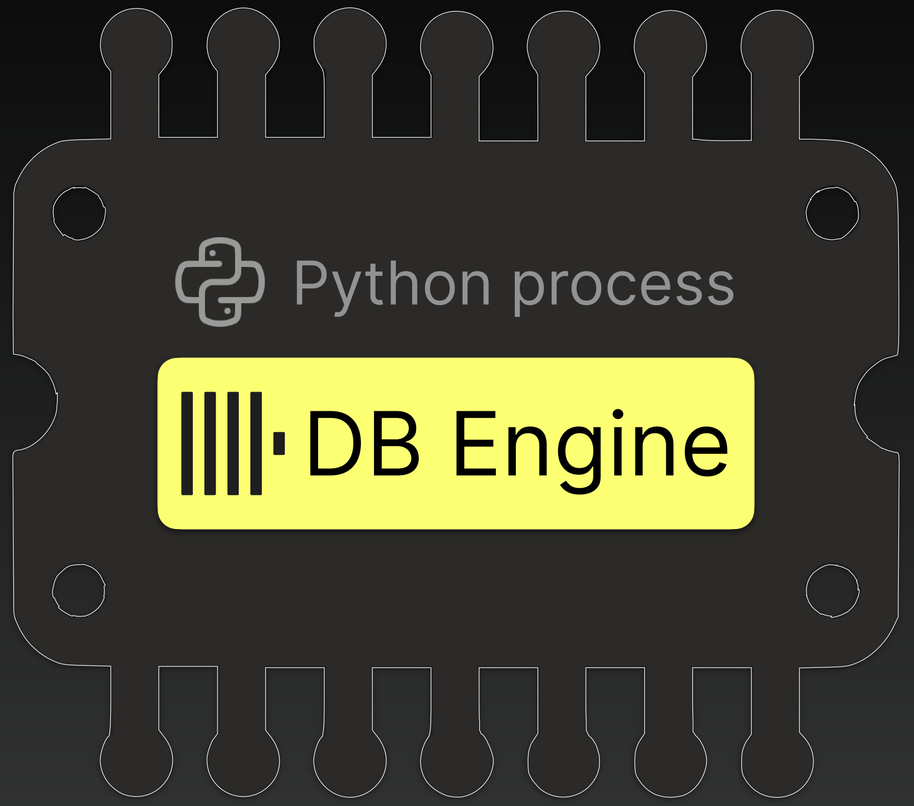
district	properties	stations
E14	55765	16
SW11	49389	1
SW19	47222	10

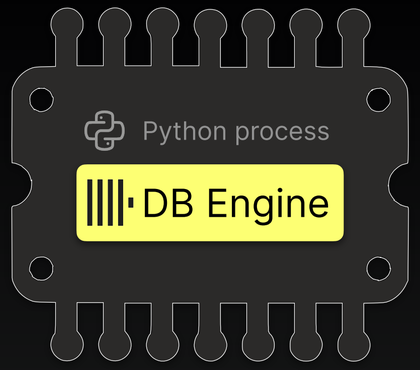
How many public transport stations are in the top 3 districts in London with the most sold properties?

```
./clickhouse local -q "
SELECT
  district,
  any(properties) as properties,
  count() as stations
FROM file('stations.csv') AS stations
JOIN (
  SELECT
    splitByChar(' ', postcode)[1] AS district,
    count() as properties
  FROM remoteSecure('HOST', ...,'uk_price_paid',...)
  WHERE town = 'LONDON'
  GROUP BY district
  ORDER BY properties DESC
  LIMIT 3) AS properties
ON stations.PC_DISTRICT = properties.district
GROUP BY district"
```

district	properties	stations
E14	55765	16
SW11	49389	1
SW19	47222	10

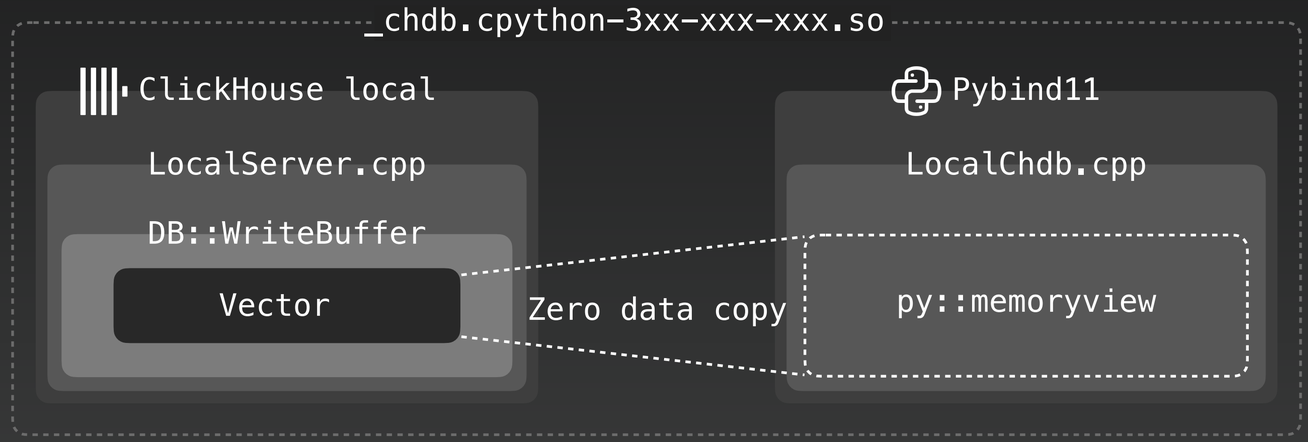

>_
Command-line
DB Engine

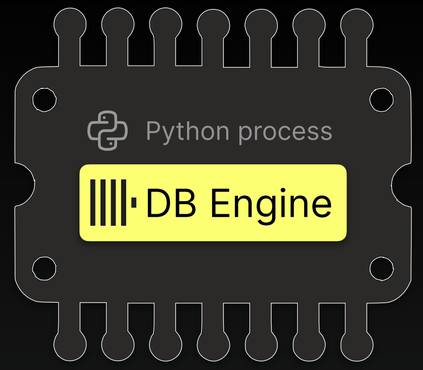

Python process
DB Engine



chDB

- Embedded in-process ClickHouse DB Engine
- Serverless: No need to install/configure/start ClickHouse
- Bindings for Python, Go, Rust, NodeJS, Bun, .NET
- Zero data copy from db engine to language library binding





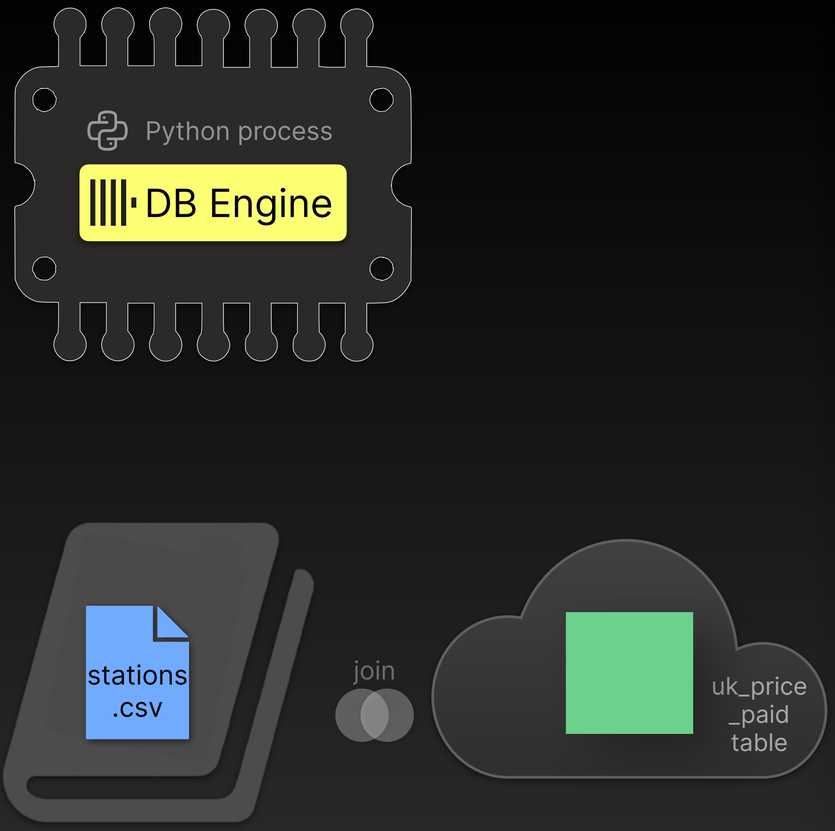
The fastest SQL on DataFrame engine in the world

Fast!

System & Machine	Relative time (lower is better)
chDB (EPYC 9654, 128G, 4TB):	×1.38
DuckDB (EPYC 9654, 128G, 4TB):	×1.49
Polars (EPYC 9654, 128G, 4TB):	×3.72
Pandas (EPYC 9654, 128G, 4TB):	×16.12

Detailed Comparison

	chDB (EPYC 9654, 128G, 4TB)	DuckDB (EPYC 9654, 128G, 4TB)	Polars (EPYC 9654, 128G, 4TB)	Pandas (EPYC 9654, 128G, 4TB)
Load time:	0	0	23s (×1.00)	0
Data size:	0.86 GiB (×1.00)	0.86 GiB (×1.00)	0.86 GiB (×1.00)	0.86 GiB (×1.00)
Q0.	0.065s (×7.50)	0.034s (×4.43)	0.000s (×1.00)	8.789s (×878.50)
Q1.	0.027s (×1.13)	0.027s (×1.10)	0.023s (×1.00)	0.162s (×5.19)
Q2.	0.024s (×1.92)	0.025s (×1.99)	0.038s (×2.70)	0.008s (×1.00)
Q3.	0.027s (×2.58)	0.022s (×2.25)	0.004s (×1.00)	0.008s (×1.24)
Q4.	0.183s (×2.19)	0.078s (×1.00)	0.133s (×1.63)	0.211s (×2.51)
Q5.	0.149s (×1.70)	0.084s (×1.00)	0.304s (×3.36)	0.669s (×7.25)
Q6.	0.026s (×2.22)	0.025s (×2.14)	0.006s (×1.00)	0.020s (×1.87)
Q7.	0.054s (×1.70)	0.046s (×1.48)	0.028s (×1.00)	0.071s (×2.13)
Q8.	0.083s (×1.00)	0.091s (×1.09)	0.261s (×2.91)	0.579s (×6.34)
Q9.	0.092s (×1.00)	0.127s (×1.34)	0.248s (×2.52)	0.682s (×6.76)
Q10.	0.094s (×1.66)	0.053s (×1.00)	0.129s (×2.21)	0.840s (×13.54)
Q11.	0.058s (×1.00)	0.059s (×1.01)	0.121s (×1.91)	0.870s (×12.84)
Q12.	0.123s (×1.22)	0.099s (×1.00)	0.200s (×1.92)	2.771s (×25.48)
Q13.	0.114s (×1.00)	0.159s (×1.36)	22.401s (×180.77)	2.760s (×22.35)
Q14.	0.109s (×1.04)	0.105s (×1.00)	0.189s (×1.74)	8.412s (×73.40)
Q15.	0.089s (×1.09)	0.081s (×1.00)	0.123s (×1.46)	0.702s (×7.83)
Q16.	0.153s (×1.08)	0.140s (×1.00)	0.200s (×1.40)	25.731s (×171.25)
Q17.	0.115s (×1.02)	0.142s (×1.24)	0.113s (×1.00)	2.752s (×22.49)
Q18.	0.196s (×1.00)	0.202s (×1.03)	0.464s (×2.30)	54.367s (×264.54)
Q19.	0.023s (×2.05)	0.029s (×2.41)	0.006s (×1.00)	0.006s (×1.01)
Q20.	0.087s (×1.00)	0.125s (×1.39)	0.183s (×2.00)	2.130s (×22.11)
Q21.	0.113s (×1.00)	0.126s (×1.10)	0.185s (×1.58)	2.547s (×20.73)
Q22.	0.184s (×1.00)	0.233s (×1.26)	0.362s (×1.92)	8.951s (×46.26)
Q23.	0.430s (×2.27)	0.476s (×2.50)	0.184s (×1.00)	2.124s (×10.99)
Q24.	0.042s (×1.00)	0.132s (×2.71)	0.279s (×5.53)	2.325s (×44.62)
Q25.	0.034s (×1.00)	0.254s (×5.93)	0.282s (×6.57)	3.937s (×88.85)
Q26.	0.046s (×1.00)	0.189s (×3.55)	0.264s (×4.88)	4.124s (×73.66)
Q27.	0.141s (×1.00)	0.155s (×1.10)	0.891s (×5.98)	11.973s (×79.56)
Q28.	0.297s (×1.00)	0.517s (×1.72)	6.253s (×20.41)	30.966s (×100.94)
Q29.	0.053s (×6.17)	0.203s (×20.96)	0.000s (×1.00)	2.759s (×272.38)
Q30.	0.065s (×1.00)	0.086s (×1.29)	0.246s (×3.43)	1.536s (×20.69)
Q31.	0.093s (×1.00)	0.106s (×1.13)	0.279s (×2.81)	1.997s (×19.49)
Q32.	0.208s (×1.00)	0.210s (×1.01)	0.580s (×2.71)	7.862s (×36.17)
Q33.	0.218s (×1.19)	0.188s (×1.03)	0.181s (×1.00)	7.771s (×40.69)
Q34.	0.195s (×1.13)	0.189s (×1.09)	0.172s (×1.00)	7.307s (×40.18)
Q35.	0.081s (×1.15)	0.102s (×1.42)	0.069s (×1.00)	32.501s (×410.84)
Q36.	0.106s (×1.71)	0.058s (×1.00)	2.075s (×30.65)	0.708s (×10.56)
Q37.	0.141s (×1.00)	0.162s (×1.13)	1.865s (×12.40)	0.680s (×4.56)
Q38.	0.122s (×2.21)	0.050s (×1.00)	0.727s (×12.31)	0.053s (×1.05)
Q39.	0.154s (×1.13)	0.134s (×1.00)	☠	0.654s (×4.60)
Q40.	0.070s (×1.31)	0.051s (×1.00)	0.231s (×3.95)	0.145s (×2.54)
Q41.	0.066s (×2.63)	0.051s (×2.12)	0.019s (×1.00)	0.072s (×2.83)
Q42.	0.052s (×1.03)	0.050s (×1.00)	☠	0.189s (×3.30)



My project | Deepnote

deepnote.com/workspace/clickhouse-2c45-3644e5e8-7d8b-457e-a95e-d2191a3a02f5/p...

ClickHouse | My project | Share | Create app

Ready | Run notebook

```
!pip install chdb==2.0.0b1
```

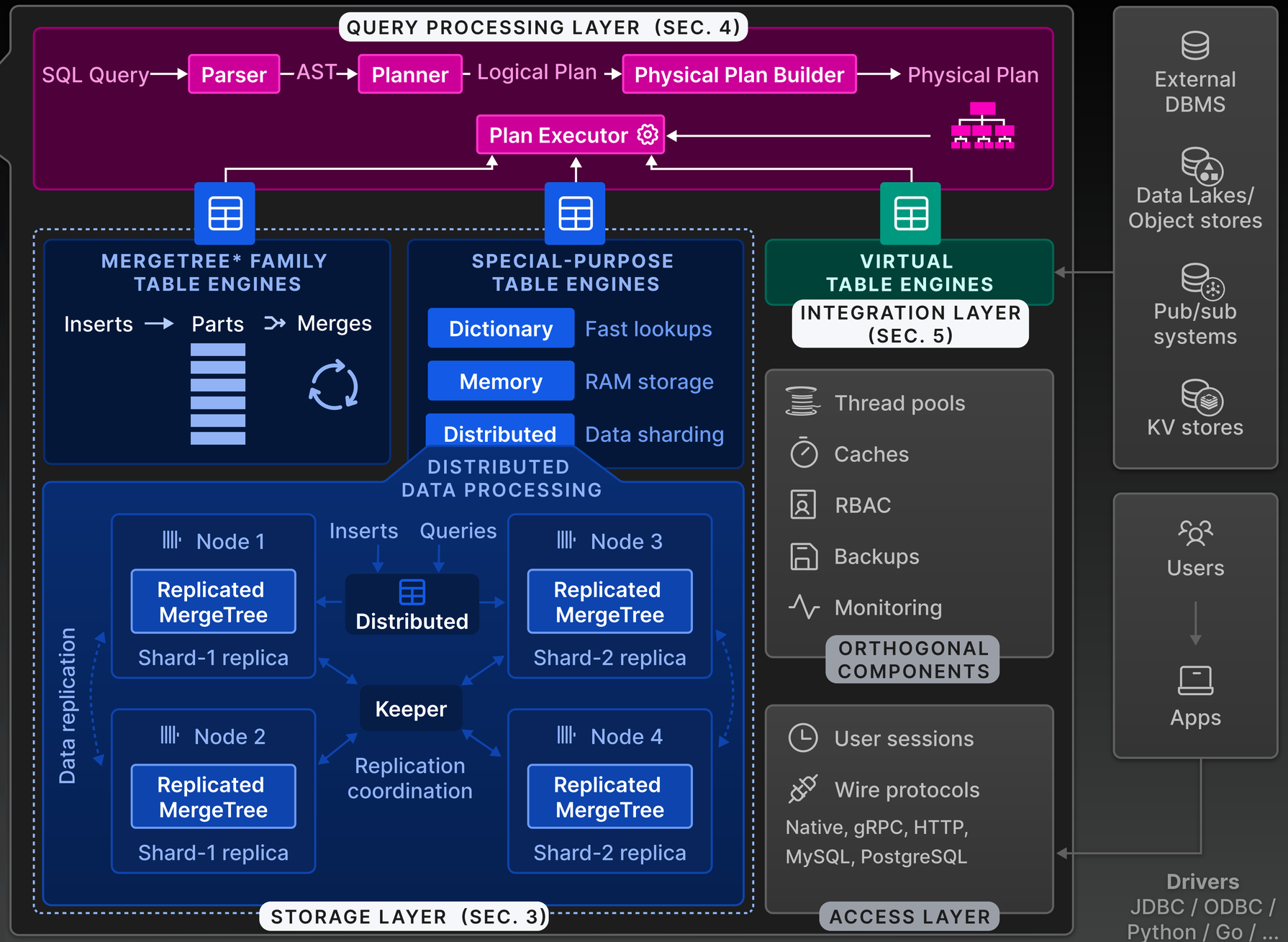
```
import chdb

chdb.sql("""
    SELECT
        district,
        any(properties) as properties,
        count() as stations
    FROM 'stations.csv' AS stations
    JOIN
        (SELECT
            postcode1 AS district,
            count() as properties
        FROM remoteSecure('HOST', ...,'uk_price_paid',...) AS properties
        WHERE town = 'LONDON'
        GROUP BY district
        ORDER BY properties DESC
        LIMIT 3) AS properties
    ON stations.PC_DISTRICT = properties.district
    GROUP BY district
""", "DataFrame")
```

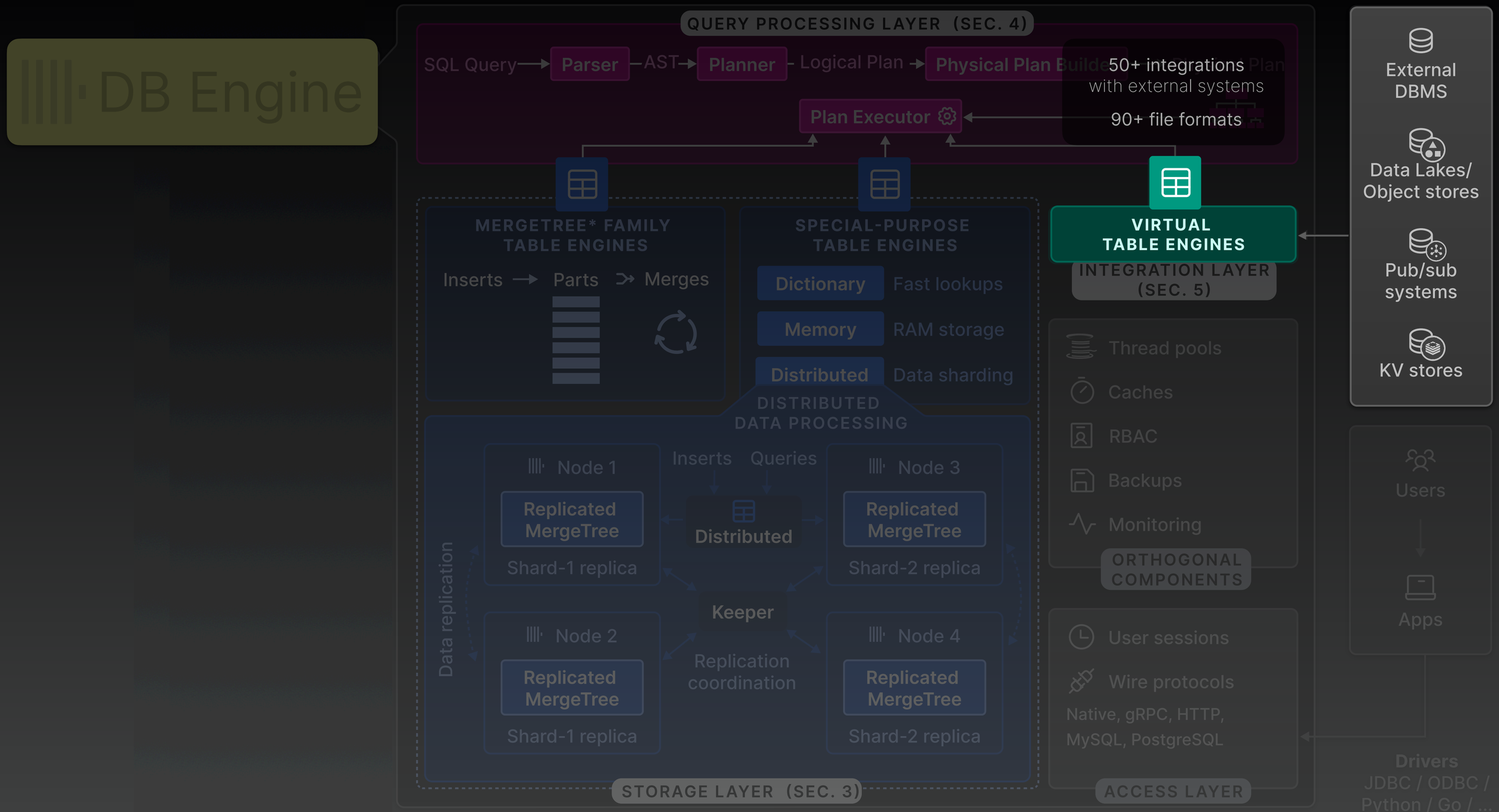
Visualize

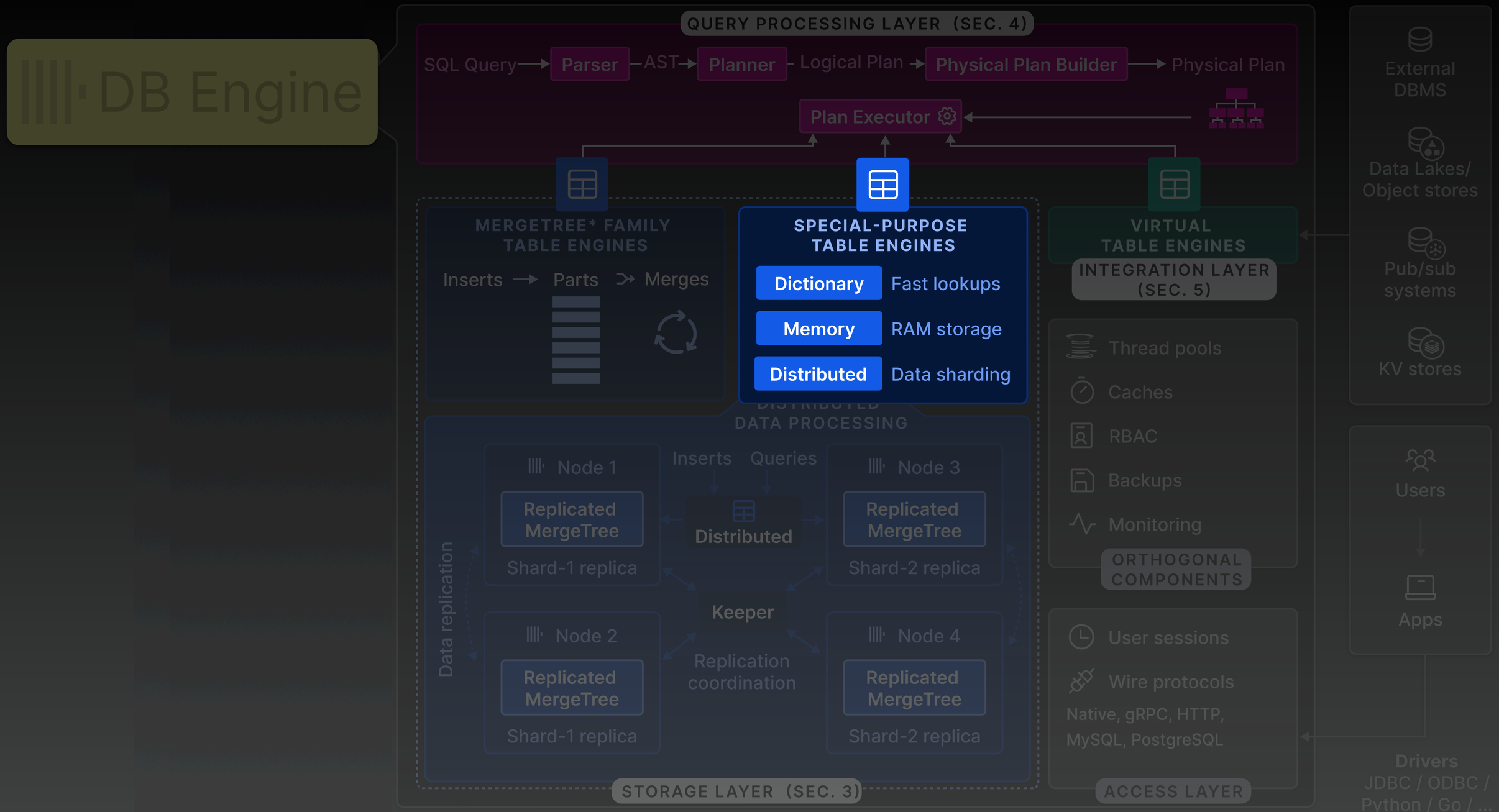
	district object	properties uint64	stations uint64
0	SW19	47222	10
1	SW11	49389	1
2	E14	55765	16

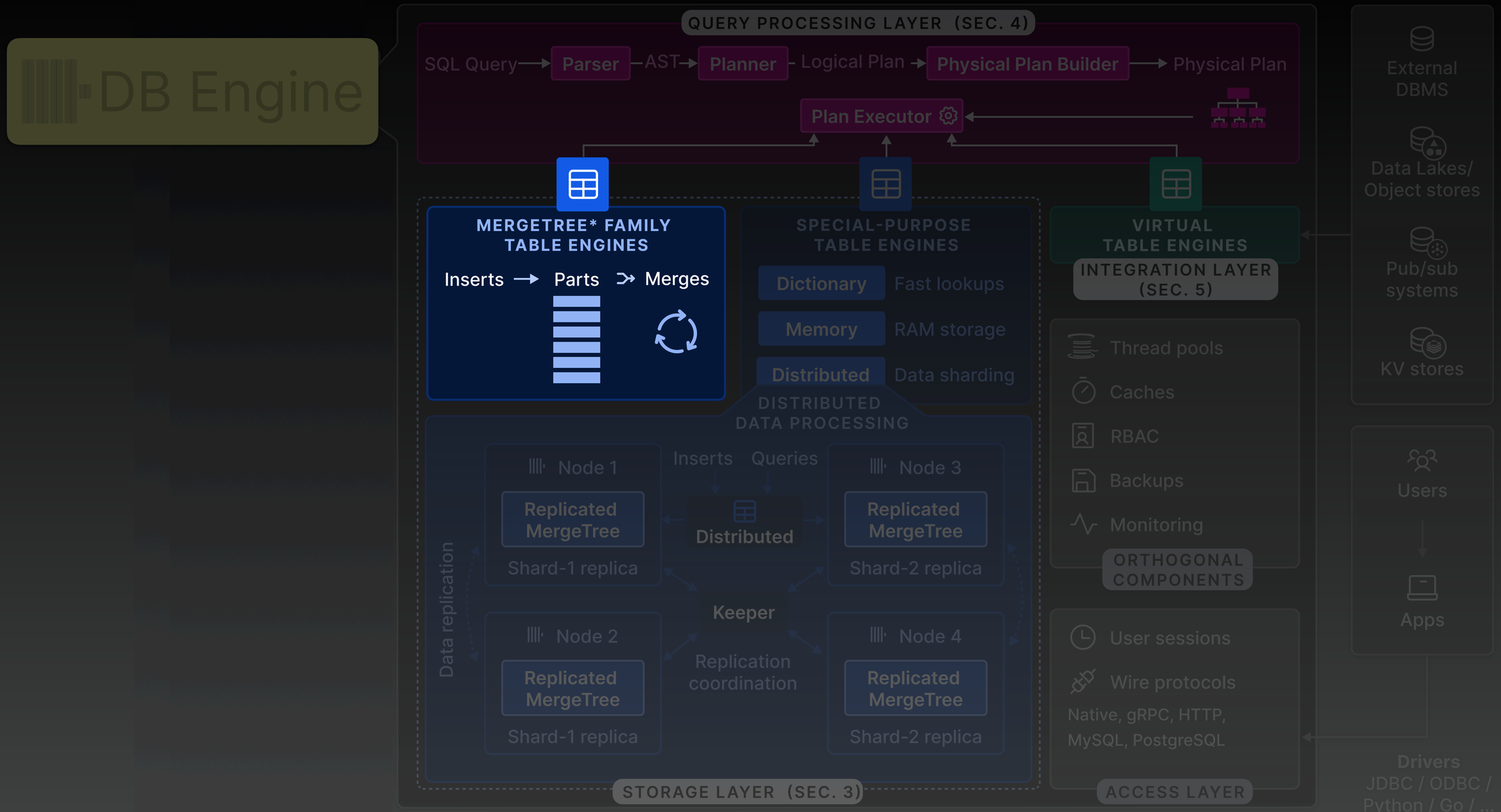
3 rows, 3 cols, showing 10 rows/page | Page 1 of 1

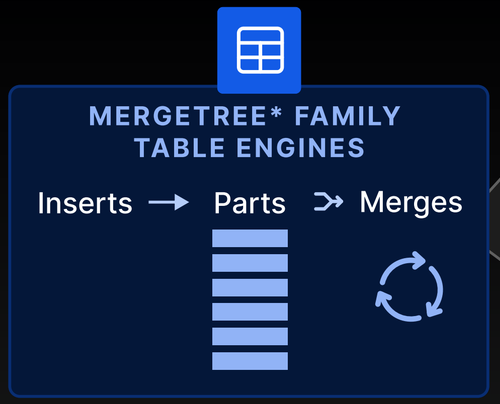
DB Engine

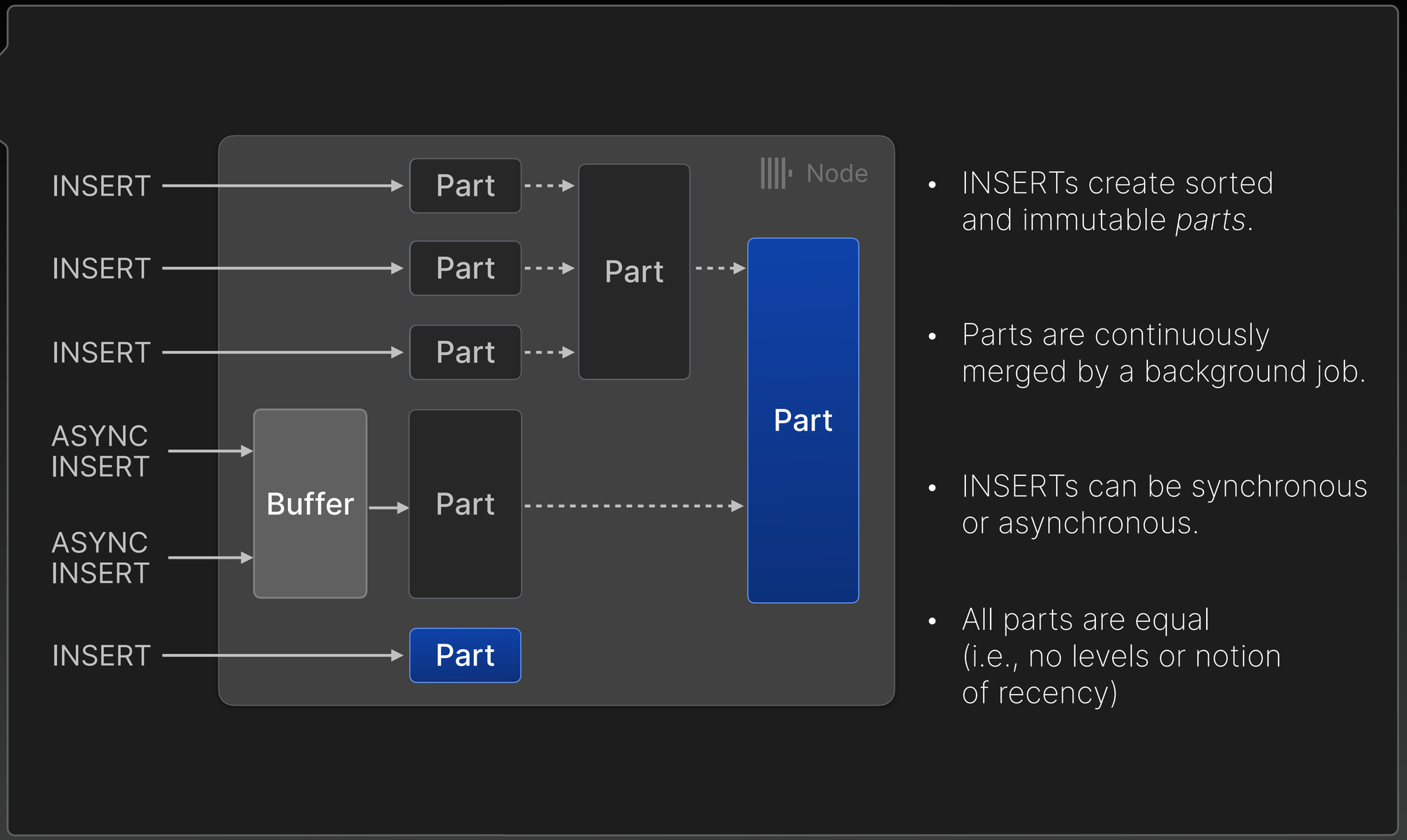

QUERY PROCESSING LAYER (SEC. 4)
SQL Query
Parser
AST
Planner
Logical Plan
Physical Plan Builder
Physical Plan
Plan Executor
MERGETREE* FAMILY TABLE ENGINES
Inserts
Parts
Merges
SPECIAL-PURPOSE TABLE ENGINES
Dictionary
Fast lookups
Memory
RAM storage
Distributed
Data sharding
VIRTUAL TABLE ENGINES
INTEGRATION LAYER (SEC. 5)
DISTRIBUTED DATA PROCESSING
Node 1
Replicated MergeTree
Shard-1 replica
Inserts
Queries
Distributed
Node 3
Replicated MergeTree
Shard-2 replica
Data replication
Keeper
Replication coordination
Node 2
Replicated MergeTree
Shard-1 replica
Node 4
Replicated MergeTree
Shard-2 replica
STORAGE LAYER (SEC. 3)
Thread pools
Caches
RBAC
Backups
Monitoring
ORTHOGONAL COMPONENTS
User sessions
Wire protocols
Native, gRPC, HTTP, MySQL, PostgreSQL
ACCESS LAYER
External DBMS
Data Lakes/ Object stores
Pub/sub systems
KV stores
Users
Apps
Drivers
JDBC / ODBC / Python / Go / ...


DB Engine
QUERY PROCESSING LAYER (SEC. 4)
SQL Query
Parser
AST
Planner
Logical Plan
Physical Plan Builder
Plan Executor
50+ integrations with external systems
90+ file formats
MERGETREE* FAMILY TABLE ENGINES
Inserts
Parts
Merges
SPECIAL-PURPOSE TABLE ENGINES
Dictionary
Fast lookups
Memory
RAM storage
Distributed
Data sharding
VIRTUAL TABLE ENGINES
INTEGRATION LAYER (SEC. 5)
External DBMS
Data Lakes/ Object stores
Pub/sub systems
KV stores
DISTRIBUTED DATA PROCESSING
Node 1
Replicated MergeTree
Shard-1 replica
Inserts
Queries
Distributed
Node 3
Replicated MergeTree
Shard-2 replica
Keeper
Data replication
Node 2
Replicated MergeTree
Shard-1 replica
Replication coordination
Node 4
Replicated MergeTree
Shard-2 replica
Thread pools
Caches
RBAC
Backups
Monitoring
ORTHOGONAL COMPONENTS
User sessions
Wire protocols
Native, gRPC, HTTP, MySQL, PostgreSQL
Users
Apps
Drivers
JDBC / ODBC / Python / Go / ...
STORAGE LAYER (SEC. 3)
ACCESS LAYER


DB Engine
QUERY PROCESSING LAYER (SEC. 4)
SQL Query
Parser
AST
Planner
Logical Plan
Physical Plan Builder
Physical Plan
Plan Executor
MERGETREE* FAMILY TABLE ENGINES
Inserts
Parts
Merges
SPECIAL-PURPOSE TABLE ENGINES
Dictionary
Fast lookups
Memory
RAM storage
Distributed
Data sharding
VIRTUAL TABLE ENGINES
INTEGRATION LAYER (SEC. 5)
DISTRIBUTED DATA PROCESSING
Node 1
Replicated MergeTree
Shard-1 replica
Inserts
Queries
Distributed
Node 3
Replicated MergeTree
Shard-2 replica
Data replication
Keeper
Replication coordination
Node 2
Replicated MergeTree
Shard-1 replica
Node 4
Replicated MergeTree
Shard-2 replica
STORAGE LAYER (SEC. 3)
Thread pools
Caches
RBAC
Backups
Monitoring
ORTHOGONAL COMPONENTS
User sessions
Wire protocols
Native, gRPC, HTTP, MySQL, PostgreSQL
ACCESS LAYER
External DBMS
Data Lakes/ Object stores
Pub/sub systems
KV stores
Users
Apps
Drivers
JDBC / ODBC / Python / Go / ...

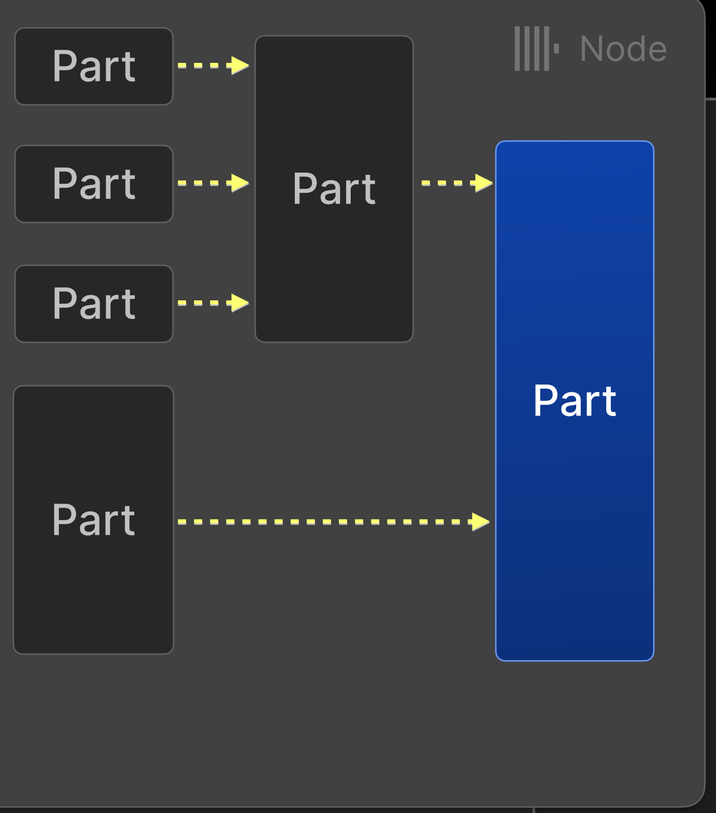

DB Engine
QUERY PROCESSING LAYER (SEC. 4)
SQL Query
Parser
AST
Planner
Logical Plan
Physical Plan Builder
Physical Plan
Plan Executor
MERGETREE* FAMILY TABLE ENGINES
Inserts
Parts
Merges
SPECIAL-PURPOSE TABLE ENGINES
Dictionary
Fast lookups
Memory
RAM storage
Distributed
Data sharding
VIRTUAL TABLE ENGINES
INTEGRATION LAYER (SEC. 5)
DISTRIBUTED DATA PROCESSING
Node 1
Replicated MergeTree
Shard-1 replica
Inserts
Queries
Distributed
Node 3
Replicated MergeTree
Shard-2 replica
Keeper
Node 2
Replicated MergeTree
Shard-1 replica
Replication coordination
Node 4
Replicated MergeTree
Shard-2 replica
Data replication
STORAGE LAYER (SEC. 3)
Thread pools
Caches
RBAC
Backups
Monitoring
ORTHOGONAL COMPONENTS
User sessions
Wire protocols
Native, gRPC, HTTP, MySQL, PostgreSQL
ACCESS LAYER
External DBMS
Data Lakes/ Object stores
Pub/sub systems
KV stores
Users
Apps
Drivers
JDBC / ODBC / Python / Go / ...





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

An LSM-tree inspired storage layer

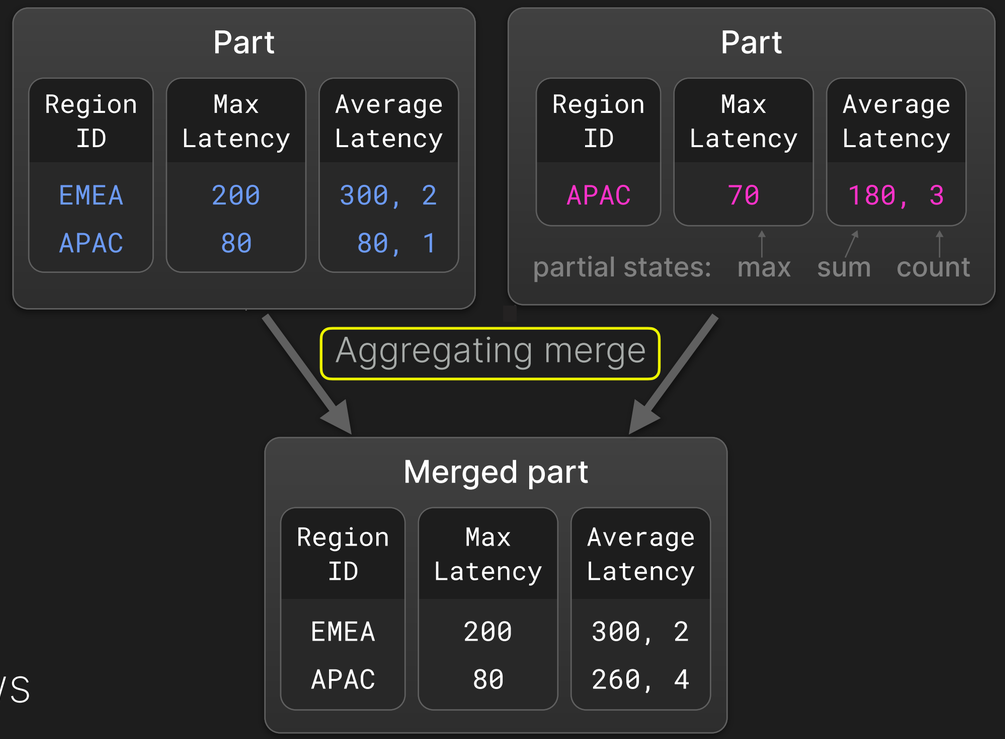


Merges optionally perform additional data transformations or maintenance.

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

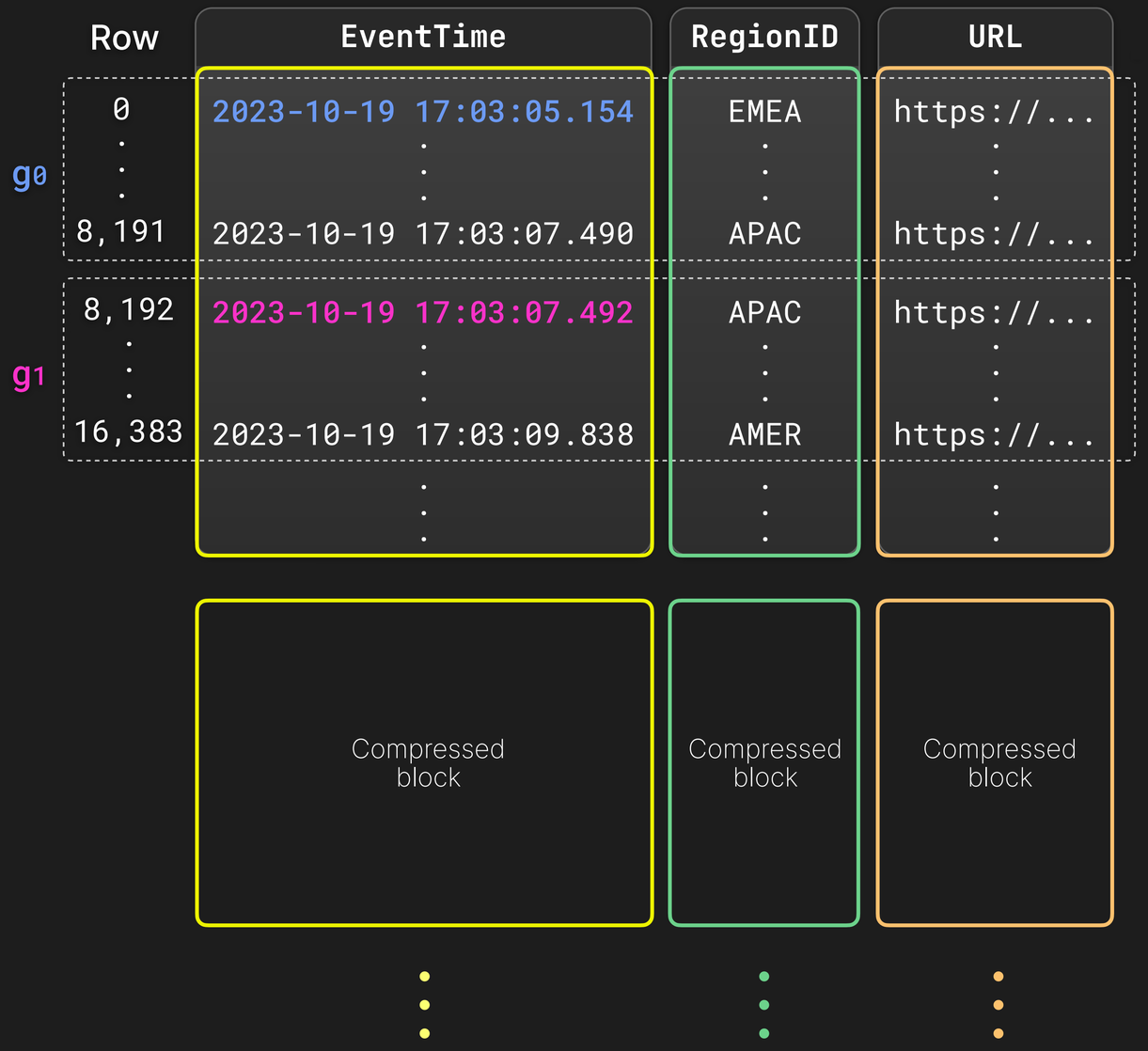
Aggregating merges
Combine aggregation states into new aggregation states.

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



Merge modes

Part

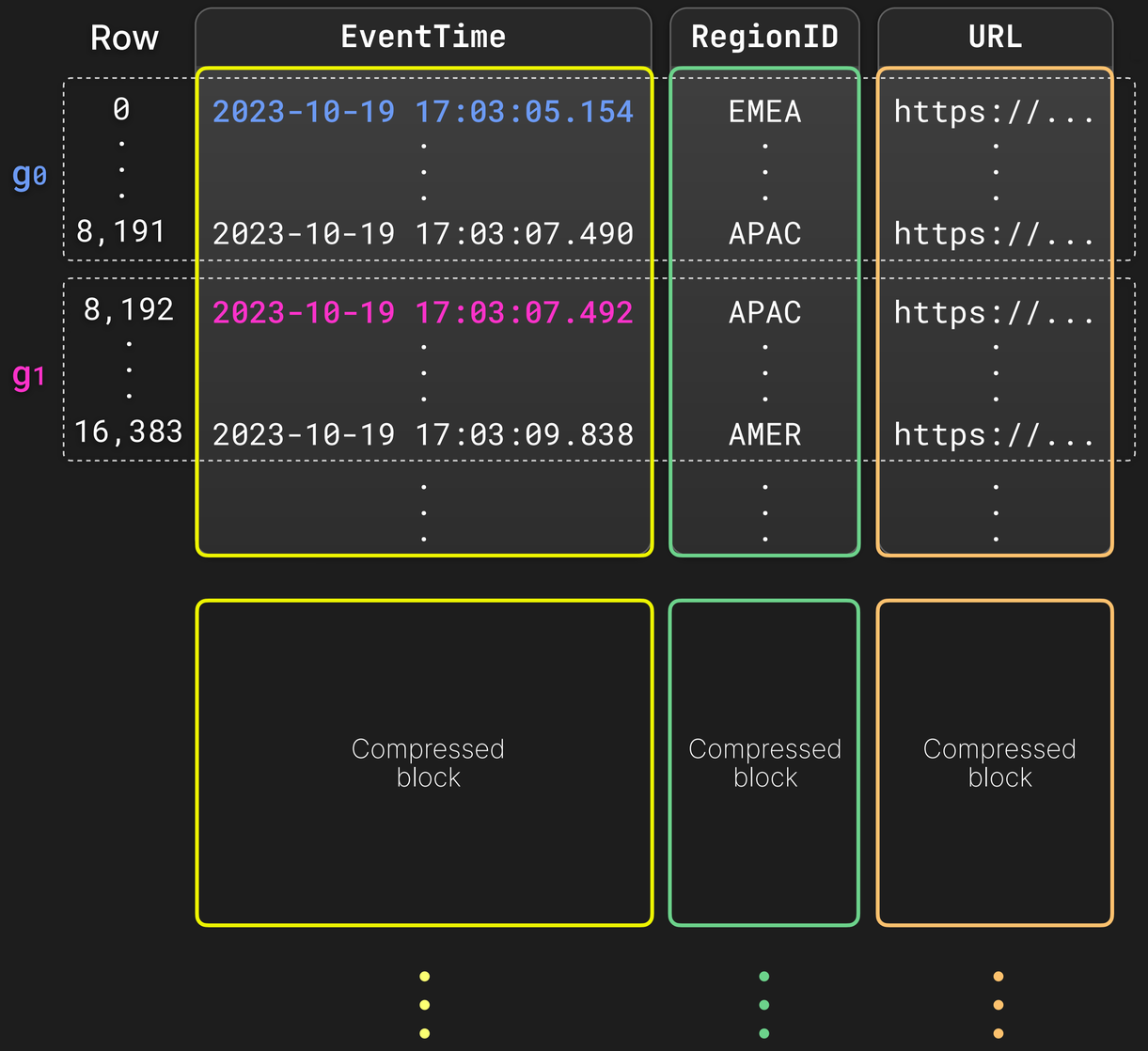



```
CREATE TABLE page_hits
(
    EventTime Date    CODEC(Delta, ZSTD),
    RegionId  String  CODEC(LZ4),
    URL       String  CODEC(AES),
    PRIMARY KEY (EventTime)
)
```

- Parts are further divided into *granules* `g0`, `g1`, ...
- Consecutive granules in a column form *blocks*
- Blocks are encoded, codecs can be combined.

Column layout and compression

Part




```
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*
- Blocks are encoded, codecs can be combined.

Column layout and compression

Part

	Row	EventTime	RegionID	URL
g0	0	2023-10-19 17:03:05.154	EMEA	https://...
	⋮	⋮	⋮	⋮
	8,191	2023-10-19 17:03:07.490	APAC	https://...
g1	8,192	2023-10-19 17:03:07.492	APAC	https://...
	⋮	⋮	⋮	⋮
	16,383	2023-10-19 17:03:09.838	AMER	https://...
		⋮	⋮	⋮

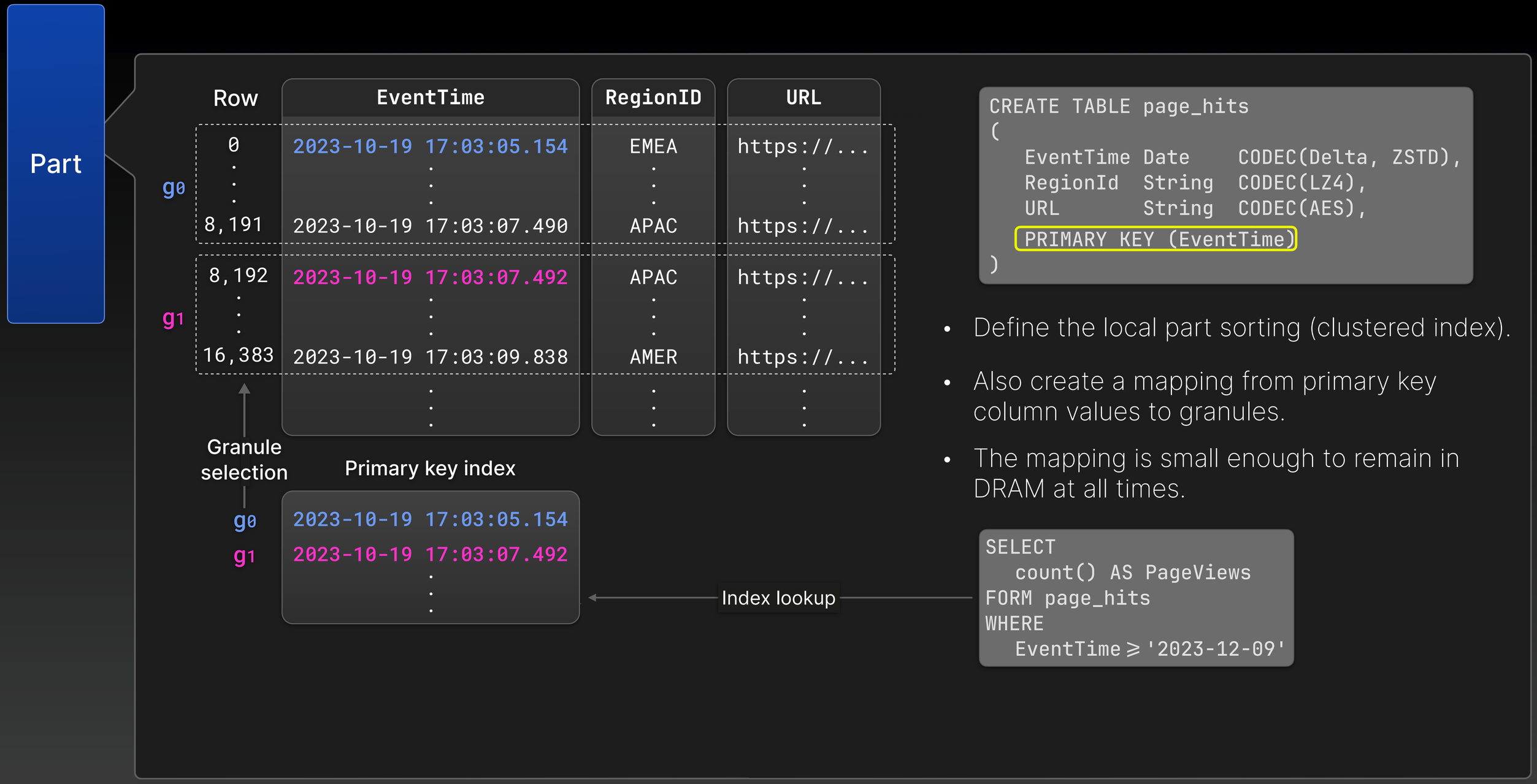
```
CREATE TABLE page_hits
(
   EventTime Date    CODEC(Delta, ZSTD),
   RegionId  String  CODEC(LZ4),
   URL       String  CODEC(AES),
   PRIMARY KEY (EventTime)
)
```

- 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.

Index lookup

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

Data pruning - Primary key indexes



Data pruning - Primary key indexes

Part

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 the primary key columns.

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

Data pruning - Table projections

Part

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 the primary key columns.

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

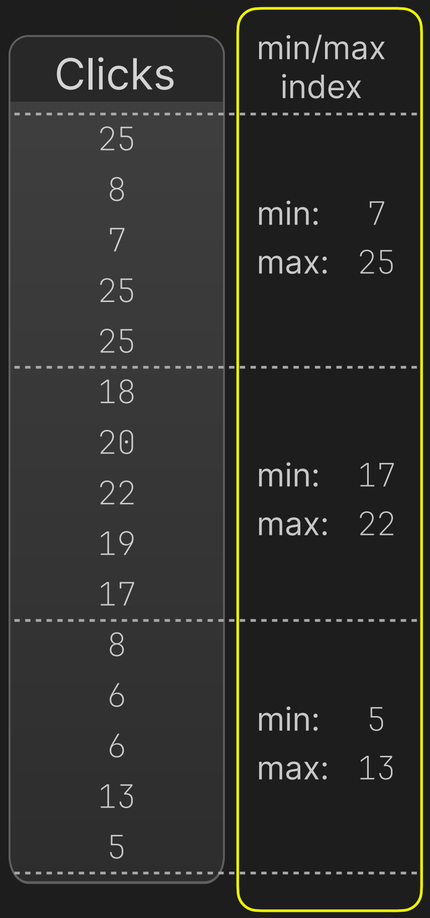
Data pruning - Table projections

Part

```
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
```

- Light-weight alternative to projections.
- 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
  - ...



Some match → Load and Scan block

All match → SKIP load

None match → SKIP load

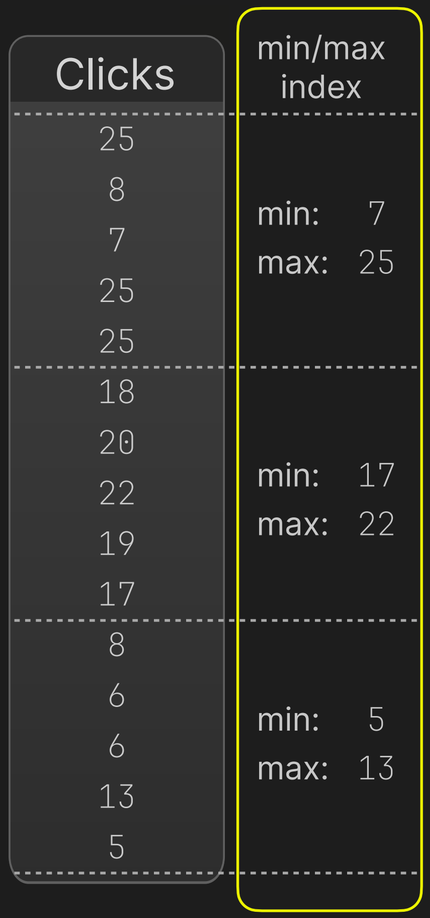
Data pruning - Skipping indexes

Part

```
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
```

- Light-weight alternative to projections.
- 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
  - ...

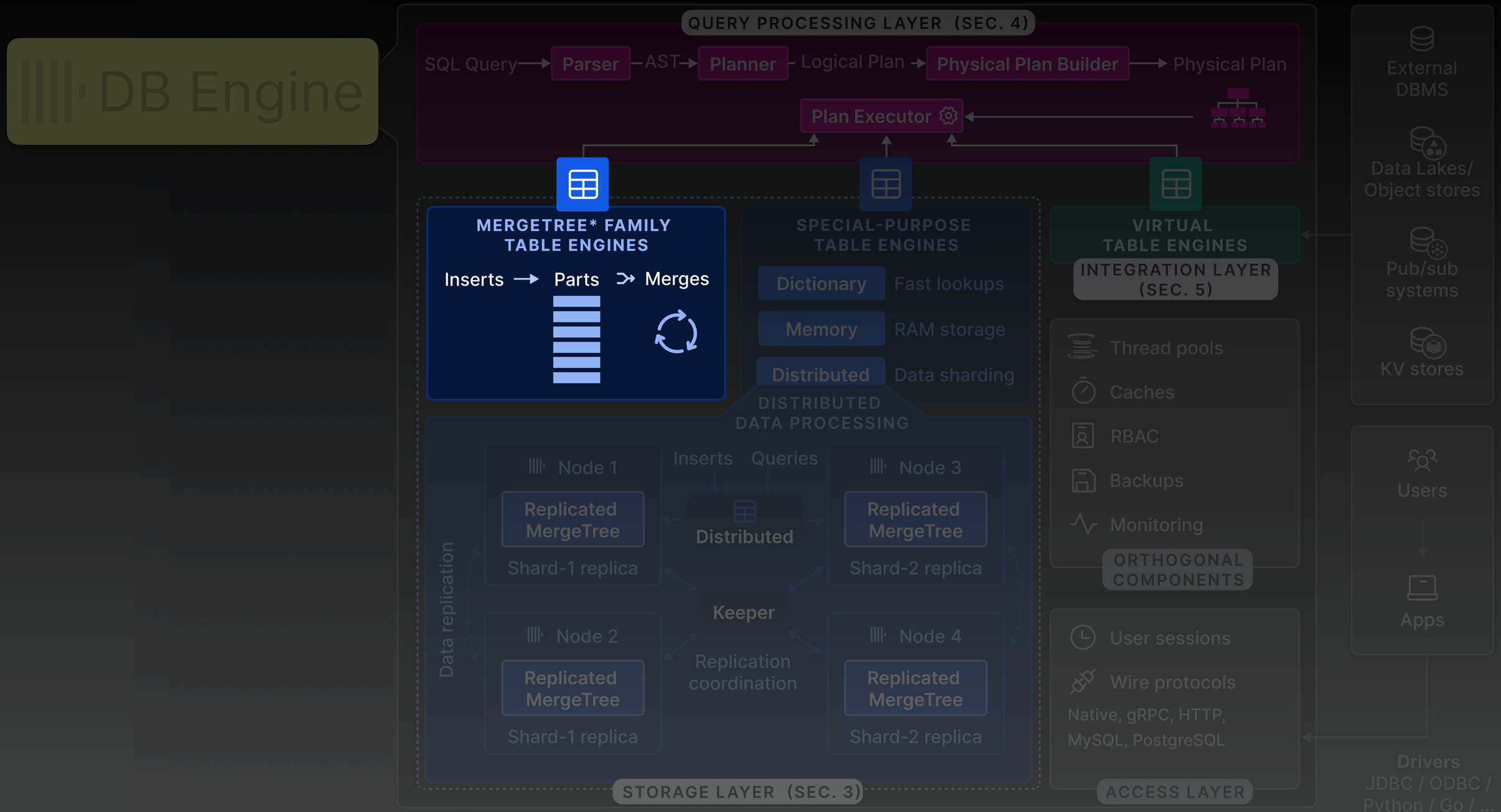


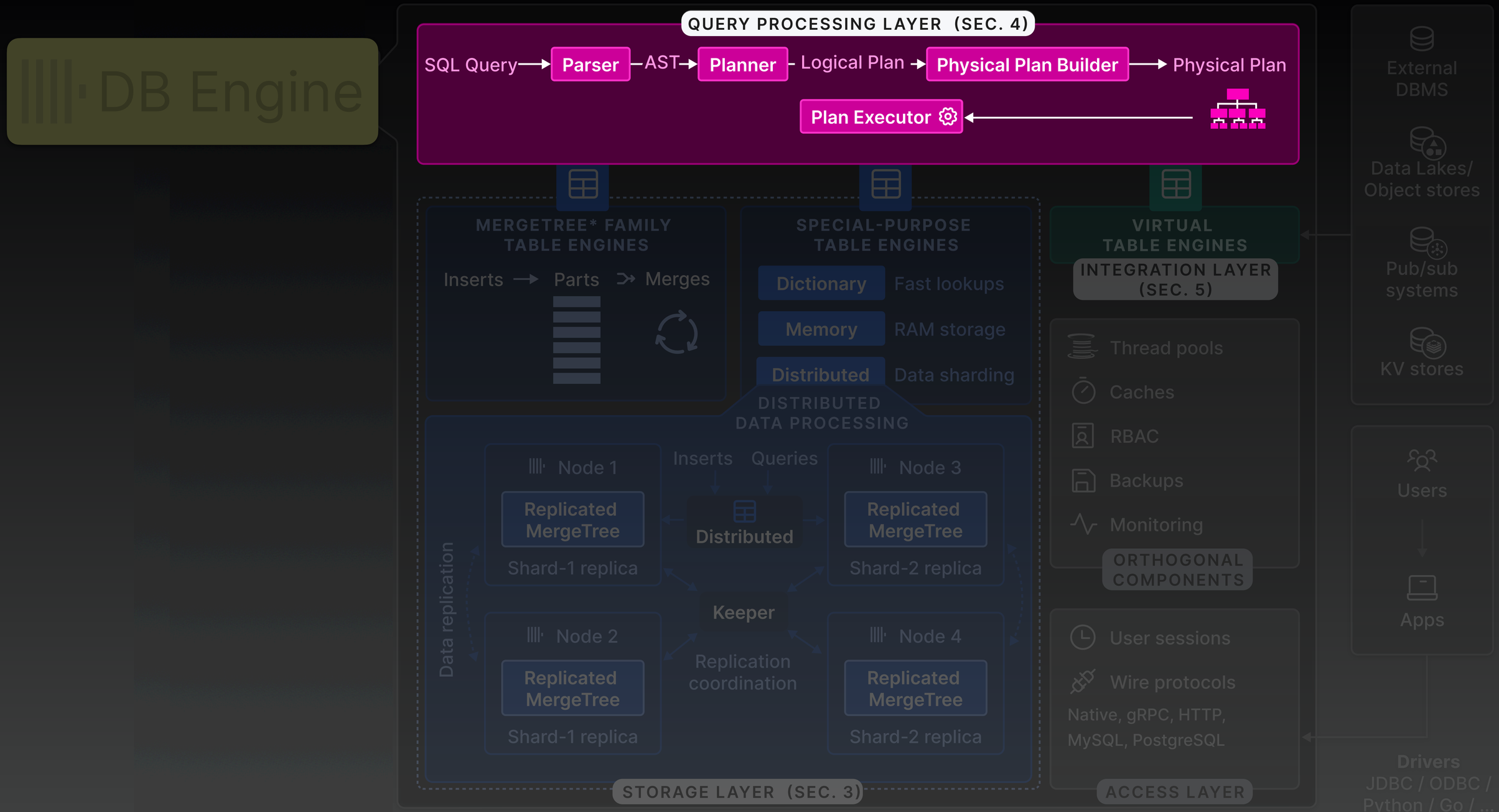
Some match → Load and Scan block

All match → SKIP load

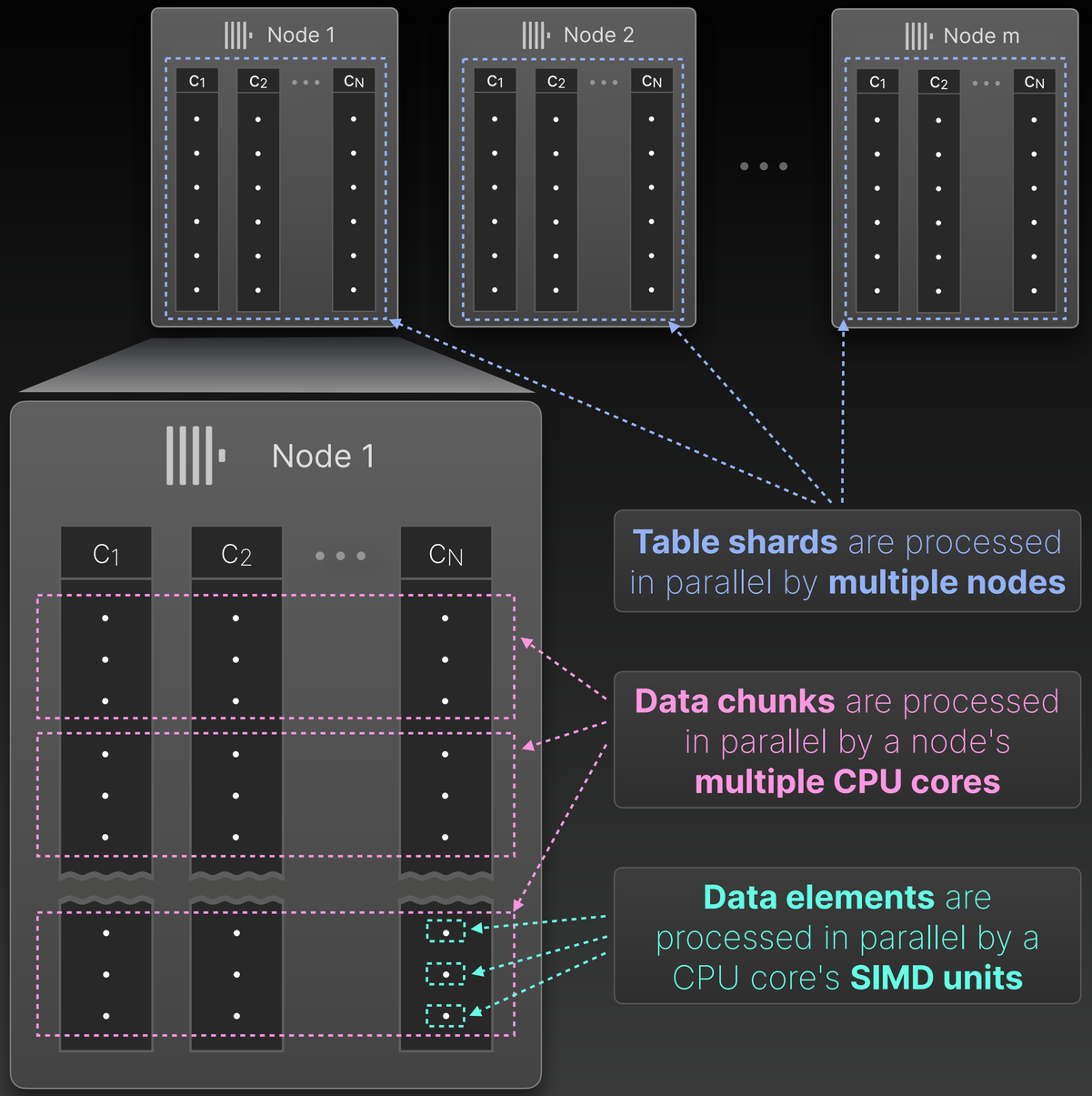
None match → SKIP load

Data pruning - Skipping indexes


DB Engine
QUERY PROCESSING LAYER (SEC. 4)
SQL Query
Parser
AST
Planner
Logical Plan
Physical Plan Builder
Physical Plan
Plan Executor
MERGETREE* FAMILY TABLE ENGINES
Inserts
Parts
Merges
SPECIAL-PURPOSE TABLE ENGINES
Dictionary
Fast lookups
Memory
RAM storage
Distributed
Data sharding
VIRTUAL TABLE ENGINES
INTEGRATION LAYER (SEC. 5)
DISTRIBUTED DATA PROCESSING
Node 1
Replicated MergeTree
Shard-1 replica
Inserts
Queries
Distributed
Node 3
Replicated MergeTree
Shard-2 replica
Keeper
Node 2
Replicated MergeTree
Shard-1 replica
Replication coordination
Node 4
Replicated MergeTree
Shard-2 replica
Data replication
STORAGE LAYER (SEC. 3)
Thread pools
Caches
RBAC
Backups
Monitoring
ORTHOGONAL COMPONENTS
User sessions
Wire protocols
Native, gRPC, HTTP, MySQL, PostgreSQL
ACCESS LAYER
External DBMS
Data Lakes/ Object stores
Pub/sub systems
KV stores
Users
Apps
Drivers
JDBC / ODBC / Python / Go / ...

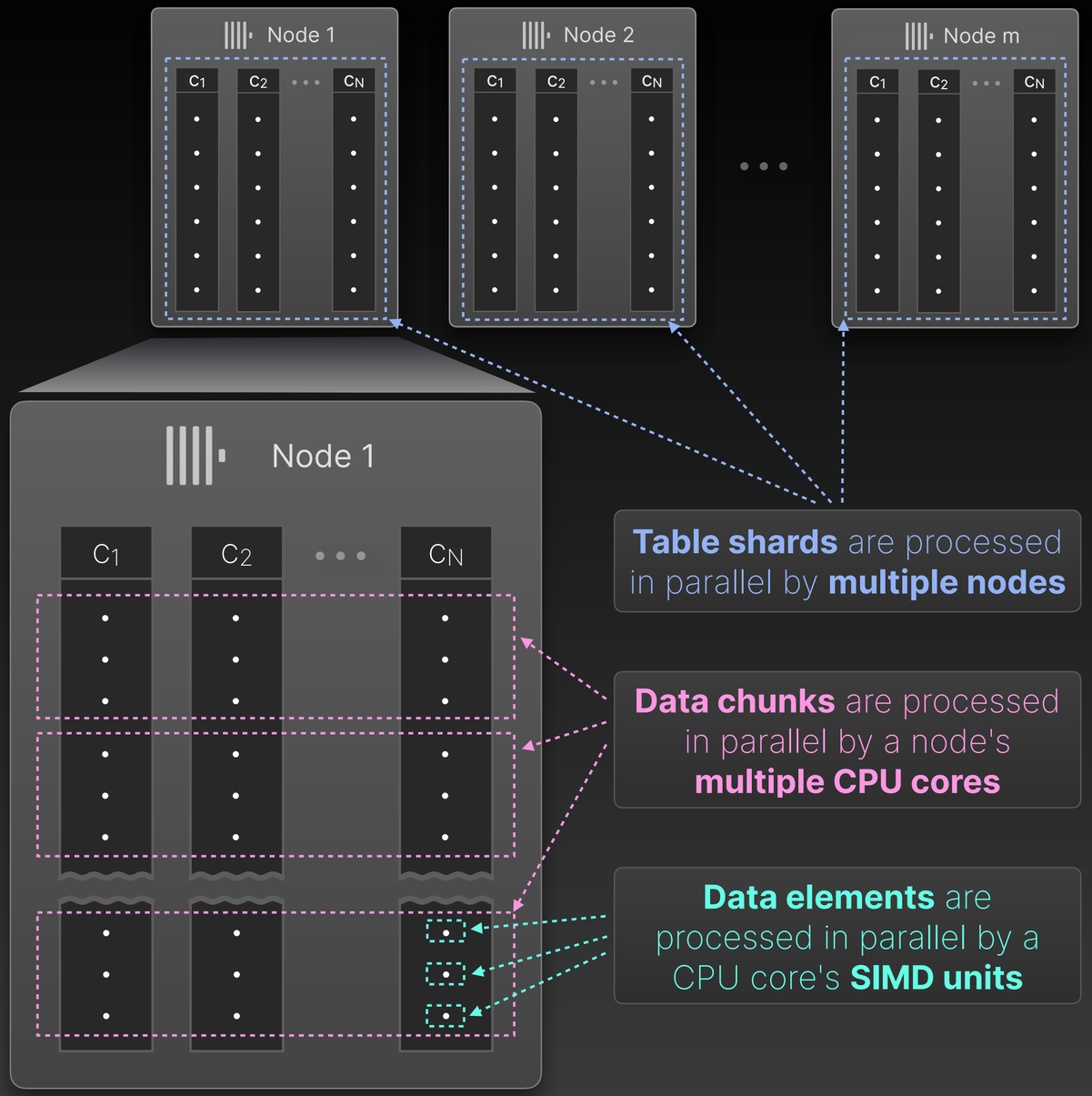

DB Engine
QUERY PROCESSING LAYER (SEC. 4)
SQL Query
Parser
AST
Planner
Logical Plan
Physical Plan Builder
Physical Plan
Plan Executor
MERGETREE* FAMILY TABLE ENGINES
Inserts
Parts
Merges
SPECIAL-PURPOSE TABLE ENGINES
Dictionary
Fast lookups
Memory
RAM storage
Distributed
Data sharding
VIRTUAL TABLE ENGINES
INTEGRATION LAYER (SEC. 5)
External DBMS
Data Lakes/ Object stores
Pub/sub systems
KV stores
DISTRIBUTED DATA PROCESSING
Node 1
Replicated MergeTree
Shard-1 replica
Inserts
Queries
Distributed
Node 3
Replicated MergeTree
Shard-2 replica
Keeper
Data replication
Node 2
Replicated MergeTree
Shard-1 replica
Replication coordination
Node 4
Replicated MergeTree
Shard-2 replica
Thread pools
Caches
RBAC
Backups
Monitoring
ORTHOGONAL COMPONENTS
User sessions
Wire protocols
Native, gRPC, HTTP, MySQL, PostgreSQL
Users
Apps
Drivers
JDBC / ODBC / Python / Go / ...
STORAGE LAYER (SEC. 3)
ACCESS LAYER

Plan Executor



State-of-the-art Vectorized Query Execution Engine

Plan Executor



State-of-the-art Vectorized Query Execution Engine

Data elements are processed in parallel by a CPU core's **SIMD units**

Data chunks are processed in parallel by a node's **multiple CPU cores**

Table shards are processed in parallel by **multiple nodes**

Data elements are processed in parallel by a CPU core's **SIMD units**

Data chunks are processed in parallel by a node's **multiple CPU cores**

Table shards are processed in parallel by **m**

- Based on compiler auto-vectorization or manually written intrinsics.
- SQL expressions are compiled into *compute kernels*.

```
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

- The fastest kernel is selected at runtime based on the system capabilities (`cpuid`).

Parallelization Across SIMD ALUs

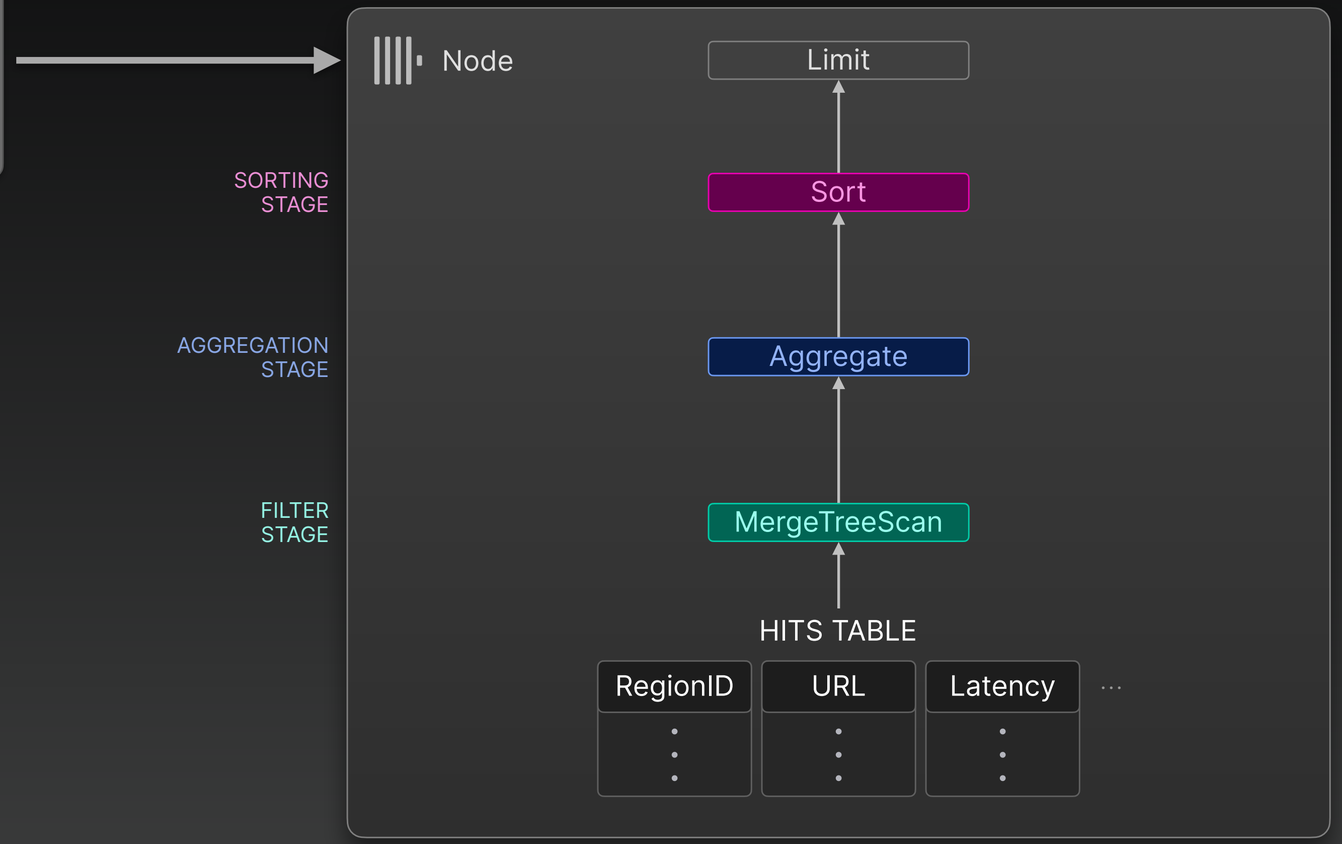
Data elements are processed in parallel by a CPU core's **SIMD units**

Data chunks are processed in parallel by a node's **multiple CPU cores**

Table shards are processed in parallel by **multiple nodes**

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

FILTER
AGGREGATE
SORT



Parallelization Across CPU Cores

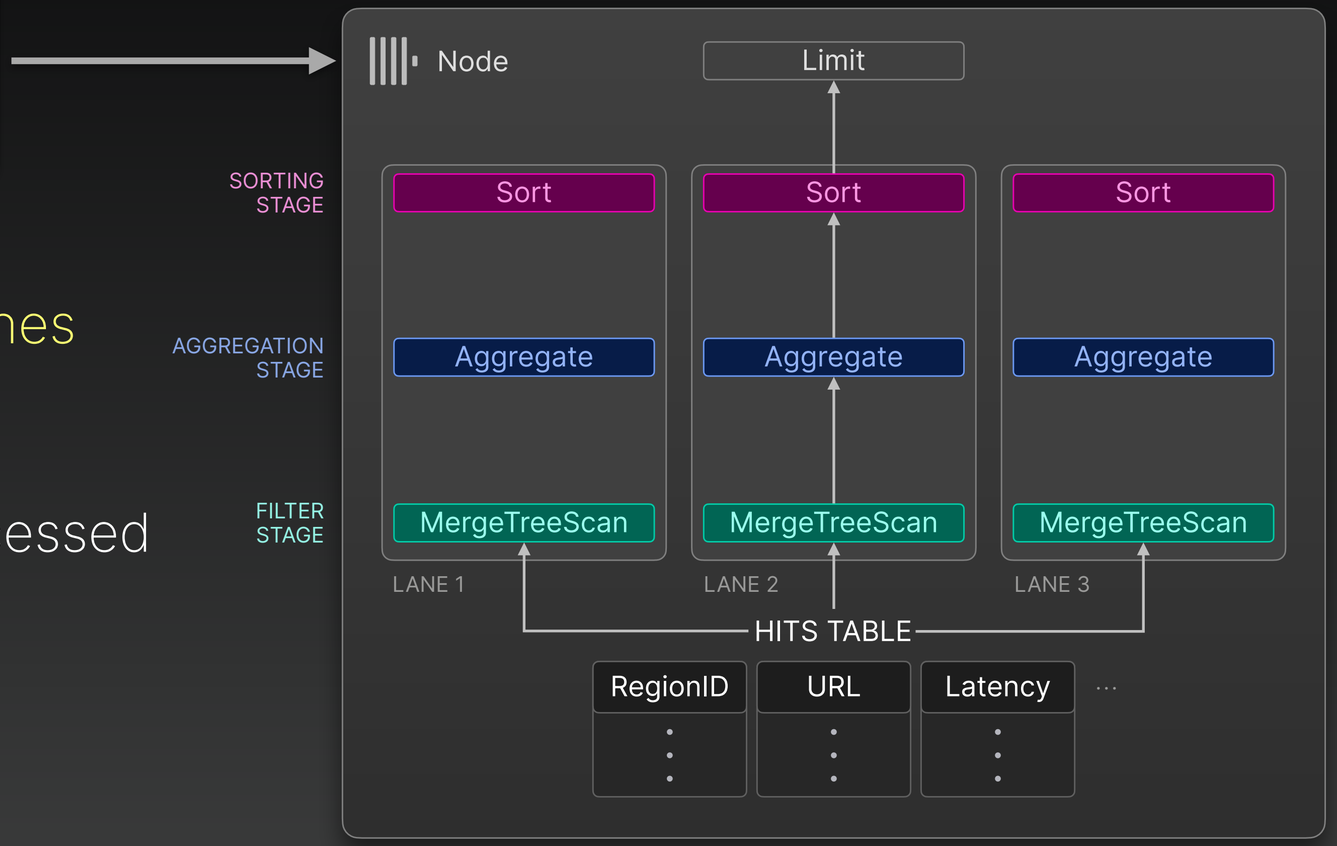
Data elements are processed in parallel by a CPU core's **SIMD units**

Data chunks are processed in parallel by a node's **multiple CPU cores**

Table shards are processed in parallel by **multiple nodes**

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

FILTER
AGGREGATE
SORT



- Execution plan gets unfolded into N lanes (typically 1 lane per CPU core).
- Lanes decompose the data to be processed into non-overlapping ranges.

Parallelization Across CPU Cores

Data elements are processed in parallel by a CPU core's **SIMD units**

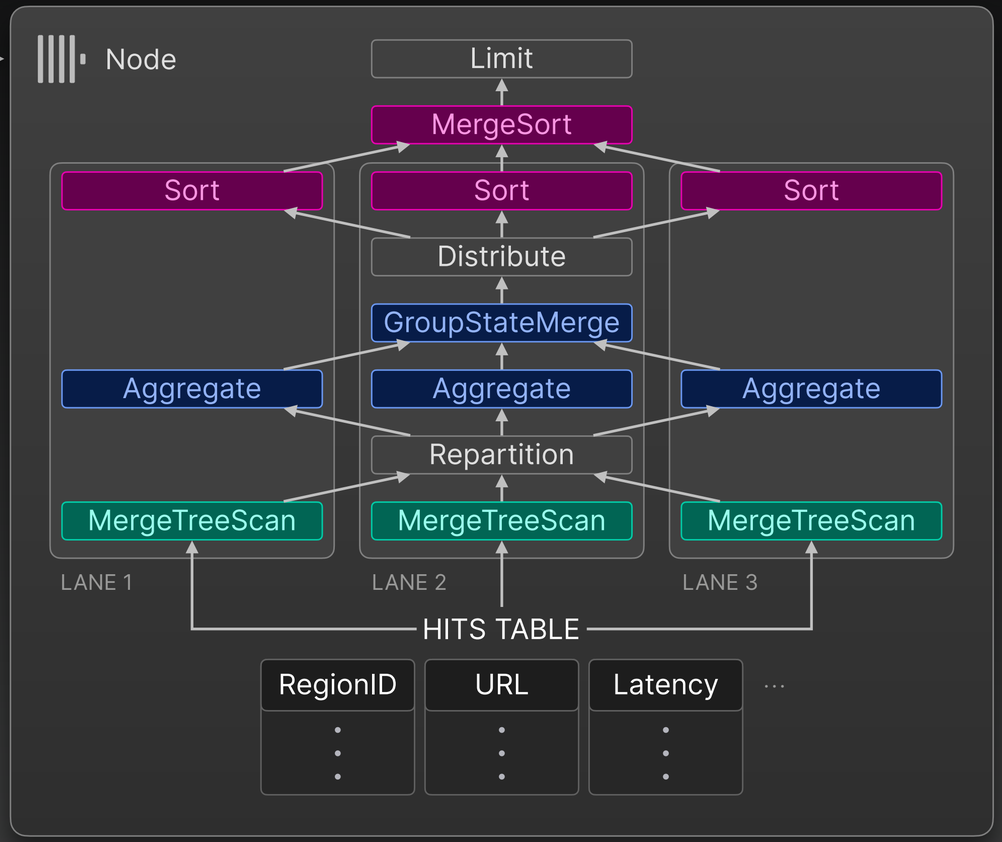
Data chunks are processed in parallel by a node's **multiple CPU cores**

Table shards are processed in parallel by **multiple nodes**

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

FILTER
AGGREGATE
SORT

- Exchange operators (Repartition, Distribute) ensure lanes remain balanced.



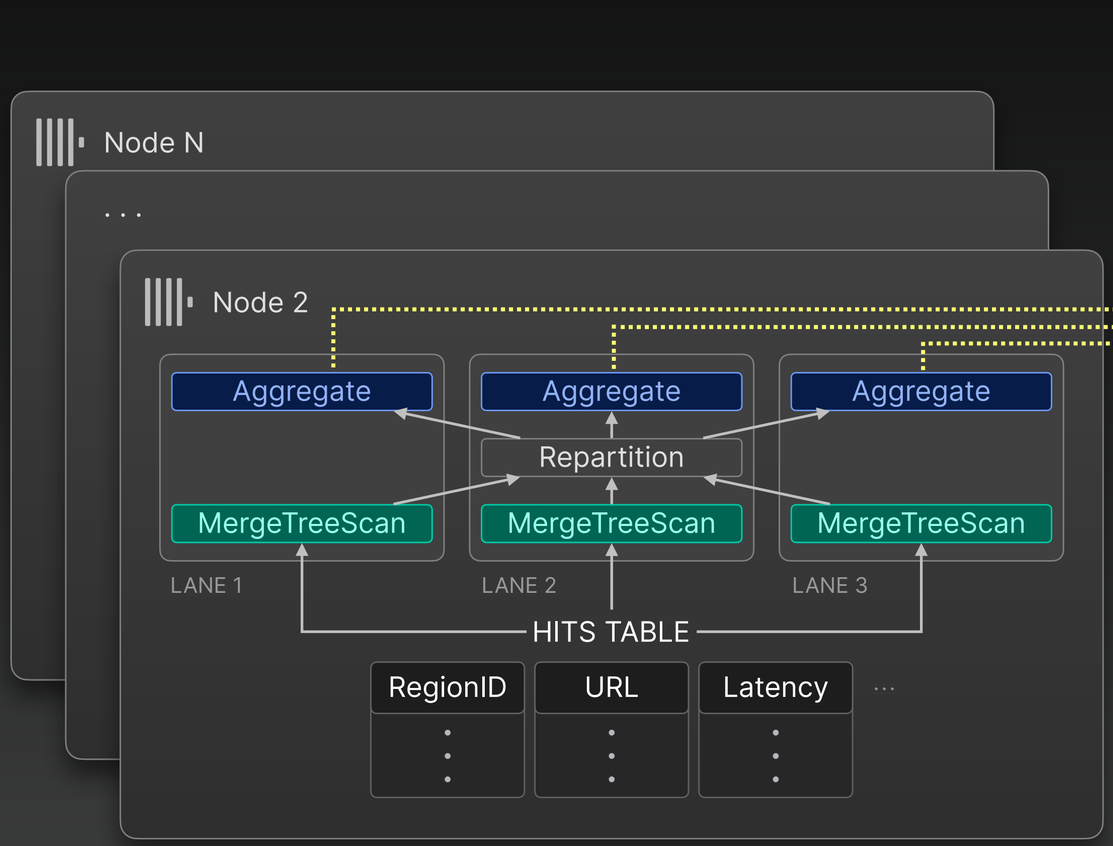
Parallelization Across CPU Cores

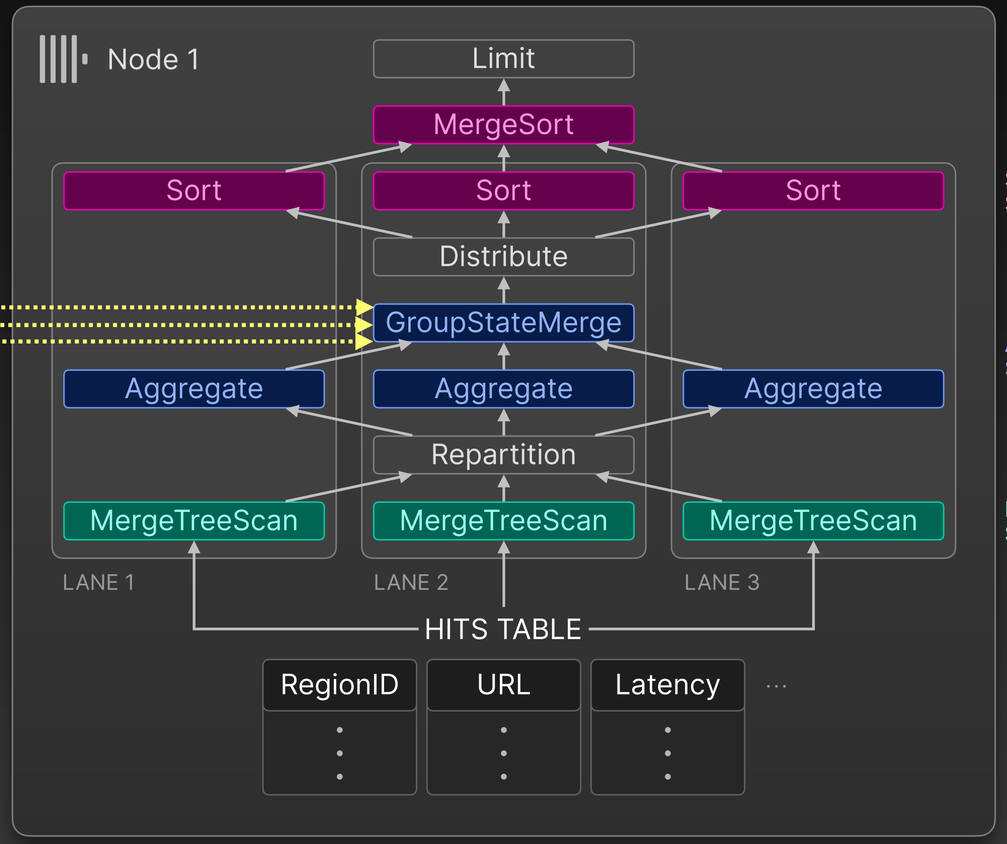
Data elements are processed in parallel by a CPU core's **SIMD units**

Data chunks are processed in parallel by a node's **multiple CPU cores**

Table shards are processed in parallel by **multiple 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.





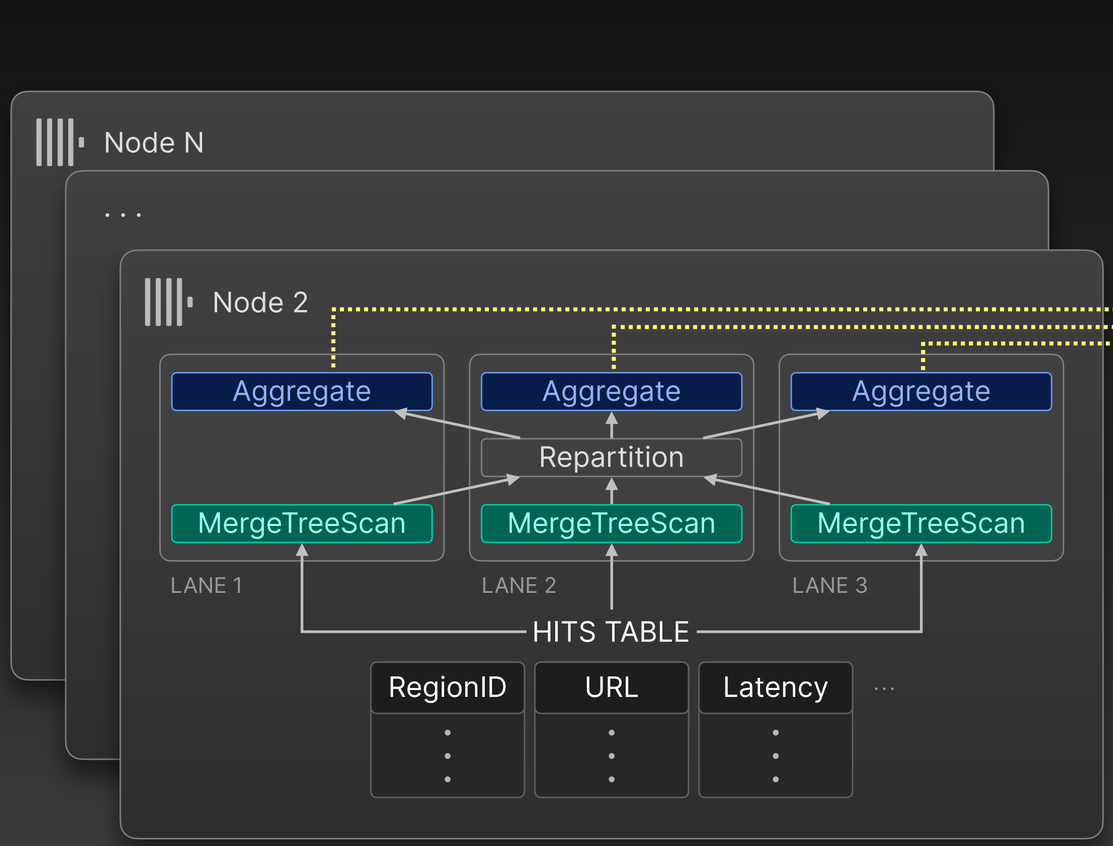
Parallelization Across Cluster Nodes

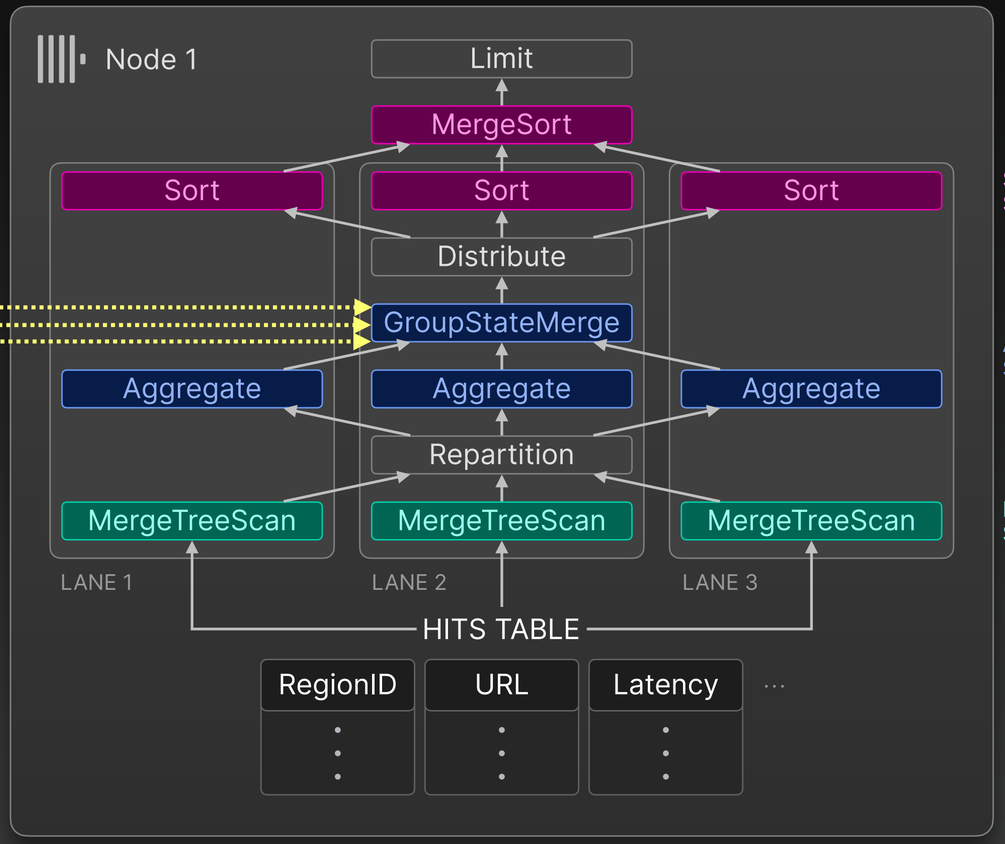
Data elements are processed in parallel by a CPU core's **SIMD units**

Data chunks are processed in parallel by a node's **multiple CPU cores**

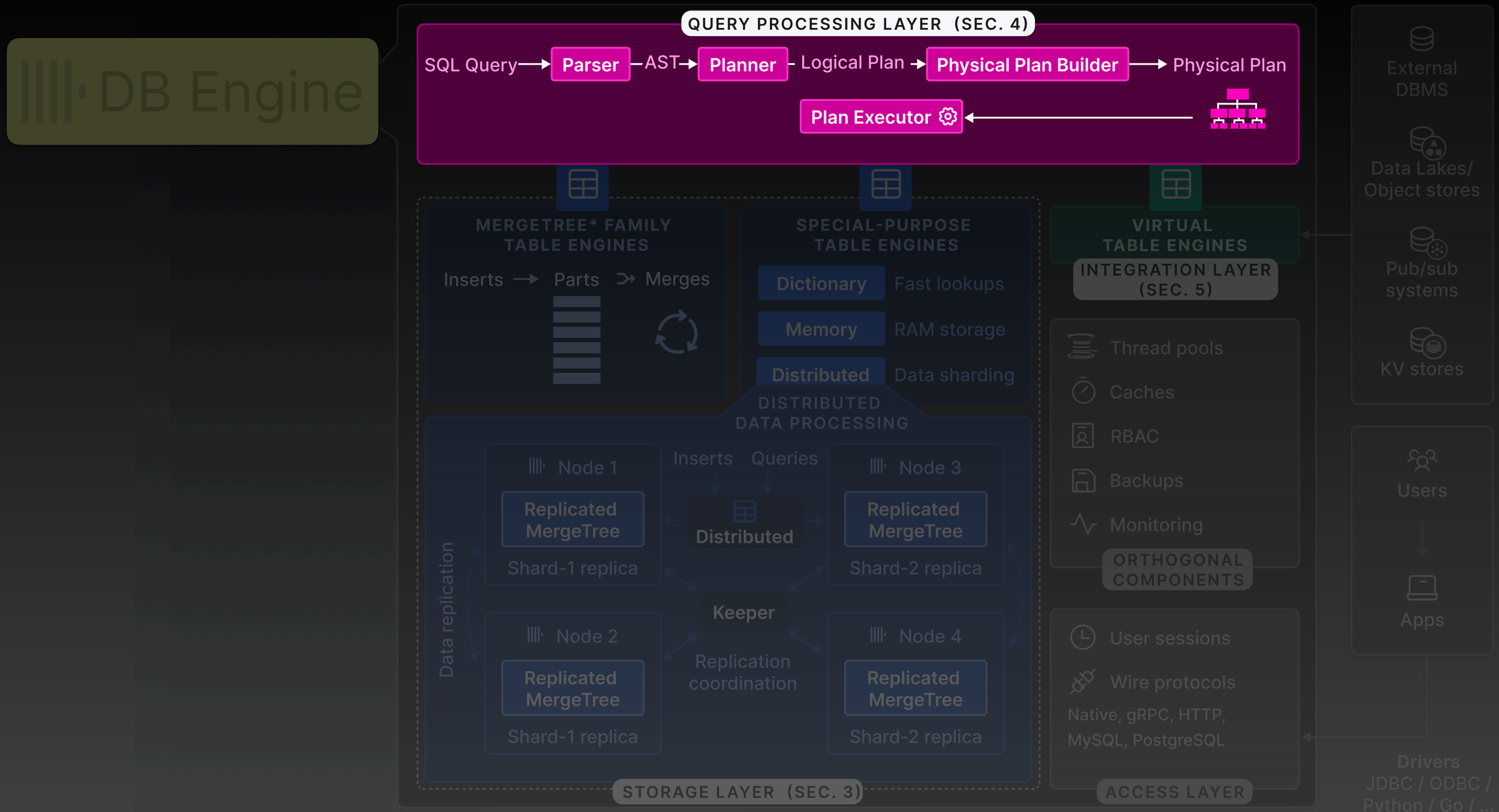
Table shards are processed in parallel by **multiple nodes**

- 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.

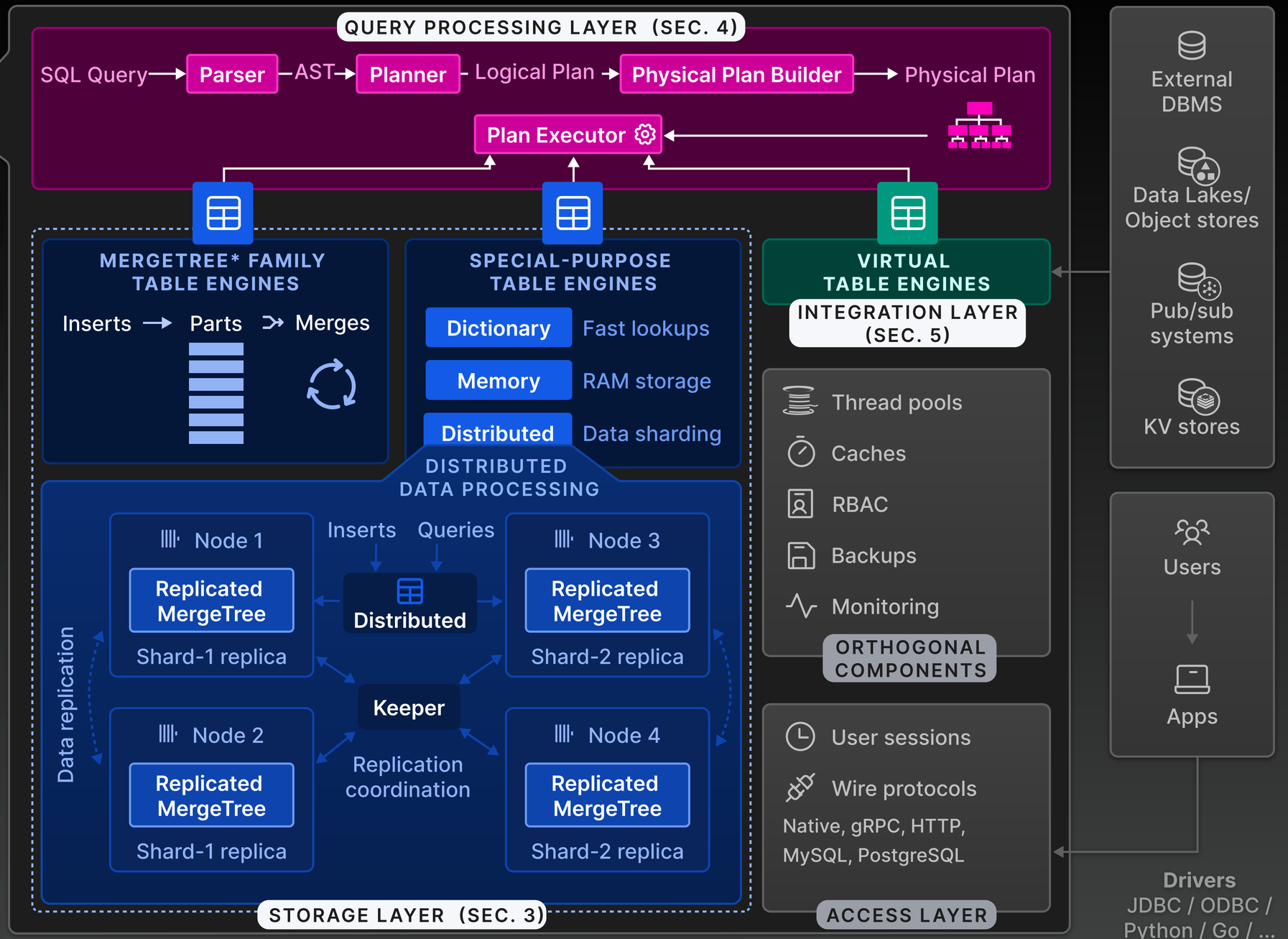




Parallelization Across Cluster Nodes


DB Engine
QUERY PROCESSING LAYER (SEC. 4)
SQL Query
Parser
AST
Planner
Logical Plan
Physical Plan Builder
Physical Plan
Plan Executor
MERGETREE* FAMILY TABLE ENGINES
Inserts
Parts
Merges
SPECIAL-PURPOSE TABLE ENGINES
Dictionary
Fast lookups
Memory
RAM storage
Distributed
Data sharding
VIRTUAL TABLE ENGINES
INTEGRATION LAYER (SEC. 5)
External DBMS
Data Lakes/ Object stores
Pub/sub systems
KV stores
DISTRIBUTED DATA PROCESSING
Node 1
Replicated MergeTree
Shard-1 replica
Inserts
Queries
Distributed
Node 3
Replicated MergeTree
Shard-2 replica
Keeper
Replication coordination
Node 2
Replicated MergeTree
Shard-1 replica
Node 4
Replicated MergeTree
Shard-2 replica
Data replication
Thread pools
Caches
RBAC
Backups
Monitoring
ORTHOGONAL COMPONENTS
User sessions
Wire protocols
Native, gRPC, HTTP, MySQL, PostgreSQL
Users
Apps
Drivers
JDBC / ODBC / Python / Go / ...
STORAGE LAYER (SEC. 3)
ACCESS LAYER

DB Engine


QUERY PROCESSING LAYER (SEC. 4)
SQL Query
Parser
AST
Planner
Logical Plan
Physical Plan Builder
Physical Plan
Plan Executor
MERGETREE* FAMILY TABLE ENGINES
Inserts
Parts
Merges
SPECIAL-PURPOSE TABLE ENGINES
Dictionary
Fast lookups
Memory
RAM storage
Distributed
Data sharding
VIRTUAL TABLE ENGINES
INTEGRATION LAYER (SEC. 5)
DISTRIBUTED DATA PROCESSING
Node 1
Replicated MergeTree
Shard-1 replica
Inserts
Queries
Distributed
Node 3
Replicated MergeTree
Shard-2 replica
Data replication
Keeper
Replication coordination
Node 2
Replicated MergeTree
Shard-1 replica
Node 4
Replicated MergeTree
Shard-2 replica
STORAGE LAYER (SEC. 3)
Thread pools
Caches
RBAC
Backups
Monitoring
ORTHOGONAL COMPONENTS
User sessions
Wire protocols
Native, gRPC, HTTP, MySQL, PostgreSQL
ACCESS LAYER
External DBMS
Data Lakes/ Object stores
Pub/sub systems
KV stores
Users
Apps
Drivers
JDBC / ODBC / Python / Go / ...