

# ClickHouse

-

## A Superfast Database for Analytics

7 May, 2025  
Robert Schulze



# Agenda

- Background
- Storage Layer
- Query Layer

## 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 merge (LSM) trees with novel techniques for continuous transformation (e.g. aggregation, archiving) of historical data in the background. Queries are written in a convenient SQL dialect and processed by a state-of-the-art vectorized query execution engine with optional code compilation. ClickHouse makes aggressive use of pruning techniques to avoid evaluating irrelevant data in queries. Other data management systems can be integrated at the table function, table engine, or database engine level. Real-world benchmarks demonstrate that ClickHouse is amongst the fastest analytical databases on the market.

### PVLDB Reference Format:

Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov: ClickHouse - Lightning Fast Analytics for Everyone. PVLDB, 17(12): 3731 - 3744, 2024.  
doi:10.14778/3685800.3685802

### 1 INTRODUCTION

This paper describes ClickHouse, a columnar OLAP database designed for high-performance analytical queries on tables with trillions of rows and hundreds of columns. ClickHouse was started in 2009 as a filter and aggregation operator for web-scale log file data<sup>1</sup> and was open sourced in 2016. Figure 1 illustrates when major features described in this paper were introduced to ClickHouse.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing [info@pvlb.org](mailto:info@pvlb.org). Copyright is held by the owner(s). Publication rights licensed to the VLDB Endowment. Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-4097. doi:10.14778/3685800.3685802

<sup>1</sup>Blog post: [clickhouse.com/en/evolution](https://clickhouse.com/en/evolution)

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.

2. **Many simultaneous queries with an expectation of low latencies.** Queries can generally be categorized as ad-hoc (e.g. exploratory data analysis) or recurring (e.g. periodic dashboard queries). The more interactive a use case is, the lower query latencies are expected, leading to challenges in query optimization and execution. Recurring queries additionally provide an opportunity to adapt the physical database layout to the workload. As a result, databases should offer pruning techniques that allow optimizing frequent queries. Depending on the query priority, databases must further grant equal or prioritized access to shared system resources such as CPU, memory, disk and network I/O, even if a large number of queries run simultaneously.

3. **Diverse landscapes of data stores, storage locations, and formats.** To integrate with existing data architectures, modern analytical databases should exhibit a high degree of openness to read and write external data in any system, location, or format.

4. **A convenient query language with support for performance introspection.** Real-world usage of OLAP databases poses additional "soft" requirements. For example, instead of a niche programming language, users often prefer to interface with databases in an expressive SQL dialect with nested data types and a broad range of regular, aggregation, and window functions. Analytical databases should also provide sophisticated tooling to introspect the performance of the system or individual queries.

5. **Industry-grade robustness and versatile deployment.** As commodity hardware is unreliable, databases must provide data replication for robustness against node failures. Also, databases should run on any hardware, from old laptops to powerful servers. Finally, to avoid the overhead of garbage collection in JVM-based programs and enable bare-metal performance (e.g. SMDI), databases are ideally deployed as native binaries for the target platform.

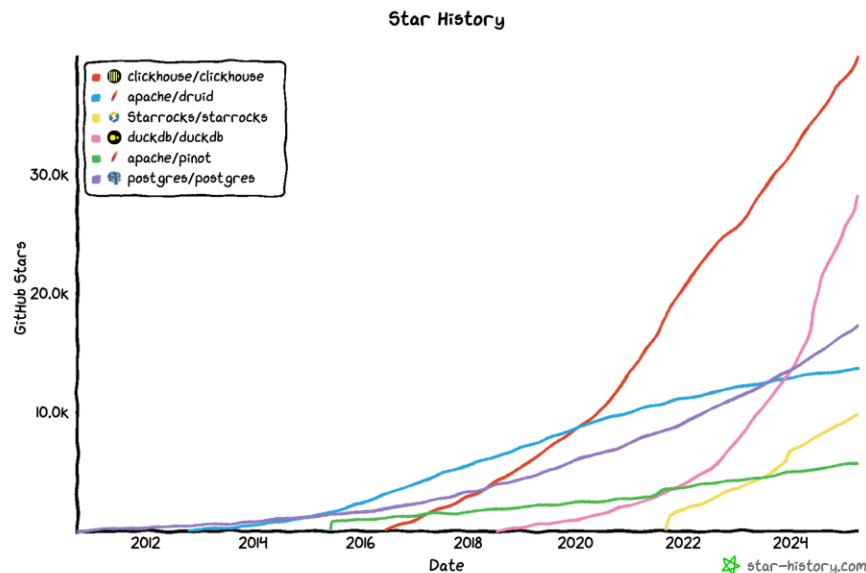
# Who Am I?

- Robert Schulze
- Engineering Manager @ ClickHouse Inc., previously at SAP and Dresden University of Technology
- Supervising student theses (BSc, MSc) and interns ([link](#))

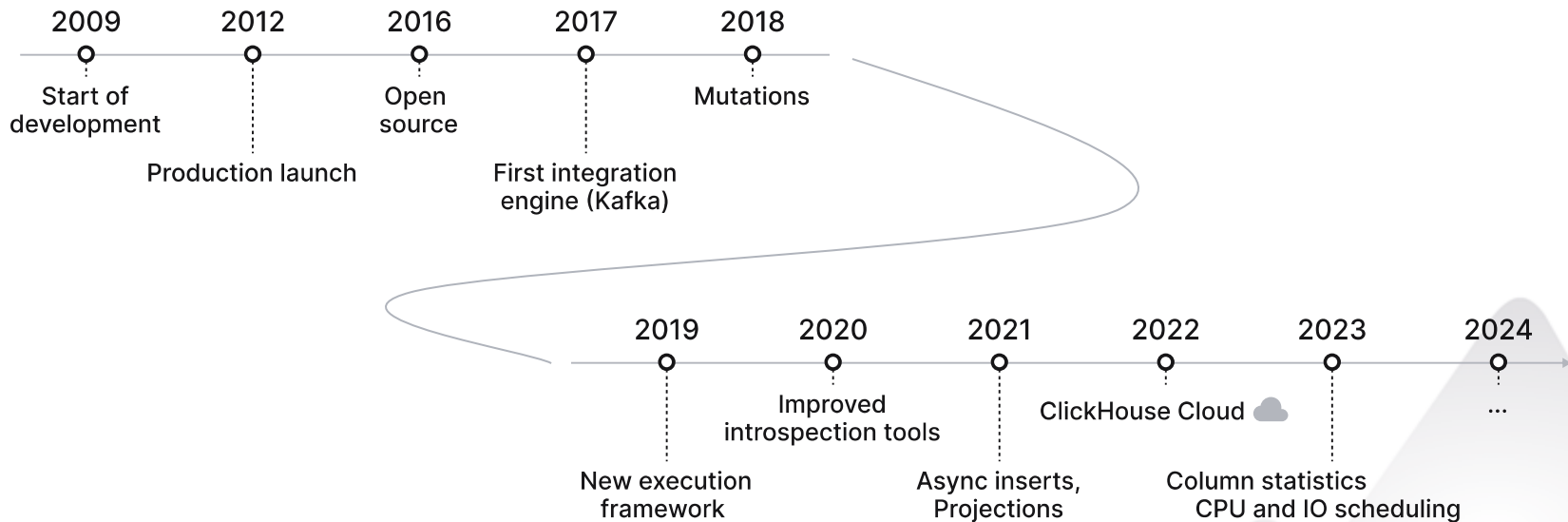


# What is ClickHouse?

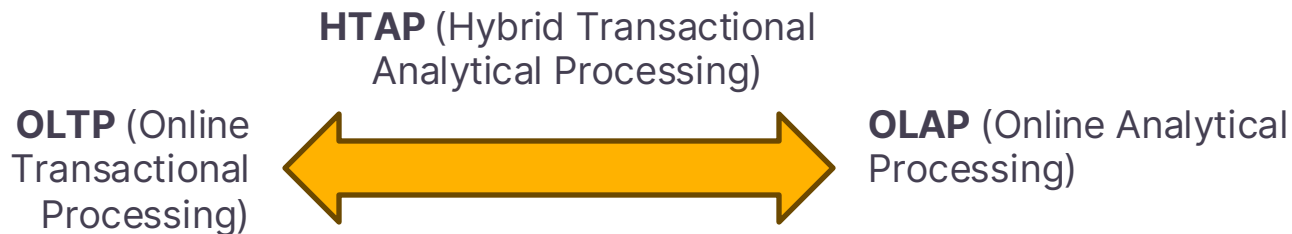
- An analytical (workload), relational (data model), columnar (data organization), shared-nothing/shared-disk (architecture, on-premises/Cloud) database with eventual consistency (consistency model).
- Goal: super-fast 🚀 and scalable analytics over tables with trillions of rows and hundreds of columns.
- Open Source (Apache 2.0, except some Cloud bits), built in C++, runs on anything from Raspberry Pi to clusters with 100s of nodes.
- Self-managed (on-premises) or ClickHouse Cloud, a database-as-a-service (DBaaS)





# ClickHouse History



# What is an Analytics Database?



<b>Workload</b>	Short-running point queries	Long-running batch queries
<b>Data Volumes</b>	Moderate (GB – TB)	Huge (TB – PB)
<b>Emphasis</b>	Data integrity and correctness	Scalability and performance
<b>Data Organization</b>	Row-wise	Column-wise
<b>Use Cases</b>	Enterprise Resource Planning (ERP)	“Big data” and decision making, e.g. dashboards and ad-hoc data exploration
<b>Examples</b>		

# Row-wise vs. Column-wise Data Organisation

Country	Product	Sales
GB	Lambda	350
FR	Kappa	400
US	Iota	1300



**Row-wise**  
(n-ary storage model, NSM)

Row 1	GB
	Lambda
	350
Row 3	FR
	Kappa
	400
Row 4	US
	Iota
	1300

**Columnar**  
(decompositional storage model, DSM)

Column 1	GB
	FR
	US
Column 2	Lambda
	Kappa
	Iota
Column 3	350
	400
	1300



**Hybrid (PAX)**

[A. Ailamaki et. al.: Weaving Relations for Cache Performance. 2001]

**Compressability**

low

high

**Tuple reconstruction cost**

low

high

**Single-row operations**  
(point lookups)

fast

slow

**Column operations**  
(scans, aggregation)

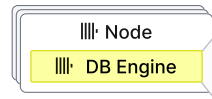
slow

fast

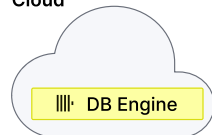


# System Architecture

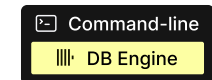
Server on-premise



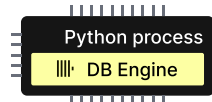
Cloud



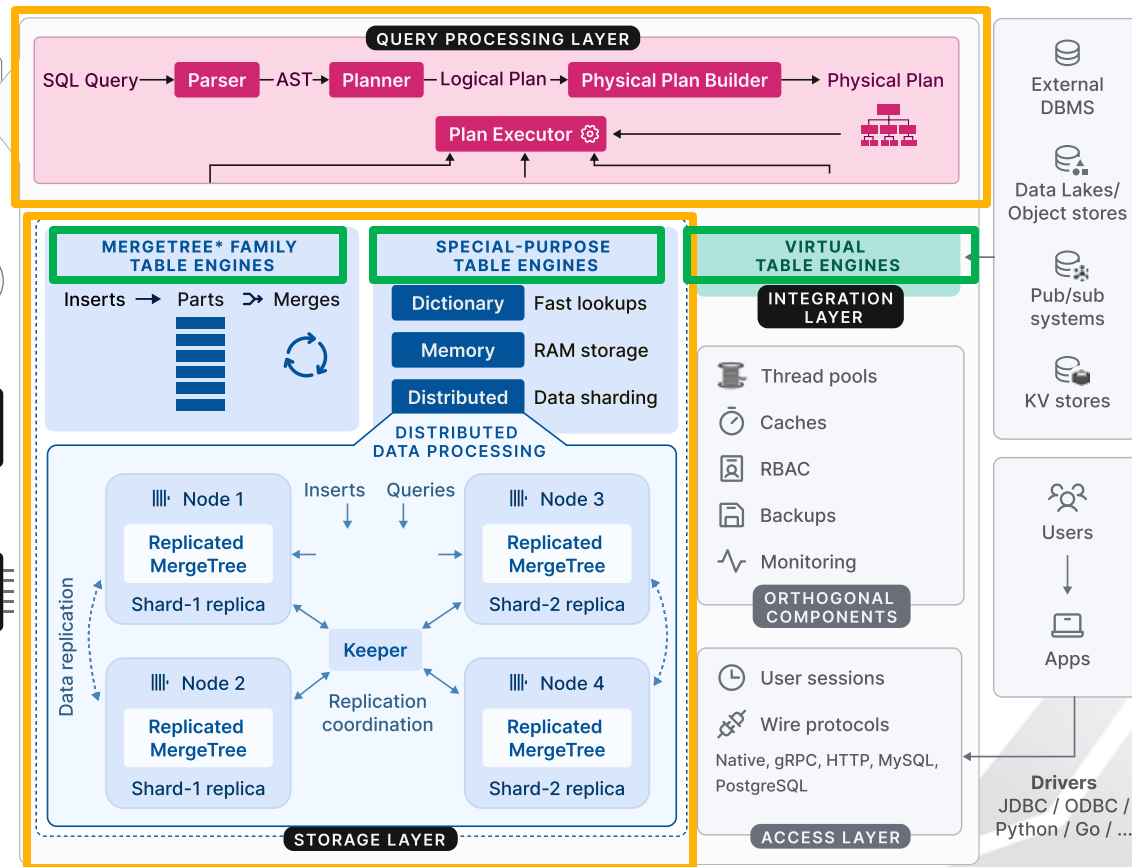
Standalone



In-process



Execution  
Modes



**Table Engine:**  
Location and format  
of table data



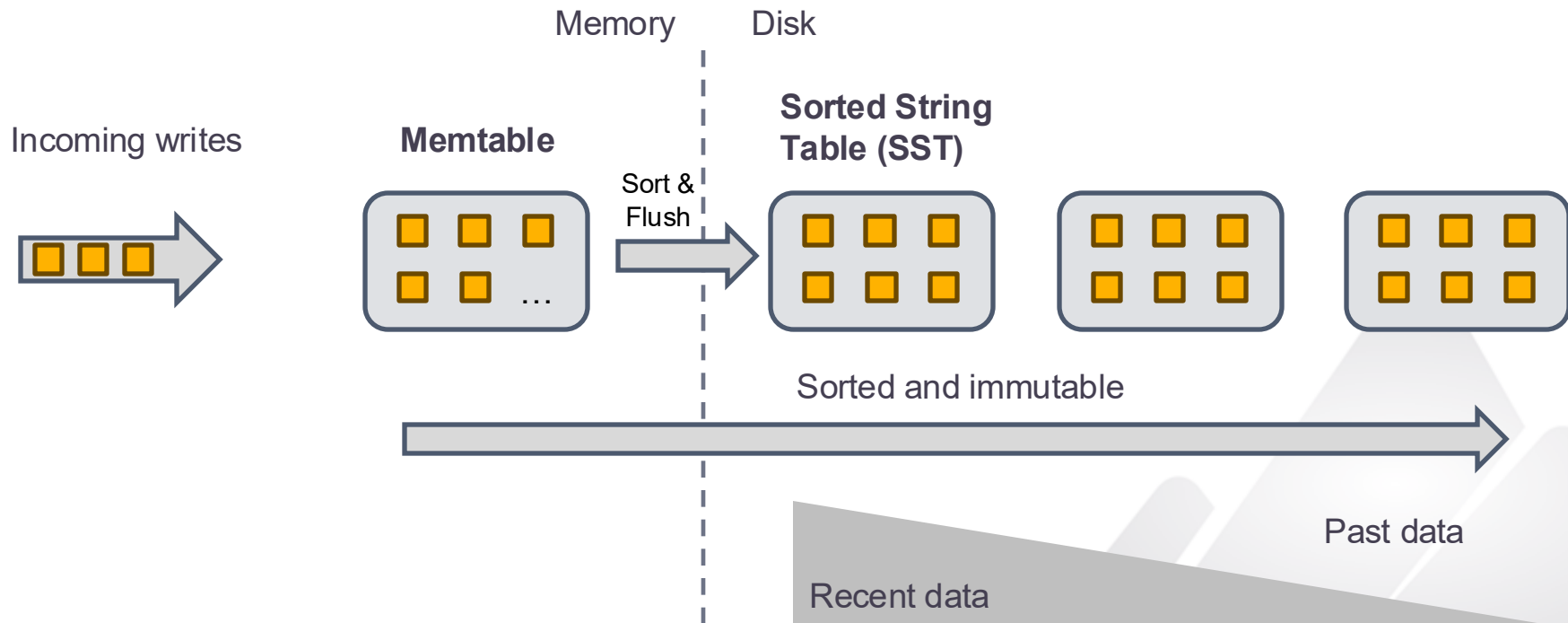
# Storage Layer



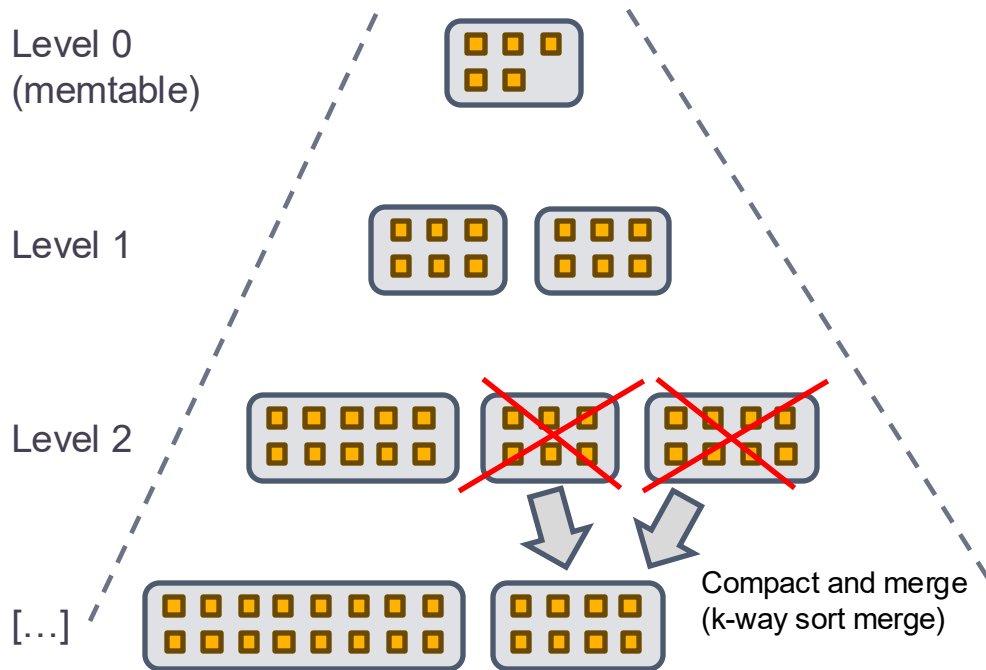
# Log-Structured Merge (LSM) Trees

[F. Chang et. al.: Bigtable: A Distributed Storage System for Structured Data, 2006]

[P. O'Neil et. al.: The Log-Structured Merge-Tree (LSM-Tree), 1996]



# LSM Trees: Compaction



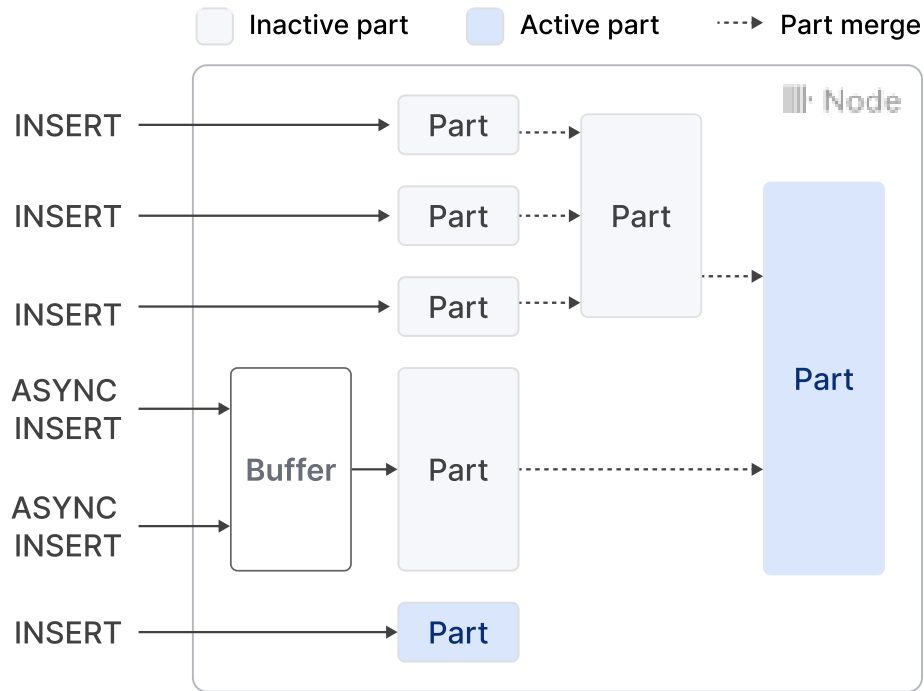
## Compaction Strategies

- When? level saturation, file size, file age, file “temperature” ...
- What? individual SSTs, entire levels, ...
- ...

[S. Sakar et. al.:  
Constructing and Analyzing  
the LSM Compaction  
Design Space, 2022]



# LSM-Tree-Based Storage in ClickHouse



- INSERTs create a sorted and immutable *part* (aka. SST).
- INSERTs are synchronous or asynchronous.
- All parts are equal, no levels or notion of recency.
- Periodic merges, the source parts are deleted once their reference count drops to 0.



# Example Part (1/2)

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

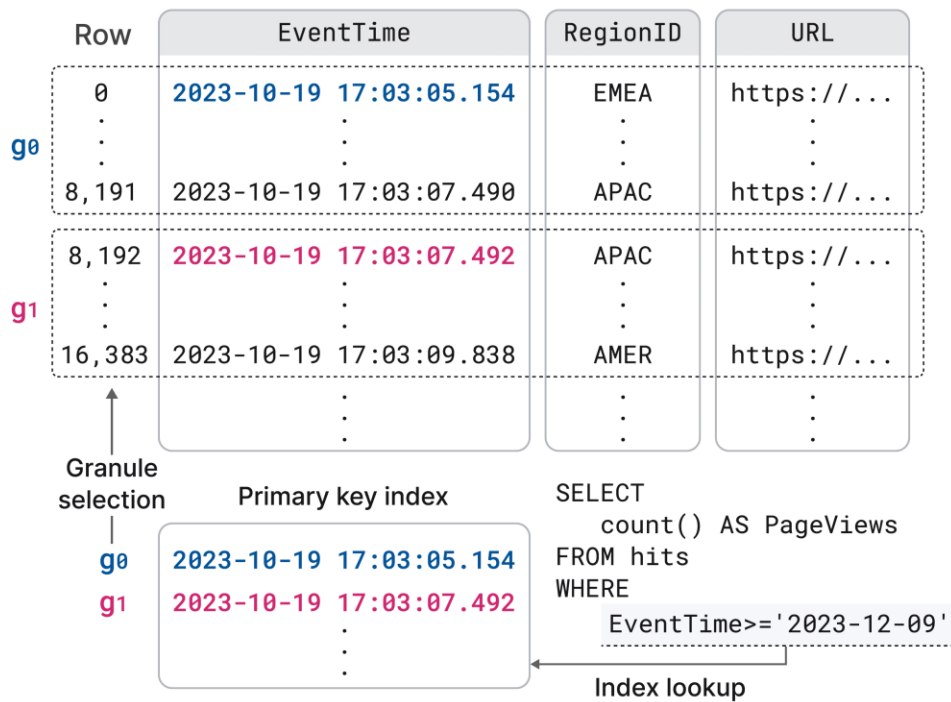
- Local (per-part) sorting defined by primary key:

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

- Part are further divided into *granules* *g0*, *g1*, ...
- Consecutive granules in a column form *blocks* which are compressed:
  - generic bit codecs: LZ4, zstd, ...
  - logical codecs: delta, ...
  - specialised codecs: Gorilla (FP), AES, ...



## Example Part (2/2)



- Primary key defines sorting AND a lightweight index structure.
- Maps primary key index values to granules.
- Small enough to reside in DRAM.
- Used to accelerate predicate evaluation on primary key columns.



# Data Pruning (1/3)

Analytical databases deal with tables sizes of many petabytes.

The fastest scan is not scanning at all!

**Primary Key: Sorting and light-weight index**



# Data Pruning (2/3)

```
ALTER TABLE hits ADD PROJECTION proj(  
    SELECT * ORDER BY RegionID  
);  
  
ALTER TABLE hits MATERIALIZE PROJECTION pj;
```



## Table Projections

- Alternative table versions sorted by a different primary key

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	APAC	https://...
2023-10-19 17:03:07.550	AMER	https://...

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://...

- Speed up queries on columns different than primary key columns.
- Work at the granularity of parts. Parts may or may not have projections.
- High space consumption and insert/merge overhead.



[M. Stonebraker et. al.: C-Store: A  
Column-oriented DBMS, 2005]





# Data Pruning (3/3)

[G. Moerkotte: Small  
Materialized Aggregates:  
A Light Weight Index  
Structure for Data  
Warehousing, 1998]



```
SELECT *  
FROM tab  
WHERE clicks BETWEEN 15 AND 30;
```

## Skipping indexes

- 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:
  - Minimum/maximum value - great for loosely sorted data.
  - Unique values - great for small cardinality.
  - Bloom filter for row / tokens / n-grams).

clicks	min/max index
25	
8	min: 7
7	max: 25
25	
25	
18	
20	min: 17
22	max: 22
19	
17	
8	
6	min: 5
6	max: 13
13	
5	

Some match → load  
and check block ☹️

All match -->  
skip scan 😊

None match →  
skip scan 😊



# Merge-Time Data Transformation

Merges (optionally) apply additional transformations.

## Replacing merges

Check for rows with same primary key values. Keeps the more recent row.

## TTL (time-to-live) merges

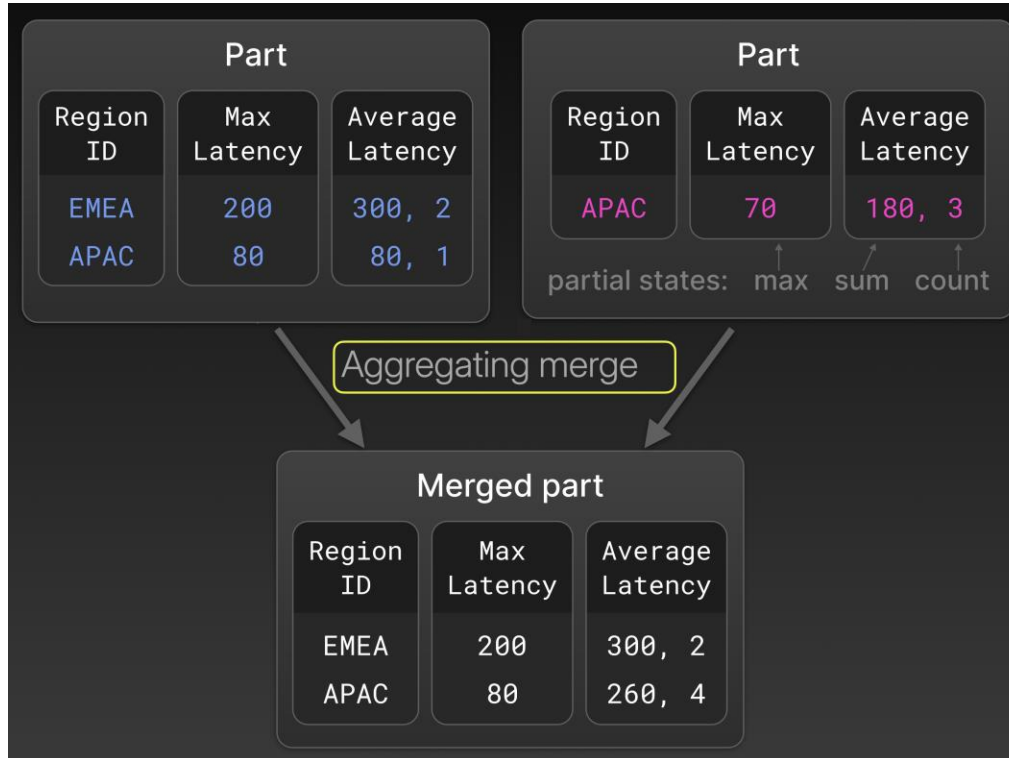
Compress, move, or, delete rows or parts based on the data age.

## Aggregating merges

Combine aggregation states into new aggregation states.



# Merge-Time Data Transformation



# Data Replication (1/3)

**Data Replication** means to store the same data redundantly across nodes. This enables *high availability* (tolerance against node failures), *load balancing* (scale-out), and *zero-downtime upgrades*.

Based on (abstract) concept of **table state**: Table are a set of table parts + table metadata.

		How is the table state advanced?		
		Add Part	Delete Part	Change Metadata
Operation	Inserts	✓		
	Merges	✓	✓	
	DDL Statements	✓	✓	✓

} Recorded in  
Global Replication Log

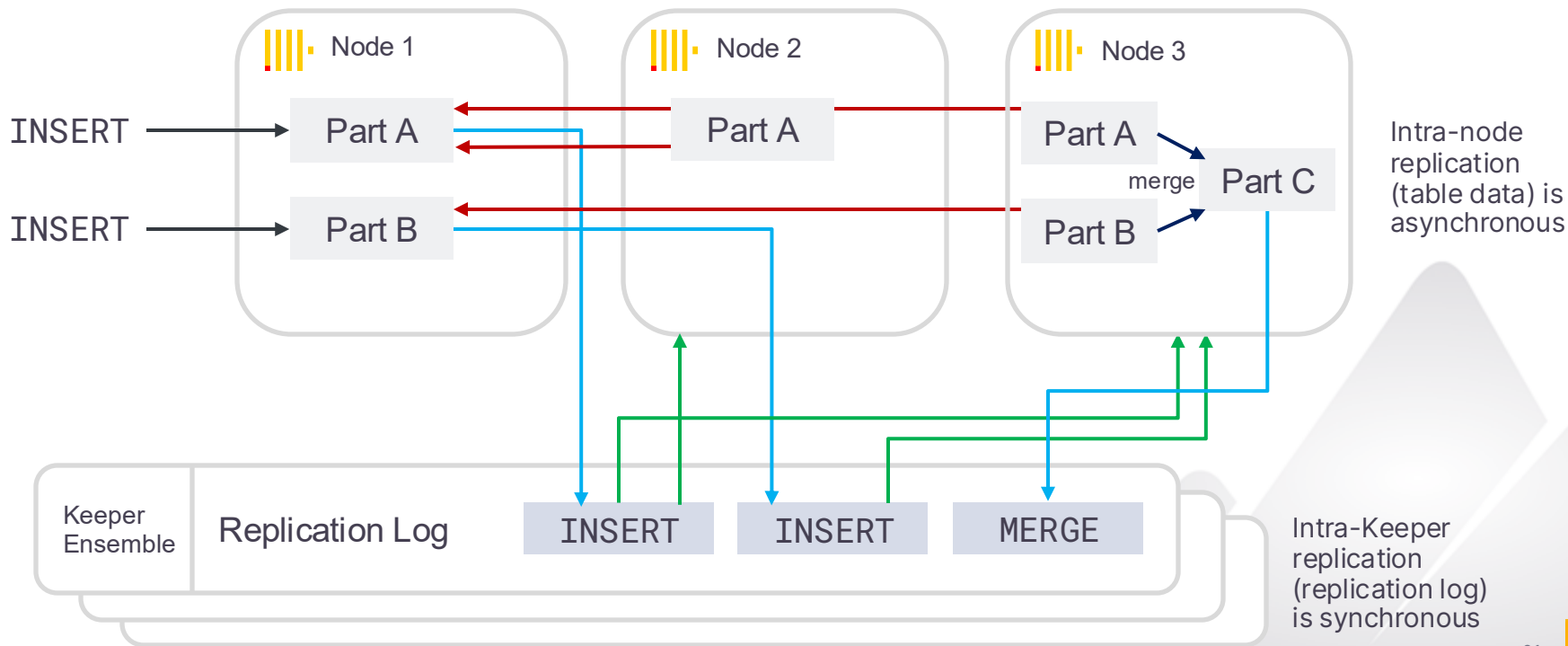


# Data Replication (2/3)



[D. Ongaro: In Search of an Understandable Consensus Algorithm, 2014]

→ Add entry    → Fetch entry    → Download part



## Data Replication (3/3)

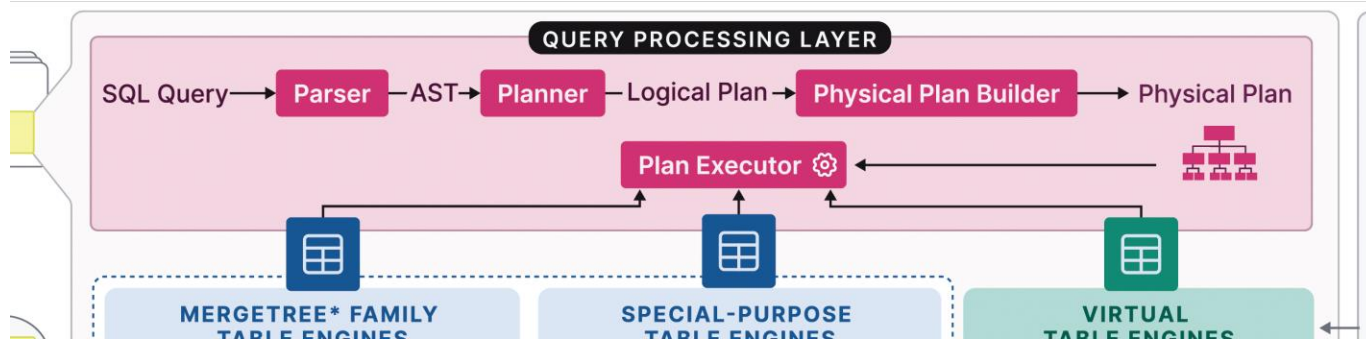
- Nodes replay the replication log asynchronously.
- Nodes may see an old table state but *eventually* they will converge to the latest state.
- The  $\Delta$  between current and latest state is typically small compared to entire table. Applications must still be okay with that.
- On-premise vs. Cloud: shared-nothing vs. shared-disk



# Query Layer

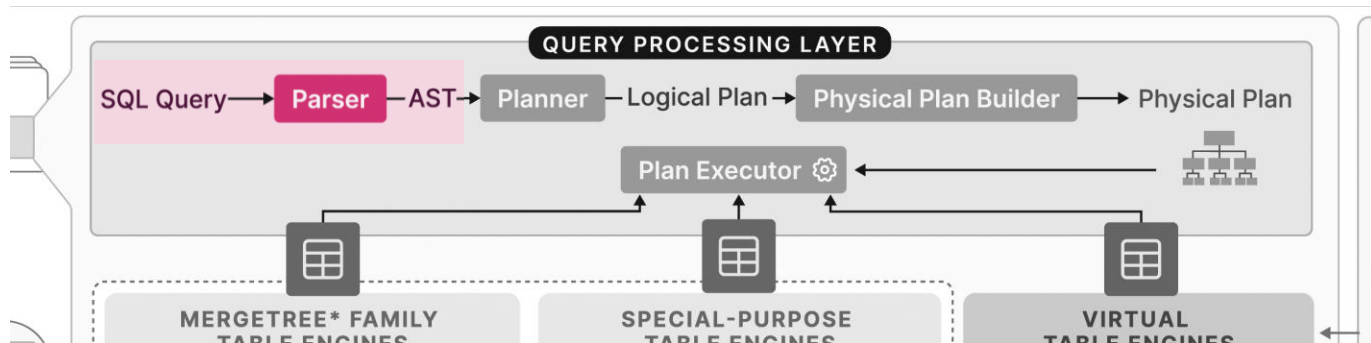


# Query Compilation and Optimization (1/4)





# Query Compilation and Optimization (2/4)



## Optimization of AST

- Constant folding
- Distributive law
- OR  $\rightarrow$  IN list
- [...]

## Example input

`concat(lower('a', upper('b')))`

`sum(2 * x)`

`x = c OR x = d`

## Example output

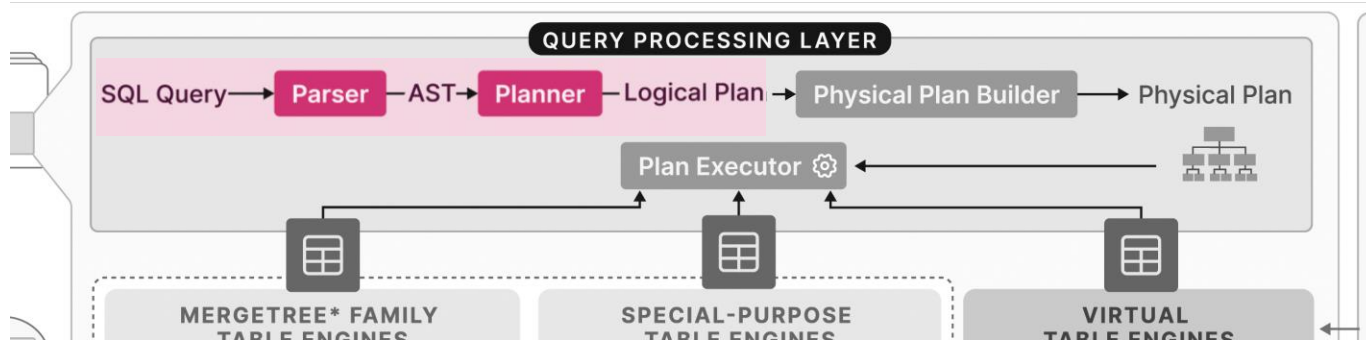
`'aB'`

`2 * sum(x)`

`x IN (c, d)`



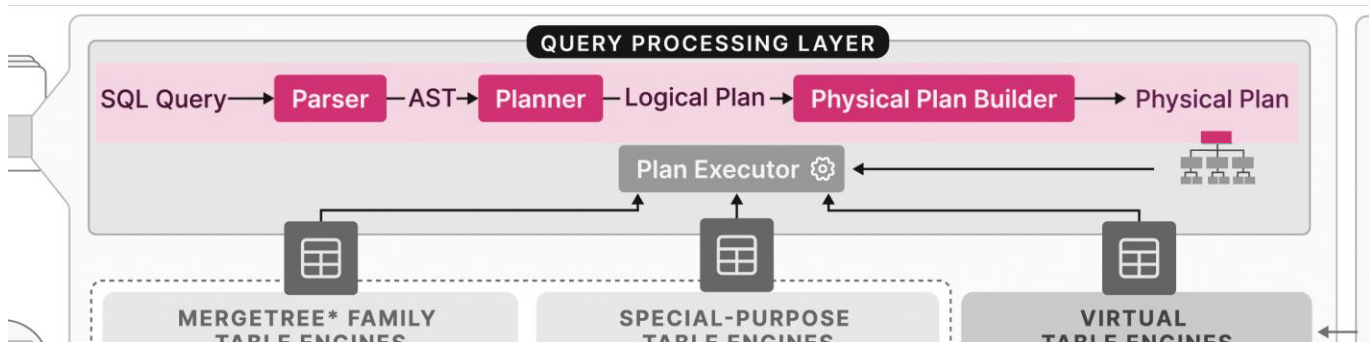
# Query Compilation and Optimization (3/4)



## Optimizations of logical plan (e.g. join, scan, aggregate)

- Filter pushdown
- [...]

# Query Compilation and Optimization (4/4)



## Optimizations of physical plan (e.g. hash join, filter evaluation with PK)

Exploit particularities of table engine. E.g. exploit primary key:

- WHERE columns form prefix of PK columns → replace full scan by PK lookup
- ORDER BY columns form prefix of PK columns → remove sort operator
- GROUP BY columns form prefix of PK columns → remove aggregation operator



# Parallelization Across Data Chunks (1/2)

## Classical Volcano-style execution

- Evaluate operator tree recursively top-to-leaf, one-tuple-at-a-time.
- **Problem 1:** Too many costly (virtual) function calls, bad L1/L2/L3 cache locality.
- **Problem 2:** Not parallelized.

Works for OLTP, unsuitable for OLAP.

## Solve problem 1: **"Vector Vulcano" model**

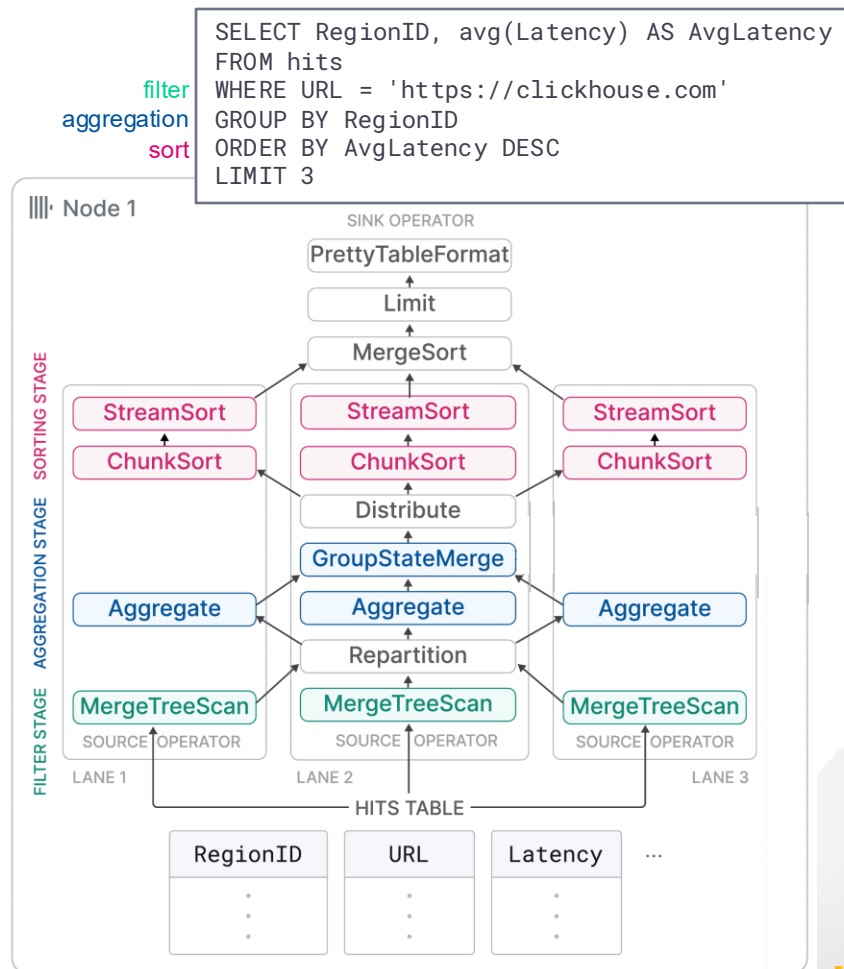
- Pass batches of tuples between operators.
- Amortize cost of calling operators, enables SIMD.

[P. Boncz: MonetDB/X100:  
Hyper-Pipelining Query  
Execution, 2005]



# Parallelization Across Data Chunks (2/2)

- Solve problem 2: **Unfold execution plan into N lanes** (typically 1 lane / core).
- Lanes decompose the data to be processed into non-overlapping ranges.
- Exchange operators (*repartition*, *distribute*) ensure lanes remain balanced.





# Parallelization Across Data Elements (1/2)

- Apply the same operation to consecutive data elements.
- Based on compiler auto-vectorisation or manually written intrinsics.
- Compiled into *compute kernels* which are selected at runtime based based on the system capabilities (cpuid).

```
SELECT col1 + col2  
FROM tab
```

Dispatch code based on cpuid

```
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);
```



# Parallelization Across Data Elements (2/2)

```
SELECT col1 + col2  
FROM tab
```

```
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);
```

AVX-512 kernel, manually vectorised

```
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;  
    }  
}))
```

AVX2 kernel, compiler auto-vectorised

```
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];  
}))
```

# Wrap up

- Looked at LSM-style data organization, data pruning techniques, and parallel query execution.
- The rabbit hole goes muuuuch deeper:
  - powerful SQL dialect,
  - regular, aggregation and window functions with rich functionality,
  - tools for performance introspection and physical database tuning,
  - interoperability with other databases and data formats,
  - user management and backup
  - drivers
  - much more ...
- ClickHouse is open source, development is in the open, contributions are welcome.

