ClickHouse – A Modern Analytical Database

1 Apr, 2024

Robert Schulze

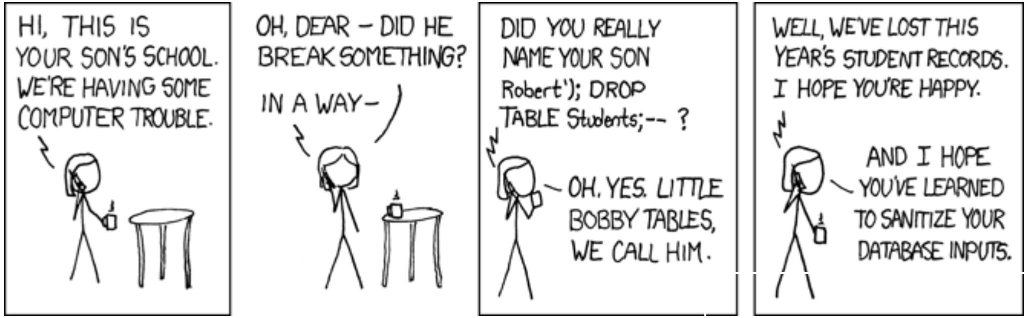
Agenda

- Introduction and Background
- Storage Layer
- Query Layer

Introduction and Background

Who Am I?

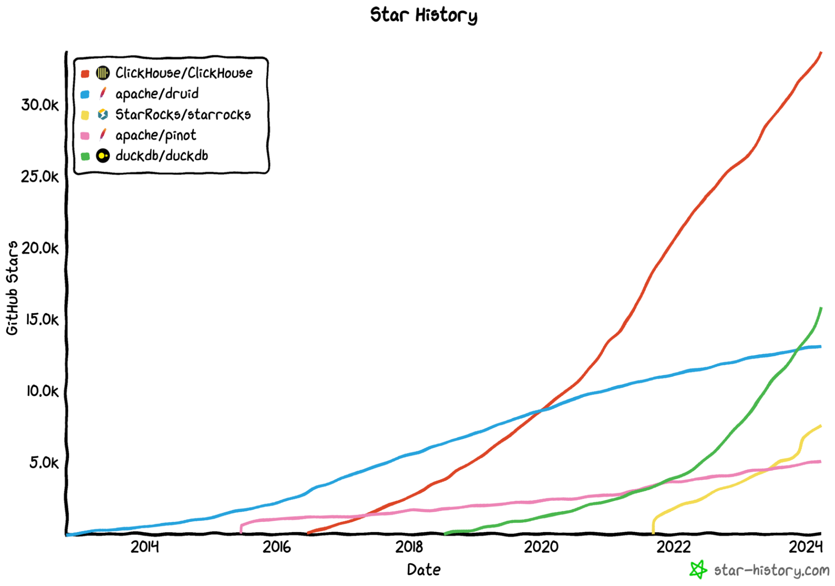
- Robert Schulze
- Senior Software Engineer @ClickHouse Inc., previously at SAP and Dresden University of Technology, Germany
- Focus on query processing, text indexing, vector search
- Supervising student theses (BSc, MSc) and interns (link)



https://xkcd.com/327

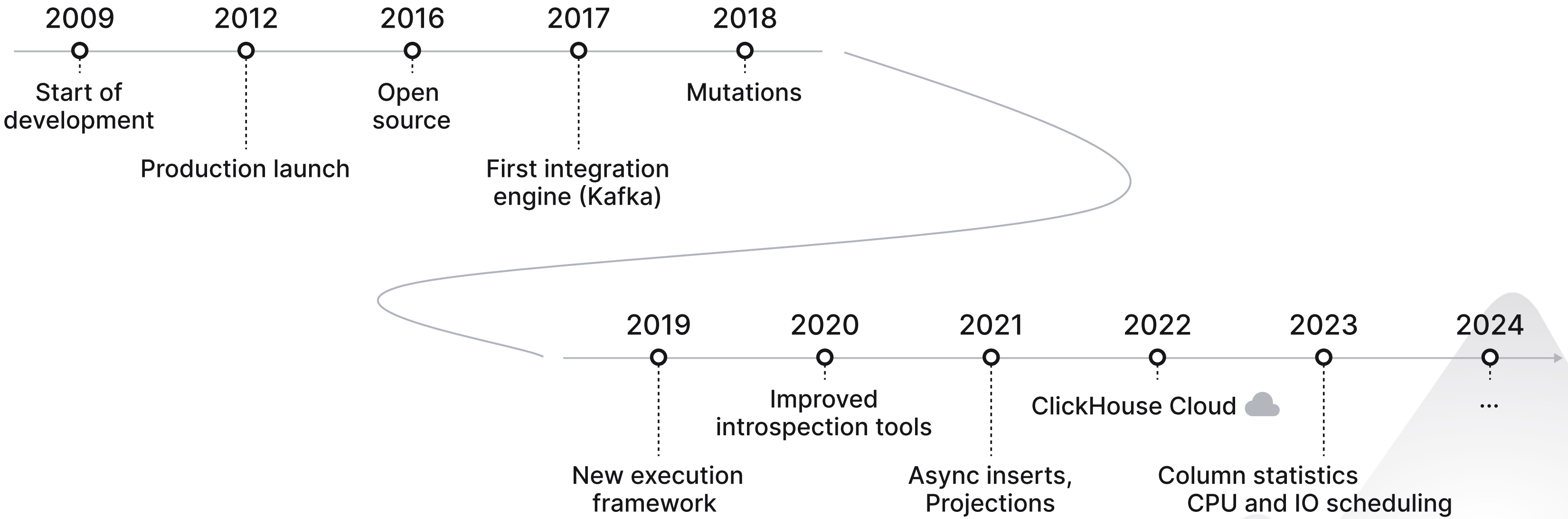
What is ClickHouse?

- An analytical (workload), relational (data model), columnar (data organisation), shared-nothing (architecture) 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), built in C++, runs on anything from Raspberry Pi to clusters with hundreds of nodes.
- Self-managed (on-premises) or ClickHouse Cloud, a database-as-a-service (DBaaS)



https://github.com/ClickHouse/ClickHouse

ClickHouse History




ClickHouse

What is an Analytical Database?

HTAP (Hybrid Transactional Analytical Processing)

OLTP (Online Transactional Processing) ⟷ **OLAP** (Online Analytical Processing)

Workload	Short-running point queries	Long-running batch queries
Data Volumes	Moderate (GB – TB)	Huge (TB – PB)
Emphasis	Data integrity and correctness	Scalability and performance
Data Organization	Row-wise	Column-wise
Use Cases	Enterprise Resource Planning (ERP)	“Big data” and decision making, e.g. dashboards and ad-hoc data exploration
Examples	ORACLE, PostgreSQL, MySQL, IBM DB2, SQL Server	ClickHouse, pinot, druid, DuckDB, UMBRA, monetdb

Row-wise vs. Column-wise Data Organisation

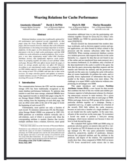
Country	Product	Sales
GB	Lambda	350
FR	Kappa	400
US	Iota	1300

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

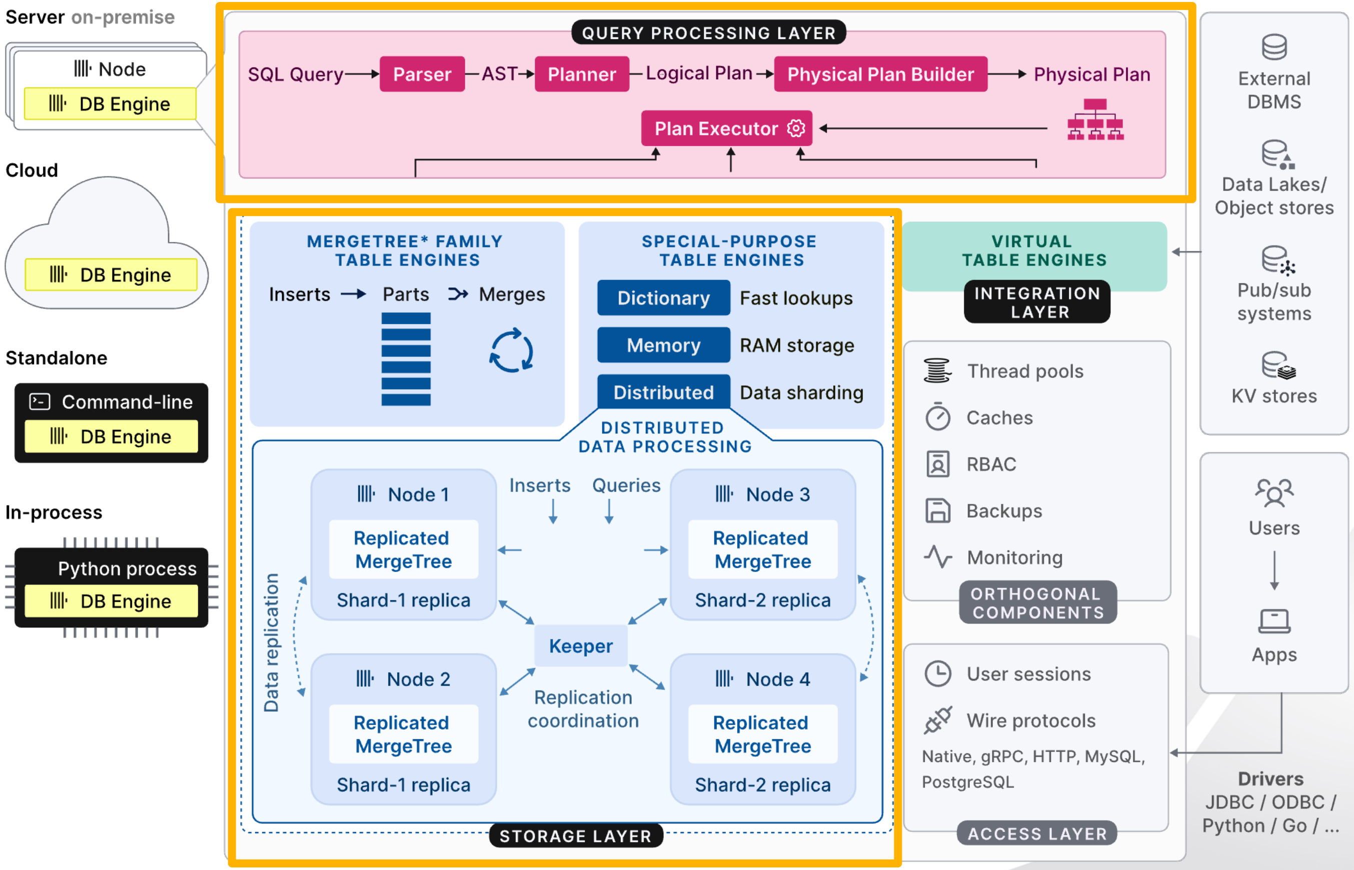


Hybrid (PAX)

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

Compressability	Moderate	High (data of same type clustered together)
Best suited for	Single-row operations	Full-column scans, aggregation

System Architecture

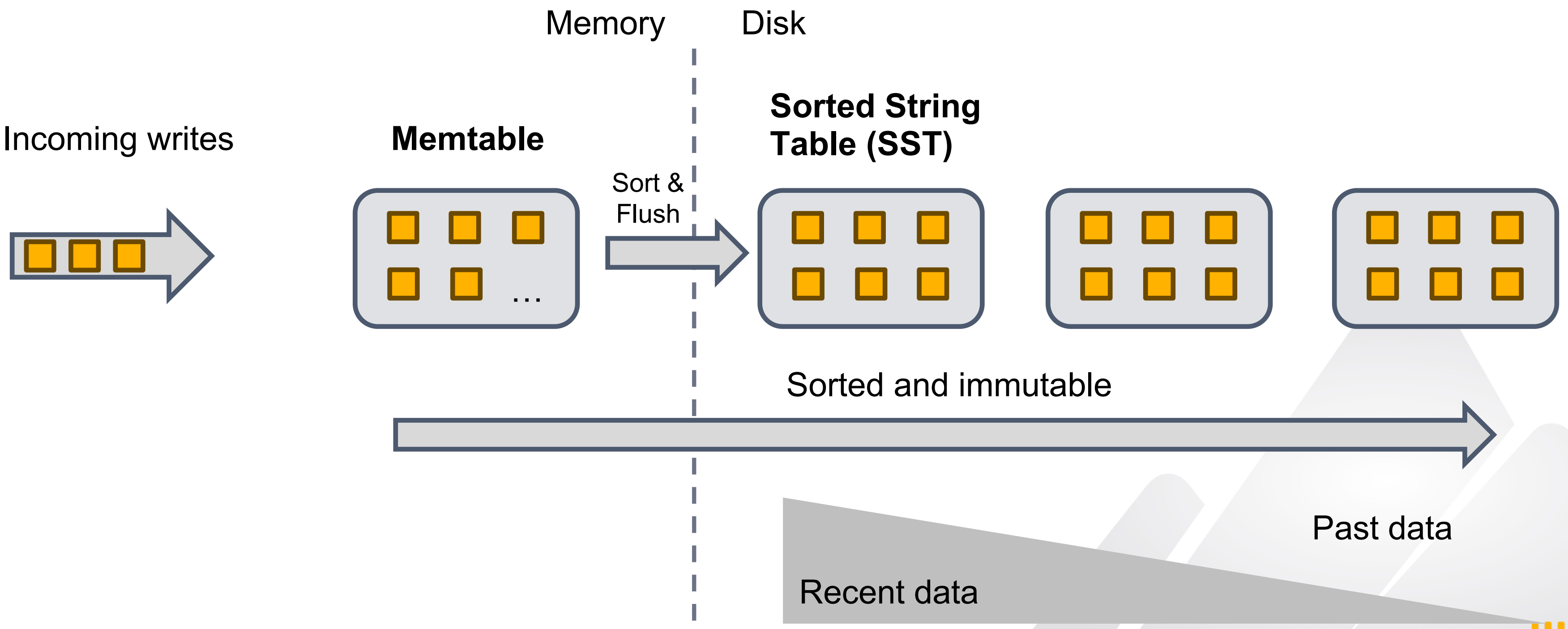


Storage Layer

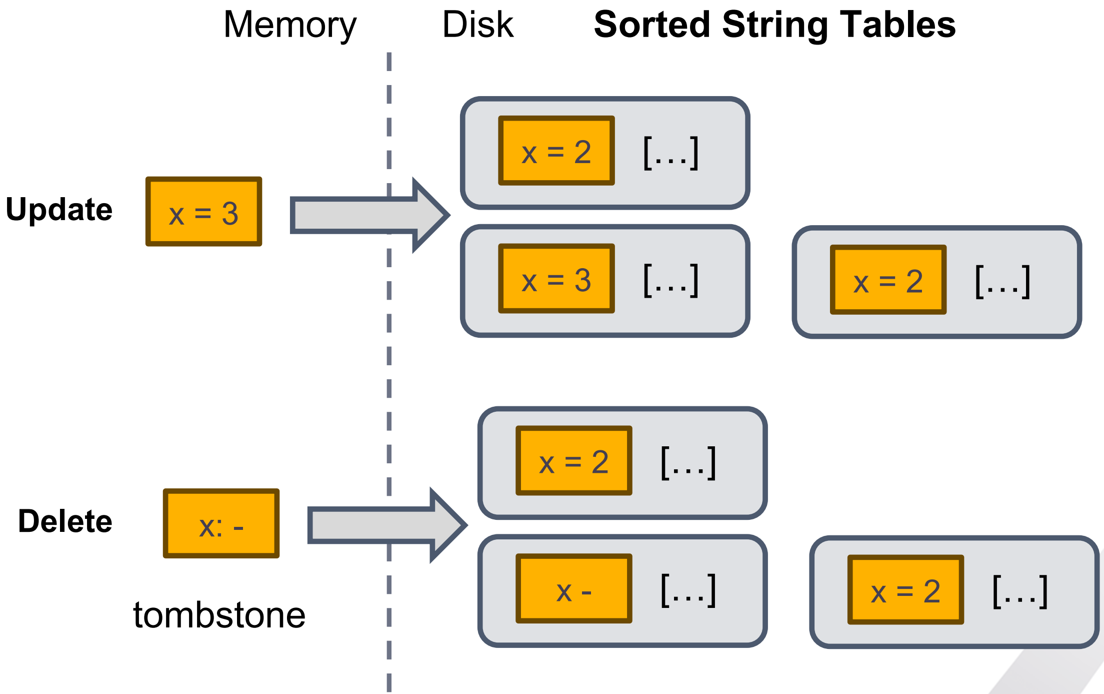
Log-Structured Merge (LSM) Trees

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

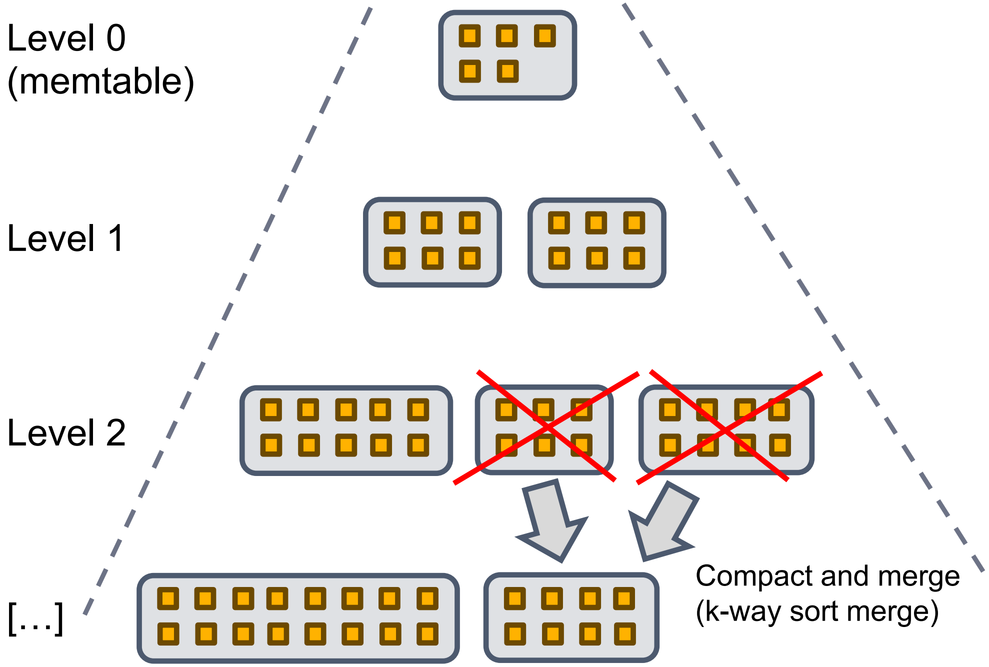
[P. O'Neil et. al.: The Log-Structured Merge-Tree (LSM-Tree), 1996]



LSM Trees: Updates and Deletes



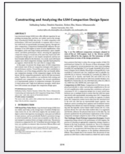
LSM Trees: Compaction



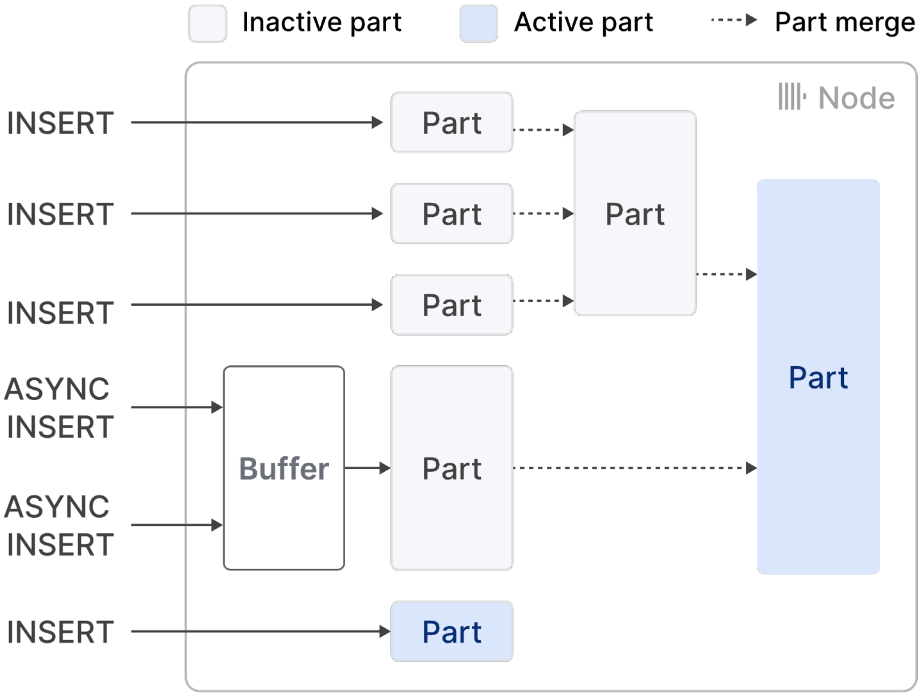
Compaction Strategies

- 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]

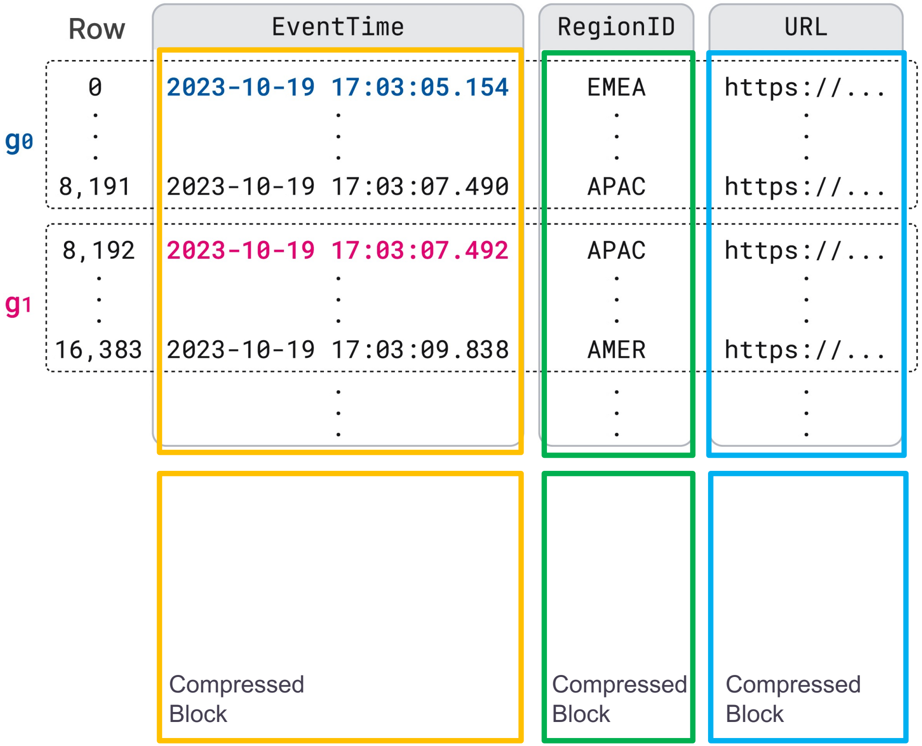


LSM-Tree-Based Storage in ClickHouse



- INSERTs create an immutable *part* (aka. SST).
- INSERTs are synchronous (default) 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)

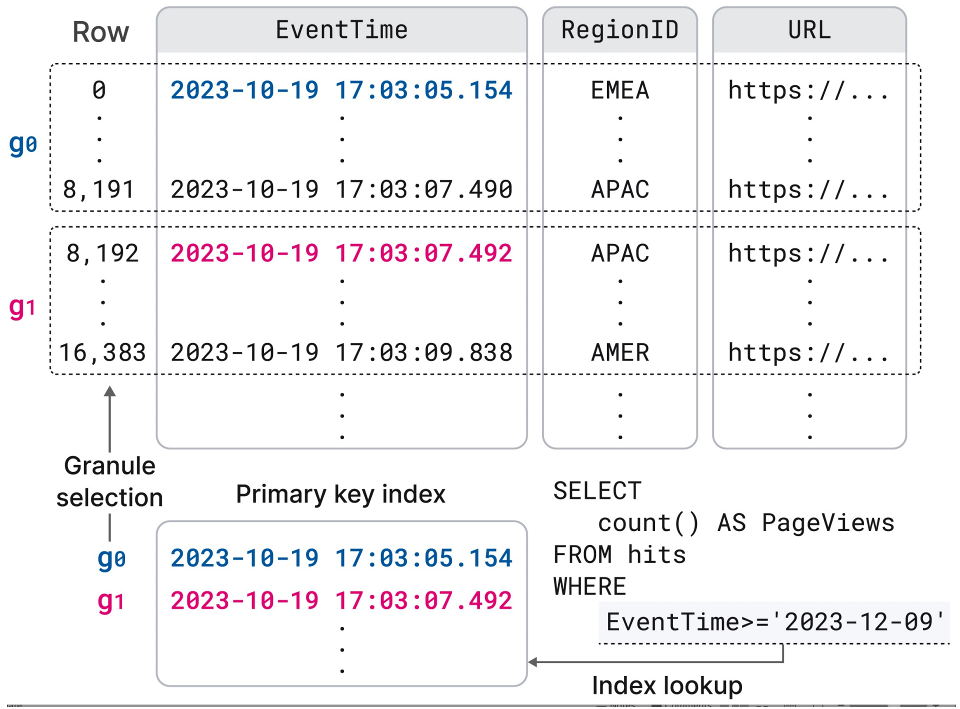


- Local (per-part) sorting defined by primary key:

```
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 *sparse* primary key index.
- 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

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	Some match → load & check
7	max: 25	
25		
25		
18		
20	min: 17	All match --> skip scan
22	max: 22	
19		
17		
8		
6	min: 5	None match → skip scan
6	max: 13	
13		
5		

Merge-time Data Transformation

- Recent data is more relevant than historical data.
- “De-prioritise” old data when parts are merged:
    - Aggregation: collapse rows into aggregated rows
    - Replacement: replace duplicates in older parts
    - Archiving: compress, move, or, delete rows/parts

Data Replication (1/2)

Data Replication means to store the same part redundantly across nodes.

- Enables high availability (tolerance against node failures) and load balancing.

Based on notion of **table state**

= set of table parts + table metadata (e.g. column names/types).

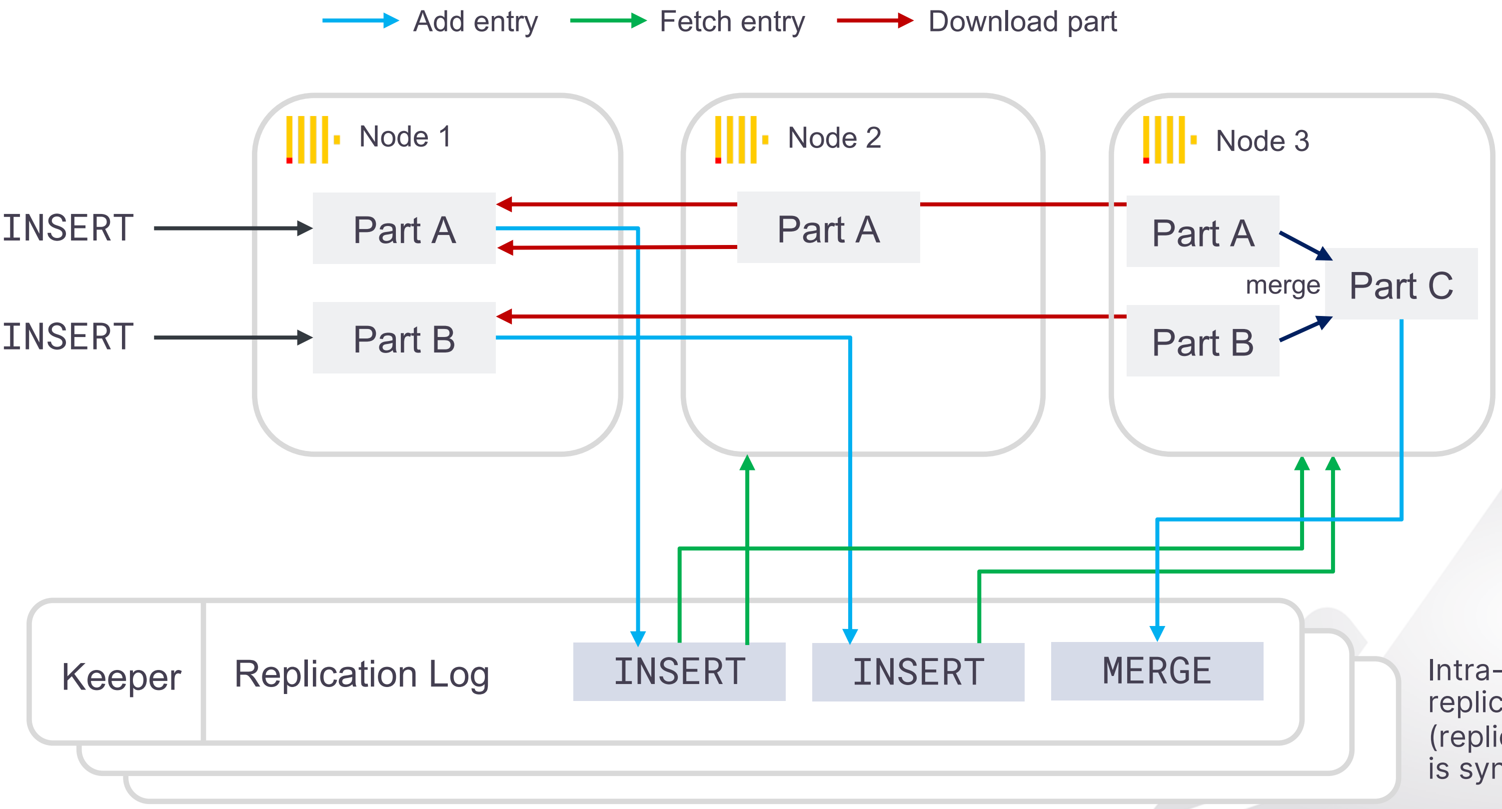
Operations which advance the table state:

- Inserts: Add parts.
- Merges: Add parts + delete parts.
- DDL statements: Add parts + delete parts+ change metadata.

} Recorded in *global replication log*

Data Replication (2/2)

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

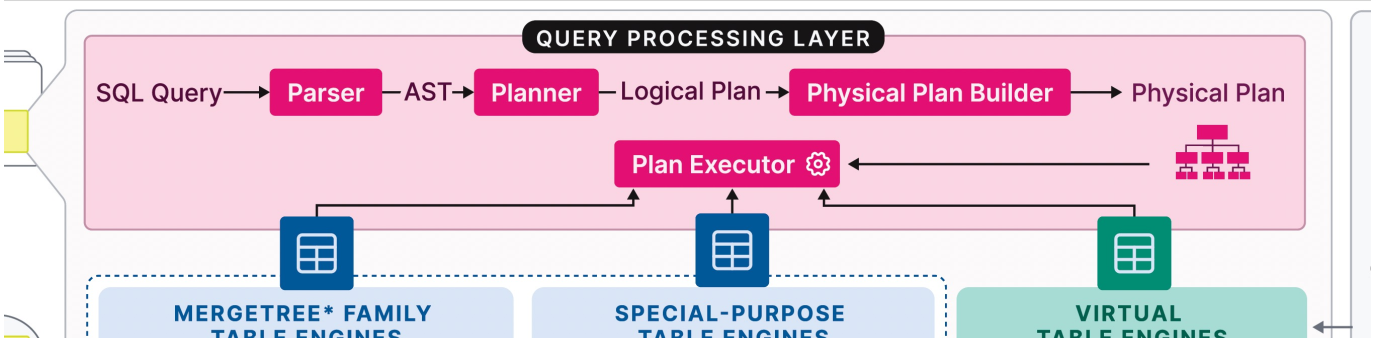


Intra-cluster replication (table data) is asynchronous

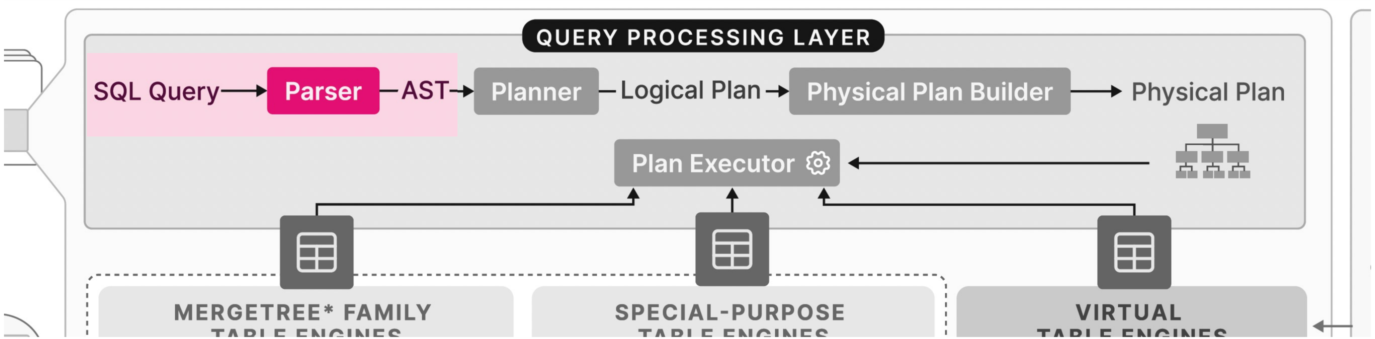
Intra-Keeper replication (replication log) is synchronous

Query Layer

Query Compilation and Optimisation (1/4)

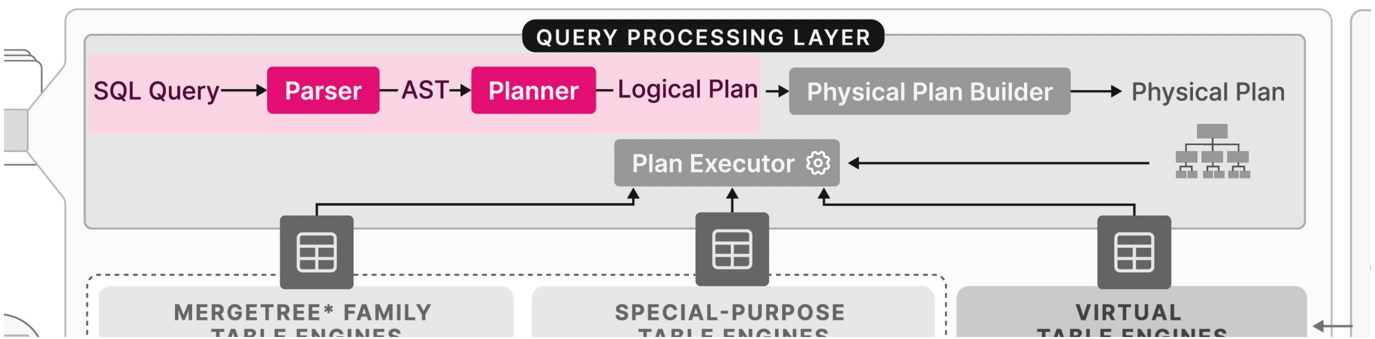


Query Compilation and Optimisation (2/4)



Optimisation of AST	Example input	Example output
• Constant folding	`concat(lower(‘a’, upper(‘b’))`	`‘aB’`
• Distributive law	`sum(2 * x)`	`2 * sum(x)`
• Transform to IN-lists	`x = c OR x = d`	`x IN (c, d)`
• [...]		

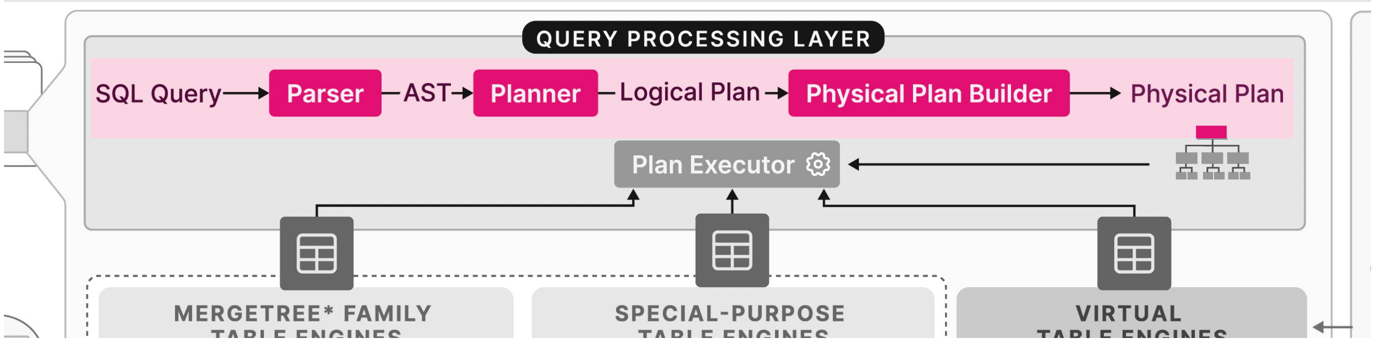
Query Compilation and Optimisation (3/4)



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

- Filter pushdown
- [...]

Query Compilation and Optimisation (4/4)

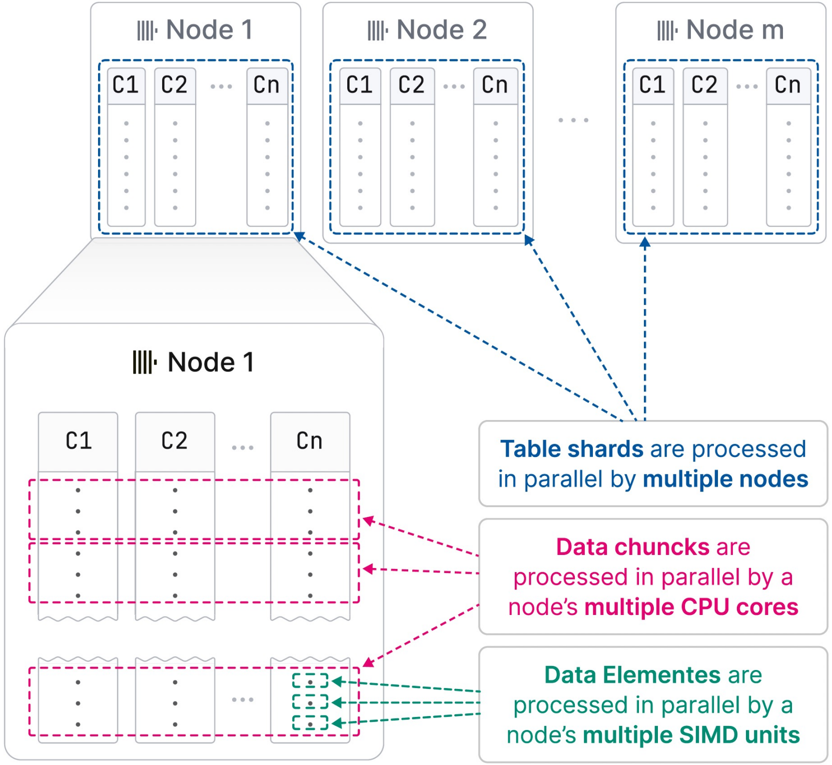


Optimisations of physical plan (e.g. hash join, filter evaluation with PK)

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

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

Query Execution and Parallelisation



Parallelisation Across Data Chunks (1/2)

Classical Volcano-style execution

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

Works for OLTP, unsuitable for OLAP.

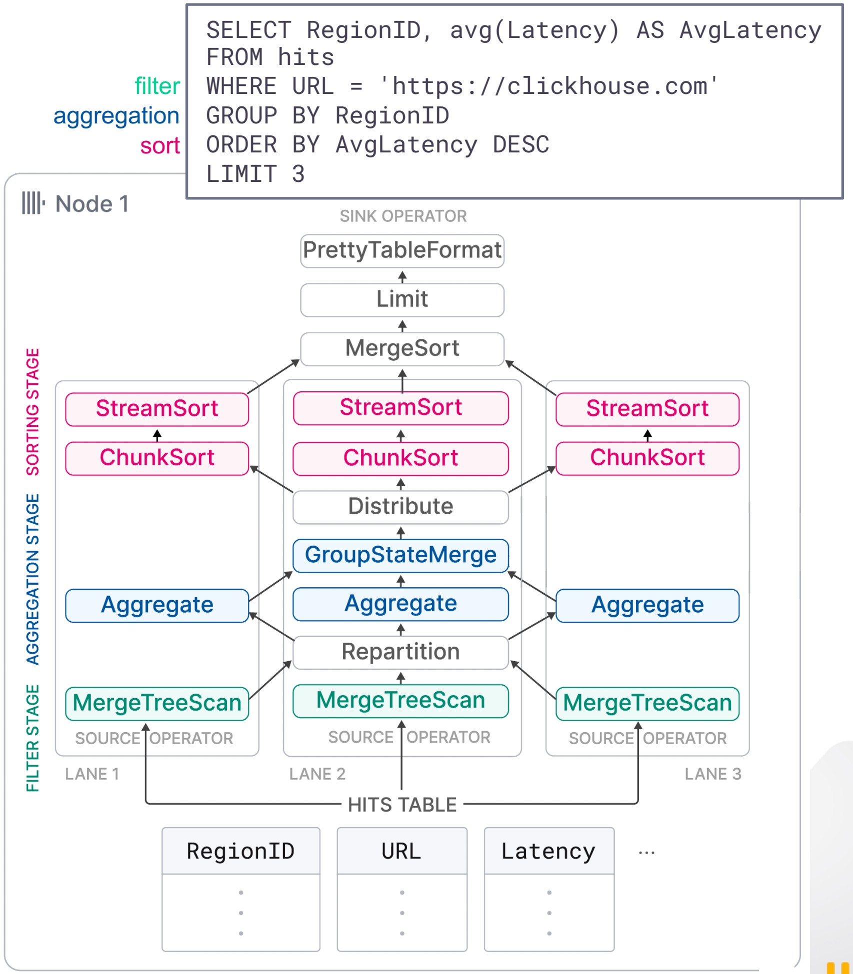
Solve problem 1: **”Vector Vulcano” model**

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

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

Parallelisation 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.



Parallelisation Across Data Elements (1/2)

[J. Zhou: Implementing Database Operations Using SIMD Instructions, 2002]

- 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);
```

Parallelisation 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);
```

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];
}))
```

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;
    }
}))
```

Wrap up

- Looked at LSM-style data organisation, data pruning techniques, and parallel query execution.
- Practical deployment comes additional “soft” requirements:
  - a 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
- ClickHouse is open source, development is in the open, contributions are welcome.