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 internetscale, 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 opensource 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 data¹ and was open sourced in 2016. Figure 1 illustrates when major features described in this paper were introduced to ClickHouse.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097. doi:10.14778/3685800.3685802

¹Blog post: clickhou.se/evolution

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

1. **Huge data sets with high ingestion rates.** Many datadriven 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. SIMD), databases are ideally deployed as native binaries for the target platform.

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@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.



Figure 1: ClickHouse timeline.

2 ARCHITECTURE

As shown by Figure 2, the ClickHouse engine is split into three main layers: the query processing layer (described in Section 4), the storage layer (Section 3), and the integration layer (Section 5). Besides these, an access layer manages user sessions and communication with applications via different protocols. There are orthogonal components for threading, caching, role-based access control, backups, and continuous monitoring. ClickHouse is built in C++ as a single, statically-linked binary without dependencies.

Query processing follows the traditional paradigm of parsing incoming queries, building and optimizing logical and physical query plans, and execution. ClickHouse uses a vectorized execution model similar to MonetDB/X100 [11], in combination with opportunistic code compilation [53]. Queries can be written in a feature-rich SQL dialect, PRQL [76], or Kusto's KQL [50].

The *storage layer* consists of different table engines that encapsulate the format and location of table data. Table engines fall into three categories: The first category is the *MergeTree*^{*} family of table engines which represent the primary persistence format in ClickHouse. Based on the idea of LSM trees [60], tables are split into horizontal, sorted parts, which are continuously merged by a background process. Individual MergeTree^{*} table engines differ in the way the merge combines the rows from its input parts. For example, rows can be aggregated or replaced, if outdated.

The second category are *special-purpose table engines*, which are used to speed up or distribute query execution. This category includes in-memory key-value table engines called dictionaries. A dictionary caches the result of a query periodically executed against an internal or external data source. This significantly reduces access latencies in scenarios, where a degree of data staleness can be tolerated.² Other examples of special-purpose table engines include a pure in-memory engine used for temporary tables and the Distributed table engine for transparent data sharding (see below).

The third category of table engines are *virtual table engines* for bidirectional data exchange with external systems such as relational databases (e.g. PostgreSQL, MySQL), publish/subscribe systems (e.g. Kafka, RabbitMQ [24]), or key/value stores (e.g. Redis). Virtual engines can also interact with data lakes (e.g. Iceberg, DeltaLake, Hudi [36]) or files in object storage (e.g. AWS S3, Google GCP).

ClickHouse supports sharding and replication of tables across multiple cluster nodes for scalability and availability. Sharding partitions a table into a set of *table shards* according to a sharding expression. The individual shards are mutually independent tables and typically located on different nodes. Clients can read and write shards directly, i.e. treat them as separate tables, or use the Distributed special table engine, which provides a global view of all table shards. The main purpose of sharding is to process data sets which exceed the capacity of individual nodes (typically, a few dozens terabytes of data). Another use of sharding is to distribute the read-write load for a table over multiple nodes, i.e., load balancing. Orthogonal to that, a shard can be *replicated* across multiple nodes for tolerance against node failures. To that end, each Merge-Tree* table engine has a corresponding *ReplicatedMergeTree** engine which uses a multi-master coordination scheme based on Raft consensus [59] (implemented by Keeper³, a drop-in replacement for Apache Zookeeper written in C++) to guarantee that every shard has, at all times, a configurable number of replicas. Section 3.6 discusses the replication mechanism in detail. As an example, Figure 2 shows a table with two shards, each replicated to two nodes.

Finally, the ClickHouse database engine can be operated in onpremise, cloud, standalone, or in-process modes. In the on-premise mode, users set up ClickHouse locally as a single server or multinode cluster with sharding and/or replication. Clients communicate with the database over the native, MySQL's, PostgreSQL's binary wire protocols, or an HTTP REST API. The cloud mode is represented by ClickHouse Cloud, a fully managed and autoscaling DBaaS offering. While this paper focuses on the on-premise mode, we plan to describe the architecture of ClickHouse Cloud in a follow-up publication. The standalone mode turns ClickHouse into a command line utility for analyzing and transforming files, making it a SQL-based alternative to Unix tools like cat and grep.⁴ While this requires no prior configuration, the standalone mode is restricted to a single server. Recently, an in-process mode called chDB [15] has been developed for interactive data analysis use cases like Jupyter notebooks [37] with Pandas dataframes [61]. Inspired by DuckDB [67], chDB embeds ClickHouse as a high-performance OLAP engine into a host process. Compared to the other modes, this allows to pass source and result data between the database engine and the application efficiently without copying as they run in the same address space.⁵

3 STORAGE LAYER

This section discusses MergeTree* table engines as ClickHouse's native storage format. We describe their on-disk representation and discuss three data pruning techniques in ClickHouse. Afterwards, we present merge strategies which continuously transform data without impacting simultaneous inserts. Finally, we explain how updates and deletes are implemented, as well as data deduplication, data replication, and ACID compliance.

²Blog post: clickhou.se/dictionaries

³Blog post: clickhou.se/keeper

⁴Blog posts: clickhou.se/local, clickhou.se/local-fastest-tool

⁵Blog post: clickhou.se/chdb-rocket-engine



Figure 2: The high-level architecture of the ClickHouse database engine.

3.1 On-Disk Format

Each table in the MergeTree^{*} table engine is organized as a collection of immutable table *parts*. A part is created whenever a set of rows is inserted into the table. Parts are self-contained in the sense that they include all metadata required to interpret their content without additional lookups to a central catalog. To keep the number of parts per table low, a background merge job periodically combines multiple smaller parts into a larger part until a configurable part size is reached (150 GB by default). Since parts are sorted by the table's primary key columns (see Section 3.2), efficient k-way merge sort [40] is used for merging. The source parts are marked as inactive and eventually deleted as soon as their reference count drops to zero, i.e. no further queries read from them.

Rows can be inserted in two modes: In synchronous insert mode, each INSERT statement creates a new part and appends it to the table. To minimize the overhead of merges, database clients are encouraged to insert tuples in bulk, e.g. 20,000 rows at once. However, delays caused by client-side batching are often unacceptable if the data should be analyzed in real-time. For example, observability use cases frequently involve thousands of monitoring agents continuously sending small amounts of event and metrics data. Such scenarios can utilize the asynchronous insert mode, in which ClickHouse buffers rows from multiple incoming INSERTs into the same table and creates a new part only after the buffer size exceeds a configurable threshold or a timeout expires.



Figure 3: Inserts and merges for MergeTree*-engine tables.

Figure 3 illustrates four synchronous and two asynchronous inserts into a MergeTree*-engine table. Two merges reduced the number of active parts from initially five to two.

Compared to LSM trees [58] and their implementation in various databases [13, 26, 56], ClickHouse treats all parts as equal instead of arranging them in a hierarchy. As a result, merges are no longer limited to parts in the same level. Since this also forgoes the implicit

chronological ordering of parts, alternative mechanisms for updates and deletes not based on tombstones are required (see Section 3.4). ClickHouse writes inserts directly to disk while other LSM-treebased stores typically use write-ahead logging (see Section 3.7).

A part corresponds to a directory on disk, containing one file for each column. As an optimization, the columns of a small part (smaller than 10 MB by default) are stored consecutively in a single file to increase the spatial locality for reads and writes. The rows of a part are further logically divided into groups of 8192 records, called granules. A granule represents the smallest indivisible data unit processed by the scan and index lookup operators in ClickHouse. Reads and writes of on-disk data are, however, not performed at the granule level but at the granularity of blocks, which combine multiple neighboring granules within a column. New blocks are formed based on a configurable byte size per block (by default 1 MB), i.e., the number of granules in a block is variable and depends on the column's data type and distribution. Blocks are furthermore compressed to reduce their size and I/O costs. By default, ClickHouse employs LZ4 [75] as a general-purpose compression algorithm, but users can also specify specialized codecs like Gorilla [63] or FPC [12] for floating-point data. Compression algorithms can also be chained. For example, it is possible to first reduce logical redundancy in numeric values using delta coding [23], then perform heavy-weight compression, and finally encrypt the data using an AES codec. Blocks are decompressed on-the-fly when they are loaded from disk into memory. To enable fast random access to individual granules despite compression, ClickHouse additionally stores for each column a mapping that associates every granule id with the offset of its containing compressed block in the column file and the offset of the granule in the uncompressed block.

Columns can further be dictionary-encoded [2, 77, 81] or made nullable using two special wrapper data types: LowCardinality(T) replaces the original column values by integer ids and thus significantly reduces the storage overhead for data with few unique values. Nullable(T) adds an internal bitmap to column T, representing whether column values are NULL or not.

Finally, tables can be range, hash, or round-robin partitioned using arbitrary partitioning expressions. To enable partition pruning, ClickHouse additionally stores the partitioning expression's minimum and maximum values for each partition. Users can optionally create more advanced column statistics (e.g., HyperLogLog [30] or t-digest [28] statistics) that also provide cardinality estimates.

3.2 Data Pruning

In most use cases, scanning petabytes of data just to answer a single query is too slow and expensive. ClickHouse supports three data pruning techniques that allow skipping the majority of rows during searches and therefore speed up queries significantly.

First, users can define a primary key index for a table. The primary key columns determine the sort order of the rows within each part, i.e. the index is locally clustered. ClickHouse additionally stores, for every part, a mapping from the primary key column values of each granule's first row to the granule's id, i.e. the index is sparse [31]. The resulting data structure is typically small enough to remain fully in-memory, e.g., only 1000 entries are required to index 8.1 million rows. The main purpose of a primary key is to evaluate

hits table with EventTime as primary key



Figure 4: Evaluating filters with a primary key index.

equality and range predicates for frequently filtered columns using binary search instead of sequential scans (Section 4.4). The local sorting can furthermore be exploited for part merges and query optimization, e.g. sort-based aggregation or to remove sorting operators from the physical execution plan when the primary key columns form a prefix of the sorting columns.

Figure 4 shows a primary key index on column EventTime for a table with page impression statistics. Granules that match the range predicate in the query can be found by binary searching the primary key index instead of scanning EventTime sequentially.

Second, users can create table projections, i.e., alternative versions of a table that contain the same rows sorted by a different primary key [71]. Projections allow to speed up queries that filter on columns different than the main table's primary key at the cost of an increased overhead for inserts, merges, and space consumption. By default, projections are populated lazily only from parts newly inserted into the main table but not from existing parts unless the user materializes the projection in full. The query optimizer chooses between reading from the main table or a projection based on estimated I/O costs. If no projection exists for a part, query execution falls back to the corresponding main table part.

Third, skipping indices provide a lightweight alternative to projections. The idea of skipping indices is to store small amounts of metadata at the level of multiple consecutive granules which allows to avoid scanning irrelevant rows. Skipping indices can be created for arbitrary index expressions and using a configurable granularity, i.e. number of granules in a skipping index block. Available skipping index types include: 1. Min-max indices [51], storing the minimum and maximum values of the index expression for each index block. This index type works well for locally clustered data with small absolute ranges, e.g. loosely sorted data. 2. Set indices, storing a configurable number of unique index block values. These indexes are best used with data with a small local cardinality, i.e. "clumped together" values. 3. Bloom filter indices [9] build for row, token, or n-gram values with a configurable false positive rate. These indices support text search [73], but unlike min-max and set indices, they cannot be used for range or negative predicates.

3.3 Merge-time Data Transformation

Business intelligence and observability use cases often need to handle data generated at constantly high rates or in bursts. Also, recently generated data is typically more relevant for meaningful real-time insights than historical data. Such use cases require databases to sustain high data ingestion rates while continuously reducing the volume of historical data through techniques like aggregation or data aging. ClickHouse allows a continuous incremental transformation of existing data using different merge strategies. Merge-time data transformation does not compromise the performance of INSERT statements, but it cannot guarantee that tables never contain unwanted (e.g. outdated or non-aggregated) values. If necessary, all merge-time transformations can be applied at query time by specifying the keyword FINAL in SELECT statements.

Replacing merges retain only the most recently inserted version of a tuple based on the creation timestamp of its containing part, older versions are deleted. Tuples are considered equivalent if they have the same primary key column values. For explicit control over which tuple is preserved, it is also possible to specify a special version column for comparison. Replacing merges are commonly used as a merge-time update mechanism (normally in use cases where updates are frequent), or as an alternative to insert-time data deduplication (Section 3.5).

Aggregating merges collapse rows with equal primary key column values into an aggregated row. Non-primary key columns must be of a partial aggregation state that holds the summary values. Two partial aggregation states, e.g. a sum and a count for avg(), are combined into a new partial aggregation state. Aggregating merges are typically used in materialized views instead of normal tables. Materialized views are populated based on a transformation query against a source table. Unlike other databases, ClickHouse does not refresh materialized views periodically with the entire content of the source table. Materialized views are rather updated incrementally with the result of the transformation query when a new part is inserted into the source table.

Figure 5 shows a materialized view defined on a table with page impression statistics. For new parts inserted into the source table, the transformation query computes the maximum and average latencies, grouped by region, and inserts the result into a materialized view. Aggregation functions avg() and max() with extension -State return partial aggregation states instead of actual results. An aggregating merge defined for the materialized view continuously combines partial aggregation states in different parts. To obtain the final result, users consolidate the partial aggregation states in the materialized view using avg() and max()) with -Merge extension.

TTL (time-to-live) merges provide aging for historical data. Unlike deleting and aggregating merges, TTL merges process only one part at a time. TTL merges are defined in terms of rules with triggers and actions. A trigger is an expression computing a timestamp for every row, which is compared against the time at which the TTL merge runs. While this allows users to control actions at row granularity, we found it sufficient to check whether *all* rows satisfy a given condition and run the action on the entire part. Possible actions include 1. move the part to another volume (e.g. cheaper and slower storage), 2. re-compress the part (e.g. with a



Figure 5: Aggregating merges in materialized views.

more heavy-weight codec), 3. delete the part, and 4. roll-up, i.e. aggregate the rows using a grouping key and aggregate functions.

As an example, consider the logging table definition in Listing 1. ClickHouse will move parts with timestamp column values older than one week to slow but inexpensive S3 object storage.

```
CREATE TABLE tab(ts DateTime, msg String)
ENGINE MergeTree PRIMARY KEY ts
TTL (ts + INTERVAL 1 WEEK) TO VOLUME 's3'
```

Listing 1: Move part to object storage after one week.

3.4 Updates and Deletes

The design of the MergeTree^{*} table engines favors append-only workloads, yet some use cases require to modify existing data occasionally, e.g. for regulatory compliance. Two approaches for updating or deleting data exist, neither of which block parallel inserts.

Mutations rewrite all parts of a table in-place. To prevent a table (delete) or column (update) from doubling temporarily in size, this operation is non-atomic, i.e. parallel SELECT statements may read mutated and non-mutated parts. Mutations guarantee that the data is physically changed at the end of the operation. Delete mutations are still expensive as they rewrite all columns in all parts.

As an alternative, *lightweight deletes* only update an internal bitmap column, indicating if a row is deleted or not. ClickHouse amends SELECT queries with an additional filter on the bitmap column to exclude deleted rows from the result. Deleted rows are physically removed only by regular merges at an unspecified time in future. Depending on the column count, lightweight deletes can be much faster than mutations, at the cost of slower SELECTs.

Update and delete operations performed on the same table are expected to be rare and serialized to avoid logical conflicts.

3.5 Idempotent Inserts

A problem that frequently occurs in practice is how clients should handle connection timeouts after sending data to the server for insertion into a table. In this situation, it is difficult for clients to distinguish between whether the data was successfully inserted or not. The problem is traditionally solved by re-sending the data from the client to the server and relying on primary key or unique constraints to reject duplicate inserts. Databases perform the required point lookups quickly using index structures based on binary trees [39, 68], radix trees [45], or hash tables [29]. Since these data structures index every tuple, their space and update overhead becomes prohibitive for large data sets and high ingest rates.

ClickHouse provides a more light-weight alternative based on the fact that each insert eventually creates a part. More specifically, the server maintains hashes of the N last inserted parts (e.g. N=100) and ignores re-inserts of parts with a known hash. Hashes for non-replicated and replicated tables are stored locally, respectively, in Keeper. As a result, inserts become *idempotent*, i.e. clients can simply re-send the same batch of rows after a timeout and assume that the server takes care of deduplication. For more control over the deduplication process, clients can optionally provide an insert token that acts as a part hash. While hash-based deduplication incurs an overhead associated with hashing the new rows, the cost of storing and comparing hashes is negligible.

3.6 Data Replication

Replication is a prerequisite for high availability (tolerance against node failures), but also used for load balancing and zero-downtime upgrades [14]. In ClickHouse, replication is based on the notion of *table states* which consist of a set of table parts (Section 3.1) and table metadata, such as column names and types. Nodes advance the state of a table using three operations: 1. Inserts add a new part to the state, 2. merges add a new part and delete existing parts to/from the state, 3. mutations and DDL statements add parts, and/or delete parts, and/or change table metadata, depending on the concrete operation. Operations are performed locally on a single node and recorded as a sequence of state transition in a global *replication log*.

The replication log is maintained by an ensemble of typically three ClickHouse Keeper processes which use the Raft consensus algorithm [59] to provide a distributed and fault-tolerant coordination layer for a cluster of ClickHouse nodes. All cluster nodes initially point to the same position in the replication log. While the nodes execute local inserts, merges, mutations, and DDL statements, the replication log is replayed asynchronously on all other nodes. As a result, replicated tables are only eventually consistent, i.e. nodes can temporarily read old table states while converging



Figure 6: Replication in a cluster of three nodes.

towards the latest state. Most aforementioned operations can alternatively be executed synchronously until a quorum of nodes (e.g. a majority of nodes or all nodes) adopted the new state.

As an example, Figure 6 shows an initially empty replicated table in a cluster of three ClickHouse nodes. Node 1 first receives two insert statements and records them ((1) (2)) in the replication log stored in the Keeper ensemble. Next, Node 2 replays the first log entry by fetching it ((3)) and downloading the new part from Node 1 ((4)), whereas Node 3 replays both log entries ((3) (4) (5) (6)). Finally, Node 3 merges both parts to a new part, deletes the input parts, and records a merge entry in the replication log ((7)).

Three optimizations to speed up synchronization exist: First, new nodes added to the cluster do replay the replication log from scratch, instead they simply copy the state of the node which wrote the last replication log entry. Second, merges are replayed by repeating them locally or by fetching the result part from another node. The exact behavior is configurable and allows to balance CPU consumption and network I/O. For example, cross-data-center replication typically prefers local merges to minimize operating costs. Third, nodes replay mutually independent replication log entries in parallel. This includes, for example, fetches of new parts inserted consecutively into the same table, or operations on different tables.

3.7 ACID Compliance

To maximize the performance of concurrent read and write operations, ClickHouse avoids latching as much as possible. Queries are executed against a snapshot of all parts in all involved tables created at the beginning of the query. This ensures that new parts inserted by parallel INSERTs or merges (Section 3.1) do not participate in execution. To prevent parts from being modified or removed simultaneously (Section 3.4), the reference count of the processed parts is incremented for the duration of the query. Formally, this corresponds to snapshot isolation realized by an MVCC variant [6] based on versioned parts. As a result, statements are generally not ACID-compliant except for the rare case that concurrent writes at the time the snapshot is taken each affect only a single part.

In practice, most of ClickHouse's write-heavy decision making use cases even tolerate a small risk of losing new data in case of a power outage. The database takes advantage of this by not forcing a commit (fsync) of newly inserted parts to disk by default, allowing the kernel to batch writes at the cost of forgoing atomicity.



Figure 7: Parallelization across SIMD units, cores and nodes.

4 QUERY PROCESSING LAYER

As illustrated by Figure 7, ClickHouse parallelizes queries at the level of data elements, data chunks, and table shards. Multiple data elements can be processed within operators at once using SIMD instructions. On a single node, the query engine executes operators simultaneously in multiple threads. ClickHouse uses the same vectorization model as MonetDB/X100 [11], i.e. operators produce, pass, and consume multiple rows (*data chunks*) instead of single rows to minimize the overhead of virtual function calls. If a source table is split into disjoint table shards, multiple nodes can scan the shards simultaneously. As a result, all hardware resources are fully utilized, and query processing can be scaled horizontally by adding nodes and vertically by adding cores.

The rest of this section first describes parallel processing at data element, data chunk, and shard granularity in more detail. We then present selected key optimizations to maximize query performance. Finally, we discuss how ClickHouse manages shared system resources in the presence of simultaneous queries.

4.1 SIMD Parallelization

Passing multiple rows between operators creates an opportunity for vectorization. Vectorization is either based on manually written intrinsics [64, 80] or compiler auto-vectorization [25]. Code that benefit from vectorization is compiled into different *compute kernels*. For example, the inner hot loop of a query operator can be implemented in terms of a non-vectorized kernel, an auto-vectorized AVX2 kernel, and a manually vectorized AVX-512 kernel. The fastest kernel is chosen at runtime based on the cpuid instruction.⁶ This approach allows ClickHouse to run on systems as old as 15 years (requiring SSE 4.2 as a minimum), while still providing significant speedups on recent hardware.

⁶Blog post: clickhou.se/cpu-dispatch

4.2 Multi-Core Parallelization

ClickHouse follows the conventional approach [31] of transforming SQL queries into a directed graph of physical plan operators. The input of the operator plan is represented by special source operators that read data in the native or any of the supported 3rd-party formats (see Section 5). Likewise, a special sink operator converts the result into the desired output format. The physical operator plan is unfolded at query compilation time into independent *execution lanes* based on a configurable maximum number of worker threads (by default, the number of cores) and the source table size. Lanes decompose the data to be processed by parallel operators into non-overlapping ranges. To maximize the opportunity for parallel processing, lanes are merged as late as possible.

As an example, the box for Node 1 in Figure 8 shows the operator graph of a typical OLAP query against a table with page impression statistics. In the first stage, three disjoint ranges of the source table are filtered simultaneously. A Repartition exchange operator dynamically routes result chunks between the first and second stages to keep the processing threads evenly utilized. Lanes may become imbalanced after the first stage if the scanned ranges have significantly different selectivities. In the second stage, the rows that survived the filter are grouped by RegionID. The Aggregate operators maintain local result groups with RegionID as a grouping column and a per-group sum and count as a partial aggregation state for avg(). The local aggregation results are eventually merged by a GroupStateMerge operator into a global aggregation result. This operator is also a pipeline breaker, i.e., the third stage can only start once the aggregation result has been fully computed. In the third stage, the result groups are first divided by a Distribute exchange operator into three equally large disjoint partitions, which are then sorted by AvgLatency. Sorting is performed in three steps: First, ChunkSort operators sort the individual chunks of each partition. Second, StreamSort operators maintain a local sorted result which is combined with incoming sorted chunks using 2-way merge sorting. Finally, a MergeSort operator combines the local results using k-way sorting to obtain the final result.

Operators are state machines and connected to each other via input and output ports. The three possible states of an operator are need-chunk, ready, and done. To move from need-chunk to ready, a chunk is placed in the operator's input port. To move from ready to done, the operator processes the input chunk and generates an output chunk. To move from done to need-chunk, the output chunk is removed from the operator's output port. The first and third state transitions in two connected operators can only be performed in a combined step. Source operators (sink operators) only have states ready and done (need-chunk and done).

Worker threads continuously traverse the physical operator plan and perform state transitions. To keep CPU caches hot, the plan contains hints that the same thread should process consecutive operators in the same lane. Parallel processing happens both horizontally across disjoint inputs *within* a stage (e.g. in Figure 8, the Aggregate operators are executed concurrently) and vertically *across* stages not separated by pipeline breakers (e.g. in Figure 8, the Filter and Aggregate operator in the same lane can run simultaneously). To SELECT RegionID, avg(Latency) AS AvgLatency FROM hits WHERE URL = 'https://clickhouse.com' GROUP BY RegionID ORDER BY AvgLatency DESC LIMIT 3



Figure 8: A physical operator plan with three lanes.

avoid over and undersubscription when new queries start, or concurrent queries finish, the degree of parallelism can be changed midquery between one and the maximum number of worker threads for the query specified at query start (see Section 4.5).

Operators can further affect query execution at runtime in two ways. First, operators can dynamically create and connect new operators. This is mainly used to switch to external aggregation, sort, or join algorithms instead of canceling a query when the memory consumption exceeds a configurable threshold. Second, operators can request worker threads to move into an asynchronous queue. This provides more effective use of worker threads when waiting for remote data. ClickHouse's query execution engine and morsel-driven parallelism [44] are similar in that lanes are normally executed on different cores / NUMA sockets and that worker threads can steal tasks from other lanes. Also, there is no central scheduling component; instead, worker threads select their tasks individually by continuously traversing the operator plan. Unlike morsel-driven parallelism, ClickHouse bakes the maximum degree of parallelism into the plan and uses much bigger ranges to partition the source table compared to default morsel sizes of ca. 100.000 rows. While this may in some cases cause stalls (e.g. when the runtime of filter operators in different lanes differ vastly) we find that liberal use of exchange operators such as Repartition at least avoids such imbalances from accumulating across stages.

4.3 Multi-Node Parallelization

If the source table of a query is sharded, the query optimizer on the node that received the query (initiator node) tries to perform as much work as possible on other nodes. Results from other nodes can be integrated into different points of the query plan. Depending on the query, remote nodes may either 1. stream raw source table columns to the initiator node, 2. filter the source columns and send the surviving rows, 3. execute filter and aggregation steps and send local result groups with partial aggregation states, or 4. run the entire query including filters, aggregation, and sorting.

Node 2 ... N in Figure 8 show plan fragments executed on other nodes holding shards of the hits table. These nodes filter and group the local data and send the result to the initiator node. The GroupStateMerge operator on node 1 merges the local and remote results before the results groups are finally sorted.

4.4 Holistic Performance Optimization

This section presents selected key performance optimizations applied to different stages of query execution.

Query optimization. The first set of optimizations is applied on top of a semantic query representation obtained from the query's AST. Examples of such optimizations include constant folding (e.g. concat(lower('a'), upper('b')) becomes 'aB'), extracting scalars from certain aggregation functions (e.g. sum(a*2) becomes 2 * sum(a)), common subexpression elimination, and transforming disjunctions of equality filters to IN-lists (e.g. x=c OR x=d becomes x IN (c,d)). The optimized semantic query representation is subsequently transformed to a logical operator plan. Optimizations on top of the logical plan include filter pushdown, reordering function evaluation and sorting steps, depending on which one is estimated to be more expensive. Finally, the logical query plan is transformed into a physical operator plan. This transformation can exploit the particularities of the involved table engines. For example, in the case of a MergeTree*-table engine, if the ORDER BY columns form a prefix of the primary key, the data can be read in disk order, and sorting operators can be removed from the plan. Also, if the grouping columns in an aggregation form a prefix of the primary key, ClickHouse can use sort aggregation [33], i.e. aggregate runs of the same value in the pre-sorted inputs directly. Compared to hash aggregation, sort aggregation is significantly less memory-intensive, and the aggregate value can be passed to the next operator immediately after a run has been processed.

Query compilation. ClickHouse employs query compilation based on LLVM to dynamically fuse adjacent plan operators [38, 53]. For example, the expression a * b + c + 1 can be combined into a single operator instead of three operators. Besides expressions, ClickHouse also employs compilation to evaluate multiple aggregation functions at once (i.e. for GROUP BY) and for sorting with more than one sort key. Query compilation decreases the number of virtual calls, keeps data in registers or CPU caches, and helps the branch predictor as less code needs to execute. Additionally, runtime compilation enables a rich set of optimizations, such as logical optimizations and peephole optimizations implemented in compilers, and gives access to the fastest locally available CPU instructions. The compilation is initiated only when the same regular, aggregation, or sorting expression is executed by different queries more than a configurable number of times. Compiled query operators are cached and can be reused by future queries.⁷

Primary key index evaluation. ClickHouse evaluates WHERE conditions using the primary key index if a subset of filter clauses in the condition's conjunctive normal form constitutes a prefix of the primary key columns. The primary key index is analyzed left-to-right on lexicographically sorted ranges of key values. Filter clauses corresponding to a primary key column are evaluated using ternary logic - they are all true, all false, or mixed true/false for the values in the range. In the latter case, the range is split into sub-ranges which are analyzed recursively. Additional optimizations exist for functions in filter conditions. First, functions have traits describing their monotonicity, e.g, toDayOfMonth(date) is piecewise monotonic within a month. Monotonicity traits allow to infer if a function produces sorted results on sorted input key value ranges. Second, some functions can compute the preimage of a given function result. This is used to replace comparisons of constants with function calls on the key columns by comparing the key column value with the preimage. For example, toYear(k) =2024 can be replaced by k >= 2024-01-01 && k < 2025-01-01.

Data skipping. ClickHouse tries to avoid data reads at query runtime using the data structures presented in Section 3.2. Additionally, filters on different columns are evaluated sequentially in order of descending estimated selectivity based on heuristics and (optional) column statistics. Only data chunks that contain at least one matching row are passed to the next predicate. This gradually decreases the amount of read data and the number of computations to be performed from predicate to predicate. The optimization is only applied when at least one highly selective predicate is present; otherwise, the latency of the query would deteriorate compared to an evaluation of all predicates in parallel.

Hash tables. Hash tables are fundamental data structures for aggregation and hash joins. Choosing the right type of hash table is critical to performance. ClickHouse instantiates various hash tables (over 30 as of March 2024) from a generic hash table template with the hash function, allocator, cell type, and resize policy as variation points. Depending on the data type of the grouping columns, the estimated hash table cardinality, and other factors, the fastest hash table is selected for each query operator individually.⁸ Further optimizations implemented for hash tables include:

SELECT UserID, h1.Referer, h1.URL, h2.URL FROM hits AS h1 JOIN hits AS h2 ON h1.UserID = h2.UserID AND h1.URL = h2.Referer SETTINGS join_algorithm = 'parallel_hash'



Figure 9: Parallel hash join with three hash table partitions.

- a two-level layout with 256 sub-tables (based on the first byte of the hash) to support huge key sets,
- string hash tables [79] with four sub-tables and different hash functions for different string lengths,
- lookup tables which use the key directly as bucket index (i.e. no hashing) when there are only few keys,
- values with embedded hashes for faster collision resolution when comparison is expensive (e.g. strings, ASTs),
- creation of hash tables based on predicted sizes from runtime statistics to avoid unnecessary resizes,
- allocation of multiple small hash tables with the same creation/destruction lifecycle on a single memory slab,
- instant clearing of hash tables for reuse using per-hash-map and per-cell version counters,
- usage of CPU prefetch (__builtin_prefetch) to speed up the retrieval of values after hashing the key.

Joins. As ClickHouse originally supported joins only rudimentarily, many use cases historically resorted to denormalized tables. Today, the database offers all join types available in SQL (inner, left-/right/full outer, cross, as-of), as well as different join algorithms such as hash join (naïve, grace), sort-merge join, and index join for table engines with fast key-value lookup (usually dictionaries).⁹

Since joins are among the most expensive database operations, it is important to provide parallel variants of the classic join algorithms, ideally with configurable space/time trade-offs. For hash joins, ClickHouse implements the non-blocking, shared partition algorithm from [7]. For example, the query in Figure 9 computes how users move between URLs via a self-join on a page hit statistics table. The build phase of the join is split into three lanes, covering three disjoint ranges of the source table. Instead of a global hash table, a partitioned hash table is used. The (typically three) worker threads determine the target partition for each input row of the build side by computing the modulo of a hash function. Access to

⁷Blog post: clickhou.se/jit

⁸Blog post: clickhou.se/hashtables

⁹Blog post: clickhou.se/joins

the hash table partitions is synchronized using Gather exchange operators. The probe phase finds the target partition of its input tuples similarly. While this algorithm introduces two additional hash calculations per tuple, it greatly reduces latch contention in the build phase, depending on the number of hash table partitions.

4.5 Workload Isolation

ClickHouse offers concurrency control, memory usage limits, and I/O scheduling, enabling users to isolate queries into workload classes. By setting limits on shared resources (CPU cores, DRAM, disk and network I/O) for specific workload classes, it ensures these queries do not affect other critical business queries.

Concurrency control prevents thread oversubscription in scenarios with a high number of concurrent queries. More specifically, the number of worker threads per query are adjusted dynamically based on a specified ratio to the number of available CPU cores.

ClickHouse tracks byte sizes of memory allocations at the server, user, and query level, and thereby allows to set flexible memory usage limits. Memory overcommit enables queries to use additional free memory beyond the guaranteed memory, while assuring memory limits for other queries. Furthermore, memory usage for aggregation, sort, and join clauses can be limited, causing fallbacks to external algorithms when the memory limit is exceeded.

Lastly, I/O scheduling allows users to restrict local and remote disk accesses for workload classes based on a maximum bandwidth, in-flight requests, and policy (e.g. FIFO, SFC [32]).

5 INTEGRATION LAYER

Real-time decision-making applications often depend on efficient and low-latency access to data in multiple locations. Two approaches exist to make external data available in an OLAP database. With *push-based* data access, a third-party component bridges the database with external data stores. One example of this are specialized extract-transform-load (ETL) tools which push remote data to the destination system. In the *pull-based* model, the database itself connects to remote data sources and pulls data for querying into local tables or exports data to remote systems. While push-based approaches are more versatile and common, they entail a larger architectural footprint and scalability bottleneck. In contrast, remote connectivity directly in the database offers interesting capabilities, such as joins between local and remote data, while keeping the overall architecture simple and reducing the time to insight.

The rest of the section explores pull-based data integration methods in ClickHouse, aimed to access data in remote locations. We note that the idea of remote connectivity in SQL databases is not new. For example, the SQL/MED standard [35], introduced in 2001 and implemented by PostgreSQL since 2011 [65], proposes *foreign data wrappers* as a unified interface for managing external data. Maximum interoperability with other data stores and storage formats is one of ClickHouse's design goals. As of March 2024, ClickHouse offers to the best of our knowledge the most built-in data integration options across all analytical databases.

External Connectivity. ClickHouse provides 50+¹⁰ integration table functions and engines for connectivity with external systems and storage locations, including ODBC, MySQL, PostgreSQL,

SQLite, Kafka, Hive, MongoDB, Redis, S3/GCP/Azure object stores and various data lakes. We break them further down into categories.

Temporary access with Integration Table Functions. Table functions can be invoked in the FROM clause of SELECT queries to read remote data for exploratory ad-hoc queries. Alternatively, they can be used to write data to remote stores using INSERT INTO TABLE FUNCTION statements.

Persisted access. Three methods exist to create permanent connections with remote data stores and processing systems.

First, *integration table engines* represent a remote data source, such as a MySQL table, as a persistent local table. Users store the table definition using CREATE TABLE AS syntax, combined with a SELECT query and the table function. It is possible to specify a custom schema, for example, to reference only a subset of the remote columns, or use schema inference to determine the column names and equivalent ClickHouse types automatically. We further distinguish *passive* and *active* runtime behavior: Passive table engines forward queries to the remote system and populate a local proxy table with the result. In contrast, active table engines periodically pull data from the remote system or subscribe to remote changes, for example, through PostgreSQL's logical replication protocol. As a result, the local table contains a full copy of the remote table.

Second, *integration database engines* map all tables of a table schema in a remote data store into ClickHouse. Unlike the former, they generally require the remote data store to be a relational database and additionally provide limited support for DDL statements.

Third, *dictionaries* can be populated using arbitrary queries against almost all possible data sources with a corresponding integration table function or engine. The runtime behavior is active since data is pulled in constant intervals from remote storage.

Data Formats. To interact with 3rd party systems, modern analytical databases must also be able to process data in any format. Besides its native format, ClickHouse supports 90+¹¹ formats, including CSV, JSON, Parquet, Avro, ORC, Arrow, and Protobuf. Each format can be an input format (which ClickHouse can read), an output format (which ClickHouse can export), or both. Some analytics-oriented formats like Parquet are also integrated with query processing, i.e, the optimizer can exploit embedded statistics, and filters are evaluated directly on compressed data.

Compatibility interfaces. Besides its native binary wire protocol and HTTP, clients can interact with ClickHouse over MySQL or PostgreSQL wire-protocol-compatible interfaces. This compatibility feature is useful to enable access from proprietary applications (e.g. certain business intelligence tools), where vendors have not yet implemented native ClickHouse connectivity.

6 PERFORMANCE AS A FEATURE

This section presents built-in tools for performance analysis and evaluates the performance using real-world and benchmark queries.

6.1 Built-in Performance Analysis Tools

A wide range of tools is available to investigate performance bottlenecks in individual queries or background operations. Users interact with all tools through a uniform interface based on system tables.

¹⁰Up-to-date list live query over system table: clickhou.se/query-integrations

¹¹Up-to-date list live query over system table: clickhou.se/query-formats



Figure 10: Relative cold and hot runtimes of ClickBench.

Server and query metrics. Server-level statistics, such as the active part count, network throughput, and cache hit rates, are supplemented with per-query statistics, like the number of blocks read or index usage statistics. Metrics are calculated synchronously (upon request) or asynchronously at configurable intervals.

Sampling profiler. Callstacks of the server threads can be collected using a sampling profiler. The results can optionally be exported to external tools such as flamegraph visualizers.

OpenTelemetry integration. OpenTelemetry is an open standard for tracing data flows across multiple data processing systems [8]. ClickHouse can generate OpenTelemetry log spans with a configurable granularity for all query processing steps, as well as collect and analyze OpenTelemetry log spans from other systems.

Explain query. Like in other databases, SELECT queries can be preceded by EXPLAIN for detailed insights into a query's AST, logical and physical operator plans, and execution-time behavior.

6.2 Benchmarks

While benchmarking has been criticized for being not realistic enough [10, 52, 66, 74], it is still useful to identify the strengths and weaknesses of databases. In the following, we discuss how benchmarks are used to evaluate the performance of ClickHouse.

6.2.1 Denormalized Tables. Filter and aggregation queries on denormalized fact tables historically represent the primary use case of ClickHouse. We report runtimes of *ClickBench*, a typical workload of this kind that simulates ad-hoc and periodic reporting queries used in clickstream and traffic analysis. The benchmark consists of 43 queries against a table with 100 million anonymized page hits, sourced from one of the web's largest analytics platforms. An online dashboard [17] shows measurements (cold/hot runtimes, data import time, on-disk size) for over 45 commercial and research databases as of June 2024. Results are submitted by independent contributors based on the publicly available data set and queries [16]. The queries test sequential and index scan access paths and routinely expose CPU-, IO-, or memory-bound relational operators.

Figure 10 shows the total relative cold and hot runtimes for sequentially executing all ClickBench queries in databases frequently used for analytics. The measurements were taken on a single-node AWS EC2 c6a.4xlarge instance with 16 vCPUs, 32 GB RAM, and 5000 IOPS / 1000 MiB/s disk. Comparable systems were used for Redshift (ra3.4xlarge, 12 vCPUs, 96 GB RAM¹²) and Snowflake (warehouse size S: 2x8 vCPUs, 2x16 GB RAM¹³). The physical database design is tuned only lightly, for example, we specify primary keys, but do not change the compression of individual columns, create projections, or skipping indexes. We also flush the Linux page cache prior to each cold query run, but do not adjust database



¹³Blog post: clickhou.se/snowflake-sizes



Figure 11: Relative hot runtimes of VersionsBench 2018-2024.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7-Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20-Q2	2
llŀ	1.86		4.13		7.01	0.39		3.59	0.83	1.53		1.00	1.04	0.48		2.18			
*	2.20		2.10		1.90	0.23		4.30	1.30	0.88		0.65	0.77	1.90		3.40			
	С	onta	ins d	corn	elate	ed su	bauerie	s	Specific optimizations are not implemented yet										

Figure 12: Hot runtimes (in seconds) for TPC-H queries.

or operating system knobs. For every query, the fastest runtime across databases is used as a baseline. Relative query runtimes for other databases are calculated as $(t_q + 10ms)/(t_{q_baseline} + 10ms)$. The total relative runtime for a database is the geometric mean of the per-query ratios. While the research database Umbra [54] achieves the best overall hot runtime, ClickHouse outperforms all other production-grade databases for hot and cold runtimes.

To track the performance of SELECTs in more diverse workloads over time, we use a combination of four benchmarks called *VersionsBench* [19]. This benchmark is executed once per month when a new release is published to assess its performance [20] and identify code changes that potentially degraded performance¹⁴: Individual benchmarks include: 1. ClickBench (described above), 2. 15 MgBench [21] queries, 3. 13 queries against a denormalized Star Schema Benchmark [57] fact table with 600 million rows. 4. 4 queries against NYC Taxi Rides with 3.4 billion rows [70]¹⁵.

Figure 11 shows the development of the VersionsBench runtimes for 77 ClickHouse versions between March 2018 and March 2024. To compensate for differences in the relative runtime of individual queries, we normalize the runtimes using a geometric mean with the ratio to the minimum query runtime across all versions as weight. The performance of VersionBench improved by 1.72 × over the past six years. Dates for releases with long-term support (LTS) are marked on the x-axis. Although performance deteriorated temporarily in some periods, LTS releases generally have comparable or better performance than the previous LTS version. The significant improvement in August 2022 was caused by the column-by-column filter evaluation technique described in Section 4.4.

6.2.2 Normalized tables. In classical warehousing, data is often modeled using star or snowflake schemas. We present runtimes of TPC-H queries (scale factor 100) but remark that normalized tables are an emerging use case for ClickHouse. Figure 12 shows the hot runtimes of the TPC-H queries based on the parallel hash join algorithm described in Section 4.4. The measurements were taken on a single-node AWS EC2 c6i.16xlarge instance with 64 vCPUs, 128 GB RAM, and 5000 IOPS / 1000 MiB/s disk. The fastest of five runs

¹⁴Blog post: clickhou.se/performance-over-years

¹⁵Blog post: clickhou.se/nyc-taxi-rides-benchmark

was recorded. For reference, we performed the same measurements in a Snowflake system of comparable size (warehouse size L, 8x8 vCPUs, 8x16 GB RAM). The results of eleven queries are excluded from the table: Queries Q2, Q4, Q13, Q17, and Q20-22 include correlated subqueries which are not supported as of ClickHouse v24.6. Queries Q7-Q9 and Q19 depend on extended plan-level optimizations for joins such as join reordering and join predicate pushdown (both missing as of ClickHouse v24.6.) to achieve viable runtimes. Automatic subquery decorrelation and better optimizer support for joins are planned for implementation in 2024 [18]. Out of the remaining 11 queries, 5 (6) queries executed faster in ClickHouse (Snowflake). As aforementioned optimizations are known to be critical for performance [27], we expect them to improve runtimes of these queries further once implemented.

7 RELATED WORK

Analytical databases have been of great academic and commercial interest in recent decades [1]. Early systems like Sybase IQ [48], Teradata [72], Vertica [42], and Greenplum [47] were characterized by expensive batch ETL jobs and limited elasticity due to their on-premise nature. In the early 2010s, the advent of cloud-native data warehouses and database-as-a-service offerings (DBaaS) such as Snowflake [22], BigQuery [49], and Redshift [4] dramatically reduced the cost and complexity of analytics for organizations, while benefiting from high availability and automatic resource scaling. More recently, analytical execution kernels (e.g. Photon [5] and Velox [62]) offer commodified data processing for use in different analytical, streaming, and machine learning applications.

The most similar databases to ClickHouse, in terms of goals and design principles, are Druid [78] and Pinot [34]. Both systems target real-time analytics with high data ingestion rates. Like ClickHouse, tables are split into horizontal parts called segments. While ClickHouse continuously merges smaller parts and optionally reduces data volumes using the techniques in Section 3.3, parts remain forever immutable in Druid and Pinot. Also, Druid and Pinot require specialized nodes to create, mutate, and search tables, whereas ClickHouse uses a monolithic binary for these tasks.

Snowflake [22] is a popular proprietary cloud data warehouse based on a shared-disk architecture. Its approach of dividing tables into micro-partitions is similar to the concept of parts in ClickHouse. Snowflake uses hybrid PAX pages [3] for persistence, whereas ClickHouse's storage format is strictly columnar. Snowflake also emphasizes local caching and data pruning using automatically created lightweight indexes [31, 51] as a source for good performance. Similar to primary keys in ClickHouse, users may optionally create clustered indexes to co-locate data with the same values.

Photon [5] and Velox [62] are query execution engines designed to be used as components in complex data management systems. Both systems are passed query plans as input, which are then executed on the local node over Parquet (Photon) or Arrow (Velox) files [46]. ClickHouse is able to consume and generate data in these generic formats but prefers its native file format for storage. While Velox and Photon do not optimize the query plan (Velox performs basic expression optimizations), they utilize runtime adaptivity techniques, such as dynamically switching compute kernels depending on the data characteristics. Similarly, plan operators in ClickHouse can create other operators at runtime, primarily to switch to external aggregation or join operators, based on the query memory consumption. The Photon paper notes that code-generating designs [38, 41, 53] are harder to develop and debug than interpreted vectorized designs [11]. The (experimental) support for code generation in Velox builds and links a shared library produced from runtime-generated C++ code, whereas ClickHouse interacts directly with LLVM's on-request compilation API.

DuckDB [67] is also meant to be embedded by a host process, but additionally provides query optimization and transactions. It was designed for OLAP queries mixed with occasional OLTP statements. DuckDB accordingly chose the DataBlocks [43] storage format, which employs light-weight compression methods such as order-preserving dictionaries or frame-of-reference[2] to achieve good performance in hybrid workloads. In contrast, ClickHouse is optimized for append-only use cases, i.e. no or rare updates and deletes. Blocks are compressed using heavy-weight techniques like LZ4, assuming that users make liberal use of data pruning to speed up frequent queries and that I/O costs dwarf decompression costs for the remaining queries. DuckDB also provides serializable transactions based on Hyper's MVCC scheme [55], whereas ClickHouse only offers snapshot isolation.

8 CONCLUSION AND OUTLOOK

We presented the architecture of ClickHouse, an open-source, highperformance OLAP database. With a write-optimized storage layer and a state-of-the-art vectorized query engine at its foundation, ClickHouse enables real-time analytics over petabyte-scale data sets with high ingestion rates. By merging and transforming data asynchronously in the background, ClickHouse efficiently decouples data maintenance and parallel inserts. Its storage layer enables aggressive data pruning using sparse primary indexes, skipping indexes, and projection tables. We described ClickHouse's implementation of updates and deletes, idempotent inserts, and data replication across nodes for high availability. The query processing layer optimizes queries using a wealth of techniques, and parallelizes execution across all server and cluster resources. Integration table engines and functions provide a convenient way to interact with other data management systems and data formats seamlessly. Through benchmarks, we demonstrate that ClickHouse is amongst the fastest analytical databases on the market, and we showed significant improvements in the performance of typical queries in real-world deployments of ClickHouse throughout the years.

All features and enhancements planned for 2024 can be found on the public roadmap [18]. Planned improvements include support for user transactions, PromQL [69] as an alternative query language, a new datatype for semi-structured data (e.g. JSON), better plan-level optimizations of joins, as well as an implementation of light-weight updates to complement light-weight deletes.

ACKNOWLEDGMENTS

As per version 24.6, SELECT * FROM system.contributors returns 1994 individuals who contributed to ClickHouse. We would like to thank the entire engineering team at ClickHouse Inc. and ClickHouse's amazing open-source community for their hard work and dedication in building this database together.

REFERENCES

- [1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreaos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. https://doi.org/10.1561/9781601987556
- [2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06). 671-682. https://doi.org/10.1145/1142473.1142548
- [3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 169-180.
- [4] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-Invented. In Proceedings of the 2022 International Conference on Management of Data (Philadelphia, PA, USA) (SIGMOD '22). Association for Computing Machinery, New York, NY, USA, 2205-2217. https://doi.org/10.1145/3514221.3526045
- [5] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems (SIGMOD '22). Association for Computing Machinery, New York, NY, USA, 2326-2339. https://doi.org/10.1145/3514221. 3526054
- [6] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. ACM Computing Survey 13, 2 (1981), 185–221. https://doi.org/10.1145/356842.356846
- Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of [7] main memory hash join algorithms for multi-core CPUs. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (Athens. Greece) (SIGMOD '11). Association for Computing Machinery, New York, NY, USA, 37-48. https://doi.org/10.1145/1989323.1989328
- [8] Daniel Gomez Blanco. 2023. Practical OpenTelemetry. Springer Nature.
- Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable [9] Errors. Commun. ACM 13, 7 (1970), 422-426. https://doi.org/10.1145/362686. 362692
- [10] Peter Boncz, Thomas Neumann, and Orri Erling. 2014. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In Performance Characterization and Benchmarking. 61-76. https://doi.org/10.1007/978-3-319-04936-6 5
- [11] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In CIDR.
- [12] Martin Burtscher and Paruj Ratanaworabhan. 2007. High Throughput Compression of Double-Precision Floating-Point Data. In Data Compression Conference (DCC). 293-302. https://doi.org/10.1109/DCC.2007.44
- [13] Jeff Carpenter and Eben Hewitt. 2016. Cassandra: The Definitive Guide (2nd ed.). O'Reilly Media, Inc.
- [14] Bernadette Charron-Bost, Fernando Pedone, and André Schiper (Eds.). 2010. Replication: Theory and Practice. Springer-Verlag.
- [15] chDB. 2024. chDB an embedded OLAP SQL Engine. Retrieved 2024-06-20 from https://github.com/chdb-io/chdb
- [16] ClickHouse. 2024. ClickBench: a Benchmark For Analytical Databases. Retrieved 2024-06-20 from https://github.com/ClickHouse/ClickBench
- [17] ClickHouse. 2024. ClickBench: Comparative Measurements. Retrieved 2024-06-20 from https://benchmark.clickhouse.com
- [18] ClickHouse. 2024. ClickHouse Roadmap 2024 (GitHub). Retrieved 2024-06-20 from https://github.com/ClickHouse/ClickHouse/issues/58392
- [19] ClickHouse. 2024. ClickHouse Versions Benchmark. Retrieved 2024-06-20 from https://github.com/ClickHouse/ClickBench/tree/main/versions
- [20] ClickHouse. 2024. ClickHouse Versions Benchmark Results. Retrieved 2024-06-20 from https://benchmark.clickhouse.com/versions/
- [21] Andrew Crotty. 2022. MgBench. Retrieved 2024-06-20 from https://github.com/ andrewcrotty/mgbench
- [22] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In Proceedings of the 2016 International Conference on Management of Data (San Francisco, California, USA) (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 215-226. https: //doi.org/10.1145/2882903.2903741 Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolf-
- [23] gang Lehner. 2019. From a Comprehensive Experimental Survey to a Cost-Based

Selection Strategy for Lightweight Integer Compression Algorithms. ACM Trans. Database Syst. 44, 3, Article 9 (2019), 46 pages. https://doi.org/10.1145/3323991

- [24] Philippe Dobbelaere and Kyumars Sheykh Esmaili. 2017. Kafka versus RabbitMQ: A Comparative Study of Two Industry Reference Publish/Subscribe Implementations: Industry Paper (DEBS '17). Association for Computing Machinery, New York, NY, USA, 227-238. https://doi.org/10.1145/3093742.3093908
- LLVM documentation. 2024. Auto-Vectorization in LLVM. Retrieved 2024-06-20 [25] from https://llvm.org/docs/Vectorizers.html
- Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: [26] Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. ACM Transactions on Storage 17, 4, Article 26 (2021), 32 pages. https://doi.org/10.1145/3483840
- [27] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H choke points and their optimizations. Proc. VLDB Endow. 13, 8 (2020), 1206-1220. https://doi.org/10.14778/3389133.3389138
- [28] Ted Dunning. 2021. The t-digest: Efficient estimates of distributions. Software Impacts 7 (2021). https://doi.org/10.1016/j.simpa.2020.100049
- [29] Martin Faust, Martin Boissier, Marvin Keller, David Schwalb, Holger Bischoff, Katrin Eisenreich, Franz Färber, and Hasso Plattner. 2016. Footprint Reduction and Uniqueness Enforcement with Hash Indices in SAP HANA. In Database and Expert Systems Applications. 137-151. https://doi.org/10.1007/978-3-319-44406-2 11
- [30] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In AofA: Analysis of Algorithms, Vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07). Discrete Mathematics and Theoretical Computer Science, 137-156. https://doi.org/10.46298/dmtcs.3545
- [31] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2009. Database Systems - The Complete Book (2. Ed.).
- [32] Pawan Goyal, Harrick M. Vin, and Haichen Chen. 1996. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. 26, 4 (1996), 157-168, https://doi.org/10.1145/248157.248171
- Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. ACM [33] Comput. Surv. 25, 2 (1993), 73-169. https://doi.org/10.1145/152610.152611
- [34] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, and Ravi Aringunram. 2018. Pinot: Realtime OLAP for 530 Million Users. In Proceedings of the 2018 International Conference on Management of Data (Houston, TX, USA) (SIGMOD '18). Association for Computing Machinery, New York, NY, USA, 583-594. https://doi.org/10.1145/3183713.3190661
- [35] ISO/IEC 9075-9:2001 2001. Information technology Database language SQL Part 9: Management of External Data (SQL/MED). Standard. International Organization for Standardization.
- Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica, and Matei [36] Zaharia. 2023. Analyzing and Comparing Lakehouse Storage Systems. CIDR.
- Project Jupyter. 2024. Jupyter Notebooks. Retrieved 2024-06-20 from https: [37] //jupyter.org/
- [38] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. Proc. VLDB Endow. 11, 13 (sep 2018), 2209-2222. https://doi.org/10.14778/3275366.3284966
- [39] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. 2010. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (Indianapolis, Indiana, USA) (SIGMOD '10). Association for Computing Machinery, New York, NY, USA, 339-350. https://doi.org/10.1145/1807167.1807206
- [40] Donald E. Knuth. 1973. The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley.
- [41] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In 2018 IEEE 34th International Conference on Data Engineering (ICDE). 197-208. https://doi.org/10.1109/ICDE.2018.00027
- Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, [42] Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. Proc. VLDB Endow. 5, 12 (aug 2012), 1790-1801. https://doi.org/10. 14778/2367502.2367518
- [43] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In Proceedings of the 2016 International Conference on Management of Data (San Francisco, California, USA) (SIGMOD '16). Association for Computing Machinery, New York, NY, USA, 311-326. https://doi.org/10.1145/2882903.2882925
- [44] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morseldriven parallelism: a NUMA-aware query evaluation framework for the manycore age. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (Snowbird, Utah, USA) (SIGMOD '14). Association for Computing Machinery, New York, NY, USA, 743-754. https://doi.org/10.1145/2588555. 2610507

- [45] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In 2013 IEEE 29th International Conference on Data Engineering (ICDE). 38–49. https://doi.org/10.1109/ICDE. 2013.6544812
- [46] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A Deep Dive into Common Open Formats for Analytical DBMSs. 16, 11 (jul 2023), 3044–3056. https://doi.org/10.14778/3611479.3611507
- [47] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads (SIGMOD '21). Association for Computing Machinery, New York, NY, USA, 2530–2542. https: //doi.org/10.1145/3448016.3457562
- [48] Roger MacNicol and Blaine French. 2004. Sybase IQ Multiplex Designed for Analytics. In Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30 (Toronto, Canada) (VLDB '04). VLDB Endowment, 1227–1230.
- [49] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568
- [50] Microsoft. 2024. Kusto Query Language. Retrieved 2024-06-20 from https: //github.com/microsoft/Kusto-Query-Language
- [51] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98). 476–487.
- [52] Jalal Mostafa, Sara Wehbi, Suren Chilingaryan, and Andreas Kopmann. 2022. SciTS: A Benchmark for Time-Series Databases in Scientific Experiments and Industrial Internet of Things. In Proceedings of the 34th International Conference on Scientific and Statistical Database Management (SSDBM '22). Article 12. https: //doi.org/10.1145/3538712.3538723
- [53] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. Proc. VLDB Endow. 4, 9 (jun 2011), 539–550. https://doi.org/10.14778/ 2002938.2002940
- [54] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumanncidr20.pdf
- [55] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (Melbourne, Victoria, Australia) (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 677–689. https://doi.org/10.1145/2723372. 2749436
- [56] LevelDB on GitHub. 2024. LevelDB. Retrieved 2024-06-20 from https://github. com/google/leveldb
- [57] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking*. Springer Berlin Heidelberg, 237–252. https: //doi.org/10.1007/978-3-642-10424-4_17
- [58] Patrick E. O'Neil, Edward Y. C. Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured Merge-Tree (LSM-tree). Acta Informatica 33 (1996), 351–385. https://doi.org/10.1007/s002360050048
- [59] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14). 305–320. https://doi.org/doi/10. 5555/2643634.2643666
- [60] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). Acta Inf. 33, 4 (1996), 351–385. https: //doi.org/10.1007/s002360050048
- [61] Pandas. 2024. Pandas Dataframes. Retrieved 2024-06-20 from https://pandas. pydata.org/

- [62] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. Proc. VLDB Endow. 15, 12 (aug 2022), 3372–3384. https: //doi.org/10.14778/3554821.3554829
- [63] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, in-Memory Time Series Database. Proceedings of the VLDB Endowment 8, 12 (2015), 1816–1827. https://doi.org/10.14778/2824032.2824078
- [64] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). 1493–1508. https://doi.org/10.1145/2723372.2747645
- [65] PostgreSQL. 2024. PostgreSQL Foreign Data Wrappers. Retrieved 2024-06-20 from https://wiki.postgresql.org/wiki/Foreign_data_wrappers
 [66] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. 2018. Fair
- [66] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. 2018. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In Proceedings of the Workshop on Testing Database Systems (Houston, TX, USA) (DBTest'18). Article 2, 6 pages. https://doi.org/10.1145/3209950.3209955
- [67] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database (*SIGMOD '19*). Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212
 [68] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-
- [68] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99). San Francisco, CA, USA, 78–89.
- [69] Navin C. Sabharwal and Piyush Kant Pandey. 2020. Working with Prometheus Query Language (PromQL). In Monitoring Microservices and Containerized Applications. https://doi.org/10.1007/978-1-4842-6216-0_5
- [70] Todd W. Schneider. 2022. New York City Taxi and For-Hire Vehicle Data. Retrieved 2024-06-20 from https://github.com/toddwschneider/nyc-taxi-data
- [71] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05). 553–564.
- [72] Teradata. 2024. Teradata Database. Retrieved 2024-06-20 from https://www. teradata.com/resources/datasheets/teradata-database
- [73] Frederik Transier. 2010. Algorithms and Data Structures for In-Memory Text Search Engines. Ph.D. Dissertation. https://doi.org/10.5445/IR/1000015824
- [74] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In Proceedings of the Workshop on Testing Database Systems (Houston, TX, USA) (DBTest'18). Article 1, 6 pages. https://doi.org/10.1145/3200950.320952
- [75] LZ4 website. 2024. LZ4. Retrieved 2024-06-20 from https://lz4.org/
- [76] PRQL website. 2024. PRQL. Retrieved 2024-06-20 from https://prql-lang.org
 [77] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000.
- The Implementation and Performance of Compressed Databases. *SIGMOD Rec.* 29, 3 (sep 2000), 55–67. https://doi.org/10.1145/362084.362137
- [78] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A Real-Time Analytical Data Store. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (Snowbird, Utah, USA) (SIGMOD '14). Association for Computing Machinery, New York, NY, USA, 157-168. https://doi.org/10.1145/2588555.2595631
- [79] Tianqi Zheng, Zhibin Zhang, and Xueqi Cheng. 2020. SAHA: A String Adaptive Hash Table for Analytical Databases. *Applied Sciences* 10, 6 (2020). https: //doi.org/10.3390/app10061915
- [80] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02). 145–156. https://doi.org/10. 1145/564691.564709
- [81] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In Proceedings of the 22nd International Conference on Data Engineering (ICDE '06). 59. https://doi.org/10.1109/ICDE. 2006.150