ChronicleDB: A High-Performance Event Store

MARC SEIDEMANN, NIKOLAUS GLOMBIEWSKI, MICHAEL KÖRBER, and
BERNHARD SEEGER, Philipps-University Marburg, Germany

Reactive security monitoring, self-driving cars, the Internet of Things (IoT), and many other novel applications require systems for both writing events arriving at very high and fluctuating rates to persistent storage as well as supporting analytical ad hoc queries. As standard database systems are not capable of delivering the required write performance, log-based systems, key-value stores, and other write-optimized data stores have emerged recently. However, the drawbacks of these systems are a fair query performance and the lack of suitable instant recovery mechanisms in case of system failures.

In this article, we present ChronicleDB, a novel database system with a storage layout tailored for high write performance under fluctuating data rates and powerful indexing capabilities to support a variety of queries. In addition, ChronicleDB offers low-cost fault tolerance and instant recovery within milliseconds. Unlike previous work, ChronicleDB is designed either as a serverless library to be tightly integrated in an application or as a standalone database server. Our results of an experimental evaluation with real and synthetic data reveal that ChronicleDB clearly outperforms competing systems with respect to both write and query performance.

CCS Concepts: • **Information systems → Physical data models**; **Data streams**; **Temporal data**; **Record and block layout**; **Stream management**; *Data compression*;

Additional Key Words and Phrases: Event processing, indexing, aggregation queries, time travel queries, storage layout, recovery

ACM Reference format:
Marc Seidemann, Nikolaus Glombiewski, Michael Körber, and Bernhard Seeger. 2019. ChronicleDB: A High-Performance Event Store. *ACM Trans. Database Syst.* 44, 4, Article 13 (October 2019), 45 pages.
https://doi.org/10.1145/3342357

1 INTRODUCTION

Data objects in time, also known as events, are ubiquitous in today's information landscape. They arise at lower levels in the context of operating systems such as file accesses, CPU usage, or network packets, but also at higher levels, for example, in the context of online shopping transactions. The rapidly growing *Internet of Things (IoT)* is reinforcing the new challenge for present-day data processing. Sensor-equipped devices are becoming omnipresent in companies, smart buildings, airplanes, and self-driving cars. Furthermore, scientific observational (sensor) data is crucial in various research, spanning from climate research over animal tracking to IT security monitoring.

This work has been supported by the German Research Foundation (DFG) under grant no. SE 553/9-1.
Authors' addresses: M. Seidemann, N. Glombiewski, M. Körber, and B. Seeger, Philipps-University Marburg, Hans-Meerwein Staße, Marburg, 35032, Germany; emails: {seidemann, glombien, koerberm, seeger}@mathematik.uni-marburg.de.

0362-5915/2019/10-ART13 $15.00
https://doi.org/10.1145/3342357

All these applications rely on low-latency processing of events attached with one or multiple temporal attributes.

Due to the rapidly increasing number of sensors, not only the analysis of such events, but also their pure ingestion is becoming a big challenge. On-the-fly event stream processing is not always the outright solution. Many fields of applications require maintaining the collected historical data as time series for the long term, e.g., for further temporal analysis or provenance reasons. For example, in the field of IT security, historical data is crucial to reproduce critical security incidents and to derive new security patterns. This requires writing various sensor measurements arriving at high and fluctuating speed to persistent storage. Data loss due to system failures or system overload is generally not acceptable, as these data sets are of utmost importance in operational applications.

Currently, there is a lack of systems supporting the above-described workload scenario (write-intensive, ad hoc temporal queries, and fault-tolerance). Standard database systems are not designed for supporting such write-intensive workloads. Their separation of data and transaction logs generally incurs high overheads. Our experiments with traditional relational systems revealed that their insertion performance is insufficient to keep up with the targeted data rates. PostgreSQL [12], for example, managed only about 10K tuple insertions per second. Relational database systems are designed to store data with the focus on query processing and transactional safety. Instead of relational systems, today, it is common to use distributed key-value systems such as Cassandra [4] or HBase [8], but they only alleviate the write problem at a very high cost. In our benchmarks, our event store *ChronicleDB* outperformed Cassandra by a factor of over 200 in terms of write performance on a single node. In other words: Cassandra would need at least 200 machines to compete with our solution. Apart from economical aspects due to high expenses for large clusters, there are many embedded systems where scalable distributed storage is not the outright solution. For example, in [16], virtual machines of a physical server are monitored within a central monitoring virtual machine, which does not allow a distributed storage solution due to security reasons. Other examples are self-driving cars and airplanes that need to manage huge data rates within a local system.

To overcome these deficiencies, particularly the poor write performance, log-only systems have emerged where the log is the only data repository [18, 43, 44]. Our system ChronicleDB also keeps its data in a log, but it differs from previous approaches by its centralized design for high volume event streams. Because there are often only modest changes in event streams, ChronicleDB exploits the great potential of their lossless compression to boost write and read performance beyond that of previous log-only approaches. This also requires the design of a novel storage layout to achieve fault tolerance and near-instant recovery within milliseconds in case of a system failure. In addition to lightweight temporal indexing, ChronicleDB offers adaptive indexing support to significantly speed up non-temporal queries on its log. ChronicleDB can either be plugged into applications as a library or run as a centralized system without the necessity to use the common distributed storage stack. In summary, we make the following contributions:

- We propose an efficient and robust storage layout for compressed data with fault tolerance and instant recovery.
- ChronicleDB offers an adaptive indexing technique comprising both lightweight temporal indexing as well as full secondary indexing to speed up queries on non-temporal dimensions.
- To support out-of-order arrival of events, we developed a hybrid logging approach between our log storage and traditional logging.
- For preserving a consistently high ingestion rate to avoid data loss, we identify key metrics and devise a load scheduling approach tailor-made for ChronicleDB's components.

- ChronicleDB features the execution of continuous queries from event stream processing systems. This feature targets plenty of applications, including failure recovery or deferred query execution during peak load times. By exploiting ChronicleDB's lightweight indexing, we are able to reduce the processing time of continuous aggregation and pattern matching queries by orders of magnitude compared to a pure replay-based approach.
- We compare the performance of ChronicleDB with commercial, open-source, and academic systems in our experiments using real event data.

This article is an extended version of [38]. The extension includes the introduction of materialized user-defined aggregates, a continuous query processing component, and an extended load scheduling approach ensuring high ingestion rates under fluctuating data rates and high fractions of out-of-order arrivals.

The remainder of this article is structured as follows: Section 2 discusses related work and proposes related solutions. In Section 3, we give a brief overview of the system architecture. Section 4 addresses ChronicleDB's storage layout; Section 5 discusses its indexing approach. Section 6 details ChronicleDB's load scheduler. Recovery issues are examined in Section 7, and Section 8 discusses continuous query processing. Before we conclude in Section 10, we evaluate our system experimentally in Section 9.

2 RELATED WORK

Our discussion of related work is structured as follows: At first, we present data stores relating to ChronicleDB. Then, we discuss previous work referring to our indexing techniques. Finally, we discuss related techniques for dealing with traditional event stream queries.

Data Stores. The domain of ChronicleDB partly relates to different types of storage systems, including data warehouse, event log processing, as well as temporal database systems.

One of the first solutions explicitly addressing event data is *DataDepot* [24]. DataDepot is a data warehouse for streaming data, running on top of a relational system. DataDepot addresses out-of-order processing and various distributed aspects of handling streaming data, e.g., partitioning. However, due to its basis in a relational system, it cannot compete with the throughput of modern systems. *Tidalrace* [26], the successor of DataDepot, pursues a distributed storage approach while getting rid of the DataDepot legacy code base. The system improves the overall ingestion rate but cannot compete with ChronicleDB's throughput on a single system, and, just like DataDepot, lacks optimizations for event queries that ChronicleDB offers. *DataGarage* [31] is a data warehouse designed for managing performance data on commodity servers, which consists of several relational databases stored on a distributed file system. Similar to ChronicleDB, DataGarage addresses aggregation and deletion of outdated events. However, DataGarage is by design a scalable distributed system that is not designed to run as a library tightly integrated in the application code. In addition, DataGarage does not address high ingestion rates.

Popular NoSQL systems in the context of storing events are Cassandra [4] and HBase [8]. Similiar to the systems mentioned above, they are designed for scalability rather than optimizing single machine performance. Due to their support for a wider variety of use cases, they do not offer explicit support for event data, i.e., strong temporal correlation and specialized index utilization for queries. A representative for log storage systems is *LogBase* [43], which is also applied for event log processing. In contrast to our approach, LogBase is designed as a general-purpose database, also applicable for media data like photos. LogBase is build on top of HDFS [7] and writes data sequentially to log files. The authors use an index similar to B^{link}-trees, augmented with compound keys (key, timestamp) to index the data in an in-memory multi-version index. However, they do not consider lightweight index support to query event data. *LogKV* [18] utilizes distributed key-value stores to process event log files with Cassandra [4] as an underlying key-value store. The authors provide a solution to balance worker nodes and deal with bursty streams in a distributed environment, and briefly sketch algorithms for windowed join and filter operations. However, they leave aggregation queries to MapReduce frameworks and do not discuss pattern matching.

Another class of storage solutions ChronicleDB partly relates to are time series databases, which address temporal correlation and compression but usually do not offer query support for typical event-processing operations like pattern matching. A representative of this class is *tsdb* [23]. Similar to our approach, the authors use LZ4 for lossless data compression. In contrast to our approach, time series databases (including tsdb) usually assume a uniform division of the time axis in which the data arrives at each time unit. *Gorilla* [36] is a main-memory time series system on top of HBase with support for ad hoc query processing. The authors propose a compression technique for uni-variate events of continuous event streams. ChronicleDB also supports multi-variate events. OpenTSDB [11] and KairosDB [3] are time series database systems on top of HBase and Cassandra. Thus, they have the same deficiencies as their underlying systems. The most related storage system is *InfluxDB* [9], an open-source time series database solution. Unlike ChronicleDB, InfluxDB is not optimized for single machine data storage but is designed from the ground up as a distributed system. Furthermore, in its current state, it is more suited for uni-variate time series data and, just like other time series systems, has no support for CEP-style queries.

Lightweight Indexing. ChronicleDB's lightweight indexing approach maintains small sets of aggregates per attribute (min,max,count,sum) on different levels of its primary index to summarize the stored event data for different time granularities. Aggregation in the context of temporal databases has been extensively investigated in the database community. Widom et al. [46] proposed the SB-tree for partial temporal aggregates. The SB-tree shares some common characteristics with our indexing approach TAB$^+$-tree. But unlike our approach, a SB-tree only maintains the aggregates for a single attribute. Moerkotte introduced Small Materialized Aggregates (SMA) [33] to speed up query processing in data warehouses. SMAs store a sequence of aggregations in a separate file on disk, whereby each aggregate value corresponds to a sequence of consecutive tuples (e.g., a page). In contrast, ChronicleDB stores these aggregates within its primary index and thus omits random writes introduced by separate index files. Further, we manage aggregates hierarchically and this way provide summarizations of different granularities instead of only a single one. More recent research concentrates on observational data and event data. The recently proposed CR-index by Wang et al. [44] is implemented in LogBase [43] and exploits temporal correlation of data. It maintains a separate index per attribute on its minimum/maximum intervals within data blocks. But instead of creating a separate index for each attribute, ChronicleDB keeps all secondary information within the primary index, and thus omits random writes during index creation. In addition, queries on multiple attributes do not need to access multiple indexes. ChronicleDB computes these aggregates incrementally as new events arrive. The applied techniques are closely related to sliding window aggregation in data stream processing systems [28, 29, 39]—especially the *Reactive Aggregator* [39] inspired ChronicleDB's lightweight indexing, since it manages partial temporal aggregates inside a tree structure and supports non-invertible aggregates such as min/max.

Event Stream Processing. ChronicleDB is designed to run side-by-side with event stream processing systems such as Esper[1] or Apache Flink [19]. It provides the ability to re-run continuous

[1]http://www.espertech.com/esper/.

queries on historical event data and boost their performance via indexes. In this work, we focus on two important operations: sliding window aggregation and pattern matching.

Sliding window aggregations are typically computed in two phases. The first phase computes partial aggregates for a certain number of events, and the second phase uses these partial aggregates to assemble the result for the whole window. For example, [29] computes partial aggregates for continuous fixed size sub-sequences of the data stream, while [20] starts a new partial aggregate at the beginning of new windows. Likewise, various approaches for the second phase exist using different data structures to manage and merge partial aggregates (for example, binary trees [39] or queues [40]). However, to the best of our knowledge, there is no work on efficiently supporting sliding window aggregations on persistent storage.

Pattern matching on event streams is also a two-step process. First the incoming events are mapped to symbols using Boolean expressions on the events' attributes before the resulting symbol stream is matched using some kind of regular expression. In contrast to traditional regular expression matching, most systems featuring event pattern matching provide advanced features such as accessing previously matched events or variable bindings (i.e., bind values of a specific event to a variable and use this value in subsequent symbol definitions). There are various approaches for pattern matching using NFAs [22, 45] or trees [22], all featuring their respective unique optimization techniques. However, regardless of the approach, these solutions are specialized for on-the-fly stream processing and thus miss optimization opportunities provided by indexed persistent data. Reference [42] is the only index supported pattern matching approach we are aware of. In this work, sets of trajectories from moving objects are matched according to a user-defined pattern. In contrast to event pattern matching, the authors are not looking for occurrences of a pattern within a stream of data, but filter out trajectories not matching the whole pattern. Hence, their approach does not support time-window constraints. Similar to our approach, they simplify the pattern and perform index lookups on certain symbols. These results are used to exclude whole trajectories from the matching process. However, all available indexes are used for processing and thus, neither a symbol's selectivity nor contiguous symbol sequences are considered, which can hurt processing performance considerably.

3 SYSTEM ARCHITECTURE

This section introduces the general architecture of ChronicleDB. At first, we present our requirements on the system. Afterwards, its main components are introduced. Finally, we discuss the main features of ChronicleDB and its fundamental design principles.

3.1 Requirements

ChronicleDB aims at supporting temporal-relational events, which consist of a timestamp t and several non-temporal, primitive attributes a_i. Sequences of events (*streams*) can be considered as multi-variate time series, but with non-equidistant timestamps. Timestamps can either refer to system time (when the event occurred at the system) or application time (when the event occurred in the application). The latter is more meaningful to temporal queries on the application level, and thus, our goal is to maintain a physical order on application time.

Our main objective is fast writing to keep up with high and fluctuating event rates. ChronicleDB should be as economical as possible with regard to storage utilization to store data for the long term, i.e., months or years. Therefore, we aimed at a centralized storage system for cheap disks running as an embedded storage solution within a system (e.g., a self-driving car).

Generally, we assume events to arrive chronologically. Events are inserted into the system **once** and are (possibly) deleted once. In the meantime, there are no updates on an event. However, we also want to support occasional out-of-order insertions, as they typically occur in event-based

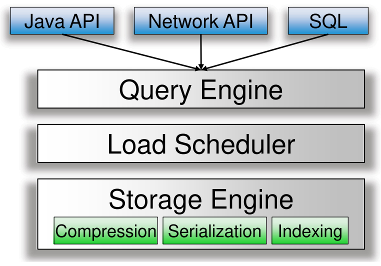


Fig. 1. ChronicleDB architecture.

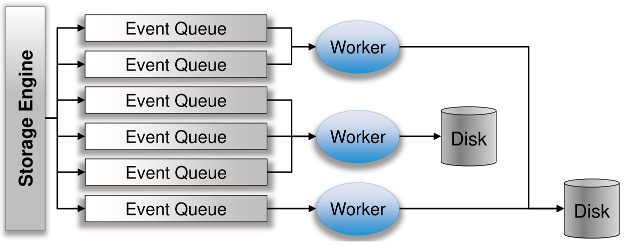


Fig. 2. Example of a ChronicleDB topology.

applications [15]. They can happen, e.g., if the clocks of the sensors are skewed or simply due to communication problems.

The most important types of queries the system has to support are *time travel queries* and *temporal aggregation queries*. Time travel queries allow requests for specific points and ranges in time, e.g., *all ssh login attempts within the last hour*. Temporal aggregation queries give a comprehensive overview of the data, e.g., *the average number of ssh logins for each day of the week during the last three months*. In addition, the system should efficiently support queries on non-temporal attributes, e.g., *all ssh logins within the last day from a certain IP range*, and also be capable to provide efficient, index-based replay of common event-processing queries like pattern matching (cf. Section 8).

3.2 Architecture Overview

Figure 1 depicts a high-level overview of ChronicleDB's architecture. In this article, we focus on the lower layer, i.e., the storage engine and the indexing capabilities of ChronicleDB, and provide an introduction into our ingestion-based load scheduler.

The storage engine of ChronicleDB logically consists of three components: event queues, workers, and disks. Event queues are memory-based FIFO queues, decoupling event producers, and ChronicleDB. We maintain one event queue per-event stream, each one processed by exactly one worker thread. Worker threads extract events from the in-memory queues, handle serialization and compression, and finally store them on the configured disk. Figure 2 depicts an example configuration of ChronicleDB. The system processes six event streams and hence manages six event queues. These queues are assigned to three worker threads, which in turn store the event data on two different disks.

The architecture of ChronicleDB is sufficiently flexible to take into account workload characteristics as well as the available system resources. The task of the load scheduler is to determine the configuration settings and assign workers accordingly.

3.3 Design Principles & System Features

In ChronicleDB, the major design principle is: *the log is the database*. We avoid additional logging, as the considered append-only scenario does not cause costly random I/Os, and therefore it does not incur buffering strategies with no-force writes. Only in case of out-of-order arrival of events do we have to deviate from this paradigm, as we want to keep the data ordered with respect to application time. We explicitly handle out-of-order events by preserving a certain amount of spare space inside the pages written to disk. This way, late events can be absorbed without splitting existing nodes. However, if the spare space is exhausted or the out-of-order rate is to high, the number of random I/Os increases drastically and consequently the ingestion performance suffers. If this situation persists, then the input queues may run full, resulting in back pressure to the producer at least and data loss due to dropped events at worst. The handling of this (rarely expected) case is left to the load scheduler (cf. Section 6).

ChronicleDB is implemented in Java and is integrated into the JEPC event processing platform [25]. It supports an embedded as well as a network mode. ChronicleDB offers a high-performance storage solution for event data while supporting load-adaptive indexing and efficient removal of outdated events.

To improve storage utilization as well as write performance, ChronicleDB makes use of (lossless) compression. Because our main objective is write optimization, we focused on fast compression with reasonable compression rate. Hence, we chose LZ4 [10] as our compression algorithm, but any other would be possible.

For data access, the query engine of ChronicleDB supports an SQL-like query language. Additionally, queries can also be processed via a Java API.

4 STORAGE LAYOUT

This section presents the storage layout of ChronicleDB. We first describe the problem statement in Section 4.1. Then, in Section 4.2, we introduce the components of the storage layout. Finally, we present the overall layout in Section 4.3.

4.1 Problem Statement

The main objective of the storage layout was to support both full sequential write performance and full sequential read performance. Furthermore, we aimed for reasonable performance for random reads and fast recovery support in case of system failures. The challenge was to support these requirements with compressed, i.e., variable-sized, data.

With variable-sized data, the physical position of a requested page cannot be computed using a formula. Hence, access to pages on disk requires a mapping from logical to physical addresses. To deliver fast recovery, this information needs to be stored on disk to avoid a full relation scan during recovery. The straightforward way of storing this information to disk is to write it to a separate file. However, switching between files causes random I/O and thus hurts the ingestion performance considerably.

To alleviate these problems, ChronicleDB's storage layout stores the address translation information interleaved with the data pages and uses back references in the address translation blocks for fast recovery.

4.2 Components

The management of compressed blocks is closely related to that of variable-length records in database systems. They usually manage variable-length records in blocks of fixed size and maintain pointers to the different records within the block. To solve the problem of addressing variable-sized blocks, we adopt this approach. We introduce logical IDs (representing virtual addresses) and an abstraction layer that maps logical IDs to physical addresses. While this solution is quite obvious, the challenging aspect is the storage of the mapping. A straightforward approach would be to store the physical mapping information *logical IDs → physical addresses* separately. Unfortunately, this incurs random writes and therefore results in a significant performance loss, as we will show in our experimental evaluation. Thus, we decided to store the mapping information *interleaved* with the data.

4.2.1 Blocks. The smallest operational unit of the proposed storage layout is a *logical block (L-block)*. Because we utilize disks as primary storage, we align the L-block size at the size of a physical disk block. Each L-block has to be separately accessible via a unique ID. This is why we

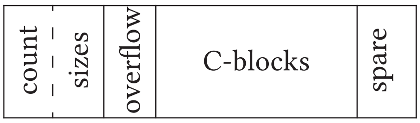


Fig. 3. Macro block layout.

chose L-blocks as unit for compression. While L-blocks are of fixed size, the size of a *compressed block* (in the remainder of the article denoted as *C-block)* depends on its individual compression ratio, and therefore, C-blocks are of variable size.

In terms of compression, the optimal physical data storage layout would be a column layout. In terms of write performance, a row layout is superior. We optimized our storage layout for better compression rates utilizing a hybrid approach. ChronicleDB stores relational events in a column-based fashion *only within a single L-block*, similar to the PAX layout [14]. Thus, all data belonging to the same row is organized within the same L-block. At the same time, the column-based ordering of the data within an L-block groups values that are expected to be very similar, which allows better compression.

4.2.2 Macro Blocks. C-blocks are managed in groups of blocks, denoted as *macro blocks*, with a fixed physical size. Nevertheless, the number of C-blocks contained in a macro block varies, depending on the compression rate of the corresponding L-blocks. Macro blocks provide the smallest granularity for physical writes to disk. The size of the macro block has to be a multiple of the L-block size. We impose this constraint for recovery purposes, as will be discussed in detail in Section 7.

Each macro block stores the number of C-blocks it contains as well as the size of each C-block. If a C-block does not completely fit into the current macro block, then the C-block is split and the overflow is written to a new macro block. So, macro blocks are dense by default. However, out-of-order events cause updates on C-blocks. In case of dense blocks, such updates result in costly macro block overflows. To avoid these costs, we reserve a certain amount of spare space for updates in macro blocks. Section 5.7 provides more details on spare space in blocks. Figure 3 shows the layout of a macro block.

4.2.3 Address Translation Tree. ChronicleDB uses a tree structure (*address translation tree*, ATT) to translate the ID of a L-block to the physical address of its corresponding C-block in the storage layout. The physical address of a C-block c is represented by a tuple (mb_c, p_c), consisting of the position of the corresponding macro block mb_c and the offset p_c within mb_c.

The IDs of L-blocks are simply consecutive numbers. Hence, the ATT solely has to store the physical addresses. This is related to the CSB$^+$-tree [37], which also uses implicit child pointers to improve the cache behavior of the B$^+$-tree.

The mapping information for recent blocks is kept in memory. Though, to support fast recovery, we have to write parts of the mapping information frequently to disk. Therefore, ATT entries are also managed in blocks, denoted as ATT-blocks. The size of an ATT-block is equal to the size of an L-block. If an ATT-block is filled, then it is written to disk. Each ATT-block contains the same amount of entries. E.g., for an L-block size of 8KiB and 64-bit address size, an ATT-block can contain up to 1,020 entries (considering meta data). To support a large address space, we organize ATT-blocks hierarchically in a tree. The resulting ATT does not require explicit routing information for address lookup.

Algorithm 1 outlines the address lookup. It starts at the root of the ATT, which is always and solely kept in memory. Thanks to the consecutive ID numbering, the index of the corresponding

ALGORITHM 1: ATT-block Address Lookup

```
Input: ID $id$ of the requested block, entries per ATT-block $b$ and ATT height $l$
Output: The physical address of the C-block
$index \leftarrow \lfloor \frac{id}{b^l} \rfloor \mod b$;
for $i = l - 1$ to $1$ do
    $address \leftarrow ATT_{i+1}[index]$;
    load $ATT_i$ from $address$;
    $index \leftarrow \lfloor \frac{id}{b^i} \rfloor \mod b$;
end
return $ATT_0[id \mod b]$
```

child entry can be easily calculated as well as the associated address. This address is used to load the child block from disk. The algorithm proceeds with the next levels until a leaf node is reached and the final C-block address can be looked up.

We try to keep at least the index levels of the ATT in memory to improve read performance. This should be possible, as the size of the ATT index (without the leaf level) is N/b^2 for N C-blocks and a L-block size of b.

4.3 Overall Layout

The storage layout is designed to avoid random I/Os. Therefore, macro blocks and ATT-blocks are stored interleaved.

In a first possible solution, from the query-processing perspective, an ATT-block with k mapping entries should ideally address its k succeeding C-blocks. Unfortunately, this requires either buffering these k blocks with the risk of data loss in case of a system failure or performing a random I/O to write the ATT-block after writing the k C-blocks. So, there is a tradeoff between performance and consistency.

In a second solution, the ATT-block is placed behind the data it refers to. So, an ATT-block with k entries always refers to its immediately **preceding** k C-blocks. This way, we do not have to buffer C-blocks during ingestion, but still avoid random I/Os for writing the mapping information. The drawback of the second solution is that read operations now cause random I/Os. To avoid these random I/Os, a sliding read buffer of k L-blocks is used when a sequential scan is performed. This requires less than 8MiB memory in case of 8KiB per L-block. In comparison to the first solution, this approach requires the same amount of buffering, but avoids possible data loss. So, we opted for the second solution.

5 ON INDEXING EVENTS

In this section, we present our indexing approach to support the queries we listed in Section 3.1. Among those are time-travel queries, temporal aggregation queries, and filter queries on non-temporal attributes.

The remainder of this section is structured as follows: At first, Section 5.1 describes the key characteristics of temporal data. Section 5.2 presents our primary index; secondary indexes are addressed in Section 5.4. We elaborate on our handling of user-defined aggregates for lightweight indexes in Section 5.3. Removal of old data is explained in Section 5.5. Section 5.6 explains how the indexes can efficiently support the targeted types of queries. Finally, Section 5.7 addresses our solution for dealing with out-of-order events.

$[min_{a_1}, max_{a_1}]$	sum_{a_1}	UDA	count
$\dots$	$\dots$		
$[min_{a_n}, max_{a_n}]$	sum_{a_n}		
t			

Fig. 4. Index entry layout of TAB$^+$-tree.

5.1 Temporal Correlation

In general, we observe in event processing that values occurring within a small time interval are often very similar. Sensor values, e.g., representing temperature or main memory consumption, typically do not change tremendously within short time periods. We call this *temporal correlation*. In agreement with [18], we introduce a formal notation of temporal correlation in the following. For a given sequence A of attribute values $a_i, 1 \leq i \leq N$, we define the average distance as:

$$dist(A) := \frac{1}{N-1} \sum_{k=2}^{N} \mid a_k - a_{k-1} \mid .$$

This sum is the arithmetic mean of the Manhattan distance. The temporal correlation (tc) is then 1 minus the average distance divided by the range of values within the sequence A. Thus,

$$tc(A) := 1 - \frac{dist(A)}{max(A) - min(A)}.$$

The value of temporal correlation is in the unit interval. If close to 1, then there is a high correlation within the sequence A. We will leverage temporal correlation for lightweight-indexing to speed up queries.

5.2 TAB$^+$-tree

As the primary index for ChronicleDB, we propose the *Temporal Aggregated B$^+$-tree (TAB$^+$-tree)*. The TAB$^+$-tree is based on the B$^+$-tree and uses the events' timestamp as key. As usual for B$^+$-trees, the TAB$^+$-tree node size matches block size, i.e., L-block size.

5.2.1 Index Layout. Inside the TAB$^+$-tree, the nodes of each level are linked *in both directions*, which improves both query and recovery performance (cf. Section 7).

Additionally, we leverage temporal correlation of event data to boost the performance of filter queries. For every node in the TAB$^+$-tree, we store the minimum and maximum (min_{a_i}, max_{a_i}) value of each attribute A_i. Figure 4 shows the index entry layout.

These min-max values are used for supporting filter queries on non-temporal attributes without the need of an index on the secondary attribute. This approach is called lightweight indexing, as it is inexpensive to offer. However, the indexing quality largely depends on the temporal correlation of attributes.

In addition to minimum and maximum values, the TAB$^+$-tree also maintains the *sum* as well as the number of entries (*count*) for each attribute in a subtree. These simple statistics are stored in the index entries, next to the timestamp (t), which represents the key in the TAB$^+$-tree. The storage overhead is very small, because aggregates are only maintained in the index levels, and the number of attributes is negligible compared to the number of entries in an index node. To allow efficient processing of temporal aggregation queries beyond min, max, sum, and count, ChronicleDB offers support for user-defined aggregates (UDAs), which we discuss in Section 5.3.

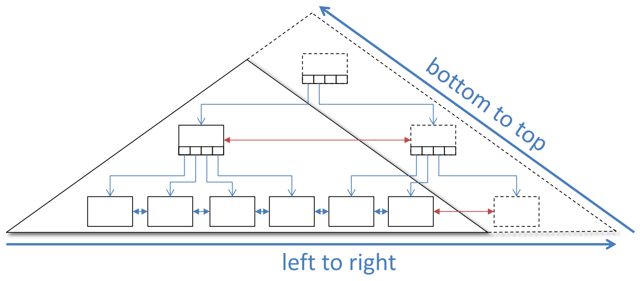


Fig. 5. TAB$^+$-tree construction.

5.2.2 Tree Construction. The problem of storing chronological events in an indexed fashion on disks can be solved with an efficient sort-based bulk-loading strategy for B^+-trees. Figure 5 sketches the index construction. Due to the key's sorted nature, the index can be built in a *from-left-to-right* fashion while holding the tree's right flank in memory. Because sorting is not required, the cost for index creation is reduced from $O(\frac{N}{b}\ log_b\ \frac{N}{b})$ to $O(\frac{N}{b})$ for N events and block size b. Hence, index construction is almost *for free.* We avoid the traversal of the right flank for each event and build the tree from bottom to top. When a leaf node is filled, its corresponding index entry is inserted into the parent node. Therefore, the parent node has to be accessed only once per child node, which also applies to the entire tree.

A problem arises due to the next-neighbor linking (indicated by red arrows in Figure 5). The next-neighbor reference has to be known in advance when the node is written to disk. Otherwise, the node would have to be updated later, resulting in random I/Os that would deteriorate the system performance notably. This issue is intensified by data compression. Therefore, stable IDs are necessary, as we have discussed in Section 4 already.

5.3 User-defined Aggregates

As stated above, each inner node of the TAB$^+$-tree manages aggregated information (min, max, sum, count) for its referenced subtrees on a per-attribute basis. We call these aggregated information *partial aggregates,* as they summarize only a fraction of the stored data. These *partial aggregates* are computed incrementally during insertion. Every time an event is received, it is appended to the rightmost leaf node and all aggregates are updated accordingly. Whenever a leaf is written to disk, the *partial aggregate* is propagated to the parent level. The handling for inner nodes is similar: If an inner node is pushed out, then all partial aggregates of its children are merged and propagated up the tree.

To consistently maintain these aggregations inside the TAB$^+$-tree, we must be able to incrementally update a partial aggregate as new events arrive and merge the values of two partial aggregates. We use and extend the techniques for incremental aggregation presented in [39] to achieve this. Every aggregate in ChronicleDB is implemented via three functions:

init(v:IN):AGG	Computes a *partial aggregate* from an event's attribute value.
merge(a:AGG, b:AGG):AGG	Merges two *partial aggregate* values.
eval(a:AGG):OUT	Transforms the given *partial aggregate* into a value of the output type.

These functions are used as follows: On the arrival of an event, we call the *init* function for each defined aggregate with the corresponding attribute's value and *merge* the result with the *partial*

Table 1. Decomposition of aggregations into functions *init*, *merge,* and *eval* including definitions for IN,AGG,OUT [39]

Aggregate	**Types**			**Functions**		
	IN	AGG	OUT	init(v:IN):AGG	merge(AGG:a, b:AGG):AGG	eval(a:AGG):OUT
Count	T	Int	Int	1	a+b	a
Sum	T	T	T	v	a+b	a
Min	T	T	T	v	min(a,b)	a
Max	T	T	T	v	max(a,b)	a
Average	T	{n:Int,s:T}	T	{1, v}	{a.n+b.n, a.s+b.s}	a.s/a.n
Population StdDev	T	{n:Int,s:T,sq:T}	T	$\{1, v, v^2\}$	{a.n+b.n, a.s+b.s, a.sq+b.sq}	$\sqrt{\frac{1}{a.n}\ (a.sq - a.s\ /\ a.n)}$
Max2	T	{m1:T,m2:T}	T	{v, ⊥}	{max(a.m1,a.m2,b.m1,b.m2), max2(a.m1,a.m2,b.m1,b.m2)}	{a.m1, a.m2}
Min2	T	{m1:T,m2:T}	T	{v, ⊥}	{min(a.m1,a.m2,b.m1,b.m2), min2(a.m1,a.m2,b.m1,b.m2)}	{a.m1, a.m2}

aggregate of the affected leaf. The levels above use the *merge* function to combine the aggregations of their subtrees. *eval* is called whenever the value for a specific aggregate is requested (e.g., by a temporal aggregation query). Table 1 lists the decomposition for some popular aggregations. *Sum*, for example, uses the identity function for *init* and *eval* and the + operation of the underlying numeric data type for *merge*. *Average* and *Population StdDev* use a tuple (triple) of values as data type for *partial aggregates* (AGG) and the *eval* function is used to compute the actual aggregation value.

We enable users to define their own aggregates by providing implementations of *init*, *merge,* and *eval* and the corresponding data types. Once defined, these aggregates are made available and accessible via a unique ID. When defining a new event stream, the user supplies the IDs of the additional aggregates to compute, and ChronicleDB uses the defined functions to manage them inside the TAB$^+$-tree. To attach a new aggregate to an existing stream, we offer three options. As ChronicleDB creates a series of TAB$^+$-tree instances per stream (cf. Section 5.5), by default, we begin storing the values of the new aggregate when a new tree is started. For the current and previous trees, the aggregate is computed on demand on a per-query basis. The second option is to force the start of a new tree and begin storing the new aggregate immediately. Querying the aggregate before that time behaves the same as for the first option. The last and most costly option is to compute the aggregation value for all events/trees in bulk and—depending on the spare-space of the tree's internal nodes—store them inside the existing trees or rebuild the trees from scratch.

5.3.1 Virtual & Dependent Aggregates. Every aggregate stored in the internal nodes of the TAB$^+$-tree reduces its fan-out. To be as space-efficient as possible, we introduced the notion *virtual aggregates*. A *virtual aggregate* is composed of values from other aggregates. As an example, recall the definition of *average* from Table 1. The data type of *partial aggregates* is a tuple of two primitive values, namely *count* and *sum*. So instead of storing these values twice (once for the primitive aggregates and once for the average aggregate), we define *average* as *virtual aggregate* with dependencies to *sum* and *count*. This also simplifies the definition of composite aggregates, as we only require the unique IDs of its dependencies, the definition of the *eval* function, and the output data type (OUT). Note that the signature of eval in this case changes to $eval(v_1 : T_1, \ldots, v_n : T_n) : OUT$ with T_1 being the output type of the first dependency, T_2 that of the second, and so on.

A hybrid between aggregates that can be stored as atomic values and pure *virtual aggregates* are so-called *dependent aggregates*. They have dependencies to other aggregates but also need to store a value. Consider for example the max2/min2 aggregates, which computes the second largest/smallest value of an attribute. They tend to be more resistant against outliers than max/min and thus may be preferred by some analytical queries. Their *partial aggregate* is a tuple holding max and max2 (min and min2) values, whereby max is already provided by ChronicleDB's default aggregates. So it is sufficient to store only the max2/min2 value and use max/min from the existing aggregates. To define a *dependent aggregate*, the user provides implementations for *init, merge,* and *eval*, as well as the unique IDs of the dependencies. However, the signatures of *merge* and *eval* are slightly different: *merge(a:AGG,b:AGG,* $l_1{:}T_1$, $r_1{:}T_1 \ldots$, $l_n{:}T_n$, $r_n{:}T_n$*):AGG* and *eval(a:AGG,* $v_1{:}T_1 \ldots$, $v_n{:}T_n$*):OUT*. *eval* consumes the stored *partial aggregate* value (a) as well as the output values of all dependencies; *merge* works similarly, consuming the stored *partial aggregate* values (a,b) and the *partial aggregates* of the dependencies ($l_1, \ldots, l_n$ and $r_1, \ldots, r_n$).

5.3.2 Rough Estimates. Some aggregates cannot be computed incrementally, i.e., it is not possible to define an exact *merge* function for them. The median or the statistical mode are classical examples of such aggregates. Another example are TOP-K queries with non-trivial scoring/ranking functions, e.g., "Find the top three (in terms of count) values of attribute a_i, whose events have a value x for attribute a_j." Nevertheless, in some cases a rough estimate for these values may be sufficient. Hence, we offer a slightly modified interface for defining such aggregates:

init(v:List[IN]):AGG	Computes a *partial aggregate* from a list of values.
merge(l:List[AGG]):AGG	Merges a list of *partial aggregate* values.
eval(a:AGG):OUT	Transforms the given *partial aggregate* into a value

The main difference here is that *partial aggregates* are computed in bulk by both the *init* and the *merge* function to reduce the information loss induced by binary merges. However, an upper bound for the introduced error can not be given. The *init* function is invoked once a leaf node becomes full and is written to disk. Consequently, the aggregation value is computed based on all events stored in that leaf, and exact values are provided to the parent level. The same is done if an inner node is pushed out: The *merge* function processes the partial aggregates for all referenced subtrees in bulk and propagates the result up the tree. However, the quality of results can decrease with every level, since at higher node levels the basis for computing aggregates is not accurate.

5.4 Secondary Indexes

To efficiently support queries on non-temporal attributes *without* high temporal correlation, ChronicleDB also provides secondary indexes. We chose log-structured indexes, as they are designed for high write-throughput. Nevertheless, maintaining secondary indexes incurs some overhead. Hence, ChronicleDB's load scheduler temporally deactivates secondary indexing in case of peak loads, as will be discussed in Section 6.3.

The most popular log-structured index used, e.g., in HBase [8] or TokuDB [27], is the LSM-tree [35]. In addition to LSM trees, ChronicleDB also supports cache-oblivious look-ahead arrays (COLA), another log-structured index. The advantage of COLA in comparison to a native LSM-tree is its better support for proximity and range queries. To speed up exact-match queries, we utilize Bloom filters [17], which can be maintained very efficiently.

5.5 Time-splitting

ChronicleDB is a hybrid between OLTP and OLAP databases. In terms of data ingestion, ChronicleDB is like a traditional OLTP system, but queries to ChronicleDB are similar to OLAP queries. Aggregation queries on (historical) data are essential in OLAP systems and commonly address predefined time ranges, like the sales within the past week or month [21]. ChronicleDB offers the

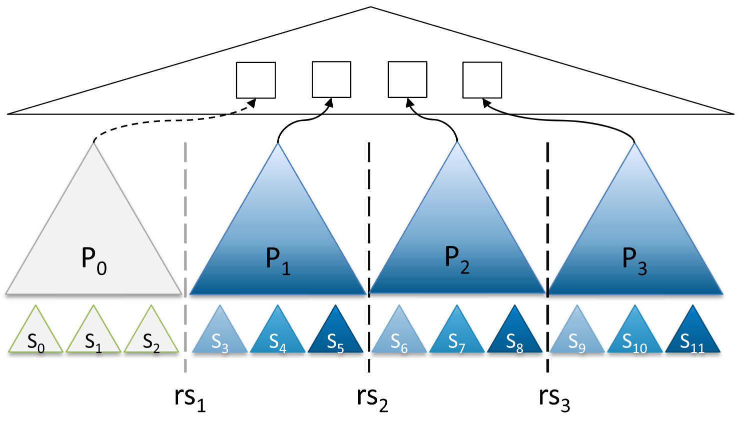


Fig. 6. Index scheduling example.

possibility to align data organization to the specific query pattern. Therefore, we introduce *regular time-splits*. After a user-defined amount of time, a new TAB$^+$-tree residing in a separate file is created; e.g, a salesman interested in weekly sales statistics would choose weeks as regular time split granularity. The same takes place for each secondary index such that the regular time split covers a fixed interval for all indexes. The regular time-splits are managed within a TAB$^+$-tree again.

Though ChronicleDB aims at long-term storage, it also addresses deletion and reduction of ancient data. While removing events from the TAB$^+$-tree could be realized via cutting off its left flank, removal in secondary indexes has to be done on a per-event basis. This leads to costly random I/O operations. Thus, instead of removing data event-by-event, ChronicleDB supports the removal of outdated events at the granularity of regular time splits. In this case, only the corresponding files have to be deleted (logically). Alternatively, the TAB$^+$-tree can be cut at any level above the leaf nodes. This way, temporal aggregation queries can still be answered but at the granularity of the corresponding level, rather than events.

Regular time splits enable ChronicleDB to keep local statistics for each time split. Especially the temporal correlation is an important metric that can be considered to decide which secondary indexes should be maintained. If the temporal correlation for the last split is above a certain threshold, then ChronicleDB can switch to lightweight indexing only. This results in systematic partial indexing. Furthermore, time splits allow for higher insertion performance while building secondary indexes compared to one large index. This also has been observed in [34]. Ancient data is removed from ChronicleDB in whole regular time splits, indicated in Figure 6 before rs_1.

5.6 Query Processing

In this section, we discuss how a TAB$^+$-tree can be utilized for query processing.

5.6.1 Time Travel Queries. Due to its descent from B$^+$-tree and the fact that events are indexed based on their timestamps, the TAB$^+$-tree performs a time travel query like a range query in a B$^+$-tree. The leaf nodes of the TAB$^+$-tree can be sequentially traversed from left to right using the linking between adjacent leaf nodes until an event occurs that is outside of the temporal query range. Thus, the total cost is logarithmic (in the number of events) plus the time to access the required leaves.

5.6.2 Temporal Aggregation Queries. Temporal aggregation queries compute an aggregation value (sum, avg, stdev, count, min, max) for a given point or range in time. Again, the TAB$^+$-tree acts as guide for the temporal dimension. Additionally, if the requested aggregate is materialized (cf. Section 5.3), results can be computed in logarithmic time. If called with the tree root, then Algorithm 2 computes the query result as follows. If the given node is a leaf of the tree, then we simply iterate its events and compute the *partial aggregate* over all qualifying events via *init* and *merge* of the given decomposed aggregate function. In case of an inner node, we iterate its *IndexEntries* and distinguish three cases: If the time interval of the subtree represented by the current *IndexEntry* is fully covered by the query range, then we *merge* the materialized *partial aggregate* into our result and proceed with the next entry. In case the query range intersects the entry's time interval, the algorithm is called recursively with the root of the corresponding subtree. Finally, if the *IndexEntry's* time interval is beyond the query range, processing can be terminated. It is easy to see that this algorithm traverses the tree from root to leaves at most twice: once for the left bound of the query range and once for the right bound. Hence, temporal aggregation queries can be answered in logarithmic time, provided the requested aggregate is materialized in the TAB$^+$-tree.

5.6.3 Secondary Queries in TAB$^+$-tree. Secondary queries, i.e., range queries on event attributes, can utilize the inherent lightweight indexing of the TAB$^+$-tree to speed up query processing. Then, the (min, max) information of the corresponding attributes is used for pruning. If the query interval for a specific attribute a and the interval (min_a, max_a) of a TAB$^+$-tree node are disjoint during tree traversal, then the node is skipped.

Algorithm 3 sketches the secondary query processing: As input, the requested time interval as well as the restrictions on the desired attributes are provided. For simplicity, the proposed algorithm reports all events of qualifying leaves; events not fulfilling the query condition need be filtered in a post-processing step. The TAB$^+$-tree is traversed in depth-first order by means of a stack while nodes are pruned as early as possible. The stack keeps track of the current tree path as well as the index of the last visited entry for each tree level.

ALGORITHM 2: Temporal Aggregation Query

```
Input: Time interval $[t_s, t_e]$, decomposed aggregate function (init,merge,eval), current node N
PartialAggregate result ← $\emptyset$;
if isLeafNode(N) then
    foreach Event $e \in N$ do
        if e.timestamp $\in [t_s, t_e]$ then
            result ← merge(result, init(e));
        end
    end
else
    foreach IndexEntry $ie \in N$ do
        if ie.interval $\subseteq [t_s, t_e]$ then
            result ← merge(result, ie.aggregate);
        else if ie.interval $\cap [t_s, t_e] \neq \emptyset$ then
            result ← merge(result, TemporalAggregationQuery($[t_s, t_e]$, (init,merge,eval), ie.node);
        else if $t_e <$ ie.interval.$t_s$ then
            break;
        end
    end
    return result;
end
```

ALGORITHM 3: TAB$^+$-tree pruning query

```
Input: Time interval $[t_s, t_e]$, range $[min_{a_i}, max_{a_i}]$ for attributes $A_i$
Stack s , Node n , Index i;
s .push((root , 0));
while !s .isEmpty() do
    (n , i ) ← s .pop();
    while i < n .size do
        if n .isLeaf() then
            if n [i ].$t > t_e$ then
                return; /* No further results */
            else if n [i ].$t \geq t_s$ then
                output n [i ];
            end
        else if n [i ] intersects $[t_s, t_e]$, $[min_{a_1}, max_{a_1}]$, ... then
            s .push((n , i +1));
            n ← n [i ].child;
            i ← 0;
            continue;
        else if i > 0 AND n [i -1].$t > t_e$ then
            return; /* No further results */
        end
        i ← i +1;
    end
end
```

5.7 Managing Out-of-order Data

So far, we have assumed an unexceptional chronological order of the incoming events. This is, however, not satisfied in real scenarios where clock skew, network delays, and faulty devices cause occasional out-of-order arrival of events. This problem is well known in event processing [15], but also has a serious impact on the design of ChronicleDB. By default, we deal with out-of-order arrivals of events by maintaining the TAB$^+$-tree as an index on application time as described in the previous sections. By utilizing out-of-order buffer strategies, we can keep the application time query capabilities of ChronicleDB with minimal performance loss for a moderate amount out-of-order data. We will pursue this approach as described in the following.

5.7.1 Out-of-order Buffers. To deal with out-of-order, we introduce Algorithm 4, which is illustrated in Figure 7. First, we try to insert incoming out-of-order events into the right flank buffer of the tree. If the timestamp of an event is too far in the past, then we insert the event into a dedicated queue sorted with respect to application time. When this queue becomes full, we flush its entries *in bulk* into the TAB$^+$-tree. To prevent data loss in case of a system crash, all events in the queue are additionally written to a mirror log in system time order.

Without any further modifications, this approach would still cause serious problems in the TAB$^+$-tree. First, an out-of-order insertion will often hit a full L-block. Consequently, a split would be triggered and the sequential layout would be damaged, causing higher costs for queries. To avoid these splits, we propose to reserve a certain amount of spare space in an L-block for absorbing out-of-order insertions without structural modifications. This is only meaningful if the number of out-of-order arrivals is not extremely high. For example, if we expect 15 out-of-order

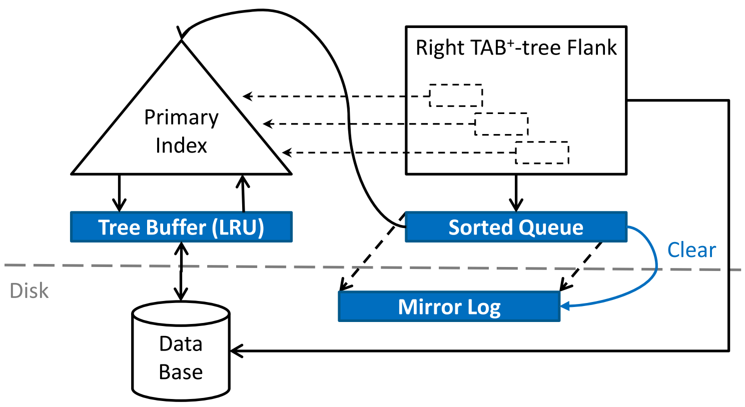


Fig. 7. Buffer layout and data flow for out-of-order data.

ALGORITHM 4: Out-of-order Insertion

Input: Out-of-order event e, right flank of TAB$^+$-tree *flank*, sorted queue *queue*, mirror log *log*

```
if $e.timestamp > flank.lastLeaf.getFirst().timestamp$ then
    // Add $e$ to leaf of the TAB$^+$-tree flank
    $flank.lastLeaf$.add($e$);
else
    // Add $e$ to sorted queue
    $queue$.add($e$);
    // Write e to the mirror log
    $log$.append($e$);
    if $queue.isFull()$ then
        // Flush all events from $queue$
        for $qe$ in $queue$ do
            regular tree insert of $qe$;
        end
        Clear $log$;
    end
end
```

events per L-block, a simple urn-based analysis shows that the probability of an overflow is less than 10% for a spare space of 20 events.

In addition, we also address additional spare space on the storage level. Each L-block corresponds to a compressed C-block whose size depends on the compression rate. A reduction of the compression rate results in an increased C-block size. For example, an update on an aggregate could lead to an increased C-block size, even though the L-block size has not changed. Thus, macro blocks also reserve a certain amount of spare space. If a C-block exceeds the remaining spare space of its macro block, then it is moved to the end of the database and a reference entry is written at its original position.

5.7.2 Keeping Secondary Indexes Consistent. References in the secondary index are represented by physical addresses. So, references to relocated C-blocks in the storage layout will become invalid in a secondary index. One solution for this problem is to update the affected references in the secondary index. However, since there can be many secondary indexes, eager reference updates are very expensive.

Thus, we use the following lazy approach instead where a split of a block does not trigger an update of the entries in the secondary indexes. To maintain the search capabilities of secondary indexes, we store the timestamp of the event in the corresponding index entry of a secondary index. In addition, a flag in each block is kept for indicating whether a block is split or not. If a search via a secondary index arrives at a block that has been split, then we use the timestamp to search for the event in the primary index. For all other blocks, we still use the direct linkage.

6 LOAD SCHEDULER

ChronicleDB is primarily designed to consistently handle huge data rates and, aside from critical hardware failure, achieve no data loss, essentially providing provenance capabilities to a variety of event-driven use cases. Nevertheless, data rates, and thus, ChronicleDB's input workload, may change, depending on multiple internal and external factors, resulting in either permanent or temporary new workload scenarios. An example for a permanent increase in sensor data is an upgrade of the sensor itself or its deployed numbers. A temporary change may arise when a sensor experiences a hardware or connection failure. In this case, it disappears for a while, lowering the workload for ChronicleDB and on its return providing a huge spike in input rate due to an accumulation of delayed data. To deal with the challenging problem of fluctuating data rates and ever-changing workloads, ChronicleDB provides a tailor-made load-scheduling strategy for its multiple components and use cases.

Since minimal data loss is at the core of ChronicleDB's mission statement, the primary goal of our load scheduling is ensuring maximum ingestion speed while providing as much querying capabilities as possible as an auxiliary condition. In the following, we will first present the relevant metrics for our approach. Then, we will describe how the relevant components of our design react based on those metrics . Together, the metrics and those reactions make up our load scheduler.

6.1 Load Metrics

As described in Section 3.2, the storage engine of ChronicleDB can basically be expressed through three logical components: event queues, workers, and disks. The general workflow in an event-driven write-intensive scenario that ChronicleDB was built for is that multiple event producers are *pushing* events to the event queues while the workers write all data to disks. Therefore, our load metrics are derived from information about producers, event queues, and workers.

The most critical component for estimating the current load are the event queues, since they are the first to come in contact with new data, effectively connecting ChronicleDB to the event scenario. Intuitively, a growing queue indicates that the current load is difficult to handle. This can be due to a rise in production from producers or a problem arising within the workers. Similarly, a constant queue size demonstrates that data is being put into and pulled out of the queue at about the same rate, signifying an adequate workload. Finally, a decreasing queue size (capping at 0) indicates that ChronicleDB can handle the load very well and probably could make use of unused resources to further boost its query performances. Based on this observation, we identified four core metrics that the load scheduler keeps track of:

Queue size: Current amount of events in the queue;
Input rate: The amount of producer based pushes to the queue;
Out-of-order rate: The amount of events arriving out of order;
Average delay: The average delay for an out of order event.

In general, those four values should be tracked in a window query, e.g., the input-rate should consider the recent history rather than the whole lifetime of ChronicleDB to react to changes effectively.

The *queue size* is an obvious inclusion based on the description above. The *input rate* helps us to identify the root cause for a changing workload. If *input rate* rises, then there is increased workload from outside. However, a growing queue without an increase in the *input rate* signifies that the problem lies within the workers. Besides hardware failures that will result in ChronicleDB's fast recovery, the biggest influence for problems within a worker is the amount of late-arriving data it has to handle. Even though our strategies for dealing with out-of-order arrivals can handle steady smaller rates of out-of-order data, a sudden huge spike like an increase in producers or a huge batch of delayed events may still negatively impact the system's performance. Thus, the two final metrics we keep track of are the *out-of-order rate* and the *average delay*. The *out-of-order rate* serves as an indicator about the current status of additional work the index has to do to accommodate for late-arriving events. Meanwhile, the *average delay* indicates the pattern of late-arriving events; e.g., a small *average delay* signifies that most events arrive near their supposed insertion time and can probably be handled within the right flank of ChronicleDB that is already in memory. Obviously, this scenario should be handled differently than a consistently large *average delay*.

6.2 Irregular Split

In times of moderate input rates and out-of-order behavior, we try to maintain as many querying boosting capabilities (i.e., secondary indexes) as possible. However, when the *queue size* rises continuously, it eventually can run full, resulting in back-pressure to the producer at least and data loss due to dropped events at worst. Thus, our load scheduler introduces another kind of split in addition to regular scheduled time splits, which we call *irregular splits*. Irregular splits occur at a configurable queue-size parameter δ with $\delta <$ queue-capacity. If producers have a certain tolerance for back-pressure and can hold events themselves, then it would be possible to allow $\delta >$ queue-capacity, but we do not support this at the moment, since ChronicleDB is designed as a reliable consumer of a lot of temporally correlated data. With an irregular split, certain non-essential components of ChronicleDB are disabled to increase the ingestion performance. The exact strategy of what to turn off depends on the other metrics mentioned above and will be explained in detail below. An irregular split's configuration lasts until the next regular time split. At this point, the current status of the metrics above is reevaluated to achieve our auxiliary condition. When we arrive at the lowest resource-consuming configuration possible, we try to avoid further changes in the configuration, because oscillating irregular splits result in lots of tree creations that may take a toll on the ingestion performance. Effectively, this would add to resources consumption, thus negatively impact the ingestion rate we wish to stabilize.

6.3 Partial Indexing Mode

The *partial indexing mode* is activated upon a growing *queue size* with high *input rates*, but a low amount of *out-of-order* data. During moderate *input rates*, we try to maintain as many secondary indexes as possible. We give higher priority to those indexes on attributes with low temporal correlation and high query frequency, since they are less likely to be sufficiently covered by our lightweight indexing solutions. More advanced strategies are possible, such as taking into account the access frequency of attributes. But in case of a system overload, the load scheduler stops building secondary indexes for attributes with high temporal correlation until ChronicleDB can handle the input again, i.e., an irregular split occurs. This results in partial secondary indexes that have to be synchronized with the primary index again at a regular split with moderate *input rates*.

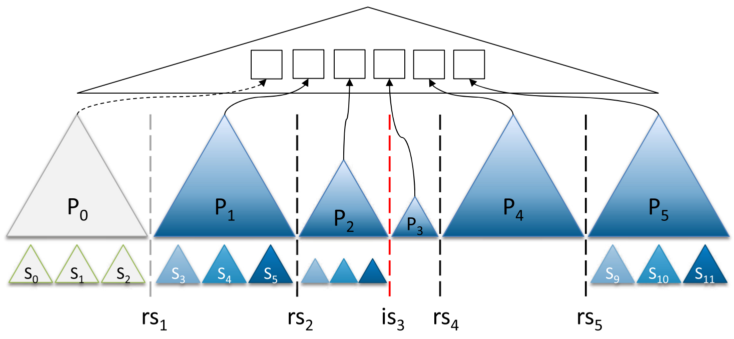


Fig. 8. Index scheduling example with irregular splits.

Figure 8 shows an example where regular splits are prefixed with rs_x and irregular splits with is_x. For each time split, P_x denotes a TAB$^+$-tree and S_x a secondary index. At is_3, secondary indexing has been switched off due to a system overload. Therefore, the primary index is split as well. If the system load decreases, then secondary indexes are switched on again. In this example, secondary indexing continues after rs_5. In case of sufficient resources, ChronicleDB can also rebuild secondary indexes for previous time splits that emerged during an overload, e.g., $[is_3, rs_4]$ as well as $[rs_4, rs_5]$.

6.4 System Time Mode

With high *out-of-order rates* that go beyond our reserved spare space, ChronicleDB may degenerate to a run-of-the-mill index tree. In general, we consider constant high out-of-order rates unlikely, especially in embedded scenarios. However, sensor failures may very well result in a brief period of heightened delay in information delivery that reduces ChronicleDB's overall performance. For such scenarios, in addition to turning off dedicated secondary indexes (*partial indexing mode*), we change our indexing strategy to *system time indexing*. In the following, we will first describe the insert and query strategy for a single system time tree before detailing the impact on those aspects in presence of multiple trees in ChronicleDB that may occur due to regular and irregular splits.

6.4.1 Insert Strategy. In *system time mode*, ChronicleDB timestamps incoming events, adds this information as a system time attribute to each event upon its arrival, and uses this attribute for building the primary index. As a result, each event arrives in chronological order—an ideal scenario for our storage layout. Furthermore, we utilize a lightweight index on the minimum and maximum application time. This consumes little additional space and adds negligible complexity to the insertion algorithm, i.e., it has almost no impact on ChronicleDB ingestion speed. By including the application time as an aggregate, we automatically gain access to the information about the current *out-of-order rate*. Thus, we can use this instead of tracking it outside of the tree. Furthermore, we can use the aggregates for queries and reconstruction of an application time index after the spike of delayed events has come to an end.

6.4.2 Query Strategy. Even though our primary goal during load scheduling is input-driven, we still want to provide query support as best and as cheaply as we can. Unless the user starts ChronicleDB in system time mode from the get-go, the user's perspective is that the system operates based on application time. By utilizing the lightweight application time aggregates, we can perform any query by using Algorithm 3: For the time interval, we specify the maximum system time range the current split covers, and the desired application time range is added to the other attribute ranges of the given query. If application time result order is not required, then we can output the results as is. Otherwise, we have to perform an additional sorting step. Currently, we employ an external merge sort for this process, using a main-memory-based heap with a configurable maximum size to sort page entries and create the merge runs. This setup is the default querying strategy in system time mode.

However, we can further optimize this process by taking into account the ***maximum-delay*** present in the data. We define the maximum delay as follows:

$$\textit{maximum-delay} := \max_{e_i \in events} \{i - j \mid j < i \wedge e_i.ts < e_j.ts\}$$

In other words, the *maximum-delay* is the maximum amount of events that are present between an out-of-order event and the place that it was supposed to be inserted at according to application time. If $\textit{maximum-delay} * eventsize < heapsize$, then we can sort the events in memory without relying on an external merge-sort.

Currently, most literature on out-of-order handling in event-processing systems, are based on punctuation events inside the event stream [41]. Alternatively, there are also passive approaches relying on a user-provided *maximum-delay* value for sorting events in queues or aggressive strategies using retraction events to correct operator results afterwards, provided the necessary events for result construction are not evicted from the system yet [30]. We also provide a configurable *maximum-delay* parameter, but we assume increased out-of-order is an unexpected exception rather than the norm. Therefore, we also utilize the benefits of having the whole event history and ChronicleDB's aggregation capabilities to estimate *maximum-delay* on the fly. In case there are no duplicate application time events, this can be done directly in the TAB$^+$-tree through three user-defined aggregates with an event-level definition of:

$$\begin{aligned} gap &:= \sum_{e_i \in node} \max\left(\min_{j<i}(e_i.ts - e_j.ts - 1), 0\right), \\ delay &:= \max_{e_i \in node} (e_i.ts - \max(appTime)), \\ \textit{max-delay}_{\text{Est.}} &:= \max(delay - gap + \textit{out-of-order-count}, 0). \end{aligned}$$

Here, *gap* computes any gaps in the application time span of a node. Those should be subtracted from the overall time span of a node, since those timestamps are not present and, therefore, not part of the *maximum-delay* definition above. Furthermore, *delay* is the application time difference between the inserted events. Combining both values with the *out-of-order-count* of a leaf node gives an accurate estimation $\textit{max-delay}_{\text{Est.}}$ at the leaf level. The propagation of those definitions to the inner node level is straightforward: *gap* is an incremental, *delay* is a dependent, and $\textit{max-delay}_{\text{Est.}}$ is virtual aggregate. The result at the root of the tree, which is already present in memory, is a tight bound for *maximum-delay*.

In case of duplicate application event times that may, for example, arise in a scenario with broad time granularity, the estimation has to be adjusted. Since all duplicates between an out-of-order event and the place it should actually be inserted in increase the amount of events needed to be looked at in the heap, we need to account for them in the estimation. Based on the aggregates above, a first intuitive upper bound definition would be the following:

$$\textit{maximum-delay} \leq \min(\textit{max-delay}_{\text{Est.}} * \#duplicates, \textit{max-delay}_{\text{Est.}} + \#duplicates).$$

This accounts for both a small overall delay (multiplication) and huge overall delay (sum). However, since duplicates can span multiple nodes and be out-of-order themselves, it is not possible to recognize all duplicates in a self-contained node, making it hard to express the equation above directly in a lightweight aggregate. Relying on our accurate event-level aggregates, we can recognize the maximum distinct count of a duplicate (duplicate-count) and out-of-order entries (ooo-count)

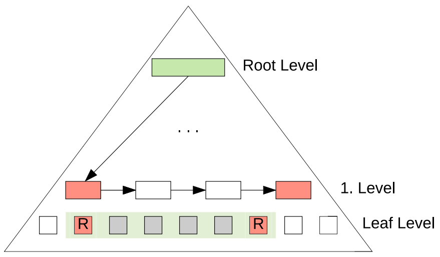


Fig. 9. Alternative query-processing algorithms for overestimated delays.

inside each node. Then, we can combine those exact leaf statistics in inner nodes. By using a dependent aggregate on the application time aggregates, we can determine whether the child-nodes cover the same time span. If not, then we can maximize duplicate-count and ooo-count. Otherwise, we have to perform a sum operation. The sum of duplicate-count and ooo-count at the root level serves as an upper bound for #duplicates.

Regardless of duplicates, we can benefit from our flexible user-defined aggregate definitions to at least get a rough upper bound of the maximum delay in the root entry. A low amount can reduce the complexity of queries on a *system time mode tree* by performing sorting completely in memory rather than reading the almost sorted run from disk, writing it back to disk, and reading it again in a full-blown external merge sort procedure.

This estimate for *maximum-delay* works well enough when the out-of-order events are exponential rather than uniformly distributed, i.e., out-of-order events happen rather closely to their source. However, even with an accurate estimate, a single event with a huge delay can easily degenerate this method to an external merge sort. To reduce this effect, we can utilize the *average-delay* that is being observed by the load scheduler. If it differs a lot from the estimated *maximum-delay* while the query posed by the user has moderate application time span, then we slightly adjust Algorithm 3 for *system-time mode* by leveraging classic tree range query patterns as depicted in Figure 9. Instead of trusting the unusual and possibly overestimated *maximum-delay*, we travel down the tree to the first level just above the leaves. Here, we find more accurate aggregates. Due to linkage between nodes on a level in ChronicleDB, we can traverse the first level and make a more educated guess on the relevant nodes (marked as R in Figure 9) for the query. Based on either the more accurate aggregates or just the number of pages we probably have to read, we can determine if the query can be answered in main memory after all. The algorithm may incur some additional random I/O, but for moderate application time ranges it is preferable to a full-blown merge sort.

6.4.3 Adaptive Processing. The insertion of events outside of *system time mode* has to be adjusted according to the presence of system time trees among all trees created through time splits. Basically, as long as the current tree is in system time mode, we insert any out-of-order events into it, regardless of whether they would fit into past application-time-based trees or not. The reason for that is simple: Opening a past tree and loading multiple tree paths into memory consumes a lot of resources. Operating in *system time mode* is done because we are low on resources and/or there is a high amount of out-of-order events present. To keep the ingestion rate consistently high, we do not make any inserts into past trees in this scenario. If the current tree indexes events based on application time, then we are not at our absolute limit and insert events into already closed trees. In total, there are three possible scenarios for an insertion into an older tree, which are displayed in Figure 10. Scenario (1) is the normal situation present during regular splits. The covered

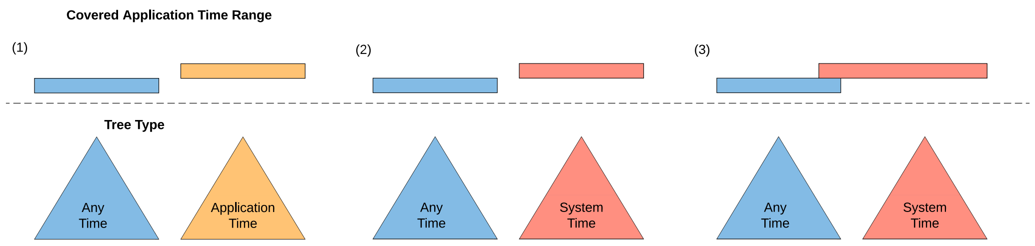


Fig. 10. Insert scenarios after a split.

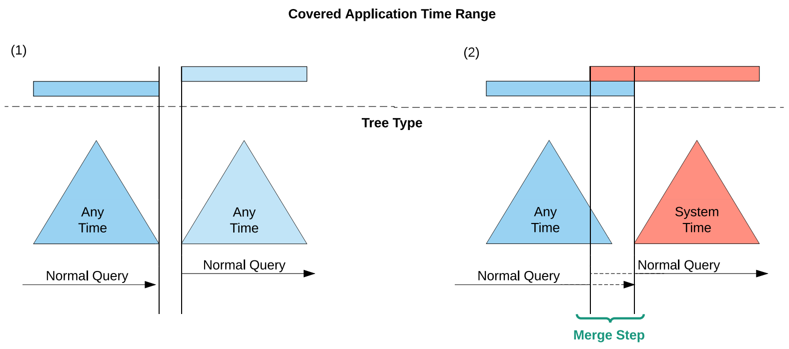


Fig. 11. Query scenarios after a split.

application time interval of a successor application-time-based tree cannot overlap with the previous tree. Thus, we can insert an event into the appropriate tree. If there is a gap between the covered time ranges, then we pick the more recent application time tree. The same method applies for the somewhat unlikely scenario (2), in which a system time tree, created during high out-of-order scenarios, does not have an application time overlap with its immediately preceding tree. Finally, scenario (3) covers the case that a system time tree overlaps with its predecessor. In this case, we could insert the event into both trees. Since both trees are already closed and flushed to disk, the main cost is reading the necessary path into main memory. Thus, for events falling into the overlap range, we choose the predecessor, since it could be an application time tree, and the event would fall into the correct position according to application time rather than into the right flank of the system time tree, causing a potentially high *maximum-delay* estimation.

Based on this insert strategy, we can formulate our general query strategy. If a query covers just one tree, then we query it normally. However, if the query overlaps with the application time range of multiple trees, we have to query each of them individually and combine their results. With only regular splits, as depicted in scenario (1) of Figure 11, we can query the trees sequentially, since there is no overlap. The new scenario that arises due to the inclusion of system time mode, is scenario (2): A system time mode tree can overlap with potentially all preceding trees. Thus, we have to sort the query results of each tree, defaulting to an expensive external merge sort. However, in practice, the overlap is unlikely to span multiple preceding trees and is probably a small temporally correlated range with some rare outliers, as covered in the previous section. Therefore, we sort the requested time span in the system tree according to our strategies above and then combine the results with its predecessor in a final merge-step with the two sorted input runs being the trees themselves. The process will initially read the first page of each tree. Afterwards,

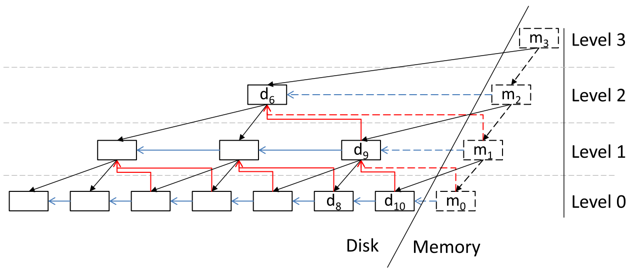


Fig. 12. ATT with recovery references.

as depicted in Figure 11, the normal query of the preceding tree will be processed sequentially until the overlap begins. Rare outliers of the system time tree that may enhance the overlapping region can in practice probably be resolved within the first page of the sorted system time tree. Afterwards, a small amount of random I/O for the densely correlated overlapping region will follow before continuing with a normal system time mode query. Unless the overlapping region is huge and contains a lot of temporal successors being unfavorably scattered across both trees, this is an adequate solution for most queries.

7 FAILURES AND RECOVERY

This section addresses the recovery capabilities of ChronicleDB after a system crash. Recovery takes place in three steps. At first, the storage layout is recovered. Subsequently, the primary index is restored. Finally, the logs are processed to transfer the system into a consistent state again.

7.1 Storage Layout Recovery

In ChronicleDB, the most critical part of the storage layout is its address translation, i.e., the ATT. As the root and the right flank of the ATT are only kept in memory, any information about block address translation is lost in case of a system crash. Rebuilding the ATT would require a full database scan. However, this is not acceptable for a database we expect to be very large (in the range of terabytes).

To support fast reconstruction of the ATT's right flank, we introduce references within the ATT. As only the recently created part of the ATT has to be restored, recovery is performed from the end of the database to its start. Each ATT-block keeps a reference to its previous ATT-block on the same level. Given the last successfully written ATT-block, its predecessor can be directly accessed. The recovery has to scan all ATT-blocks that are children of the last (and therefore lost) ATT-block in the parent level. Then, recovery continues with the next level. To support the direct access to upper levels, ATT-blocks additionally store a reference to its parent's predecessor ATT-block. Thus, these references implicitly create checkpoints for each level. Figure 12 shows the linking of ATT-blocks. For presentation purposes, we assume two address entries per ATT-block. For example, the leaf d_{10} is a child of m_1 and keeps an extra pointer to d_9 that is the predecessor of m_1.

The ATT recovery is outlined in Algorithm 5. In case of a crash, the last written ATT-block is sought on disk. This is simple, as the size of a macro block is a multiple of ATT-block size. There are two possibilities for the classification of the last written L-block: either it is an ATT-block or it is part of a macro block. In the latter case, the previous L-block is read until the last successfully written ATT-block is found. The upper bound for the number of L-blocks to be read before finding an ATT-block is the number of entries per ATT-block. After having located the last successfully written ATT-block, the recovery of the ATT continues by leveraging the introduced references.

ALGORITHM 5: ATT Recovery

```
Input: Database size in bytes $s$, L-block size in bytes $b$
$b_{addr} \leftarrow \lfloor \frac{s}{b} \rfloor * b$ ;                    // Last complete block address
$block \leftarrow read(b_{addr})$;
// Lookup last ATT-block
while $block$ is not an ATT-block do
    $b_{addr} \leftarrow b_{addr} - b$;
    $block \leftarrow read(b_{addr})$;
end
// Rebuild ATT
while $block.prev$ != $null$ do
    $prev \leftarrow read(block.prev)$;                      // Read previous entry
    if $prev.prevParent$ != $block.prevParent$ then
        // Switch to the next higher ATT level
        $block \leftarrow read(block.prevParent)$;
    else
        // Restore the reference in the new parent entry
        Add $b_{addr}$ to $ATT_{block.level+1}$;
    end
end
```

The predecessors of the last written ATT-blocks are located until the parent reference is different. The corresponding references of these ATT-blocks (except the last) are used to rebuild the parent node. The recovery continues at the next upper level with the parent's previous entry. At each level of the ATT, the number of entries to be read is limited by the number of entries per ATT-block.

Figure 12 gives an example. In case of a system failure, the ATT-blocks $m_0 - m_3$ are lost. The recovery starts to discover d_{10} first, which was referenced in m_1 before the crash. Its predecessor, d_8, has a different previous parent reference. So, recovery continues with d_9 and afterwards with d_6. After that, $m_1 - m_3$ are recovered. m_0 is restored by simply scanning all macro blocks after d_{10}.

7.2 TAB$^+$-tree Recovery

In the second step of the system recovery, the right flank of the TAB$^+$-tree is reconstructed. This reconstruction is very similar to the ATT recovery and starts scanning the database in reverse order for the last successfully written TAB$^+$-tree node. After locating the last node n_i at level i, a new index entry is inserted into the tree's right flank at level $i + 1$. In the next step, all nodes of level i belonging to the same parent node are iterated utilizing the previous neighbor linking at all levels of the TAB$^+$-tree. The recovery continues recursively with the last written node of the parent level until the root is reached. Finally, the sorted queue is restored by scanning the mirror log.

8 CHRONICLEDB & EVENT STREAM PROCESSING

Besides analytical ad hoc queries, we equipped ChronicleDB with the ability to re-run continuous queries (CQs) from Event Stream Processing (ESP) systems. This feature can be used in plenty of scenarios, including:

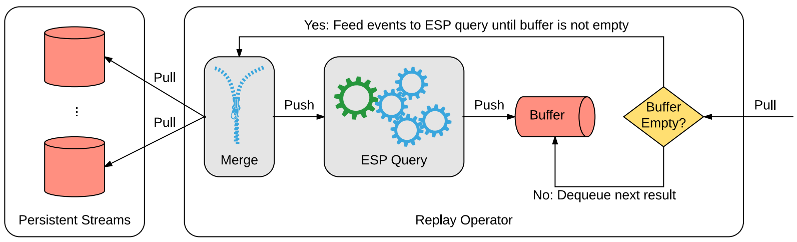


Fig. 13. Dataflow of the generic replay operator.

Post mortem analysis: Explore why an alarm rule did *not* fire in a specific situation.
Loading ESP operator states: Quickly populate the states of ESP operators after a system failure.
Load shedding: Postpone the execution of some continuous queries if the system is overloaded (instead of simply dropping events or reject the query to be executed).

In this section, we first present the design of ChronicleDB's generic replay operator, which can be used to re-run arbitrary continuous queries. Afterwards, we present optimized versions of sliding window aggregation and pattern matching that reduce the processing time considerably. ChronicleDB is integrated into JEPC [25], a middleware for uniform event-stream processing. Continuous queries in JEPC are composed of the following four basic operators:

Filter: Filters an event stream based on a user-defined predicate.
Sliding Window Join: Joins two event streams based on a user-defined predicate and window condition.
Sliding Window Aggregation: Aggregates event data with respect to the given window.
Pattern Matcher: Matches a user-defined pattern (regular expression) to an event stream.

In addition to the four basic operators, JEPC also supports user-defined operators that consume multiple input streams and result in a single result stream.

In contrast to ChronicleDB, which offers a pull-based iterator interface for query results, JEPC uses a publish/subscribe interface to deliver results in a push-based manner. This means events arriving in JEPC trigger the processing of operators and the produced results are sent to upstream operators. To seamlessly integrate ESP queries into ChronicleDB, we need to provide a bridge between the two different processing paradigms. Figure 13 shows our solution of the replay operator: The operator is initialized by passing the compiled CQ and the desired replay time range. We treat the CQ as a black box and our operator registers itself as a consumer to this query. Thus, every result of the CQ is delivered to our operator that stores them inside an (in-memory) result buffer. To feed the CQ with input data from ChronicleDB, we issue a time-travel query with the specified range for every stream specified in the queries' inputs. The results of these queries are merged into a single virtual stream ordered by the events' timestamps. When an application asks for the next tuple from our replay operator, the operator first probes its (in-memory) result buffer. If there is a buffered result, then it is removed from the buffer and returned. Otherwise, events are pulled from the source queries and pushed into the CQ via the virtual input stream until one or more results are reported or the virtual input stream is empty.

Because of ChronicleDB's storage layout, replaying (fractions of) streams only requires a sequential read operation, which is reasonably fast. However, the processing of CQs can take advantage from ChronicleDB's lightweight indexing. A filter query can be answered using secondary attribute queries presented in Section 5.6.3, but the remaining operators require specialized algorithms. In the following, we present the corresponding algorithms for two of the four operators, namely the sliding window aggregation and pattern matching, while the remaining operators will be treated in our future work.

8.1 Sliding Window Aggregation

In stream processing applications, aggregations are typically performed over sliding windows to capture the most recent fraction of an event stream. There are two known variations of sliding windows called time-based and count-based windows. In the following, we outline our approach for sliding time-based windows, STW for short. A STW is defined by two parameters: *duration* and *slide*. *Duration* defines the length of the time-period for which events are stored, and the *slide* parameter defines the amount of time the window is slid forward in each step. For aggregations, this means that the defined aggregates are computed based on events arrived in the past *duration* time units every *slide* time units. More formally, let d be the window's *duration*, s the *slide* parameter, and γ a set of aggregate functions. The STW aggregation $Agg_{d,s,\gamma}$ of an event stream $E := <(p_1, t_1), (p_2, t_2), \ldots>$ is defined as:

$$\begin{aligned} Agg_{d,s,\gamma} =< & \left(\gamma\left(E_{[t_1, t_1+d)}\right), t_1 + d\right), \\ & \left(\gamma\left(E_{[t_1+s, t_1+s+d)}\right), t_1 + s + d\right), \\ & \left(\gamma\left(E_{[t_1+2s, t_1+2s+d)}\right), t_1 + 2s + d\right), \ldots > \end{aligned}$$

with $E_{[t,t')}$ being the subset of events whose timestamps lie in the half open interval $[t, t')$. As can be seen from the above definition, STW aggregation produces exactly 1 output event per *slide*, which is exploited by JEPC's STW aggregator. It works similar to the approach presented in Reference [39]: The aggregator manages a balanced tree (2–3–4 tree) of partial aggregates. Each leaf entry of the tree holds aggregated event data for exactly one *slide*. The levels above hold partial aggregates, each of them composed from the partial aggregates of its children. Hence, to answer an aggregation query, only the root is visited and updates (insert and evict) are processed on a single path in logarithmic time. Because an entry summarizes the events of a whole *slide*, the logarithmic costs for insert and evict arise once the window is slid forward. The only computation required in between is to merge incoming events into the current (latest) partial aggregate.

For an arbitrary *slide* parameter, the replay-based solution processes every event of the query range exactly once. This is already very efficient for small *slides*. For larger *slides*, we can take advantage of ChronicleDB's lightweight indexing by issuing appropriate temporal aggregation queries and thus omit visiting *all* events in the query range. Of course, this is only possible if the requested aggregate is materialized (cf. Section 5.3). Hence, to deliver the best processing performance for any *slide*, we must combine both approaches by introducing a cost model for decision making. In the following, we first show how sliding window aggregation can be answered via temporal aggregation queries and afterwards present a cost model that is able to decide whether to use the replay operator or our TAB$^+$-tree based version. For the ease of presentation, we initially assume that the *duration* parameter is a multiple of the *slide* parameter. At the end of this section, we describe how to extend this approach to arbitrary *slides*.

8.1.1 STW Aggregation via Temporal Aggregation Queries. A straightforward implementation of STW aggregation is to issue a temporal aggregation query for every window from scratch (Figure 14, left). However, this approach has two deficiencies: First, two consecutive windows may overlap to a great degree (depending on the *slide* parameter), which leads to duplicate computations. Second (and more important), this method may induce a random disk access pattern: In general, a temporal aggregation query accesses the leaf level of the TAB$^+$-tree twice: Once for the

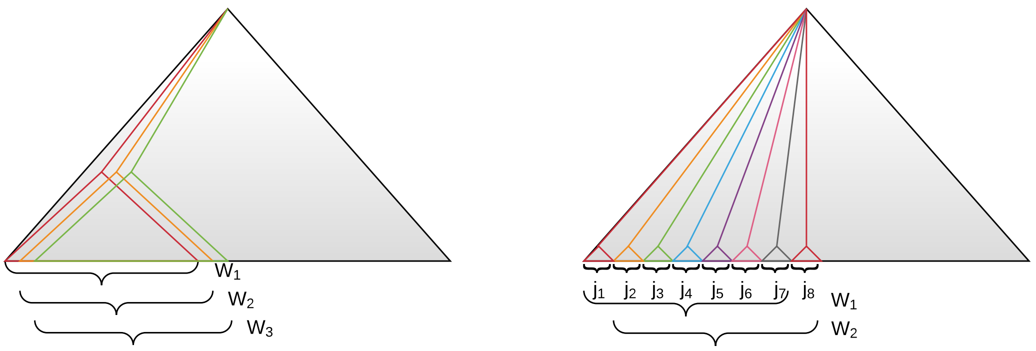


Fig. 14. Processing units for sliding window aggregates: naive approach (left) and optimized approach (right).

left bound and once for the right bound of the specified time range—except for the unlikely case that the queried time range is aligned with the time bounds on any level of the tree. Please note that the tree path traversed for the left bound at window i is the same as the right path for window $i - \frac{d}{s}$. Consequently, this approach induces a forward scan only, if the database buffer is large enough to hold all pages required for answering $\frac{d}{s}$ queries. Otherwise, there are two (random) accesses per window.

To overcome these deficiencies, we implemented a different approach: We use the tree structure of the ESP operator described above, but instead of performing updates on a per-event basis, we issue a temporal aggregation query to compute the partial aggregate of a single *slide* in one go. As can be seen in Figure 14 (right) the results of these queries do not overlap; hence, we do not execute duplicate computations. Further, the left tree path of query i is equal to the right path of query $i - 1$, requiring only a fraction of the database buffer (twice the tree height) for a forward scan access pattern. As we will show in our experimental evaluation, this approach outperforms the naive approach for any value of *slide*.

8.1.2 Cost Estimation. In the following, we present how the costs for both the replay-based STW aggregation and the TAB$^+$-tree based version can be estimated to decide which method to use in a given scenario. Table 2 summarizes the symbols used throughout the next paragraphs.

From an I/O perspective, the TAB$^+$-tree based approach should be faster than a replay when the *slide* parameter covers a certain amount of leaf nodes. Since in ChronicleDB the transfer unit between memory and disk is a macro block, we should see improvements if $s \cdot f$ is greater than the average number of events per macro block. In fact, modern spinning disks avoid seeks up to a much-larger skip size (in our experimental setup 750KB $\hat{=}$ 24 macro blocks à 32KB), resulting in identical I/O costs for both approaches up to skips of that size. However, we experimentally validated that our tree-based aggregator starts to outperform replay at much lower *slide* values. Consequently, the cost estimation focuses on the required computations: Because we reuse the ESP aggregator's tree structure, the only difference in computational costs is the number of partial aggregate merges performed per *slide*. The costs for obtaining a window's aggregate values and the management of the 2-3-4 tree are equal. So, to compare both approaches, we simply estimate the number of merges of partial aggregates per *slide*. As already stated above, the replay operator performs 1 merge per event. Hence, the costs for the replay variant are estimated as: $C_{\text{replay}}(s) = s \cdot f$.

For the TAB$^+$-tree based approach, we count the merges while traversing the tree. This number is composed of the following four cases (cf. Figure 15):

Table 2. Symbols Used in the Aggregator Cost Estimation

Symbol	Description
d	Window duration.
s	Window slide.
f	Event frequency (events per time unit).
h	Height of the TAB$^+$-tree.
B	Number of events stored per leaf node.
D	Number of child references per inner node.
L	leaves per *slide* $L := \frac{s \cdot f}{B}$.

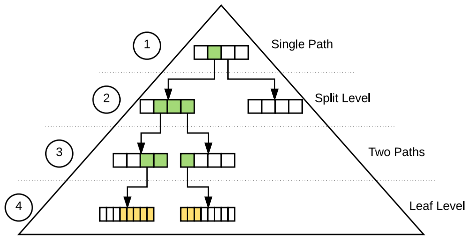


Fig. 15. Illustration of performed partial aggregate merges during a temporal aggregation query.

(1) 1 merge per tree level, as long as the queried time interval is fully covered by one child reference.
(2) $s - i$ merges on the level where the path splits (l_s). Here, i denotes the index of the child reference intersecting the query range's lower bound and s the index of the reference intersecting the upper bound.
(3) For every level below (except for the leaves), we estimate $\frac{D}{2}$ for each of the two paths, resulting in approx. D merges per level.
(4) We do the same on the leaf level, resulting in a constant of B merges. There is one exception to that rule: If $s \cdot f < B$, then the query uses a single path down to that leaf, so we simply add $(s \cdot f)$ instead of B.

With the split-level computed as $l_s := \lceil log_D(L) \rceil$ (0 is the leaf level), the costs are summarized as:

$$C_{\text{tree}}(s) = \underbrace{(h - 1 - l_s)}_{(1)} + \underbrace{\left(\min(1, l_s) \cdot \left\lceil \frac{L}{D^{l_s - 1}} \right\rceil\right)}_{(2)} + \underbrace{(\max(0, l_s - 1) \cdot D)}_{(3)} + \underbrace{(\min(1, L) \cdot B)}_{(4)}.$$

8.1.3 Arbitrary Slide Sizes. So far, we assumed the *duration* to be a multiple of the *slide* parameter. To overcome this limitation, we split *slide* into two parts: $s_1 = s - (d \bmod s)$ and $s_2 = (d \bmod s)$. Thus, the full *duration* of a window can be composed as

$$d = s_2 + \left\lfloor \frac{d}{s} \right\rfloor \cdot (s_1 + s_2).$$

After populating the initial window state with alternating *slide* slices ($< j_2, j_1, j_2, \ldots, j_1, j_2 >$), the computation for each *slide* simply has to remove the first two and insert two new *slide* slices into the tree before computing the result of the next window. Note that we always remove and insert slide pairs of sizes s_1 and s_2 and hence guarantee the correctness of the produced results. In the general case case of an arbitrary slide size, the costs per *slide* is the sum of the costs for both *slide* slices:

$$C_{\text{tree}}(slide) = C_{\text{tree}}(s_1) + C_{\text{tree}}(s_2).$$

8.2 Temporal Pattern Matching

Pattern matching is among the most important operators in ESP systems [32, 45]. For an event stream, usually ordered by time, it returns all event subsequences matching a regular expression of symbols within a given time window. Just recently, pattern matching has also become a new operator in the SQL standard [6] with almost the same semantics as its streaming counterpart. In fact, the motivation of supporting this operator in a DBMS is to offer streaming functionality for a temporal database. Therefore, there is great interest in efficient algorithms for supporting pattern matching on very large temporal databases. One naïve approach is to stream the data from the temporal database into the equivalent pattern matching operator of an ESP system. However, even if the data already exists in the desired (temporal) order, this is a very time-consuming approach, since all data records need to be transferred into memory and processed.

The goal of this section is to present efficient processing strategies that exploit the powerful indexing capabilities of ChronicleDB to reduce the amount of data that pattern matching is performed upon. In the following, we assume that the temporal order maintained in ChronicleDB corresponds to the one specified in the pattern matching query. This is usually the case, as a primary purpose of ChronicleDB is the materialization of event streams of an ESP system such that the semantics are fully preserved. In addition, we assume that users are generally interested in rare patterns, and thus, the selectivity of such a pattern query is quite low. Finally, index support is only possible if the symbols of the patterns relate to predicates that correspond to Boolean expressions on attributes for which an index is actually available.

In the following, we first introduce our terminology and then present different algorithms.

8.2.1 Preliminaries. This subsection introduces the formal notions of the pattern-matching query operator we use in the rest of the article. The basic goal of our index-based variant is using predicates that are part of the operator to prune irrelevant events from consideration. Thus, for the sake of presentation, we purposefully restrict the operator to predicates that can benefit from ChronicleDB indexes in the pruning process. This results in a simplified definition. However, even though the definition here is a subset of state-of-the-art definitions [47], actual queries in ChronicleDB can make use of all features provided by most advanced implementations by treating the actual matching process as a black box: Any predicates outside of this article's definition are resolved to true. This results in a superset of events that more specific implementations can handle accordingly. To provide the necessary intuition, we will briefly sketch our index-supported feature set and discuss an example for the respective subclass of pattern queries before providing a more formal definition.

We assume a query pattern being composed of a sequence of symbols, each of them with or without a Kleene operator. In addition, we restrict the query pattern to a time window of length w such that only those continuous subsequences are of interest with entries in temporal distance of at most w. A symbol generally refers to a Boolean expression, but we limit our discussion to expressions consisting of a conjunction of range predicates. As an example, let us consider an event stream received from a stock exchange where each event consists of the attributes price and volume as well as the mandatory temporal attribute. We consider the symbols A, B, C, and D representing the Boolean expressions "price < 100", "vol > 10,000", "true", and "price > 120" respectively. Then, an interesting pattern is "ABC*D within 7 days" that returns the stocks with a sharp increase of the price within a week after a large order transaction occurred.

The following definitions provide the foundation for the algorithm introduced later in this section. $TS = (e_1, \ldots, e_N)$ is a time series of a fixed size N, where $e_i = (a_i^1, \ldots, a_i^d, t_i)$ is a d-dimensional event over a schema $(A^1, \ldots, A^d, T)$. Here, A^i denotes the ith attribute of the d-dimensional payload and T refers to a discrete time domain. Furthermore, timestamps of events are assumed to be unique and the time series is in temporal order, i.e., it holds $t_i < t_j$ for $i < j$. Let dom_A denote the d-dimensional domain of attributes $A^1, \ldots, A^d$. A *basic symbol* $S_B : dom_A \mapsto \{true, false\}$ is a predicate composed of a conjunction of atomic predicates, each of them expressing a range condition on a single attribute. For a basic symbol S_B, the derived *Kleene-star symbol* S_B^* is a predicate defined on a continuous subsequence of TS with arbitrary length (including length 0). A *symbol* S is either a basic symbol or a Kleene-star symbol.

A pattern matching query $PQ = ((S_1, \ldots, S_k), w)$ is composed of a sequence of symbols $S_1, \ldots, S_k$ and a (temporal) window length w. Without loss of generality, we assume symbols within a pattern to be different with respect to their name. A subsequence $(r_1, \ldots, r_m)$ of TS of length m is a *match* for the pattern query PQ if the following conditions are satisfied:

(1) $w > 0$
(2) $r_m.T - r_1.T \leq w$
(3) If S_1 is a basic symbol, then $S_1(r1)$ and $(r_2, \ldots, r_m)$ is a match of the pattern query $((S_2, \ldots, S_k), w - (r_2.T - r_1.T))$
(4) If S_1 is a Kleene-star, then there exists a $m' \leq m$ such that $S_1(r_i)$ for $1 \leq i \leq m'$. Furthermore, if $m' < m, (r_{m'+1}, \ldots, r_m)$ is a match of the pattern query $((S_2, \ldots, S_k, w - (r_{m'+1}.T - r_1.T))$
(5) There is no longer subsequence $(r_1, \ldots, r_m, \ldots, r_x)$, $m < x \leq N$, satisfying the first four conditions.

Note that our definition allows pattern matching queries with $w \leq 0$, but these queries have no match in the time series due to condition (1). Condition (2) guarantees that the subsequence occurs in a window of size w. Condition (3) addresses the case of a basic symbol at the beginning, whereas condition (4) treats the case of a Kleene-star symbol. Condition (5) ensures that the query accepts only the longest subsequences starting with r_1 that satisfy the first four conditions.

A pattern matching query PQ applied to a time series $TS = (e_1, \ldots, e_N)$ returns a sequence of time intervals $([ts_1, te_1], \ldots, [ts_x, te_x])$ with $ts_i < ts_j$ for $i < j$ such that

(1) The events in $[ts_j, te_j]$ are a match of PQ, $1 \leq j \leq x$.
(2) There are no other matches of PQ starting at t_s and ending at t_e such that there exists a $j \in \{1, \ldots, x\} : [ts_j, te_j] \subset [t_s, t_e]$.
(3) There are no other matches starting at event e where $e.t \notin \cup_{1 \leq i \leq x}[ts_i, te_i]$.

The second condition states that the interval of a match must be of maximum length. The third condition ensures that PQ returns all results.

8.2.2 Algorithms. In this subsection, we discuss the different algorithms for pattern queries available in ChronicleDB. As mentioned before, the naïve algorithm called BaseLine reads the entire time series and processes the input as a stream using a standard automaton-based algorithm like SASE [45], known for pattern matching on streams. It is particularly useful when there are many matches in the result set. However, for rare patterns, other algorithms employing the capabilities of indexes are more appropriate. These more advanced algorithms use BaseLine as a subroutine on a continuous subsequence of the original time series. In the rest of this section, we present two algorithms that are applicable when the involved symbols refer to attributes for which an index exists. Before going into the details of the algorithms, we introduce the notion of a temporal scope.

For a given pattern query PQ, a basic subpattern consists of a continuous sequence of *basic* symbols in PQ. In our example PQ = ABC*D, AB and D are basic subpatterns, but also A and B. In the following, we are interested in subpatterns of maximal length. $SP(PQ)$ denotes the sequence of these maximal basic subpatterns and the order relation $\leq$ refers to the position in that sequence. In addition, $BS(PQ)$ denotes the sequence of basic symbols. In our example, $SP(PQ)$ = (AB, D) and $BS(PQ)$ = (A,B,D). Furthermore, $dur_{min}(s)$ denotes the *minimal* duration of a basic subpattern $s \in SP(PQ)$. Because the minimal temporal distance between two consecutive basic symbols is 1 time unit, dur_{min} is equal to the length of the subpattern. For each subpattern $s \in SP(PQ)$, we introduce a temporal lower bound $t_{min}(s) = w - \sum_{sp \geq s} dur_{min}(sp)$ and a temporal upper bound

ALGORITHM 6: IPM_Basic

```
Input: PatternQuery PQ, TimeSeries TS
$Res \leftarrow \emptyset$;
$S \leftarrow$ basic symbol of PQ with lowest selectivity and indexing support;
foreach event e of TS matching S do
    $TS' \leftarrow$ TS restricted to $TScope(e, S, PQ)$;
    $Res \leftarrow Res \cup BaseLine(PQ, TS')$;
end
return Res;
```

$t_{max}(s) = w - \sum_{sp \leq s} dur_{min}(sp)$. These bounds serve to restrict the temporal scope of a pattern matching query. Assume that $e_D \in dom_A \times dom_T$ is an event that qualifies for the subpattern D. Then, the largest possible range where a result can occur is $[e_D.T - t_{min}(D), e_D.T + t_{max}(D)]$. This interval is called the temporal scope $TScope(e_D, sp, PQ)$ of event e_D with respect to a matching subpattern *sp* of a pattern query *PQ*. Note that these definitions are directly transferable to $BS(PQ)$, the sequence of basic symbols of *PQ*.

A first solution to utilize the indexing capabilities of ChronicleDB (Algorithm 6 IPM_Basic) performs in the following way: After initialization, the basic symbols of the pattern query *PQ* are examined for whether an index exists for an attribute to which the corresponding predicate refers. Recall that predicates consist of a conjunction of range conditions and that each range condition refers to only one attribute. Among those symbols, we choose the one denoted by *S* for which a range condition has the lowest selectivity, i.e., the range query is expected to return the smallest number of results among all basic symbols with indexing support. In the third step, the associated index returns all events *e* of the time series that are a match of symbol *S*. Within the body of the loop, a continuous subsequence TS' of TS is computed by restricting it to the temporal scope of *e*. Then, BaseLine is called for TS' and the results are accumulated in *Res*. After the end of the loop, the algorithm returns its total results in the final step. It is easy to see that this algorithm is correct, i.e., it produces every interval of the result and it does not return an interval that is not a match.

IPM_Basic does not perform well in all cases. Consider again our pattern ABC*D where the selectivity of D is 0.2, of A is 0.24, and of B is 0.25. Assume that there is an index for each attribute referenced by A, B, and D. Because the selectivity of D is the lowest, IPM_Basic decides to use the index of D. The problem of this decision is that the adjacent symbols A and B by itself offer a higher selectivity, but their combined selectivity is much lower than the one of D (assuming that A and B are independent). Thus, even though the selectivity of A is slightly higher than the one of D, it is a better choice, because it is very inexpensive to reject candidates from further evaluations very early, and thus, there are less calls of algorithm BaseLine. This problem is addressed in the next algorithm, IPM_Advanced.

The algorithm IPM_Advanced (Algorithm 7) first computes the subpattern *sp* with the lowest score, and then it determines the symbol with the highest pruning power within *sp*. The score of a subpattern is simply the product of the selectivities obtained from the corresponding basic symbols. This is equivalent to selectivity estimation in database systems under the assumption that attributes are independent from each other. All the techniques known from database systems to improve the accuracy of selectivity estimation would also apply to our problem setting, but it is beyond the scope of this article to examine them in detail. The loop determines all events *e* (using the associated index). For each matching event *e*, it examines the surrounding of *e* for matches of the subpattern *sp*. Only for those matches BaseLine is called. Similar to IPM_Basic, the results are accumulated and the total results are returned in the final step.

ALGORITHM 7: IPM_Advanced

```
Input: PatternQuery PQ, TimeSeries TS
Res ← ∅;
sp ← basic subpattern of PQ with lowest score and indexing support in at least one of its symbol;
S ← basic symbol of sp with lowest selectivity and indexing support;
foreach event e of TS matching S do
    if the neighborhood of e is a match of sp then
        TS′ ← ts restricted to TScope(e, S, PQ);
        Res ← Res ∪ BaseLine(PQ, TS');
    end
end
return Res;
```

There is an opportunity for performance improvement in step 4 of algorithm IPM_Advanced. Instead of examining only one symbol, all symbols with indexing support might be considered. In fact, it is possible to compute a ranking S_1, S_2, ... regarding the score of its subpattern as the primary criterion and its selectivity as the secondary. Then, the top-k indexes are used in an iterative fashion to continuously thin out the candidate list of events. This allows early pruning of events and reduces the number of calls of BaseLine. For example, if an event e is a match of symbol S_1, but there is no match of symbol S_2 in a certain distance range of e, it is safe to remove e from the list of candidate events. Similar to the temporal scope, it is easy to introduce a lower bound and an upper bound to define a temporal range for events to match the two symbols. Moreover, this creates an interesting optimization problem: In general, the cost for using an index will increase (because the selectivity increases), but the possible gain of an index will decrease (because the number of remaining candidate events decreases). Thus, it is not advisable to use all available indexes, but to stop after a certain number of indexes. To determine the value of an index, we track the ratio of the cost for querying an index and the number of pruned events times the expected cost for a call of BaseLine. As long as we observe that this ratio is greater than 1, the use of the index turned out to be beneficial. Otherwise, if we observe no benefit anymore, we stop the further use of indexes and call BaseLine for each *TScope* in the remaining candidate list.

Complexity Analysis. In the following, we present an I/O analysis of our generic approach that uses BaseLine as a black box method and our ranking-based selection of indexes. Since BaseLine is treated as a black box in the implementation, we do not consider the CPU cost of BaseLine for our analysis. Instead, we assume a linear I/O cost model in the number of index entries and events accessed for answering a query. Obviously, a more advanced analytical study would be possible for specific instances of BaseLine, but that is beyond the scope of the article.

Let us assume that after the selection of k indexes our method stops, $k \geq 1$. Thus, the number of accessed entries during traversal of the first $k - 1$ indexes was less than the number of pruned events. Since the total number of pruned events is at most N, the total number of entries accessed during the first $k - 1$ index retrievals is also at most N. The cost of the traversal of the last index is higher than its benefit. However, there are at most N entries accessed during the traversal of the kth index. It follows that there are at most $2N$ accesses in total. Thus, our method is a factor of two worse than BaseLine in the worst case while being able to speed up the BaseLine solution through indexing in other cases.

9 EXPERIMENTAL EVALUATION

This section presents a selection of important results from an extensive performance comparison. Section 9.1 describes the experimental setup; Section 9.2 evaluates the storage layout of

Table 3. Indicators of the Data Sets

Data set	#Events	Bytes/Event	Compression	minimum *tc*	Input Processing (s)
SoccerGame	24,278,210	68	17.26%	0.476	46.88
CDS	20,000,000	72	68.46 %	0.7306	0.59
SafeCast	35,770,979	446	84.40 %	0.9971	168.49
BerlinMOD	56,129,943	48	59.05 %	0.9996	180.02
LinearRoad	886,664,544	35	41,61 %	0.7267	295.42
SmartHomes	4,055,508,721	41	71,23 %	0.1526	3809.33

ChronicleDB, and Section 9.3 evaluates the query performance. Section 9.4 compares ChronicleDB with open-source (Cassandra, ElasticSearch, Apache Kafka, RocksDB), commercial (InfluxDB), and academic systems (LogBase in combination with CR-index). After evaluating our sliding window aggregation in Section 9.5, we discuss index-supported pattern matching in Section 9.6. Finally, we will analyze our strategies for out-of-order handling and load scheduling in Section 9.7 and Section 9.8, respectively.

9.1 Experimental Setup

All experiments were conducted on a workstation equipped with an AMD Ryzen7 2700X CPU (8 cores, 16 threads), 32GB of memory, a 1TB HDD, and a 512GB NVMe SSD, running an Ubuntu Linux (18.04, kernel version 4.16). We run various experiments to identify the impact of parameters on the performance of ChronicleDB and to choose the best settings. The L-block size and the size of macro blocks are two parameters we set to 8KiB and 32KiB, respectively. Smaller block sizes (e.g., 4KiB) as well as larger block sizes (e.g., 32KiB) perform slightly inferior to our standard settings. Because we measured only a minor impact of these parameters, we do not detail these results. Unless specified otherwise, the experiments with ChronicleDB were conducted with 10% spare for an L-block and without partial indexing on a single worker.

In our experiments, we used six data sets termed SoccerGame, BerlinMOD, SafeCast, CDS, LinearRoad, and SmartHomes. *SoccerGame* is a real data set, extracted from the DEBS Grand Challenge 2013 data [5]. The data provides sensor readings of a soccer game. We used the data set obtained from the ball. *BerlinMOD* is a semi-synthetic data set, sampled from a collection of taxi trips in Berlin. We used the precalculated trips data available at [1]. *SafeCast* [13], a citizen science project, contains spatio-temporal radiation data. *CDS* is a synthetic data set with eight numerical attributes and a timestamp. This data set was generated based on real-world CPU data [16]. *LinearRoad* is a data generator for car trips on an expressway. We generated data simulating five hours of traffic on a single expressway with 1K active cars per hour. Each active car reports its state every second, leading to about 887M events. Finally, *Smart Homes* is a data set from the DEBS Grand Challenge 2014 data [2], which is based on smart plug recordings that can be deployed in households. Table 3 reports important properties: the number of events, the size of an event, the compression rate, the minimum temporal correlation among all attributes of the corresponding data set, and the time for reading the input into memory. As these data sets are ordered by time, they are not suited for out-of-order experiments. We will postpone the generation of out-of-order data to Section 9.7.

9.2 Compression and Recovery

First, we evaluate the performance of the storage layout presented in Section 4 in the following denoted as *ChronicleDB layout*. We compare ChronicleDB layout with a completely separated storage layout (*separate layout*), storing the address information of the blocks from the data of the TAB$^+$-tree in a separate file.

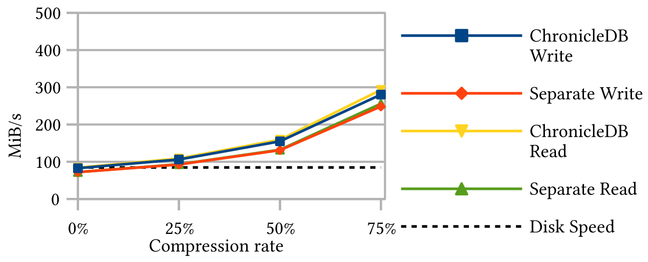


Fig. 16. Throughput as a function of the compression rate for different storage layouts.

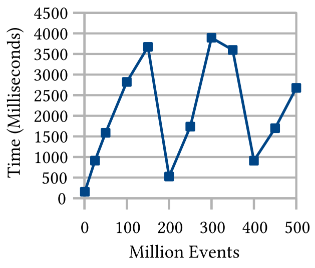


Fig. 17. ATT recovery time after ingesting various numbers of events.

To evaluate the storage layout, we measured the impact of compression. We run experiments with a hypothetical compression rate that is constant for all blocks. Figure 16 shows that the write as well as the read performance of ChronicleDB layout scales almost linearly with the compression rate.

In addition, we measured the sequential disk speed by writing data without address information and without compression. This results in 82.81MiB/s. Without compression, ChronicleDB achieves almost the same results, while the write performance of the separate layout drops to 72.06MiB/s. This shows the advantage of ChronicleDB layout where data blocks and ATT-blocks are kept interleaved in a single file.

Finally, we discuss the recovery times of ChronicleDB layout. For this experiment, we triggered a system crash after ingesting a predefined number of events from the SmartHomes data set and measured the recovery time for the ATT. The results are depicted as a function of the number of ingested events in Figure 17. Note that even for millions of events, the recovery of the storage layout requires only a few seconds. In general, the costs for recovering the storage layout are $\log_B(N)$ in the best and $B \cdot \log_B(N)$ in the worst case, with B being the number of references stored in an ATT-block. Recap: to recover from failures, we need to re-create the right flank of the ATT; this means, to restore one node per tree level holding at max B references. In the best case, we only need to read one reference per level, while in the worst case every node of the flank contained B references such that all of them need to be read. In Figure 17, both cases can be observed: At 200M and 400M events, the right flank contained only few references; while at 150M and 300M, the nodes were almost completely filled.

9.3 Query Performance

At first, we discuss the TAB$^+$-tree lightweight indexing performance for the data set SoccerGame. Therefore, we report the impact of the number of (lightweight) indexed attributes on the overall ingestion performance, depicted in Figure 18. There is a very mild linear performance decrease in the number of indexed attributes, because of the capacity reduction of internal nodes in the TAB$^+$-tree.

9.3.1 Time-travel & Temporal Aggregation Queries. Next, we discuss the query times for time-travel queries as well as temporal aggregation queries in ChronicleDB while varying the temporal range (selectivity). We used the SoccerGame data set in this experiment. Figure 19 depicts the total processing time as a function of selectivity. The performance of the time travel queries decreases linear in the selectivity, while the logarithmic performance of the aggregate query seems to be constant.

9.3.2 Secondary Indexes. Finally, we evaluate the index capabilities of ChronicleDB. For this purpose, we ingested the SoccerGame data set into ChronicleDB twice. First, we used lightweight

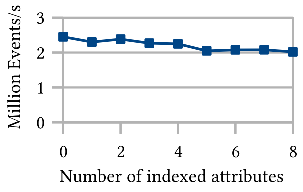


Fig. 18. Write throughput as a function of the number of indexed attributes.

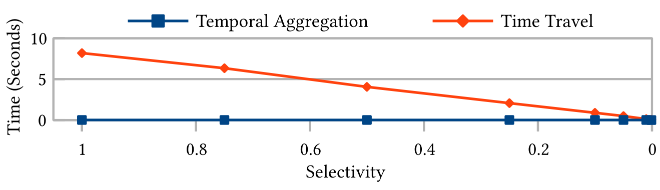


Fig. 19. Performance evaluation of time-travel queries and temporal aggregation queries on SoccerGame.

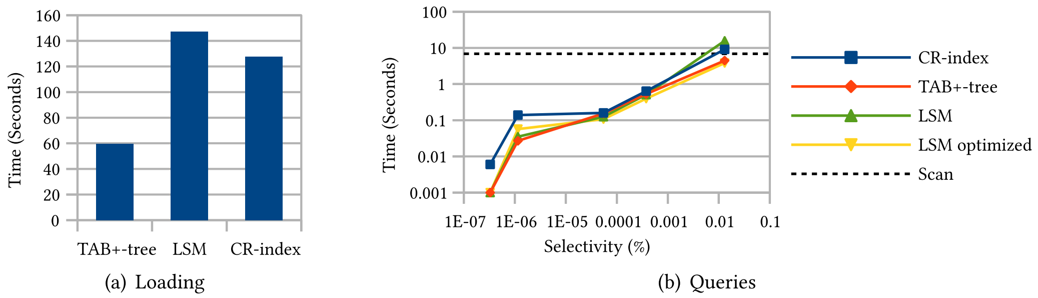


Fig. 20. Secondary index evaluation on SoccerGame in LogBase (CR-index) and in ChronicleDB with lightweight index (TAB$^+$-tree) and with secondary index (LSM).

indexing (TAB$^+$-tree) on *velocity*, the attribute with smallest temporal correlation. In the second build, we used a secondary index (LSM-tree) on the same attribute. As a point of comparison, we also ingested the dataset into LogBase while utilizing the CR-index. We used the configuration from [44] and deployed LogBase on the local file system of the same machine as ChronicleDB.

As shown in Figure 20(a), the build time in ChronicleDB is substantially higher in case an LSM-tree is generated. Furthermore, in terms of loading, CR-index is competitive with ChronicleDB, outperforming the LSM-variant, but taking twice as much build time as a native lightweight indexing configuration.

Figure 20(b) presents our query results for ChronicleDB with TAB$^+$-tree, the CR-index, an LSM-tree, and an optimized LSM tree, respectively (note the log-scale). For the optimized LSM tree variant, we first query the index and then reorder the access to the primary index to enforce a sequential read pattern. We also depict the time for a full range scan in ChronicleDB as a dashed line. In summary, LogBase with CR-index is inferior to ChronicleDB. For low selectivity, the TAB$^+$-tree is faster than LSM and CR-Index and slightly faster than the optimized LSM variant. In case of the basic LSM implementation, the low temporal correlation of *velocity* introduces many random accesses resulting in poor query performance. To find the break-even in query performance between LSM and TAB$^+$-tree, the selectivity as well as the temporal correlation have to be taken into account. But due to the high cost for index creation, LSM is only justified for highly read-intensive applications.

9.4 Benchmarking ChronicleDB

In the following, we compare ChronicleDB with InfluxDB (v1.6.3), Cassandra (v3.6), Elastic-Search (v6.4.1), LogBase, Apache Kafka (v2.0.0), and RocksDB (v5.14.2), all running on the same machine as ChronicleDB for a variety of data sets. Each system writes its data onto the NVMe SSD. Because RocksDB is ChronicleDB's closest competitor, we additionally compare their

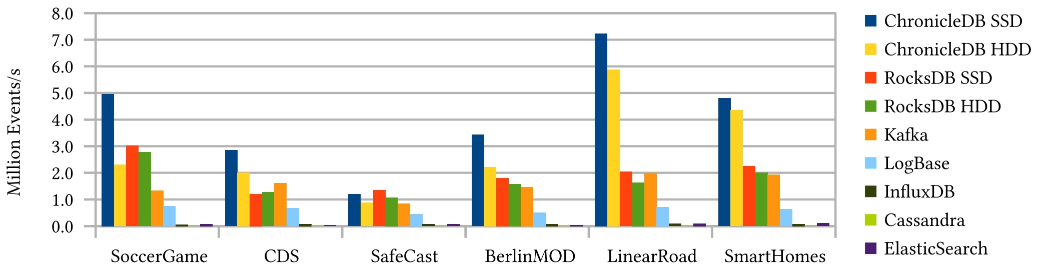


Fig. 21. Ingestion throughput benchmark.

performance on HDDs. We used the Java client libraries for InfluxDB,[2] Cassandra,[3] ElasticSearch,[4] and RocksDB.[5] For InfluxDB, we used batches of 5K events for writes and chunks of 10K events for reads. Furthermore, we configured batch interval of 1s to reduce network overhead. Similarily, we used 10K event batches for both reading and writing in ElasticSearch. Cassandra does not offer batches for performance improvement. For LogBase, we applied the suggested configuration from [44] again. We used the standard configurations for Apache Kafka, which includes batches of 10K and occasional creation of new log segments. For RocksDB, we used the configuration applied for its respective sequential benchmark.[6] In particular, this configuration uses a Vector MemTable as its LSM data structure, which allows for maximum sequential write speed. In case of all key-value stores, we used system time as key and the entire event as value to boost the respective performance. Since processing LinearRoad and SmartHomes led to ingestion times of over 8h or even crashes for some systems, we report throughput for those datasets based on the first 60 Million Events for LogBase, InfluxDB, Cassandra, and ElasticSearch.

Figure 21 reports the write throughput in Million Events per second (y-axis) of the systems for our data sets. We did not include here the time for reading and converting the input data (see Table 3). Independent of the storage medium, ChronicleDB clearly outperforms InfluxDB, Cassandra, and ElasticSearch across the board; e.g., in case of the BerlinMOD data set, ChronicleDB is superior by a factor of 44 to InfluxDB and by a factor of over 200 to Cassandra. This is to be expected, since InfluxDB, Cassandra, and ElasticSearch are designed for scalability. Meanwhile, ChronicleDB is optimized for append-only event ingestion. Furthermore, InfluxDB is primarily designed to support single value time series rather than multi-variate event data. Similarly, ElasticSearch is a document store for full-text search queries rather than a storage system for high-rate event workloads. The closest competitors to ChronicleDB are LogBase, Kafka, and RocksDB, because they follow the same design principles. In terms of LogBase and Kafka, the log is the database. However, even on SSD, LogBase fails to achieve the same throughput as ChronicleDB does on HDD. We attribute this to the fact that LogBase is a multi-machine-oriented research prototype built around HBase and MapReduce. In a similiar vein, Kafka is designed to scale and thus has to deal with the respective overhead. For RocksDB, the LSM-based datastore works well for ingestion workloads and even outperforms ChronicleDB on HDDs for the SoccerGame and SafeCast data sets, since the vector data structure does not incur the overhead of ChronicleDB tree structure. However, for larger data sets (BerlinMod, LinearRoad, SmartHomes), the overhead of LSM merging slightly deteriorates its performance. In summary, both RocksDB and ChronicleDB can achieve ingestion speeds of a

[2]https://github.com/influxdb/influxdb-java.
[3]https://github.com/datastax/java-driver.
[4]https://www.elastic.co/guide/en/elasticsearch/client/java-api/6.4/client.html.
[5]https://github.com/facebook/rocksdb/tree/master/java.
[6]https://github.com/facebook/rocksdb/wiki/RocksJava-Performance-on-Flash-Storage.

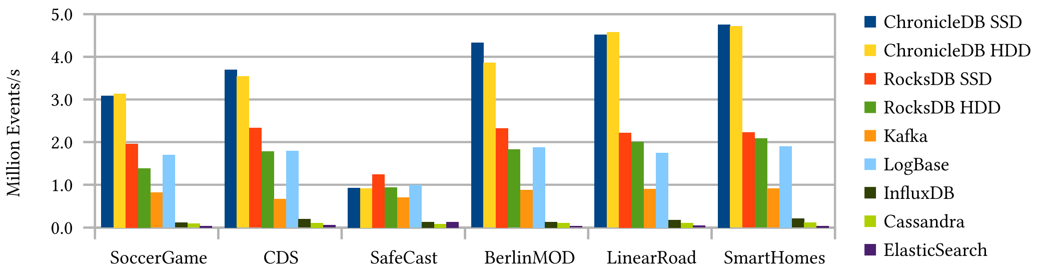


Fig. 22. Read throughput benchmark.

million events per second on a single machine, which should cover most stream-based use cases in terms of recording all data.

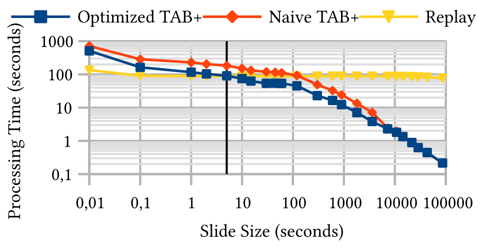
For our read benchmark (cf. Figure 22), we performed a full relation scan for all data sets and systems (configured as mentioned above), because replaying the history as it was is an important feature of a historical data store. In this case, individual systems can play to their strength, e.g., overall ElasticSearch performs worse than InfluxDB, but is better at handling the larger, more string-based dataset SafeCast. Nevertheless, the results mirror the ingestion benchmark: ChronicleDB, LogBase, Kafka, and RocksDB perform best, since this is the type of workload they are designed for. In particular, RocksDB generally outperforms LogBase, since it has a more mature code base and doesn't suffer the drawbacks of HBase. Furthermore, RocksDB also outperforms Apache Kafka, because it doesn't suffer from the drawbacks of distribution. This is not surprising, since RocksDB is the default database Kafka Streams uses to store operator states on each worker node.[7] In turn, ChronicleDB outperforms RocksDB for most workloads, because the replay is essentially a large range query and range queries are the classic case for which B$^+$-trees are better suited than LSM data structures. This is due to the multiple levels that LSM has to visit and the inclusion of linkage between sibling nodes in B$^+$-tree. In summary, ChronicleDB delivers superior performance for replaying event streams.

9.5 Sliding Window Aggregates

We compared the processing time of both TAB$^+$-tree based approaches (optimized, naive) and a pure replay-based solution. Therefore, we created two synthetic event streams with random data. The streams cover one week of data with a frequency of 100 events/second, resulting in 60,480,000 events. The first stream consists of 16-byte-sized events (a single double-valued attribute and the timestamp), which leads to a fan-out of 152 for the inner TAB$^+$-tree nodes and 458 events per leaf node. The events of the second stream are 80 bytes in size (9 double-valued attributes and the timestamp), resulting in a fan-out of 23 and 90 events per leaf. The disk sizes of both streams are 555MB and 2.9GB, respectively. We computed *sum, average*, and *count* aggregates for a single attribute and a window duration of one day. We varied the *slide* parameter from 10ms (1 event per *slide*) up to 1 day (whole window) and measured the execution time for all approaches. ChronicleDB's page cache (LRU) was limited to 50 pages to highlight impact of the disk access pattern induced by each of the approaches.

The results of this experiment are shown in Figure 23. As expected, the execution time of the replay implementation is nearly constant. Both, TAB$^+$-tree based approaches are significantly slower for very small values of *slide*, because the TAB$^+$-tree is traversed for every output event. However, our optimized approach outperforms replay for *slide* values greater than 5s (80-byte events) and 2s

[7]https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Internal+Data+Management.



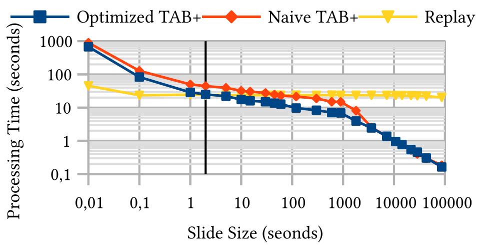


Fig. 23. Processing time of sliding time window aggregation for varying values of *slide* using event sizes of 80 bytes (left) and 16 bytes (right).

Table 4. Symbol selectivities used in the Pattern Matching Evaluation

Symbol	Q1	Q2	Q3
A	0.0001	0.01	0.025
B	0.1	0.01	0.025
C	1.0	1.0	1.0
D	0.1	0.009	0.0001

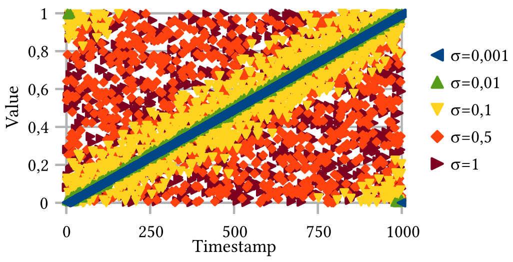


Fig. 24. Data distribution for varying values of σ.

(18-byte events), because the temporal aggregation query performs less merges per *slide*. The naive approach needs much larger *slide* values to outperform the replay version: 120s (80-byte events) and 30s (16-byte events). For *slide* values of 300s (5mins) and 900s (15mins) the skip of macro blocks is large enough, so the processing time for both TAB$^+$-tree based solutions drops faster than for smaller values of *slide*. The vertical black line shows the break-even point estimated by our cost model and clearly confirms its validity.

9.6 Temporal Pattern Matching

In this experiment, we discuss the results of executing the three pattern matching algorithms (BaseLine, IPM_Basic, IPM_Advanced) presented in Section 8.2 for the pattern ABC^*D with a window-size of 1h. Every symbol represents a range condition on one of the attributes of the stream. The experiments consists of three classes of queries with different selectivities for the symbols (cf. Table 4). The index-based algorithms IPM_Basic and IPM_Advanced utilized either lightweight (TAB$^+$-tree) or secondary indexing (LSM tree) to support the search of sub-patterns. In addition, we distinguish for IPM_Advanced the utilization of one (LSM-1) or two indices (LSM-2).

To reveal the performance differences of the algorithms, we used synthetic data sets with a varying *temporal correlation*. A data set contains 50M events that consist of 9 double-valued attributes $A_1, \ldots, A_9$ and a timestamp T (summing up to 80 bytes per event). The events (e_i) are generated as follows:

$$e_i.T = i$$
$$e_i.A_j \sim \mathcal{N}\left(\frac{i}{50M}, \sigma^2\right), for\ j \in \{1, \ldots, 9\}$$

That is, timestamps are incremented by one and the attribute values follow a normal distribution with a mean $\mu = i/50M$, which is a linear function of the timestamp i. This allows the temporal

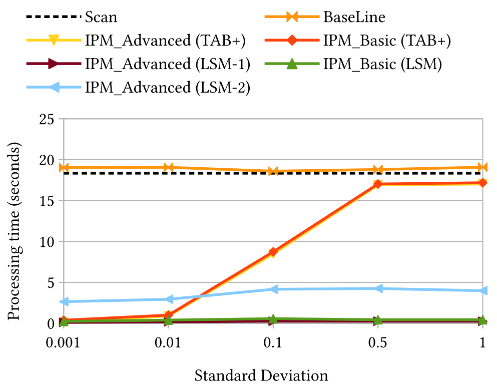


Fig. 25. Pattern matching processing time for Q1 with varying σ.

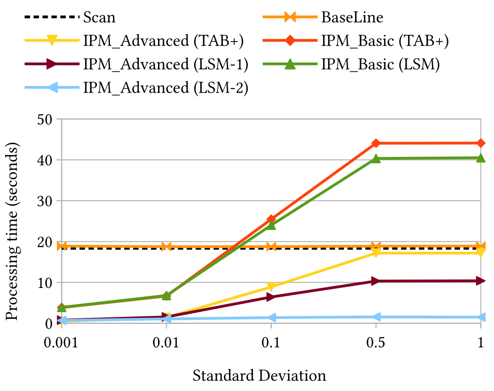


Fig. 26. Pattern matching processing time for Q2 with varying σ.

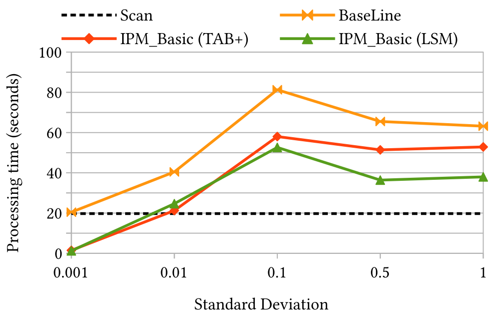


Fig. 27. Pattern matching processing time for Q3 with varying σ.

correlation of the data set to be controlled by changing the standard deviation of the normal distribution. Figure 24 illustrates the value distribution for various values of σ. Small σ-values (0.001, 0.01) generate highly temporally correlated data, while for large σ-values the data is almost uniformly distributed.

The plots of Figure 25, Figure 26, and Figure 27 show the processing time of the pattern matching queries as a function of the standard deviation for the three query classes Q1, Q2, and Q3, respectively. Before discussing each of the experiments on its own, we want to point out two important results. First, all figures show that our index-based approaches are superior to BaseLine for all of the three query types. One reason is that the cost for the scan over the data, depicted as the dotted line, is often substantially higher than the processing cost of the other methods (see Figure 25). Second, the TAB$^+$-tree offers similar performance to full secondary indexing (LSM) for highly correlated data (low σ) despite its low cost for creation. Only in the case of low correlation (high σ) secondary indexes really pay off.

Q1 (Figure 25) shows no difference between IPM_Basic and IPM_Advanced, because the selectivity of symbol A is so low that there is no gain in checking its neighborhood for a match with subpattern AB. For the same reason and also because of the relatively high selectivity of B, the performance of IPM_Advanced with two indices (LSM-2) is worse than the one using only one index (LSM-1).

This is different for Q2 (Figure 26). Here, D is the most selective symbol, but the combined selectivity of the sub pattern AB is much lower. Consequently using two indexes (IPM_Advanced

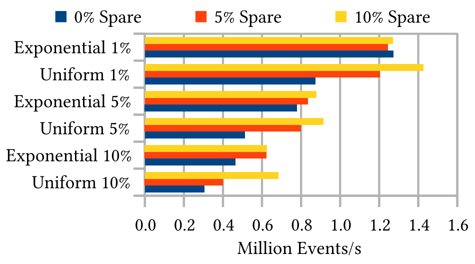


Fig. 28. Out-of-order ingestion performance.

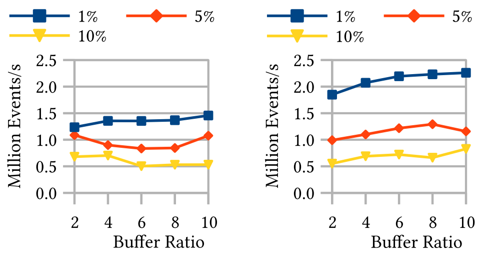


Fig. 29. Evaluation of the buffer ratio impact: uniform (left) and exponential (right).

(LSM-2)) is superior to all other approaches. For $\sigma > 0.07$, IPM_Basic falls behind BaseLine for the following reason: Due to its low selectivity among all attributes, D is used for the index lookup. However, compared to the selectivity of the sub pattern AB, it selects roughly 10 times more elements. As a result, BaseLine is invoked 10 times more often than in the other index-based approaches, which is more costly than a single replay operation.

Q3 (Figure 27) showcases a hard-to-handle case for automaton-based pattern matching: The most selective symbol occurs at the end of the pattern, while the preceding symbols have a much higher selectivity. In this case, all approaches suffer from maintaining many automaton runs that need to be updated with every incoming event. However, IPM_Basic(TAB+) and IPM_Basic(LSM) are superior to BaseLine, since the temporal scope heavily reduces the amount of data to read from external storage. We have not presented the results for IPM_Advanced, because they do not behave differently from IPM_Basic.

9.7 Out-of-order Data

To examine the out-of-order insertion performance, we modified the timestamps of the SoccerGame data as follows: Out-of-order insertions take place in bulk after every 10K insertions of chronological events. The delay of out-of-order data is restricted to the time interval since the last out-of-order bulk insertion, simulating late arrivals from a sensor. We consider two distributions for a delay: uniform and exponential. For an exponential distribution, smaller delays occur more often than longer ones (with an expected delay of 40ms).

Figure 28 shows the results of our experiments with different fractions of out-of-order data as well as varying amounts of spare for uniform and exponential delay distribution. Out-of-order inserts are expensive. The throughput for 10% out-of-order is smaller by a factor of two than that of 1%. As expected, exponential distribution overall performs slightly better during ingestion because of higher locality in the buffer. In general, leaving spare space improves ingestion performance as well as read performance, because larger reorganizations can be avoided and there is no need to remap blocks.

We also measured the influence of the ratio between the range of out-of-order data and the buffer size, depicted in Figure 29. For example, a buffer ratio of 2 indicates that the buffer covers half of the out-of-order data. The size of the out-of-order buffer does not have a significant influence on the overall performance, as the system is CPU-bound due to overheads for compression and serialization.

9.8 Adaptive System Time Mode

For the experimental evaluation of our system time-based load-scheduling components, we generated multiple runs of 100M 64-byte events. Then, we inserted those into the ChronicleDB and

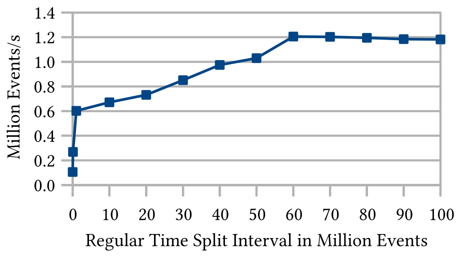


Fig. 30. Performing splits at different intervals.

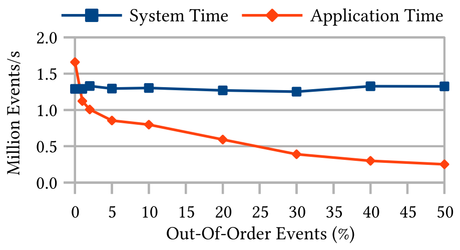


Fig. 31. Ingestion speed impact of system time mode.

measured the maximum possible ingestion rate for different configurations of out-of-order data and delays, averaging the results.

First, we measured the impact of splits. For this purpose, we performed regular splits on various time intervals ranging from 10K (5K splits) to 100M milliseconds (no split). The results are depicted in Figure 30. We can easily see that too frequent splits (<10M) have a huge negative impact on the ingestion performance. At the same time, an occasional split is very manageable (>25M) confirming our strategy of rarely doing irregular splits.

Taking into consideration that a split does consume resources, we next evaluate the performance gains of a *systime time mode* tree. We vary the amount of out-of-order data from 0 to 50% and choose an exponential delay pattern. The results in Figure 31 show that *systime time mode* is not preferable for a small out-of-order rate of under 1%, because attaching new timestamps to each event adds some complexity. However, for larger out-of-order rates *systime time mode* significantly outperforms the standard ChronicleDB configuration.

The results of an active switch by the load scheduler are displayed in Figure 32. For this experiment, we started to insert events with an out-of-order rate of about 1% into a standard ChronicleDB configuration. At about 40% of the inserted events, we triggered a sudden increase of out-of-order events to 15%. We configured the queue-size parameter δ in a way so it triggers a switch at about the 60% mark. We ran three configurations: a switch to *system time mode* tree, a switch to just another application time tree, and no switch at all. It can be observed that before the switch all three configurations obviously suffer the same significant dip in ingestion speed at the 40% mark. For this experiment, we input events at maximum rate and the switch is induced synchronously, i.e., the queue piles up while we create new files and data structures. Therefore, both switch configurations suffer another dip between the 50% and 60% mark due to the creation of a new tree. For lower, steady data rates and asynchronous switches, this second dip can be hidden. Nevertheless, the *system time mode* has a huge boost in performance that even improves upon the previously acceptable rate before the out-of-order increase. Meanwhile, the other configurations stay at a lower ingestion rate.

In summary, our *system time mode* load scheduler can improve the ingestion rate of ChronicleDB on the fly during high-load, high out-of-order rate situations. However, the user has to choose a suitable delta so there are enough resources to first perform an irregular switch before the ingestion performance will improve.

For measuring our auxiliary goal of keeping the query performance as respectable as possible during *system time mode*, we repeated the experiment above with 50M 64-byte events and a constant out-of-order occurrence probability of 15%. Afterwards, we inserted another event that should have occurred at about the 25M event mark, creating a large outlier. In Figure 33, we provide

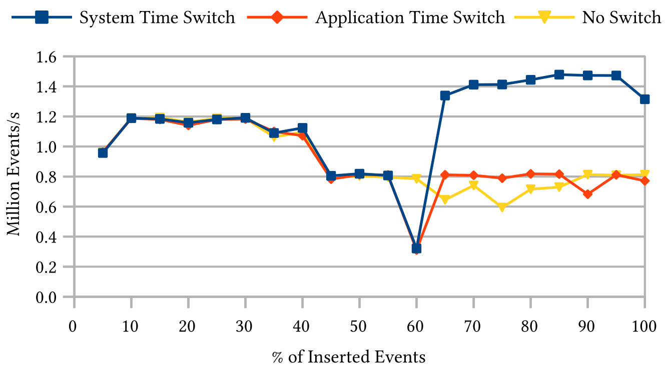


Fig. 32. Adaptive load scheduling with irregular splits.

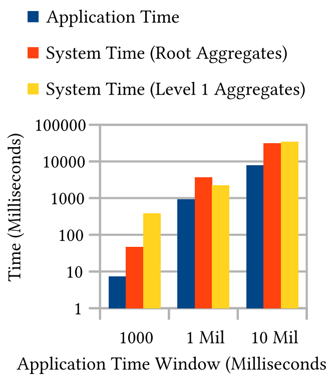


Fig. 33. Application time range queries with temporal outliers.

results for three different tree configurations when processing application time range queries. Those queries cover the middle of the application time and span equally into preceding and succeeding application times around that mark. The processing time is displayed on the y-axis. Obviously, an application time tree vastly outperforms other solutions. Furthermore, we note that the optimization strategy for system time trees of visiting the first level is a detriment for small window queries, since the additional random I/O to recognize the opportunity for in-memory sorting outweighs the cost of directly sorting small results sets. However, for moderate windows, the first-level aggregates show an improvement of 50% processing time over using the inaccurate root-level aggregate, which does not recognize the presence of the large outlier.

10 CONCLUSION & FUTURE WORK

Database systems for event streams have to handle very high ingestion rates in new applications related to IoT. Not all of these applications allow large-scale distributed database systems, but require a tightly integrated database solution in the application code. In this article, we presented ChronicleDB, a new type of centralized database system that exploits the temporal arrival order of events. We discussed in detail its storage management, indexing support, load scheduling, and recovery capabilities. Due to its dedicated system design, our experimental results showed a great superiority of ChronicleDB in comparison to distributed systems such as Cassandra and InfluxDB, which are widely used for write-intensive applications like the management of event streams.

Further, we enabled ChronicleDB to replay continuous queries from event-stream processing system and showed how its lightweight indexing can improve the processing time of sliding window aggregation and pattern matching queries. In the future, we plan to explore how our lightweight indexing can be used to process window joins and other event-processing queries and develop a cost model to always pick the best strategy—analogous to windowed aggregation and pattern matching.

So far, we put our focus on the careful design of ChronicleDB as a centralized system and showed that simply making standard systems scalable is not the right answer. However, it would also be possible to adapt ChronicleDB for a distributed environment. In our current and future work, we examine how to exploit the benefits of distributed frameworks for write-intensive applications. We even believe that ChronicleDB's write-once policy and its storage layout integrates well into distributed file systems like HDFS.

REFERENCES

[1] 2011. BerlinMOD. Retrieved from: http://dna.fernuni-hagen.de/secondo/BerlinMOD/BerlinMOD.html.

[2] 2014. DEBS Grand Challenge 2014. Retrieved from http://debs.org/debs-2014-smart-homes/.

[3] 2015. KairosDB. Retrieved from https://kairosdb.github.io/.

[4] 2016. Apache Cassandra. Retrieved from http://cassandra.apache.org/.

[5] 2016. DEBS Grand Challenge 2013. Retrieved December 10, 2017 from http://debs.org/debs-2013-grand-challenge-soccer-monitoring/.

[6] 2016. ISO/IEC TR 19075-5:2016, Information technology — Database languages — SQL Technical Reports — Part 5: Row Pattern Recognition in SQL. Retrieved from: http://standards.iso.org/ittf/PubliclyAvailableStandards/.

[7] 2017. Apache Hadoop. Retrieved from: http://hadoop.apache.org/.

[8] 2017. Apache HBase. Retrieved from: http://hbase.apache.org/.

[9] 2017. InfluxDB. Retrieved from: https://github.com/influxdata/influxdb.

[10] 2017. LZ4 Compression. Retrieved from: https://github.com/lz4/lz4.

[11] 2017. OpenTSDB. Retrieved from: http://opentsdb.net/.

[12] 2017. PostgreSQL. Retrieved from: http://www.postgresql.org/.

[13] 2017. SafeCast. Retrieved from: http://blog.safecast.org/data/.

[14] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving relations for cache performance. In *Proceedings of the VLDB*. 169–180.

[15] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. 2007. Consistent streaming through time: A vision for event stream processing. In *Proceedings of the CIDR*. 363–374.

[16] Lars Baumgärtner, Christian Strack, Bastian Hoßbach, Marc Seidemann, Bernhard Seeger, and Bernd Freisleben. 2015. Complex event processing for reactive security monitoring in virtualized computer systems. In *Proceedings of the DEBS*. 22–33.

[17] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[18] Zhao Cao, Shimin Chen, Feifei Li, Min Wang, and Xiaoyang Sean Wang. 2013. LogKV: Exploiting key-value stores for log processing. In *Proceedings of the CIDR 2013*.

[19] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.

[20] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate sharing for user-defined windows. In *Proceedings of the CIKM*. 1201–1210.

[21] Surajit Chaudhuri and Umeshwar Dayal. 1997. An overview of data warehousing and OLAP technology. *SIGMOD Rec.* 26, 1 (1997), 65–74.

[22] Alan Demers, Johannes Gehrke, Mingsheng Hong, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker White. 2007. Cayuga: A general purpose event monitoring system. In *Proceedings of the CIDR*. 412–422.

[23] Luca Deri, Simone Mainardi, and Francesco Fusco. 2012. TSDB: A compressed database for time series. In *Proceedings of the TMA*. 143–156.

[24] Lukasz Golab, Theodore Johnson, J. Spencer Seidel, and Vladislav Shkapenyuk. 2009. Stream warehousing with datadepot. In *Proceedings of the SIGMOD*. ACM, 847–854.

[25] Bastian Hoßbach, Nikolaus Glombiewski, Andreas Morgen, Franz Ritter, and Bernhard Seeger. 2013. JEPC: The Java event processing connectivity. *Daten.-Spekt.* 13, 3 (2013), 167–178.

[26] Theodore Johnson and Vladislav Shkapenyuk. 2015. Data stream warehousing in tidalrace. In *Proceedings of the CIDR*.

[27] B. Kuszmaul. 2010. *How TokuDB Fractal Tree Indexes Work*. Technical Report. TokuTek.

[28] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.* 34, 1 (2005), 39–44.

[29] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the SIGMOD*. 311–322.

[30] Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal Claypool. 2009. Sequence pattern query processing over out-of-order event streams. In *Proceedings of the ICDE*. IEEE, 784–795.

[31] Charles Loboz, Slawek Smyl, and Suman Nath. 2010. DataGarage: Warehousing massive performance data on commodity servers. *PVLDB* 3, 1–2 (2010), 1447–1458.

[32] Yuan Mei and Samuel Madden. 2009. ZStream: A cost-based query processor for adaptively detecting composite events categories and subject descriptors. In *Proceedings of the SIGMOD*. 193–206.

[33] Guido Moerkotte. 1998. Small materialized aggregates: A lightweight index structure for data warehousing. In *Proceedings of the VLDB*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 476–487.

[34] Peter Muth, Patrick O'Neil, Achim Pick, and Gerhard Weikum. 2000. The LHAM log-structured history data access method. *VLDB J.* 8, 3-4 (2000), 199–221.

[35] 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.

[36] 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. *PVLDB* 8, 12 (2015), 1816–1827.

[37] Jun Rao and Kenneth A. Ross. 2000. Making B+- trees cache conscious in main memory. In *Proceedings of the SIGMOD*. 475–486.

[38] Marc Seidemann and Bernhard Seeger. 2017. ChronicleDB: A high-performance event store. In *Proceedings of the EDBT*. 144–155.

[39] Kanat Tangwongsan and Martin Hirzel. 2015. General incremental sliding-window aggregation. *PVLDB* 8, 7 (2015), 702–713.

[40] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2017. Low-latency sliding-window aggregation in worst-case constant time. In *Proceedings of the DEBS*. 66–77.

[41] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. Knowl. Data Eng.* 15, 3 (2003), 555–568.

[42] Fabio Valdés and Ralf Hartmut Güting. 2014. Index-supported pattern matching on symbolic trajectories. In *Proceedings of the SIGSPATIAL 2014*. ACM Press, New York, New York, 53–62.

[43] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. 2012. LogBase: A scalable log-structured database system in the cloud. *PVLDB* 5, 10 (2012), 1004–1015.

[44] Sheng Wang, David Maier, and Beng Chin Ooi. 2014. Lightweight indexing of observational data in log-structured storage. In *PVLDB*, Vol. 7. 529–540.

[45] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *Proceedings of the SIGMOD*. ACM, 407–418.

[46] Jun Yang and Jennifer Widom. 2003. Incremental computation and maintenance of temporal aggregates. *VLDB J.* 12, 3 (2003), 262–283.

[47] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *Proceedings of the SIGMOD*. ACM, 217–228.

Received December 2017; revised March 2019; accepted May 2019