ChronicleDB: A High-Performance Event Store

Marc Seidemann
Database Systems Group
University of Marburg, Germany
seidemann@informatik.uni-marburg.de

Bernhard Seeger
Database Systems Group
University of Marburg, Germany
seeger@informatik.uni-marburg.de

ABSTRACT

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 to deliver 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 paper, we present ChronicleDB, a novel database system with a well-designed storage layout to achieve high write-performance under fluctuating data rates and powerful indexing capabilities to support ad-hoc 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.

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 like 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 or self-driving cars. Furthermore, scientific observational (sensor) data is crucial in various research, spanning from climate research over animal tracking to IT security monitoring. 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 application require to maintain 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.

There is a current 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 [9], 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 like Cassandra [1] or HBase [3], 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 47 in terms of write-performance on a single node. In other words: Cassandra would need at least 47 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 does not suit. For example, in [14], 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.

In order to overcome these deficiencies, particularly the poor write performance, log-only systems have emerged where the log is the only data repository [32, 16, 33]. 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 compromising both lightweight temporal indexing as well as full secondary indexing to speed-up queries on non-temporal dimensions.
- In order to support out-of-order arrival of events, we developed a hybrid logging approach between our log storage and traditional logging.
- We compare the performance of ChronicleDB with commercial, open-source and academic systems in our experiments using real event data.

The remainder of this paper 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. Recovery issues are examined in Section 6. In Section 7, we evaluate our system experimentally and Section 8 concludes the paper.

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.

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* [20]. DataDepot is a data warehouse for streaming data, running on top of a relational system. Hence, DataDepot achieves a throughput of only about 10 MiB/s. *Tidalrace* [22], the successor of DataDepot, pursues a distributed storage approach and reaches data rates of up to 500.000 records per second, which still does not compete with ChronicleDB running on a single machine. *DataGarage* [25] 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.

The most popular NoSQL systems in the context of storing events are Cassandra [1] and HBase [3]. As shown in our experimental section, ChronicleDB clearly outperforms Cassandra when running on a central system.

A representative for log storage systems is *LogBase* [32], 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 based on HDFS [2] and simply writes data to logs. 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.

LogKV [16] utilizes distributed key-value stores to process event log files. In fact, Cassandra [1] was used as underlying key-value store. In experiments, the authors achieved a throughput of 28K events/s ingestion bandwidth per worker node, each consisting of an Intel Xeon X5675 system with 96GB memory and a 7200rpm SAS drive, connected via a 1GB/s network. In comparison to ChronicleDB, the ingestion rate is lower by about a factor of 100.

The third class of storage solutions ChronicleDB partly relates to is that of time series databases. A representative of this class is *tsdb* [18]. Similar to our approach, the authors use a LZ compression for loss-less data compression. In contrast to our approach, time series databases (including tsdb) usually assume that data arrives at every tick.

Gorilla [29] proposes 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 data.

OpenTSDB [8] and KairosDB [6] are time series database systems on top of HBase and Cassandra. Thus, they have the same deficiencies as their underlying systems.

The mostly related storage system is *InfluxDB* [5], a new open-source time series database solution. As will be discussed in our experimental section, the performance and functionality of InfluxDB on a central system are inferior to ChronicleDB.

Indexing Aggregation in the context of temporal databases has been extensively investigated before in the database community. Widom et al. [34] 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 certain attribute.

More recent research concentrates on observational data and event data. The recently proposed CR-index by Wang et al. [33] is based on LogBase [32] and also utilizes 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 a single index. The cost for writing events is lower when the event is written once. In addition, queries on multiple attributes do not need to access multiple indexes.

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.

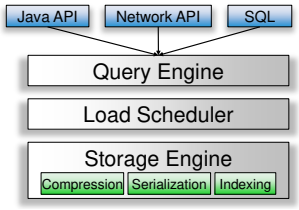


Figure 1: Layers of the ChronicleDB architecture.

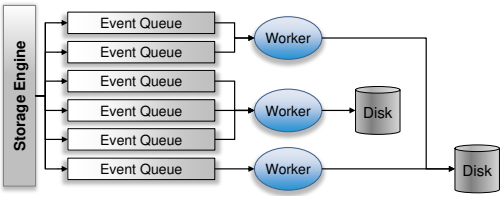


Figure 2: Example of a ChronicleDB topology.

3.1 Requirements

ChronicleDB aims at supporting temporal-relational events, which consist of a timestamp t and several non-temporal, primitive attributes a_i. So, 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 in order to keep-up with high and fluctuating event rates. ChronicleDB should be as economical as possible in order 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 mean time, 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 applications [13]. They can happen, e.g., if sensors are sending their events in batches based on asynchronous clocks 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., *alls ssh logins within the last day from a certain IP range*.

3.2 Architecture Overview

Figure 1 depicts a high level view of ChronicleDB's architecture. In this paper, we focus on the lower layer, i.e., the storage engine and the indexing capabilities of ChronicleDB. Nevertheless, we also give a short description of the other layers, which will be discussed in more detail in future work.

The storage engine of ChronicleDB logically consists of three components: event queues, workers and disks. Basically, *event queues* have two functions. Primarily, they decouple the ingestion of events from further processing. As a side effect, they also compensate chronologically out-of-order event insertion. The *workers* are responsible for writing data to disks and therefore reside in their own threads. Each worker processes the events from its assigned event queues, as long as they are non-empty. All events of a stream are separately stored on one of the worker's dedicated 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. Figure 2 shows an exemplary topology for seven streams, three workers and two disks.

3.3 Design Principles & System Features

In ChronicleDB, the major design principle is: *the log is the database*. We avoid costly additional logging as the considered append-only scenario does not cause costly random I/Os and therefore does not incur buffering strategies with no-force writes. Only in case of out-of-order arrival of events, we have to deviate from this paradigm as we want to keep the data ordered with respect to application time.

ChronicleDB is implemented in Java and is integrated into the JEPC event processing platform [21]. 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. The main objective is write-optimization, thus we focused on fast compression with reasonable compression rate. Hence, we chose LZ4 [7] as 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. At first, Section 4.1 describes the problem statement. Section 4.2 introduces the components of the storage layout. Finally, we present the overall layout.

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.

We decided to use blocks as compression unit, see Section 4.2.1. Naïvly, the physical offset within a file could be used as identifier (ID) for block addressing. In case of fixed block sizes, the position of a block could be easily computed. But due to compression, blocks are of variable size. So, the final physical offset cannot be computed in advance, which

also causes a rethink of the indexing architecture, see Section 5.2 and 5.3.

Therefore, a smart address mapping was required while offering low storage overhead. To the best of our knowledge, there is no existing solution that satisfies these requirements. For example, TokuFS [19] proposes a compressed file system for micro data, but only reaches about 35% of sequential disk speed in the presented experiments. So, we developed a novel storage layout that meets our requirements while offering 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 straight-forward 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 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 paper 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 [12]. 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 a 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 6.

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, 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. In order to avoid these cost, 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 Translation Lookaside Buffer

Translation of virtual to physical memory addresses is a fundamental task in computer systems, typically conducted in hardware in the memory management unit (MMU).

In ChronicleDB, we followed a similar approach, but used a software-based *translation lookaside buffer* (TLB). In ChronicleDB, a virtual address is the ID of an L-block. The physical address is the position of the 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 TLB only has to store the physical addresses, while the virtual addresses are implicitly given by the position. This TLB structure is related to the CSB$^+$-tree [31], 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, TLB entries are also managed in blocks, denoted as TLB-blocks. The size of a TLB-block is equal to the size of an L-block. If a TLB-block is filled, it is written to disk. Each TLB-block contains the same amount of entries. E.g., for an L-block size of 8 KiB and 64 bit address size, a TLB-block can contain up to 1020 entries (considering meta data). To support a large address space, we organize TLB-blocks hierarchically in a tree. The resulting TLB tree does not require explicit routing information for address lookup.

Algorithm 1 outlines the address lookup. It starts at the root of the TLB tree, which is always and solely kept in memory. Thanks to the consecutive ID numbering, the index of the corresponding 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. To speed up address translation, we use a write buffer for each level of the TLB. Furthermore, at least the index levels of the TLB are kept in memory to improve read performance. This should be possible as the size of the TLB index (without the leaf level) is N/b^2 for N C-blocks and L-block size b.

4.3 Overall Layout

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

In a first possible solution, from the query processing perspective, a TLB-block with k mapping entries should ideally address its k succeeding C-blocks. Unfortunately, this requires either to buffer these k blocks with the risk of data loss in case of a system failure, or to perform a random I/O

Algorithm 1: TLB-block Address Lookup

Input : ID id of the requested block, entries per TLB-block b and TLB height l
Output: The physical address of the C-block

$index \leftarrow \left\lfloor \frac{id}{b^l} \right\rfloor \mod b;$
for $i = l - 1$ **to** 1 **do**
  $address \leftarrow TLB_{i+1}[index]$;
  load TLB_i from $address$;
  $index \leftarrow \left\lfloor \frac{id}{b^i} \right\rfloor \mod b;$
end
return $TLB_0[id \mod b]$

to write the TLB-block after writing the k C-blocks. So, there is a tradeoff between performance and safety issues.

In a second solution, we solve this problem by placing the TLB-block behind the data it refers to. So, a TLB-block with k entries always refers to its immediately **preceding** k C-blocks. In 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 8 MiB memory in case of 8 KiB 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 5.3. Removal of old data is explained in Section 5.4. In Section 5.5, we propose our temporal partitioning and load scheduling approach. 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.

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 [16], 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

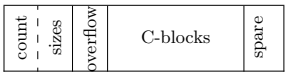


Figure 3: Macro block layout.

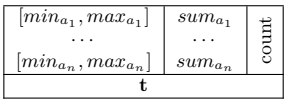


Figure 4: Index entry layout of TAB$^+$-tree.

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, 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 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

For query processing and recovery issues, we use a linking *in both directions* at *every level* of the tree. We utilize this linking to speed-up query processing as well as to enhance recovery, discussed in Section 6.

To improve query processing, we leverage temporal correlation of event data. 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.

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 *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

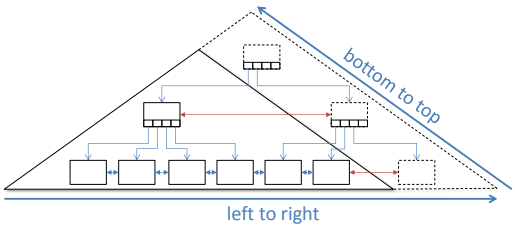


Figure 5: TAB$^+$-tree construction.

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 which 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 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, secondary indexes incur high overheads. Hence, ChronicleDB's load scheduler temporally deactivates secondary indexing in case of peak loads, as will be discussed in Section 5.5.

The most popular log-structured index used, e.g., in HBase [3] or TokuDB [23], is the LSM-tree [28]. 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 [15], which can be maintained very efficiently.

5.4 Time-Splitting

ChronicleDB is a hybrid between OLTP and OLAP database. 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 last week or month ([17]). ChronicleDB offers the 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 is interested in weekly sales statistics, so he 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. This enables aggregation queries even in constant time.

Though ChronicleDB aims at long-term storage, it also addresses deletion and reduction of ancient data. Removing events from the TAB$^+$-tree could be realized via cutting off its left flank. However, this would result in costly I/O operations, as the data has to be removed event-by-event. Even more critical: secondary indexes have to be kept consistent. Thus, all events contained in the left flank of the TAB$^+$-tree also have to be removed from the secondary index. Instead of removing data event-by-event, ChronicleDB supports the removal of outdated events at the granularity of regular time splits. Thus, only the corresponding files have to be deleted (logically). Alternatively, outdated events can be thinned out or condensed via aggregation, leveraging the aggregates in the TAB$^+$-tree again.

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, 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 [27]. Ancient data is removed from ChronicleDB in whole regular time splits, indicated in Figure 6 before rs_1.

5.5 Partial Indexing

Fluctuating data rates are always a very challenging problem, especially in the context of sensor data. We addressed this problem with load scheduling to ensure maximum ingestion speed. In times of 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. More advanced strategies are possible taking into account the access frequency of the 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. This results in (unplanned) partial indexes, which have to be synchronized with the primary index again. Therefore, we introduce a second kind of time split, termed *irregular split*.

Figure 6 shows an example where regular splits are prefixed with rs_x, irregular splits with is_x. For each time split, P_x denotes a TAB$^+$-tree, S_x a secondary index. At is_3, secondary indexation has been switched off due to a system overload. Therefore, the primary index is split, too. If the system load decreases, secondary indexes are switched on again. Re-activation only takes place at regular splits. In this example, secondary indexation 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 $[rs_4, rs_5]$.

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 (start) 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

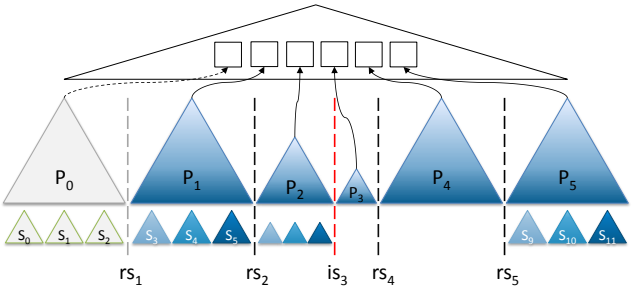


Figure 6: Index scheduling example.

required leaves.

In addition, we also utilize restrictions on non-temporal attributes specified in the query 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, the node is skipped.

Algorithm 2: 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);
s.push(0);
while !s.isEmpty() do
    n ← s.pop();
    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 ≥ t_s then
                output n [i ];
            end
        else if Intersection (n [i ],intervals) then
            s.push(n);
            s.push(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
```

5.6.2 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, the aggregation information per node can be utilized. If a node in the TAB$^+$-tree is fully covered by the query range, ChronicleDB can exploit the node's aggregate value (covering all of its child nodes). If the node's time interval is intersected by the given query range, the query proceeds with its qualifying child nodes. Therefore, also temporal aggregation queries are answered in logarithmic time.

5.6.3 Secondary queries in TAB$^+$-tree

Secondary queries can utilize the inherent lightweight indexing of the TAB$^+$-tree. Algorithm 2 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 query results; in fact, query processing in ChronicleDB is demand-driven. 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.

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 asynchronous clocks, network delays and faulty devices cause exceptional out-of-order arrival of events. This problem is well-known in event processing [13], but also has a serious impact on the design of ChronicleDB. There are two basic solutions for dealing with out-of-order arrivals of events. First, we could change the notion of time in the TAB$^+$-tree. Instead of using application time as the primary attribute for indexing, we could use system time. By definition, the events are then always in correct order because an event item receives its timestamp at arrival in ChronicleDB. Furthermore, application time should be used as an additional attribute indexed in a lightweight fashion within the TAB$^+$-tree. This causes additional cost in query processing, in particular for aggregate queries. The second solution still maintains the TAB$^+$-tree as an index on application time (as described in the previous sections). We will pursue this approach in the following.

5.7.1 Out-of-order Buffers

In order to deal with out-of-order, we introduce the Algorithm 3 that 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, 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.

While the sorted queue serves to leverage temporal locality, we also target at leveraging physical locality in the storage layout. Therefore, an additional buffer in combination with a write-ahead log [26] and no-force write strategy is introduced for the TAB$^+$-tree .

Without any further modifications, this approach would still cause serious problems in the TAB$^+$-tree. First of all, 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. In order 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

Algorithm 3: 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[1].getLast().timestamp then
    // Add e to leaf of the TAB+-tree flank
    flank[0].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
            insert qe from bottom to top into flank;
        end
        Clear log;
    end
end
```

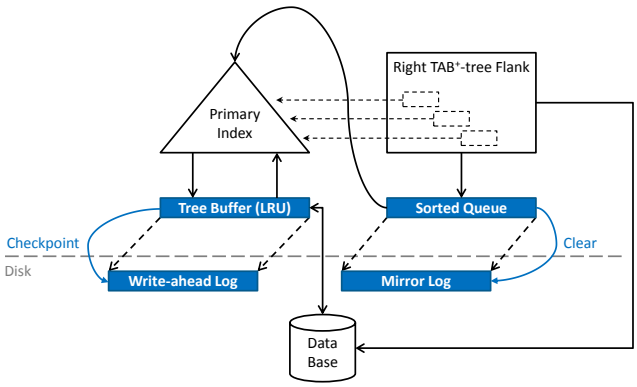


Figure 7: Buffer layout and data flow for out-of-order data.

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 which 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, 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. In order 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, we use the timestamp to search for the event in the primary index. For all other blocks we still use the direct linkage.

6. 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 and finally the logs are processed to transfer the system into a consistent state again.

6.1 Storage Layout Recovery

In ChronicleDB, the most critical part of the storage layout is its address translation, i.e., the TLB. As the root and the right flank of the TLB are only kept in memory, any information about block address translation is lost in case of a system crash. Rebuilding the TLB 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).

In order to support fast reconstruction of the TLB's right flank, we introduce references within the TLB. As only the recently created part of the TLB has to be restored, recovery is performed from the end of the database to its start. Each TLB-block keeps a reference to its previous TLB-block on the same level. Given the last successfully written TLB-block, its predecessor can be directly accessed. The recovery has to scan all TLB-blocks that are children of the last (and therefore lost) TLB-block in the parent level. Then, recovery continues with the next level. To support the direct access to upper levels, TLB-blocks additionally store a reference to its parent's predecessor TLB-block. Thus, these references implicitly create checkpoints for each level. Figure 8 shows the linking of TLB-blocks. For presentation purposes, we assume two address entries per TLB-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 TLB recovery is outlined in Algorithm 4. In case of a crash, the last written TLB-block is seeked on disk. This is simple as the size of a macro block is a multiple of TLB-block size. There are two possibilities for the classification of the last written L-block: either it is a TLB-block or it is part of a macro block. In the latter case, the previous L-block is read until the last successfully written TLB-block is found. The upper bound for the number of L-blocks to be read before finding a TLB-block is the number of entries per TLB-block. After having located the last successfully written TLB-block the recovery of the TLB continues by leveraging the introduced references.

The predecessors of the last written TLB-blocks are located until the parent reference is different. The corresponding references of these TLB-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 TLB, the number of entries to be read is limited by the number of entries per TLB-block.

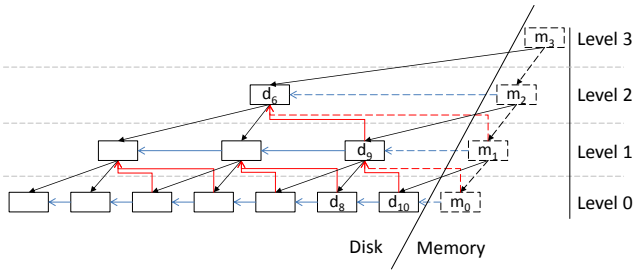


Figure 8: TLB structure with recovery references.

Figure 8 gives an example. In case of a system failure, the TLB-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}.

Algorithm 4: TLB Recovery

```
Input: Database size in bytes s, L-block size in bytes b
b_addr ← ⌊s/b⌋ * b ;    // Last complete block address
block ← read(b_addr);
// Lookup last TLB-block
while block is not a TLB-block do
    b_addr ← b_addr − b;
    block ← read(b_addr);
end
// Rebuild TLB
while block.prev != null do
    prev ← read(block.prev) ; // Read previous entry
    if prev.prevParent != block.prevParent then
        // Switch to the next higher TLB level
        block ← read(block.prevParent);
    else
        // Restore the reference in the new
           parent entry
        Add b_addr to TLB_{block.level+1};
    end
end
```

6.2 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 TLB recovery and starts scanning the data base 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.

6.3 Log Recovery

Finally, the consistency of the data base has to be ensured. This step only matters in case of previously occurred out-of-order events. At first, the write-ahead log is processed from start to end. For each log entry, its LSN is compared with the LSN of the block it refers to. If the LSN of the block is smaller than the entry's LSN, the associated event is regularly inserted into the TAB$^+$-tree. Finally, the sorted queue is restored by scanning the mirror log.

7. EXPERIMENTAL EVALUATION

This section presents a selection of important results from an extensive performance comparison. Section 7.1 describes the experimental setup, Section 7.2 evaluates the storage layout of ChronicleDB and Section 7.3 evaluates the query performance. Section 7.4 compares ChronicleDB with open-source (Cassandra), commercial (InfluxDB) academic systems (LogBase in combination with CR-index). Finally, Section 7.5 investigates the performance impact of out-of-order data.

7.1 Experimental Setup

All experiments were conducted on a Windows 7 desktop computer with Intel I7 2600 quad-core CPU at 3.4 GHz and 8 GB DDR3 RAM, equipped with an 1 TB HDD and 128 GB SSD. The latter is only used for writing the out-of-order logs. We run various experiments to identify the impact of parameters on the performance of ChronicleDB and to chose the best settings. The L-block size and the size of macro blocks are two parameters we set to 8 KiB and 32 KiB, respectively. Smaller block sizes (e.g. 4 KiB) as well as larger block sizes (e.g. 32 KiB) 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 where conducted with 10 % spare for an L-block and without partial indexing on a single worker.

In our experiments we used four data sets termed CDS, BerlinMod, DEBS and SafeCast. *CDS* is a synthetic data set with eight numerical attributes and a timestamp. This data set was generated based on real-world cpu data [14]. *DEBS* is a real data set, extracted from the DEBS Grand Challenge 2013 data [11]. 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 pre-calculated trips data available at [4]. *SafeCast* [10] contains spatio-temporal radiation data collected by the community. We extracted spatial and temporal attributes as well as the radiation. Table 1 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 7.5.

7.2 Compression and Recovery

First of all, 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.

In order to evaluate the storage layout, we measured the impact of compression. We run experiments with a hy-

Table 1: Indicators of the data sets.

Data set	#Events	Bytes/Event	Compression	minimum tc	Input Processing (s)
DEBS	24,278,210	76	34.37%	0.476	53.14
BerlinMOD	56,129,943	48	71.14 %	0.9996	285.655
SafeCast	40,193,450	36	64.08 %	0.9622	354.093
CDS	20,000,000	72	68,36 %	0.869	0.618

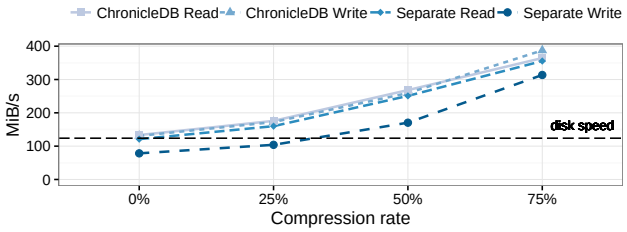


Figure 9: Throughput as a function of the compression rate for different storage layouts.

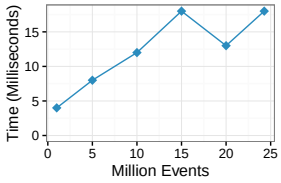


Figure 10: TLB recovery time after ingesting various numbers of events.

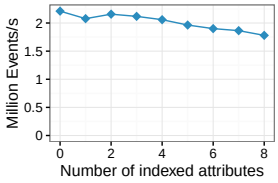


Figure 11: Write throughput as a function of the number of indexed attributes.

pothetical compression rate that is constant for all blocks. Figure 9 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 123.89 MiB/s that matches sequential disk speed. Without compression ChronicleDB achieves almost the same results, while the write performance of the separate layout drops to 71.59 MiB/s. This shows the advantage of ChronicleDB layout where data blocks and TLB blocks are kept interleaved in a single file.

Finally, we discuss the recovery times of ChronicleDB layout. Therefore, we triggered a system crash after ingesting a predefined number of events from DEBS and measured the recovery time for the TLB. The results are depicted as a function of the number of ingested events in Figure 10. Note that recovery of the storage layout requires only a few milliseconds, independent of the number of events. The recovery time is not a perfect monotonic function because it is determined by the fill degree of the nodes from the right flank of the TLB.

7.3 Query Performance

At first, we discuss the TAB$^+$-tree lightweight indexing performance for the data set CDS. Therefore, we report the impact of the number of (lightweight) indexed attributes on the overall ingestion performance, depicted in Figure 11. 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.

7.3.1 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 DEBS data set in this experiment. Figure 12 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.

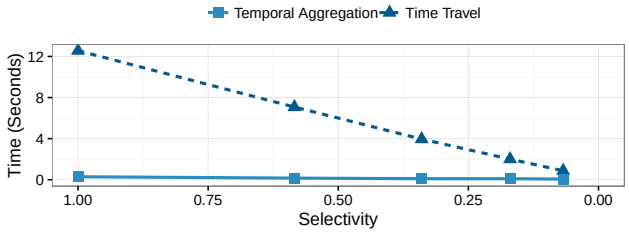
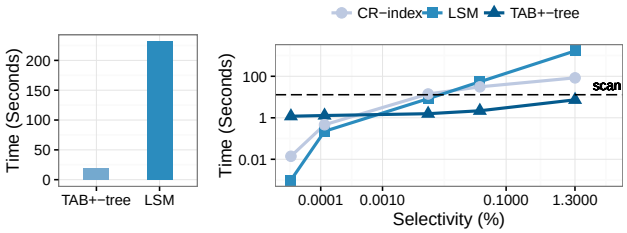


Figure 12: Performance evaluation of time-travel queries and temporal aggregation queries on DEBS.

7.3.2 Secondary Indexes

Finally, we evaluate the index capabilities of ChronicleDB. Therefore, we ingested the DEBS data set into ChronicleDB twice. First, we used lightweight indexing (TAB$^+$-tree) on *velocity*, the attribute with smallest temporal correlation. In the second built, we used a secondary index (LSM-tree) on the same attribute. As shown in Figure 13a, the build time is substantially higher in case a LSM-tree is generated.

Figure 13b presents our query results for ChronicleDB with TAB$^+$-tree and LSM-tree respectively (note the log-scale). Additionally, we compared the results with the CR-index in LogBase. Therefore, we used the configuration from [33] and deployed LogBase on the local file system of the same machine as ChronicleDB. 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 very low selectivity, the secondary LSM index in ChronicleDB performs best, slightly better than the CR-index. In contrast to CR-index, TAB$^+$-tree is not fully kept in memory, which explains the lower query performance for very low selectivities. In case of higher selectivities, the TAB$^+$-tree is significantly faster than both LSM and CR-Index. In case of LSM, 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



(a) Loading (b) Queries

Figure 13: Secondary index evaluation on DEBS in LogBase (CR-index) and in ChronicleDB with lightweight index (TAB$^+$-tree) and with secondary index (LSM).

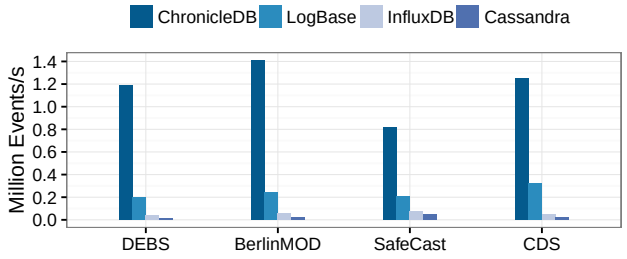


Figure 14: Ingestion throughput benchmark.

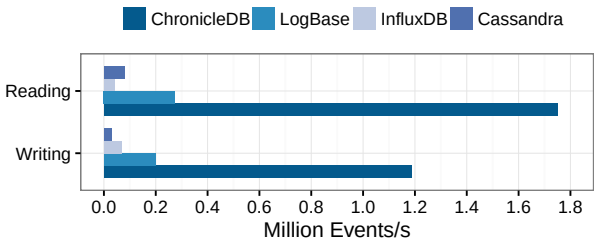


Figure 15: Write and read throughput comparison with LogBase, Cassandra and InfluxDB on DEBS.

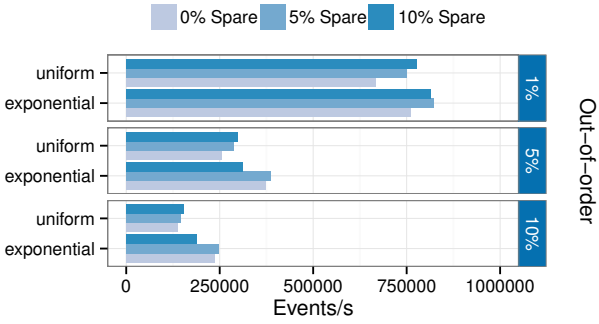


Figure 16: Out-of-order ingestion performance

cost for index creation, the LSM is only justified for highly read-intensive applications.

7.4 Benchmarking ChronicleDB

In the following, we compare ChronicleDB with sInfluxDB (v0.9), Cassandra (v2.0.14) and LogBase, all running on the same machine as ChronicleDB. In terms of write-performance as well as read throughput, Cassandra is currently one of the fastest representatives of distributed key-value stores (see Rabl et al. [30]). For InfluxDB we used batches of 5K events and a batch interval of one second to reduce network overhead. Cassandra does not offer batches for performance improvement. We used the JAVA client libraries for InfluxDB[1] and Cassandra[2]. For LogBase, we applied the suggested configuration from [33] again. Figure 14 reports the throughput of the four systems for our data sets. We did not include here the time for reading and converting the input data, see Table 1. In summary, ChronicleDB clearly outperforms Cassandra, InfluxDB and Logbase. In case of CDS, ChronicleDB is superior to Cassandra and InfluxDB by a factor of 50 and 22, respectively. For LogBase, the speedup is still more than a factor of three.

We also report the performance of full relation scans (exemplary for DEBS), as replaying of historical data is an important feature of a historical data store. In case of InfluxDB, we used only half of the data due to limitations regarding the response size of a query. As presented in Figure 15, ChronicleDB outperforms LogBase by a factor of 5, Cassandra and InfluxDB by a factor of 22 and 43, respectively.

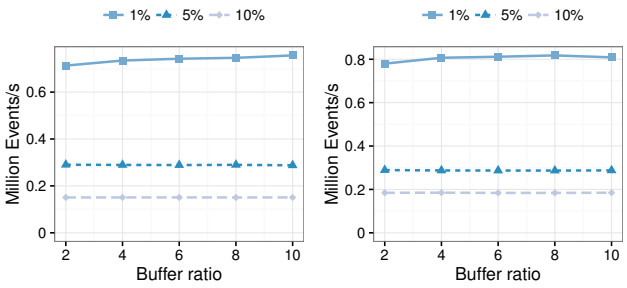
[1]https://github.com/influxdb/influxdb-java
[2]https://github.com/datastax/java-driver

7.5 Out-of-order Data

In order to examine the out-of-order insertion performance, we modified the timestamps of the CDS 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 40 ms).

Figure 16 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 three than that of 1%. Nevertheless, even for an out-of-order rate of 10%, ChronicleDB outperforms InfluxDB by more than an order of magnitude. As expected, exponential distribution performs slightly better during ingestion because of higher locality in the buffer. The read performance is very similar for all approaches at about 1.4M events per second. In general, sparing 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 17. For example, a buffer ratio of 2 indicates that the



(a) Uniform (b) Exponential

Figure 17: Evaluation of the buffer ratio impact.

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.

8. CONCLUSION & FUTURE WORK

The design of a database system for event streams is nowadays important to tackle the very high ingestion rates in new application 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 paper, 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 and recovery capabilities. Due to its dedicated system design, our experimental results showed a great superiority of ChronicleDB in comparison to distributed systems like Cassandra and InfluxDB that are widely used for write-intensive applications like the management of event streams.

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, there is no reason for not using ChronicleDB in 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 suits well to distributed file systems like HDFS.

9. REFERENCES

[1] Apache Cassandra. http://cassandra.apache.org/.

[2] Apache Hadoop. http://hadoop.apache.org/.

[3] Apache HBase. http://hbase.apache.org/.

[4] BerlinMOD. http://dna.fernuni-hagen.de/secondo/BerlinMOD/BerlinMOD.html.

[5] InfluxDB. https://github.com/influxdata/influxdb.

[6] KairosDB. https://kairosdb.github.io/.

[7] LZ4 Compression. https://code.google.com/p/lz4/.

[8] OpenTSDB. http://opentsdb.net/.

[9] PostgreSQL. http://www.postgresql.org/.

[10] SafeCast. http://blog.safecast.org/data/.

[11] DEBS Grand Challenge 2013. http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails, 2013.

[12] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001.

[13] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, pages 363–374, 2007.

[14] L. Baumgärtner, C. Strack, B. Hoßbach, M. Seidemann, B. Seeger, and B. Freisleben. Complex Event Processing for Reactive Security Monitoring in Virtualized Computer Systems. In *DEBS*, pages 22–33, 2015.

[15] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13(7):422–426, 1970.

[16] Z. Cao, S. Chen, F. Li, M. Wang, and X. S. Wang. LogKV: Exploiting Key-Value Stores for Log Processing. In *CIDR*, 2013.

[17] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[18] L. Deri, S. Mainardi, and F. Fusco. Tsdb: A compressed database for time series. In *TMA*, pages 143–156, 2012.

[19] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. The TokuFS Streaming File System. In *HotStorage*, pages 14–14, 2012.

[20] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream Warehousing with DataDepot. In *SIGMOD*, pages 847–854, New York, NY, USA, 2009. ACM.

[21] B. Hoßbach, N. Glombiewski, A. Morgen, F. Ritter, and B. Seeger. JEPC: The Java Event Processing Connectivity. *Datenbank-Spektrum*, 13(3):167–178, 2013.

[22] T. Johnson and V. Shkapenyuk. Data Stream Warehousing in Tidalrace. In *CIDR*, 2015.

[23] B. Kuszmaul. How TokuDB Fractal Tree Indexes Work. Technical report, TokuTek, 2010.

[24] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, Dec. 1981.

[25] C. Loboz, S. Smyl, and S. Nath. DataGarage: Warehousing Massive Performance Data on Commodity Servers. *PVLDB*, 3(1-2):1447–1458, 2010.

[26] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[27] P. Muth, P. O'Neil, A. Pick, and G. Weikum. The LHAM Log-structured History Data Access Method. *The VLDB Journal*, 8(3-4):199–221, 2000.

[28] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[29] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A Fast, Scalable, In-memory Time Series Database. *PVLDB*, 8(12):1816–1827, 2015.

[30] T. Rabl, S. Gómez-Villamor, M. Sadoghi, V. Muntés-Mulero, H.-A. Jacobsen, and S. Mankovskii. Solving Big Data Challenges for Enterprise Application Performance Management. *PVLDB*, 5(12):1724–1735, 2012.

[31] J. Rao and K. A. Ross. Making B+- Trees Cache Conscious in Main Memory. In *SIGMOD*, pages 475–486, 2000.

[32] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi. LogBase: A Scalable Log-structured Database System in the Cloud. *PVLDB*, 5(10):1004–1015, 2012.

[33] S. Wang, D. Maier, and B. C. Ooi. Lightweight Indexing of Observational Data in Log-Structured Storage. In *PVLDB*, volume 7, pages 529–540, 2014.

[34] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. *The VLDB Journal*, 12(3):262–283, 2003.