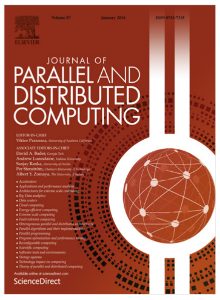


FluteDB: An efficient and scalable in-memory time series database for sensor-cloud

Chen Li, Bo Li, Md Zakirul Alam Bhuiyan, Lihong Wang, Jinghui Si, Guanyu Wei, Jianxin Li

PII: S0743-7315(18)30542-2
DOI: https://doi.org/10.1016/j.jpdc.2018.07.021
Reference: YJPDC 3926

To appear in: *J. Parallel Distrib. Comput.*

Received date : 21 March 2018
Revised date : 22 May 2018
Accepted date : 27 July 2018

*Highlights (for review)

- Propose an efficient and scalable in-memory time series database for sensor-cloud -- FluteDB.
- FluteDB presents a novel indexing structure named Triggered Time Series Merge Tree.
- The TTSM tree splits the original large B+ tree structure into several small B+ trees based on the different data ”temperature”.
- The TTSM tree uses the batched append-only mode to insert new data and a special trigger and merging strategy to update the trees’ structure.
- FluteDB adopts a suitable fault tolerant strategy for its own special indexing structure.
- FluteDB designs two targeted and efficient data compress algorithms for time stamps and observation values.

FluteDB: An Efficient and Scalable In-Memory Time Series Database for Sensor-Cloud

Chen Li[1, 2, 3], Bo Li* [1,2], Md Zakirul Alam Bhuiyan[4], Lihong Wang[3], Jinghui Si[1, 2], Guanyu Wei[1, 2], Jianxin Li [1, 2]

[1]Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing, China
[2]SKLSDE Lab, Beihang University, Beijing, China
[3]National Internet Emergency Center, Beijing, China
[4]Department of Computer and Information Sciences, Fordham University, Bronx, NY, USA
E-mail: {lichen, libo, sijh, weigy, lijx}@act.buaa.edu.cn, zakirulalam@gmail.com, wlh@isc.org.cn

Abstract

Recently, with the widespread use of large-scale sensor network, time series data is vastly generated and requires to be processed. However, those traditional databases show their limitations on storage when handling such a large stream data in cloud, and even their actual reliability and availability are also difficult to be guaranteed. To deal with the problem, this paper proposes FluteDB, an efficient and scalable in-memory time series database for sensor-cloud. We adequately analyze the unique characteristics of time series data and its relevant operations to strike the right balance among efficiency, scalability, resources consumption, reliability and availability. Specifically, on basis of the aggregate analysis of root cause for ongoing time series problems, FluteDB targeted optimizes the strategies for key operations in memory and physical storage, at the expense of partial acceptable data precision and consistency. FluteDB's enhanced strategies are primarily comprised of Triggered Time Series Merge Tree (TTSM Tree), time series enhanced cache management and corresponding compression algorithms for different data types. Additionally, to remain high availability, and responsive in the presence of unexpected failures, FluteDB is equipped with a set of particularly effective and appropriate strategies. The validations of all sub-modules have demonstrated that our improved strategies outperform existing methods in real time series environment significantly. Global experimental results also show that the integrated FluteDB reduces query latency by 17x, improves write rate by 98x and saves about 47% storage resources. The average available service time and recovery rate and degree of FluteDB are competitive with the state-of-the-art reliability and availability strategy in real and simulated faults, which demonstrates FluteDB can provide highly stable large-scale data cloud services.

Key words: Time Series; Sensor-Cloud; In-Memory Database; Scalability; Disaster Tolerant;

1. INTRODUCTION

With the appearance and popularization of Internet of Things (IoT), Wireless Sensor Network (WSN), Smart City (SC) and other Internet hot spots, massive time series data is generated continuously and waits for further processing [1,2,8,10]. Among them, many mature mining algorithms have achieved satisfactory applied results in many practical fields by analyzing and extracting specific features from massive time series data, e.g. Real-Time Vehicle Traffic (RTVT), Smart Grid (SG). As the basis of such downstream algorithms, how to store and query time series data efficiently and stably has aroused wide attentions especially in cloud service [12,19,20].

Due to the increasing demand for source data and real-time compute in cloud services, multimedia data sampling devices are widely available, and their sampling intervals have been shortened immensely nowadays, which lead to a vast growth of the scale of time series data directly. Besides, in order to provide highly reliable and available time series services certainly, the necessary preparations for fault tolerant and data recovery for cloud services are indispensable. Some existing works, which are time series specialized, collectively referred to as Time Series Databases (TSDBs), have achieved satisfactory applied effect in real cloud environment [1,2,5,8,10]. Among them, some improved versions arise from the modifications of classic databases, and the rest are targeted designed for time series. To further meet the demands of time series and promote the efficiency, reliability and availability of all sub-modules and entire system, we next specify several detailed and strict constraints.

Write dominate. The most primary requirement for TSDB is that to keep the services stable at an ultra-high write rate [10]. In the real cloud applications, time series services always tend to cope with millions of write requests per second. The existing databases generally optimize their indexing strategy, data persistence, and accumulate cluster performance to satisfy the demand. To further enhance the write performance, it is necessary to optimize the above strategies and explore more efficient ways.

Query management. Because most of the downstream service objects of query operations are periodical monitoring or management systems, the query rate is usually a couple orders of magnitude lower than write rate [1,15]. However, it is still very difficult to achieve efficient query within such large-scale time series data. Existing methods attempt to solve this problem by distinguishing different storage media and indexing structures for storing cold/hot data, which will bring a lot of common side effects (e.g. data redundancy in memory, too long latency for the worst query).

Resource control. Though the hardware prices are generally declining, the usage efficiency of resources in cloud environment (including memory and disk) are also an important indicator for the evaluation of storage services [13,14,16]. The scale of time series data becomes much larger. Take Facebook as an example, it daily produces about 10T of logs, texts and other streaming data. Storing such massive data directly is bound to consume a lot of storage resources. If taking into account the version control, single-writer, append-only and redundancy strategy, the resource utilization of entire system will become much lower. It will reduce the retention time of persistent historical data (cyclic writer), as well as the scale of cached data in memory directly. Therefore, a set of specific data compression algorithms are essential, though at the expense of partially acceptable loss of data precision.

Highly reliability and availability. Reliability and availability are indicators which describe respectively the ability for providing cloud services normally and fault tolerant [8,10]. Existing cluster management approaches can meet the demand to provide services in the face of partial node failure continuously, but how to recover from disaster or fault quickly and provide services again is the key to ensuring the reliability of time-series cloud services.

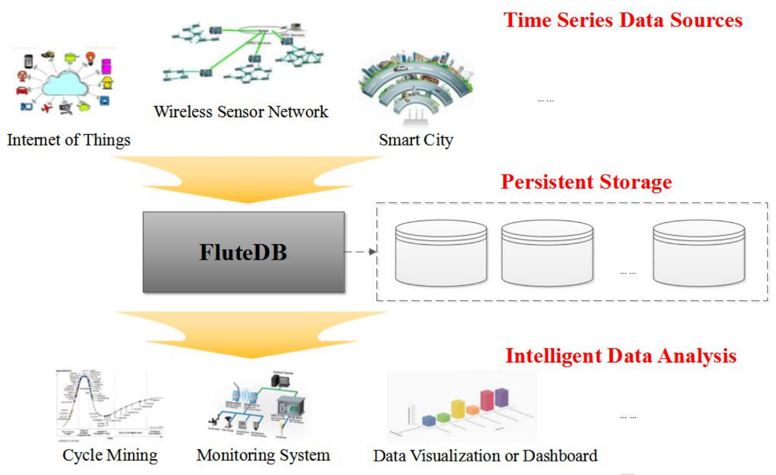


Figure 1: The role of FluteDB in practical application environment.

This paper proposes FluteDB (as an extension of [24]), a novel time series database for sensor-cloud (as shown in Figure 1), which aimed to satisfy mentioned constraints and provide efficient, scalable and stable cloud services. It enhances all of the sub-modules in database based on the aggregate analysis of time series data and its relevant operations. FluteDB also re-adjusts the communication and data exchange modes among memory, hard drives and other resources to keep database running more efficient. All the considerations and designs in FluteDB take fully into account the linearly scalable to ensure it can scale up as needed. Specifically, it has the following novel methods to improve the present situation.

Since time series services are mainly composed of vast majority of write operations and a few point queries and region queries, a novel insight behind FulteDB is that to make the overall architecture more inclined to the optimization of write operation, that is, to exchange for higher write performance at the expense of part performance of query operation or acceptable data precision [1,2]. In order to achieve this goal, FluteDB re-organizes the structure of indexing, and further optimizes the configuration and use of different types of resources. To be specific, combined with the temporal characteristics of time series, FluteDB proposes a new TTSM tree algorithm based on Log-Structure Merge Tree (LSM Tree) [6,9,11]. The TTSM tree divides indexing into two parts based on the distribution of data values. The hot data (newly inserted) is stored in special tree structure in memory, while the cold one is stored on a specific B* tree structure on physical disks. Periodically, partial indexing in memory will be persisted to disk. Since time series data is in a strict ordered sequence, FluteDB designs a more flexible triggering mechanism to determine when to perform the above operations considering the complexity of tree structure merging. Moreover, FluteDB also enhances the corresponding efficient storage structures for different storage media to meet different operational requirements.

Although query operations account for a small percentage, the relevant data service efficiency for upstream tasks also need to be guaranteed. Therefore, as long as the write rate is not affected, the actual query demands are able to be meet indirectly by optimizing the hit rate of both cache and in-memory indexing. FluteDB updates the original cache replacement algorithms and its implementation to guarantee high-value data stay in memory as much as possible based on time series characteristics. Then, vast majority of query operations can be handled in memory, the overall query efficiency will be improved significantly.

FluteDB also presents specific compression algorithms for different data types (Integer, Float, Double, String and etc.) to reduce the resource consumption. In order to maximize the compression ratio of compression algorithms on the basis of acceptable compression and decompression complexity, we adopt a more flexible compression concept and coded format, and only sacrifice partial data coding precision. Besides, FluteDB integrates a complete set of reliability strategies for all sub-modules and entire system, enabling it to recover and continue to provide services in the face of power outage, network outage or other faults and disasters.

As a time series database used in real cloud environments, FluteDB encapsulates data connection management, data manipulation as functional layer on the top of above functions. This layer is designed to conform to the classic structure of traditional distributed database, which provides a scalable guarantee for FluteDB.

By evaluating FluteDB in large-scale cloud storage environment, its writing efficiency, query latency and storage resource consumption significantly outperforms existing methods. And the system's stable running time and disaster recovery effect can be also guaranteed.

The rest of paper is structured as follows. Section 2 introduces the existing databases and the optimization methods for time-series data. Section 3 presents FluteDB's architecture, and further analyzes its design principle and implementaion in detail. In section 4, extensive experiments show that FluteDB obtain better performances than existing systems. Finally, conclusion is summarized in Section 5.

2. Related Work

Since a large number of famous researches focus on analyzing the time series characteristics, continuous interests in management of time series data have been followed closely in field of database for decades [1,2,5,8,10]. At present, people have begun to pay more and more attention to explore how to efficiently and steadily store and query time series data because of its immense growth and valuable implicit information.

In general, the previous TSDBs can be divided into modified versions and re-design versions. By adding plug-ins or updating existing classic databases, the modified versions hijack or apply original data services to complete relevant operations with time series characteristics [1,3,4,5]. The other methods re-design complete bottom-up TSDB systems, including physical storage and upper components [8,10]. These two versions of TSDB are widely used in real sensor-cloud environment and have gained satisfactory practical application effect.

As a scalable and efficient TSDB, OpenTSDB is based on HBase, and serves massive amounts of time series data without losing granularity [5]. Its main design idea is running enough Time Series Daemons (TSDs) to achieve data interaction considering users' data load. Each TSD is independent, and uses the open source database HBase or hosted Google Bigtable service to store and retrieve time series data. Hence, the bottom physical storage strategy is transparent to users. OpenTSDB also has a richer data model for identifying time series, on the basis of which aggregate data quickly to achieve storage resource reduction. In addition, in order to persistent raw data forever, OpenTSDB uses acceptable loss of data accuracy in exchange for cheaper long-term queries and storage consumption.

InfluxDB is a stand-alone open source TSDB with even richer data model than OpenTSDB [2]. Its meta data model is able to not only make data management more convenient, but also effectively improve the efficiency of data query. Meanwhile, InfluxDB also presents a time-structured merge tree as indexing of its storage engine considering time series characteristics, and proposes the concept of shard to manage time series data sectionally. In order to reduce the storage resource used by time series data, InfluxDB presents several various compression strategies. In real time series environment, InfluxDB has achieved great success and has become the most widely used TSDB. However, because of the unsatisfied query performance of disk, the simple strategy for merging indexing, the unoptimized compression algorithms and other obvious drawbacks, InfluxDB also need to be upgraded to improve its service quality.

As a TSDB used by FaceBook to manage its massive time series data, Gorilla is fast, scalable and stable [8]. Unlike other common TSDBs, Gorilla is a scheme for managing and storing time series data in memory. In order to meet the users' requirements, Gorilla stores high value data in memory, and proposes a large number optimization methods for indexing and storage. In Gorilla, the final write rate, query latency and system throughput have increased substantially based on mentioned improvements. In addition, Gorilla proposes a detailed approach to deal with the fault which is used to ensure the reliability of data stored in memory. For long-term data, Gorilla stores it in HBase and tries to add flash storage between memory and HBase to further blur the boundaries of hot and cold data.

Recently, a relational TSDB named LittleTable is proposed [10]. In order to promote the query efficient, it clusters tables in two dimensions (timestamps and hierarchically-delineated key) to optimize the indexing structure. LittleTable further optimizes for time series services by capitalizing on the reduced consistency and durability. Likewise to other systems, LittleTable designs appropriate fault tolerant strategy to ensure dependability of data services combining its applied environment. In Cisco Meraki, Several hundred LittleTable servers are certified to be able to efficiently manage hundreds of TB time series data.

There are numerous studies and applications for storing, querying and managing time series data. Almost all of these works treat time series characteristics as optimized object, and have similar optimized ideas and implementations. To ensure the stability of system, these works formulate their own reliability strategies, which are different from each other because of their different application environment. According to our survey, the storage, query and reliability assurance of time series data still have enough room to be optimized.

3. FluteDB Architecture

In this section, we define the time series data, and analyze its numerical and operating characteristics. On this basis, we present FluteDB's basic architecture, enhanced sub-modules' design concept and their realization.

3.1 Definition and Characteristics of Time Series Data

Time-series data has extensive sources and is valuable for further analyzing and extracting meaningful statistical characteristics from it [1,8]. As a series of data points with fixed meaning in chronological order, time-series data can be represented as $C = (t_1, v_1), (t_2, v_2), \ldots, (t_n, v_n)$, where t_i denotes the sampling time and v_i is the corresponding observation values. As illustrated in Table 1, time series data is different from general stream data, which is mainly embodied in three aspects.

Specifically, since the shorter sampling interval and increasing number of devices, the scale of time series data becomes much larger. The sample values between continuous sampling timestamps are similar and continuous, which means that there are lots of redundant segments in time-series data. By observing the operation logs of database, we find that the composition of time-series data operations includes 97% write operations, and as well as point query or range query.

Table 1: Several real examples for time series data.

Data Source	Time Stamp t_i	Observation Values v_i
Temperature Sensor	1401335396.997	+46.5℃
	1401335397.998	+46.4℃
	1401335399.002	+46.3℃
Smart Grid	1363894780	220.15V, 50Hz, 2.768×10-6 Kw/h
	1363894781	219.95V, 50Hz, 2.883×10-6 Kw/h
	1363894782	220.85V, 50Hz, 2.425×10-6 Kw/h
Road Monitor	2016-09-11T17:20:48.134Z	09204821c7c2.jpg
	2016-09-11T17:20:49.635Z	092049dc62ad.jpg
	2016-09-11T17:20:51.011Z	09205135e380.jpg

3.2 Overall Architecture

The proposed FluteDB is a standard in-memory database whose architecture, strategy, etc. are designed to meet the specific requirements of time series data. In general, FluteDB functions as a write through cache, which involves roughly two major parts to support its use: the upper data services and time series storage engine.

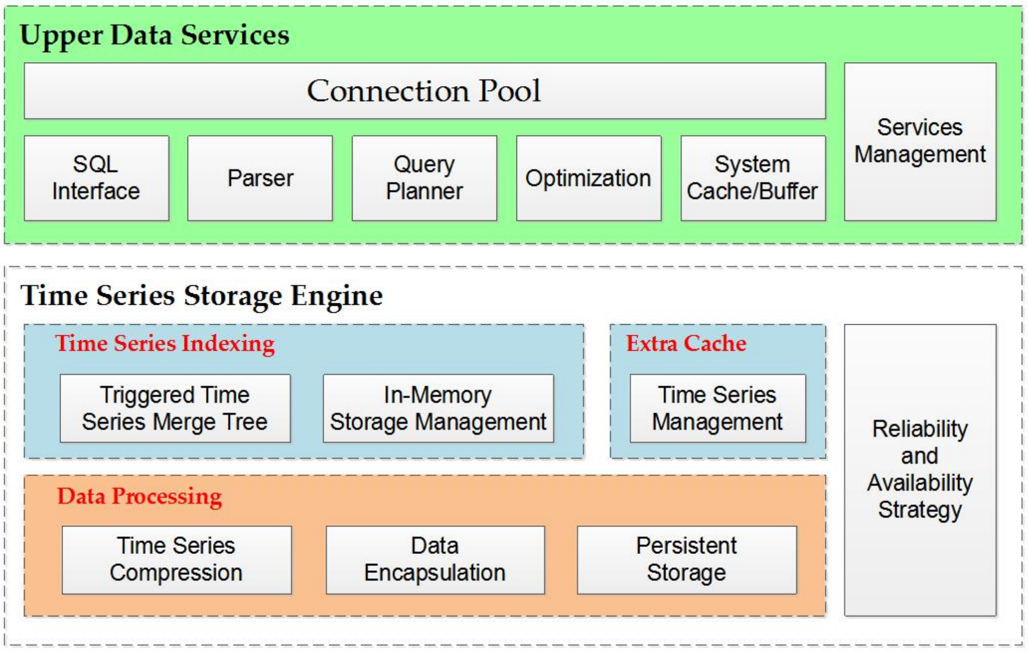


Figure 2: The overall architecture of FluteDB.

As illustrated in Figure 2, the upper data services mainly manage the connections of all external applications, and provide corresponding functions. The components (e.g. operation parse, content based implementation optimization for input operation) in this layer are not different from traditional database. In order to improve the mobility of services, FluteDB also supported some SQL operations.

Meanwhile, the time series storage engine allows us to write, query and manage time series data expediently, and its applied effect will determine the service performance of entire FluteDB most directly. In FluteDB, we re-design the function architecture and related algorithms in memory to meet the constraints mentioned in Section 1. Corresponding, we also adjust the coordination pattern for physical storage and memory to further optimize the resource utilization and system performance. As shown in Figure 2, time series storage engine consists of four parts: time series indexing, extra cache, data processing and reliability and availability strategy, which are enhanced correspondingly based on time series requirements. Thus, FluteDB not only has an efficient applied effect, but also can provide more reliable and available services.

In FluteDB, all of the storage structures and considers are linear, and its nodes do not share any resources or structures. Hence, FluteDB has good horizontal scalability, which means that we can adjust the scale of FluteDB cluster according the real cloud environment by simply increasing or deleting nodes. Of course, in order to achieve efficient entire cluster performance, an appropriate cluster strategy is necessary.

3.3 Time Series Data Insertion and Indexing

As a kind of data with strict ordered of time, the application and analysis value of time series data (the probability of being queried) are inversely proportional to the length of its generated time in most cases [8]. Based on this assumption, existing works label the cold or hot data according to its generated time, and use different storage strategies to deal with the corresponding time series data.

In previous TSDBs, the LSM tree has achieved desired performance since its high write rate and flexible data structure [6,9,11]. In a nutshell, the LSM tree is a B+ tree's variant and consists of an append-only part and a smaller update-in-place tree, which are stored in disk and memory respectively. The in-memory indexing is used to manage new data quickly, and the full B+ tree in disk stores all previous data persistently. On a regular basis, the database merges the two partial indexings and updates them on disk for persistent storage in batches. In this way, based on the assumption of cold/hot data, the vast majority of the insert and query operations have been done in memory, and the I/O operations of disk can be reduced directly. The original randomly write strategy for data persistence is replaced by batched form, which greatly reduces the movements of disk arm and raises the write efficiency by several orders of magnitude.

However, despite the probability of cold data access operation is small, it leads to a large number of I/O operations on disk, which will significantly reduce the query performance of entire database. Simultaneously, since LSM tree achieves the update, delete operations in physical media by utilizing version control, a large number of additional storage resources are wasted for storing multiple versions of the similar data. Besides, there are a lot of designs in LSM tree that cannot match the characteristics of time series data well [1,2,8].

In view of the characteristics and known defects of time series data, we now propose a novel TTSM tree, a LSM tree variant in time series applied environment.

3.3.1 Basic Structure of TTSM Tree and Its Time Series Optimization

TTSM tree is similar to LSM tree which is generally comprised of two or more data storage structures. If differentiated by the type of storage media, the TTSM can be divided into two major components, one component is resident in memory and the other one is resident in disk (as shown in Figure 3).

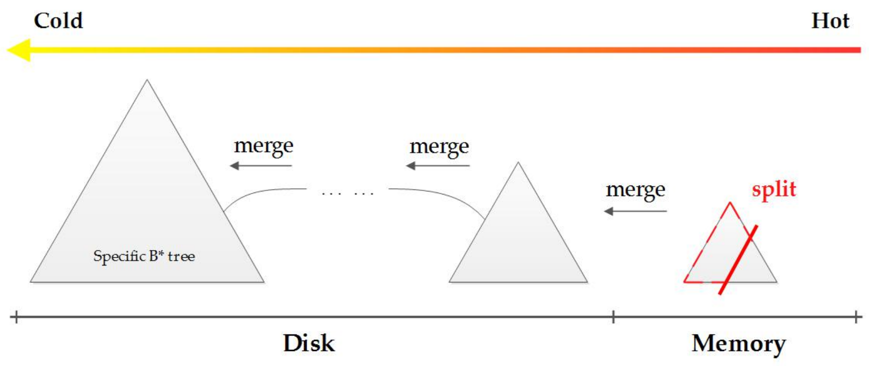
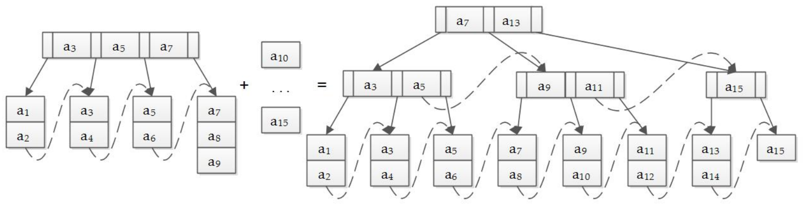


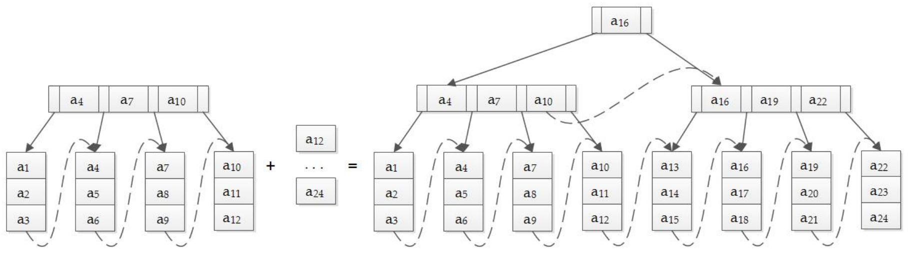
Figure 3: Data structures in FluteDB and its distribution.

In order to match the time series characteristics, TTSM tree has been targeted optimized as follow:

1. **Efficient tree structure**: For data storage and query, a suitable data structure can effectively improve the operational efficiency. The storage structures of different components in TTSM have been confirmed based on the numerical characteristic of input time series data, which facilitate the tree operations in TTSM tree, e.g. tree merge, tree segment. To be specific, due to the input time series data are ordered, monotone increasing and non-redundant, the chain structure of tree's leaf node and the splitting characteristics of non-leaf nodes are inherently suitable for the operation of time series data. Therefore, we use B* tree as the storage structures for both in-memory and disk data. The fundamental difference between B* tree and B+ tree is that the pointer between brother nodes, which are non-leaf node and non-root node, is added. This setting is able to match our data insertion strategy. Of course, the B* tree we selected here is able to support some special operations.



(a) Traditional B* tree



(b) Improved B* tree in FluteDB

Figure 4: A comparison example for B* tree with different data insert strategies, where (a) uses original method, (b) inserts in a batched append-only way (two B* trees' order in Figure 3 are four).

2. **Simplified insertion:** The B* tree is generally implemented in code by using the chain structure. Considering the characteristics of time series (e.g. ordered and monotone increasing), all newly inserted data only appears in the rightmost leaf node. If still using

original insertion strategy (maintaining the balance on both sides while split), the B* tree structure is not a full tree (as shown in Figure 4(a)), which brings about a lower structural efficiency and a waste of storage resources. Additionally, the non-leaf nodes in B* tree splits continuously, resulting in the lower overall efficiency of indexing. FluteDB changes the B* tree construction of time series to batched append-only mode, which means new data will be inserted to the end of chain batched instead of searching insert location, and then, the essential pointers are filled. The non-leaf node layer also can be updated directly in a similar way. The B* tree built in FluteDB is a full tree, and its computational complexity is also lower than traditional methods. As shown in Figure 4(b), the data insertion in memory can be greatly simplified in the case of known B+ tree order.

3. **Multi-component balance:** Because of the cold/hot data assumption, the time series data will be stored in memory and disk respectively. Furthermore, the TTSM tree also splits the original large B+ tree structure into a forest composed of several small B+ trees based on the difference of data "temperature" (defined in 3.3.2). As shown in Figure 3, the amount of data stored in each small B* tree in forest is dynamically adjusted according to the degree of data to achieve optimal performance of the entire system. In the forest based on this strategy, the colder the data is, the deeper the depth of its storage structure is, which conforms to the status of data usage. Specifically, we define the temperature of time series data more precisely and reduce the number of I/O disk access by storing warm data in a shallower B* tree. The depth of each B* tree will be determined by the specific needs, and its storage space will also be allocated in the first use.

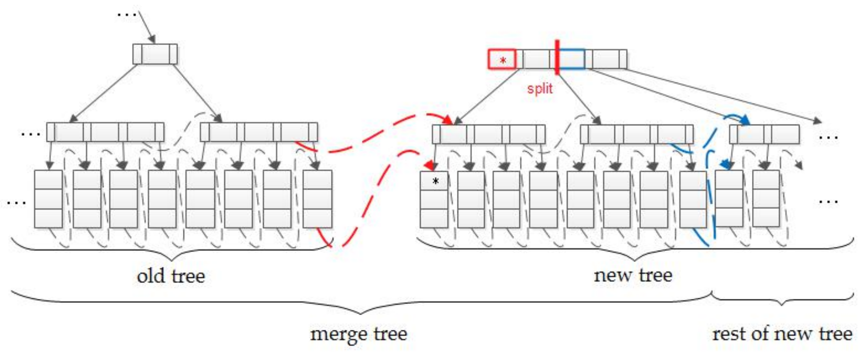


Figure 5: An example of a merge and split operation. The red portion is the added portion in the merge operation, and the blue portion is the removed portion in the split operation.

4. **Triggered split and merge:** As illustrated in Figure 3, the merging operation in traditional LSM tree is embodied in the merging of in-memory tree and in-disk tree. However, the design of multi-component in TTSM tree increases the tree objects needed to be merged. Since the time series data is inherently ordered and monotone increasing, the changes caused by data insertion always appear at the rightmost leaf node and its corresponding non-leaf nodes. In particular, the two trees to be merged also retain this characteristics, that is, the chain of leaf nodes in merged tree can be formed by ordered connecting the leaf node chains of input trees end to end directly. Thus, the merge operation does not require to reinsert data or rotate structure as in the original B* tree, and can be done more efficient at the premise of preserving existing tree structure. In the traditional LSM tree, the merging operation of two trees is periodic, or the data accommodated by tree structure is more than its performance load limit. Such a simple merging strategy cannot achieve satisfactory merging efficiency in

time series applied environment, so we propose a special trigger strategy and an enhanced merging strategy to improve the merging effect and efficiency in TTSM tree. We bind the trigger opportunity, new tree's data storage amount and its query latency together, which means that the merge and split operation will be triggered if the depth in new tree exceed its acceptable query latency (the query latency is directly proportional to the depth of B* tree) and the rightmost old tree node which is as same depth as the root node in new tree is not full. Additionally, merging two trees by one time will cause a lot of data to be locked, and the entire in-memory indexing will also be emptied, which results in a large number of disk access operations in a short period of time. As shown in Figure 5, the fundamental idea of the merge strategy in TTSM tree is to embed (split and reconstruct) the appropriate largest leftmost sub-structure of new tree to old tree, and remains the structure of the old tree and the rest of new tree unchanged, which ensures that only partial colder data in the new tree is locked, thereby increases the availability of data in FluteDB. Once the merge operation occurs on the disk, FluteDB appends the new data in the reserved position to ensure its physical continuous.

3.3.2 Cost Analysis

In order to evaluate and compare the performance of TTSM tree and analyze the performance of different methods in dealing with cold/hot data, we theoretically analyze the cost of B + tree, LSM tree and TTSM tree.

For the sake of convenience, we predefine some hard disk and memory related symbols. $cost_n$ and $cost_l$ respectively denote the space cost for non-leaf and leaf node storage ($cost_n < cost_l$). $cost_r$ and $cost_m$ respectively denote the I/O cost required to read or write a page for random access and continuously access to the disk ($cost_r >> cost_m$). $cost_D$ and $cost_M$ are the storage cost of 1M bytes in disk and memory ($cost_D < cost_M$). We also define the data temperature as the number of access times of 1M bytes data per unit time.

As shown in Table 2, our cost analysis is mainly from two aspects: space cost and time cost. Among them, we use a n -order tree as an example in space cost analysis: the LSM tree's version control number is m , and the total number of time series data is N . The B+ tree and LSM tree are split automatically at the time of insertion and the tree structure is not full but balance, so we define the depth is d . The TTSM's tree structure is mainly full and its depth is $D = \left\lceil \log_n \frac{N}{n} \right\rceil + 1 \ (D \le d-1)$. Obviously, The space cost of LSM tree is the highest because of its version control and strict sequential write operations. When the data inserted into B+ tree sequentially, the leaf nodes and non-leaf nodes will not be full due to its balance strategy. Then, B+ tree's space cost is larger than TTSM tree's (because the description is more complex, the TTSM tree mentioned here is not multi-component version. In fact, the multi-component version takes up less space than current version).

To compare the I/O costs, we assume there are total n data waiting for insertion, and the original disk total contains m pages which include m_1 pages non-leaf nodes (still m pages in disk after inserting). Since the insert operation in B+ tree needs to repeat some same sub-operations (e.g. search insert location), its I/O cost is the highest (assuming that only one I/O disk operation in each insert location search, and ignoring the node split and other operational consumption). The I/O cost of LSM tree is significantly cheaper than B+ tree because LSM tree batched inserts mass data and updates all pages at a new physical address continuously. TTSM tree reserves the location for new data at each layer, then it just links the new leaf node chain and appends the upper non-leaf nodes to the appropriate location when new data needs to be inserted.

Table 2: Comparison of the three tree structures' costs.

	Space Cost	**I/O Cost**
B+ Tree	$N*[cost_l+(\sum_{i=2}^{d}\frac{2}{n^{i-1}})*cost_n]$	$2n*cost_r$
LSM Tree	$m*N*[cost_l+(\sum_{i=2}^{d}\frac{2}{n^{i-1}})*cost_n]$	$(2m-2)*cost_m+2*cost_r$
TTSM Tree	$N*[cost_l+(\sum_{i=2}^{D}\frac{1}{n^{i-1}})*cost_n]$	$(2m-2m_l-1)*cost_m+2*cost_r$

On the basis of the definition of data temperature, we analyze the theoretical consumption of three algorithms to further compare the performance among them. We assume that there are C M bytes data need to be stored per second, which will be access R different times. Then, the temperature of data can be described as R/C . As shown in Figure 6, the B+ tree is less effective in handing cold/hot data, regardless of whether the disk utilization or memory utilization is the lowest. The resource utilization of LSM tree and TTSM tree are both higher, but the overall performance of TTSM tree is better.

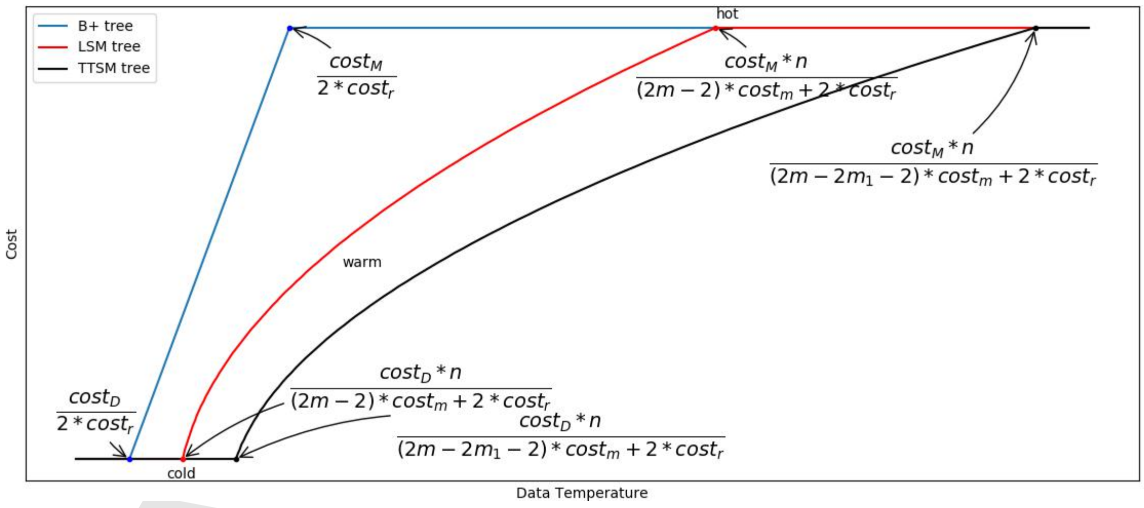


Figure 6: The consumption of three indexing structures in different data temperature.

Specifically, B+ tree stores data on disk directly and reloads the data into memory when the disk data is accessed. By definition, the storage cost of B+ tree on the disk is $C*cost_D$, and the storage cost of the data in memory is $C*cost_M$ when all the data is loaded into memory. Besides, the access operations in B+ tree are random disk I/O operation, so the cost of single access operation is $2*cost_r$ (as known from Table 2). With the increase of stored data and the accumulation of external queries, the value of R becomes larger, which leads to the increase of the cost of overall data access. Furthermore, the cost of data access in B+ tree takes two big turnings due to the different data temperatures (as shown in Figure 6). The first turning occurs when the cost of random access is equal to the cost of the disk storage cost ($R/C = cost_D / 2 * cost_r$), which means that the data at this temperature can be stored in memory partially. The another turning occurs when the cost of random access is equal to the cost of the memory storage cost ($R/C = cost_M / 2 * cost_r$), which means that the data at this temperature should be all stored in memory.

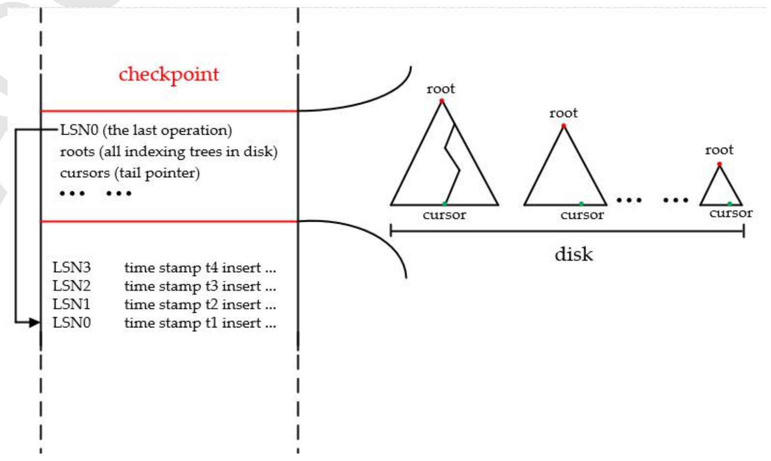
Table 3: The position of the turning points of the three tree structures.

	Cold to Warm	Warm to Hot
B+ Tree	$R/C = cost_D / 2 * cost_r$	$R/C = cost_M / 2 * cost_r$
LSM Tree	$R/C = \frac{cost_D * n}{(2m-2) * cost_m + 2 * cost_r}$	$R/C = \frac{cost_M * n}{(2m-2) * cost_m + 2 * cost_r}$
TTSM Tree	$R/C = \frac{cost_D * n}{(2m-2m_l-1) * cost_m + 2 * cost_r}$	$R/C = \frac{cost_M * n}{(2m-2m_l-1) * cost_m + 2 * cost_r}$

Similarly, we can get the turning points of LSM tree and TTSM tree in the same conditions, which as shown in Table 3. From Table 2, the I/O cost of B+ tree is the largest, the LSM tree's cost followed, and the TTSM tree's is the smallest. Thus, by observing Figure 6, the positions of turning points of LSM tree is later than B+ tree's, and the TTSM tree's is the latest, which means that both the disk utilization and memory utilization of TTSM tree are the highest among these tree structures.

3.3.3 Fault Tolerant

In database, particularly in cloud environment, indexing is much more important than other components. The fault tolerant strategy for indexing in TTSM tree needs to be adjusted because it is involved in memory and disk at the same time and even related in load correlation and lock state [10]. To be specific, during normal operation, the new data is inserted into B* tree in batches, but the handled data is still cached in memory temporarily. Only when the data in caches are written to disk are the above operations considered to be finished. However, if system broke down, partial data in memory (e.g. split or merge two B* trees) will be lost. In order to avoid this phenomenon, we record a special Write Ahead Log (WAL) for TTSM tree. Because of the particularity of time series data, TTSM tree's WAL mainly constitutes of insert operations. These WALs are periodically written to disk and form a checkpoint, which can be used to recover the TTSM tree at that time.



LSN3 time stamp t4 insert ...
LSN2 time stamp t3 insert ...
LSN1 time stamp t2 insert ...
LSN0 time stamp t1 insert ...

Figure 7: The checkpoint composition in WAL.

To form a checkpoint, the following conditions or operations have to be required [21]. Firstly, all current split and merge operations have been completed, and the newly insert operation, split or merge operation are postponed until the checkpoint has been set successfully. Secondly, the B* tree in memory need to be written in a known location on the disk (the insert operation can start again, but the split or merge operation still needs to wait), and all the cached changes from previous operations also need to be written in disk. Then, a setup checkpoint can be added into WAL. As shown in Figure 7, checkpoint in WAL contains the Log Serial Number (LSN), the physical addresses of the roots of all the associated indexing in disk and the merge cursors. The normal split or merge operation can restart when the checkpoint has been set.

When the system is experiencing a failure at runtime, it can be recovered as follows based on the checkpoint [21]. The recovery operation locates an appropriate checkpoint in WAL and reloads the previous data into memory firstly. Then, it loads all of the data behind checkpoint in WAL into memory in turn and inserts that data in batches. Besides, by reading the split or merge cursors of indexing component stored in checkpoint, the corresponding split or merge operations can be restarted. After all operations mentioned above have been done, the recovery task of TTSM tree is officially over. Since all of the storage space of TTSM tree has been allocated as a full tree, the physical addresses of leaf nodes are fixed regardless of whether they are merged or inserted, which means the partial original pointers still can be used. The strategy of the TTSM tree is more concise than the original LSM tree fault tolerant strategy, and the recovery is more rapid (at the expense of an acceptable partial space cost).

3.4 Time Series Data Query

Data query is one of the main service forms provided by database, and query latency is one of the main indicators to measure the performance of database and cloud service [8,10]. In the time series applied environment, due to the query operations only hold a small portion of all operations, FluteDB greatly improves its write rate at the price of sacrificing partial query efficiency. In fact, some targeted optimizations have been applied in FluteDB based on the characteristics of time series queries, such as timeliness, continuity and cyclical. Specifically, the timeliness means that the probability of the time series data being queried is inversely proportional to the time duration from data insertion to current. Continuity is similar to the continuous access hypothesis in computer system, that is, the probability of data around the accessed data is higher. Meanwhile, cyclical means that part of the cold data in disk will be periodically queried, mainly from a large number of upstream applications periodic query needs. Obviously, time series data query is different from ordinary query. By utilizing time series characteristics, FluteDB is reasonable to optimize the data scheduling and cache strategy to improve the hit rate in memory, reduce the disk I/O operations and improve the overall query performance. Additionally, several specific additional acceleration components can also avoid some unnecessary high cost operations.

3.4.1 Extra Cache

Data caching is one of the important measures to enhance the performance of the database query [18]. Particularly in the application environment of time series, a suited cache mechanism can optimize the efficiency of data retrieval and write operation by reducing the number of times of operating disk. To be specific, FluteDB designs a special cache manage strategy based on the three characteristics in time series query. Firstly, in order to reduce the time consuming disk I/O operations, FluteDB dynamically adjusts the stored data in memory based on the changes in data temperature at the appropriate time (such as when the memory tree merges with the disk tree). Meanwhile, FluteDB continuously loads the data blocks around target data block into cache to prepare for possible queries. In addition, FluteDB is also equipped with an improved cache replacement strategy (as shown in Algorithm 1, which can retain the higher temperature data effectively and maintains the cache running stably. This means that we no longer load a single block into cache, but rather a number of consecutive data blocks in a single query. And FluteDB replaces the corresponding number of blocks (not necessary continuous) each time the data is not hit in cache.

3.4.2 Accelerated Components

In addition to the extra cache, FluteDB still has several accelerating components that can reduce the number of high consumption operations.

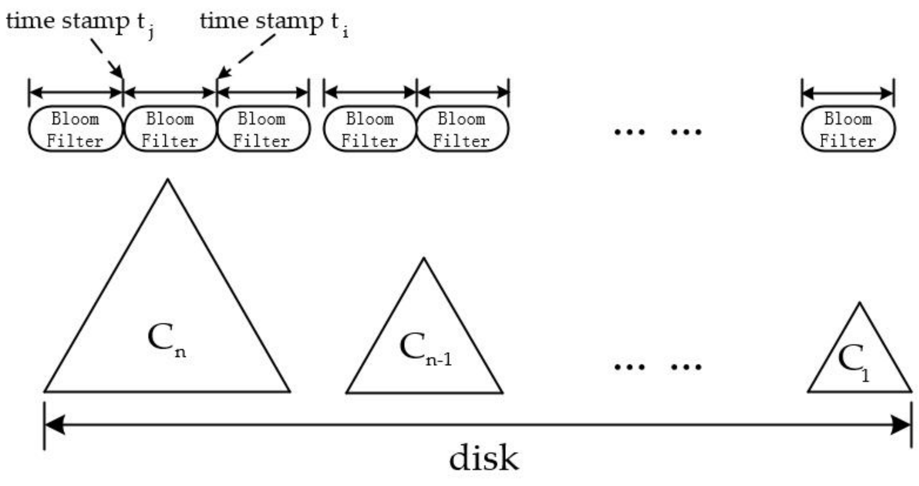


Figure 8: The checkpoint composition in WAL.

Bloom Filter: Traditional LSM tree have to query on the disk when the query operations is not hit in memory [6]. Because the disk I/O operation is too costly, the TTSM tree still causes a lot of costs though the TTSM tree splits the indexing in disk. If TTSM tree can quickly determine whether the data exists, and further confirm the detailed location of data, the entire query performance in disk will be greatly enhanced. To avoid a large number of reset, FluteDB set the size of Bloom Filter based on the smallest split unit [22]. As shown in Figure 8, FluteDB constructs the Bloom Filters for each subtree in TTSM tree. The advantage of this component is that it is possible to immediately determine whether the data is present with little I/O operation and calculation consumption.

Continuous Replacement: In the ordinary cache strategy, the new loading data block will replace the old one whenever a cache miss encountered [18]. However, in time series environment, the data stored in different media has discrepant caching value because of its source and composition. The main reason for this is that cyclical operations from requirements such as certain monitoring will cause the data to show periodic requirements. Hence, the replace operation in FluteDB has been given an execution probability by considering above reasons.

Because our improved replacement strategy can load continuous hot data into cache at once, the hit rate, efficiency of read and write operations are significantly enhanced, despite the overall number of read and write operations do not change. In order to verify our design, we modify a number of existing replacement algorithms by using the same improvement. The experimental results (in section 5) show that our method can significantly improve the hit rate and the overall efficiency, and the time-series enhanced Clock outperforms best among the several algorithms. Meanwhile, to guarantee the basic dependability of cache, we partially follow the original fault tolerance mechanism, which mainly includes the mentioned log mechanism and persistent operations. In fact, we found that time-series data has more advantages than normal data on storage dependability when we investigated it. Because most sensors have their own cache for responding the IDC's requests.

3.5 Time Series Data Processing

Data processing is the component which directly connects the storage engine and physical storage media [16,17]. An efficient and stable data processing module can greatly reduce the consumption of disk I/O operation and storage space. Additionally, it can also make the complexity of some data operations, such as data migration, backup and indexing, to be greatly reduced.

3.5.1 Time Series Compression

The main purpose of data compression is to reduce the resource cost, which is especially obvious in the cloud environment [1,2,8]. Compared with the traditional data compression task, time series data, on the one hand, has some special additional constraints for time series compression owing to its special characteristics, but on the other, it is properly relaxed in other common constraints as well. To obtain the best compression efficiency, FluteDB respectively adopts the suited compression strategies to the corresponding object based on its numeric characteristics.

Time-Series Compression: The time stamp $t_1, t_2 ..., t_n$ is a monotonically increasing sequence. And the delta between t_{i-1}, t_i is roughly fixed since the intervals of most sampling sources are fixed (Though the interval of data sampling is not specified, it will be limited in a fixed range according its own streaming and large-scale characteristics.). Hence, rather than storing the whole time stamp, only storing a delta of deltas is able to save a lot of resources and does not significantly increase the compression and decompression time. Specifically, we encode the time stamp into blocks by utilizing an improved variable length encoding algorithm named Sliced Delta of Deltas (SDD).

Table 4: The position of the turning points of the three tree structures.

Time Stamp(μs)	Delta of Delta(μs)	SDD Code
1452202444 009 003	-	-
1452202444 010 028	25	0011 1001
1452202444 011 031	3	1011

As described in Table 4, we pre-establish the delta between two continuous sampling points as 1000 μs , and the delta of deltas is much smaller than original delta obviously. In order to further minimize the compression rate, we store the true form of delta of deltas in fixed-size slices (The size of slice is determined by the average size of delta of the deltas.), which is able to reduce the use of the flag bits (if the slice is the last slice of a value, its first bit is set to bit '1', otherwise the first bit is set to bit '0'). As shown in Table 4, we select four bits as the length of single slice. So we can directly encode the binary code of 25 as 011001 (the first bit is sign bit), and further divide it into two slices, 0011 and 1001.

The theoretical compression ratio as shown in Equation 1, where $length(\cdot)$ denotes the number of bits of the corresponding numeric, $f(\cdot)$ is the function of SDD, $size\_of(\cdot)$ is the length of t_i (e.g. int, float and so on).

$$compressio \quad n\_ratio_{SDD} = \frac{size\_of(file_{compressed})}{size\_of(file)} = \frac{length(t_0) + length(\delta t) + \sum_{i=1}^{n} f(\delta\delta t_i, t_i, t_{i-1}, l_b)}{\sum_{i=0}^{n} size\_of(t_i)}$$

Observation values compression: In addition to the time stamp compression, we also compress the observation values. Apparently, as real data collected by sensors or other devices, observation values are still continuous like time stamp [2,8]. However, the delta between two continuous sampling points is not stable, which means the variance of delta become much larger. Therefore, we present a novel algorithm AXOR by considering the data characteristics and calculation complexity to achieve efficient compression. Likewise, we still use variable length encoding to compress the observation values. Due to the type of observation values are mostly double, we felicitously and efficiently compress the XOR values between continuous observation values.

As a real instance in Table 5, we will introduce AXOR step by step. In the compression process, we encode the input value into double representation, approximately set the trailing bits of its double representation to 0 (the number of variation bits is shown in Equation 5 and ensure its approximate error to be limited at $1 \times 10^{-(n+2)}$,

$$n_{omitted\_bits} = l_{decimals} - \lfloor \log_2 observation\_values \rfloor - \left| \log_2 10^{n+2} \right|$$

where n denotes the precision of the initial input value, $l_{decimals}$ is the length of decimal part in double representation (default value is 52). Then, we calculate a simple XOR between the current and previous values, and break the XOR's value into three parts, which include the meaningful code's start position, length and specific content (meaningful part denotes the code between first '1' and last '1' in XOR binary value). Of course, there are three control bits to explain the following codes. The first bit indicates whether the XOR'd value is 0, the second bit denotes whether the meaningful bits start at the same position as previous bits, and the final bit means whether the length of the meaningful bits is same as the previous ones.

Table 5: The position of the turning points of the three tree structures.

Original Data	Approximate Double Representation	XOR with Previous	Control Bits	Start Point	Length	Meaningful Code
40.687612	0x40445803AB800000	-	-	-	-	-
40.673564	0x4044563758400000	0x00000E34F3C00000	111	10101	010110	1110001101001111001111
40.723569	0x40445C9DE8A00000	0x00000AAAB0E00000	101	-	010111	10101010101010110000111
40.583876	0x40444ABC72E00000	0x000016219A400000	100	-	-	10110001000011001101001
0.000000	-	-	000	-	-	-
40.892842	0x40447248A5800000	0x000038F4D7600000	101	-	011001	1110001111010011010111011

In addition, in order to deal with other types of data, we selected some more diverse compress algorithms. Among them, the Boolean observation value can be expressed with one bit which do not need to be compressed, the Float observation value can also be compressed by utilizing AXOR algorithm, and the String observation value can be compressed by using LZ4 or snappy algorithm [15].

Both of SDD(lossless) and AXOR(approximate lossless) are compressed in an unit of database blocks, and the compressed information is stored in the block header of the data file. In addition, in order to ensure the dependability, consistency and completeness of the compressed data, we add the check code behind the tail of each block. Independent experimental results in Table 6 and 7 show that SDD and AXOR can efficiently compress the time-series data. Besides, our compressed result can be easily converted into other compressed formats (e.g. PAA, APCA) for downstream tasks.

Table 6: Comparison of different compression methods for compressing 100,000 time stamps.

Compression Method	Compression Ratio	Compression Time	Decompression Time
RLE [17]	95.63%	0.6635ms	0.9754ms
LZ77 [15]	61.46%	1.7367ms	2.9964ms
Delta	35.06%	0.4391ms	1.1078ms
Original DD [2]	18.74%	0.5433ms	1.2127ms
SDD	11.01%	0.6468ms	1.2931ms

Table 7: Comparison of different compression methods for compressing 100,000 observation values.

Compression Method	Compression Ratio	Compression Time	Decompression Time
RLE [17]	98.84%	0.7362ms	0.9535ms
LZ77 [15]	68.43%	1.8453ms	3.4363ms
Delta	77.54%	0.5884ms	1.0435ms
XOR [8]	48.48%	0.8582ms	1.5324ms
AXOR	43.98%	0.7964ms	1.3624ms

3.5.2 Data Encapsulation

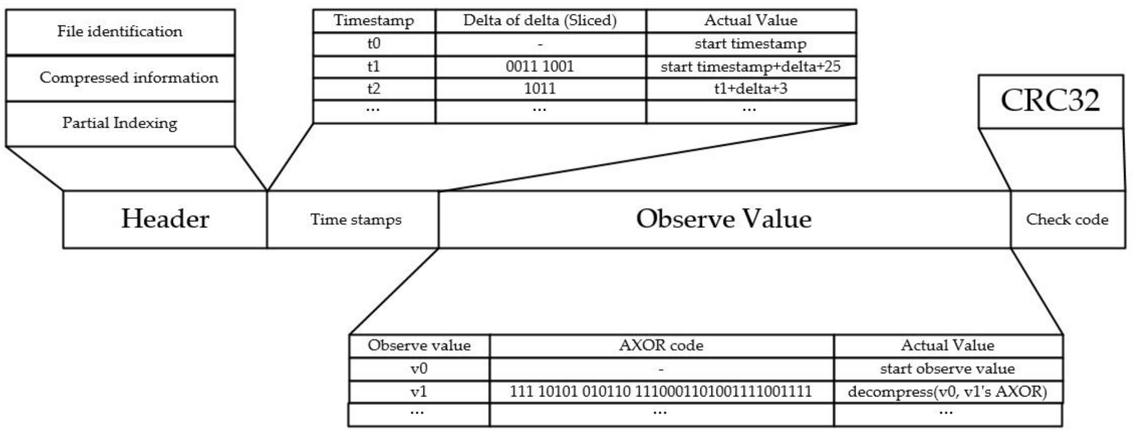


Figure 9: The composition of persistent data file.

The data file stored on the disk is the final form of persistent data, and how to save that data in an optimal way is also the content has to be considered [13,14]. As illustrated in Figure 9, FluteDB stores the compressed data in data file by column, which makes it easy to retrieve and query data quickly. In particular, to ensure data consistency and security, we design a check code strategy for the data file. Because the time series data is compressed by variable length methods, the final data file is also variable length. Besides, the file header of each individual data file contains partial index information, which is utilized to manage the data easily.

4. Evaluation

In this section, we analyze the performance of FluteDB via benchmarks and present measurements of our production deployment.

4.1 Experimental Setup

We run our services on a machine with Intel Xeon CPU E5-2650 8-core processors, 256 G of 1600MHz DDR3 RAM, and a software RAID 0 array of one SATA Western Digital WD2000FYYZ 2TB, 7,200 RPM hard drive with 64 MB of cache. The machine run Linux kernel version 3.2.0 using the default file system readahead of 128 KB unless otherwise noted.

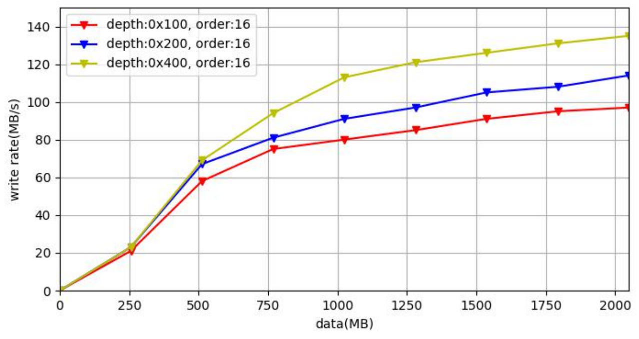
Before we start our data services, we clear Linux's disk cache and the drive's internal cache, and write and read 64MB data to a random location on disk. All services are single-threaded. To guarantee the reliability of experimental results, we run each test 10 times, and we plot the average of the metric with a 95% confidence interval, computed using the *t* test.

4.2 Pressure Test

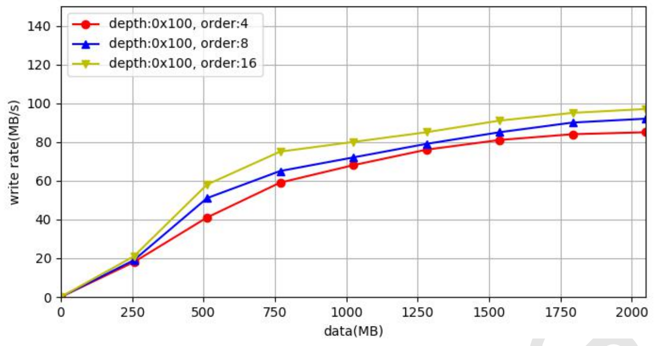
We evaluate the performance of FluteDB with a stand-alone system by inserting 20 GB data into a table. And then we evaluate the read throughput of FluteDB through executing 100, 000 query operations.

4.2.1 Write Rate

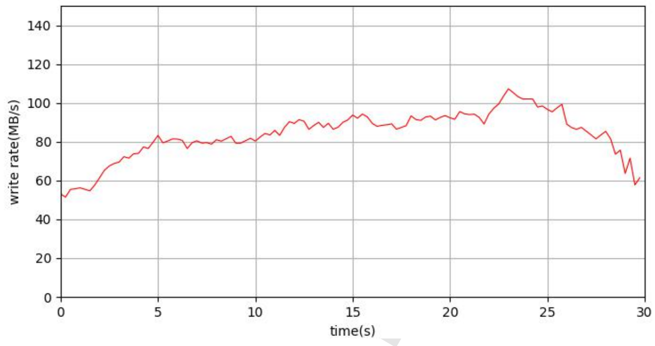
In our experiments, we use fixed variables, e.g. row size, batch size, to evaluate the throughput of FluteDB by experimenting with a combination of TTSM tree order and split depth. The results are shown in the Figure 10 (a) and (b), as the increasing of threshold of TTSM tree, the write rate increases at first and eventually flattens. In particular, due to the use of the TTSM tree, merge events are shown as impulses with the increasing of data size as illustrated in Figure 10 (c).



(a) write rate vs. split depth



(b) write rate vs. TTSM tree order



(c) the variation of write rate with time

Figure 10: The experimental results of write rate.

4.2.2 Query Efficiency

Because the query operations of time series data only include time point query and range query, its efficiency is independent of numbers of readers and other parameters of table. In contrast, the query efficiency of FluteDB is sensitive to the data temperature due to the difference in data storage location and the cost of disk arm movement. As illustrated in Figure 11, the fluctuations on query efficient curve mainly come from the access of the hard disk.

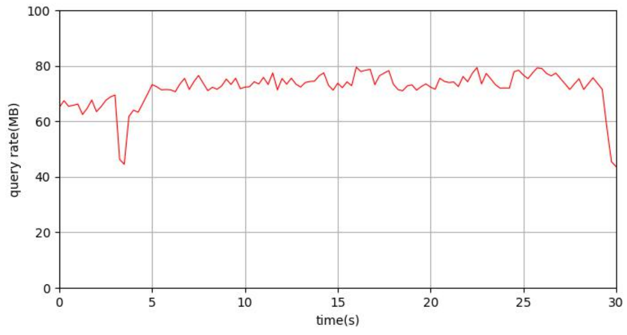


Figure 11: The experimental results of query efficiency.

4.3 Horizontal Comparison

Since most of the key sub-modules in FluteDB are targeted to enhanced based on the time series characteristics, its performance for data processing is also greatly improved. In order to evaluate and further compare the performances of different sub-modules, we select some general state-of-the-art algorithms as contrast to demonstrate the advancement and efficiency of FluteDB.

4.3.1 Insert and Query Efficiency

As an important indicator of database, we analyze the theoretical consumption of different data indexing algorithms in detail in section 3.3. To verify our analysis and obtain the efficiency indicator of TTSM tree in real environment, we generate the indexing for inserting data based on B+ tree, LSM tree and TTSM tree. We only perform the insert operations (ignore the query operations) to facilitate the control of the variate. The experimental dataset in this paper is the upload data stream of taxi sensors in New York [7]. This dataset contains 2.1 million valid data insertion operations, which includes 12 items, such as timestamps (primary keys), direction, speed and etc.

Experimental results as illustrated in Figure 12, the insert time consumption of TTSM tree keeps a linear relation with the amount of data in indexing haply, however, the B+ tree and LSM tree are not that (step or curve). The reason for that phenomenon is that the TTSM tree still retain itself as a stable append-only state, and the complexity of insertion is $\Theta(1)$. Certainly, the resulting fluctuations on curve of TTSM tree mainly come from the cyclical tree split and merge operations. This means that the insert complexity of them will be increased with an increase in the amount of data. Experimental results show that TTSM tree significantly reduce the time required for insertion due to the characteristics of time series, which is better than the B+ tree and LSM tree.

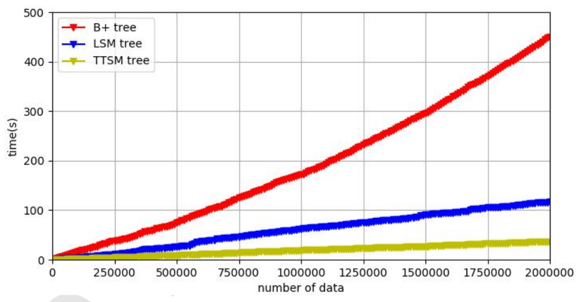


Figure 12: The comparison of insert consumption.

4.3.2 Data Size

In addition, we also compare the memory and disk resources occupied by three indexing structures. The experimental results are shown in Figure 13, we can obviously find that the disk consumption of TTSM tree is step-by-step randomly because it applies the continuous storage space for full tree when creating a new subtree structure for response trigger of split. The B+ tree is completely stored in disk, so its disk consumption is linearly related to the amount of input data. The LSM tree imports its data in memory into disk in batches, so its disk consumption also grows in a ladder fashion. However, since its additional version control and append-only strategy, the disk consumption of LSM are much larger than others.

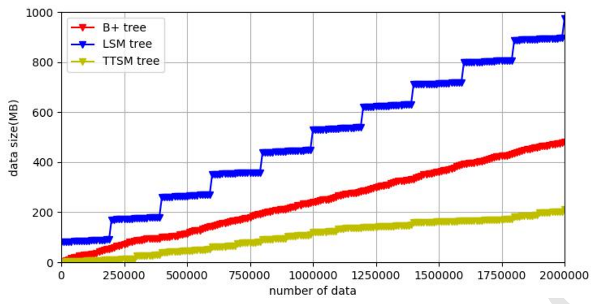


Figure 13: The comparison of storage consumption.

On the whole, the indexing based on the TTSM tree has a better performance both in time and space consumption when write data.

5. Conclusion

FluteDB is a novel memory TSDB for sensor-cloud which efficiently manages time series data by rationally processing memory data and interacts data in disk in batch. To fully adapt to the real time series data application environment, FluteDB has optimized its corresponding indexing structure, query components, data compression and data encapsulation by considering the characteristics of time series. Furthermore, FluteDB is equipped with a complete fault tolerant and recovery strategy in order to provide a more reliable and available data service. In particular, FluteDB can adjust its data storage medium and data structure, adaptive data query probability and balance the resource consumption and service performance according to the data temperature characteristic defined by itself. By analyzing the working status of FluteDB in a real cloud environment, we can find that the ultimate throughput of FluteDB can meet the vast majority of data needs. On this basis, FluteDB obtains the competitive effect on the average writing rate, the average query latency and so on through the comparison with the existing advanced methods. Meanwhile, FluteDB is equipped with a highly efficient compressor packaging strategy, so its use of disk and memory resources is significantly reduced. In addition, FluteDB can quickly recover from errors and failures in real and simulated errors and re-provide services.

In order to further enhance the performance of FluteDB, we will be suitable for the future of a reasonable cluster strategy to fully use the cluster concurrency. At the same time, according to some of the more unique features of timing data, some scalable policies and plug-ins can also make data service performance in this upgrade.

Reference

[1] TimescaleDB: SQL made scalable for time-series data. http://www.timescale.com/papers/timescaledb.pdf, 2017 (accessed 17 December 2017).
[2] Storage Engine of InfluxData. https://docs.influxdata.com/influxdb/v1.2/concepts/storage_engine/, 2017 (accessed 17 December 2017).
[3] The world's most popular open source database. https://www.mysql.com/, 2017 (accessed 17 December 2017).
[4] The world's most advanced open source database. https://www.postgresql.org/, 2017 (accessed 17 December 2017).
[5] OpenTSDB - A Distributed, Scalable Monitoring System. http://opentsdb.net/, 2017 (accessed 17 December 2017).
[6] O. Patrick E. , C. Edward, G. Dieter, O. Elizabeth J., The Log-Structured Merge-Tree (LSM-Tree). Acta Inf.

33(4) (1996) 351-385.
[7]New York City Taxi Trip Duration. https://www.kaggle.com/c/nyc-taxi-trip-duration/data, 2017 (accessed 17 December 2017).
[8] P. Tuomas, F. Scott, P. Cavallaro, Q. Huang, M. Justin, T. Justin, V. Kaushik, Gorilla: a fast, scalable, in-memory time series database, publication of the Very Large Database Endowment (PVLDB), 8(12) (2015) 1816-1827.
[9] R. Mendel, O. John K., The Design and Implementation of a Log-Structured File System, ACM Trans. Comput. Syst. 10(1) (1992) 26-52.
[10] R. Sean, W. Eric, W. Edmund, A. Ethan, S. Nat, LittleTable A Time-Series Database and Its Uses, In Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD), 2017, pp.125 – 138.
[11] S. Russell, R. Raghu, bLSM: a general purpose log structured merge tree, In Proceedings of the 2012 ACM International Conference on Management of Data (SIGMOD), 2012, pp.217 – 228.
[12] C. Yongjie, T. Hanghang, F. Wei, J. Ping, H. Qing, Facets: Fast Comprehensive Mining of Coevolving High-order Time Series. In Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2015, pp.79 – 88.
[13] P. Stavros, D. Kushal, M. Samuel, M. Timothy G., The TileDB Array Data Storage Manager, publication of the Very Large Database Endowment (PVLDB), 10(4) (2016) 349-360.
[14] J. Christopher M., O. Edward, Y. Wai Gen, The partitioned exponential file for database storage management, The VLDB Journal, 16(4) (2007) 417-437.
[15] C. Kaushik, K. Eamonn J., M. Sharad, P. Michael J., Locally adaptive dimensionality reduction for indexing large time series databases, ACM Trans. Database Syst. 27(2) (2002) 188-228.
[16] Z. Jacob, L. Abraham, A universal algorithm for sequential data compression, IEEE Trans. Information Theory 23(3) (1977) 337-343.
[17] B. Mostafa A., Data Compression in Scientific and Statistical Databases, IEEE Trans. Software Eng. 11(10) (1985) 1047-1058.
[18] P. Stefan, B. László, A survey of Web cache replacement strategies, ACM Comput. Surv. 35(4) (2003) 374-398.
[19] G. Robson E. D., B. Azzedine, A. Raed, Time Series-Oriented Load Prediction Model and Migration Policies for Distributed Simulation Systems, IEEE Trans. Parallel Distrib. Syst. 28(1) (2017) 215-229.
[20] G. Mario, E. Hugo J., O. Fernando, T. Eric S., Time series forecasting with genetic programming, Natural Computing 16(1) (2017) 165-174.
[21] L. Gregory, X. Liudong, B. Hanoch, D. Yuanshun, Reliability of Series-Parallel Systems With Random Failure Propagation Time, IEEE Trans. Reliability 62(3) (2013) 637-647.
[22] M. Michael, Compressed bloom filters, IEEE/ACM Trans. Netw. 10(5) (2002) 604-612.
[23] O. Lehtikangas, T. Tanja, A. D. Kim, A. Simon R., Finite element approximation of the radiative transport equation in a medium with piece-wise constant refractive index, J. Comput. Physics 282 (2015) 345-359.
[24] L. Chen, L. jianxin, S. Jinghui, Z. Yangyang, FluteDB: An Efficient and Dependable Time-Series Database Storage Engine, SpaCCS Workshops, 2017, pp.446-456.

*Author Biography & Photograph

Chen Li is currently a Ph.D. student at the School of Computer Science and Engineering, Beihang University. His research interests include knowledge graph, Information retrieval and data analysis and processing.

Bo Li is an assistant professor at the School of Computer Science and Engineering, Beihang University. He received the Ph.D. degree in Jan. 2012. He was a visiting scholar in computer science department of University of Edinburgh in 2014. His current research interests include virtualization, system reliability and data mining etc.

Md Zakirul Alam Bhuiyan is currently an assistant professor in the Department of Computer and Information Sciences at Fordham University. Previously, he worked as an assistant professor at Temple University. His research focuses on dependable cyber physical systems, WSN applications, big data, and cyber security.

Lihong Wang is a professor in National Computer Network Emergency Response Technical Team/Coordination Center of China. Her current research interests include information security, cloud computing, big data mining and analysis, Information retrieval and data mining.

Jinghui Si is currently an undergraduate student at the School of Computer Science and Engineering, Beihang University. His research interests include data mining and natural language processing.

Guanyu Wei is currently an undergraduate student at the School of Computer Science and Engineering, Beihang University. His research interests include system reliability and distributed systems.

Jianxin Li is a professor at the School of Computer Science and Engineering, Beihang University, and a member of IEEE and ACM. He received the Ph.D. degree in Jan. 2008. He was a visiting scholar at machine learning department of CMU in 2015, and a visiting researchers of MSRA in 2011. His current research interests include virtualization and cloud computing, data analysis and processing.