Cherry: A Data Series Index with Efficient Writing and Low-Latency Query Answering

Han Hu, Jiye Qiu, Hongzhi Wang, Bin Liang
Harbin Institute of Technology
{huhan@stu.,qiujiye@stu.,wangzh@,liangbin@stu.}hit.edu.cn

ABSTRACT

Applications constantly generate and collect large data series, and analysts require real-time data series similarity search to meet the demands of applications. However, even the state-of-the-art works are unable to meet the performance requirements of many scenarios. We propose Cherry, an index with extremely fast writing performance and superior query answering capability by utilizing four techniques. Firstly, we design a novel index structure, ZSBtree, that draws inspiration from the combined features of Z-order curve and Symbolic Aggregate approXimation (SAX), to efficiently merge the clustering ability of SAX with the performance of B^+-tree. With the aid of an elegant bulk-loading algorithm in construction, the ZSBtree exhibits remarkable pruning capability and performance. Secondly, we present a pipelined writing system based on the idea of the Log-Structured Merge-tree (LSM-tree), achieving efficient writing and minimal space usage. Thirdly, during the query answering, we propose several optimized algorithms to reduce the amount of accesses and calculations to data summaries and raw data series based on the properties of ZSBtree and SAX, thereby greatly reducing the processing time. Finally, we leverage the parallel capability of modern hardware to accelerate the expensive vector operations in our algorithms. Compared with the current fastest multi-core architecture work on a 1 billion dataset, Cherry achieves an index construction speed that is 0.42x faster and an exact query answering speed that is 10.8x faster with two orders of magnitude less memory and half the index size.

PVLDB Reference Format:
Han Hu, Jiye Qiu, Hongzhi Wang, Bin Liang. Cherry: A Data Series Index with Efficient Writing and Low-Latency Query Answering. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

A data series is a collection of data points ordered according to a certain dimension. In the case where the dimension is time, we call it a time series. Modern applications generate numerous data series in various domains, including engineering, finance, multimedia, astronomy, medicine and so on [25, 27, 41, 52–54].

Real-time analysis of massive data series collections is crucial [18, 19, 52] in many applications. As its integral process, similarity search is also essential in many data analysis techniques, such as clustering, classification, outlier detection and so on [9, 44, 57].

Challenges and Limitations. The similarity search on data series brings two major challenges. For one thing, data series consist of numerous data points, making direct operations on them computationally and storage-wise expensive. For another thing, increasing data series generation speed and decreasing data storage costs lead to larger data series collections. Consequently, scanning the entire dataset for each query is impractical for real-time requirements, and we need effective index to achieve high performance.

Facing these challenges, a superior data series index must exhibit the following features [18, 19, 30]: rapid index construction speed to accommodate the constant inflow of voluminous data, minimal consumption of computing and storage resources to enhance availability, faster speed and higher precision for query answering. Besides, stream processing such as updating and real-time query answering are often necessary instead of batch processing in many scenarios, needing the capability similar to those of time-series databases [1, 6, 42, 58]. Some indices have been proposed. However, they still fail to address the challenges.

The state-of-the-art works [3, 17, 23, 30, 37, 45, 47–50, 55, 59, 61, 63, 65] construct indices using compressed data summaries [14, 21, 29, 35, 36, 38, 50, 56, 60] with a distance approximating that of the raw data series to improve writing performance and screen a subset of the data series to enhance query answering performance.

ISAX-based approaches [11, 12, 46, 64] employ a binary tree structure with a high depth, necessitating a larger data summary. Its internal nodes are costly to compare variable-length strings, and its clustering effect is suboptimal, resulting in reduced accuracy for approximate query answering [32, 63]. Tardis [63] uses its proposed sigTree to index data series but faces the challenge of generating many child nodes during each split. These child nodes fail to efficiently use space, leading to low I/O utilization when exchanging data with the disk. Moreover, like another distributed index [62], Tardis is only suitable for batch processing scenarios and cannot handle updating workloads. Coconut [30–32] leverages the Z-order curve to sort the SAX summaries but merely treats them as sortable strings, failing to utilize SAX's full properties. Additionally, Coconut relies on the B^+-tree [16] to enhance approximate query answering but does not optimize exact query answering.

In general, previous works cannot outperform in both writing and query answering processes. The primary reason is that their index structure itself is imperfect or they fail to fully exploit the properties of SAX.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Our Solution. To address these problems, we propose Cherry[1], an innovative approach that outperforms the state-of-the-art works comprehensively, with respect to the aforementioned features of the data series index. Cherry incorporates four key ideas that enable exceptional performance:

Firstly, iSAX-based indices [11, 48, 63, 65] exhibit impressive pruning capabilities, whereas B^+-tree [16] excels in performance. Our aim is to amalgamate the most desirable attributes of both approaches. Unlike Coconut [32], which directly uses the B^+-tree, we recognize the unique properties from the combination of the Z-order curve and SAX, and propose a novel index structure named ZSBtree. By storing the left and right boundaries of child nodes in internal keys, the ZSBtree has the nodes pruning capability in query answering that the B^+-tree does not have. To achieve better clustering, we develop an optimized bulk-loading algorithm that partitions nodes based on the similarity of data summaries.

Secondly, previous works involve numerous random I/O operations [11, 48, 65] and occupy too large space, which is not fully used and unnecessary [63]. Drawing inspiration from the LSM-tree [43], we devise a pipelined writing system to tackle these issues. Additionally, we utilize techniques such as parallelism and compression to further enhance writing speed and minimize space consumption.

Thirdly, the bottleneck of the query answering lies in the accesses and calculations to data summaries and raw data series. Previous works [32, 48, 65] rely on parallelism to diminish answering time, without reducing the amount of accesses and calculations to the data. We employ ZSBtree to effeciently prune data summaries, then employ an algorithm to filter and select the relevant data series, futher reducing both CPU and I/O latency while consuming fewer computing resources. To achieve concurrent reading and writing, we develop a multiversion control system [8].

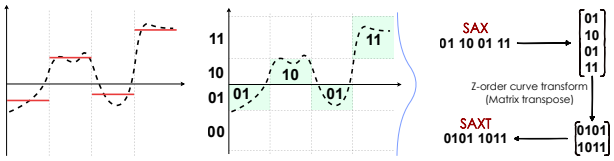
Finally, we harness the parallelism of contemporary hardware, including SIMD [39], a parallel architecture of CPU, to accelerate the expensive vector operations in our algorithms.

In summary, we propose a novel index structure, ZSBtree, and carefully design its writing and query answering algorithms. According to experimental results, Cherry exhibits exceptional performance with minimal computing and storage resources. For instance, even the fastest multi-core architecture index, ParIS+ [48], requires 16-125 times more memory than Cherry and has an index size twice that of Cherry. Despite this, ParIS+ cannot match the performance of Cherry. Specifically, Cherry's index construction speed is 0.38x-0.42x faster than ParIS+, approximate query answering speed is 4.1x-6.2x faster (with an average distance 25% lower than that of ParIS+), and exact query answering speed is 0.7x-10.8x faster.

Contributions. Our primary contributions are as follows:

- We propose ZSBtree, a novel index for data series with an excellent pruning capability and performance. Additionally, we present a bulk-loading algorithm to enhance the similarity of data in each node of ZSBtree, further improving its properties.
- We seamlessly integrate ZSBtree with the LSM-tree and then carefully divide the writing process of ZSBtree into multiple modules to form a pipelined system, achieving a complete overlap of CPU time with I/O time. These operations allow Cherry to achieve more efficient writing and less space usage.

[1]https://github.com/imarcher/Cherry



(a) PAA (b) SAX (c) SAXT

Figure 1: Summarizations

- We design several pruning algorithms based on the properties of ZSBtree and SAX to reduce the amount of accesses and calculations to data summaries and raw data series during the query answering. Combining with the use of multithreading, we significantly reduce the query answering time. Moreover, we provide a multi-version control system to enable read-write parallelism.
- We harness modern hardware parallelism techniques including SIMD, to accelerate the expensive vector computing process in our algorithms. This contributes to a significant reduction of over 50% of the computing time required for this step.
- In our experiments, we utilize various synthetic and real datasets. Firstly, we analyze the impact of our algorithms on Cherry. Then, we conduct comparative experiments with the state-of-the-art works, demonstrating significant improvements in index construction and query answering performance.

2 PRELIMINARIES AND RELATED WORK

Data Series. A data series, TS = $\{a_1, a_2, ..., a_n\}$, is composed of n data points that are sorted based on a certain criterion, such as time or location. This series can be viewed as a high-dimensional vector with a dimensionality of n.

Distance Measures. When dealing with data mining tasks, Dynamic Time Warping (DTW) [52] is generally considered effective in measuring the distance between two data series. However, in situations where the collection of data is exceptionally large, the effectiveness of Euclidean Distance (ED) [2] is equivalent to that of DTW [66]. Moreover, the performance of ED is significantly higher than that of DTW. Therefore, in cases involving large data loads, ED is the preferred option [11, 30, 48, 65].

Similarity Search. Many data mining tasks require similarity search on data series, specifically via k-Nearest Neighbor (k-NN) queries. The exact query answering entails identifying the k data series with the smallest Euclidean Distance (ED) within a collection of data series, based on a given query (a data series, k). The approximate query answering returns k answer with the small ED.

Space-filling Curve. Space-filling curves demonstrate clustering properties and map points in multi-dimensional space to one-dimensional space to facilitate sorting points [7, 15, 33, 34]. Although the Hilbert curve [13] is commonly employed, this work employs the Z-order curve, which outperforms the former despite having inferior clustering properties.

Data Series Summarizations. Due to the typically large size of raw data series, similarity search techniques often entail compressing the data series into a summary and constructing an index based on that summary. Numerous data summarization techniques have been proposed in recent years, including PAA [28], DFT [21],

STS3 [50], SAX [4, 35], PCA [56], APCA [29], iSAX [11, 55], iSAX-T [63], invSAX [30, 32], and others. In this paper, we employ the SAXT summary (Transpose of Symbolic Aggregate approXimation).

Here is the process of how we get a SAXT from a row data series. Firstly, we partition a data series into several segments and obtain its PAA (Figure 1a) representation by computing the average value for each segment. Then, we divide the entire value range into intervals of equal probability using a Gaussian function, and map the PAA representation to discrete intervals to derive the binary representation of SAX (Figure 1b). We consider SAX as a high-dimensional vector and use the Z-order curve to perform dimensionality reduction and enable sorting. That means mathematically, each segment of SAX is assigned to a corresponding row of a matrix, and the resulting matrix is transposed to obtain the SAXT representation(Figure 1c), which is considered as a string that can be sorted. Unlike previous works [30, 32], we treat each transposed row as an indivisible entity.

SIMD. Single instruction, multiple data (SIMD) is a modern CPU technique that can parallelize instruction processing. Both the data series and their summaries mentioned above involve vector operations. If parallelized, program performance can be enhanced.

Similarity Search Indices. Our primary focus is on indices developed based on the SAX summary, which possesses an excellent pruning capability and computing performance [18], rendering it more suitable for large collections of data series.

ISAX [11, 12, 55] offers a variable-size iSAX summary distinct from SAX and leverages an iSAX index that resembles a binary tree to index a vast collection of data series and answer queries. Moreover, it provides a way to compute the lower bound distance between iSAX and PAA, which is smaller than the Euclidean Distance (ED) between two data series, achieving a more efficient pruning capability. However, the tree depth of the iSAX index is notably high, and its construction speed is no longer sufficient to meet the demand. Furthermore, its clustering effect is unsatisfactory, leading to a lower accuracy of approximate query answering. The iSAX-based improvements mentioned later all have this issue.

ADS [64, 65] proposes an adaptive index based on iSAX. The main idea is the principle of laziness, which means that the index is not constructed in one step. Instead, further construction is done when the query reaches the corresponding branch of the index. This approach can reduce the time for index construction and enable faster query answering initiation.

ParIS [46, 48] cleverly optimizes ADS by leveraging modern CPU multi-core architecture. It parallelizes operations as much as possible, maximizing the overlap between I/O time and CPU time. And it utilizes SIMD instructions to accelerate the calculation of the lower bound distance of SAX and PAA, and the ED of two data series.

Tardis [63] designs a distributed index. It proposes using iSAX at the word-level (a row of the transposed matrix in Figure 1c) instead of character-level (an element of the matrix in Figure 1c) to construct the index, achieving a higher accuracy for approximate query answering. However, their algorithm for splitting nodes may lead to low space utilization within numerous nodes.

Coconut [30–32] proposes using the Z-order curve to reduce the dimensionality of SAX, to obtain a sortable summary and then employs traditional B^+-tree or LSM-tree to construct index and answer queries. However, Coconut does not take into account the inherent properties of SAX and treats it as a regular high-dimensional vector, resulting in missed optimization opportunities.

Cherry utilizes a range of writing and query answering optimization algorithms relying on our proposed ZSBtree, an index that exhibits exceptional performance and fully leverages the properties of SAX, outperforming the state-of-the-art works in all aspects.

Table 1: Notations and definitions

Notation	Definition
TS	Data series or time series
PS	Number of PAA or SAX segments
CB	Cardinality bits of a single segment of SAX
SAXT	Transpose of the SAX summary that has *CB* words
SAXT(n)	First *n* words of a SAXT
COD	Common prefix degree of multiple SAXTs
LN	Lower bound on the number of keys in a node
UN	Upper bound on the number of keys in a node
LBD	Lower bound distance between SAX and PAA
TKD	The largest distance in top *k* answers

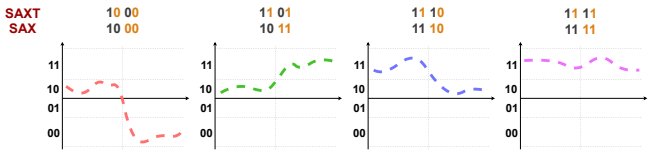


Figure 2: TS and their SAXTs

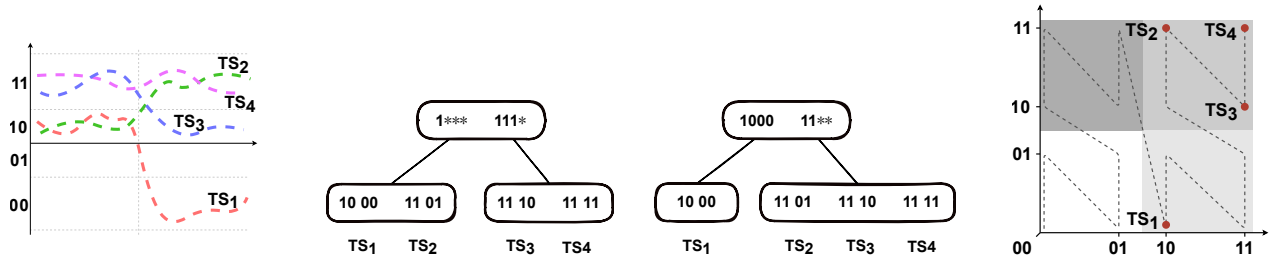
3 THE ZSBTREE INDEX

In this section, we propose our index, ZSBtree. We first discuss an idea which runs through our algorithms (§ 3.1). Then we provide details on the specific structure of ZSBtree and explain its motivations (§ 3.2). Finally, we present insertion (§ 3.3) and bulk-loading (§ 3.4) algorithms for ZSBtree. For the sake of clarity, we refer to each row of the SAX matrix as a "segment", and each row of the SAXT matrix as a "word" (Figure 1c). The notations used are listed in Table 1.

3.1 SAX and Z-order Curve

With the consideration that the Z-order curve has two properties that allow SAX to be sorted by transforming SAX to SAXT and cluster similar SAXTs together, we aspire to fully leverage these two attributes. The first property enables us to use sort-based indices. We analyze how to better utilize the second property below.

We discuss the basic idea with an example. The overview of the example is as follows. Referring to Figure 2 ($CB = 2, PS = 2$), here are 4 sorted TS by their corresponding SAXTs. TS_2, TS_3, and TS_4 are similar, while TS_1 is an outlier (Figure 3a). Median-based partitioning approach (Figure 3b) groups TS_1 and TS_2 together,



(a) TS (b) Partition SAXTs by bit (c) Partition SAXTs by word (d) Z-order curve

Figure 3: Disadvantages of median-based partitioning

but word-based partitioning approach (Figure 3c) shows that TS_2, TS_3, and TS_4 should be grouped. To facilitate later discussion, we introduce a novel concept, COD, which is defined in Definition 3.1. For instance, the COD of TS_2, TS_3, and TS_4 is 1.

DEFINITION 3.1. *COD (Degree of Cohesion): The number of the common prefix words shared by multiple SAXTs is referred to as the COD of these SAXTs. Furthermore, the COD of a particular collection of SAXTs is equivalent to the COD of the smallest and largest SAXTs within that collection.*

To better relate the Z-order curve to the analysis, we mark coordinates of SAXs of 4 TS (Figure 2) on the Z-order curve (Figure 3d). It can be observed that those TS located within a smaller square correspond to a higher COD value of SAXTs. Median-based partitioning approach groups TS_1 and TS_2 within the largest square that possesses a low COD. Despite sharing 1 bit, they are not considered similar, and we explain the reason below. Although the Z-order curve clusters SAXTs with common prefixes together, it does not consider a word of SAXT as a whole. Instead, a word is split into *PS* bits, and in the sorting process, the lower bits have higher priority than the higher bits (in a word, the left-to-right bit order corresponds to lower-to-higher bit order). This order is artificially set without knowing which segment is more important, so sorting bits in a word lacks of consideration. For instance, the first bit of the first segment of all TS in Figure 2 is 1, which is insufficient to get them divided, while that of the second segment is able to divide them into two groups (Figure 3d). It perfectly shows the fact in Figure 3a that the second segment has a higher value in TS partition. Consequently, making the first segment the priority (Figure 3) in clustering is imperfect.

To address this issue and harness the clustering property more effectively, we suggest a more robust node partition approach based on words (or COD). When encountering a word, we can identify several segments with the most significant partition influence in the current context. This is because viewing a word individually is like observing each segment of SAX. Given that lower bits of each segment in the SAX are more informative than higher bits [11], we prioritize using the lower words for partition. As illustrated in Figure 3c, the COD of TS_1 and TS_2 is 0, indicating dissimilarity, and therefore TS_1 will not be grouped with TS_2. Conversely, TS_2, TS_3, and TS_4 have a COD of 1, and thus will be grouped together. Evidently, maximizing the COD per node yields greater rewards than maximizing the number of bits per node prefix. Note that [63] also propose to treat a word as a whole, but we get this conclusion from another perspective and our motivations are also different. Considering word as a whole is equivalent to reducing SAXT's dimensionality, contributing to improved performance. Our index construction and query answering algorithm prioritize the COD of the SAXT collection, with words serving as the fundamental unit.

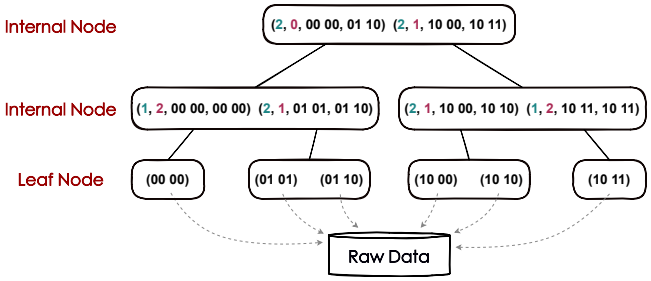


Figure 4: ZSBtree

3.2 Index Structure

As mentioned above, Z-order curve offers two important properties for SAXT, sorting and clustering, and we require an index to take full advantage of these properties. Coconut [32] employs a B^+-tree that can sort, but cannot maintain the necessary clustering property for effective pruning. This is because each internal key only stores the SAXT value of its child node's left or right boundary. However, the discrete nature of SAXT values makes its representation unable to accurately depict the collection of SAXTs within a node. As a result, it fails to preserve the clustering property adequately, which is crucial for effective pruning. To overcome this limitation, we propose a novel B^+-tree variant, called ZSBtree. In contrast to the B^+-tree, the most significant disparity is that ZSBtree's internal keys represent consecutive clusters of SAXTs on a Z-order curve by preserving the left and right boundaries of child nodes, thus maintaining the critical clustering property which the B^+-tree lacks. The full index structure is shown in Figure 4 ($CB = 2, PS = 2$).

Leaf Key. As a key-value separated structure can offer superior writing speed and adaptability [24, 32, 48], we adopt a de-materialized approach to construct the index. A raw data series is associated with a leaf key, and a leaf node has multiple leaf keys. A leaf key has the structure of ($Time, Pos, SAXT$). $Time$ refers to the timestamp of the data series, which can be null. Pos denotes the disk address

of the data series, while $SAXT$ is its summary. The leaf keys in Figure 4 only shows $SAXT$, while Pos is shown as an arrow.

Internal Key. Each internal node contains multiple internal keys that record the metadata of a leaf node or an internal node. It is structured as ($Num, Cod, Pos, Lsaxt, Rsaxt$). Num represents the quantity of keys within the node. Cod represents the COD of the SAXT collection managed by the node corresponding to the internal key. Pos points to the TS or node, represented as a memory pointer or disk address. $Lsaxt$ represents the left boundary of the SAXT collection, and $Rsaxt$ represents the right boundary. In Figure 4, the internal key (2, 1, 0101, 0110) corresponding to the second leaf node on the left, stores the smallset SAXT (0101) and the largest SAXT (0110). It also stores the number (2) of keys in the leaf node and the COD (1) of 0101 and 0110.

Apart from the key content alteration and the lack of connections between leaf nodes, the ZSBtree remains similar to the B$^+$-tree, and preserves the desirable feature that the number of keys contained in each node is between the minimum and maximum values we set (as LN and UN). Unlike the B$^+$-tree, which only stores a single boundary, the internal key of the ZSBtree stores both boundaries of a node so that each ZSBtree internal key can represent its corresponding SAXT collection, exhibiting Z-order curve clustering properties with full precision. These boundaries enable ZSBtree to efficiently prune at arbitrary position, relying on two key point:

Firstly, SAXT can be used to calculate the distance with PAA by transposing to SAX (we can calculate a lower bound distance using SAX and PAA [11]). A special SAXT can be used for pruning the ZSBtree instead of any SAXT contained in the node corresponding to an internal key and its children, thus endowing ZSBtree with pruning functionality. Specifically, the Cod and $Lsaxt$ the internal key stores produce $Lsaxt(Cod)$. The $Lsaxt(Cod)$ is a special SAXT, which represents all SAXTs of this node and its children. For the aforementioned internal key (2, 1, 0101, 0110) in Figure 4, the special SAXT is 01 with $CB = 1$ and $PS = 2$. We can use $Lsaxt(Cod)$ and the PAA of a query's TS to calculate a lower bound distance (LBD) which is smaller than the true lower bound distance of any contained SAXT and the query's PAA [11]. The subtree can be pruned if the value exceeds the ED of the k-th answer.

Secondly, we can obtain the special SAXT and perform ZSBtree pruning efficiently at any position by constructing internal keys from the bottom up and assigning each node its own internal key. We can create a new internal key for each node in the tree from keys that it contains. In this way, propagating the clustering information contained in the leaf nodes upwards in the tree enables the entire ZSBtree to have clustering property. In Figure 4, an internal key (2, 1, 1000, 1011) in the root node and the aforementioned internal key (2, 1, 0101, 0110) which is not on the same layer as the prior key, have the special SAXT 10 and 01, respectively.

3.3 Insertion

Our insertion algorithm is fundamentally similar to that of the B$^+$-tree, with one key difference. We store node boundaries, but there could be a gap between the boundaries of two nodes, which cannot cover the entire value domain. If a SAXT to be inserted falls into a gap, we calculate two CODs by merging it with the front and back nodes respectively. We then insert the SAXT into the node that leads to a smaller drop in the COD value.

Algorithm 1: FindCandidateRanges

```
   Input  : 2UN SAXTs array saxts
   Output : Candidate ranges for nodes construction
1  candidateRanges ← ∅;
2  d ← getCodFrom(saxts[0], saxts[2UN − 1]);
3  lId ← 0, rId ← lId + LN − 1;
4  mark ← true;
5  while true do
6      if mark then
7          if getCodFrom(saxts[lId], saxts[rId]) = d + 1 then
8              mark ← false;
9          else
10             tmpSaxt ← saxts[rId](d + 1) << (CB − d − 1) ∗ PS;
11             lId ← lowerBound(saxts + lId + 1, saxts + rId, tmpSaxt);
12             rId ← lId + LN − 1;
13             if rId ≥ 2UN then
14                 break;
15     else
16         tmpSaxt ← saxts[rId](d + 1) + 1 << (CB − d − 1) ∗ PS;
17         rId ← lowerBound(saxts + rId, saxts + 2UN, tmpSaxt) − 1;
18         num ← rId − lId + 1;
19         if num ≤ UN then
20             candidateRanges.pushBack({lId, rId});
21         else
22             midId ← lId + num/2;
23             candidateRanges.pushBack({lId, midId − 1});
24             candidateRanges.pushBack({midId, rId});
25         lId ← rId + 1;
26         rId ← lId + LN − 1;
27         if rId ≥ 2UN then
28             break;
29         mark ← true;
30 return candidateRanges
```

3.4 Bulk-loading

ZSBtree provides a significant opportunity for pruning. We can put more similar data into a node to improve its pruning capability with a bulk-loading algorithm that leverages the insights from § 3.1. To obtain nodes containing SAXTs within the $[LN, UN]$ range, we slide a window of size UN over the ordered SAXTs. At each iteration, we gather the SAXTs within the window into a node, and then shift the window forward by UN. Due to the inherent clustering characteristics of Z-order curves, the normal bulk-loading algorithm above can successfully group similar SAXTs within a node. However, it does not fully exploit the COD property. We design an elegant bulk-loading algorithm to increase the CODs of nodes without incurring any additional time. The main idea is to increase the COD of a node from d to $d + 1$.

We employ a sliding window of size $2UN$ to create the nodes. Each operation on SAXTs in the window is divided into two phases. In the first phase, we select the left and right boundaries that correspond to the range of SAXTs contained in the candidate nodes (or ranges). If the COD of the SAXTs in the sliding window is d. We choose candidate ranges where the number of SAXTs falls between $[LN, UN]$, and their COD is $d + 1$. To satisfy the constraint of $[LN, UN]$ keys per node, a second stage involves traversing and considering through candidate ranges and the corresponding gap ranges (SAXTs between the right boundary of the previous candidate range and the left boundary of the next candidate range).

Stage 1: Finding Candidate Ranges. The algorithm is shown in Algorithm 1. We first calculate the COD of SAXTs within the whole window, which is denoted as d (Line 2). Next, we select a smallest candidate left boundary (lId) and a largest candidate right boundary (rId) with the following criteria: the number of SAXTs within the boundaries should be in $[LN, UN]$, and their COD should be $d + 1$. Our algorithm consists of two phases: seeking lId (Lines 7-14)and determining rId (Lines 16-29).

Algorithm 2: ConstructNodes

```
   Input  : 2UN SAXTs array saxts, Candidate ranges candidateRanges
1  todoId ← 0;
2  foreach {lId, rId} in candidateRanges do
3      todoNum ← lId − todoId;
4      if todoNum = 0 then buildNode(saxts, lId, rId);
5      else if todoNum < LN then
           // preCodDrop ← +∞ if no node in the front
6          preCodDrop, nextCodDrop ← Merge the gap range with two nodes;
7          preNum ← The number of SAXTs in the previous node;
8          nextNum ← rId − lId + 1;
9          if preCodDrop < nextCodDrop then
10             tmpNum ← preNum + todoNum;
11             Merge saxts[todoNum, lId − 1] with the previous node;
12             if tmpNum > UN then Split the previous node into two nodes;
13             buildNode(saxts, lId, rId);
14         else if preCodDrop > nextCodDrop then
15             tmpNum ← todoNum + nextNum;
16             if nextNum ≤ UN then buildNode(saxts, todoId, rId);
17             else
18                 buildNode(saxts, todoId, todoId + tmpNum/2 − 1);
19                 buildNode(saxts, todoId + tmpNum/2, rId);
20         else
21             if preNum + todoNum ≤ UN then
22                 Merge saxts[todoNum, lId − 1] with the previous node;
23                 buildNode(saxts, lId, rId);
24             else if nextNum + todoNum ≤ UN then
25                 buildNode(saxts, todoId, rId)
26             else
27                 tmpNum ← todoNum + nextNum;
28                 buildNode(saxts, todoId, todoId + tmpNum/2 − 1);
29                 buildNode(saxts, todoId + tmpNum/2, rId);
30     else if todoNum ≤ UN then
31         buildNode(saxts, todoId, lId − 1);
32         buildNode(saxts, lId, rId);
33     else
34         buildNode(saxts, todoId, todoId + todoNum/2 − 1);
35         buildNode(saxts, todoId + todoNum/2, lId − 1);
36         buildNode(saxts, lId, rId);
37     todoId ← rId + 1;
38 if todoId = 0 then buildNode(saxts, saxts, 0, UN − 1) ;
```

In the first phase, we set lId to 0 (Line 3) and rId to $lId + LN - 1$ (Lines 3, 12, 26) . This is because if lId is the left boundary, the subsequent LN SAXTs must have a COD of $d + 1$ according to the definition of the candidate range. We then verify whether the COD of SAXTs within the boundaries, set by lId and rId, equals $d + 1$ (Line 7). If it does, we move to the second phase (Line 8); otherwise, we perform binary search for a new lId within $(lId, rId]$ such that the COD of the SAXTs between the new lId and rId is $d + 1$ (Lines 10-11). The rationale for this approach is that if there are SAXTs between lId and rId belonging to the candidate range with the least LN SAXTs, then rId is within that range. Then, if rId falls within the candidate range, then the smallest SAXT satisfying a COD of $d + 1$ with rId should be chosen as the new lId. We repeat the above process until the second stage begins or the window is fully scanned.

In the second phase, we employ binary search to locate the largest SAXT with a COD of $d + 1$ with lId as rId (Lines 16-17), which is the actual right boundary of the candidate range. If the number of SAXTs found is no greater than UN, we treat the SAXTs as a single candidate range, otherwise we divide SAXTs into two equally sized ranges (Lines 19-24). This process is repeated until the entire window is traversed (Lines 13-14, 27-28).

Stage 2: Constructing Nodes. The algorithm is illustrated in Algorithm 2, where each node's range is considered based on the candidate ranges. The main idea of the algorithm is to analyze different scenarios, where we traverse the candidate and gap ranges in batches while processing SAXTs. Each batch includes a gap range $[todoId, lId - 1]$ (where COD=d) and a candidate range $[lId, rId]$ (where COD=$d + 1$), with $todoId$ as the left boundary of the SAXTs to be processed and rId as the right boundary. The quantity of SAXTs in the gap range (Line 3) is the basis for discussion.

If there are no SAXTs in the gap range, we construct a node directly from the SAXTs of the candidate range (Line 4). If the quantity of SAXTs falls within $[LN, UN]$, we form one node from the gap range and another from the candidate range (Lines 30-32). If the quantity exceeds UN, the gap range is split into two nodes of equal size, and the candidate range forms a single node (Lines 33-36). If the number in the gap range is less than LN, we first check whether a node precedes the gap range. If negative, we amalgamate it directly with the candidate range (Lines 14-19). In the presence of a node, we choose to merge with either the node in front or the candidate range in the back. The objective of the merging process is to maximize the sum of CODs of all nodes. We calculate CODs of node and candidate range before and after merging, and compare to select the one with a smaller drop (Lines 6-19). If COD decrease is equal, we consider resulting SAXTs number and select range with count between LN and UN (Lines 20-29). If all SAXTs are exhausted without forming a node, it indicates the initial UN SAXTs are insufficient to form a node with a higher COD. Therefore, we construct a node using the first UN SAXTs (Line 38).

In our algorithm, the basic unit of operation is SAXT, which can actually be replaced by leaf key and internal key, as they can be represented using one or two SAXTs they include. Compared to normal bulk-loading, our algorithm mainly consumes extra time in the first stage of filtering candidate ranges (Algorithm 1), where the time is mainly spent on binary search. The range size of each binary search is $(0, 2UN]$, and the time complexity is approximately $O(\log_2 UN)$. Assuming the total data size is M, the number of nodes selected by bulk-loading is about M/UN, and each binary search generates approximately one node, leading to a time complexity of $O(M/UN * \log_2 UN)$ for the bulk-loading algorithm. In fact, there are other time costs during the construction, such as the sorting time complexity of $O(M \log_2 M)$, which greatly exceeds the bulk-loading time complexity, so our algorithm has little impact on the total time consumption.

4 WRITING SPEEDING UP

In situations where massive data series are continuously being inserted, traditional index structures may require a large number of random I/O operations, which leads to poor performance. To achieve higher writing performance, our algorithm is designed based on the concept of LSM-tree, as it facilitates sequential read and write operations. We implement our algorithm using LevelDB [22] and make significant modifications to further enhance the writing performance. Firstly, we redesign the insertion process by transitioning from a single-table structure to a multi-table structure [24, 40] and implementing multithreaded writing and compaction. Those techniques allow us to divide the entire system into multiple modules, forming a novel pipeline-style insertion system (§ 4.1). Secondly, we introduce two small technologies for Memtable, template trees [10] and prefix compression (§ 4.2).

4.1 Index Construction System

As depicted in Figure 5, our index construction system consists of several components. Firstly, the TS Reader fetches the raw data

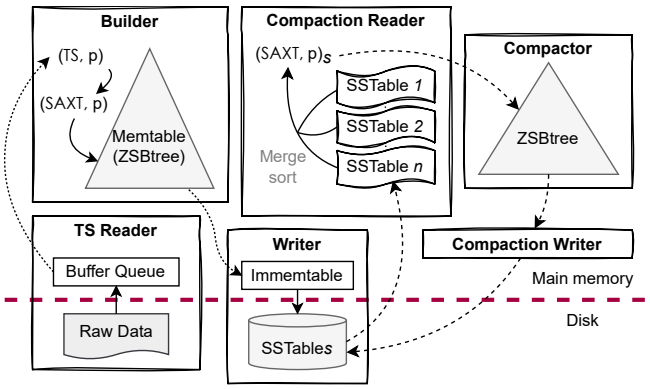


Figure 5: Index construction system

series and saves it in the Buffer Queue. Then, the Builder extracts the TS from the Buffer Queue, converts it into SAXT, and inserts it into the Memtable. Next, the Writer obtains the full Memtable, referred to as Immemtable, and compresses it into a SSTable at level 0. The remainder of the construction process involves LSM-tree compaction. The Compaction Reader reads the SSTables that require compression and applies a merge-sort to the SAXTs from SSTables. The Compactor continually retrieves the sorted SAXTs and performs a stream-based bulk-loading. When the generated index reaches a specific size, the Compaction Writer compresses it into a new SSTable. Note that the TS Reader, Writer, Compaction Reader, and Compaction Writer are primarily responsible for consuming I/O resources. These components are separated from the rest of the system to enable the overlap of disk accesses with CPU operations. The entire process is performed in a pipeline style, with each module separated and highly scalable. Additionally, each module can be executed on multiple threads, which reduces the CPU time required for that particular module.

4.2 Memtable

After compressing the Immemtable into a SSTable, a new Memtable creation involves expensive node splitting operations during insertion. Since the distribution of time-series data is generally continuous, data distribution is not expected to change dramatically. Therefore, we can directly reuse the ZSBtree structure from the previous Memtable and apply a load balancing policy, achieving faster performance than building a new ZSBtree from scratch. However, using the template tree may give rise to two issues. Firstly, the Memtable insertion process does not employ our bulk-loading algorithm, which fails to leverage the pruning capability provided by the ZSBtree fully and does not increase the overall CODs. However since Memtables represent only a small fraction of the data in the LSM-tree system and are frequently transformed to SSTables by bulk-loading during the compression process, the overall impact is insignificant. Secondly, if the data distribution is continuously changing, the ZSBtree may be unbalanced. There are two solutions: (i) Rebuild in bulk when the table becomes unbalanced. (ii) Replace with traditional skip list and rebuild as ZSBtree when compressing to a SSTable. However, the performance and accuracy of query answering in-memory data may be slightly worse.

Algorithm 3: ApproximateQueryAnswering

```
Input  : A query (ts, k, sTime, eTime)
Output : k answers
heap ← ∅;
saxt, paa ← getSaxtFromTs(ts);
nowCb ← CB;
isFirst ← true;
while heap.size() < k do
    tables ← SSTables found from the R-tree according to (saxt(nowCb), sTime, eTime);
    if isFirst then
        Add a Memtable found from Memtables to tables;
        isFirst ← false;
    candidates ← Get (lbd, pos)s from tables in parallel according to paa;
    // Get Answers with the Least I/O (GAL)
    candidates.sort();
    todo ← true;
    while candidates.size() and todo do
        posQueue ← Get some pos from candidates;
        if heap.size() = k then
            todo ← All lbds of posQueue <= heap.top;
            posQueue ← Some pos in posQueue whose lbd <= heap.top();
        posQueue.sort();
        while posQueue.size() do
            posBatch ← Get some pos in posQueue according to disk queue length;
            tsSet ← Get (dist, ts)s from tsfiles according to posBatch;
            foreach item in tsSet do
                if heap.size() < k then
                    heap.push(item);
                else if item < heap.top() then
                    heap.pop(), heap.push(item);
    nowCb ← nowCb − 1;
return heap
```

When converting the Memtable into SSTable, we utilize node-based compression. Our indexing method groups similar SAXTs into a node and compresses keys by merging prefixes based on the node's COD. For instance, given two SAXTs, 1010 and 1011, with a COD of 1 and a PS of 2, we can only store the last words, 10 and 11. Each table, namely the Memtable and SSTable, retains the time boundaries of all leaf keys. This allows us to filter tables based on time boundaries when answer the time-windowed queries.

5 QUERY ANSWERING

We design novel algorithms for both approximate query answering (§ 5.1) and exact query answering (§ 5.2), with the aim of achieving unparalleled accuracy and efficiency. Previous works [32, 48, 65] focus on using multithreading to reduce query answering time. However, our approach aims to exploit the pruning capability of ZSBtree and SAX, which can minimize the amount of accesses and calculations to data summaries and raw data series during query answering, thereby fundamentally enhancing the speed of query answering. Naturally, we also employ a sorting algorithm to make the disk access pattern more closely resemble sequential access to reduce I/O time and multithreading to reduce CPU time. Additionally, we design a multi-version control system to enable parallel read and write operations (§ 5.3).

5.1 Approximate Query Answering

The approximate query answering can be decomposed into three distinct stages, as described in detail in Algorithm 3. (i) We obtain the required tables for the query answering. (ii) We retrieve potential leaf keys by searching tables. (iii) We get query answering results based on the leaf keys.

Stage 1: Obtaining Tables. Prior to conducting an approximate or exact query answering, we obtain a snapshot and only execute query answering based on the tables within the snapshot,

which makes the entire system remain static. The LSM-tree architecture includes Memtables and SSTables, and the query answering must filter tables based on conditions. To process a query ($ts, k, startTime, endTime$), we calculate $saxt$ and paa from ts (Line 2). Then, we use $saxt(CB)$ and the query's timestamp range to find SSTables in the R-tree that intersect with the query. In this context, $saxt(CB)$ can refer to the smallest and largest SAXT values with the same prefix of $saxt(CB)$. Furthermore, we obtain the relevant Memtable for the query according to Memtables' metadata (Lines 6-9). Once we have Memtable and SSTables, we obtain the candidate leaf keys for all relevant tables and search for the corresponding TS to update query answering results. If there are not enough k answers that meet the condition, we use $saxt(CB-1)$ and the query's timestamp range to search for more SSTables from the R-tree. Then we repeat the query answering. We decrease CB until the number of answers reaches k (Lines 5, 6, 27).

Stage 2: Retrieving Leaf Keys. The ZSBtree clusters similar SAXTs, making it easy to find the relevant leaf nodes and extract the time-constrained leaf keys based on the query's $saxt$. However, this process may face two challenges. For one thing, a narrow time range or a large value of k of the query can result in a scarcity of qualified leaf keys relatively within a leaf node. For another, if the size of a leaf node is small, the answers found in the node may lack sufficient accuracy. To tackle these concerns, we employ a search-in-sibling-nodes strategy. As internal keys of leaf nodes undergo bulk-loading, similar nodes are clustered together. Here, we propose an approach: while searching for SSTables with level > 0, we consider exploring the left and right sibling nodes of the corresponding leaf node. This is due to the fact that merged SSTables are more condensed, necessitating more node searching, leading to a substantial enhancement in query answering precision. In addition, we utilize multiple threads to simultaneously search multiple tables, speeding up the process of acquiring leaf keys (Line 10).

Stage 3: Getting Query Answering Results. We emphasize the three techniques used to enhance system performance in the processing of getting results. These three acceleration techniques are referred to as the *Get Answers with the Least I/O* (GAL).

Firstly, the leaf keys can be converted to (lbd, pos)s. The lbd is computed using the internal SAXT and the paa of query. And the pos comes from the leaf key itself. Then we sort the tuples in ascending order based on their $lbds$ (Line 11). If we have already obtained k answers, we can compare the lbd with TKD. If the lbd exceeds TKD, then the TS corresponding to pos is not a potential answer, and we need not access the TS. Placing tuples which have larger $lbds$ towards the end of processing queue can increase the opportunity to filter them out, thereby reducing I/O costs and computation. However, if the obtained leaf keys account for more than half of the entire index, it is recommended to employ pos for sorting, thereby enabling sequential disk access.

Secondly, we extract a certain number of tuples from the front and sort them based on their pos (Lines 14-18). Sorting is preferred to optimize disk access for sequential access, achieving faster I/O speed.

Finally, we check the size of the disk queue and add the same number of pos to the queue for accessing TS (Lines 20-21). As the pos order may not reflect the real order on the disk, allowing the disk to optimize the scan sequence again can further improve

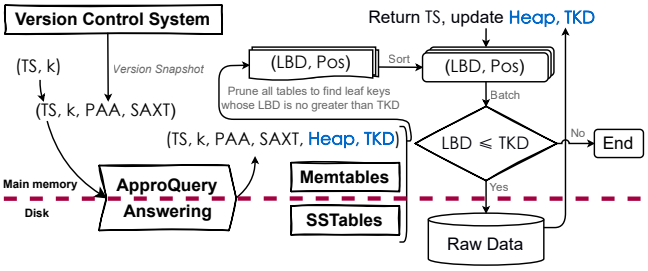


Figure 6: Exact query answering

Algorithm 4: ExactQueryAnswering

```
Input  : A query (ts, k, sTime, eTime)
Output : k answers
1  heap ← approSearch(ts, k, sTime, eTime);
2  tables ← SSTables found from the R-tree according to (sTime, eTime);
3  Add Memtables to tables according to (sTime, eTime);
   // Different from approximate query answering
4  candidates ← Get (lbd, pos)s from tables in parallel according to heap;
5  Get Answers with the Least I/O (GAL);
6  return heap
```

efficiency. After accessing TS, we can calculate the ED with the query's TS. If it is less than TKD, we add the corresponding TS to the answers and update TKD (Lines 22-26).

5.2 Exact Query Answering

Figure 6 illustrates the process of an exact query answering. An exact query is represented by the tuple ($ts, k, startTime, endTime$), that requires retrieval of the k most relevant answers. Examining all TS and their timestamps for appropriate answers can be time-consuming. Therefore, pruning techniques and high efficiently disk access are necessary to reduce computing and I/O costs.

We have two strategies for pruning. The first strategy utilizes the inherent properties of SAX. Specifically, for our leaf keys, the LBD calculated from the PAA of query's ts and the SAXT will always be greater than the true distance. In case this distance is already greater than the TKD, we need not access the raw data series. The second strategy leverages the attributes of the ZSBtree. Each internal key in the ZSBtree having a COD greater than 0 can be considered as a SAXT. A LBD can be computed between this SAXT and the PAA of query's ts. If this LBD is greater than TKD, then all members of this subtree can be pruned. To enhance the overall speed of query answering, we first adopt an approximate query answering to retrieve k estimated answers. This enables us to leverage TKD to boost the pruning efficiency of the overall process.

Compared to the approximate query answering, the exact query answering requires more time in three areas. (i) The approximate query answering investigates only a fraction of the tables that may contain potentially answers, while the exact query answering explores all the tables with overlapping time ranges. (ii) The approximate query answering examines only the most promising leaf nodes of each table, while the exact query method traverses all the nodes of the entire table. (iii) Owing to the cumulative impact of the first two aspects, the approximate query answering accesses seldom raw data series, whereas the exact query accesses a considerable number of raw data series, leading to a significant I/O overhead.

Algorithm 5: UpdateVersion

```
Input  : Current version curVer, New version content verC
Output : A new version
1  if curVer.ref > 1 then newVer ← Copy(curVer);
2  else newVer ← curVer;
3  curVer.ref − −;
4  if curVer.ref = 0 then
5      curVer.memVer.ref − −, curVer.diskVer.ref − −;
6      if curVer.memVer.ref = 0 then Delete(curVer.memVer);
7      if curVer.diskVer.ref = 0 then Delete(curVer.diskVer);
8  newVer.memVer ← verC.memVer, newVer.diskVer ← verC.diskVer;
9  newVer.rTree.add(verC.addFiles, verC.addSaxtRG, verC.addTimeRG);
10 newVer.rTree.delete(verC.delFiles, verC.delSaxtRG, verC.delTimeRG);
11 newVer.ref ← 1;
12 newVer.memVer.ref ← newVer.memVer.ref + 1;
13 newVer.diskVer.ref ← newVer.diskVer.ref + 1;
14 return newVer
```

To overcome the aforementioned challenges, we employ the following three strategies. (i) We harness the power of multithreading to expedite the query answering by assigning each thread to query one or multiple tables concurrently. (ii) During the DFS traversal of a ZSBtree, we utilize the second pruning method mentioned above for internal keys, and the first pruning method for leaf keys to further reduce I/O and computing time. (iii) We leverage the *Get Answers with the Least I/O* (GAL) technique, which we introduce in the approximate query answering section (§ 5.1).

The specific algorithm of the exact query answering refers to Algorithm 4. Firstly, the answers obtained from the approximate query answering are placed in a heap (Line 1). Next, all SSTables and Memtables meeting the time conditions are located (Lines 2-3). Then, we select the *pos* and corresponding *lbd* from each table to add to the candidate list in parallel (Line 4). However, unlike the approximate query answering, the entire ZSBtree of each table needs to be searched and the TKD of the approximate answers is used to prune the search and reduce the search volume. Finally, the true answers are obtained using the GAL approach (Line 5).

5.3 Multi-version Control

As data series keeps arriving, there is also a constant influx of data insertion while processing a query answering. To enable parallel read and write operations, a real-time snapshot is necessary to determine which Memtables and SSTables should be accessed when a query is received. A version snapshot structure comprises a R-tree, a reference count, and the corresponding memory and disk version numbers. The R-tree stores the positions of SSTables, maintaining the range of SAXTs and timestamps to enable swift determination of which SSTables are pertinent to a given query. The reference count records whether this snapshot is being used by a query answering. Upon arrival of a query, the reference count is incremented, and upon completion, the reference count is decremented. Due to the possibility of different version snapshots corresponding to the same memory and disk version numbers, we need two global Maps to record the corresponding version numbers and their respective reference counts for both memory and disk versions. This approach ensures no version snapshots are using any memory and disk versions removed, thereby preventing potential conflicts.

We update the version when compacting a Memtable into a new SSTable or merging SSTables (Algorithm 5). To minimize overheads in copying snapshots, the original snapshot is updated directly if no query answering involved (Lines 1-2). If a new version snapshot is created, the reference count of the old snapshot is decreased (Line 3). If no query answering owns it, the memory and disk versions of the old snapshot are released (Lines 4-7). The new version snapshot is then updated (Lines 8-10), with the reference count of itself being set 1 (Line 11). And the reference counts of its memory and disk versions are also increased (Line 12-13).

6 VECTOR OPERATIONS ACCELERATING

There are multiple vector operations throughout the algorithmic process. The vector operations in our Builder module (§ 4.1) account for more than half of the execution time. By effectively utilizing modern hardware features, we improve the parallelism of vector operations and greatly enhance overall computing performance. Our experiments indicate that these optimizations make vector operations 1.2x faster.

TS to PAA. The essence of this operation lies in computing the mean of a group of float numbers. By leveraging SIMD instructions, we can add two sets of 8 floats in a single operation, thereby minimizing the number of addition operations required.

PAA to SAX. Discretizing PAA into SAX requires using a binary search algorithm to determine the corresponding Gaussian interval for each PAA segment. Since SIMD can perform comparison operations on two sets of 8 floats simultaneously, we can optimize the binary search algorithm by implementing an octonary or quaternary search, thereby reducing the number of comparisons required. Specifically, the search space is divided into 8 equal parts. We extract 7 boundary points and fill a *MAXFLOAT* to create an array. We use PAA to compare this array and determine its corresponding part. This process is repeated until the search space is divided into the smallest possible unit. Note that two points should be taken into consideration to avoid any negative optimization. (i) It is necessary to preprocess the boundaries of the intervals and reserve the array if CB is fixed. (ii) The octonary search involves numerous if-else statements, which may cause performance degradation. We can employ a hash function to overcome this problem. By performing a SIMD comparison, we obtain a binary number, which is mapped to the position of the array to be compared next using hash mapping, thus circumventing the need for if-else statements.

SAX to SAXT. This operation involves the transposition of a SAX matrix with a bit as its fundamental unit. Thanks to modern CPU capabilities, we can directly manipulate 64-bit (8-byte) data, allowing us to decompose the matrix into multiple uint64s and execute vectorized parallel operations. Assuming that we possess a matrix of size 8 bytes and 8 bits per row that needs to be transposed, we can first use bitwise AND operations to extract a single column containing 8 elements, each of which is dispersed across a byte. Then, we use bit shifting to move each number to the highest bit of each byte, and multiply it by 0x8080808080808080 to pack each element into the highest 8 bits of an uint64 variable. This allows us to obtain a row of the transposed matrix. Finally, we perform a bit shift to move this row to its corresponding position. Certainly, using SIMD can also achieve the same effect.

Comparison of SAXTs. Comparing two SAXTs is essential for various operations such as index insertion, query answering, and merge sorting. Our SAXT storage scheme uses a little-endian pattern, meaning that we store the SAXT low bits (the more significant

bits) in the high address location. Since modern systems use 64-bit architecture, we can split SAXT into 8-byte integers for comparisons. While previous works [32, 63] treat SAXTs as strings and compare them based on 1-byte characters, our approach achieves higher performance by reducing the quantity of comparisons.

7 EXPERIMENTAL EVALUATION

In this section, we first evaluate the impact of our algorithms and parameters on Cherry (§ 7.2). Then, we conduct extensive comparative experiments with the state-of-the-art works, demonstrating the superiority of Cherry (§ 7.3).

7.1 Framework

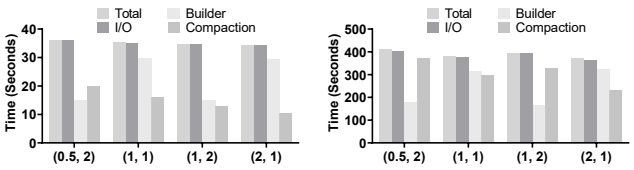
Setup. We conduct the experiments on a server with an Intel Core i9-13900 (36MB cache, 24 cores, 36 hyperthreads), 80GB of RAM, a 2T NVMe SSD with 3GB/sec sequential access throughput and 1.5GB/sec random access throughput. We implement algorithms in C and C++ except the Java interface for Cherry, and use the GCC 9.4.0 with the O3 optimization flag on Ubuntu Linux 20.04.

Datasets. To offer a detailed analysis of Cherry and compare it with existing approaches, we perform experiments on three synthetic datasets containing 100 million, 500 million, and 1 billion data series (dataset sizes are 100 GB, 500 GB, and 1 TB), respectively. Each data series consists of 256 data points. We use a random walk data series generator. At each time point, it adds a new number drawn from a Gaussian distribution N(0,1) to the value of the last number. This kind of data generation has been shown to effectively model real-world financial time series [20] and been extensively used in the past [2, 11, 48, 51, 65]. Our query dataset consists of 70% inserted data and 30% newly generated data.

We also use two real datasets to verify the adaptability of our work. (i) TEXMEX [26] contains 1 billion SIFT feature vectors of size 128 representing images. (ii) Deep [5] also contains 1 billion vectors, each of size 96, extracted from the last few layers of a convolutional neural network. We use the entire TEXMEX along with 50% of the Deep, and perform Z-normalization on both. The queries are generated by selecting series from the datasets and adding 5% Gaussian noise ($\mu = 0, \sigma^2 = 0.05$).

Algorithms. We conduct comparative experiments with the state-of-the-art works. ADS+ [65] is an adaptive data series index. It uses large leaf nodes during index construction and splits the nodes during query answering to improve the index construction speed. ParIS+ [48] parallelizes ADS+ using the parallelization characteristics of modern hardware. To the best of our knowledge, it is the fastest multithreaded algorithm at present. Coconut [32] proposes that iSAX [11] can be sorted, and uses the traditional B^+-tree. We utilize code obtained from the original authors or their respective laboratories in our experiments. For the ParIS+ construction thread number, we configure it to 6, as we use a faster SSD. The remaining experimental parameters are adopted from the original paper.

The default parameters are as follows: $PS = 16$, $CB = 8$, $UN = 512$, and $LM = UN/2$. We use one Memtable with a size of one million leaf keys. Except for the Compaction Reader, which uses two threads, the index construction system (§ 4.1) employs one thread per module, with a total of six threads, but only Builder and Compactor processes have intensive CPU operations, while the other threads are mainly used to enable asynchronous I/O. In contrast, ParIS+ [48] uses six threads, with five of them engaging in intensive CPU operations, resulting in higher computing resource consumption. We use 24 threads for query answering. The default size of dataset is 100GB, with 100 exact queries and 100,000 approximate queries. Unless otherwise stated, our queries are 100-NN. Without special instructions, we record the time of processing all queries rather than processing one query. We conduct each experiment more than 5 times and record the average value. Before executing index construction or query answering, we clear the caches.



(a) 100GB (b) 1TB

Figure 7: Impact of Memtable on index construction time

7.2 Performance Analysis

Memtable. In the first experiment, we evaluate the impact of Memtable size and quantity on index construction using datasets of 100GB and 1TB. Each table is assigned a single Builder thread. In Figure 7, the horizontal axis is a tuple with the Memtable size (number of leaf keys) ranging from half a million to 2 million, and the number of Memtables ranging from 1 to 2. Cherry's exceptional speed often contributes to the overlap of CPU time with I/O time. As a result, we measure several time-consuming components separately, which may overlap partially with each other. (i) I/O time, which has three components: raw data series reading, Immemtables for disk writing, and writing the indices generated by compaction to the disk. (ii) The time required for the Builder thread to convert TS to SAXT and insert it into the Memtable. (iii) SSTables compaction time (including CPU time and I/O time).

Figure 7a demonstrates that increasing Memtable size from 0.5M to 2M reduces the frequency of compact operations, thereby halving the compaction time. The overall time decreases slightly with an increase in table size due to reduced I/O resulting from less frequent compaction. However, this reduction is negligible compared to reading the raw data series. Besides, increasing table quantity from 1 to 2 reduces builder time by half.

A similar trend is observed on the 1TB dataset, as seen in Figure 7b. However, the percentage of time spent on compaction increases by 30-40% compared to the 100GB dataset. This is an unique property of LSM-tree, as compaction does not scale linearly.

In general, to minimize the total time, we aim to overlap CPU time with I/O time. Builder and Compactor modules handle CPU operations in our system. To achieve full overlap, we can increase table count and size to speed up Builder and Compactor respectively.

Leaf Size and Bulk-loading. We evaluate the impact of leaf size (128-2048) and our optimized bulk-loading algorithm on performance (Figure 8). "Normal" indicates the conventional algorithm, and "Optimized" indicates our proposed algorithm. For index construction time (Figure 8a), larger leaf sizes slightly reduce the time.

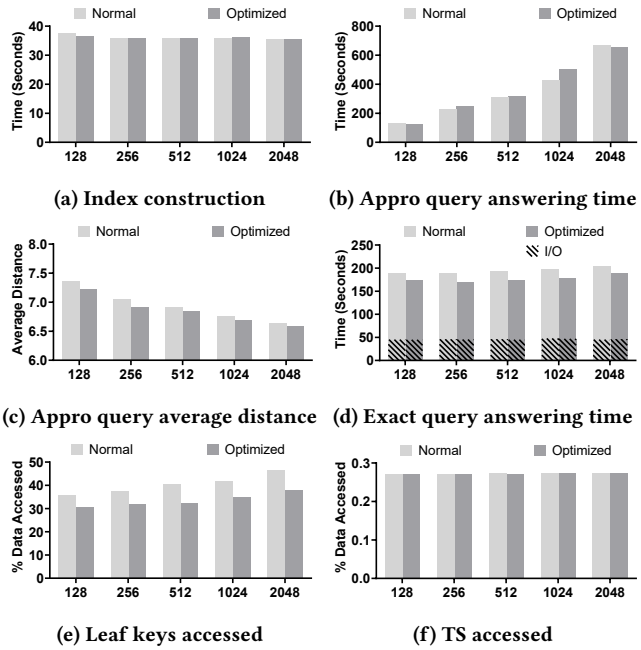


Figure 8: Leaf size and bulk-loading algorithm

Our bulk-loading algorithm has negligible additional time overhead, as expected from theoretical analysis. It is worth noting that, setting the leaf size to 128 causes the normal bulk-loading algorithm to take more time due to numerous Memtable reconstructions during the Builder phase. However, the optimized bulk-loading algorithm avoids this issue. This highlights the potential for the optimized bulk-loading algorithm to better fit the data distribution.

The execution time of the 100,000 approximate query answering (Figure 8b) aligns with expectations, with larger leaf sizes resulting in longer execution time but more accurate results (Figure 8c). In addition, utilizing the optimized bulk-loading algorithm reduces the average distance by about 0.1. To demonstrate the impact of the optimized bulk-loading algorithm on exact query answering time more effectively, we use a single thread. Analyzing the execution time of 100 exact query answering in Figure 8d, we observe that utilizing optimized bulk-loading consistently outperforms normal bulk-loading by approximately 10%. This is because the optimized bulk-loading algorithm allows to cluster more similar SAXTs in a node, achieving more efficient pruning and reducing the number of accessed leaf keys by 14-20% (Figure 8e). The constant I/O time is caused by the utilization of the GAL algorithm (§ 5.1), which accesses only the minimum amount of raw data series (Figure 8f).

Multithreaded Query Answering. In Figure 9a, we demonstrate the influence of multithreading on exact query answering. As the number of threads increases, the overall time decreases progressively. While multithreading can reduce CPU time during query answering, the bottleneck for query answering time is typically the random access of raw data series.

Vector Operations. The execution time for vector operations in the TS to SAXT transformation is shown in Figure 9b. Our algorithm performs 1.2x faster with significant reduction in each stage

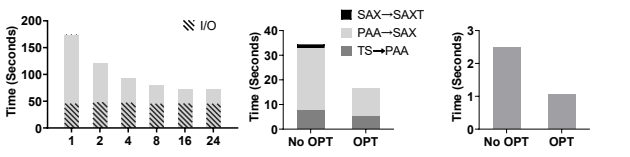


Figure 9: Parallelization

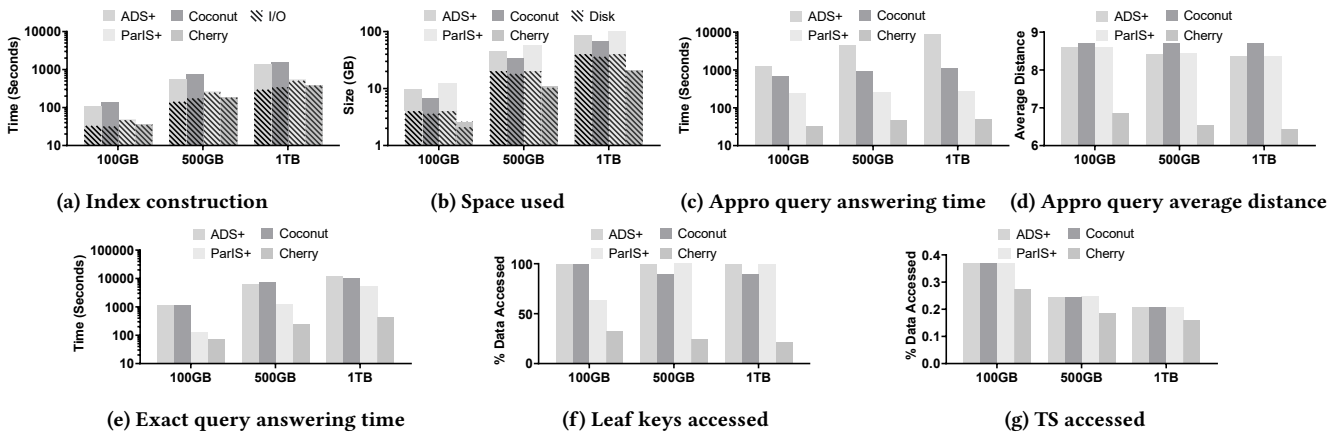
time. Optimization of this period is crucial as it accounts for approximately half of the total time. Figure 9c shows the impact of storage structure (§ 6) on comparing 100 million pairs of SAXTs. Our storage structure achieves a 1.4 times speed improvement.

7.3 Comparative Experiments

Synthetic Datasets. We compare the speed of index construction across datasets of varying size (Figure 10a). Cherry presents a notably improved pace in contrast to ADS+ (by 1.9x-2.5x faster) and Coconut (by 2.9x-3.2x faster). Even the fast index construction technique to date, ParIS+, is 38.6%-42.7% slower than ours. Furthermore, we are the only index whose CPU time is entirely overlaped by I/O time, which means that we can achieve faster performance theoretically by upgrading to better disks. Figure 10b illustrates the total space employed by each approach throughout index construction. Cherry expends no more than 500MB of memory, amounting to a 5-126 fold reduction in memory used compared to preceding works. Besides, our disk space (index size) is minimal, only half of the previous works. Furthermore, due to the implementation of LSM-tree architecture, our memory consumption remains unaffected by dataset size. These experiments demonstrate the efficient writing ability of Cherry.

We perform 10,000 approximate query answering across datasets of diverse sizes. As shown in Figure 10c, Cherry outperforms ADS+ by 37.2x-166.3x, Coconut by 19.9x-20.5x, and ParIS+ by 4.1x-6.2x in terms of query answering performance. In addition, Figure 10d shows that Cherry's approximate query answering results have higher precision, with an average distance 25% lower than previous works. Two factors could led to this result. Firstly, our algorithm helps cluster more similar data series in a node. Secondly, using the LSM-tree architecture may cause more accessed data series during approximate query answering than in previous works.

We conduct 100 exact query answering. The results in Figure 10e show that Cherry is 15x-27x faster than ADS+, 14.6x-26.6x faster than Coconut, 0.7x-10.8x faster than ParIS+. Figure 10f and Figure 10g explain the reasons: Cherry accesses less leaf keys and data series than previous works (accessing 22%-51% and 74%-76% of previous works, respectively). This reduction in the number of accessed keys and data series translates to a noteworthy reduction in CPU and I/O time demands. These results validate our theoretical propositions: (i) The pruning capability of ZSBTree, coupled with our bulk-loading algorithm, allows us to prune unnecessary leaf keys. (ii) The GAL algorithm (§ 5.1) further reduces the accesses of the data series. As shown by the above experiments, ParIS+ always outperforms ADS+ and Coconut. Therefore, in subsequent experiments, we only compare our approach with ParIS+.



(a) Index construction (b) Space used (c) Appro query answering time (d) Appro query average distance

(e) Exact query answering time (f) Leaf keys accessed (g) TS accessed

Figure 10: Compare with the state-of-the-art works

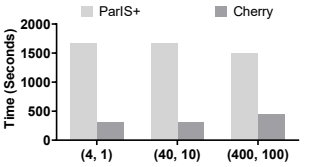


Figure 11: Updates

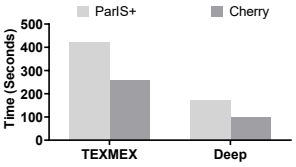
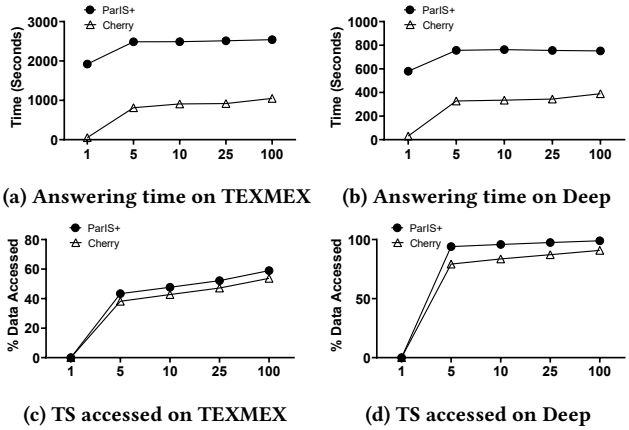


Figure 12: Real datasets



(a) Answering time on TEXMEX (b) Answering time on Deep

(c) TS accessed on TEXMEX (d) TS accessed on Deep

Figure 13: Exact query answering on real datasets

Updating Workloads. We construct an index on a synthetic dataset of 500GB and perform 100 exact query answering. The initial 100GB is used to index construction, while the remaining 400GB is utilized for alternating between query answering and construction. Figure 11 depicts the total time needed under diverse workloads. The first component of each tuple on the horizontal axis signifies the size (GB) of data inserted in each insertion operation, while the second component indicates the number of queries to process after each insertion operation. Cherry is 2.2x-4.4x faster than ParIS+, and exhibits much faster performance under updating workloads than the approach of batch processing. This can be attributed not only to Cherry's fast construction and query answering speed but also its multi-version concurrency control algorithm, which supports concurrent read and write operations and leads to a partial overlap between index construction and query answering time.

Real Datasets. We construct indices for two real datasets. As shown in Figure 12, Cherry still outperforms ParIS+ by 0.63x-0.72x on real datasets. We subsequently conduct 100 exact k-NN (k=1-100) query answering and record the mean time and the number of accessed TS (Figure 13). Figure 13a and Figure 13b demonstrate that Cherry significantly outperforms ParIS+ by 0.9x-34.9x, and the accessed TS of Cherry on real datasets is still 9%-15% lower than that of ParIS+ (Figure 13c and Figure 13d). It is worth noting that when we increase the difficulty of queries, i.e., by increasing k, the performance of both works sharply decline. This is caused by the high similarity of the data series in the two datasets, which makes the search task exceedingly challenging (with Deep being more difficult than TEXMEX). When accessing most of the data in the dataset, the index loses its effectiveness because SAX is unable to differentiate between data series. Increasing the CB and PS may alleviate some of this problem, but it may result in larger space requirements, which is not an optimal solution. The index based on SAX is not suitable for such demanding workloads, and this is an issue that needs to be addressed in the future.

8 CONCLUSIONS

We propose Cherry, a novel data series index that offers significantly higher writing and query answering performance while using minimal resources compared to the state-of-the-art works. We introduce a novel index structure named ZSBtree, which exhibits exceptional performance and space utilization, while retaining the excellent pruning capability of SAX. Subsequently, we design a pipeline system to accelerate the writing process. Then, leveraging properties of ZSBtree and SAX, we devise several pruning algorithms to fundamentally reduce query answering time. Finally, we use hardware features to accelerate vector operations. In our future work, we will explore the integration of Cherry with diverse data summarization methodologies to achieve significant advances in exceedingly demanding similarity search tasks.

REFERENCES

[1] Vaneet Aggarwal, Emir Halepovic, Jeffrey Pang, Shobha Venkataraman, and He Yan. 2014. Prometheus: Toward quality-of-experience estimation for mobile apps from passive network measurements. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*. 1–6.

[2] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. 1993. Efficient similarity search in sequence databases. In *Foundations of Data Organization and Algorithms: 4th International Conference, FODO'93 Chicago, Illinois, USA, October 13–15, 1993 Proceedings 4*. Springer, 69–84.

[3] Noura Alghamdi, Liang Zhang, Huayi Zhang, Elke A Rundensteiner, and Mohamed Y Eltabakh. 2020. ChainLink: indexing big time series data for long subsequence matching. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 529–540.

[4] Ira Assent, Ralph Krieger, Farzad Afschari, and Thomas Seidl. 2008. The TS-tree: efficient time series search and retrieval. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. 252–263.

[5] Artem Babenko and Victor Lempitsky. 2016. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2055–2063.

[6] Andreas Bader, Oliver Kopp, and Michael Falkenthal. 2017. Survey and comparison of open source time series databases. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband* (2017).

[7] Rudolf Bayer. 1997. The universal B-tree for multidimensional indexing: General concepts. In *WWCA*, Vol. 97. 198–209.

[8] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[9] Paul Boniol, Michele Linardi, Federico Roncallo, and Themis Palpanas. 2020. Automated anomaly detection in large sequences. In *2020 IEEE 36th international conference on data engineering (ICDE)*. IEEE, 1834–1837.

[10] Ruichu Cai, Zijie Lu, Li Wang, Zhenjie Zhang, Tom ZJ Fur, and Marianne Winslett. 2017. DITIR: distributed index for high throughput trajectory insertion and real-time temporal range query. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1865–1868.

[11] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn Keogh. 2010. isax 2.0: Indexing and mining one billion time series. In *2010 IEEE International Conference on Data Mining*. IEEE, 58–67.

[12] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn Keogh. 2014. Beyond one billion time series: indexing and mining very large time series collections with SAX2+. *Knowledge and information systems* 39, 1 (2014), 123–151.

[13] Lu Chen, Yunjun Gao, Xinhan Li, Christian S Jensen, and Gang Chen. 2015. Efficient metric indexing for similarity search. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 591–602.

[14] Qiuxia Chen, Lei Chen, Xiang Lian, Yunhao Liu, and Jeffrey Xu Yu. 2007. Indexable PLA for efficient similarity search. In *Proceedings of the 33rd international conference on Very large data bases*. 435–446.

[15] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A Nascimento. 2008. ST2B-tree: a self-tunable spatio-temporal B+-tree index for moving objects. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 29–42.

[16] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[17] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules against data series similarity search. *arXiv preprint arXiv:2212.13297* (2022).

[18] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2020. The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art. *arXiv preprint arXiv:2006.11454* (2020).

[19] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2020. Return of the lernaean hydra: Experimental evaluation of data series approximate similarity search. *arXiv preprint arXiv:2006.11459* (2020).

[20] Christos Faloutsos, Mudumbai Ranganathan, and Yannis Manolopoulos. 1994. Fast subsequence matching in time-series databases. *ACM Sigmod Record* 23, 2 (1994), 419–429.

[21] Hakan Ferhatosmanoglu, Ertem Tuncel, Divyakant Agrawal, and Amr El Abbadi. 2000. Vector approximation based indexing for non-uniform high dimensional data sets. In *Proceedings of the ninth international conference on Information and knowledge management*. 202–209.

[22] Sanjay Ghemawat and Jeff Dean. 2011. LevelDB. https://github.com/google/leveldb.

[23] Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. 2020. iDEC: indexable distance estimating codes for approximate nearest neighbor search. *Proceedings of the VLDB Endowment* 13, 9 (2020).

[24] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-lsm: A distributed, component-based lsm-tree key-value store. In *Proceedings of the 2021 International Conference on Management of Data*. 749–763.

[25] Pablo Huijse, Pablo A Estevez, Pavlos Protopapas, Jose C Principe, and Pablo Zegers. 2014. Computational intelligence challenges and applications on large-scale astronomical time series databases. *IEEE Computational Intelligence Magazine* 9, 3 (2014), 27–39.

[26] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 861–864.

[27] K. Kashino, G. Smith, and H. Murase. 1999. Time-series active search for quick retrieval of audio and video. In *Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on*.

[28] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. 2001. Dimensionality reduction for fast similarity search in large time series databases. *Knowledge and information Systems* 3 (2001), 263–286.

[29] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. 2001. Locally adaptive dimensionality reduction for indexing large time series databases. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. 151–162.

[30] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2018. Coconut: A Scalable Bottom-Up Approach for Building Data Series Indexes. PVLDB 11, 6 (2018), 677–690.

[31] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2019. Coconut palm: Static and streaming data series exploration now in your palm. In *Proceedings of the 2019 International Conference on Management of Data*. 1941–1944.

[32] Haridimos Kondylakis, Niv Dayan, Kostas Zoumpatianos, and Themis Palpanas. 2019. Coconut: sortable summarizations for scalable indexes over static and streaming data series. *The VLDB Journal* 28 (2019), 847–869.

[33] Ken CK Lee, Wang-Chien Lee, Baihua Zheng, Huajing Li, and Yuan Tian. 2010. Z-SKY: an efficient skyline query processing framework based on Z-order. *The VLDB Journal* 19 (2010), 333–362.

[34] Ken CK Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. 2007. Approaching the skyline in Z order.. In *VLDB*, Vol. 7. 279–290.

[35] Jessica Lin, Eamonn Keogh, and Wagner Truppel. 2003. Clustering of streaming time series is meaningless. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*. 56–65.

[36] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. 2007. Experiencing SAX: a novel symbolic representation of time series. *Data Mining and knowledge discovery* 15 (2007), 107–144.

[37] Michele Linardi and Themis Palpanas. 2018. ULISSE: ultra compact index for variable-length similarity search in data series. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 1356–1359.

[38] Vebjorn Ljosa and Ambuj K Singh. 2007. APLA: Indexing arbitrary probability distributions. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 946–955.

[39] Chris Lomont. 2011. Introduction to intel advanced vector extensions. *Intel white paper* 23 (2011).

[40] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[41] Katsiaryna Mirylenka, Vassilis Christophides, Themis Palpanas, Ioannis Pefkianakis, and Martin May. 2016. Characterizing home device usage from wireless traffic time series. In *19th International Conference on Extending Database Technology (EDBT)*.

[42] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. 2017. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles* 12 (2017).

[43] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[44] Themis Palpanas. 2015. Data series management: The road to big sequence analytics. *ACM SIGMOD Record* 44, 2 (2015), 47–52.

[45] Themis Palpanas. 2016. Big sequence management: A glimpse of the past, the present, and the future. In *SOFSEM 2016: Theory and Practice of Computer Science: 42nd International Conference on Current Trends in Theory and Practice of Computer Science, Harrachov, Czech Republic, January 23-28, 2016, Proceedings 42*. Springer, 63–80.

[46] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2018. Paris: The next destination for fast data series indexing and query answering. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 791–800.

[47] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2020. Messi: In-memory data series indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 337–348.

[48] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2020. Paris+: Data series indexing on multi-core architectures. *IEEE Transactions on Knowledge and Data Engineering* 33, 5 (2020), 2151–2164.

[49] Botao Peng, Panagiota Fatourou, and Themis Palpanas. 2021. SING: Sequence Indexing Using GPUs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1883–1888.

[50] Jinglin Peng, Hongzhi Wang, Jianzhong Li, and Hong Gao. 2016. Set-based similarity search for time series. In *Proceedings of the 2016 International Conference on Management of Data*. 2039–2052.

[51] Davood Rafiei and Alberto Mendelzon. 1997. Similarity-based queries for time series data. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. 13–25.

[52] Thanawin Rakthanmanon, Bilson Campana, Abdullah Mueen, Gustavo Batista, Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn Keogh. 2012. Searching and mining trillions of time series subsequences under dynamic time warping. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 262–270.

[53] Usman Raza, Alessandro Camerra, Amy L Murphy, Themis Palpanas, and Gian Pietro Picco. 2015. Practical data prediction for real-world wireless sensor networks. *IEEE Transactions on Knowledge and Data Engineering* 27, 8 (2015), 2231–2244.

[54] Dennis Shasha. 1999. Tuning time series queries in finance: Case studies and recommendations. *IEEE Data Eng. Bull. 22*, 2 (1999), 40–46.

[55] Jin Shieh and Eamonn Keogh. 2008. i SAX: indexing and mining terabyte sized time series. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 623–631.

[56] Chanop Silpa-Anan and Richard Hartley. 2008. Optimised KD-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 1–8.

[57] Chang Wei Tan, Geoffrey I Webb, and François Petitjean. 2017. Indexing and classifying gigabytes of time series under time warping. In *Proceedings of the 2017 SIAM international conference on data mining*. SIAM, 282–290.

[58] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A McGrail, Peng Wang, et al. 2020. Apache iotdb: time-series database for internet of things. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2901–2904.

[59] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A data-adaptive and dynamic segmentation index for whole matching on time series. *Proceedings of the VLDB Endowment* 6, 10 (2013), 793–804.

[60] Ruidong Xue, Weiren Yu, and Hongxia Wang. 2022. An Indexable Time Series Dimensionality Reduction Method for Maximum Deviation Reduction and Similarity Search.. In *EDBT*. 2–183.

[61] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2017. Dpisax: Massively distributed partitioned isax. In *2017 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1135–1140.

[62] Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2018. Massively distributed time series indexing and querying. *IEEE Transactions on Knowledge and Data Engineering* 32, 1 (2018), 108–120.

[63] Liang Zhang, Noura Alghamdi, Mohamed Y Eltabakh, and Elke A Rundensteiner. 2019. TARDIS: Distributed indexing framework for big time series data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1202–1213.

[64] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for interactive exploration of big data series. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1555–1566.

[65] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2016. ADS: the adaptive data series index. *The VLDB Journal* 25 (2016), 843–866.

[66] Kostas Zoumpatianos, Yin Lou, Themis Palpanas, and Johannes Gehrke. 2015. Query workloads for data series indexes. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1603–1612.