TARDIS: Distributed Indexing Framework for Big Time Series Data (Technical Report)

Liang Zhang, Noura Alghamdi, Mohamed Y. Eltabakh, Elke A. Rundensteiner
Worcester Polytechnic Institute, Worcester, MA 01609, USA
(lzhang6, nalghamdi, meltabakh, rundenst)@wpi.edu

***Abstract*—The massive amounts of time series data continuously generated and collected by applications warrant the need for large scale distributed time series processing systems. Indexing plays a critical role in speeding up time series similarity queries on which various analytics and applications rely. However, the state-of-the-art indexing techniques, which are *iSAX-based* structures, do not scale well due to the small adopted fan-out (binary) that leads to a highly deep index tree, and the expensive search cost through many internal nodes. More seriously, the iSAX character-level cardinality adopted by these indices suffers from a poor maintenance of the proximity relationships among the time series objects, which leads to severe accuracy degradation for approximate similarity queries. In this paper, we propose the *TARDIS* distributed indexing framework to overcome the aforementioned limitations. *TARDIS* introduces a novel iSAX index tree that is based on a new word-level variable cardinality. The proposed index ensures compact structure, efficient search and comparison, and good preservation of the similarity relationships. *TARDIS* is suitable for indexing and querying billion-scale time series datasets. *TARDIS* is composed of one centralized global index and local distributed indices—one per each data partition across the cluster. *TARDIS* uses both the global and local indices to efficiently support exact match and *kNN approximate* queries. The system is implemented using Apache Spark, and extensive experiments are conducted on benchmark and real-world datasets. Evaluation results demonstrate that for over one *billion* time series dataset (TB scale), the construction of a clustered index is about 83% faster than the existing techniques. Moreover, the average response time of exact match queries is decreased by 50%, and the accuracy of the kNN approximate queries has increased more than 10 fold (from 3% to 40%) compared to the existing techniques.**

I. Introduction

Many emerging applications in science, manufacturing, and social domains generate time series data at an explosive speed. For example, the sensors on a Boeing787 produce around half a terabyte of time series data per flight [1]. As a result, the data mining techniques on these big time series data, e.g., similarity search, clustering, classification, motif discovery, outlier patterns, and segmentation have drawn a lot of recent interest [2], [3]. In particular, similarity search operations are of paramount importance since they form the basis of virtually all of the more complex operations mentioned above [4]. Since full scans on large-scale time series data are prohibitively expensive, indexing techniques become a critical backbone to make such similarity queries practical. Unfortunately, as it would be presented in this paper, the state-of-the-art indexing techniques over big time series data lack both the desired scalability and accuracy.

Since time series are inherently high-dimensional data, a common approach before indexing the data is to first apply a dimensionality reduction technique to extract key features, and then index these features instead. Many summarization and feature extraction techniques have been proposed including Discrete Fourier Transforms (DFT) [5], Discrete Wavelet Transforms (DWT) [6], Piecewise Aggregate Approximation (PAA) [7], [8], and Symbolic Aggregate approXimation (SAX) [9]. Typically, these representations are then indexed by Spatial Access Methods (SAMs) like the R-tree or its variants. However, SAMs are known to degrade quickly due to the curse of dimensionality. Indexable Symbolic Aggregate approXimation (iSAX) [10] is proposed as an index-friendly feature extraction technique. It first divides a time series into equal size segments, and then uses characters of variable cardinalities to represent the mean of these segments, which results in representing a given time series as a sequence of characters. The iSAX Binary Tree [11] is later proposed as a binary index structure of the iSAX representation.

Unfortunately, all implementations of the above techniques are designed for centralized systems and assume the dataset is small enough to fit in one machine. For this reason, it takes about 400 hours to build an index for one TB of data [11].

More recently, some distributed time series management systems have been proposed to transfer, store or analyze time series data (refer to a recent survey in [3]). These systems mainly leverage mathematical models and sampling to support approximate *Select-Project-Transform* (SPT) queries with or without error bounds. A more relevant system to our work that supports similarity search is the DPiSAX index [12]. However, the built index tree (we refer to it as "iBT") suffers from too many internal nodes and large depths of the leaf nodes due to the inherent binary fan-out. Moreover, since iBTs are based on the direct iSAX character-level cardinalities, the comparisons are shown to be very expensive, and the accuracy of the returned results of kNN approximate queries tend to be poor (below 10% in many cases).

In this paper, we propose a novel iSAX-based distributed indexing framework for big time series data, called *"TARDIS"*[1] for supporting *exact match* and *approximate kNN* queries. The index still adapts the iSAX representation but with a new *word-level variable cardinality* instead of the character-level cardinality. The word-level cardinalities enable better in-parallel processing, which suits the target distributed systems. In addition, we propose the iSAX-Transpose (*iSAX-T*) as a string-like signature to get rid of the costly conversions during the comparisons. On top of these signatures, we introduce a

[1]TARDIS is the Time Machine name introduced in Dr. Who TV series.

K-ary index tree (called *sigTree*) to overcome the limitations of the former binary trees. *sigTrees* enable compact index structure with fewer internal nodes and shorter paths to leaf nodes.

TARDIS uses the *sigTrees* to construct a single centralized global index based on statistics collected from the data. The global index acts as a skeleton (or partitioning scheme) to re-partitioned the time series data across the cluster machines to localize the similar objects together. Then, each partition is locally indexed−using *sigTrees* as well−for faster access within a given partition. To better support *exact match* queries, each local index is augmented with a partition-level Bloom Filter index, which is synchronously generated with the local index, to avoid many unnecessary accesses to the actual partition. Moreover, the combination of the word-level cardinality and the compactness of the *sigTrees* significantly enhance the accuracy of the kNN Approximate queries mainly because they preserve the proximity of the similar time series objects much better than the current techniques.

In summary, the contributions of this paper are as follow:

- Identifying the core limitations of the state-of-art indexing techniques in processing big time series data. And then, proposing *TARDIS*, a scalable distributed indexing framework to address these limitations. *TARDIS* consists of a centralized global index, and distributed local indices to facilitate efficient similarity query processing.
- Proposing a new *iSAX-T* signature scheme that dramatically reduces the cardinality conversion cost, and *sigTree* that constructs a compact index structure at the word-level similarity.
- Introducing efficient algorithms for answering the *exact match* and *kNN approximate* queries. We introduce different query processing strategies to greatly improve the accuracy of the approximate queries.
- Conducting extensive experiments on benchmark, synthetic, and real-world datasets to compare TARDIS with the state-of-the-art techniques. The results show significant improvement in index construction time ($\approx$ 8x speedup), and more critically, more than 10x accuracy improvement in some of the kNN approximate queries.

The rest of this paper is organized as follow. We review the background in Section 2. The new iSAX signature scheme and the index tree are defined in Section 3. TARDIS index construction is presented in Section 4, and the query processing algorithms are discussed in Section 5. The experimental evaluation is presented in Section 6. Finally, we review related work in Section 7, and present the conclusion remarks in Section 8.

II. Preliminaries

A. Key Concepts of Time Series

Definition 1: ***[Time Series Dataset]*** *A time series $X = \langle x_1, x_2, \cdots, x_n \rangle, x_i \in R$ is an ordered sequence of n real-valued variables. Without loss of generality, we assume that the readings arrive at fixed time granularities, and hence the timestamps are implicit and no need to store them. A time series dataset $DB = \{X_1, X_2, \cdots, X_m\}$ is a collection of m time series objects all of the same length n.*

Similarity queries consider each time series as an entire object rather than individual numerical fields because of the continuous nature of the data. Many similarity metrics have been proposed including *Dynamic Time Warping*(DTW) and *Euclidean Distance*(ED). The work in [13] shows that DTW degenerates to simple ED in large datasets because a close match is more likely to be found without warping. Therefore, targeting large scale datasets, we assume the ED similarity metric. In the following, we define the terms used throughout this paper.

Definition 2: ***[Euclidean Distance(ED)]*** *Given two time series $X = \langle x_1, x_2, \cdots, x_n \rangle$ and $Y = \langle y_1, y_2, \cdots, y_n \rangle$, their Euclidean distance is defined as:*

$$ED(X, Y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \quad (1)$$

Similar to exiting techniques, *TARDIS* supports two fundamental similarity queries, namely *exact match* and *kNN approximate* queries. The exact kNN queries tend to be very expensive and time consuming, and most applications, especially those working with big datasets, typically prefer faster responses even with some accuracy loss.

Definition 3: ***[Exact Match Query]*** *Given a query time series $Q = \langle q_1, q_2, \cdots, q_n \rangle$ of length n, and a time series dataset $DB = \{X_1, X_2, \cdots, X_m\}$, the exact match query finds the complete set $S = \{X_i \in DB\}$ such that $\forall X_i \in S, ED(X_i, Q) = 0$, and $\nexists Y_j \notin S | ED(Y_j, Q) = 0$.*

Definition 4: ***[kNN Approximate Query]*** *Given a query time series $Q = \langle q_1, q_2, \cdots, q_n \rangle$, a time series dataset $DB = \{X_1, X_2, \cdots, X_m\}$ and an integer k, the query finds the set $S = \{X_i \in DB\}$ such that $|S| = k$. The error ratio of S, which represents the approximation accuracy, is defined as $\frac{1}{k} \sum_{i=1}^{k} \frac{ED(X_i, Q) \ \forall X_i \in S}{ED(Y_i, Q) \ \forall Y_i \in T} \geq 1$, where $T = \{Y_1, Y_2, \cdots, Y_k\}$ is the ground truth kNN answer set.*

B. iSAX-Representation Overview

iSAX [10] is based on Piecewise Aggregate Approximation (PAA) [7] and Symbolic Aggregate approXimation (SAX) [9]. Figure 1 illustrates an example of how these techniques summarize a time series.

PAA(T,w): Given a time series, say T in Figure 1(a), PAA divides T into equal-length segments and represents each segment by the mean of its values. The number of segments is called *"word length"* (w), which is an input to the technique, and the entire representation vector is called a *"word"*. For example, the PAA of word length = 4 of T is PAA(T,4) = [-1.5, -0.4, 0.3, 1.5] as illustrated in Figure 1(b).

SAX(T,w,c): SAX takes the PAA representation of time series T as its input, and then discretizes it into characters or binary alphabet labels. This discretization is achieved by

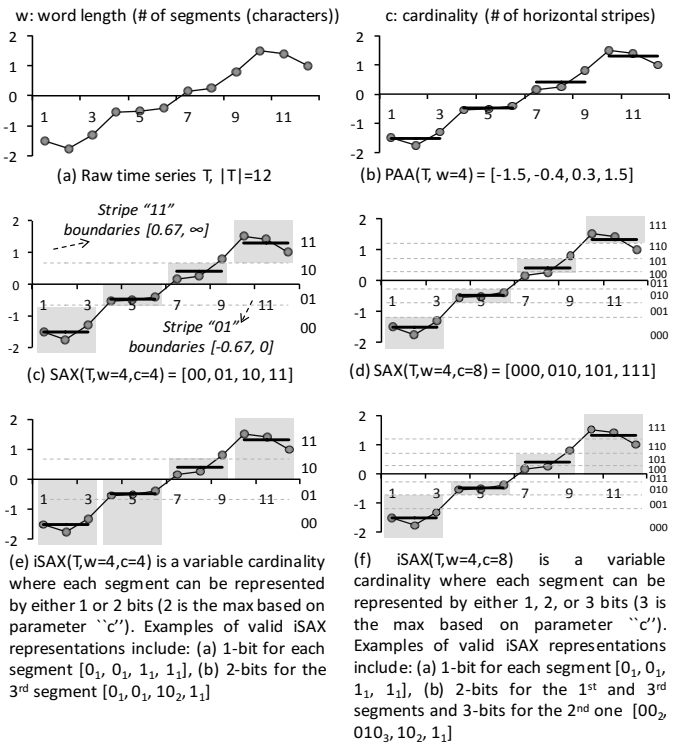


Fig. 1: PAA, SAX, and iSAX Representations.

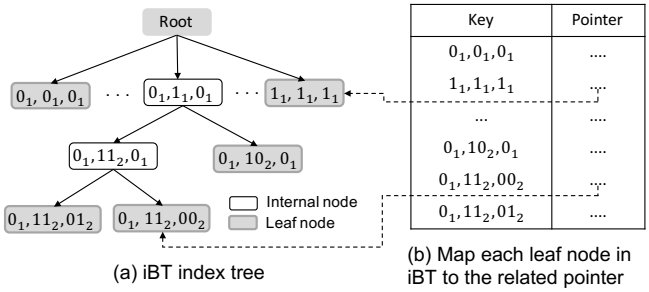


Fig. 2: The iBT Index and iSAX Map Table.

dividing the value space (the y-axis) into horizontal stripes. The number of the stripes equals an input parameter referred to as *"cardinality"* (c), which is typically a power of 2. For example, in Figures 1(c) and (d), the cardinality is set to 4 (2 bits) and 8 (3 bits), respectively. The authors in [9] proposed an algorithm to decide on the boundaries of each stripe. For example, in Figure 1(c), stripes "11" and "01" have the boundaries of [0.67, ∞], and [-0.67, 0], respectively. Then, each stripe is assigned a character label−which can be an arbitrary character or binary bits. Finally, each segment in the time series is assigned its corresponding stripe label. Figures 1(c) and (d) illustrate the $SAX(T, 4, 4)$ and $SAX(T, 4, 8)$ for the time series T presented in Figure 1(b). Two main observations to highlight for the SAX representation:

• *Fixed Cardinality*: A drawback of the SAX representation is that a time series representation is fixed, i.e., each segment is represented by the number of bits corresponding to the cardinality parameter. This means, for large datasets, a high cardinality must be used to increase the possibility of creating enough distinct representations among the time series objects.

• *Lower-Bound Distance*: A nice property of the SAX representation is that it guarantees for two time series T_1 and T_2, their Euclidean distance in the SAX domain (calculated based on the boundaries of the SAX stripes) is smaller than or equal to their true distance in the data space. That is: $ED(T_1.SAX, T_2.SAX) \leq ED(T_1, T_2)$. This property is effective in pruning many candidates during a similarly search query, e.g., range or kNN, only based on the SAX representation and without checking the raw time series values.

iSAX(T,w,c): iSAX maintains the nice lower-bound distance property of SAX. However, unlike SAX, iSAX uses *variable cardinality* for each segment in the time series. This is achieved by first enforcing the representation of the stripes' labels to be binary bits (not arbitrary characters). And then leveraging these bits to allow for variable cardinality for each segment. For example, Figure 1(e) illustrates the iSAX(T, w=4, c=4) for time series T. iSAX takes the same input as SAX, however, each segment can be independently represented by a number of bits up to the max number identified by parameter "c". The figure shows two possible representations for T. Figure 1(f) also illustrates two possibilities when the cardinality is set to 8 (3 bits). The decision of how many bits to use for a given segment is dynamically determined while indexing the time series data and building the iSAX Binary Tree index (iBT) overviewed next.

C. iSAX Binary Tree Index (iBT)

The iBT index [10] is an unbalanced binary tree index with the exception of the 1^{st} level (see Figure 2(a)). It starts with one root node, and a set of leaf nodes in its 1^{st} level using 1 bit representation for each segment, i.e., the number of nodes in the 1^{st} level is 2^w, where w is the word length. The time series objects are inserted one at a time to the corresponding leaf node based on is iSAX representation. Once the number of time series contained by a leaf node exceeds a threshold, which is an input parameter, the node switches to be an internal node and it splits into two child leaf nodes. The splitting is performed by increasing the cardinality of one of the segments, which means using more bits to represent this segment. This will probably lead to distributing the node's time series objects over the two child nodes. For example, the internal node $[0_1, 11_2, 0_1]$ in Figure 2(a) is divided into two leaf nodes $[0_1, 11_2, 01_2]$ and $[0_1, 11_2, 00_2]$ by extending the cardinality of the 3^{rd} segment (also called *character*) from 1 *bit* to 2 *bits*.

The Round-robin split policy initially proposed in [10] to determine the split character has shown to perform excessive and unnecessary subdivision. An optimized policy is proposed in [11] to pick the character having a high probability to equally split the leaf node. Ultimately, the cardinality increase over any segment cannot exceed the max cardinality c. The work in [11] also proposes a bulk-loading mechanism of time series data that first determines the shape of the iBT tree, and then routes each time series to its leaf node.

Limitations of iBT: Although it is an interesting structure and performs well for small datasets, iBT indices suffer from severe limitations under big datasets, which include:

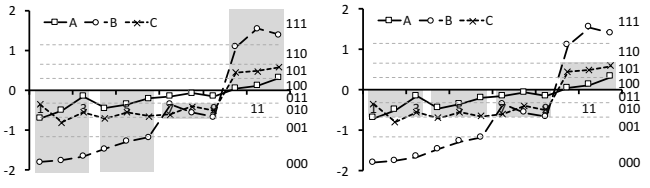
- *Loose structure and long traversal*: The superabundance of internal nodes caused by the binary fan-out results in deep height for many leaf nodes, and thus increases the tree height and its traversal at query time.
- *Large initial cardinality*: To guarantee leaf nodes to have the same granularity, the conversion from time series to iSAX needs to put aside *enough large initial* cardinality for the split mechanism due to the uncertainty of segment skewness and the amount of data. Hence, it results in unnecessary conversion and storage.
- *High matching overhead*: The alternative to solve the long tree paths is to convert the iBT to a map table [12] (see Figure 2(b)). The signature of each leaf node becomes a key in the table, and a pointer is maintained to either the tree node (if the tree is kept) or the actual partition holding the data. However, given a query time series Q, the search for matches within the map table is complex and very expensive due to the variable character cardinality in the keys. It requires creating all possible signatures from Q and then performing repetitive search in the map table, which is a clear bottleneck.
- *Weak proximity preservation and poor search accuracy*: iBT uses character-level variable cardinality to solve the segment skewness problem and construct the hierarchical tree. However, character-level matching is not efficient in preserving the proximity of the relatively similar objects, and these objects may end up in far away leaf nodes. This results in poor accuracy for approximate queries.

Example 1: Referring to Figure 3(a), assume a character-level variable cardinality of (1,1,3,1) bits. In this case, the time series A, B, C are represented as [$0_1, 0_1, \underline{011_3}, 1_1$*], [*$0_1, 0_1, \underline{010_3}, 1_1$*] and [*$0_1, 0_1, \underline{010_3}, 1_1$*], respectively. Under this representation, the closest series to "C" is "B" (their distance in the iSAX space is zero). However, it is clear, that the closest to "C" is "A".*

D. Distributed iSAX Time Series System

To the best of our knowledge, DPiSAX [12] is the only iSAX-based distributed system in literature to support index construction and kNN Approximate queries. It constructs a global index and local indices. It leverages the cluster machines to sample a subset of the time series and convert them into *iSAX signatures*. These signatures are then sent to the master node to construct the global index, which is a *partition table* instead of the *loose iBT structure*. For local indices construction, all time series are converted into iSAX signature with *a large initial cardinality* of size 512 to guarantee the split requirement. Then, for each iSAX signature (not the raw time series) a lookup over the partition table is performed (with *high matching overhead*) to re-partition the signatures. Finally, all workers concurrently build iBTs as local indices over their partitions.

Given a query Q, DPiSAX converts Q into its iSAX signature. It then matches the partition signature in the global index to identify the corresponding partition. Then, a worker loads this partition and traverses the local index to find leaf node(s) for post-processing. DPiSAX is an un-clustered index, i.e., the time series original data remain un-partitioned, the leaf nodes in local indices only store the iSAX signatures and the record id of the corresponding time series.



(a) Character-level: B and C are covered by $[0_1, 0_1, 010_3, 1_1]$

(b) Word-level: A and C are covered by $[01_2, 01_2, 01_2, 10_2]$

Fig. 3: Similarity of Time Series.

Limitations of DPiSAX:

- *Inheriting the limitations of iBT*: Although it achieves its *relative* scalability over the iBT indices by supporting distributed processing, DPiSAX is still based on the iBTs and inherits its limitations as highlighted above.
- *Additional degradation in result's accuracy*: To speedup the creation of the DPiSAX index, it builds an un-clustered index. However, answering queries based only on the iSAX representation without the final refine phase further degrades the accuracy of the results. On the other hand, retrieving the raw time series to apply the refine phase involves expensive random I/O operations across the cluster machines.

III. TARDIS Building-Block Structures

To address the aforementioned limitations, we propose a new iSAX-based signature scheme and its accompanied index tree to optimize the index construction and similarity queries over massive time series datasets. The frequently used notations in this paper are listed in Table I.

A. iSAX-Transposition Signature (iSAX-T)

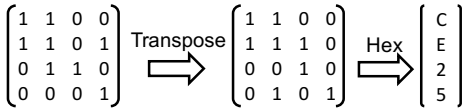
The objective of the indexable Symbolic Aggregate approXimation Transposition (iSAX-T) is to simplify the representation conversion from a higher cardinality, e.g., 5 bits, to a lower cardinality, e.g., 3 or 4 bits, which is a common operation during both index construction and query search. This guarantees the efficiency of the parallel process. Unlike iSAX, iSAX-T utilizes *word-level variable cardinality* defined as follows:

• *Word-Level Variable Cardinality*: In this representation scheme, all characters in one word, i.e., the characters across all segments in a time series, must use the same cardinality. This cardinality is decided by the level of the index tree in which the time series resides.

Example 2: Referring to Figure 3(b), assume the Time series A, B, C reside in a leaf node at the 2^{nd} *level of the index tree. In this case, all characters (segments) will use a 2 bit cardinality, and thus represented as [*$\underline{01_2}$,$\underline{01_2}$,01_2,$\underline{10_2}$*],*

Notation	Description
(ts,rid)	(A time series, its record id)
w	Word length
b	# of cardinality bits, i.e., cardinality = 2^b
pid	Partition id
isaxt(n)	iSAX-T signature with 2^n cardinality
freq(n)	Frequency of $isaxt(n)$
Tardis-G	TARDIS global index
Tardis-L	TARDIS local indices
G-MaxSize	Split threshold for *Tardis-G* leaf nodes
L-MaxSize	Split threshold for *Tardis-L* leaf nodes

TABLE I: Frequently Used Notations.



(a) SAX(T,4,16) $[1100_4, 1101_4, 0110_4, 0001_4]$ Matrix Transposition and Hexadecimal

						iSAX-T
SAX(T,4,2)	=	{1,	1,	0,	0 }	= C
SAX(T,4,4)	=	{11,	11,	01,	00 }	= CE
SAX(T,4,8)	=	{110,	110,	011,	000 }	= CE2
SAX(T,4,16)	=	{1100,	1101,	0110,	0001}	= CE25

(b) iSAX-T Signature for Different Cardinality

Fig. 4: iSAX-T Signature.

[$\underline{\underline{00_2}}$,$\underline{\underline{00_2}}$,$01_2$,$\underline{\underline{11_2}}$] and [$\underline{01_2}$,$\underline{01_2}$,$01_2$,$\underline{10_2}$], respectively. Compared to Example 1, the closest series to "C" is now "A". This is mainly because the word-level cardinality is intended to preserve the proximity of similar time series better than the character-level cardinality.

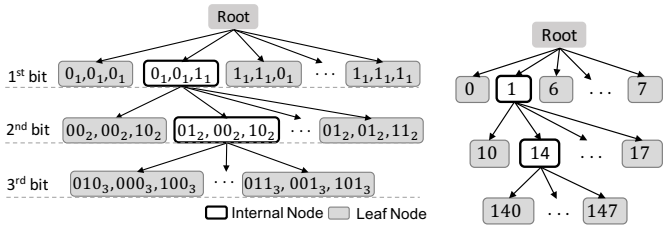
iSAX-T adapts a string-like signature based on matrix transposition to speedup the cardinality conversion operation. Thanks to the uniform word-level cardinality for a whole word, the binary signature can be considered as a binary matrix as presented in Figure 4(a). After transposing this matrix and transforming the binary into a hexadecimal, the signature is represented as a *string*.

As a consequence, the conversion is simplified as a string drop-Right operation. Equation 2 shows how to calculate the drop-Right letter number n. hc, lc and w represent the high cardinality, the low cardinality and the word length respectively (see Figure 4(b) as an example).

$$n = (log_2 hc - log_2 lc) * \frac{w}{4} \quad (2)$$

B. iSAX-T K-ary Tree (sigTree)

sigTrees are hierarchical K-ary trees based on the cardinality of the iSAX-T signature. Each node has no more than 2^w children (this bound results from increasing the cardinality representation by 1 bit over the w characters (segments) of the time series). For example, referring to Figure 5(a), the node with signature $[0_1, 0_1, 1_1]$ in the 1^{st} layer has been expanded to its children by adding an additional bit to each of the three characters, which results in having 8 children in the 2^{nd} layer. Three classes of nodes are involved in the *sigTrees*:



(a) Binary alphabet labels (b) String-like

Fig. 5: *sigTree* with fan-out = 2^3.

- **Root Node:** It represents the entire space and only contains the number of time series in the whole tree and pointers to its children nodes.
- **Internal Nodes:** They designate splits in the sub-tree space. When the quantity of time series contained in a leaf node exceeds a given *split threshold*, this leaf node gets promoted to an internal node and splits all its data entries into at most 2^w leaf nodes by increasing a 1 *bit* cardinality for all characters. Each internal node stores its iSAX-T signature, the number of time series in its sub-tree, and pointers to its children nodes.
- **Leaf Nodes:** They are the storage nodes at the bottom. They store the iSAX-T signatures and the number of time series they hold. Moreover, they store additional content that differs depending on the index type they belong to, i.e, global or local index, as will be explained later.

Figure 5 shows a *sigTree* with internal and leaf nodes represented by binary and compact string-like signatures. To insert a time series into the tree, we iteratively move down in the tree based on the iSAX-T signature until a leaf node is found. If a split is needed, it is performed as mentioned above. In addition, each node is able to reach all sibling nodes with the same cardinality from the parent node as we maintain the nodes double-linked (point to their parents as well as their children).

Example 3: Assume inserting a time series T = [$0110_4, 0011_4, 1011_4$] into the sigTree in Figure 5(b). First, T is converted into its iSAX-T signature "1473" according to Figure 4(b). Then, it starts in the tree from root node, drops 3 letters down to 1 bit cardinality to match the internal node "1" in the 1^{st} layer. This process repeats downward until finally traverse to the leaf node "147" in the 3^{rd} layer.

Benefits: The careful design of our representation of the iSAX-T and *sigTree* solutions offer the following benefits for massive time series processing in a distributed infrastructure:

- *Compact structure:* Compactness means fewer internal nodes and shorter depth of leaf nodes due to the large fan-out up to 2^w.
- *Small initial cardinality:* The short height can be achieved with a small cardinality which saves conversion costs and storage space.
- *Efficient signature conversion:* The conversion is simplified as a string drop-Right operation. Given the frequency of this operation during the index construction and query

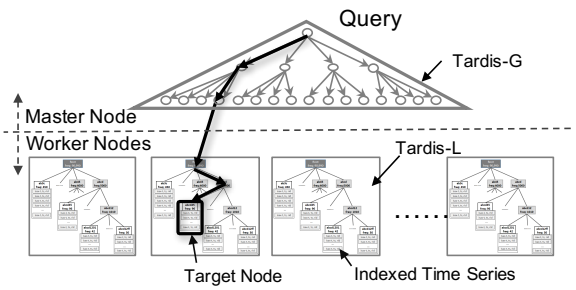


Fig. 6: TARDIS Key Components and Search Process.

processing phases, the cumulative time savings are considerable.

- *Word-level similarity:* iSAX-T effectively preserves the proximity relationship of similar time series due to the used word-level cardinalities.

IV. TARDIS Indexing Structure

Based on the *sigTree* structure, we now introduce the design of the TARDIS indexing framework. For ease of presentation, we start by giving an overview on the whole framework, and then the construction of the global and local indices in detail.

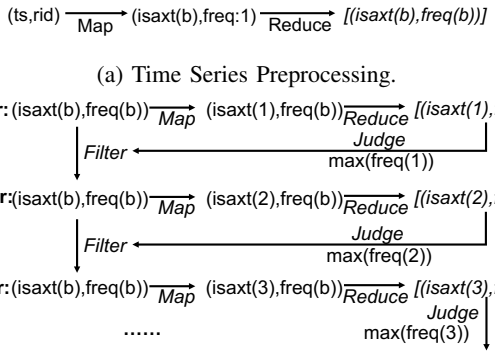
A. TARDIS Overview

TARDIS consists of two level indices as illustrated in Figure 6. *TARDIS Global Index* (Tardis-G) is a centralized global index maintained in the master node. It is used for efficiently identifying the target partition in a distributed system. The compactness of the index tree structure is controlled by a split threshold *G-MaxSize*. *TARDIS Local Index* (Tardis-L) is a distributed local structure to index the data entries within a single partition. Both structures are *sigTree*-based indices with a different content only at the leaf nodes. The leaf nodes of Tardis-G store the partition information, i.e., pointers to where they are located in the cluster, whereas the leaf nodes of local Tardis-L store the actual time series objects. The overall index framework is constructed given an initial cardinality, a word length, two split thresholds for Tardis-G and Tardis-L leaf nodes (these notations are summarized in Table I).

B. TARDIS Global Index (Tardis-G)

Tardis-G is a lightweight *sigTree* structure that resides in the master node of the cluster. It is the entry point for the index search. Unlike other iSAX-based indices which are constructed based on the representations of time series as they are loaded gradually, it is based on statistics collected from the cluster nodes in a distributed way. The construction consists of the following steps.

Data Preprocessing: The dataset is sampled at the *block level* and each time series is converted to a pair of values consisting of the *iSAX-T signature* and the frequency. This is performed by the worker nodes in parallel. A percentage of blocks are randomly chosen to reduce the disk access. The generation of pairs is completed by a single map-reduce job. All time series *(ts,rid)* are transformed to *(isaxt(b), freq:1)* in the map phase, and then aggregated to *[(isaxt(b), freq(b))]* in the reduce phase, where *b* represents the initial cardinality.



(a) Time Series Preprocessing.

(b) Node Statistics Collection.

Fig. 7: Tardis-G Workflow.

Node Statistic: The node statistics are collected for each layer in ascending order. The *[(isaxt(b), freq(b))]* generated previously is processed to retrieve *[(isaxt(i), freq(i))]* in which each entry is the information of a node in the i^{th} layer. *isaxt(i)* means the iSAX-T signature of a node and *freq(i)* means the frequency of this signature. In other words, the quantity of time series under this node. The procedure of the i^{th} layer involves the following operations: (1) *Map* that converts *(isaxt(b), freq(b))* to *(isaxt(i), freq(b))*; (2) *Reduce* that aggregates *(isaxt(i), freq(b))* to *[(isaxt(i), freq(i))]* which is the collected result for i^{th} layer; (3) *Judge* that decides to stop collection or not: if *max[freq(i)]* exceeds the *G-MaxSize*, filter out *(isaxt(b), freq(b))* contained by leaf nodes in i^{th} layer and continue $(i+1)^{th}$ layer, otherwise, stop to finish this step. Note that the entries in *[(isaxt(b), freq(b))]* are filtered out layer by layer, though the whole size is small. Finally, we get several groups of *[(isaxt(i), freq(i))]* with their respective layer id.

Skeleton Building: The index structure is constructed and the node information collected is put in the right places. All collected information is sent to the master. It is completed layer by layer in ascending order using a *tree insertion* mechanism. Root node is the entry point. Each inserted node recursively matches the internal node at each layer. Take node *(isaxt(3):"0202ff", freq(3):550)* in Figure 8 for example, it starts from the root node, finds the matched node *isaxt:"02"* in 1^{st} layer, and then *isaxt:"0202"* in 2^{nd} layer, finally reaches its position in 3^{rd} layer. During this process, we observe that *isaxt(3):"0202ff"* is converted for 2 times in this traversal path. The master node is not the bottleneck of this process due to the small size of tree. To facilitate retrieving siblings' information from the parent node, all nodes are doubly linked.

Partition Assignment: The goal is to package all under-utilized sibling leaf nodes into as few partitions as possible to facilitate parallel processing. Distributed infrastructures, like Hadoop and Spark [14], prefer to launch parallel tasks over large files rather than too many tasks over small files because the big data is processed in the unit of a partition or a block. Assembling sibling leaf nodes together has two benefits: (1) all records are similar at the parent node level;

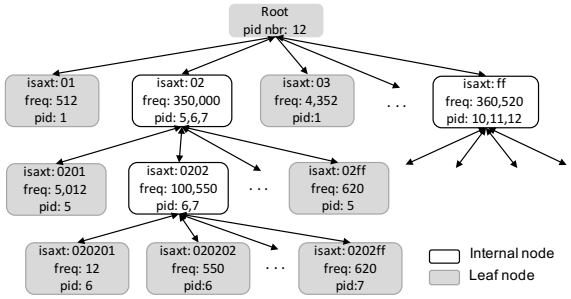


Fig. 8: Tardis-G Structure, the word length is 8 so the signature uses 2 letters to hex each bit cardinality.

(2) the partition is represented by the signature of the parent node to facilitate pruning the search space. In addition, sibling leaf nodes are indexed into finer granularity by the Tardis-L at each partition even though assembled together. Our problem can be considered as a *Partition Packing* problem.

Definition 5: ***[Leaf Partitions Packing]*** *Given a list of n leaf nodes under an internal (or root) node $L = \{l_1, l_2, \cdots, l_n\}$ and a partition capacity C, the* ***Partition Packing problem*** *is to group leaf nodes into as few partitions as possible, such that the sum of each partition is no greater than C.*

Since it is an NP-hard optimization problem [15], the exact algorithms typically leverage the branch-and-bound strategy which uses approximate algorithms to compute the bounds. We adopt *first fit decreasing* (FFD) [15] which is the best known approximate algorithm with time complexity $O(nlogn)$ and *worst-case performance ratio* $3/2$, to solve this problem. It sorts all leaf nodes by the record number in descending order, and then inserts each leaf node into the first partition with sufficient remaining space. After finishing packing, the partition ids in the descendant nodes are synchronized to the *id list* of ancestor nodes to facilitate future information retrieval of sibling nodes.

C. TARDIS Local Index (Tardis-L)

Tardis-L is an *sigTree*-based structure that indexes data entries within each partition. TARDIS leverages the high I/O rate and powerful in-memory computation of distributed infrastructures to construct local structures for all partitions in parallel. The data pipeline in Figure 9 shows the overall procedure. It involves the following steps:

Data Shuffle: Each record is shuffled to the target partition based on the assigned partition id. This step is finished in one map-reduce job by workers. Before starting the job, the master broadcasts the Tardis-G to all workers as the partitioners for the reduce operation. It resides in the memory of the workers until the job is done. In the map phase, each time series *(ts, rid)* in Table I is read and converted to *(isaxt(b), ts, rid)*. In the reduce phase, they are aggregated into matched partitions in two sub-steps: (3.1) each time series obtains the partition id by traversing the partitioner; and (3.2) shuffle each to the target partition using the distributed infrastructure. The data entries within partitions are out of order after repartitioning.

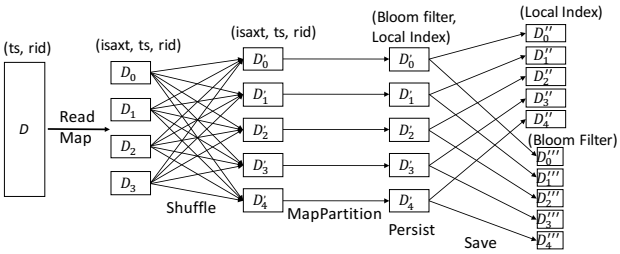


Fig. 9: Pipeline of Tardis-L Construction.

Local Structure Construction: The Tardis-L is constructed within each partition to organize data entries. This step is implemented in *mapPartition* operation in Figure 9 using the *tree insertion* mechanism: each data entry *(isaxt(b), ts, rid)* enters at the root node of local index, and then traverses to the matched leaf node. If the leaf node contains more data entries than the given split threshold, it is promoted into an internal node and split all entries to $\leq 2^w$ leaf nodes. Meanwhile, each node in the traversal path increases the quantity of records by one. Note that both Tardis-G and Tardis-L employ a similar insertion mechanism to construct the *sigTree* but three key differences exist: (1) *Scope:* Tardis-G is a complete dataset whereas Tardis-L corresponds to one partition; (2) *Element:* Tardis-G inserts the information of nodes whereas Tardis-L inserts time series entries; (3) *Split:* Tardis-G finishes nodes split in statistics collection phrase whereas Tardis-L splits nodes in the index construction phase.

Bloom Filter Index Construction: A small size local index is built for the Exact-Match query. Bloom Filter [16] is a space efficient probabilistic data structure to test whether an element is a member of a set. It can raise false positive but not false negative. The iSAX-T signature is used as the input for Bloom Filter because: (1) it is already contained by each data entry so no extra conversion is needed; (2) a given initial cardinality and a high fan-out guarantee a high probability of *one-to-one* relationship between each time series and its signature. The Bloom Filter index is synchronously generated with the Tardis-G in *mapPartition* operation in Figure 9: when each data entry is inserted during Tardis-G construction, $isaxt(b)$ is encoded into bloom filter data structure at the same time.

V. TARDIS Query Processing

TARDIS supports classical Exact-Match queries and kNN-Approximate queries. While for some problems, finding the exact match item is essential, for many data mining applications of massive time series [10] an approximate query processing may be all that is required. Take big data visualization [17] for example, Tableau [18] takes 71 *mins* to plot a scatterplot for a dataset with 2 *billion* tuples. In contrast, only 3 *seconds* are taken to produce practically the same visualization by carefully choosing 1 *million* tuples. In this spirit, TARDIS supports very fast approximate searches, as only single disk access is required.

The intuition behind processing both query types in a single I/O is that the iSAX-T signature of two similar time series

Algorithm 1 Exact-Match Algorithm

Input: qts
Output: $exists\ or\ not$

• Traverse Tardis-G to identify the partition in the master
$isaxt = convertToiSaxt(qts)$
$pid = tardisG.fetchPid(isaxt)$
$nonExist = BloomFilter.querySax(pid, isaxt)$
• True if the BF guarantee non-existence
if $!nonExist$ **then**
    $partition = readHdfsBlock(pid)$
    $records = partition.leafNode(isaxt)$
    for $r \leftarrow records$ **do**
        if $r.ts == qts$ **then**
            return $true$
        end if
    end for
end if
return $false$

Algorithm 2 kNN Approximate: Multi-Partitions Access

Input: qts, k, pth
Output: $topK\ list(dist, rid)$

• Traverse Tardis-G to identify the partition in the master
$isaxt = convertToiSaxt(qts)$
$pid = tardisG.fetchPid(isaxt)$
$pidList = tardisG.fetchFromParent(isaxt)$
if $size(pidList) > pth$ **then**
    $pidList = randomSelect(pidList, pth)$
end if
• Load all partitions by workers
$partitions = spark.readHdfsBlock(pidList)$
• Get the threshold from the partition by one worker
$partition = partitions.select(pid)$
$node = partitions.fetchKnnNode(isax, k)$
$records = node.fetchRecords().calEuSort(qts)$
$th = records.take(k).last.distance$
• Scan partitions using the threshold in parallel
$candidates = partitions.scan(th).calEuSort(qts)$
return $candidates.take(k)$

would be either the same or very similar. Thus they are close to each other in the *sigTree*. Given such assumption, the query result is obtained by attempting to find the matched internal or matched leaf node. Since the hierarchical structure of *sigTree* has no overlap, such node should be identified if it exists. Besides the standard process described above, optimization strategies are also proposed to improve efficiency. kNN-Exact and Range queries can also be supported by scanning the global and Local indices from top to bottom to obtain the expected results. Since the implementation of these two queries is straightforward, they are omitted for brevity. However, the index scanning method is also adopted in our kNN-Approximate optimized query processing.

A. Exact Match Query

The *Exact-Match Algorithm* harnesses the TARDIS index framework to fetch leaf nodes for validation. It is composed of the following steps: (1) convert the query time series to its iSAX-T signature; (2) traverse the Tardis-G to identify the partition; (3) test existence of such signature in the Bloom Filter index of the partition; if the test result is false, this query terminates with zero results; (4) if the Bloom Filter search is positive, then load the partition and traverse the Tardis-L to retrieve the leaf node, and then lookup the query time series. The failure of traversal in either Tardis-G or Tardis-L means a non-existent result.

The algorithm leverages Bloom Filter Index to prevent the high-latency disk access in the case non-existence with low false positive in the above 3^{rd} step. As we know, the distributed infrastructures prefer to store data in large files, for example, the default block size used by Hadoop and Spark is $64M$ or $128M$. Thus the loading of such file is high latency. For the Exact-Match query, the query time series either exists or doesn't exist in the dataset. In the first case, the access to a partition is unavoidable. In the second case, however, it may not be necessary. The algorithm uses the Bloom Filter Index to test if the partition contains the query time series or not. Due to the small size, it resides in memory or is read from disk with low latency. Furthermore, we also provide *Exact-Match Algorithm Non-Bloom Filter* which takes more time with the same query accuracy because one partition has to been loaded if the partition is identified in the above 2^{nd} step.

B. kNN Approximate Query

The *Target Node Access* leverages Tardis-G and Tardis-L to fetch *Target node* which is the leaf or internal node with more data entries than *k* at the lowest position of Tardis-L. Note that if it is an internal node, any child node should contains less data entries than *k*. The process is composed of the following steps: (1) convert the query time series to its iSAX-T signature; (2) traverse the Tardis-G to identify the partition; (3) load the partition and traverse the Tardis-L to the *target node*; (4) fetch all candidates under this node and take the *k* closest records as the result.

Besides the *Target Node Access* algorithm, two optimized algorithms are proposed based on the intuition that the candidate scope can be extended by reducing the word-level cardinality of iSAX signatures to loosen the bounds. Because of the approximation of iSAX-based representation, the larger the candidates scope is, the more accurate the result is. *One Partition Access* alogrithm scans the Tardis-L of the loaded partition to extend the scope whereas *Multi-Partitions Access* algorithm harnesses the parallel processing power of the distributed infrastructure to concurrently exploit sibling partitions to extend it. Both methods use the low bound feature of iSAX-T to prune the search space. PAA is used to obtain a tighter bound since the query time series is provided.

Given the PAA representation T_{paa} of query time series T and iSAX-T signature N_{isaxt} of node N with the same word length w. We use T_i to represent T_{paa} value at the i_{th} character and N_{iL} and N_{iU} to represent the lower and upper breakpoints of cardinality at the i_{th} character, $N_{iL} < N_{iU}$. The low bound

LB is defined as:

$$LB(T_{paa}, N_{isaxt}) = \sqrt{\frac{n}{w}} \sqrt{\sum_{i=1}^{w} LB_i} \quad (3)$$

$$LB_i = \begin{cases} (N_{iL} - T_i)^2 & if \quad N_{iL} > T_i \\ (N_{iU} - T_i)^2 & if \quad N_{iU} < T_i \\ 0 & otherwise \end{cases} \quad (4)$$

One Partition Access uses the distance of the k^{th} time series obtained at the 4^{th} step of Target Node Access as the *threshold* to prune the search space of Tardis-L from top to bottom. It collects all candidates in the residual nodes and takes the first *k* closest records as the result. Unlike Target Node Access and One Partition Access, Multi-Partitions Access fetches the *partition list* of all sibling partitions in the parent node at the 2^{nd} step. This list in the upper layers of Tardis-G may be large. For example, the list size corresponds to the total of all partitions in Figure 8 because the parent node of leaf node *isaxt(1):"ff"* is the root node. In response, a partition threshold pth is set to control the maximum quantity of partitions loaded. If the list size exceeds it, pth elements are randomly chosen in the list. After loading all these partitions, Multi-Partitions Access uses the pruning method above to process all partitions in parallel. It collects all candidates in the residual nodes and takes the first *k* closest records as the result. Algorithm 2 shows the detailed strategy.

VI. Experiment

We first introduce the implementation and the experimental setup, and then empirically evaluate the performance of the index construction, the query processing.

A. Implementation & Setup Details

Implementation. Since the core features of TARDIS are infrastructure-independent, they are applicable to big data engines generally. As a proof of concept, TARDIS prototype has been realized on top of the Apache Spark 2.0.2 [14]. We opt for Spark due to its efficient main memory caching of intermediate data and the flexibility it offers for caching hot data. An important design choice of TARDIS is not to touch the internal of the core spark engine so to be portable. This allows easy migration of TARDIS into a new version of Spark released in the future. We implement our approach for both clustered and un-clustered indices at the local structure. For clustered index, the data entry of Local Structure is *(isaxt(b),ts,rid)*. In contrast, the un-clustered index keeps *(isaxt(b),rid)*. Both adopt *isaxt(b)* as the search key. All data structures and algorithms presented about TARDIS are built from scratch in Scala and we extend DPiSAX [12] to support clustered index, Exact-Match query and kNN-Approximate query as the baseline of evaluation.

Cluster Setup. All experiments were conducted on a cluster consisting of 2 nodes. Each node consists of 56 Intel@Xeon E5-2690 processors, 500GB RAM, 7TB SATA hard drive and is connected to a Gigabyte Ethernet switch and runs Ubuntu 16.04.3 LTS with Spark-2.0.2 and Hadoop-2.7.3. The Spark cluster is deployed in standalone mode. and 64GB RAM and 6 cores are reserved for the Spark Driver Program.

Parameters	Value
HDFS block size	128 *M*
Word length	8
Sampling percentage	10%
L-MaxSize	1,000
Initial cardinality (TARDIS)	64
Initial cardinality (Baseline)	512
Multi-Partition Access threshold: pth	40

TABLE II: Experimental Configuration.

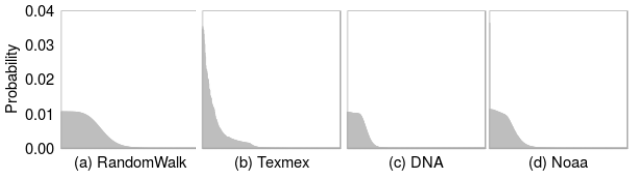
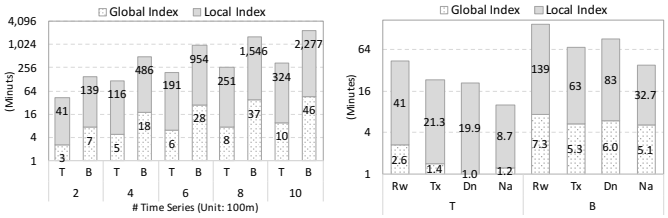


Fig. 10: Datasets Distribution.



(a) RandomWalk Dataset (b) Dataset Comparison (200m)

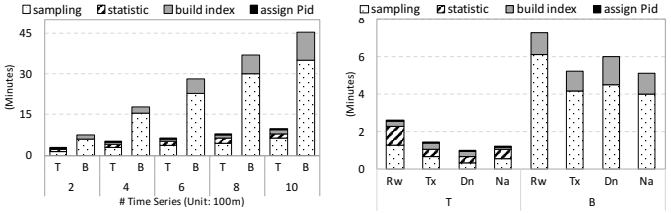
Fig. 11: Index Construction Time. (T: TARDIS, B: Baseline, Rw: RandomWalk, Tx: Texmex, Dn: DNA, Na: Noaa)

Datasets. We use one benchmark and three real-world datasets from different domains for the evaluation. ***RandomWalk Benchmark Dataset*** is extensively used as the benchmark for time series index in other projects [9]–[11], [13]. This dataset is generated for 1 *billion* time series with 256 points. ***Texmex Corpus Dataset*** [19] is an image dataset which contains 1 *billion* SIFT feature vectors of 128 points each. ***DNA Dataset*** [20] contains assembly of the human genome collected from 2000 to 2013. Each DNA string is divided into subsequences of length 192 and then converted into time series [11]. It contains 200 *million* time series with 192 points. ***Noaa Dataset*** [21] involves weather and climate data from global 20,000 stations from 1901 to present. The temperature feature is extracted into 200 *million* time series with 64 points. Each dataset is z-normalized before being indexed. The datasets are chosen to cover a wide range of skewness with respect to the values' occurrence frequencies as illustrated in Figure 10. Each point has a floating point precision of 4 *byte*.

As shown in Table II, TARDIS and the baseline systems adopt the same configuration except the initial cardinality. It is 64 for TARDIS, whereas the default value of Baseline is 512. For reproducibility, all source code, cluster configuration and technical report are provided [22].

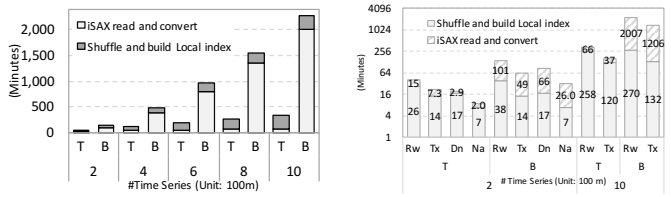
B. Index Construction

1) Clustered Index: The capacity of a HDFS block is set as the Tardis-G threshold *G-MaxSize* in terms of the indexed



(a) RandomWalk Dataset (b) Dataset Comparison (200m)

Fig. 12: Global Index Construction Time Breakdown. (T: TARDIS, B: Baseline, Rw: RandomWalk, Tx: Texmex, Dn: DNA, Na: Noaa)



(a) Local Index(RandomWalk) (b) Local Index comparison

Fig. 13: Local Index Construction Time Breakdown.

time series Take 1 *billion* time series RandomWalk dataset as an example, it needs about $10,189$ partitions if each partition $110,000$ data entries. As shown in Figure 11(a), TARDIS takes 334 *mins* to finish the index construction process for 1 *billion* dataset whereas the baseline takes $2,323$ *mins*. From 200 *million* to 1 *billion* in RandomWalk dataset, the index construction time of TARDIS increases 7.6 *times* as that of baseline is 16 *times*. Figure 11(b) shows the performance in all datasets and the difference between different datasets are caused by the time series length and value distribution. Our new system demonstrates excellent scalability for the large partition number because the *sigTree* structure of Tardis-G has a short height for leaf nodes and iSAX-T signature simplifies the cardinality conversion to identify the partition for shuffling operation whereas the partition table derived from the iBT for the global index introduces high look-up costs and iSAX signature needs expensive cardinality conversion.

For the global index, TARDIS takes 10 *mins* for 1 *billion* data whereas the baseline takes about 46 *mins* in Figure 11(a). TARDIS leverages Block-Level sampling to reduce data reading time in sampling steps, and harnesses powerful workers to collect node statistics. Figure 12(a) shows that TARDIS finishes *node statistic*, *build index tree* and *partition assignment* in a few minutes even for large datasets. It shows good scalability to construct global index quickly for different dataset scales. Note that the master node isnot the bottleneck even if the *index tree construction* and *partition assignment* are completed by itself. In contrast, the time of building *index tree* taken by the baseline increases linearly as dataset size increases. Figures 12(b) shows the global index construction time in all datasets.

For the local index in Figures 11, the difference of two systems is the *read and convert data* caused by the partition id assignment completed by partitioner, since both systems

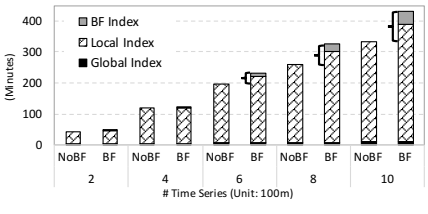
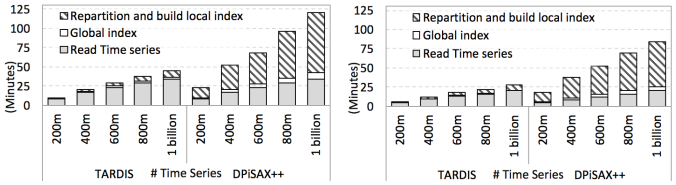


Fig. 14: Bloom Filter Index Construction (RandomWalk)



(a) RandomWalk Dataset (b) Texmex Dataset

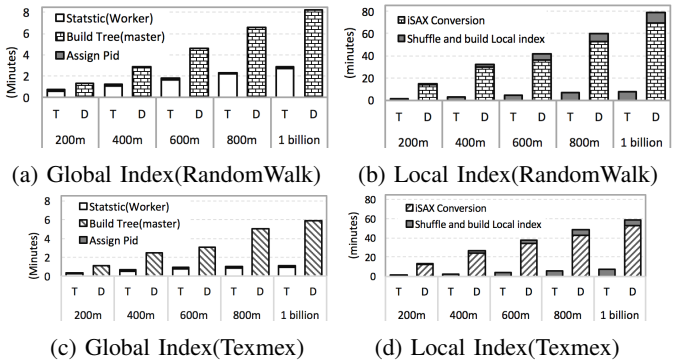
Fig. 15: Un-clustered Index Construction Time

spend the same time on reading data. Each record in the baseline takes the character selection into consideration in the lookup of partition table. The table matching process and cardinality conversion causes this procedure to be time-consuming. In contrast, TARDIS leverages the Tardis-G and iSAX-T signature to finish this process within a short time. TARDIS takes 66 *mins* to *read and convert data* for 1 *billion* dataset, whereas the baseline takes 2007 *mins*. Note that the time includes the fixed overhead of reading the dataset. In our technical report [22], we also include the breakdown construction time for the local indices.

For the Bloom Filter Index construction, the persistence of data in memory impacts the performance. If the intermediate data is persisted in memory, no obvious overhead exists because the cost only corresponds to dumping this small size index, $66k$ for each partition, to disk. As shown in Figure 14, when the RandomWalk dataset is less than 400 *million*, the difference is negligibly small. In contrast, if it could not be persisted in memory totally, the intermediate data has to be persisted in memory and on disk. Taking 1 *billion* dataset for example, the construction takes extra 97 *mins* in which 57 *mins* are spent on dumping the intermediate result and 40 *mins* is to read them.

2) Un-clustered Index: Since the un-clustered index repartitions only the *rid*, an integer, the partition quantity is fixed due to the small indexed file. As shown in Figure 15(a), TARDIS takes 44 *mins* to finish the whole process whereas the baseline takes 120 *mins* for 1 *billion* Random Walk dataset.[2] They are 28 *mins* and 87 *mins* respectively for 1 *billion* Texmex dataset in Figure 15(b). Although the construction time in a different scale from 200 *million* to 1 *billion* is linear growth, the baseline increases 2.8 times as TARDIS in Random Walk Dataset and 3.1 for Texmex Dataset. For the global index of Random Walk dataset, TARDIS takes 3 *mins*, mostly spent on *node statistic* by workers, whereas the baseline takes 8 *mins*. Note that the *index tree construction* and *partition id assignment* are finished

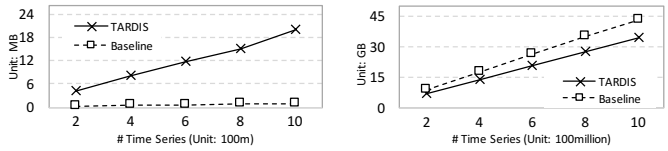
[2]The RAM size and CPU number of the Cluster in DPiSAX [12] is almost two times as that of our cluster.



(a) Global Index(RandomWalk) (b) Local Index(RandomWalk)

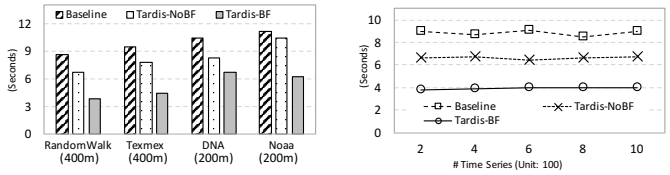
(c) Global Index(Texmex) (d) Local Index(Texmex)

Fig. 16: Un-clustered Index Construction Time Breakdown, *T: TARDIS, D: Baseline*



(a) Global Index (b) Local Index

Fig. 17: Index Size (RandomWalk)



(a) Dataset (b) Dataset Size (RandomWalk)

Fig. 18: Exact Match Average Query Time.

by TARDIS in seconds. For local index in Figure 16(b) and 16(d), the baseline spends extra 80 *mins* to match data to the partition table whereas TARDIS takes a negligibly small time.

3) Index Size: The Global Index is impacted by the index structure and dataset size while the local index is impacted also by the setting for indexed data like initial cardinality. For the global index, TARDIS keeps the whole *sigTree* structure as the index, whereas the baseline saves all leaf nodes as the partition table. For 1 *billion* time series in Figure 17(a), TARDIS uses 20 *M* while the baseline uses 1 *M*. However, the Tardis-G is still lightweight to be broadcasted to worker nodes and resides in RAM. In consideration of the improved efficiency of index construction and query processing, the trade-off of increased index size is reasonable. For the local index which excludes indexed data in Figure 17(b), the difference is mainly caused by the different initial cardinality. Since *sigTree* leverages a large fan-out to reduce the depth of leaf nodes, TARDIS uses a small value, 64 here, while the baseline uses a large initial value, 512 by default, to guarantee enough cardinalities for spliting. For 1 *billion* RandomWalk time series, TARDIS uses 34.9 *G* while the baseline uses 43.5 *G*.

C. Query Performance

1) Exact-Match Query Processing Performance: We evalute the query performance in all datasets. Each experiment involves 100 time series queries, with the exactly same length as the time series in the dataset. In particular, 50% are randomly selected from the dataset while the other 50% are guaranteed to be not exist in the dataset. The average query time is measured because the recall rates are all 100%. Although all queries need to identify the partition in the master and figure out the existence in workers, the key difference is caused by the operations in workers because the time taken by the master node is negligibly small compared with operations in the workers.

As shown in Figure 18, Tardis-NoBF takes fewer time than the baseline because of the shorter depth of leaf nodes and fewer records stored in leaf nodes in the Tardis-L, though both need to read one partition. In Figure 18(a), Noaa takes more time because more time series are stored in each partition due to the short length. In RandomWalk dataset, Tardis-BF has the best performance with 4 *sec*, which is about half of the baseline 9 *sec*, because the non-existence time series query avoid loading partitions. Therefore, the Bloom Filter Index effectively prevents the disk access for such queries. In Figure 18(b), the scale of the dataset has no obvious impact on the performance since each query only accesses one partition.

2) kNN-Approximate Query Processing Performance: The ground truth is critical for evaluation. However, the naive method, which calculates the distance between query time series and each record in the dataset to obtain the top k nearest neighbors, is impractical due to the prohibitive time cost. We instead leverage TARDIS to quickly figure out the ground truth: for each q_i in $Q = \{q_1, \cdots, q_p\}$, use the low bound feature of Tardis-G to filter out "large" partitions and then use this feature of Tardis-L to filter out nodes in the residual partitions; if the candidates number within the residual nodes equals or exceeds k, we take the top k nearest neighbors as the ground truth for q_i. Note that a threshold (7.5 in our paper) is given for above filtering processes in advance.

In contrast with DPiSAX [12] that considers the query answering time for *10-NN*, we study the effect of *query processes*, *k value* and *dataset size* to evaluate the search quality and search speed. Search quality is measured by *recall* and *error ratio* that are standard metrics in high dimension nearest neighbor query [23], [24]. Search speed is measured by *average query time*. Given a query q, the set of exact K nearest neighbors is $G(q) = \{g_1, \cdots, g_k\}$ and the query result is $R(q) = \{r_1, \cdots, r_k\}$. *recall* is defined as:

$$recall = \frac{|G(q) \cap R(q)|}{|G(q)|} \tag{5}$$

Obviously, the *recall* score is less than 100%. In the ideal case, 100% means all the k nearest neighbors are returned. *error ratio* is defined as:

$$error\ ratio = \frac{1}{k}\sum_{j=1}^{k}\frac{ED(q, r_j)}{ED(q, g_j)} \tag{6}$$

It measures how close the distance of the K nearest neighbors

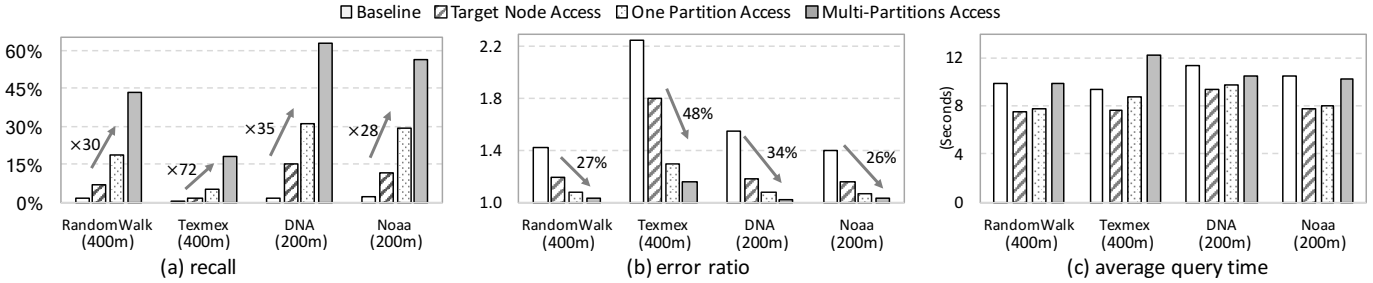


Fig. 19: kNN Approximate Performance in Different Datasets (k: 500).

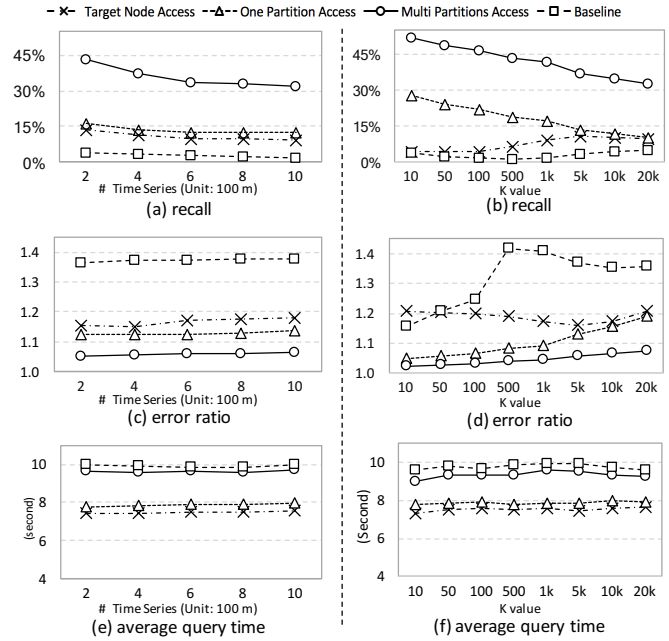


Fig. 20: Impact to kNN-Approximate Performance (RandomWalk), left: different dataset size for k value: 5000; right: different k for 400 *million*

found are compared to that of the exact K nearest neighbors. The value is larger than 1.0 and the idea case is 1.0.

As shown in Figure 19 and 20, TARDIS fetches a better performance in both the *recall* and the *error ratio* than the baseline. It is credited to TARDIS's word-level similarity and the extended candidates scope. At first, we study the effect of different query processes using the case of 400 *million* dataset and *500 k* value in Figure 19. For the *recall*, the baseline is 1.5% while Target Node Access is about 6.7%, One Partition Access is 18.9% and Multi-Partitions Access is 43.4%. For the *error ratio*, the baseline is 1.42 and Target Node Access is about 1.19, One Partition Access is 1.07 and Multi-Partitions Access is 1.03. For *average query time*, the baseline takes about 9.8 *sec* while Target Node Access takes about 7.5 *sec*, One Partition Access is 7.7 *sec* and Multi-Partitions Access is 9.9 *sec*. One Partition Access has better performance compared with Target Node Access because scanning the partition results in a larger candidates scope. Note that Multi-Partitions Access has obvious advantage because the extended candidate scope derives from sibling partitions. Even more partitions are loaded, the *average query time* of Multi-Partitions Access is similar to that of the baseline because of concurrently process.

As shown in Figure 20(left), the performance under different dataset sizes follows the same pattern aforementioned, particularly the *error ratio* and the *average query time*. The *recall* of Multi-Partitions Access decrease faster than others and the *error ratio* is the same as others. This tendency results from that the distance of records in one partition tends to be small but the ground truth disperse over more partitions. The *error ratio* increases and the *recall* decreases because the large dataset size leads to a dispersedness of the ground truth. The key point is that any iSAX-based signature is an approximate rather than exact representation. The *average query time* does not change significantly because the partition number loaded for all query processes has no change.

In Figure 20 (right), we evaluate the effect of different k values on the benchemark dataset. The performance is affected greatly by the candidate scope determined by the granularity of *target node*. The lowest level for *target node* is leaf node. For the same *L-MaxSize* 1000, however, the average leaf node size of TARDIS is 32 whereas the baseline is 634 caused by the different fan-out. The *recall* for the baseline has no obvious change. The slight increase of Target Node Access means that the target node of TARDIS more effectively holds similar time series because of the word-level similarity and both One Partition Access and Multi-Partitions Access decrease due to the dispersedness of the ground truth for larger k value.

Note that Multi-Partitions Access keeps the best accuracy even if k value varies. For the *error ratio*, the turning point of the baseline in Figure 20(d) is caused by the promotion of *target node* position from a leaf node to an internal node in the iBT structure. When k is less than 500, the *target node* is a leaf node with the large candidate scope 634. One Partition Access is the up bound of *recall* and the low bound of *error ratio* for Target Node Access, because the candidate scope of One Partition Access becomes similar to that of Target Node Access as the k increases but One Partition Access is the best case of Target Node Access. The average query time does not change significantly because the partition number loaded for all query processes stay consistent for different k values.

D. Impact of Sampling

Since the sampling percentage impacts the estimation of iSAX-T representation distribution which determines the construction and index quality. The construction quality is mea-

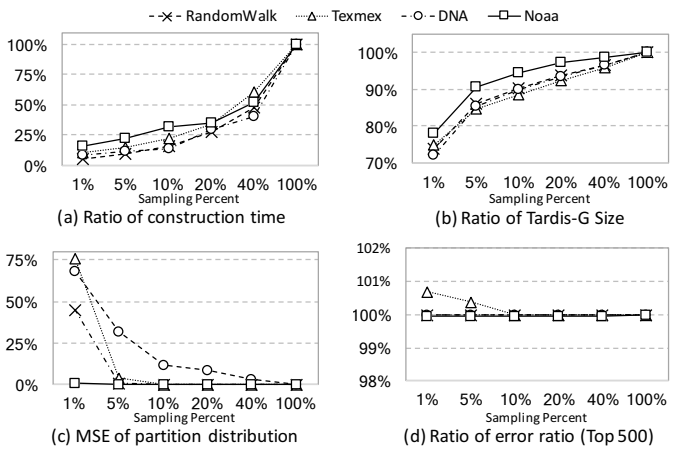


Fig. 21: Impact of Sampling Percentage

sured by *construction time* and *global index size* while the index quality is measured by *error ratio* which evaluates the cohesiveness of partition and *MSE of partition size distribution* which evaluates partition size distribution estimation. Like histogram method, the last metric gets the probability distribution of partition sizes, with the 15 *megabyte* bucket interval, and then calculates the Mean squared error (MSE).

We compare 1%, 5%, 10%, 20%, 40% percentage with the 100% case. In Figure 21(a), the sampling method greatly decreases the global index construction time in all datasets. In Figure 21(b), the smaller the percentage is used, the smaller the index is generated because the sampling method get part of the representations. However, the Tardis-G generated from 1% data is able to support the shuffle operation of the whole dataset. In Figure 21(c), the small percent has large MSE value and the 10% has the similar effect as the 100% in all datasets. We run Top-500 kNN-Approximate query to get the *error ratio* of Multi-Partitions Access. In Figure 21(d), the small percentage leads to high ratio of error ratio.

VII. Related Work

Recent advances in sensor, network and storage technologies have sped up the process of generating and collecting time series. Similarity queries, the fundamental problem of time series data mining, relies on the summarization and indexing of massive datasets. The literature on these topics is vast; see paper [2] and references therein for useful surveys and empirical comparisons. The iSAX-based indexing family, such as iSAX [10], iSAX2.0 [11] and Adaptive Data Series Index (ADS) [25], demonstrates good scalability for bulk-loading mechanism in centralized machine. The round bin split policy [10] and the statistic-base split policy [11] are proposed for the binary split. While both iSAX and iSAX2.0 build indices over the dataset up-front and leverage these indices for query process, ADS [25] shifts the costly index creation steps from the initialization time to the query processing time. It interactively and adaptively builds parts of the index only for the subsets of data on which the users pose queries. All aforementioned methods are based on a centralized machine.

The authors [26] propose a distributed system which constructs vertical inverted tables and horizontal segment trees based on the PAA summarization of time series data. However, the work in [26] cannot handle our target scale of billions of time series objects (they only experimented with 1K, 10K, and 100K objects). Moreover, it is explicitly stated in [26] for large $k > 50$, their kNN query performance degrades quickly and converges to the brute force search. in contract, TARDIS is designed for scalable k as well, e.g., k in thousands.

Several other recent distributed systems have been proposed for managing different aspects of time series data. For example, Druid [27] and Gorilla [28] focus only on the storage and compression aspects in a distributed environment. In contrast, SciDB [29] focus on distributed linear algebra and statistical operations on time series data, and BTrDB [30] addresses primitive operations on big time series data such as selection, projection, and simple aggregations. All of these systems operate at the record-level, e.g., they support insertion, deletion, and update of time series records. TARDIS is fundamentally different from these systems as it a batch oriented and designed for other complex operations such as kNN queries.

VIII. Conclusions

In this paper, we propose TARDIS, a scalable distributed indexing framework to index and query massive time series. We introduce *sigTree* and iSAX-T signature to simplify and speed up the compact index construction in distributed environments. Particularly, the word-level similarity feature of *sigTree* effectively keeps better similarity of time series. Additionally, optimized similarity query processes leverage this flexible framework to improve the performance. Our experiments over the synthetic and real world datasets validate that our new approach dramatically reduces the index construction time and substantially improves the query accuracy.

References

[1] J. Ronkainen and A. Iivari, "Designing a data management pipeline for pervasive sensor communication systems," *PCS*, 2015.
[2] P. Esling and C. Agon, "Time-series data mining," *CSUR*, vol. 45, 2012.
[3] S. K. Jensen, T. B. Pedersen, and C. Thomsen, "Time series management systems: A survey," *TKDE*, vol. 29, no. 11, pp. 2581–2600, 2017.
[4] T. Palpanas, "The parallel and distributed future of data series mining," in *HPCS*. IEEE, 2017, pp. 916–920.
[5] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, *Fast subsequence match in time-series databases*. ACM, 1994, vol. 23.
[6] K.-P. Chan and A. W.-C. Fu, "Efficient time series matching by wavelets," in *ICDE*. IEEE, 1999, pp. 126–133.
[7] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra, "Dimensionality reduction for fast similarity search in large time series databases," *KAIS*, vol. 3, pp. 263–286, 2001.
[8] K. Chakrabarti, E. Keogh, S. Mehrotra, and M. Pazzani, "Locally adaptive dimensionality reduction for indexing large time series databases," *TODS*, vol. 27, no. 2, pp. 188–228, 2002.
[9] J. Lin, E. Keogh, L. Wei, and S. Lonardi, "Experiencing sax: a novel symbolic representation of time series," *DMKD*, vol. 15, 2007.
[10] J. Shieh and E. Keogh, "isax: indexing and mining terabyte sized time series," in *SIGKDD*. ACM, 2008, pp. 623–631.
[11] A. Camerra, T. Palpanas, J. Shieh, and E. Keogh, "isax 2.0: Indexing and mining one billion time series," in *ICDM*. IEEE, 2010, pp. 58–67.
[12] D.-E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas, "Dpisax: Massively distributed partitioned isax," in *ICDM*, 2017, pp. 1–6.
[13] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh, "Querying and mining of time series data: experimental comparison of representations and distance measures," *PVLDB*, vol. 1, 2008.

[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, 2010.
[15] M. Delorme, M. Iori, and S. Martello, "Bin packing and cutting stock problems: Mathematical models and exact algorithms," *EJOR*, 2016.
[16] B. H. Bloom, "Space time trade-offs in hash coding with allowable errors," *Commun ACM*, vol. 13, pp. 422–426, 1970.
[17] Y. Park, M. Cafarella, and B. Mozafari, "Visualization-aware sampling for very large databases," in *ICDE*. IEEE, 2016, pp. 755–766.
[18] Tableau. [Online]. Available: http://www.tableau.com
[19] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *TPAMI*, vol. 33, pp. 117–128, 2011.
[20] U. G. Institute. [Online]. Available: https://genome.ucsc.edu/
[21] NCEI. [Online]. Available: https://www.ncdc.noaa.gov/cdo-web/datasets
[22] Tardis report. [Online]. Available: https://github.com/lzhang6/TARDIS
[23] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing." VLDB, 1999, pp. 518–529.
[24] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: efficient indexing for high-dimensional similarity search." VLDB, 2007, pp. 950–961.
[25] K. Zoumpatianos, S. Idreos, and T. Palpanas, "Ads: the adaptive data series index," *VLDB Journal*, vol. 25, pp. 843–866, 2016.
[26] X. Wang, Z. Fang, P. Wang, R. Zhu, and W. Wang, "A distributed multi-level composite index for knn processing on long time series," in *DASFAA*. Springer, 2017.
[27] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *SIGMOD*. ACM, 2014.
[28] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," *PVLDB*, pp. 1816–1827, 2015.
[29] M. Stonebraker, P. Brown, D. Zhang, and J. Becla, "Scidb: A database management system for applications with complex analytics," *CiSE*, pp. 54–62, 2013.
[30] M. P. Andersen and D. E. Culler, "Btrdb: Optimizing storage system design for timeseries processing." in *FAST*, 2016, pp. 39–52.