WPI-CS-TR-19-05 October 2019

Chainlink: Indexing Big Time Series Data For Long Subsequence Matching

by

Noura Alghamdi
Liang Zhang
Huayi Zhang
Elke A. Rundensteiner
Mohamed Y. Eltabakh

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Chainlink: Indexing Big Time Series Data For Long Subsequence Matching. (Technical Report Version)*

Noura Alghamdi, Liang Zhang, Huayi Zhang, Elke A. Rundensteiner, Mohamed Y. Eltabakh
Worcester Polytechnic Institute, Worcester, MA 01609, USA
(nalghamdi, lzhang6, hzhang4, rundenst, meltabakh)@wpi.edu

***Abstract*—Scalable subsequence matching is critical for supporting analytics on big time series from mining, prediction to hypothesis testing. However, state-of-the-art subsequence matching techniques do not scale well to TB-scale data sets. Not only does index construction become prohibitively expensive, but also the query response time deteriorates quickly as the length of the query subsequence exceeds several 100s of data points. Although Locality Sensitive Hashing (LSH) has emerged as a promising solution for indexing long time series, it relies on expensive hash functions that perform multiple passes over the data and thus is impractical for big time series. In this work, we propose a lightweight distributed indexing framework, called *Chainlink*, that supports approximate kNN queries over TB-scale time series data. As a foundation of Chainlink, we design a novel hashing technique, called *Single Pass Signature* (*SPS*), that successfully tackles the above problem. In particular, we prove theoretically and demonstrate experimentally that the similarity proximity of the indexed subsequences is preserved by our proposed single-pass SPS scheme. Leveraging this SPS innovation, Chainlink then adopts a three-step approach for scalable index building: (1) in-place data re-organization within each partition to enable efficient record-level random access to all subsequences, (2) parallel building of hash-based local indices on top of the re-organized data using our SPS scheme for efficient search within each partition, and (3) efficient aggregation of the local indices to construct a centralized yet highly compact global index for effective pruning of irrelevant partitions during query processing. Chainlink achieves the above three steps in one single map-reduce process. Our experimental evaluation shows that Chainlink indices are compact at less than 2% of dataset size while state-of-the-art index sizes tend to be almost the same size as the dataset. Better still, Chainlink is up to 2 orders of magnitude faster in its index construction time compared to state-of-the-art techniques, while improving both the final query response time by up to 10 fold and the result accuracy by 15%.**

I. Introduction

A. Background and Motivation

Time series data are ubiquitous and pervasive across almost all human endeavors. Over the last decade, the explosion of new technologies ranging from wearable sensors to social networks has lead to an unprecedented growth of time series data. For example, in Zhejiang Province of China, 20 million smart meters have been deployed, producing around 20 million time series per year each of length around 0.5 million readings [1]. As a consequence, scalable solutions for processing, querying and mining long time series that leverage modern distributed compute infrastructures become a necessity. Most time series mining algorithms from query-by-content, anomaly detection, classification to segmentation rely on subsequence similarity search, e.g., kNN subsequence search, as a core subroutine. Examples from real-world applications are highlighted below.

Motivating Example 1 (Neuroscience Applications): *Given a massive archive of Electroencephalography (EEG) data, a neurologist may need to search for a certain number (k) of epileptic spikes most similar to a given patient's epileptic spike to decide whether or not to classify him as an epilepsy patient. The value k needs to be large to avoid the influence of noise on the result, while still being appropriate for subsequent human analysis. In this use case, the length of an epileptic spike, here equal to the length of the matching query sequence, typically reaches over 7000 data points [2].*

Motivating Example 2 (Smart Grid Applications): *In smart grid applications, if a smart meter were to collect one meter reading every minute, it would collect more than 0.5 million meter readings per year for each of its sensors [1]. A business analyst may want to learn about comparative customer electricity consumption behaviours by answering queries such as: 'find k households whose electricity consumption starting at any period of time are approximately similar to a certain given household Q". Since we aim to identify similar behavior, approximate match results suffice.*

B. Common Problem Characteristics and Technical Challenges

The examples highlighted above, among many others, share common characteristics that drive our research. First, the kNN subsequence search is the core building block operation for these queries. In fact, in the context of time series applications−and high dimensional data in general−approximate kNN tends to be the norm compared to the exact kNN match [3], [4]. This is because: (1) Approximate processing tend to be highly accurate and satisfactory for most applications, and (2) This relaxation opens up opportunities for substantial improvements in query response time and preprocessing overheads involved in the developed solution. Second, time series data sets can be in the order of 100s of millions of time series (TBs of data). At such scale, the data

* This version can be cited as WPI Dept of Computer Science Technical Report number *WPI-CS-TR-19-05*, released October 2019.

sets are stored and processed in distributed systems such as Hadoop or Spark. Third, query subsequences in modern time series applications tend to be very long, and in some cases reaching thousands of readings [1]. Therefore, the focus of our work is on *supporting approximate kNN similarity search over long (thousands-scale) time series subsequences in a distributed environment*, which has not yet been satisfactorily solved in the literature.

The challenges in solving this problem include:

(1) *High Dimensionality*: Long subsequences are high dimensional objects. Traditional indexing techniques such as R-trees and their variations are known to perform poorly in retrieval of such data [5]. Other state-of-the-art non-indexing techniques, such as prune-and-bound [2], also suffer from a degraded performance for such long subsequences because the later lead to loose bounds and thus ineffective pruning as confirmed in our experimental section (Section V-C).

(2) *Significant Overlap*: The enumeration of all possible subsequences for indexing is prohibitively expensive in both processing time and space as we will show experimentally in (Section V-C) for existing technique that perform such enumeration [6]. That is due to the significant overlap among consecutive subsequences.

(3) *Random Access*: In subsequence search, there is no clear criteria for re-partitioning and clustering the input time series data to create clustered index based on a similarity function. This is because two time series objects can be very similar with respect to one subsequence, while at the same time, very distinct with respect to another subsequence. Therefore, subsequence queries are more likely to require record-level (i.e. object-level) random access to retrieve time series objects from different partitions. Nevertheless, modern distributed infrastructures, such as Hadoop and Spark, are designed to support full scans over the data partitions, which are inefficient and unfortunately involve unnecessarily high overhead.

(4) *Speed vs. Accuracy Trade-off*: Typically, in approximate matching, an algorithm's speed and accuracy are contradictory objectives. Unfortunately, with the big scale of modern time series datasets, small blocks in an algorithm can magnify and become a critical scalability bottleneck, and that is exactly what we have observed in the state-of-art technique [7]. Therefore, a key objective of our work is to design a strategy that achieves high accuracy results for processing kNN matching queries while making the algorithm bottleneck free.

C. Limitations of the State-of-the-Art

Existing distributed techniques fail to address one or more of the aforementioned challenges, which hinders their scalability. For example, a recent distributed indexing solution called *KV-Match* [6], builds an index over all possible subsequences. Our experimental study demonstrates that building such an index over a few TBs of data takes days, while requiring storage in the scale of the original data (refer to 2^{nd} and 4^{th} challenges above). In contrast, the distributed indexing solution in [1] does not solve the subsequence matching in its general form. Instead, it supports a restricted type of queries that involve prefix matching, e.g., users input the precise offset where the search should starts in each time series as query parameter.

Further, these techniques [1], [6] use HBase as their underlying storage layer for time series data because HBase internally supports record-level random access (see 3^{rd} challenge above). Unfortunately, the subsequence index itself is then stored as an Hbase table regardless of its structure, e.g., inverted table or tree structure. This design decision leads to severe disadvantages, including the limited query capability of HBase (limited API) that requires a huge sequential access across the Hbase table to retrieve the whole index structure to query, need to having to maintain copies of the data also in other systems such as Hadoop or Spark, and additional latency due to extra communication between Hbase, HDFS and these systems [8].

In contrast to distributed systems, UCR Suite [2] is the state-of-the-art centralized technique for time series subsequence search. UCR is a prune-and-bound technique that does not preprocess the data set into an index. However, when we adopt and adapt this UCR strategy into its distributed UCR variant for processing datasets distributed across HDFS consisting of millions of time series, the performance suffers notably due to in part having to reset the search bounds frequently for each time series. Moreover, as our experimental study in Section IV shows, when the query length grows, the bounds become too loose to produce effective pruning (See 1^{st} challenge above).

D. Proposed Approach: Chainlink

In this paper, we propose a distributed indexing framework over time series data called *"Chainlink"*. ChainLink adopts the *duality-based* approach [9] for subsequence generation, which avoids the enumeration of all possible subsequences in the dataset and hence achieves a compact index size. Given our target of *approximate kNN search*, we utilize the popular *Locality Sensitive Hashing* (LSH) [10], [11], [12] as the base of our index. One key property of LSH is its ability to preserve the similarity among high-dimensional objects upon hashing with a high probability [10], [11], [12]. Unfortunately, the state-of-art techniques in LSH for indexing time series data, e.g., the *SSH* algorithm [7], suffer from scalability limitations as they inherently rely on very expensive multi-pass hash functions.

To overcome this scalability limitation, we propose a novel hashing technique, called *Single Pass Signature* (SPS). SPS achieves around $200x$ speedup compared to state-of-the-art techniques. In addition, our theoretical analysis of the SPS scheme guarantees that the similarity proximity between subsequences is preserved after hashing without sacrificing the accuracy for kNN query results.

To enable efficient search, we design ChainLink as a two-layer distributed index composed of a centralized global index (CL-Global) to direct the search to specific partition(s), and multiple distributed local indices spread across the worker nodes. Within each partition, the time series data are first re-organized locally and then the compact local ChainLink index (CL-Local) is built.

The key contributions of this paper include:

- We address the four core technical challenges highlighted in Section I.B by the design of a cohesive framework, namely ChainLink. To address the 1^{st} challenge, we utilize hash-based approach containing our novel SPS that reduces the dimensionality while preserving the similarity of the hashed subsequences. For the 2^{nd} challenge, ChainLink leverages the duality-based method to generate disjoint subsequences and thus minimize the index size.
- We design a two-layered distributed index structure that leverages a partition-level data re-organization to achieve fast search and efficient random access operations (the 3^{rd} challenge).
- We propose a novel hashing technique called *Single Pass Signature* (SPS) that hashes the subsequences in a single pass achieving $\approx$ 200 speedups compared to the standard technique adopted in existing system [7] while maintaining excellent result accuracy, thus tackling the 4^{th} challenge.
- We conduct an extensive experimental study on benchmark data sets. The results show significant improvement in index construction time (up to two orders of magnitude speedup), index size compactness (the local index size is less than 2% of dataset size while the global index size is only few MBs for TB-Scale dataset), and query response time (which is 10 fold faster than state-of-art technique [2]).

The rest of the paper is organized as follows. Section II introduces preliminaries for our work. We then describe the ChainLink framework and its innovations in Section III. Section IV introduces the ChainLink query processing strategy, while Section V presents our experimental results. Finally, we discuss related work in Section VI and conclude in Section VII.

II. Preliminaries

A. Key Concepts of Time Series

Definition 1. [Time Series Dataset] A time series dataset $D = \{X_1, X_2, \cdots, X_M\}$ is a collection of M time series objects X_i, each with an arbitrary length denoted as $|X_i|$.

Definition 2. [Time Series] A time series $X = \langle x_1, x_2, \cdots, x_m \rangle$, $x_i \in R$ where $1 \leq i \leq m$ is an ordered sequence of m real-valued variables. Without lost of generality, we assume that the readings arrive at fixed time granularities, and hence timestamps are implicit. For simplicity, we thus do not store timestamps.

Definition 3. [Subsequence of Time Series] A subsequence $X_i^{(j,l)}$ of object X_i is a time series of length l starting at position j in X_i, namely, $X_i^{(j,l)} =< x_1^{sub}, x_2^{sub}, ..., x_l^{sub} >$ such that $x_{1+u}^{sub} = x_{j+u}$ where $u = [0, l-1]$.

Definition 4. [Euclidean Distance (ED)] Given two subsequences of equal length l, $X = \langle x_1, x_2, \cdots, x_l \rangle$ and $Y = \langle y_1, y_2, \cdots, y_l \rangle$, Euclidean Distance (ED) is defined as:

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

Definition 5. [kNN Approximate Query] Given a query subsequence $Q = \langle q_1, q_2, \cdots, q_l \rangle$, a time series dataset $D = \{X_1, X_2, \cdots, X_M\}$ and an integer k, the query returns a set R of k subsequences such that $R = \{X_i^{(j,l)} \in D, \ \forall\, i, j\}$ with $|R| = k$ subject to the approximation accuracy measured by the error ratio $(err) = \frac{1}{k}\sum_{i=1}^{k} \frac{ED(X_i^{(j,l)}, Q) \ \forall X_i \in R}{ED(Y_i, Q) \ \forall Y_i \in T}$, where $T = \{Y_1, Y_2, \cdots, Y_k\}$ corresponds to the exact kNN answer set of Q on D.

The *error ratio (err)* in Def. 5, a standard metric used in the LSH-related literature [11], [13], measures the distance between the returned approximate knn set and the exact knn set. The smaller the error err, the closer the approximation and with err=1 denoting the exact solution.

B. Background on LSH

With *locality sensitive hashing (LSH) [10]* a core component of our proposed indexing technique, we overview the main idea of LSH below. LSH is a widely adopted technique for searching nearest neighbors in high-dimensional spaces. LSH provides a high probability guarantee that it will return the correct answer or a very close one [10], [11], [12]. For each object in the dataset D, LSH performs a dimensionality reduction operation to extract random features among the high-dimensional features, and then hashes each object based on these extracted features. The key principle here is that the hashing step is *repeated multiple times*, with at each time, different random features being selected and each object being hashed to a different hash table. The intuition is that similar objects are highly probable to collide and thus to go into the same bucket in at least one of the hash tables.

Weighted Minwise Hashing (WMH). Several techniques of the LSH family have been proposed in the literature to handle different similarity functions. They differ in the random function that extracts the features and the manipulation of the features before hashing. A popular LSH variant, Weighted Minwise Hashing (WMH) [14], [15], [16] estimates the *Jaccard similarity* on weighted sets. Weighted sets means that the algorithm not only keeps track of the features (exist or not), but also their weights, i.e., the frequency of their presence. The definition of the Jaccard similarity is given next in Def. 6.

Definition 6. [Jaccard similarity (JS)] Given two weighted sets S_1 and S_2, each a vector in the D-dimensional space of integer values, i.e., $S_1 =< s_{11}, ..., s_{1i}, s_{1D} >, S_2 =< s_{21}, ..., s_{2i}, s_{2D} >$ such that $s_{1i}, s_{2i} \in \mathbb{Z}$, the JS is defined as:

$$JS(S_1, S_2) = \frac{\sum_{i=1}^{D} min(s_{1i}, s_{2i})}{\sum_{i=1}^{D} max(s_{1i}, s_{2i})} \tag{2}$$

WMH generates randomized hash values such that the probability of a hash collision of a pair of sets S_1 and S_2 is given by Eq. 3 [14]:

$$Pr[WMH(S_1) = WMH(S_2)] = JS(S_1, S_2). \tag{3}$$

Equation 3 is an important property of *WMH* because it enables us to compare the hashing signatures of two weighted sets and estimate their Jaccard similarity without the need to actually retrieve the raw time series data [12].

To generate a *weighted minhash signature* for a weighted set S_i, it is passed g times to apply g randomly picked hash functions on it, where g is a fixed integer parameter resulting in a signature (sig) of length g.

Consistent Weighted Sampling (CWS) [15] is a popular technique adopted for hashing weighted sets. *CWS* samples from some well-tailored distribution to avoid replication. This scheme computes the exact distribution of minwise sampling with time complexity $O(d)$, where d corresponds to the number of non-zeros. However, this computation has to be performed g times, and thus the total complexity becomes $O(d \times g)$. As pointed out in [14] and confirmed by our experiments (Section V-B), this operation, which most sophisticated, can still be quite expensive resulting in a major bottleneck, especially when the data sets are big and g is large [14].

Observation. To overcome these scalability limitations of CWS, a new hashing technique has been proposed [14]. It offers remarkable speedup over CWS but only under the constraint that the weighted set vectors are relatively dense, i.e., most entries are non-zeros. However, unfortunately, the time series data we work with does not meet this assumption. That is, although in itself it is not sparse, meaning there are no necessarily missing values, the weighted set vectors generated (as will be explained later in III-B, Step 3) are typically very sparse (See our exp. in Section V-A). This can be explained by the fact that time series data do not involve drastic fluctuations nor variations across its values – in contrast to other dataset types such as text or image data (more details in III-B, Step 4).

For this reason, this state-of-the-art technique [14] is not effective in our context.

III. Our proposed ChainLink Index

A. ChainLink Overview

We now introduce ChainLink, a distributed indexing framework for time series data sets. The overall work flow of ChainLink is illustrated in Figure 1. ChainLink builds a local index (CL-Local) respectively over all time series stored in each partition managed by a cluster machine and then aggregates these local indices to construct a global index (CL-Global).

The data are stored in distributed files that can be either disk-based, e.g., Hadoop HDFS files, or memory-based, e.g., Spark RDDs. In either case, each file is divided into several partitions stored across cluster machines. Since our system is implemented using Spark, in the rest of the paper, and without loss of generality, we use the Spark RDD terminology. Each RDD partition consists of a set of time series objects {X$_1$, X$_2$, ...}, where each X$_i$ is as defined in Def. 2. Each X$_i$ is represented within the partition as a single *record*.

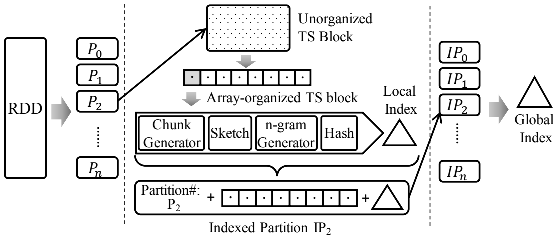


Fig. 1. Two-level Indexing Strategy in ChainLink.

B. ChainLink Local Indices (CL-Local)

The CL-Local building process is composed of four steps: 1) Record Organization: the raw time series objects are reorganized within each partition into an array structure to enable efficient random access. 2) Chunk Generation: each time series is divided into non-overlapping equal length chunks. 3) Chunk Feature Extraction: each chunk is transformed into a low-dimensionality feature vector by sketching the chunk followed by generating n-grams of the sketches along with their frequencies as weights to form a weighted set. 4) Hashing: our proposed SPS scheme is applied on the generated weighted set to hash each into a lower-dimension signature. The intuition of this step is that given two weighted set vectors S^i and S^j corresponding to base chunks X^i and Y^j, the similarity of S^i and S^j should capture the similarity of their underlying chunks. However, calculating set similarity is excessively expensive especially for massive scale data sets (refer to Def. 6). For this reason, we instead propose a new hashing scheme that maps S^i and S^j into lower-dimension signatures which preserve the proximity of the weighted sets. Lastly, we hash those signatures into hash tables which collectively serve as the local index. Finally, the local index is stored along with the arranged array time series data in the same partition.

Step 1 (Record Organization): The first challenge in indexing time series objects is that RDDs are primarily designed for sequential scans. Thus random access to a specific subsequence within a partition tends to be very expensive. Worse yet, a subsequence query may match with few time series objects in many distinct partitions. Thus, we need to design an efficient record look-up mechanism within each partition to avoid unnecessary full scans, where each record corresponds to a time series object. To achieve this, we arrange the time series records within every partition into an array structure as illustrated in Figure 1. A time series in partition p and array slot t gets assigned a new physical id $pid = (p, t)$ that is used in all subsequent computations during the index construction. These ids are subsequently used to locate a particular time series efficiently within each partition in *O(1)* cost.

Step 2 (Chunk Generation): *Duality-based* [9] approach has been explored in literature for chunk generation. It divides time series objects into disjoint chunks using a *disjoint jumping window* mechanism, while the query sequence is divided into overlapping chunks using a *one-step sliding window*.

In ChainLink, we aim to keep the size of the index compact. We thus adopt the duality-based approach as it significantly reduces the number of the generated chunks at data level during

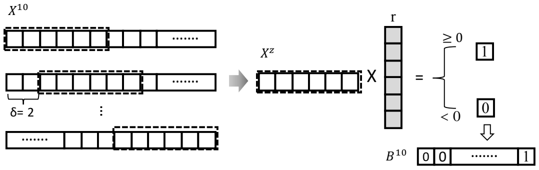


Fig. 2. Generating a sketch B^{10} capturing the pattern of chunk X^{10}.

index construction. Therefore, for a given time series object X with physical id (p, t), we generate its disjoint chunks of length w, each with a unique id (p, t, i), where i denotes the chunk number ($0 \leq i \leq (|X|/w - 1)$). Since local indices are at the granularity of a single partition p, the value p is implicit and need not be physically stored within the local index. In the rest of the paper, we refer to the i^{th} chunk of X by X^i.

As studied in [9], the determination of the window size w is based on the query workload and the minimum query length parameter that is to be supported. Typically, if the minimum length of a query sequence is $minLen$, then the maximum window size should be at most $\lfloor (minLen+1)/2 \rfloor$. The authors in [9] provide a theoretical analysis that if these bounds are honored, then it is guaranteed no patterns in the time series will be missed under this slicing scheme. ChainLink inherits these guarantees under these same assumptions.

Step 3 (Chunk Feature Extraction): In this step, each chunk is transformed into a low-dimensionality feature vector. This vector will later form the weighted set vector on which SPS is applied. Similar to the technique proposed in [7], the feature extraction in ChainLink consists of two procedures, namely *sketching* and *n-gram* generation. We opt for these two procedures since they bring the following benefits. First, they reduce the dimensionality of the possibly long chunks. Second, they convert the continuous domain of the time series readings into a discrete domain on which hashing can be applied. Third, they capture the trends in each chunk using the fine-grained n-gram elements such that the more similar two chunks are the more n-grams they are likely to share.

Sketching: The sketching procedure converts each chunk of continuous values into a sequence of discrete binary values, which also capture the overall trend of the chunk [17].

Given a chunk X^i, a random vector r of length $|r|$, where each component of r is selected from a normal distribution $N(0, 1)$ and a sketching step size δ, the extracted sketch B^i then corresponds to : $B^i = (b_1, b_2, \ldots, b_{|B^i|})$, where $|B^i| = \frac{|X^i| - |r|}{\delta}$. Let b_z be the z^{th} component of B^i, where $1 \leq z \leq |B^i|$, b_z is calculated as follows :

$$b_z = \begin{cases} 1 & \text{if} \quad r \cdot X^z \geq 0 \\ 0 & \text{if} \quad r \cdot X^z < 0 \end{cases} \tag{4}$$

where $z = i * \delta$, therefore $X^z = \{X_z, X_{z+1}, \ldots, X_{z+|r|-1}\}$ is the subseries of length $|r|$ within the chunk X^i.

Example 1. *Assume we sketch the chunk X^{10} starting at position 10 within its time series X and $|X^{10}| = 20$. Given a random filter r of length 6, and a sketching δ set to 2. As shown in Fig. 2, the random filter r slides over X^{10} to extract subseries of length 6 by calculating the dot product. This product generates a bit (0 or 1) indicating the sign of the output. The generated sketch B^{10} is of length $|B^{10}|$=7, i.e., the dimensionality is reduced from 20 to 7 in this example.*

The following lemma states that if two chunks are very similar to each other, then their binary sketches will also be very similar.

Lemma 1. *If the Euclidean norm between two similar chunks X^i and Y^j is close to 0, i.e., $\left\|X^i - Y^j\right\| < \Omega$, where Ω is close to 0, then the probability that their respective sketches B^i and B^j will share the same value for each of the dimensions is close to 1.*

Proof. The fact that $\|X^z - Y^z\| < \|X^i - Y^j\| < \Omega$, implies:

$$\|X^z - Y^z\| < \Omega \tag{5}$$

where X^z and Y^z are the z^{th} subseries of length $|r|$ of X^i and Y^j, respectively. Since in practice X^z and Y^z are large vectors of length around 60 or larger, the Euclidean Norm of these vectors is very close. Hence, we assume that $\|X^z\| = \|Y^z\|$. Here we can assume that they are equal to 1. Hence, by [12] the cosine similarity between these vectors is:

$$\begin{aligned} &\|X^z - Y^z\|^2 = \|X^z\|^2 + \|Y^z\|^2 - \cos\alpha \|X^z\| \|Y^z\| \\ &\|X^z - Y^z\|^2 = 2 - 2\cos\alpha \end{aligned} \tag{6}$$

where α is the angle between the vectors X^z and Y^z.
From Equations 5 and 6:

$$\cos\alpha > 1 - \frac{\Omega}{2}. \tag{7}$$

The hash function used in sketching (refer to Eq. 4) is known to preserve the cosine similarity between vectors. Thus, the probability that their two sketches share same hash value is high, namely, by [18] it is:

$$p = 1 - \arccos\frac{\alpha}{\pi}. \tag{8}$$

Combining Eq. 7 and Eq. 8, we get:

$$p > 1 - \arccos\frac{2\pi - 2\Omega}{\pi}. \tag{9}$$

By plugging Ω which is close to 0 (see Lemma 1) in Eq. 9, p becomes close to 1. Lemma 1 is thus proven. □

N-Gram Generation: For a chunk X^i, a weighted set S^i is generated by extracting all grams of length n (i.e., n-grams) from the corresponding sketch B^i along with the frequencies of each n-gram as its weights. Thus corresponds to:

$$S^i = \{(s_j, f_j) | s_j = \{B^i_j, B^i_{j+1}, ..., B^i_{j+n-1}\}, 1 \leq j < |B^i| - n\} \tag{10}$$

where s_j denotes the j^{th} n-gram and f_j denotes its corresponding weight, i.e., frequency of its occurrence in B^i.

Example 2. *Given the sketch B^{10} generated in Ex. 1, we slide a window of length 3, over the sketch to extract the 3-grams along with their frequencies in B^{10} as their weights in Fig. 3. The weighted set S^{10} of B^{10} is represented as a vector of these*

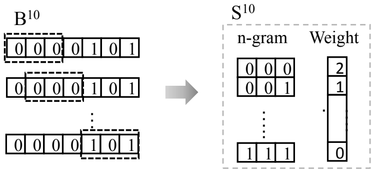


Fig. 3. Generating n-grams from sketch B^{10} outputs a weighted set S^{10}

```
Algorithm 1: SPS: Single Pass Signature
Input  : Weighted Set Vector $S_j$, Signature Length $g$
Output : Signatue $sig$
Initialize $sig[] = 0$
Declare:
    $P$ & $Z$ : Large primes where $Z \leq P$
    $a$ and $f$ : random Integer numbers
    $r = \frac{|S_j|}{g}$
foreach $i$=1 to $|S_j|$ do
    band $\leftarrow minimum((i)/r, g-1)$
    Sig[band] $\leftarrow Sig[band] + (a * S(i) + f)$
    Sig[band] $\leftarrow Sig[band]\ mod\ P\ mod\ Z$
end
return Sig
```

$2^3 = 8$ *grams. We keep the grams sorted based on their binary representation. Thus, the binary representation is implicit and we only need to maintain the weights vector.*

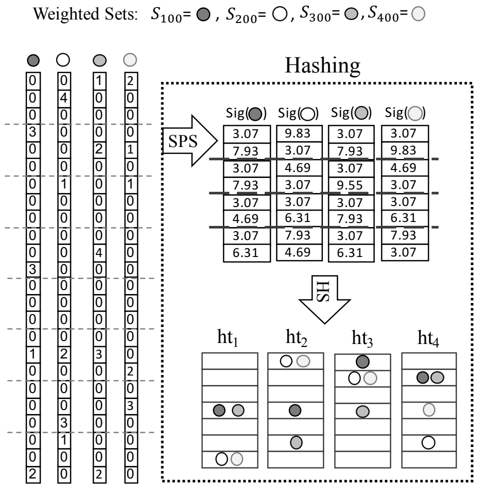
The distances between two weighted set vectors can be measured using the Hamming Distance metric as in Def. 7.

Definition 7. [Hamming Distance (HD)] Given two vectors V_1 and V_2 of equal length in the D-dimensional space of integer values, the hamming distance $HD(V_1, V_2)$ is defined as the number of places where V_1 and V_2 differ [12].

Lemma 2. *If two sketches B^i and B^j are similar, i.e. , differ in few places (refer to Def. 7), then their weighted sets S^i and S^j are also similar, i.e., differ in few places (refer to Def. 7).*

Proof. To prove the correctness of this lemma, we pick the case where two sketches differ in one position, i.e., $HD(B^i, B^j) = 1$, then prove that $HD(S^i, S^j) \leq 2n$. Given that the n-gram size is n and the step size to generate n-grams is 1, then S^i and S^j will differ in at most $2n$ grams due to the mismatch existing in one single position in their sketches. Note that the set size, i.e., total possible n-grams is 2^n. The ratio of the n-gram size to the total number of possible n-grams is $n : 2^n$. Thus, the weighted sets differ in very few places. In addition, as the size of n-grams n increases, the number of mismatches increases linearly in $2n$ and the number of matched n-grams grows exponentially, i.e., (2^n-$2n$). The rest of the cases can be proven in a similar manner. Lemma 2 is thus proven. □

Step 4 (Hashing): In practice, due to the use of n-grams, the resulted vectors tend to be extremely long and sparse, especially with large n [16]. Although sketching [17] uses a binary alphabet, describing a sketch using n-grams illustrates the fact that even for a small size alphabet and a relatively large n-gram size (i.e., $n \geq 15$, to meaningfully represent time series sketches [7]), one often has to deal with large very sparse vectors, e.g., $2^{15} = 32,768$.


Weighted Sets: S_{100}= ● , S_{200}= ○, S_{300}= ◍, S_{400}= ◌
Hashing
SPS
HS
ht$_1$
ht$_2$
ht$_3$
ht$_4$

●	○	◍	◌
0	0	1	2
0	4	0	0
0	0	0	0
3	0	0	0
0	0	2	1
0	0	0	0
0	1	0	1
0	0	0	0
0	0	0	0
0	0	0	0
0	0	4	0
3	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
1	2	3	0
0	0	0	2
0	0	0	0
0	0	0	3
0	3	0	0
0	1	0	0
0	0	0	0
2	0	2	0

Sig(●)	Sig(○)	Sig(◍)	Sig(◌)
3.07	9.83	3.07	3.07
7.93	3.07	7.93	9.83
3.07	4.69	3.07	4.69
7.93	3.07	9.55	3.07
3.07	3.07	3.07	3.07
4.69	6.31	7.93	6.31
3.07	7.93	3.07	7.93
6.31	4.69	6.31	3.07

Fig. 4. Hashing Weighted Sets Using Algorithm 1 Single Pass Signature (SPS) Followed by Hashing Signatures (HS) using Algorithm 2.

The sparsity results from the fact that usually time series have a limited pattern compared to other data. This is also confirmed by our experiments in Section V-A where sparsity is measured using the following equation:

$$Sparsity(\%) = \frac{number\ of\ zeros\ in\ S^i}{|S^i|} \times 100. \qquad (11)$$

Unfortunately, to the best of our knowledge, no existing hashing algorithm *tailored* for *extremely* sparse weighted sets has been proposed to date in the literature. Therefore, we propose a new hashing technique, called *Single Pass Signature (SPS)*, which overcomes the drawbacks of existing techniques.

Single Pass Signature (SPS). Given a weighted set S^i, our proposed solution SPS generates a signature of length g in a $single$ pass such that $g <<< |S^i|$. To generate a signature, SPS breaks the weighted set vector into g disjoint bins, followed by mapping the weights in each bin into a single value using some hashing function H (See Algorithm 1) with the number of non-zeros weights being very small due to the sparsity of the vector. In our work, we select H to be *2-universal hashing function* [19] since it achieves a low number of collisions, however, other hash functions could equally be plugged in.

Given that the weights of features as explained above (i.e., n-grams), the vector is sparse and consequently bins are sparse, we hash entire bin weights into a single integer number. The result g non-zero hashes are then concatenated as SPS signature.

Example 3. *Referring to Fig. 4, suppose we have* 4 *sparse weighted sets* S^{100}, S^{200}, S^{300} *and* S^{400}. *Then to hash them using the SPS technique, we pick a hashing function* H; *a 2-universal hashing in our case. The simplest strategy for* H *is to pick two large prime numbers* P *and* Z *where* $P \geq Z$, *sample two random numbers* a, f *and compute [19]: H(x) = (((ax+f) mod P) mod Z). In this example, a=1.62, f=4.69, P=101*

Algorithm 2: HS: Hashing Signatures into Hash Tables

```
Input   : Signature $sig$, number of bands $b$ and rows $r$
Output : mapping $sig$ to a bucket in each of $b$ hash tables
Initialize:
    $hashes[b] = 0$
    $hts\_bkts[b] = 0$
Declare:
    $P$ : Large prime
foreach $i$=1 to $g$ do
    Band ← $minimum((i)/r, b-1)$
    hashes[Band] ← $(hashes[Band] + sig(i))\ mod\ P$
end
foreach $m$=1 to $b$ do
    $hts\_bkts$.add ((m, hashes(m)))
end
return $hts\_bkts$
```

and Z=11 . For ease of presentation, sets are represented as circles where similar colors of the circles indicate the similarity of their corresponding sets. Each set is of length 24 *and the degree of sparsity is* $\approx 83\%$ *(Eq. 11). g is set to* 8*. Thus, the weighted sets will divided into* 8 *bins of size* $24/8 = 3$*. Then to generate a signature, each bin will be hashed using the hash function* H *as in Algorithm 1.*

This generates a lower-dimensional signature sig of length g ($g <<< |S^i|$). Next, to index the chunks, we build hash tables and index their weighted set signatures. We construct the hash tables by applying the well-known LSH approach known as Banding Technique [12]. Given a signature of length g, the signature would be broke down into b bands, each of size r. The choice of parameters r and b corresponds to a trade-off between speed and accuracy, the smaller r, the more accurate the result to answer the query, but the slower the response time [12]. Besides, the choices of r and b also decide how similar a pair of signatures needs to be to share a bucket in at least one hash table. This similarity (JS) (See Def. 6) approximately corresponds to $s = (\frac{1}{b})^{1/r}$ [12]. This similarity (JS) as a parameter is passed along with the signature sig to Algorithm 2, which maps the signature into b hash tables.

Example 4. *Following Example 3 referring to Fig. 4, suppose that we want signatures that are* $\geq 50\%$ *similar to share a bucket in at least a single hash table* ht_i*, then s is set to* 0.50*. As a result we got* $r = 2$*,* $b = 4$*. Each of the* 4 *bands will be hashed into* 4 *hash tables as shown in Figure 4. In Algorithm 2, we use the mod hash function with in this example* $P = 7$*.*

Lemma 3. *If two weighted set vectors* S^i *and* S^j *share the majority of positions, e.g.,* $HD(S^i, S^j) \leq 2n$ *(where* $|S^i| = |S^j| = 2^n$*), then ChainLink guarantees that by generating an SPS signature of lengths* $2n+1$ *and hash it, vectors will collide in at least a single hash table.*

Proof. By Lemma 2, assuming that the weighted sets S^i and S^j have $2n$ dissimilar grams, and given that SPS breaks the weighted sets S^i and S^j into equal length bins, the worst case scenario of the distribution of the dissimilar $2n$ grams within the set is to have a single affected gram in each bin. Given that, to guarantee that the signatures will share at least a single bit, then the number of bins, i.e., the length of signature g should be $> 2n$. *Notice* that n is in the order of few tens while signatures can range from a few hundreds to a few thousands, i.e., the majority of the signatures will be identical for both sets. However, assuming the worst case, we set g to $2n + 1$, then to hash the signatures and guarantee that they will collide in at least a single hash table, we set r to 1 and b to $2n + 1$, i.e., $2n + 1$ hash tables. Lemma 3 is proven. □

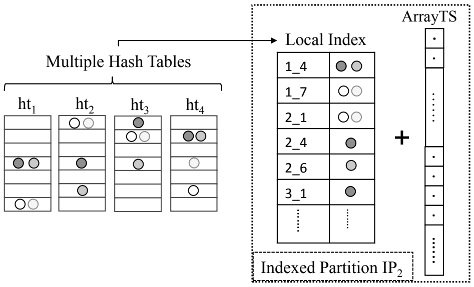


Fig. 5. Indexed Partition (IP_2). Consists of: Local Index, Array-Organized Time Series and Partition Number ($p = 2$).

Based on Lemmas 1, 2, and 3, the following important theorem can now be formulated that ChainLink index preserves the similarity of the original chunks that were indeed similar.

Theorem 1. *Given two similar chunks* X^i *and* Y^j *, i.e., ED(*X^i*,* Y^j*) close to 0, it is guaranteed that* X^i *and* Y^j *will collide in at least a single hash table by constructing ChainLink index.*

Proof. This Theorem is proved using Lemma 1, 2 and 3 □

C. ChainLink Global Index : CL-Global

Lastly, we design the global index CL-Global to correspond a centralized yet highly compact index used for effective pruning of irrelevant partitions during query processing. CL-Global, which resides in the cluster's master node is constructed by summarizing and aggregating the local indices. To build the global index, the master node retrieves from each indexed partition p the keys of its compact CL-Local hash map (only the keys which constitute the hash table ids and bucket numbers). All results are then *grouped by* the key, and the partition ids are combined together. In short, the global index (CL-Global) format is a centralized hash map, where the key field is a concatenation of a hash table id and a bucket number, and the value field is a list of partition ids.

To further compress the global index, the list of partition ids are compressed either using *range compression*, i.e., contiguous ids are compressed by keeping only the first and last ids as a range, or *bitmap compression*, i.e., the partition ids are represented as a bit vector and only the present ones are set to 1. Each entry in the global index is compressed using the more effective method between these two compression types.

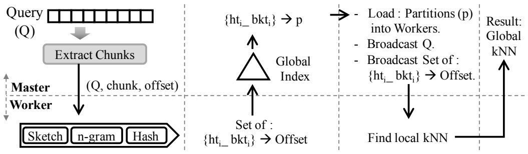


Fig. 6. ChainLink Query Processing.

Overall Index Construction: Following the map-reduce distributed paradigm, the entire ChainLink index, both the local index and the global one, is built in a single map-combine-reduce job. First, the mappers process each partition and create the corresponding indexed partition as in Figure 5. The combiners, one per each worker machine, combine the hash-map keys from all partitions within each machine, to create a partial intermediate global index, and then a single centralized reducer combines these partial indexes to form the final global index. As will be confirmed by the experimental evaluation in Section V-B, the creation of the entire index is an efficient operation that involves minimal data shuffling and distribution among the cluster machines.

IV. ChainLink Query Processing

The strategy of answering an approximate kNN query Q using ChainLink index is as follows (see Figure 6). Assume an input query sequence Q and a user-defined parameter k. The master node divides Q into chunks using a sliding window of length w, where w is the window size used for building the ChainLink index. These chunks, which can be many for long query sequences, are distributed randomly across the worker nodes. Each chunk is processed by *sketching, n-gram generation* and *hashing* into multiple hash tables using our proposed SPS technique (See Section III-B).

The hashing results, which are a collection of entries with a key of $\{$*hash table id* (ht_i)*, bucket id* $(bkt_i)\}$ (Algorithm 2), and a value of a chunk offset within Q, are shipped back to the master node as in Figure 6. The master node accesses the corresponding entries (having the same key) in the global index CL-Global, and retrieves their respective partition ids. The worker nodes holding these partitions start to perform the local processing. The worker nodes receive the broadcasted Q and its hashing results. Each worker node accesses the corresponding entries in the CL-Local (the entries having the same key), and retrieves the matching time series ids and their chunk ids. From these ids, the actual time series data are retrieved in O(1) time complexity due to the array organization.

Finally, the collected time series chunks represent the candidate set within which the search for the best local kNN is performed using Euclidean Distance (Def. 4). The best kNN from each worker are then collected by the master node for producing the final results.

V. Experimental Evaluation

A. Experimental Methodology and Setup

Cluster Setup. All experiments were conducted on a cluster consisting of 2 nodes each composed of 56 Intel@Xeon CPU E5-2690 2.60GHz processors, 500GB RAM, 3.5 TB HDD.

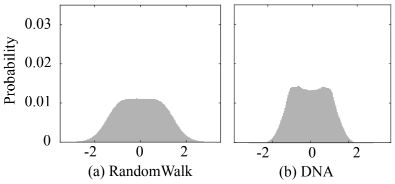


Fig. 7. Datasets Distribution.

Each node is connected to a Gigabyte Ethernet switch and runs Ubuntu 16.04.3 LTS with Spark-2.0.2 and Hbase-1.4.7.

Baseline Indexing Solutions. To show the effectiveness of our novel *SPS* hashing algorithm designed for ChainLink, we have created two variations of *ChainLink*. In one variation we use *SPS*, referred to as **CL-SPS**, and in the other we use CWS [15], referred to as **CL-CWS**.

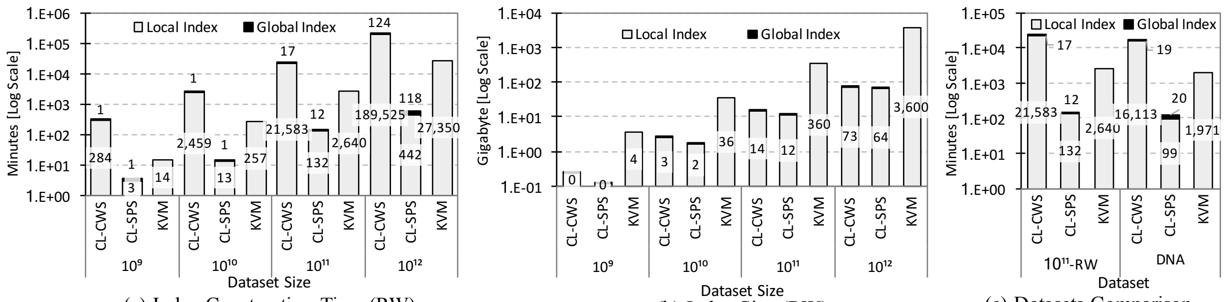
Given long subsequence matching as our target, we also compare ChainLink with UCR Suite [2]. We develop a *distributed variant* of UCR-ED that leverages the full compute power of distributed workers to assure we conduct a fair comparison. We refer to this solution as **UCR-ED**. Lastly, we also consider KV-match [6] as a baseline for indexing subsequences in a distributed system, referred to as **KVM**.

Although the work in [14] is relevant, it is excluded from our comparison because: (1) The construction of their red-green map structure requires not only multiple passes over the weighted sets but also mandates materializing them on disk, which is prohibitively expensive for TB scale data, and (2) The sampling time to generate the hashing signatures is inversely proportional to the density of the weighted set vectors, and thus for very sparse data the sampling time becomes impractical.

Data Sets. We use two benchmark datasets. ***RandomWalk (RW)*** has been used extensively as the benchmark for time series indexing in other projects [5], [7], [2]. This data set is generated for 10^8 time series each with $2 * 10^4$ data points (i.e. $10^{12} \approx$ 3.5 TB). ***DNA*** [20] contains an assembly of the human genome collected in (2000-2013). Each DNA string is converted into time series. We then concatenate them and use a sliding window of length 10^4 to generate a 200 GB dataset. Refer to Fig. 7 for these datasets distribution.

Parameter Settings. For this study, two ChainLink indices are built, one with a window of length $1,000$ and another of length $2,000$For the parameter selection, the choice of the length of the random filter R is data dependent. Even though, a 1-bit sketch with a large R would be non-informative while it would be noisy with a very small R. When $|R| \in$ [40 - 80] for RW and $|R| \in$ [30, 60] for DNA, query results are returned with an acceptable error ratio $\in$ [0.2 - 0.9] for different k values. When $|R| > 80$ for RW and $|R| > 60$ for DNA, the error ratio increases. The effect of decreasing $|R|$ on the processing speed is negligible, just a few minutes.

Regarding the step size δ, we found that the best range to pick for δ is from [0 - 10] to limit the error ratio from exceeding 0.9. The effect on the processing speed by decreasing δ is negligible, only few minutes. For n-gram generation, the size of n is set to be in [15-20] to assure that the error ratio wont exceeds 0.9. For RW, we chose the following: $|R| = 60$, $\delta = 5$,



(a) Index Construction Time (RW)

(b) Index Size (RW)

(c) Datasets Comparison

Fig. 8. Index Construction Analysis [Log Scale].

$n = 15$, where we chose $|R| = 50$, $\delta = 3$, $n = 15$ for DNA. For the hashing parameters, we choose the length of signature to be $sig = 256$, the number of rows and bands are $r = 8$ and $b = 32$ respectively. LSH applications require the hash table to have a large number of buckets, like 500.

Degree of Sparsity. A sample is taken, sketched, a weighted set is generated using n-grams generation and then Eq.11 is applied on each weighted set and the results are averaged. The degree of sparsity of RW is about 99.92%, meaning that only $2,621$ out of $32,768$ are non-zero values whereas it is about 99.80% containing $\approx 6,554$ non-zeros for DNA.

B. Evaluation of Index Construction

Index Construction Time. The ChainLink index construction time is dominated by the *CL-local* construction time. Comparing *CL-SPS* to *CL-CWS*, *CL-local* construction time demonstrates the speedup obtained using *SPS*. *CL-SPS* CL-local construction time is *two* orders of magnitude faster than *CL-CWS* as shown in Figure 8 (a). Better yet, *CL-SPS* is more than an *order* of magnitude faster than *KVM*. *Note that those numbers refer to the process of building the two indices.*

Figure 8 (c) shows the difference between two data sets with the same size, 10^{11}-RW and 10^{11}-DNA. There is no huge difference between them, *CL-SPS* takes a total of 144 minutes for RW while it takes 119 for DNA. The difference is due to the data dependent parameters used for sketching. Different data sets have different patterns. Hence, different parameters for sketching are needed to capture the trends. Despite that, *CL-SPS* continuous to be superior by achieving *two order of magnitude* and *one order of magnitude* faster speed than *CL-CWS* and *KVM*, respectively.

Index Size. The size of the CL-local index is affected by the number of $buckets$ for each hash table, the length of the signature and as a consequence r and b (i.e., number of hash tables). Since we use the same parameters for both *CL-SPS* and *CL-CWS*, the size of their indices are comparable. That is, the former is somewhat smaller than the latter. For instance, for 10^{12}-RW, the size of CL-local generated by *CL-SPS* equals to 63.9 GB ($\approx$ 1.78% of dataset size) whereas it is about 72.7 GB ($\approx$ 2.03% of dataset size) for that generated by *CL-CWS*. On the other hand, the *KVM* index size is prohibitively large in size, namely, the size of the index equals almost the size of the data set.

While CL-global is affected by the aforementioned parameters and the number of partitions, the compression technique described in Section III-C overcomes the effect of this parameter. For 10^{12}-RW, the size of CL-global generated by *CL-SPS* is about 6 MB whereas it is about 7 MB for that generated by *CL-CWS*. This lightweight index size enables ChainLink to persist CL-global in the master node's RAM to process consecutive queries.

C. Evaluation of Query Processing

We measure the query performance by taking the average of over 50 distinct subsequence queries. We study the query response time and the quality of the kNN-approximate answer under a rich variety of parameter settings. The quality of the kNN-approximate answer is measured by the *error ratio* and *recall* metrics that are standard metrics in the context of high-dimension nearest neighbor queries [13], [11], [5]. Given a query Q, we denote the set of exact k nearest neighbors as $G(Q) = \{g_1, \cdots, g_k\}$, and the actual query result as $R(Q) = \{r_1, \cdots, r_k\}$. Then the *error ratio* $\geq$ 1 is defined as:

$$error\ ratio = \frac{1}{k}\sum_{j=1}^{k}\frac{ED(Q, r_j)}{ED(Q, g_j)} \qquad (12)$$

where 1.0 is the best score and it indicates an exact match between sets G and R, and the closer to 1.0 the better the quality of the results.

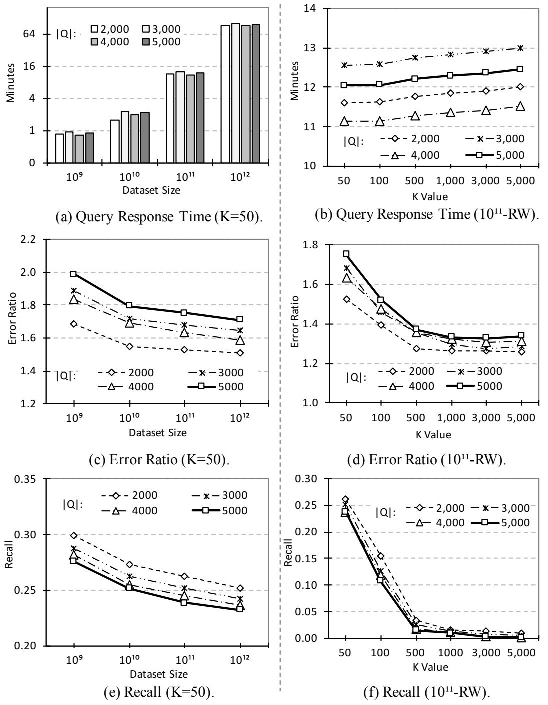
On the other hand, *recall* $\in [0, 1]$ is defined as:

$$recall = \frac{|G(q) \cap R(q)|}{|G(q)|} \qquad (13)$$

In the ideal case, the *recall* score is 1.0, which means all k nearest neighbor are returned, and the closer to 1.0 the better the match with the exact answer set.

Since we compare CL-SPS against three systems with different characteristics (CL-CWS, distributed UCR-ED and KVM), we evaluate its query processing against each solution separately to achieve a fair comparison. Below, we first study the effect of varying the dataset size and the k value for different query lengths on the response time and, error ratio and recall for CL-SPS.

Impact of Varying Parameters on Query Performance for CL-SPS. On the left hand side of Fig. 9, we study the effect of varing the dataset size. In Fig.9 (a), the query response time



(a) Query Response Time (K=50).

(b) Query Response Time (10^{11}-RW).

(c) Error Ratio (K=50).

(d) Error Ratio (10^{11}-RW).

(e) Recall (K=50).

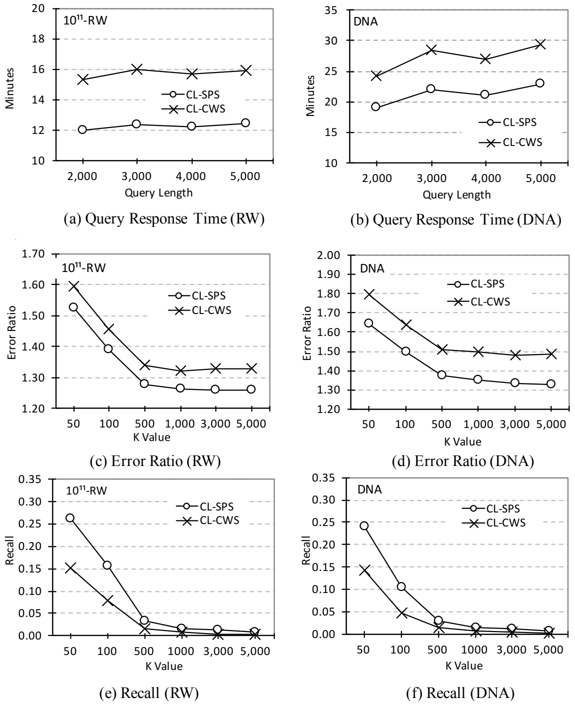
(f) Recall (10^{11}-RW).

Fig. 9. Impact on Query Response Time, Error Ratio & Recall. Left: Varying Data Set Sizes. Right: Varying K Values.

for different query lengths ($|Q| \in \{2000, 3000, 4000, 5000\}$) under different dataset sizes follows the same pattern. That is, the time for different lengths is almost the same on the same dataset size. The increase in time for a given query length on different datasets is linear in the size of the dataset, e.g., the response time for $|Q| = 5,000$ on ($10^9, 10^{10}, 10^{11}$ & 10^{12}) is respectively (0.9, 2.4, 12.5, 100.5) minutes.

In Figure 9 (c), we study the error ratio for different query lengths under various data set sizes. We observe that it again follows the same pattern, i.e., as the size of the data set increases, the error ratio decreases for all query lengths. Also the error ratio of the approximate answer approaches 1, i.e., it is very similar to exact answer. As shown, it is decreasing from about 1.7 to about 1.5 for $|Q| = 2000$ as the data set size increases from 10^9 to 10^{12}. This proves the efficiency of answering near-exact kNN using LSH approaches for TB-scale datasets.

In Figure 9 (e), we study the recall for different query lengths under various data set sizes. As shown in the figure, the recall is within the range of 30% to 25%. As expected, the recall gets decreased as the dataset size increases since more approximate matches become possible. On the right hand side of Figure 9, we study the effect of varying the number of neighbors k. In Figure 9 (b), as the value of k increases (50 to $5,000$), the response time slightly increases (in seconds). In Figure 9 (d), the error ratio for different query lengths follows a similar pattern. The error ratio for a given query length decreases as k increases because the search space gets larger and objects very similar to the query object are found. For large k, e.g., $k \in [500 - 5,000]$, the error ratio becomes constant since a larger set of values are averaged according to Eq. 12. In Figure 9 (f), the recall value over small k is around 25%, and it gets smaller as k increases.



(a) Query Response Time (RW)

(b) Query Response Time (DNA)

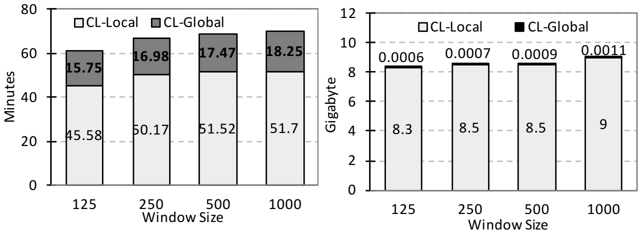
(c) Error Ratio (RW)

(d) Error Ratio (DNA)

(e) Recall (RW)

(f) Recall (DNA)

Fig. 10. Query Evaluation (CL-SPS vs. CL-CWS).



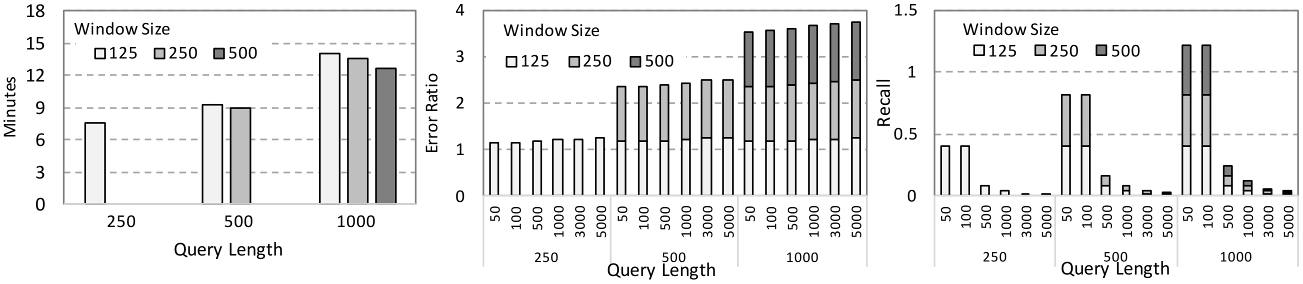
(a) Index Construction Time

(b) Index Size

Fig. 11. Impact of varying window size on index construction time and size.

Although the recall value when analyzed in isolation seems low, it is crucial to combine the recall results with the error ratio results for better interpretation. More collective analysis over Figures 9 (c) & (e), and (d) & (f) indicates that for big datasets there is a very good chance to find very close high-quality approximate matches to your query object even if they are not identical to the exact answer set. Moreover this chance increases as the dataset size and k increase, which is aligned with our main assumption that approximate processing is more suitable for big data applications.

ChainLink Query Evaluation using Different Hashing Schemes CL-SPS vs. CL-CWS. In Figure 10, we evaluate the query response time, error ratio and recall for two different data sets, namely, 10^{11}-RW and DNA. In Figures 10 (a) and (b), the query response time of our CL-SPS method is up to 33% faster than CL-CWS for both 10^{11}-RW and DNA for all query lengths where $k = 50$.



(a) Query Response Time

(b) Error Ratio

(c) Recall

Fig. 12. Impact of varying window size on query response time, error ratio and recall.

For the error ratio (Figures 10 (c) & (d)) and the recall (in 10 (e) & (f)), the behavior is very similar to that in Figure 9. The CL-SPS algorithm shows better error ratio and recall than CL-CWS because the proposed SPS hashing mechanism preserves the proximity among the time series subsequences better.

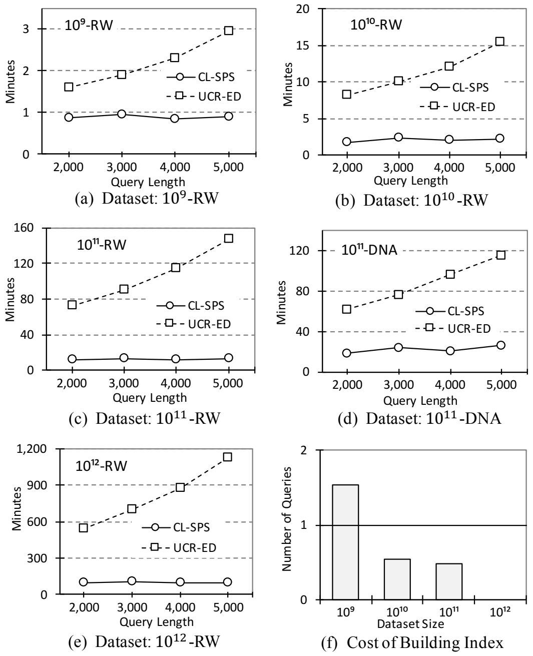
ChainLink Index Construction and Query Evaluation using Shorter Window Sizes.

we performed analysis on shorter window sizes, i.e., $\{125, 250, 500\}$, and studied the effect on the index construction time, index size, query response time, error ratio and recall. Due to space limitations, these experiments are not added to the main paper. But instead, we now have created an extended version (a technical report) that now includes these additional experiments. In Figure 11, we study the effect of varying the window size on the index construction time and size.

We fixed all other parameters as follow: $dataset\ size = 10^{11}, r = 40,\ \delta = 2,\ n = 15,\ sig = 256,\ r = 8\ and\ b = 32$. Figure 11(a) illustrates that the index construction time is slightly affected by the window size. As we double the window size, the construction time increases by a few minutes < 5 for both CL-Local and CL-Global. Better yet, the effect on the index size is also negligible. The minor effect is resulted from having different length sketches when using different window lengths, $|sketch| = \frac{|window\ length| - |sketching\ vector\ (r)|}{|sketching\ stepsize(\delta)|}$. The longer the window, the longer the time it takes to sketch time series chunk. In addition, the longer the window, the longer the sketch (i.e. the more pattern captures), thus *slightly* larger index size.

In Figure 12, we study the effect of varying the window size on the query response time, error ratio, recall and the support for short queries. As depicted in Figure 12(a), as the length of the window increases, the response time decreases and vice versa. However, the effect is minimal. This is because we use the *one-step sliding window* mechanism at query time and query the index for each extracted window. Thus, given large window size, the number of extracted windows would be less than that of smaller windows. Where number of extracted windows = |query|- window size + 1.

On the other hand, varying the window size has no effect on the error ratio (Figure 12(b)) nor the recall (Figure 12(c)). This is because sketching, n-gram generation and hashing are done linearly over the chunk, and varying the window size do not affect the pattern approximation of the time series.



(a) Dataset: 10^9-RW

(b) Dataset: 10^{10}-RW

(c) Dataset: 10^{11}-RW

(d) Dataset: 10^{11}-DNA

(e) Dataset: 10^{12}-RW

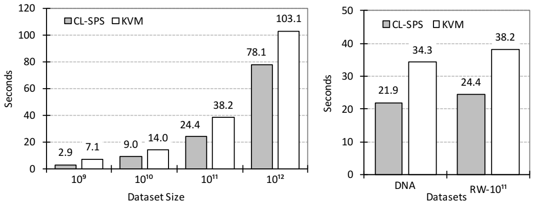
(f) Cost of Building Index

Fig. 13. Query Evaluation (CL-SPS vs. UCR-ED).

It is worth to also highlight the good effect of small r value ($r = 40$) on the error ratio and recall for small k values in Figure 12(b) compared to the results reported in Figs.9 (d) & (f) where the used r equals 60. This is because smaller r captures the patterns more accurately, however, with a slower processing time. We used a smaller $|r|$ in this experiment since we use smaller window sizes.

Query Evaluation of ChainLink CL-SPS vs. UCR-ED. UCR-ED employs branch and bound algorithms to safely prune the search space without building an index beforehand. Although it is said to support long subseqences better than existing systems, as the query length grows, we notice that the bounds become looser causing decrease in the query response time as our experimental study demonstrates.

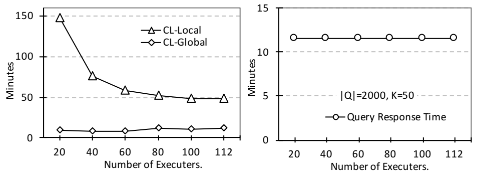
We study query response time for different query lengths where $k = 5,000$ while varying data set sizes. We also study how many queries need to be executed by both UCR-ED and CL-SPS before the overcome of first building the CL-SPS respective indices before utilizing them for query processing.



(a) Index Probing Time (RW)

(b) Datasets Comparison

Fig. 14. Query Evaluation (CL-SPS vs. KVM).



(a) Index Construction Time (10^{11}-RW).

(b) Query Response Time (10^{11}-RW).

Fig. 15. Scalability with Increasing Number of Executers.

This is calculated as follows:

$$Number\ of\ Queries = \frac{idxTime - Saving}{UCR\ response\ time} \tag{14}$$

where *idxTime = CL-SPS total index construction time* and *saving = UCR-ED response time - CL-SPS response time.*

Generally, as shown in Figures 13 (a-e), the query response time for both UCR-ED and CL-SPS follows the same pattern across different data sets sizes. As the query lengths grow, CL-SPS almost remains constant while the UCR-ED response time increases linearly. In Fig. 13 (a) & (b), CL-SPS is 85% to 600% faster than UCR-ED on 10^9-RW & 10^{10}-RW. As can be seen, the larger the data set the faster the query excution performed by CL-SPS becomes compared to UCR-ED.

In Fig. 13 (c) & (d), the query response time is studied for different data sets with the same size, 10_{11}-RW and DNA. For the former data set, CL-SPS is $500\% - 1000\%$ faster than UCR-ED whereas for the latter it is faster by $240\% - 500\%$.

In Fig. 13 (e), CL-SPS records its fastest response on the largest dataset in our study, namely, 10^{12}-RW. It is $500\% - 1019\%$ faster than UCR-ED.

To evaluate CL-SPS more thoroughly, we consider not only the query processing time but we also incorporate the time consumed for first building the index. As explained above, the metric in Equation 14 determines the number of queries executed until the cost of building indices is amortized. This number is calculated for each dataset size, given that CL-SPS supports arbitrary query lengths, the query length selected for this study corresponds to the mid-point of all the different query lengths considered in this study, namely, $3,500$. As a consequence, the query response time for both CL-SPS and UCR-ED corresponds to that of query length $|Q| = 3,500$.

It is important to observe, as shown in Fig. 13 (f), that our ChainLink CL-SPS solution starts to pay back the costs of index construction time already after approximately *two* queries for 10^9-RW and after *a single* query for both 10^{10}-RW and 10^{11}-RW. Moreover, for the largest dataset of size $\approx$ 3.5 TB, i.e., size 10^{12}, there is no cost to pay back. In fact, the index construction time for this dataset is smaller than the query response time of UCR-ED.

Query Evaluation of ChainLink CL-SPS vs. KVM. Although KVM is the state-of-the-art distributed index for subsequence matching, it only supports range queries (i.e., ϵ-queries) instead of the more complex kNN queries we target. To overcome this discrepancy in functionality, we adopt the following commonly-adopted strategy for realizing the kNN functionality by leveraging a range query. Namely, the index is queried with different ϵ values until the number of results retrieved is $\geq k$. The procedure selects a specific value like $\epsilon = 20$ to start with, and then tracks the number of retrieved answers. If it is less than k then epsilon is doubled (i.e., $\epsilon = 40$) recursively until the number of retrieved results becomes $\geq k$. A second challenge in this comparison is that KVM is implemented on top of HBase, where the time series objects and the index entries themselves are all stored in Hbase tables. Thus a metric that is infrastructure-independent is needed, because the query response time for KVM also includes the time for fetching consecutive rows from the HBase index tables and HBase time series tables multiple times from disk to memory whereas the CL-SPS query response time includes loading partitions that contain candidate matches into workers' memory. Since both systems support the indexing of subsequences, we measure the quality of the index by calculating the index probing time (i.e., the time needed to visit the index to get candidate matches).

As illustrated in Figure 14 (a), the index probing time for both systems CL-SPS and KVM increases linearly in the dataset size. However, our CL-SPS solution is $32\% - 144\%$ faster than KVM. In Figure 14 (b), CL-SPS is 57% faster than KVM on both data sets 10^{11}-RW and DNA, meaning that the index probing time is almost the same on different dataset sizes.

ChainLink Scale-out Performance. Figure 15 displays results from an experiment where we fix all parameters and vary the number of executers in the cluster over {20, 40, 60, 80, 100, 112}. We measure both critical metrics of the index construction time and the query response time, while other metrics are not highly sensitive to the cluster configuration. The results in Fig. 15(a) show that ChainLink scales up very well as the degree of parallelization increases. This is because the index construction process is a single map-reduce job with no hidden communication, synchronization, or bottleneck. Thus it is expected to scale-up very nicely as more resources are added to the cluster. Moreover, Fig. 15(b) shows that the query response time is constant. This is because the global index of ChainLink typically assures that very few partitions are accessed, which then can be performed in parallel all at once.

VI. RELATED WORK

Although the literature on time series topics is vast; little work exists to date in distributed systems for tackling massive sets of such time series data for similarity exploration. Some work focuses on the simple problem of complete time series matching, i.e., whole sequence matching. In such whole sequence matching, it is assumed that the time series objects to be indexed and compared have the same length [21]. Two recently proposed systems [22] and [5] address the problem of whole sequence matching, which is a fundamentally different problem from subsequence matching.

In addition, few systems focus on the more general subsequence matching. The work in [23] and [24] support long subsequences, however they are designed for centralized processing. Therefore, these systems [23], [24] and our system target different infrastructures, different scale of data, and consequently different challenges and design decisions. For example, it is reported in [23] that building the index over 750 GBs dataset takes around 41 hours. In contrast, our system, as a distributed solution, takes 9 hours to build the index over 3 TB dataset.

Wang et al. [1] propose a distributed system which constructs a vertical inverted table and horizontal segment trees based on the PAA summarization of time series data. This system only supports a simplified subsequence matching, namely, to match prefixes on a fixed offset location of the time series specified by the user in the query. Moreover, the authors state that for large $k > 50$, their kNN query performance degrades quickly and converges to the brute force search [1]. In contrast, ChainLink is scalable for k in the thousands. KV-Match [6], focusing on *distributed* range query support instead of the more complex kNN queries, implements a file-based structure on HBase tables. As our experiments in Section V-C confirm, KV-Match has serious scalability issues regarding the index size in the same scale of the original data set and in index construction time. In contrast, as shown by our experiments, our ChainLink is an order of magnitude faster in index construction time.

VII. Conclusion

In this work, we demonstrated that the combination of *big time series data*, *distributed processing*, *long query sequences*, and *approximate kNN similarity search*, introduces real challenges to many modern applications that to date has not yet been addressed properly. To address these challenges, we proposed *ChainLink*, a scalable distributed indexing framework for big time series data. ChainLink introduces an integrated solution including data re-organization for efficient record-level accesses, scalable indexing structure for approximate similarity search, and a novel hashing technique (SPS) as a core building block of the index. Our experiments over an extensive set of parameter settings and data sets show the superiority of ChainLink over existing technique by orders of magnitudes in terms of index construction overheads and query processing while maintaining excellent result accuracy.

References

[1] 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, pp. 215–230.

[2] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh, “Searching and mining trillions of time series subsequences under dynamic time warping,” in *SIGKDD*. ACM, 2012, pp. 262–270.

[3] S. Har-Peled, P. Indyk, and R. Motwani, “Approximate nearest neighbor: Towards removing the curse of dimensionality,” *Theory of computing*, vol. 8, no. 1, pp. 321–350, 2012.

[4] T. Liu, A. W. Moore, K. Yang, and A. G. Gray, “An investigation of practical approximate nearest neighbor algorithms,” in *Advances in neural information processing systems*, 2005, pp. 825–832.

[5] L. Zhang, N. Alghamdi, M. Y. Eltabakh, and E. A. Rundensteiner, “Tardis: Distributed indexing framework for big time series data,” in *ICDE*. IEEE, 2019.

[6] J. Wu, P. Wang, C. Wang, W. Wang, and J. Wang, “Kv-match: An efficient subsequence matching approach for large scale time series,” 2017.

[7] C. Luo and A. Shrivastava, “Ssh (sketch, shingle, & hash) for indexing massive-scale time series,” in *NIPS*, 2017, pp. 38–58.

[8] Z. A. Neseeba P.B, “Performance analysis of hbase,” *International Journal of Latest Technology in Engineering, Management & Applied Science*, vol. 8, no. 10, pp. 84–89, 2017.

[9] Y.-S. Moon, K.-Y. Whang, and W.-K. Loh, “Duality-based subsequence matching in time-series databases,” in *ICDE*. IEEE, 2001, pp. 263–272.

[10] J. Wang, H. T. Shen, J. Song, and J. Ji, “Hashing for similarity search: A survey,” *arXiv preprint arXiv:1408.2927*, 2014.

[11] A. Gionis, P. Indyk, R. Motwani *et al.*, “Similarity search in high dimensions via hashing,” in *Vldb*, vol. 99, 1999, pp. 518–529.

[12] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive datasets*. Cambridge university press, 2014.

[13] 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.

[14] A. Shrivastava, “Simple and efficient weighted minwise hashing,” in *NIPS*, 2016, pp. 1498–1506.

[15] S. Ioffe, “Improved consistent sampling, weighted minhash and l1 sketching,” in *ICDM*. IEEE, 2010, pp. 246–255.

[16] A. Shrivastava and P. Li, “Densifying one permutation hashing via rotation for fast near neighbor search,” in *ICML*. IMLS, 2014, pp. 557–565.

[17] P. Indyk, N. Koudas, and S. Muthukrishnan, “Identifying representative trends in massive time series data sets using sketches,” in *VLDB*, 2000, pp. 363–372.

[18] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, “Practical and optimal lsh for angular distance,” ser. NIPS’15. MIT Press, 2015, pp. 1225–1233.

[19] J. L. Carter and M. N. Wegman, “Universal classes of hash functions,” in *JCSC*. Elsevier, 1979, pp. 143–154.

[20] UCSC, https://genome.ucsc.edu/.

[21] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, *Fast subsequence matching in time-series databases*. ACM, 1994, vol. 23.

[22] D.-E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas, “Dpisax: Massively distributed partitioned isax,” in *ICDM*, 2017, pp. 1–6.

[23] M. Linardi and T. Palpanas, “Scalable, variable-length similarity search in data series: The ulisse approach,” vol. 11, no. 13, pp. 2236–2248, 2018.

[24] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim, “The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art,” *Proc. VLDB Endow.*, vol. 12, 2018.