Université Paris Cité

FreSh: Lock-Free Index for Sequence Similarity Search

Panagiota Fatourou, Eleftherios Kosmas, Themis Palpanas, George Paterakis

LIPADE-TR-Nº 9

October 15, 2022

Technical Report

LIPADE
Laboratoire d'Informatique PAris DEscartes

FreSh: Lock-Free Index for Sequence Similarity Search

Panagiota Fatourou
LIPADE, Université Paris Cité &
FORTH-ICS
faturu@ics.forth.gr

Eleftherios Kosmas
University of Crete
ekosmas@csd.uoc.gr

Themis Palpanas
LIPADE, Université Paris Cité &
French University Institute (IUF)
themis@mi.parisdescartes.fr

George Paterakis
University of Crete
csd4024@csd.uoc.gr

Abstract

We present FreSh, the first *lock-free* (thus, highly *fault-tolerant*) data series index that, surprisingly, exhibits the same performance as the state-of-the-art *lock-based* in-memory indexes. For developing FreSh, we studied in depth the design decisions of current state-of-the-art data series indexes, and the principles governing their performance. We distilled the knowledge we obtained to come up with a theoretical framework which enables the development and analysis of data series indexes in a modular way. After introducing the concept of *locality-awareness*, we proposed Refresh, a *generic approach* for supporting lock-freedom in a highly efficient way on top of any locality-aware data series algorithm. We built FreSh by repeatedly applying Refresh to make its different stages lock-free. Experiments, using several synthetic and real datasets, illustrate that FreSh, albeit *lock-free*, achieves performance that is as good as that of the state-of-the-art *blocking* in-memory data series index.

1 Introduction

Processing big collections of data series is of paramount importance for a wide spectrum of applications [41, 4, 43]. In the heart of analyzing such collections lies the process of similarity search. Given a query series Q, *similarity search* returns a set of data series from the collection that have the closest distance to Q. Similarity search comes at considerable cost, mainly due to two factors: (i) the increasing size of data series collections, and (ii) the *high dimensionality* of the data series produced by modern applications. To address these challenges, current state-of-the-art data series indexes [12, 7, 56, 44, 46, 45, 47, 48, 10], are based on data series summarization. They develop a tree index containing data series summaries, which they then use to prune data series of the collection in order to restrict the execution of costly computations only to a small subset of the original data series. This results in significantly enhanced performance in most cases [12, 13]. Moreover, recent work in designing highly-efficient indexes [44, 46, 45, 47, 48] has focused on exploiting the parallelism supported by modern multicore machines.

Naturally, the use of parallel (and/or distributed) components in designing data series indexes increases the need for fault-tolerance. The complexity and interdependence of such solutions may result in failures, usually observed at the software level, but sometimes also in hardware. When a failure occurs, it is crucial for the system, in terms of both reliability and performance, to continue its operation and produce valid outcomes. Unfortunately, the state-of-the-art data series indexes [12, 42, 44, 45, 47, 48] are largely lock-based in order to achieve synchronization in multi-threaded settings. Using locks results in *blocking* implementations, which are not fault-tolerant: if a thread that holds a lock fails, the entire system may block.

Lock-free algorithms avoid the use of locks to ensure fault-tolerance and enhance parallelism. *Lock-freedom* [35] is a widely-studied property when designing concurrent trees [16, 14, 28, 3] and other data structures [32, 35, 27]. It ensures that the system, as a whole, will continue to make progress, even if processes crash asynchronously. Lock-free solutions could, thus, be a promising approach for efficiently supporting fault-tolerance. Designing lock-free data series indexes is the focus of this work.

Challenges. To achieve lock-freedom, some form of *helping* is usually employed. That is, appropriate mechanisms are provided to make threads aware of the work that other threads perform, so that a thread may help others to complete their work whenever needed (e.g., when they crash, or are very slow). Unfortunately, conventional helping mechanisms are rather expensive and introduce high overheads [32, 35, 37, 57]. For this reason, the vast majority of the software stack is still based on locks. Ensuring lock-freedom without sacrificing the good performance of existing data series indexes is a major challenge.

State-of-the-art data series indexes are designed to maintain some form of *data locality*, and avoid synchronization as much as possible. For instance, they often separate the data into *disjoint sets*, and have a distinct thread manipulate the data of each set [46, 47, 48]. This processing pattern enables threads to work in parallel and independently from each other, which results in reduced synchronization and communication costs. These properties (which are crucial for performance) can be easily achieved when locks are employed. However, the way helping works seems to be inherently incompatible. them. It is thus unclear how to implement helping on top of such indexes without sacrificing these properties. Providing lock-freedom while maintaining load-balancing among threads, and ensuring a good data pruning degree are further challenges that need to be addressed.

State-of-the-art data series indexes encompass several data processing phases, which often employ different data structures to accomplish their processing in an efficient way. Designing lock-based versions of these data structures is possible. However, coming up with lock-free versions of them, while also respecting the reduced communication and synchronization cost principles that govern existing indexes, is another major challenge to address.

We observe that in order to develop a *generalized approach* for supporting lock-freedom on top of data series indexes in a *systematic way*, we need to study and understand the design decisions of current state-of-the-art indexes and the performance principles that govern them. Then, we need appropriate abstractions for the data series processing stages and their properties, as well as a set of design principles that need to be respected for efficiency. This is an ambitious task that generates additional challenges.

Our approach. We propose FreSh, a novel *lock-free* data-series index, that efficiently addresses all of the above challenges. To the best of our knowledge, FreSh is *the first* lock-free data series index proposed in the literature, and thus, the only one that is fault-tolerant. Our experimental analysis shows that the performance of FreSh is as good as that of MESSI [47], the state-of-the-art concurrent data-series index. Moreover, in many cases, FreSh performs better than MESSI, as it allows for increased parallelism when constructing the tree index. (FreSh and MESSI are in-memory indexes.)

To come up with FreSh, we developed a generic approach, called Refresh, which can be applied on top of a family of state-of-the-art blocking indexes to provide lock-freedom without introducing any additional cost (in comparison to the blocking versions). Refreshintroduces the concept of *locality-aware lock-freedom* which encompasses the properties of data locality, high parallelism, low synchronization cost, and load balancing met in the designs of many existing parallel data series indexes. None of the conventional lock-free techniques we are aware of has been designed with the goal of respecting these properties. Indeed, our experiments show that applying such conventional techniques for achieving lock-freedom results in significantly lower performance that using our technique.

Refresh respects the workload and data separation of the underlying data series index,

in order to not hurt the degrees of parallelism and load balancing of the index. Moreover, it provides a mechanism for threads to *determine* whether a specific part of a workload has been processed, and *help* only whenever it is necessary (i.e., threads that have crashed, or are really slow). Refresh introduces two modes of execution for each thread: (i) *expeditive* and (ii) *standard*. A thread that executes on the *standard* mode may incur synchronization overhead, as it needs to synchronize with helper threads, whereas a thread that executes on the expeditive mode executes a code that avoids all synchronizations. A thread starts by processing its assigned workload on expeditive mode. Helping is performed only after a thread has finished processing its own workload. Then, threads have to synchronize to execute on standard mode. This way, Refresh maintains the synchronization and communication costs as low as that of the underlying index.

Refresh can be applied on top of any *locality-aware algorithm* (Section 4) to get a lock-free version of it. FreSh (Section 5) follows the design decisions of locality-aware iSAX-based indexes [42]. However, to develop FreSh, we had to replace all data structures of the original index (i.e., MESSI [47]) with corresponding locality-aware lock-free versions. We present implementations of several concurrent data structures, such as counters, binary trees, and priority queues (Section 5). The novelty of these implementations is mainly their support for the expeditive and standard execution modes. Some of them also provide enhanced parallelism compared to employing existing concurrent data structures. We believe that these implementations, as well as Refresh, are of independent interest, and could be employed to get highly-efficient lock-free versions of several other big data processing solutions.

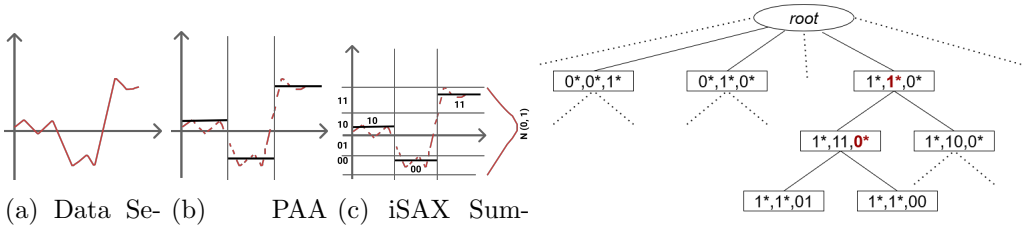
To be able to apply Refresh in a systematic way throughout all processing stages of an iSAX-based index, we introduce the abstraction of a *traverse object* (Section 3). The traverse object is an abstract data type that enables the design of an iSAX-based index in a modular way. It abstracts the main processing pattern used during the operation of iSAX-based indexes (and possibly also of other big-data processing solutions). Specifically, the iSAX-based index can be implemented as a sequence of operations on traverse objects. This way, we provide a generic methodology for designing iSAX-based indexes.

Contributions. Our contributions are summarized as follows:

- We develop a theoretical framework for designing, building and analyzing highly-efficient data series indexes in a modular way, and for supporting lock-freedom in a systematic way on top of them. Refresh is a *generic approach* that can be applied on top of any locality-aware data series algorithm.
- Based on Refresh, we develop FreSh, the first *lock-free*, highly fault-tolerant, and very efficient data series index.
- Our experiments, with large synthetic and real datasets, demonstrate that FreSh performs as good as the state-of-the-art *blocking* index, thus, paying no penalty for providing fault-tolerance.

2 Preliminaries and Related Work

Data Series, Indexing, and Similarity Search. A *data series* (DS) of *size* (or *dimensionality*) n is a sequence of n pairs, each consisting of a real value and its dimension. The *Picewise Aggregate Approximation (PAA)* [38] of a data series is a vector of w components which are calculated by spliting the x-axis into w equal segments, and by taking the mean values of the points of the data series that each segment contains (Figure 1). To calculate the iSAX summary [51] of the data series, the y axis is partitioned into a number of regions and a bit representation is introduced for each region. The *iSAX summary* is a vector of w components, each of which is a binary number representing the region in which the corresponding component of the data



(a) Data Series (b) PAA Summary (c) iSAX Summary (d) iSAX Tree

Figure 1: From data series to iSAX index

series PAA resides. The number of bits can be different for each region, and this enables the creation of a hierarchical tree index (*iSAX-based tree index* [42]), as shown in Figure 1.

We focus on *exact similarity search* (a.k.a. exact *1-NN*) which returns a single data series from a collection, which is the most similar to a query data series. Similarity is measured based on Euclidean Distance (ED), but our techniques are general enough to work for other popular *similarity measures*, such as Dynamic Time Warping (DTW) [49]. We call the distance between the *iSAX summaries* of two data series *lower-bound distance*. The lower bound distance between two data series is always smaller than or equal to their euclidean distance, which we call *real distance*. This property, called the *pruning property*, enables pruning of data series during query answering. Specifically, a data series can be *pruned* whenever its lower bound distance from the query series Q is higher than the real distance of any data series in the collection from Q.

Leaf-Oriented Trees. In a leaf-oriented tree, all data are stored in the leaves, which may store up to M keys each. During an insertion, if the appropriate leaf ℓ has room, the new key is placed in ℓ. Otherwise, ℓ is *split*: it is replaced by a subtree consisting of an internal node and two leaves. The keys of ℓ are distributed to the new leaves. If one of the newly created leaves is empty, the splitting process is repeated.

iSAX-Based Indexing. Concurrent iSAX-based data series indexes [44, 46, 45, 47, 48] (or *iSAX-based indexes* for short) work in two phases. During the *tree index construction phase* (1st phase), a set of *worker threads* work on a collection of input data series (called *raw data*), calculate an iSAX summary for each one of them, and build a *tree index* containing pairs of iSAX summaries and pointers to the corresponding data series. In an iSAX-based index, these pairs are first stored into a set of array buffers, called *summarization buffers* (*buffers creation* stage). Then, the worker threads traverse these buffers and insert their entries in a leaf-oriented tree (*tree population* stage). The use of buffers is employed to achieve high parallelism, a good degree of locality, and low synchronization overhead in building the index tree. Data series that have similar summarizations are placed into the same buffer and later in the same root subtree of the index tree.

Given a query data series Q, the following actions occur during query answering. A thread calculates the iSAX summary of Q and uses it to traverse a path of the tree, reaching a leaf ℓ. Then, the thread calculates the *real distance* between Q and each of the data series of ℓ, and stores the smallest distance among them in a variable called *BSF*. This distance will serve as an initial approximate query answer. Query answering proceeds in two stages. A set of *query answering threads* traverse the tree and use BSF to select those data series that are potential *candidate series* for being the final answer to Q (*pruning phase*). Those nodes whose lower bound distance to Q is larger than BSF are *pruned*. The candidate series are often stored in (one or more) priority queues [46, 47, 48]. Multiple threads process the elements of the priority queues by calculating their real distances from Q (*refinement stage*), and updating the BSF each time a new minimum is met. At the end of the query answering phase, the final answer is contained in BSF. *Barriers* among threads are often used at the end of each stage to ensure correctness.

System. We consider a system of N threads which are executed asynchronously and may

fail by crashing. When a thread *crashes*, it stops executing its algorithm and never recovers. An algorithm is *blocking* is a thread has to wait for actions by other threads before it makes progress. *Lock-freedom* guarantees that the system as a whole continues to make progress, even if all but one thread crash.

2.1 Additional Related Work

Answering a similarity query using an index typically involves two steps: a filtering step where the pre-built index on data series summaries is used to prune candidates, and a refinement step where the surviving candidates are compared to the query in the original high-dimensional space. Following this design, several tree-based techniques for efficient and scalable data series similarity search have been proposed in the literature [12, 13, 11]. Out of those, the iSAX-based indexes [42] have proven to be very competitive in terms of both index building and query answering time performance [12, 13]. This family of indexes also includes parallel and distributed solutions for similarity search that are in a much better position than traditional single-node techniques to address the scalability challenges of modern data series analytics applications that have to deal with very large data collections. These methods support both exact and approximate similarity search query answering, and make use of modern hardware (e.g., SIMD, multi-core, multi-socket, GPU), such as ParIS+ [46], MESSI [47], and SING [48], as well as distributed computation (e.g., Spark) such as DPiSAX [59, 60]. Other distributed solutions include TARDIS [61] and L-match [29], which, nevertheless, solve slightly different problems, i.e., exact-match (determine whether the query series appear *exactly* the same in the dataset) and subsequence matching (determine the most similar match between a short query series and a subsequence of a long series), respectively.

The first lock-free implementation of a concurrent search tree appears in [16]. We use the main ideas from this paper to come up with a baseline algorithm, which we discuss and experimentally compare with FreSh in Section 6. Many other non-blocking concurrent search trees have appeared in the literature (e.g., [6, 34, 2, 36, 40, 9, 5, 14, 28, 3]; this list is by no means exhaustive). The novelty of the tree implementation we present in Section 5 is that it allows multiple insert operations to concurrently update (in a lock-free way) the array that stores the data in a (fat) leaf. Additionally, it supports the expeditive-standard mode of execution. These innovations result in enhanced parallelism and excellent performance. Our algorithm is designed to only provide the functionality needed to implement traverse objects. The implementations above support different functionalities, have different goals or have been designed for other settings.

Concurrent priority queues appear in [1, 50, 58, 53, 54, 39]. Some of them are blocking, whereas others provide relaxed semantics. None of them is based on sorted arrays, and none provides support for different modes of execution. In the baseline lock-free implementations we developed, we use the skip-list based lock-free priority queue proposed in [39], which has been shown to have good performance. Our experiments show that the scheme of priority queues we propose and use in FreSh to implement the refinement stage, outperforms by far this implementation (Section 6).

Universal constructions [20, 21, 22, 23, 19, 24, 25, 15, 26] can be used to provide wait-free or non-blocking concurrent versions of any sequential data structure. However, because of their generality, they are usually less efficient than implementations tailor-made for specific sequential data structures. The algorithms in [20, 22, 26] are highly efficient for implementing shared objects of small size (such as stacks and queues), but they are not appropriate for our purpose.

The idea of transforming an algorithm to get an implementation that ensures a different progress guarantee is not new. Examples of such transformations appear in [55, 30, 33]. Although Refresh shares some high-level ideas with some of these algorithms, they have all been introduced to solve different problems. The main technique proposed in Refresh departs from

all these approaches.

3 Traverse Objects

In this section, we introduce the traverse object, an abstract data type, based on which we can design an iSAX-based index in a modular way. Each of the last three stages of an iSAX-based index processes data that are produced by the previous stage. The first stage processes the original collection of data. This processing pattern has inspired the definition of a traverse object.

Definition 1 *Let U be a universe of elements. A* traverse object *S stores elements of U (not necessarily distinct) and supports the following operations:*

- PUT$(S, e, param)$*, which adds an element $e \in U$ in S; param is an optional argument that allows an implementation to pass certain parameters in* PUT*.*
- TRAVERSE$(S, f, param, del)$*, which traverses S and applies the function f on each of the traversed elements. If the del flag is set, then each of the traversed elements is deleted from S.*

A traverse object satisfies the traversing property*:* TRAVERSE *applies f at least once on all distinct elements that have been added in S and have not yet been deleted, by the time* TRAVERSE *is invoked.*

To implement the four stages of an iSAX-based index, we require four instances of a traverse object, one for each stage. We call *BC*, *TP*, *PS*, and *RS*, the traverse objects we employ to implement the buffer creation, tree population, prunning, and refinement stages, respectively. To get an iSAX-based index, these traverse objects should be implemented using different data structures. The buffers creation phase uses an array *RawData* to store the raw data series, thus, the elements of *BC* are stored in *RawData*. The tree population phase uses a set of arrays (*summarization buffers*) where the pairs of iSAX summaries and pointers to data series are initially stored. *TP* stores these pairs. The prunning stage employs a leaf-oriented tree to store these pairs. Thus, *PS* also stores pairs but it organizes them into as many sets as the leaf nodes of the tree. Each set contains the pairs stored in each leaf. Finally, the refinement stage uses priority queues to store those tree leaves containing candidate series.

Answering a query using these traverse objects, is comprised of a sequence of four invocations of TRAVERSE on the different traverse objects. Figure 1 provides pseudocode for the implementation of an iSAX-based index using traverse objects. Multithreaded processing is hidden under the implementation of PUT and TRAVERSE.

The four stages of an iSAX-based index do not overlap with one another. This is usually ensured with the use of synchronization barriers. In the scheme of Figure 1, the barriers, if needed, should be incorporated in the implementation of PUT and TRAVERSE. Thus, an iSAX-based index satisfies the following property.

Definition 2 (Non-Overlapping Property) *In an iSAX-based index,* TRAVERSE *is performed only after the execution of all instances of* PUT *that add distinct elements in S have been completed.*

Assume that the non-overlapping property holds for *BC*, *TP*, *PS*, and *RS* and that *RawData* initially stores all raw data series in the collection. The traversing property implies that the BUFFERCREATION function is invoked at least once for each data series *ds* in *RawData*, so at least one appropriate pair is added for it in *TP*. Recall that *TP* is implemented using a

set of summarization buffers. These and the semantics of PUT imply that the summarization buffers are populated appropriately. By the non-overlapping and the traversing properties, TREEPOPULATION is invoked for all these pairs. Since TREEPOPULATION invokes PUT on PS, it follows that at least one pair for each of the data series of the collection is added in PS (i.e. in the tree index). By the traversing property, all elements of PS are traversed and PRUNNING is called on them. Thus, all tree leaves that cannot be pruned are added in RS. Note that TRAVERSE on RS is invoked with the *del* flag being `True`. This allows to use (one or more) priority queues for implementing RS, and to employ DELETEMIN to delete each traversed element during TRAVERSE. REFINEMENT will be applied on every traversed element of RS. Therefore, those leaves that cannot be pruned will be further processed by calculating real distances and for the data series they store, and by updating BSF whenever needed. Implementations for PUT and TRAVERSE for BC, TP, PS, and RS in FreSh are presented in Section 5.

4 Locality-Aware Lock-Freedom

Locality-awareness aims at capturing a collection of design principles (Definition 3) for data series indexes which are crucial for achieving good performance. A *locality-aware* implementation respects these principles. All iSAX-based indexes are locality-aware.

Definition 3 *Principles for* locality-aware *processing:*

1. **Data Locality.** *Separate the data into* disjoint sets *and have a distinct thread processing the data of each set. This results in reduced communication cost between the threads (i.e., reduced number of cache misses and branch misprediction).*

2. **High Parallelism & Low Synchronization Cost.** *Threads should work in parallel and independently from each other as much as possible. Whenever synchronization cannot be avoided, design mechanisms to minimize its cost.*

3. **Load Balancing.** *Share the workload equally to the different threads, thus avoiding load imbalances between threads and having . all threads busy at each point in time.*

In existing iSAX-based indexes, a thread operates on chunks of *RawData* and processes disjoint sets of summarization buffers and subtrees of the index tree. Also, an iSAX-based index employs several priority queues to store leaf nodes containing candidate series. Thus, iSAX-based indexes are *locality-aware*. Enuring locality awareness results in good performance and is thus a desirable property for big data processing. In what follows, we focus on a *blocking* locality-aware implementation $\mathcal{A}$, which splits its workload into disjoint parts and assigns them to different threads for processing.

We are now ready to present Refresh. Refresh transforms $\mathcal{A}$ into a *locality-aware* implementation $\mathcal{B}$ that achieves high parallelism *in a lock-free way*. Pseudocode for Refresh is presented in Algorithm 2. Let W be the workload to be processed by $\mathcal{A}$ and let $w_1, \ldots, w_k$ be the parts it is separated to ensure locality awareness (line 1). Refresh applies the following steps:

(1) It attaches a *flag variable* d_i, $1 \leq i \leq k$, (initially `False`) with each w_i to identify whether w_i's processing is done. As soon as a thread finishes processing w_i, it sets d_i to `True` (line 11).

(2) Threads in $\mathcal{B}$ execute the same algorithm as in $\mathcal{A}$ to acquire parts of W to process, until all parts have been acquired (lines 5-11). The thread that has acquired a workload is its *owner*.

(3) To achieve lock-freedom, t then *scans* all the flag variables to find those parts that (although acquired) are still unprocessed (line 12).

(4) Thread t *helps* by processing, one after the other, each part found unprocessed during scan. For each part w_i that t helps, it periodically checks d_i to determine whether other threads completed the processing of w_i. If this is so, t stops helping w_i (line 16). If t completes the processing of w_i, it changes d_i to true (line 17).

(5) Due to helping, each data structure D, employed in $\mathcal{A}$, may be concurrently accessed by many threads. Thus, $\mathcal{B}$ should provide an efficient lock-free implementation for all data structures employed in $\mathcal{A}$.

In locality-aware implementations, threads are expected to work on their own parts of the data most of the time (*contention-free phase*), and they may help other threads only for a small period of time at the end of their execution (*concurrent phase*). In the contention-free phase, Refresh avoids synchronization overheads incurred to ensure lock-freedom. Specifically, it employs two implementations for each data strucutre D of $\mathcal{A}$, one with low synchronization cost that does not support helping (*expeditive mode*), and another that supports helping and has higher synchronization overhead (*standard mode*). To enable threads operate on the appropriate mode, a *helping-indicator flag* h_i (initially `False`) is attached with each w_i. A thread t starts by processing its assigned workload on expeditive mode (lines 3 and 8-9). As soon as t starts helping some part w_i, it first sets h_i to `True` (line 15), so that the owner thread of w_i figures out that it now has to run on standard mode (line 9), and then it processes w_i on standard mode (line 16).

To avoid helping whenever it is not absolutely necessary, Refresh provides an optional *backoff scheme* that is used by every thread t (line 13) before it attempts to help other threads (line 14-16). In some cases, the small delay before switching to standard mode, introduced by this scheme, positively affects performance. To minimize the work performed by a helper, Refresh could be applied *recursively* by splitting each part w_i to subparts. This way, a helper thread helps only the remaining unprocessed subparts of w_i.

Lock-freedom is ensured due to the *helping code* (lines 12-17). In the absence of this code, threads' crashes could result in workloads that remain unprocessed. In Refresh, only after a thread t processes a workload w_i, it sets h_i to `True` (lines 11 and 17). Moreover, each thread t performs the helping code after finishing with their assigned workloads. Thus, when t completes the execution of Refresh, the processing of all parts of the workload has been completed. This implies that when a thread finishes executing Refresh, it may continue directly to the execution of the next stage, without having to wait for the other threads to complete the execution of the current stage. Therefore, these scheme renders the use of barriers useless. This is necesssary for achieving lock-freedom.

Theorem 1 *Refresh is a general scheme for processing a locality-aware workload in a lock-free way, without sacrificing locality-awareness.*

By inspection of the code Algorithm 1, we see that TRAVERSE often adds the elements of a traverse object to the traverse object of the next stage. Since helping may result in processing elements more than once, some elements may be added in a traverse object multiple times. This is why the definition of a TRAVERSE object S allows for elements to appear more than once in it. If the number of multiple instances becomes large, it may result in performance overhead. Experiments showed that in FreSh, this number is (on average) $5 \cdot 10^{-5}$ of the initial dataset size; this overhead is negligible.

5 FreSh

We follow the data processing flow described in Section 3 and employ Refresh to come up with FreSh, the first lock-free locality-aware data series index. FreSh employs the four traverse objects, *BC*, *TP*, *PS*, and *RS* (see Section 3), to implement the buffer creation, the tree population, the pruning and the refinement stages, respectively.

5.1 Buffers Creation and Tree Population

BC is implemented using a single buffer, called *RawData*. A number of worker threads process *RawData* by acquiring parts of it. In *BC*, PUT is never used, as we assume that the data are already in *RawData* when FreSh starts its execution. To implement TRAVERSE, we employ Refresh. We split *RawData* into k equally-sized *chunks* of consecutive elements to get k workloads. Threads use a counter object to get assigned chunks to process. To reduce the cost of helping, FreSh calls Refresh recursively. Specifically, it splits each chunk into smaller parts, called *groups*, and employ Refresh a second time for processing the groups of a chunk. In more detail, FreSh maintains an additional counter object for each chunk of *RawData*. Each thread t that acquires or helps a chunk, uses the counter object of the chunk to acquire *groups* in the chunk to process. FreSh also applies a third level of Refresh recursion, where each workload is comprised of the processing of just a single element of a group.

Pseudocode for *BC*.PUT and *BC*.TRAVERSE is provided in Algorithm 3. *RawData* is comprised of k chunks, each containing m groups. Moreover, each group contains r elements. FreSh uses three sets of done flags, *DChunks*, *DGroups*, and *DElements*, storing one done flag for each chunk, for each group, and for each element, respectively. Similarly, FreSh employs three sets of counter objects, *Chunks*, *Groups*, and *Elements*, to count the chunks, groups and elements, assigned to threads for processing. FreSh also employs two sets of *helping* flags, *HChunks* (for helping chunks) and *HGroups* (for helping groups). In an invocation of TRAVERSE(&BUFFERCREATION, *RawData*, *Dchunks*, *DGroups*, *DElements*, *HChunks*, *HGroups*, `False`, *Chunks*, *Groups*, *Elements*, 1), h is equal to `False`. By the way a counter object works, it follows that no expeditive mode is ever executed at the first level of the recursion. Note that at this level, the roles of D_1 and H_1 are played by the one-dimensional arrays *DChunks* and *HChunks*, respectively. Moreover, *DGroups* and *DElements* play the role of D_2 and D_3, respectively, and *HGroups* plays the role of H_2. Each chunk is processed by recursively calling TRAVERSE (*level-2 recursion*). The goal of a level-2 invocation of TRAVERSE is to process an entire chunk by splitting it into groups and calling TRAVERSE once more (*level-3 recursion*) to process the elements of each group. Note that in a level-2 invocation corresponding to some chunk i, *RawData* is the two-dimensional array containing the elements of the groups of chunk i. Moreover, the role of D_1 is now played by the one-dimensional array $DGroups[i]$, and the role of D_2 by the two-dimensional array $DElements[i]$, whereas D_3 is no longer needed and is *NULL*. The role of H_1 is now played by the one-dimensional array $HGroups[i]$.

FreSh implements the backoff scheme of Refresh in a way that the backoff time depends on the average execution time required by each thread to process a group. Specifically, each thread t counts the average time T_{avg} it has spent to process all the parts it acquired, and whenever it encounters a group to help, it sets the backoff time to be proportional to T_{avg} and performs helping only after backoff, if it is still needed.

FreSh implements *TP* using a set of 2^w summarization buffers (where w is the number of segments of an iSAX summary), one for each bit sequence of w bits. To decide to which summarization buffer to store a pair, FreSh examines the bit sequence consisting of the first bit of each of the w segments of the pair's iSAX summary, and places the pair into the corresponding sumarization buffer. Each of the summarization buffers is split into N parts, one for each of the N threads in the system. Each thread uses its own part in each buffer to store the elements it inserts.

To implement *TP*.TRAVERSE, we split the elements of *TP* into as many workloads as the number of summarization buffers, and apply Refresh. As in Algorithm 3, threads use a counter object to get assigned summarization buffers to process. Each summarization buffer could be further split into chunks and groups and Refresh could be called recursively. Pseudocode for TRAVERSE of *TP* closely follows that for *BC*, and is omitted due to lack of space.

By inspection of the pseudocode of routines BUFFERCREATION and TREEPOPULATION (Algorithm 3), and by Theorem 1, we get:

Lemma 1 *The following claims hold:*

(1) *BC and TP are lock-free implementations of a traverse object that supports* PUT *and* TRAVERSE*(∗, ∗, 0).*

(2) *For every thread t, when an invocation of* TRAVERSE*(&BufferCreation, ∗,* `False`*) by t on BC (TP) completes, the following hold. For every data series ds in RawData, a pair comprised of the iSAX summary of ds and an index to the position of ds in RawData, has been added in TP (PS).*

5.2 Prunning and Refinement

In FreSh, *PS* is implemented as a forest of 2^w leaf-oriented trees with fat leaves, one for each of the summarization buffers. The trees of the forest are the root subtrees of a standard iSAX-based tree. To support the concurrent population of a subtree by multiple threads, FreSh utilizes Algorithm 5 presented in Section 5.2.1 (and discussed later). To implement *PS*.TRAVERSE (Algorithm 4), FreSh uses Refresh to process the different subtrees of the index tree. Specifically, each thread t is assigned a subtree T to process, using *FAI*. To process the nodes of T, Refresh is applied recursively. Specifically, the threads working on T (owner and helpers) use a counter object to get assigned nodes in T. A thread t that is assigned node i of T to process, first searches for the i-th node in an inorder traversal of T, and then processes it by invoking the PRUNNING function, presented in Algorithm 1.

To find the i-th node of a subtree T in an efficient way, for each node nd of T, FreSh maintains a counter cnt_{nd} that counts the number of nodes in the left subtree of nd. FINDNODE (line 1d) uses these counters to find the i-th node of T by simply traversing a path in T. Function TOTALNODES calculates the total number of nodes in a subtree, by simply traversing the righmost path of T and summing up the counters stored in the traversed nodes.

Lemma 2 *The following claims hold:*

(1) *PS is a lock-free implementation of a traverse object that supports* PUT *and* TRAVERSE*(∗, ∗, 0).*

(2) *For every thread t, when an invocation of* TRAVERSE*(&*PRUNNING*, ∗,* `False`*) by t on PS completes, for every leaf ℓ in PS, if the lower bound distance of ℓ from the query series Q is smaller than BSF, ℓ has been added in RS.*

5.2.1 Insert in Leaf-Oriented Tree

Pseudocode is provided in Algorithm 5. Each node of the tree stores a key and the pointers (left and right) to its left and right children. A leaf node stores additionally an array D, where the leaf's data are stored. We assume that each data item is a pair containing a key and the associated information. Note that a node may have its own key. For instance, in iSAX-based indexes, this key is the node's iSAX summary.

The novelty of the proposed implementation is that it allows to multiple insert operations to concurrently update array D of a leaf. This results in enhanced parallelism and good performance. To achieve this, each leaf ℓ contains a counter object, called *Elements*. Each thread t that tries to insert data in ℓ, uses *Elements* to acquire a position *pos* in the array D of ℓ. If D is not full, t stores the new element in $D[pos]$. Otherwise, t attempts to split the leaf node. Note that during spliting, D may contain empty positions, since some threads may have acquired positions in D but have not yet stored their elements in it. To avoid situations of missing elements, each leaf contains an *Announe* array with one position for each thread. A thread announces its operation in the Announce array before it attempts to acquire a position in D. During spliting, a thread distributes to the new leaves it creates not only the elements found in D but also those in *Announce*.

More specifically, a thread t executing TreeInsert repeatedly (line 6) executes the following actions. It first calls a standard Search routine to traverse a path of the tree and find an appropriate *leaf* node and its *parent* (line 7). Pointer *ptr* is a reference to the appropriate child field of *parent* which needs to be changed to perform TreeInsert (lines 8-10). Next, t access the counter object to acquire a position in D (line 13) and proceeds to announce the data that it wants to insert in the tree (line 14). Afterwards, it announces this position in *Announce* and stores the data in $D[pos]$ (line 17), if D is not full (line 15). If D is full, it calls SplitLeaf to split *leaf*.

SplitLeaf first creates three new nodes (lines 24-26), an internal node and its two children which are leaves. Next, it collects the elements of *Announce* in *splitbuffer* (lines 29-32). It then distributes the keys of *leaf* to the newly created leaves (line 34) and performs a *CAS* on the appropriate child of ℓ's parent in an effort to replace ℓ with this subtree of three nodes. If this *CAS* succeeds, then the data have been added and TreeInsert completes. Otherwise, some other thread has successfully split the node.

To better understand the necessity of *Announce*, let's assume that *Announce* (and all code lines that refer to it) do not exist. Then, the following scenario may occur. A thread t that wants to insert its data in a leaf ℓ may acquire a position in D of ℓ, and before recording its data there, another thread t' may split ℓ. Then, t may continue and insert its data in ℓ, which however has already been replaced in the tree. Thus, the data of t are lost. Algorithm 5 addresses this problem as follows.

Thread t attempts to acquire a position in D (line 13). If it t sees that helpers have arrived, it announces its operation *op* in *Announce* (line 14). While t' splits l, it takes into account not only the data of *op* but also of all other insert operations already announced (lines 29-32). Moreover, t' marks *op* as applied by copying into the announce array of the new (internal) node *newNode* it creates, a non-$\perp$ position value for *op* (line 33). This allows t to determine that *op* has been applied, even if concurrent splitting is performed. Specifically, t uses *ptr* to re-read the appropriate child pointer of *parent* (line 20) and examines the *position* field in $Announce[t]$ of the node *parent* points to (line 20). If the *CAS* of line 35 is successfully executed by t, or another thread t' sees the data of *op* in *Announce* (or in D) and takes them into account while splitting, then *position* will not be $\perp$. This way t discovers that *op* has been applied and returns. If *position* is still $\perp$, t tries again (line 22).

If t's operation has been applied, t cleans its record in the announce array (line 21) to support additional insert operations on the same leaf node in the future.

We finally discuss the following subtle scenario. Assume that the owner thread t calls TreeInsert, reaches a leaf l, and acquires the last valid position in array D of l. Thread t executes in expeditive mode (so it does not announce its data), and before it records its data in D, it becomes slow. Next, a helper thread t' reaches l, switches l's execution mode to standard, and splits ℓ (executing on standard mode). Unfortunately, during this split, t' will not take into consideration the data of t, since t neither has announced its operation (since t was executing in expeditive mode), nor has yet written its data into D. To disallow thread t from finishing its

operation without inserting its data, TreeInsert provides the following mechanism (lines 19-22). Before it terminates, thread t re-reads the appropriate child field of the parent of ℓ (through *ptr*) and checks the *helpersExist* flag of the node *nd* that *ptr* points to, to figure out whether it can still operate on expeditive mode. In the scenario above, *nd* will be the node that t' has allocated to replace ℓ, and thus it has its *helpersExist* flag equal to `True` (line 24). This way, t discovers that the execution mode for l has changed (line 19 and first condition of line 20), and re-attempts its Insert (line 22).

Lemma 3 *Algorithm 5 is a* linearizable, lock-free *implementation of a leaf-oriented tree with fat leaves, which supports only insert operations.*

5.3 Refinement

To implement *RS*, FreSh uses a set of priorities queues each implemented using an array, as shown in Algorithm 6. In FreSh, a thread inserts elements in all arrays in a round-robin fashion. This technique results in almost equally-sized arrays, which is crucial for achieving load-balancing.

To implement *RS*.Traverse, FreSh first calls InitDeletePhase (Algorithm 6) on each of the arrays, to come up with sorted versions of them, shared to all threads. Then, it uses Refresh to assign sorted arrays to threads for processing. To process the elements of a sorted array *SA*, Refresh is applied recursively. Specifically, the threads working on *SA* use a counter object to get assigned elements of *SA*. Processing of an element is performed by invoking the Refinement function (Algorithm 1). Helping is done at the level of 1) each individual priority queue and 2) the set of priority queues, in a way similar to that in *PS*. Pseudocode for implementing Put and Traverse of *RS* resemles that of *PS* and is omitted.

For the implementation of UpdateBSF, FreSh uses a *CAS* object O. If a thread calls UpdateBSF with value x, it repeatedly reads the current value y of O, and attempts to atomically change it from y to x using *CAS*, until it either succeeds or some value smaller than or equal to x is written in BSF.

Lemma 4 *The following claims hold:*

(1) *RS is a linearizable lock-free implementation of a traverse object that supports* Put *and* Traverse$(*, *, 0)$.

(2) *For every thread t, when an invocation of* Traverse$(\&$Prunning$, *,$ `True`$)$ *by t on RS completes, for every leaf ℓ in RS, ℓ either has been processed or it has been pruned.*

Lemmas 1, 2 and 4 imply the following.

Theorem 2 *It holds that:*

(1) *FreSh solves the 1-NN problem.*

(2) *FreSh provides a lock-free implementation of* QueryAnswering *(Algorithm 1).*

6 Experimental Evaluation

Setup. We used a 40-core machine equipped with 4 Intel(R) Xeon(R) E5-4610 v3 1.7Ghz CPUs with 10 cores each, and 25MB L3 cache. The machine runs CentOS Linux 7.9.2009 with kernel version 3.10.0-1160.45.1.el7.x86_64 and has 256GB of RAM. Code is written in C and compiled using the gcc compiler (version 11.2.1) with O2 optimizations.

Datasets We evaluated FreSh and the competing algorithms (remember that they are all in-memory algorithms) using both real and synthetic datasets. The synthetic data series, called *Random*, were generated as random-walks (i.e., cumulative sums) of steps that follow a Gaussian distribution (0,1). This type of data has been extensively used in the past [18, 8, 63, 62, 12, 13], and models the distribution of stock market prices [18]. Our real datasets come from the domains of seismology and astronomy. The seismic dataset, *Seismic*, was obtained from the IRIS Seismic Data Access archive [31]. It contains seismic instrument recordings from thousands of stations worldwide and consists of 100 million data series of size 256, i.e. its size is 100GB. The astronomy dataset, *Astro*, represents celestial objects and was obtained from [52]. The dataset consists of 270 million data series of size 256, i.e. its size is 265GB. Since the main memory of our machine is limited to 256GB, we only use the first 200GB of the Astro dataset in our experiments. In the evaluation, we report results for 50GB, 100GB, 150GB, and 200GB for Random and Astro, and 50GB and 100GB for Seismic.

Evaluation Measures. During each experiment, we measure (i) the *summarization time* required to calculate the data-series iSAX summarization and fill-in the receive buffers, (ii) the *tree time* required to insert the items of the receive buffers in the tree-index, and (iii) the *query answering time* required to answer 100 out-of-dataset queries (i.e., queries that belong to the same dataset, but are not part of the index). The above times (all together) constitute the *total time*. Each experiment is repeated 5 times, and the average of each measure is reported. All algorithms return in all situations the exact, correct results.

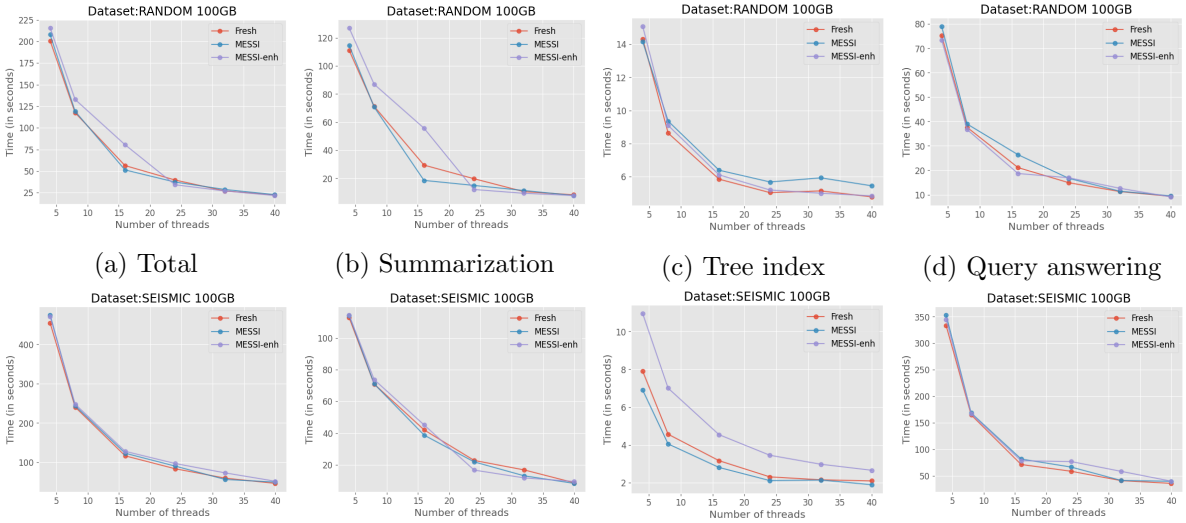
6.1 Results

FreSh vs MESSI. We compare FreSh against MESSI, which is the state-of-the-art in-memory data series indexing algorithm. Recall that MESSI is a blocking algorithm. To enable fair comparison, the MESSI implementation we use is an optimized version of the original MESSI, where we have applied all the code enhancements incorporated by FreSh.

Additionally, we compare FreSh against an extended version of MESSI, called MESSI-enh, that allows several threads to concurrently populate the same sub-tree, during the tree creation phase, instead of using a single thread to populate each subtree (as MESSI does). Specifically, similarly to FreSh, after a thread has finished creating all the subtrees it acquired, instead of remaining idle waiting the remaining threads to also finish this phase, it continues by scanning and helping non-completed subtrees to be created. This is implemented in a blocking way, using fine-grained locks that are attached on each of the leaf nodes of the sub-trees. MESSI-enh allows to compare the lock-free index creation phase of FreSh against a blocking one.

Figure 2 shows that all algorithms (FreSh, MESSI, and MESSI-enh) continue scaling as the number of threads is increasing, for Random 100GB and Seismic 100GB (top and bottom row of Figure 2, respectively). This is true for all three phases. Moreover, the total execution time of FreSh (Figures 2a and 2e) is almost the same as the total execution time of all its competitors, although it is the only lock-free approach. As expected, the tree index creation time of FreSh is smaller than MESSI's for the Random dataset (Figures 2c), since FreSh allows subtrees to be populated concurrently by multiple threads, thus it allows parallelism during this phase, in contrast to MESSI. Interestingly, FreSh achieves better performance than MESSI-enh, in most cases. On the other hand, FreSh performs slightly worse than MESSI for Seismic (Figure 2g), since the tree index of this dataset does not favor parallelism when multiple workers are working on the same subtree, but this rather results on increased contention. This becomes obvious when comparing MESSI with MESSI-enh. In any case, this performance gap between FreSh and MESSI becomes negligible as the number of threads increases.

Considering scalability as the size of the dataset increases, Figure 3 demonstrates that FreSh scales well on all datasets (Random, Astro, and Seismic). Actually, in most cases, FreSh outperforms (is faster than) MESSI.



(a) Total (b) Summarization (c) Tree index (d) Query answering

(e) Total (f) Summarization (g) Tree index (h) Query answering

Figure 2: Comparison of FreSh against MESSI and MESSI-enh: (a)-(d) on 100GB Random and (e)-(h) on 100GB Seismic.

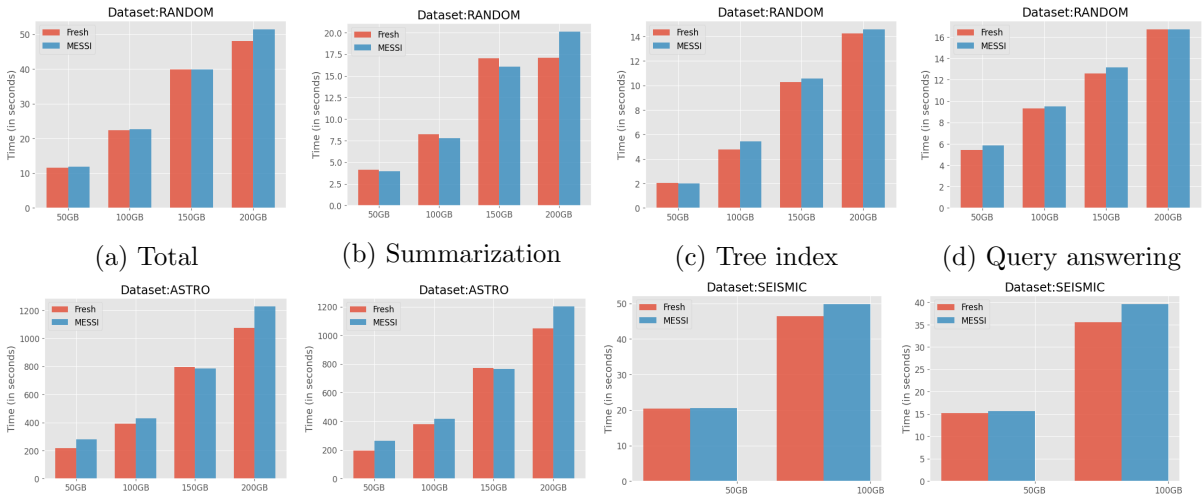
We have also studied the performance of FreSh when answering query workloads of increasing difficulty. Following previous works [62, 47], we also conducted experiments with query workloads of increasing difficulty. For these workloads, we select series at random from the collection, add to each point Gaussian noise ($\mu = 0$, $sigma = 0.01 - 0.1$), and use these as our queries. Figure 4 presents the results for the Seismic dataset, where FreSh performs better than MESSI in most cases.

FreSh vs Baselines. In the following, we compare FreSh against several baseline lock-free implementations of the different stages of an iSAX-based index. Our results, presented in Figure 5, including Random, Seismic, and Astro, with size 100GB, show that FreSh performs better than all these implementations.

Summarization Baseline: For buffer creation, we have experimented with four implementations, called DoAll, DoAll-Split, FI-Based, and CAS-Based. All algorithms use a single summarization buffer which has as many elements as RawData. The iSAX summary of the data series stored in the i-th position of RawData is stored in the i-th position of the summarization buffer. In DoAll, each thread t traverses RawData and for every traversed data series, it calculates its iSAX summary and stores it in the summarization buffer. As soon as these actions have occured, we say that the data series has been processed. Since the performance of DoAll during summarization phase was almost 5x worse than the next worse algorithm, we avoid including it in our results (Figure 5) to enable easier comparison of the remaining algorithms.

DoAll-Splitis an optimized version of DoAll that splits RawData into N equaly-sized chunks (where N is the number of threads). Each thread traverses RawData starting from the first element of its assigned chunk. Thus, if each chunk has size m, thread i will start from position $i * m$ and the last position to examine will be position $i * m - 1$. DoAll-split stores a flag with each data series. The flag is set after the data series has been processed. For each data series it traverses, a thread first checks whether its flag is set, and if not it processes the data series. Otherwise, it simply proceeds to the next data series. The thread stops when it has traversed the entire array (which is now handled as if it is circular).

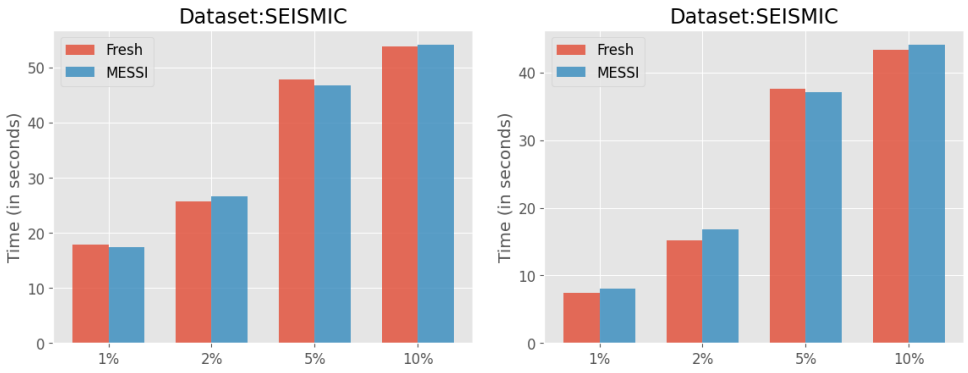
To avoid having all threads calculating all iSAX summaries, FI-Based uses a *FAI* object O with initial value 0. Each thread t repeatedly performs the following: It executes a *FAI* on O to get a position v of RawData, and then process the data series stored in this position. To achieve lock-freedom, FI-Based stores in RawData, a boolean flag (initally false) with each data



(a) Total (b) Summarization (c) Tree index (d) Query answering

(e) Total (f) Query answering (g) Total (h) Query answering

Figure 3: Comparison of FreSh against MESSI: (a-d) on Random with size 50GB, 100GB, 150GB, and 200GB, (e)-(f) on Astro with size 50GB, 100GB, 150GB, and 200GB, and (g)-(h) on Seismic with size 50GB and 100GB, for 40 threads.



(a) Total (b) Query answering

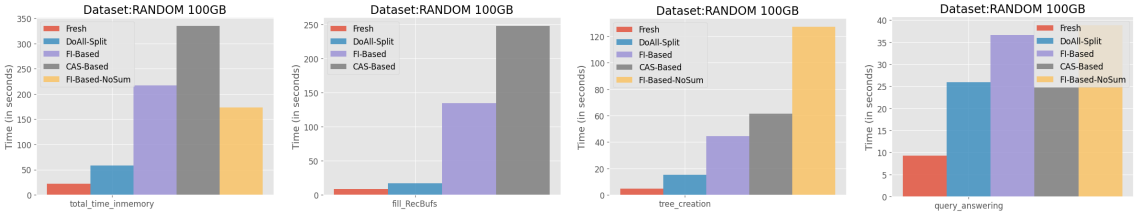
Figure 4: Comparison of FreSh against MESSI on Seismic 100GB with variable queries difficulty, where an increasing percentage of noise is added to the original queries.

series. This flag, which we call *done*, is set to true after the data series has been processed and its iSAX summary has been stored in the summarization buffer. When a thread figures out that all RawData elements have been assigned for processing (i.e., *FAI* returns a value higher than the number of elements of RawData), it re-traverses RawData to identify data series whose flag is still false, and processes them. CAS-Based works similarly to FI-Based, while it uses *CAS* instructions, instead of *FAI*.

Figure 5b shows that FreSh performs significantly better than all these implementations, during its summarization phase.

Tree Population Baseline: To populate the tree, each thread is assigned elements of the summarization buffer using *FAI* and inserts them in the index tree. To achieve lock-freedom in traversing the summarization buffer, we apply the same DoAll-Split, FI-Based, and CAS-Based techniques we describe above (but with the summarization buffer playing the role of RawData).

To achieve lock-freedom in accessing the tree, we borrow the flagging technique employed by Ellen *et al.* [17]. A thread calls a search routine to traverse a path of the tree and reach an appropriate leaf node l. Then, it flags the parent of l using *CAS*. If the flagging is successful, the insert is performed. Then, l's parent is unflagged (using *CAS*). Flagging stores a pointer



(a) Total (b) Summarization (c) Tree index (d) Query answering

Figure 5: Comparison of FreSh against baseline implementations on Random 100GB.

to an info record in the flagged node. Other threads may use this info record to help the insert complete.

We have also experimented with FI-Based-NoSum, a lock-free implementation that avoids using the summarization buffers and inserts directly iSAX summaries in the index tree, by applying the FI-Based technique on RawData.

Figure 5c shows again that FreSh performs significantly better than all these implementations, for tree index creations.

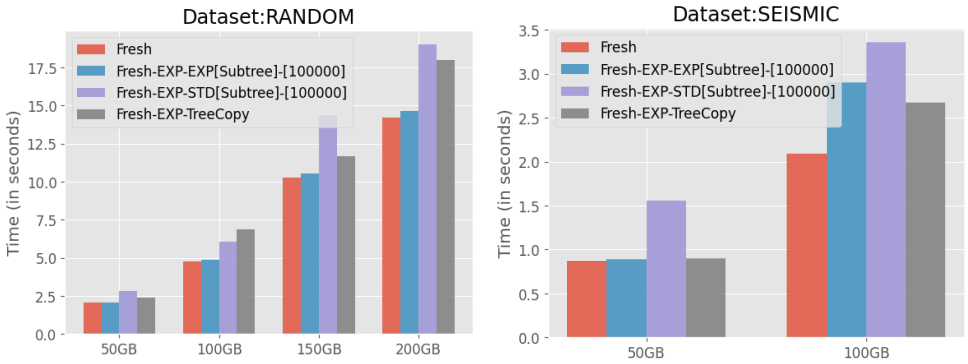
Pruning Baseline: All baselines use a single instance of an existing skip-based lock-free priority queue [39] to store the candidate data series. Threads uses *FAI* to find the next node to process in the index tree. A thread t that has been assigned the i-th tree node, first executes a search to find this node. To do so in $O(h)$ time (where h is the height of the tree), we maintain a counter in each node nd, counting how many nodes are contained in the left subtree of nd. Using these counters, we can fast navigate to the i-th node, by traversing a path of the tree.

When t reaches the i-th node, it checks whether the node can be pruned. If it cannot be pruned and the node is a leaf, the thread appends it in the priority queue. In either case the node is marked as processed. To ensure lock-freedom, when a thread t discovers that all nodes of the tree have been assigned for processing, it re-traverses the tree to examine whether there are any nodes that are still unprocessed and if there are such nodes, it processes them. A flag is maintained for each tree node to indicate whether its processing has been completed. This flag is set when the node is marked as processed. During re-traversal, t examines the flag of each element it visits, and does not process it if its flag is set (i.e., if the node is marked as processed).

Refinement Baseline: All threads, repeatedly call DeleteMin, on the priority queue to remove elements from the queue and calculate their real distance computation. This simple algorithm is not lock-free as a thread may crash after it has deleted an element from the queue. In this case, the deleted leaf will not be processed. This problem can be easily fixed but that would increase the cost of DeleteMin. Experiments show that even the non lock-free simple technique above is quite costly in comparison to our approach.

Figure 5d shows again that FreSh performs significantly better than all these implementations, for query answering time, that includes pruning and refinement.

Performance breakdown for index creation phase. In this section we evaluate the techniques incorporated by FreSh to create its tree index by comparing it against three modified versions of it. Recall that in FreSh each thread populates each of the subtrees it acquires in expeditive mode, as long as no helper reaches the same leaf of the tree; when this happens it changes its execution mode to standard. So, FreSh allows leaves of the same subtree to be processed in different modes of execution. In the first modified version, called Subtree, threads start again by populating a subtree in expeditive mode, while they change to standard mode as long as a helper reaches this subtree (and not when it reaches one each leaves, as FreSh does); so, in Subtree all the leaves of a subtree are executed in a single mode at each point in time. In the second modified version, called Standard, threads populate subtrees using only the standard execution mode; i.e., there is no expeditive mode. In the third modified implementation, called



(a) Random

(b) Seismic

Figure 6: Comparison of FreSh tree index creation against other tree implementations: (a) on Random of size 50GB, 100GB, 150GB, and 200GB, and (b) on Seismic of size 50GB and 100GB.

TreeCopy, a thread t first populates a private copy of the subtree (i.e. one that is accessible only to t) and only after its creation finishes, t tries to make it the (single) shared version of this subtree (by atomically changing a pointer using a *CAS* instruction); threads help each other by following the same procedure.

Figure 6 compares FreSh against the modified versions on Random and Seismic with variable dataset sizes and shows that it performs better than them, in all cases. Interestingly, for Seismic 50GB FreSh performs similarly to TreeCopy. Recall that each thread works on its own private copy and, on each subtree, they contend at most once on the corresponding *CAS* object. So, TreeCopy both restricts parallelism and minimizes the synchronization cost, which are properties that provide an advantage on Seismic, as already discussed (Figure 2g).

7 Conclusions and Discussion

Processing big collections of data series is of paramount importance for a wide range of applications, which need support for efficient and scalable similarity search. Current state-of-the-art data series indexes exploit the parallelism supported by modern multicore machines. Yet, their design is lock-based, which means that they cannot tolerate failures. In this work, we present FreSh, the first *lock-free* (thus, highly *fault-tolerant*) data series index. Moreover, we describe Refresh, a generic approach for designing, building and analyzing highly-efficient data series indexes in a modular way, and for supporting lock-freedom, which can be applied on top of any locality-aware data series algorithm. FreSh was built using the Refresh principles, and the experimental evaluation demonstrates that it performs as good as the state-of-the-art *blocking* index, thus, paying no penalty for providing the desirable fault-tolerance.

In a related avenue, the work in [26, 3] focuses on synchronization primitives and data structures for settings that support more general types of faults where a thread may recover after a crash. Such settings are met e.g., in machines that support Non-Volatile Memory (NVM).

In [26], the power of software combining [20, 21, 22, 23] in achieving recoverable synchronization and designing persistent data structures is studied. *Software combining* is a general synchronization approach, which attempts to simulate the ideal world when executing *synchronization requests* (i.e., requests that must be executed in mutual exclusion). A single thread, called the *combiner*, executes all active requests, while the rest of the threads are waiting for the combiner to notify them that their requests have been applied. *Software combining* significantly decreases the synchronization cost and outperforms many other synchronization techniques in various cases.

Three persistence principles are identified that are crucial for performance [26]. An algo-

rithm's designer has to take into consideration these principles when designing highly-efficient recoverable synchronization protocols or data structures. The paper [26] illustrates how to make the appropriate design decisions in all stages of devising recoverable combining protocols to respect these principles. Specifically, it proposes recoverable software combining protocols that are many times faster and have much lower persistence cost than a large collection of existing persistent techniques for achieving scalable synchronization. It also provides fundamental recoverable data structures, such as stacks and queues, based on these protocols that outperform *by far* existing recoverable implementations of such data structures, and the first recoverable implementation of a concurrent heap and present experiments to show that it has good performance when the size of the heap is not very large.

To cope with more complicated data structures and objects of large size, the work in [3] presents a generic approach for deriving *detectably recoverable* implementations of many widely-used concurrent data structures. Such implementations are appealing for emerging systems featuring byte-addressable *non-volatile memory* (*NVM*), whose persistence allows to efficiently resurrect failed threads after crashes. Detectable recovery ensures that after a crash, every executed operation is able to recover and return a correct response, and that the state of the data structure is not corrupted. The proposed approach, called *Tracking*, amends descriptor objects used in existing lock-free helping schemes with additional fields that track an operation's progress towards completion and persists these fields in order to ensure detectable recovery. Tracking avoids full-fledged logging and tracks the progress of concurrent operations in a *per-thread* manner, thus reducing the cost of ensuring detectable recovery. Tracking has been applied to derive detectably recoverable implementations of a linked list, a binary search tree, and an exchanger.

Acknowledgments: This work has been done while P. Fatourou was working at the LIPADE, Université Paris Cité, as an MSCA Individual Fellow in the context of the PLATON project (MSCA grant agreement #101031688). The work was also partially supported by the FMJH Program PGMO in conjunction with EDF-THALES.

References

[1] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 11–20, New York, NY, USA, 2015. Association for Computing Machinery.

[2] H. Attiya, O. Ben-Baruch, P. Fatourou, D. Hendler, and E. Kosmas. Tracking in order to recover: Dectable recovery of lock-free data structures. In *Proc. 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 503–505, 2020.

[3] H. Attiya, O. Ben-Baruch, P. Fatourou, D. Hendler, and E. Kosmas. Detectable recovery of lock-free data structures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 262–277, New York, NY, USA, 2022. Association for Computing Machinery.

[4] A. J. Bagnall, R. L. Cole, T. Palpanas, and K. Zoumpatianos. Data series management (dagstuhl seminar 19282). *Dagstuhl Reports*, 9(7), 2019.

[5] A. Braginsky and E. Petrank. A lock-free B+tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 58–67, 2012.

[6] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *Proc. 19th ACM Symposium on Principles and Practice of Parallel Programming*, pages 329–342, 2014.

[7] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh. Beyond One Billion Time Series: Indexing and Mining Very Large Time Series Collections with iSAX2+. *KAIS*, 39(1), 2014.

[8] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with isax2+. *Knowl. Inf. Syst.*, 39(1):123–151, 2014.

[9] B. Chatterjee, N. Nguyen, and P. Tsigas. Efficient lock-free binary search trees. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing*, pages 322–331, 2014.

[10] K. Echihabi, P. Fatourou, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. Hercules Against Data Series Similarity Search. *PVLDB*, 2022.

[11] K. Echihabi, K. Zoumpatianos, and T. Palpanas. Big sequence management: Scaling up and out. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT*, pages 714–717, 2021.

[12] 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. *PVLDB*, 12(2), 2018.

[13] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. Return of the lernaean hydra: Experimental evaluation of data series approximate similarity search. *Proc. VLDB Endow.*, 13(3):403–420, 2019.

[14] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert. The amortized complexity of non-blocking binary search trees. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing*, pages 332–340, 2014.

[15] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. *Distributed Computing*, 29:251–277, 2016.

[16] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010.

[17] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 131–140, 2010.

[18] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, New York, NY, USA, 1994. ACM.

[19] P. Fatourou and N. D. Kallimanis. The RedBlue Adaptive Universal Constructions. pages 127–141, 2009.

[20] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 325–334, New York, NY, USA, 2011. Association for Computing Machinery.

[21] P. Fatourou and N. D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–626, New York, NY, USA, 2012. Association for Computing Machinery.

[22] P. Fatourou and N. D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55(3):475–520, Oct. 2014.

[23] P. Fatourou and N. D. Kallimanis. Lock oscillation: Boosting the performance of concurrent data structures. In *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[24] P. Fatourou and N. D. Kallimanis. The RedBlue family of universal constructions. *Distributed Computing*, 33:485–513, 2020.

[25] P. Fatourou, N. D. Kallimanis, and E. Kanellou. An efficient universal construction for large objects. In *22nd International Conference on Principles of Distributed Systems (OPODIS'18)*, pages 18:1–18:15, 2018.

[26] P. Fatourou, N. D. Kallimanis, and E. Kosmas. The performance power of software combining in persistence. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 337–352, New York, NY, USA, 2022. Association for Computing Machinery.

[27] P. Fatourou, N. D. Kallimanis, and T. Ropars. An efficient wait-free resizable hash table. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 111–120, New York, NY, USA, 2018. Association for Computing Machinery.

[28] P. Fatourou and E. Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. *CoRR*, abs/1805.04779, 2018.

[29] K. Feng, P. Wang, J. Wu, and W. Wang. L-match: A lightweight and effective subsequence matching approach. *IEEE Access*, 8:71572–71583, 2020.

[30] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free algorithms can be practically wait-free. In P. Fraigniaud, editor, *Distributed Computing*, pages 78–92, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[31] I. R. I. for Seismology with Artificial Intelligence. Seismic Data Access. `http://ds.iris.edu/data/access/`, 2018.

[32] K. Fraser. Practical lock-freedom. Technical Report 579, University of Cambridge Computer Laboratory, 2004.

[33] R. Guerraoui, M. Kapałka, and P. Kouznetsov. The weakest failure detectors to boost obstruction-freedom. In S. Dolev, editor, *Distributed Computing*, pages 399–412, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[34] M. He and M. Li. Deletion without rebalancing in non-blocking binary search trees. In *Proc. 20th International Conference on Principles of Distributed Systems*, pages 34:1–34:17, 2016.

[35] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[36] S. V. Howley and J. Jones. A non-blocking internal binary search tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 161–171, 2012.

[37] R. Izadpanah, S. Feldman, and D. Dechev. A methodology for performance analysis of non-blocking algorithms using hardware and software metrics. In *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 43–52, 2016.

[38] E. J. Keogh, K. Chakrabarti, M. J. Pazzani, and S. Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowl. Inf. Syst.*, 3(3):263–286, 2001.

[39] J. Lindén and B. Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In R. Baldoni, N. Nisse, and M. van Steen, editors, *Principles of Distributed Systems*, pages 206–220, Cham, 2013. Springer International Publishing.

[40] A. Natarajan, A. Ramachandran, and N. Mittal. FEAST: a lightweight lock-free concurrent binary search tree. *ACM Transactions on Parallel Computing*, 7(2), May 2020.

[41] T. Palpanas. Data series management: The road to big sequence analytics. *SIGMOD Record*, 2015.

[42] T. Palpanas. Evolution of a Data Series Index - The iSAX Family of Data Series Indexes. In *Communications in Computer and Information Science (CCIS)*, volume 1197, 2020.

[43] T. Palpanas and V. Beckmann. Report on the first and second interdisciplinary time series analysis workshop (ITISA). *SIGREC*, 48(3), 2019.

[44] B. Peng, P. Fatourou, and T. Palpanas. Paris: The next destination for fast data series indexing and query answering. *IEEE BigData*, 2018.

[45] B. Peng, P. Fatourou, and T. Palpanas. Messi: In-memory data series indexing. In *ICDE*, 2020.

[46] B. Peng, P. Fatourou, and T. Palpanas. Paris+: Data series indexing on multi-core architectures. *TKDE*, 2020.

[47] B. Peng, P. Fatourou, and T. Palpanas. Fast data series indexing for in-memory data. *The VLDB Journal*, 30(6):1041–1067, nov 2021.

[48] B. Peng, P. Fatourou, and T. Palpanas. Sing: Sequence indexing using gpus. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1883–1888, 2021.

[49] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *SIGKDD*, 2012.

[50] A. Rukundo and P. Tsigas. Tslqueue: An efficient lock-free design for priority queues. In L. Sousa, N. Roma, and P. Tomás, editors, *Euro-Par 2021: Parallel Processing*, pages 385–401, Cham, 2021. Springer International Publishing.

[51] J. Shieh and E. Keogh. iSAX: Indexing and Mining Terabyte Sized Time Series. In *SIGKDD*, 2008.

[52] S. Soldi, V. Beckmann, W. Baumgartner, G. Ponti, C. R. Shrader, P. Lubiński, H. Krimm, F. Mattana, and J. Tueller. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics*, 563:A57, 2014.

[53] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[54] O. Tamir, A. Morrison, and N. Rinetzky. A Heap-Based Concurrent Priority Queue with Mutable Priorities for Faster Parallel Algorithms. In E. Anceaume, C. Cachin, and M. Potop-Butucaru, editors, *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*, volume 46 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–16, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[55] S. Timnat and E. Petrank. A practical wait-free simulation for lock-free data structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 357–368, New York, NY, USA, 2014. Association for Computing Machinery.

[56] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. *VLDB*, 2013.

[57] A. D. Williams. *C++ Concurrency in Action: Practical Multithreading.* Manning Publications, 2012.

[58] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas. The lock-free k-lsm relaxed priority queue. *SIGPLAN Not.*, 50(8):277–278, jan 2015.

[59] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. DPiSAX: Massively Distributed Partitioned iSAX. In *ICDM*, pages 1135–1140, 2017.

[60] D.-E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. Massively distributed time series indexing and querying. *TKDE*, 32(1), 2020.

[61] L. Zhang, N. Alghamdi, M. Y. Eltabakh, and E. A. Rundensteiner. Tardis: Distributed indexing framework for big time series data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1202–1213. IEEE, 2019.

[62] K. Zoumpatianos, Y. Lou, I. Ileana, T. Palpanas, and J. Gehrke. Generating data series query workloads. *VLDB J.*, 2018.

[63] K. Zoumpatianos, Y. Lou, T. Palpanas, and J. Gehrke. Query workloads for data series indexes. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 1603–1612, 2015.

Algorithm 1: Implementation of an iSAX-based index using the traverse objects BC, TP, PS, RS.

```
   ▷ Shared objects:
1  TraverseObject BC, initially containing all raw data series
2  TraverseObjects TP, PS, RS, initially empty
3  int BSF

   ▷ Code for thread t_i, i ∈ {0, ..., n − 1}:
   Procedure QueryAnswering(QuerySeriesSet SQ): returns int
4      BC.Traverse(&BufferCreation(), BCParam, False)
5      TP.Traverse(&TreePopulation(), TPParam, False)
6      PS.Traverse(&Prunning(), PSParam, False)
7      RS.Traverse(&Refinement(), RSParam, True)
8      return BSF

   Procedure BufferCreation(DataSeries ds)
9      iSAXSummary iSAX := Calculate the iSAX summary for ds
10     Index bind := index to appropriate buffer based on iSAX
11     TP.Put(⟨iSAX, index of ds ⟩, bind)

   Procedure TreePopulation(Summary iSAX, Index ind, Index bind, Boolean flag)
12     PS.Put(⟨iSAX, ind⟩, bind, flag)

   Procedure Prunning(DataSeries Q, DataSeriesSet E, Boolean flag): returns boolean
13     iSAXSummary iSAX := Calculate the iSAX summary for E
14     int lbDist := lower bound distance between iSAX and Q
15     if lbDist < BSF then
16         RS.Put(⟨E, iSAX⟩, flag)
17         return TRUE
18     return FALSE

   Procedure Refinement(DataSeries Q, DataSeriesSet E, Summary iSAX,
   Function *UpdateBSF): returns Boolean
19     int lbDist, rDist
20     lbDist := lower bound distance between iSAX and Q
21     if lbDist < BSF then
22         for each pair ⟨iSAX_ds, ind_ds⟩ in E do
23             lbDist := lower bound distance between iSAX_ds and Q
24             if lbDist < BSF then
25                 rDist := real distance between ds and Q
26                 if rDist < BSF then
27                     *UpdateBSF(BSF, rDist) ▷ user-provided routine
28         return True
29     else return False
```

Algorithm 2: Refresh- A general approach for transforming a blocking data structure D of a big-data application $\mathcal{A}$ into a lock-free one.

```
   ▷ Shared variables:
1  workload part W := [w_1, w_2, ..., w_k]
2  boolean F := [d_1, d_2, ..., d_k], initially d_i = False, 1 ≤ i ≤ k
3  boolean H := [h_1, h_2, ..., h_k], initially h_i = False, 1 ≤ i ≤ k

   ▷ Code for each thread:
   Procedure Refresh()
4      // acquire and process parts of W
5      while W has available parts do
6          w_i := acquire an available part of W
7          mark w_i as acquired
8          if h_i = False then
9              process w_i in expeditive mode, while checking that h_i remains False; in case h_i = True,
               switch to standard mode
10         else process w_i in standard mode
11         d_i := True
       // scan flags for unprocessed parts of W and help
12     for each d_i ∈ D with d_i = False do
13         Backoff()                                          // avoid helping, if possible
14         if d_i = False then
15             h_i := True
16             process w_i in standard mode, while periodically checking that d_i remains False; in case
               d_i = True, stop processing w_i
17             d_i := True
```

Algorithm 3: Pseudocode for TRAVERSE of BC in FreSh. Code for thread t.

```
▷ Shared variables:
1  Set RawData[1..k][1..m][1..r], initially containing all data series
2  boolean DChunks[1..k], DGroups[1..k][1..m],
       DElements[1..k][1..m][1..r], initially all False
3  boolean HChunks[1..k], HGroups[1..k][1..m], initially all False
4  CounterObject Chunks, Groups[1..k], Elements[1..k][1..m]
5  int Size[1..3] = {k, m, r}

   Procedure TRAVERSE(Function *BufferCreation, DataSeries RawData[], Boolen D1[], Boolean D2[],
   Boolean D3[], Boolean H1[], Boolean H2[], Boolean h, CounterObject Cnt1, CounterObject Cnt2[],
   CounterObject Cnt3[], int rlevel)
6      int i
       // acquire and process parts of W
7      while True do
8          ⟨i, ∗⟩ := Cnt1.NEXTINDEX(&h)
9          if i > Size[rlevel] then break
10         mark RawData[i] as acquired
11         if rlevel < 3 then TRAVERSE(BufferCreation, RawData[i], D2[i], D3[i], D3[i],
               H2[i], NULL, H1[i], Cnt2[i], Cnt3[i], Cnt3[i], rlevel + 1)
12         else *BUFFERCREATION(RawData[i])
13
14         D1[i] := True
       // scan flags for unprocessed parts of W and help
15     for each j such that D1[j] is equal to False do
16         Backoff()                                          // avoid helping, if possible
17         if D1[j]] = False then
18             H1[j] := True
19             if rlevel < 3 then TRAVERSE(BufferCreation, RawData[j], D2[j], D3[j], D3[j],
                   H2[j], NULL, H1[j], Cnt2[j], Cnt3[j], Cnt3[j], rlevel + 1)
20             else *BUFFERCREATION(RawData[j])
21             D1[j] := True
```

Algorithm 4: Pseudocode for PUT and TRAVERSE of *PS* in FreSh. Code for thread $t \in \{1, \ldots, N-1\}$.

```
   ▷ Shared variables:
1  TreeNode *IndexTree[1..2^w]
2  boolean DTree[1..2^w], HTree[1..2^w], initially all False
3  CounterObject TreeCnt[1..2^w]

   Procedure TRAVERSE(Function *PRUNNING, TreeNode *T, CounterObject *Cnt, int x, Boolean h,
   int rlevel)
4      int i
5      while True do
6          ⟨i, ∗⟩ = Cnt.NEXTINDEX(&h)
7          if i > x then break
8          if rlevel < 2 then
9              mark IndexTree[i] as acquired
10             totalNds := TOTALNODES(IndexTree[i])
11             TRAVERSE(Prunning, IndexTree[i], TreeCnt[i],
                       totalNds, False, rlevel + 1)
12             DTree[i] := True
13         else
14             nd := FINDNODE(T, i)
15             mark nd as acquired
16             *PRUNNING(nd)
17             mark nd as done
18     for each j such that DTree[j] is equal to False do
19         Backoff()                                   // avoid helping, if possible
21         if DTree[j]] = False then
23             HTree[j] := True
25             HELPTREE(PRUNNING, IndexTree[j])
27         DTree[j] := True

   Procedure FINDNODE(TreeNode *T, int i): returns TreeNode*
28     TreeNode *p = T
29     int nds = 0
30     while p ≠ NULL and nds ≠ i do
31         if nds + p → cnt + 1 < i then
32             nds = nds + p → cnt + 1
33             p = p → rc
34         else p = p → lc
35     return p

   Procedure HELPTREE(Function *f, TreeNode *T)
36     if T == NULL then return
37     HELPTREE(f, T → lc)
38     if ∗T is unprocessed then *f(∗T)
39     HELPTREE(f, T → rc)
```

Algorithm 5: TraverseTree: a lock-free leaf-oriented tree with fat leaves, implementing a traverse object. Code for thread $t \in \{0, \ldots, n-1\}$.

```
▷ type Node
    int key
    {Node,Leaf} *left
    {Node,Leaf} *right
    InsertRec
    Announce[0..n−1]
    Boolean
    helpersExist
▷ type InsertRec
    Data data
    int position
▷ type Leaf extends Node
    Data D[0..m − 1]
    CounterObject
    Elements;
▷ Shared variables:
1 {Node,Leaf} *Tree := NULL, initially pointing to a Leaf that is initialized with
  ⟨key, NULL, NULL, ⟨⟨⊥, ⊥⟩, ..., ⟨⊥, ⊥⟩⟩, False, ⟨⊥, ..., ⊥⟩, ⊥, 0⟩

  Procedure TreeInsert(Data data, Boolean isHelper)
2   Leaf *leaf
3   {Node,Leaf} *parent := Tree, **ptr
4   int pos, val
5   Boolean expeditive
6   while True do
7     ⟨leaf, parent⟩ := Search(data, parent)
8     if parent = NULL then ptr := &Tree
9     else if parent → left = leaf then ptr := &parent → left
10    else ptr := &parent → right
11    if isHelper = True and leaf → helpersExist = False then
12      leaf → helpersExist := True
13    ⟨pos, expeditive⟩ := Elements.NextIndex(&leaf → helpersExist) if expeditive = False then
      leaf → Announce[t] := ⟨data, ⊥⟩
15    if pos < M then
16      if expeditive = False then
          leaf → Announce[t].position := pos
17      leaf → D[pos] := data
18    else SplitLeaf(leaf, ptr, expeditive)                    // split leaf node
19    if (*ptr) → helpersExist = True then
20      if expeditive = False and
        (*ptr) → Announce[t].position ≠ ⊥ then
21        (*ptr) → Announce[t] := ⟨⊥, ⊥⟩
22      else continue
23    return

  Procedure SplitLeaf(Leaf leaf, Node **prt, Boolean expeditive)
24  Node *newNode := new Node initialized with ⟨⊥, NULL, NULL, ⟨⟨⊥, ⊥⟩, ..., ⟨⊥, ⊥⟩⟩, not expeditive⟩
25  Node *newNode → left := new Leaf initialized with ⟨⊥, NULL, NULL, ⟨⟨⊥, ⊥⟩, ...,
    ⟨⊥, ⊥⟩⟩, False, ⟨⊥, ..., ⊥⟩, ⊥, 0⟩
26  Node *newNode → right := new Leaf initialized with ⟨⊥, NULL, NULL, ⟨⟨⊥, ⊥⟩, ...,
    ⟨⊥, ⊥⟩⟩, False, ⟨⊥, ..., ⊥⟩, ⊥, 0⟩
27  Set splitBuffer := ∅
28  if expeditive = False then
29    for int i ∈ {0, ..., n − 1} with leaf → Announce[i].data ≠ ⊥ do
30      Data ldata := leaf → Announce[i].data
31      if leaf → Announce[i].position ≠ ⊥ then
          leaf → D[leaf → Announce[i].position] := ldata
32      else add ldata to splitBuffer
33      newNode → Announce[i] := ⟨ldata, -1⟩
34  distribute non-⊥ distinct elements of leaf → D[i] ∪ splitBuffer into newNode → left and
    newNode → right, with the appropriate keys, and fix key of newNode      // may result in more
    leaf splits
35  CAS(*ptr, leaf, newNode)
```

Algorithm 6: Priority queue of FreSh. Code for thread t.

```
   ▷ Shared variables:
1  ⟨int, Data⟩ A[0..k − 1], initially all ⟨⊥, ⊥⟩
2  CounterObject Cnt
3  Boolean helpersExist, initially False
4  int insPos, initially 0
5  ⟨int, Data⟩ *SA, initially NULL

   Procedure InsertPhase(int priority, Data values, Boolean isHelper)
6      int pos
7      pos := FAI(insPos)
8      A[pos] := ⟨priority, value⟩
   Procedure InitDeletePhase()
9      ⟨int, Data⟩ *sa
10     sa := allocate local sorted array of insPos elements, initially all ⟨⊥, ⊥⟩
       copy into sa the non-⊥ elements of A and sort them

11 CAS(&SA, NULL, sa)
   Procedure DeleteMin(Boolean isHelper): returns Data
12     int pos
13     if isHelper = True and helpersExist = False then
14         helpersExist := True
15     pos := Cnt.NextIndex(&helpersExist)
16     if pos ≥ insPos then return ⊥
17     return SA[pos].data
```