FreSh: A Lock-Free Data Series Index

Panagiota Fatourou
ICS-FORTH &
University of Crete
faturu@csd.uoc.gr

Eleftherios Kosmas
Hellenic Mediterranean
University
ekosmas@hmu.gr

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

George Paterakis
ICS-FORTH &
University of Crete
csdp1311@csd.uoc.gr

Abstract—We present FreSh, a *lock-free* data series index that exhibits good performance (while being robust). FreSh is based on Refresh, which is a *generic approach* we have developed for supporting lock-freedom in an efficient way on top of any *locality-aware* data series index. We believe Refresh is of independent interest and can be used to get well-performed lock-free versions of other locality-aware blocking data structures. For developing FreSh, we first studied in depth the design decisions of current state-of-the-art data series indexes, and the principles governing their performance. This led to a theoretical framework, which enables the development and analysis of data series indexes in a modular way. The framework allowed us to apply Refresh, repeatedly, to get lock-free versions of the different phases of a family of data series indexes. Experiments with several synthetic and real datasets illustrate that FreSh achieves performance that is as good as that of the state-of-the-art *blocking* in-memory data series index. This shows that the helping mechanisms of FreSh are light-weight, respecting certain principles that are crucial for performance in locality-aware data structures.

I. INTRODUCTION

Processing big collections of data series is of paramount importance for a wide spectrum of applications, across many domains, such as: operation health monitoring in data centers, vehicles and manufacturing processes, internet of things data analysis, environmental and climate monitoring, energy consumption analysis, decision taking in financial markets, telecommunications traffic analysis, detection of medical and health problems, improvement of web-search results, identification of pests invading agricultural crops, etc. [1]–[3]. 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, due to very large size of data series collections, and the high dimensionality (i.e., length) of the data series that modern applications need to analyze. To address these challenges, current state-of-the-art data series indexes [4]–[13] are based on data series summarization. They develop a tree index containing data series summaries used to prune the series collection in order to restrict the execution of costly computations only to a small subset of it.

State-of-the-art data series indexes [4], [7], [9]–[12], [14] exploit the parallelism supported by modern multicore machines, but are largely lock-based to achieve synchronization. Using locks results in *blocking* implementations: if a thread that holds a lock delays, other threads block, without making any progress, until the lock is released. Such thread delays can degrade performance. Some applications (eg, operation health-monitoring in nuclear plants, or gravitational-wave detection in astrophysics), are sensitive to delays, and would benefit from our approach. Locks may also result in known problems, such as deadlock, priority inversion, and lock convoying [15].

Lock-freedom [16] is a widely-studied property when designing concurrent trees [17]–[20] and other data structures [15], [16], [21]. It avoids the use of locks, ensuring that the system, as a whole, makes progress, independently of delays (or failures) of threads. The relative performance of lock-free vs. lock-based algorithms depends on the setting: when threads are pinned to distinct cores, algorithms using fine-grained locks do well. However, in oversubscribed settings (more threads than cores), lock-based algorithms can suffer due to threads getting de-scheduled while holding a lock. Lock-free algorithms address these issues, but may be complicated and result in worse performance in settings with no delays/failures. Designing lock-free data series indexes, which exhibit good performance, is the focus of this paper: we develop a lock-free data-series index, which always produces the correct output, while maintaining the good performance of state-of-the-art lock-based indexes.

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. Unfortunately, conventional helping mechanisms are rather expensive and often introduce high overheads [15], [16], [22], [23]. For this reason, the vast majority of the software stack is still based on locks. Ensuring lock-freedom while maintaining the good performance of existing data series indexes is a major challenge.

State-of-the-art indexes are designed to (a) maintain some form of *data locality*, and (b) 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 [8], [10], [11]. This processing pattern enables threads to work in parallel and independently from each other, resulting in reduced synchronization and communication costs. These principles for reduced communication and synchronization are easily achieved when locks (or barriers) are utilized [7], [9]–[11]. However, the way helping works in conventional lock-free data structures is inherently incompatible to these principles, thus making it challenging to implement helping on top of such indexes without sacrificing them. Providing lock-

freedom while maintaining load-balancing among threads, and ensuring good data pruning are further challenges to address.

State-of-the-art data series indexes encompass several data processing phases, which often employ different data structures to accomplish their efficient processing. Coming up with lock-free versions of these data structures, while respecting the communication and synchronization cost principles that govern existing indexes, is another major challenge to address.

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 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. Accomplishing these goals leads to additional challenges.

Our approach. We propose FreSh, a novel *lock-free* data series index, that efficiently addresses all of the above challenges. Our experimental analysis shows that the performance of FreSh is as good as that of MESSI [10], the state-of-the-art concurrent data-series index, which is lock-based. This attests to the high efficiency of the helping scheme we propose for FreSh. Moreover, in many cases, FreSh performs better than MESSI, as it allows for increased parallelism when constructing the tree index. Note that if threads *crash*, MESSI (and all other lock-based approaches [7], [10]–[12]) never terminate (so, we do not provide experiments for this case). FreSh always successfully and correctly terminates.

To get 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 adding any cost. Refresh introduces 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 such conventional techniques result in significantly lower performance.

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 necessary. Refresh introduces two modes of execution for each thread: (i) *expeditive* and (ii) *standard*. A thread executing in *standard* mode may incur synchronization overhead, as it needs to synchronize with helper threads; a thread executing in expeditive mode executes a code that avoids synchronization altogether. A thread starts by processing its assigned workload in 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 al-gorithm* (Section IV) to get a lock-free version of it. FreSh (Section V) follows the design decisions of locality-aware iSAX-based indexes [10], [14] (see Section II). However, to develop FreSh, we had to replace all data structures of the original index [10] with corresponding locality-aware lock-free versions; we present lock-free implementations of several concurrent data structures, such as binary trees and priority queues (Section V). The proposed lock-free tree contains several new ideas. Previous solutions [17], [19], [44], [57] require that when a key is inserted in a leaf ℓ, ℓ is copied and updated locally, and then replaced in the shared tree. This results in bad performance. Instead, we designed a novel algorithm for updating leaves, which provides enhanced parallelism compared to existing algorithms. Another novelty is the support of the new implementations for the expeditive and standard execution modes, which was a challenge by itself, as synchronisation is needed to transfer from one mode to the other. We believe that these implementations, as well as Refresh, 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 III). The traverse object is an abstract data type that leads to a generic methodology for designing an iSAX-based index in a modular way. It abstracts the main processing pattern used during the operation of iSAX-based indexes. Specifically, the iSAX-based index can be implemented via traverse object operations. The introduction of the traverse object is one of the novelties of our work.

Contributions. Our contributions are summarized as follows.
• We develop a theoretical framework for supporting lock-freedom in a systematic way on top of highly-efficient data series indexes. In particular, we present Refresh, a novel *generic approach* that can be applied on top of any locality-aware data series algorithm to ensure lock-freedom.
• Based on Refresh, we develop FreSh, the first *lock-free*, efficient iSAX-based data series index. To get FreSh, we present new lock-free implementations of several data structures which support the needed functionality.
• 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 lock-freedom (and in many cases achieves better performance).
• Experiments show that by providing lock-freedom without jeopardizing locality-awareness, FreSh outperforms by far several lock-free baselines we have designed, based on conventional approaches for ensuring lock-freedom.
• We present a theoretical framework, which introduces the traverse object, and utilizes it to enable the development of locality-aware data series indexes in a modular way.

II. PRELIMINARIES AND RELATED WORK

Data Series, Indexing, and Similarity Search. A *data series* (DS) of *size* (or *dimensionality*) n is a sequence of n

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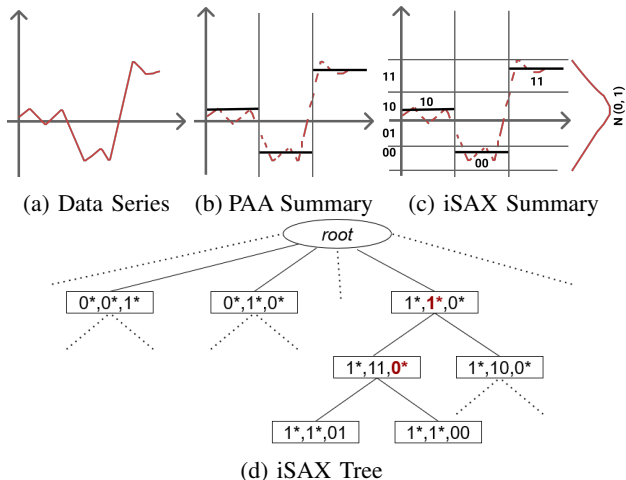
(d) iSAX Tree

Fig. 1: From data series to iSAX index

(value, position) pairs. The *Piecewise Aggregate Approximation (PAA)* [24] of a data series is a vector of w components which are calculated by splitting the x-axis into w equal segments, and representing each segment with the mean value of the corresponding points (depicted by the black horizontal lines in Figure 1(b)). To calculate the iSAX summary [25] 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 that represent each of the w segments of the series not by the real value of the PAA, but with the symbol of the region the PAA falls into, forming the word $10_2 00_2 11_2$ shown in Figure 1(c) (subscripts denote the number of bits used to represent the symbol of each segment). The number of bits can be different for each region, and this enables the creation of a hierarchical tree index (*iSAX-based tree index*), as shown in Figure 1(d). The index is implemented as a leaf-oriented tree with each leave storing up to M keys. 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 that receive the keys of ℓ. If one of the newly created leaves is empty, the splitting process is repeated. For more details on iSAX-based indices, see [14].

We focus on *exact similarity search* (a.k.a. exact *1-NN*) which returns the data series from a collection that 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) [26]. We call the distance between the *iSAX summaries* of two data series *lower-bound distance*. The way this distance is calculated ensures the *pruning property*: 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 enables pruning of data series during query answering: A data series can be *pruned*, if its lower bound distance from the query series Q exceeds any collection series' real distance to Q.

iSAX-Based Indexing. Concurrent iSAX-based indexes [7]–[11] 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 (i.e., *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, i.e., *summarization buffers* (*buffers creation* stage). Then, the worker threads traverse these buffers and insert their entries in the tree index (*tree population* stage). Data series that have similar summarizations are placed into the same buffer and later in the same root subtree of the index tree. This ensures high parallelism, a good degree of locality, and low synchronization overhead in building 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 index, 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 serves 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 stage*). 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 [8], [10], [11]. 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. Locks synchronize threads when they accesss the same parts of data structures.

System. We consider a shared-memory system of N threads which are executed asynchronously and communicate by accessing shared objects. A shared object O can be atomically read or written. Moreover, *FAI*(O,v) atomically reads the current value of O, adds the value v to it and returns the value read. A *CAS*(O, u, v) reads the value of O and if it is equal to u, it changes it to v and returns `True`; otherwise, O remains unchanged and `False` is returned.

Threads may experience delays (e.g., due to page faults, power consumption issues or overheating [27]) or they may fail by crashing (e.g., due to software errors). An algorithm is *blocking* if a thread has to wait for actions to be taken by other threads in order to make progress. *Lock-freedom* guarantees that the system as a whole continues to make progress, independently of the speed of threads or their failures.

A. Other Related Work

Several tree-based techniques for efficient and scalable data series similarity search have been proposed in the literature [4], [28]–[30], including approximate [31], [32] and progressive [33]–[36] solutions. Out of those, the iSAX-based indexes [14] have proven to be very competitive in terms of both index building and query answering time performance [4], [12], [13], [28], [37]. These indexes also include

parallel and distributed solutions that make use of modern hardware (e.g., SIMD, multi-core, multi-socket, GPU), such as ParIS+ [8], MESSI [10], and SING [11], as well as distributed computation, such as DPiSAX [38], [39] and Odyssey [37].

The first lock-free implementation of a concurrent search tree appears in [17]. We use the main ideas from that paper to come up with a baseline algorithm, which we discuss and experimentally compare with FreSh in Section VI. Many other non-blocking concurrent search trees have appeared in the literature (e.g., [18]–[20], [40]–[46]). The novelty of the tree implementation we present in Section V 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 better performance. Our algorithm is designed to only provide the functionality needed to implement traverse objects. The above-mentioned implementations support different functionalities, have different goals, or have been designed for other settings.

Concurrent priority queues appear in [47]–[52]. In the baseline lock-free implementations we developed, we use a skip-list based lock-free priority queue [52], which has been shown to perform well. Our experiments show that the scheme of priority queues we designed for FreSh, outperforms by far this implementation (Section VI).

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 [53]–[55] but they have all been introduced to solve different problems and the main technique of Refresh departs from all these approaches.

III. TRAVERSE OBJECTS

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 the traverse object, whose sequential specification is provided below.

Definition III.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. *param* plays the same role as in PUT.

S satisfies the *traversing property*: Each instance of TRAVERSE in every (sequential) execution of the object applies f at least once on all distinct elements added in S and not yet been deleted by the invocation of TRAVERSE.

We use four instances of a traverse object to implement the four stages of an iSAX-based index. We call BC, TP, PS, and RS, the traverse objects that implement the buffer creation, tree population, prunning, and refinement stages, respectively.

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, $i \in \{0, \dots, n-1\}$:
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($\langle iSAX$, index of $ds \rangle$, $bind$)

Procedure TREEPOPULATION(*Summary iSAX, Index ind, Index bind, Boolean flag*)
12   PS.PUT($\langle iSAX, ind \rangle$, $bind$, $flag$)

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($\langle E, iSAX \rangle$, $flag$)
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* $\langle iSAX_{ds}, ind_{ds} \rangle$ *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

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 organizes the pairs 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 is now comprised of a sequence of four invocations of TRAVERSE on the different traverse objects. Algorithm 1 provides pseudocode for the implementation of an iSAX-based index using traverse objects. 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 Algorithm 1, the barriers, if needed, (as well

as multithreading processing) should be incorporated in the implementation of PUT and TRAVERSE. Thus, an iSAX-based index satisfies the following property.

Definition III.2 (Non-Overlapping Property). In every (concurrent) execution of the index and every traverse object S accessed in the execution, each instance of TRAVERSE on S is performed only after the execution of all instances of PUT that add distinct elements in S has been completed.

Assume that the non-overlapping property holds for BC, TP, PS, and RS and that $RawData$ initially stores all raw data series. 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, i.e., 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 V.

IV. LOCALITY-AWARE LOCK-FREEDOM

Locality-awareness aims at capturing several design principles (Definition IV.1) for data series indexes which are crucial for achieving good performance. A locality aware implementation respects these principles.

Definition IV.1. 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 (cache misses and branch misprediction) among the threads.
2) **High Parallelism & Low Synchronization Cost.** Threads should work in parallel and independently from each other. 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.

Enuring locality awareness results in good performance and is thus a desirable property for big data processing. 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

---

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 $W := [w_1, w_2, \ldots, w_k]$
2 boolean $F := [d_1, d_2, \ldots, d_k]$, initially $d_i = $ False, $1 \leq i \leq k$
3 boolean $H := [h_1, h_2, \ldots, h_k]$, initially $h_i = $ False, $1 \leq i \leq k$

▷ 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 unfinished parts of W and help
12   **for each** $d_i \in D$ *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

---

employs several priority queues to store leaf nodes containing candidate series. Thus, iSAX-based indexes are *locality-aware*.

To describe Refresh in more detail, consider a *blocking* locality-aware implementation $\mathcal{A}$, which splits its workload into disjoint parts and assigns them to threads for processing. Refresh (Algorithm 2) transforms $\mathcal{A}$ into a *lock-free* locality-aware implementation $\mathcal{B}$ that achieves high parallelism.

Let W be the workload that $\mathcal{A}$ processes and let $w_1, \ldots, w_k$ be the parts it is separated to ensure locality awareness. Refresh applies the following steps (depicted in Figure 2):
(1) It attaches a *flag* 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 acquires a workload is its *owner*.
(3) To achieve lock-freedom, every thread t, then, *scans* all the flags to find those parts that are still unfinished (line 12).
(4) Thread t *helps* by processing, one after the other, each part found unfinished during scan. For each part w_i that t helps, it periodically checks d_i to see whether other threads completed the processing of w_i. If this is so, t stops helping w_i (line 16). A thread that completes the processing of w_i, changes d_i to true (line 17).
(5) Due to helping, every 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 of $\mathcal{A}$.
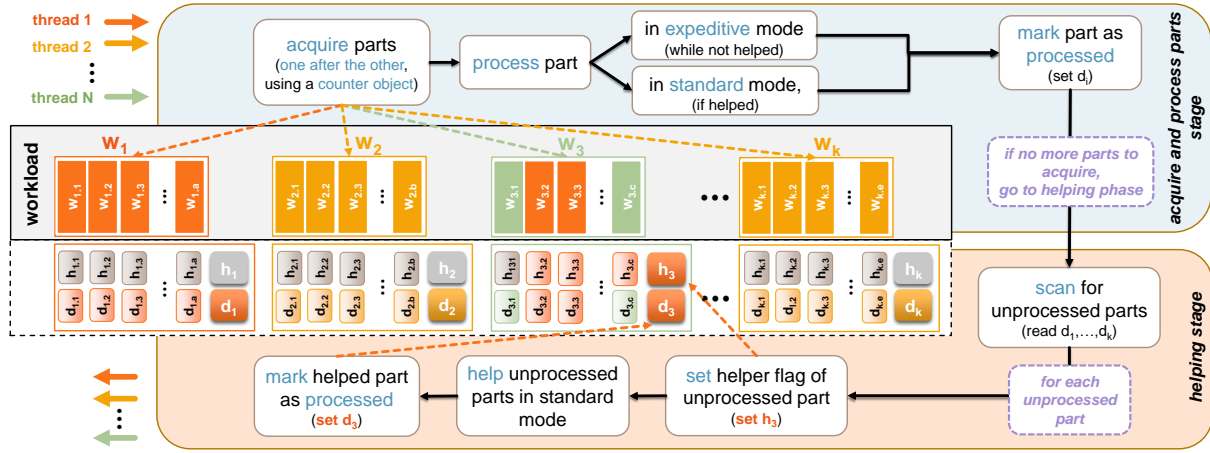
Fig. 2: Refresh flowchart.

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 structure 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). Before t starts helping some part w_i, it sets h_i to `True` (line 15), to alert w_i's owner thread to start running on standard mode (line 9).

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). A small delay before switching to standard mode, often positively affects performance. The delay is usually an estimate of the actual time a thread requires to finish its current workload, calculated at run time (see Section V-A for more details). 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 helps only the remaining unfinished subparts of w_i.

Lock-freedom is ensured due to the *helping code* (lines 12-17). In Refresh, only after a thread t processes a workload w_i, it sets h_i to `True` and t performs the helping code after finishing with their assigned workloads. Thus, when t completes its helping code the processing of all parts of the workload has been completed. This means that t may continue directly to the execution of the next stage, without waiting for the other threads to complete the execution of the current stage. Therefore, this scheme renders the use of barriers useless, as needed to achieve lock-freedom.

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

V. FreSh

We follow the data processing flow, described in Section III, employ BC, TP, PS, and RS (from Section III), and repeatedly apply Refresh (from Section IV) to come up with FreSh.

A. Buffers Creation and Tree Population

BC is implemented using a single buffer, called $RawData$. In BC, PUT is never used, as we assume that the data are initially in $RawData$. 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 (line 1). FreSh uses three sets of done flags, $DChunks$, $DGroups$, and $DElements$ (line 2), 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 $Elements$ (line 4)), to count the chunks, groups and elements, assigned to threads for processing. FreSh also employs two sets of *helping* flags (line 3), $HChunks$ (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

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 D_1[], Boolean D_2[], Boolean D_3[], Boolean H_1[], Boolean H_2[], Boolean h, CounterObject Cnt_1, CounterObject Cnt_2[], CounterObject Cnt_3[], int rlevel)*

6   int i
    // acquire and process parts of W
7   **while** *True* **do**
8      $\langle i, * \rangle := Cnt_1$.NEXTINDEX(&h)
9   **if** $i > Size[rlevel]$ **then break**
10   **mark** $RawData[i]$ as acquired
11   **if** $rlevel < 3$ **then**
       TRAVERSE($BufferCreation, RawData[i], D_2[i], D_3[i],$
         $D_3[i], H_2[i], NULL, H_1[i], Cnt_2[i], Cnt_3[i], Cnt_3[i])$
         $rlevel + 1$)
12   **else** *BUFFERCREATION($RawData[i]$)
13
14     $D_1[i] :=$ True
    // scan flags for unprocessed parts of W
    // and help
15   **for each** j such that $D_1[j]$ is equal to False **do**
16   Backoff()   // avoid helping, if possible
17   **if** $D_1[j]] = False$ **then**
18      $H_1[j] :=$ True
19   **if** $rlevel < 3$ **then**
       TRAVERSE($BufferCreation, RawData[j], D_2[j], D_3[j],$
         $D_3[j], H_2[j], NULL, H_1[j], Cnt_2[j], Cnt_3[j],$
         $Cnt_3[j]), rlevel + 1$)
20   **else** *BUFFERCREATION($RawData[j]$)
21      $D_1[j] :=$ True

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*) on line 11 (with $rlevel$ being equal to 2). 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 (recursive call of line 11 with $rlevel$ being equal to 3). 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 $HGroups[i]$. Helping (lines 15-21) follows the general pattern described in Algorithm 2.

The backoff time in FreSh depends on the average execution time required by each thread to process a group. 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 (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 summarization 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 2^w workloads and apply Refresh. 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. BC and TP are lock-free implementations of a traverse object.

B. Pruning and Refinement

In FreSh, PS is implemented as a forest of 2^w leaf-oriented trees, one for each of the summarization buffers. The trees of the forest are the root subtrees of a standard iSAX-based tree. To implement PS.TRAVERSE, FreSh uses Refresh to process the different subtrees of the index tree. Specifically, each thread t access a counter to get assigned a subtree T to process. To process the nodes of T, Refresh is applied recursively. A thread t that is assigned node i of T, first searches for the i-th node, according to inorder, and then processes it by invoking the PRUNING function of Algorithm 1. To find the i-th node of 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. FreSh uses these counters to find the i-th node of T by simply traversing a path in T. The total number of nodes in T is calcualted by simply traversing the righmost path of T and summing up the counters stored in the traversed nodes.

1) Insert in Leaf-Oriented Tree: Each node of the tree stores a key and the pointers 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. A node may have its own key. For instance, in iSAX-based indexes, this key is the node's iSAX summary. The proposed implementation allows multiple insert operations to concurrently update array D of a leaf. This results in enhanced parallelism and performance. To achieve this, each leaf ℓ contains a counter, 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.

During spliting, D may contain empty positions, since some threads may have acquired positions in D but have not yet stored their elements there. To avoid situations of missing

elements, each leaf contains an *Announce* array with one position for each thread. A thread announces its operation in *Announce* before it attempts to acquire a position in D. During splitting, a thread distributes to the new leaves it creates not only the elements found in D but also those in *Announce*.

Our approach is a *linearizable, lock-free* implementation of a leaf-oriented tree with fat leaves (supporting only insert).

C. Refinement

To implement RS, FreSh uses a set of priorities queues each implemented using an array. 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 comes up with sorted versions of the arrays, 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. Processing of an array 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. RS is a linearizable lock-free implementation of a traverse object.

To update BSF, FreSh repeatedly reads the current value y of BSF, and attempts to atomically change it from v to the new value v', using *CAS*, until it either succeeds or some value smaller than or equal to v' is written in BSF.

Theorem V.1. FreSh solves the 1-NN problem and provides a lock-free implementation of QUERYANSWERING (Alg. 1).

VI. EXPERIMENTAL EVALUATION

Setup. We used a machine equipped with 2 Intel Xeon E5-2650 v4 2.2GHz CPUs with 12 cores each, and 30MB L3 cache. The machine runs Ubuntu Linux 16.04.7. LTS and has 256GB of RAM. Code is written in C and compiled using gcc v11.2.1) with O2 optimizations.

Datasets. We evaluated FreSh and the competing algorithms (all algorithms are in-memory) using both real and synthetic datasets. The synthetic data series, *Random*, are 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 [4], [5], [28], [58]–[60], and models the distribution of stock market prices [58]. Our real datasets come from the domains of seismology and astronomy. The seismic dataset, *Seismic*, was obtained from the IRIS Seismic Data Access archive [61]. 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 [62]. 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.

Evaluation Measures. We measure (i) the *summarization time* required to calculate the iSAX summaries and fill-in the summarization 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 queries that are not part of the dataset. The sum of the above times constitute the *total time*. Experiments are repeated 5 times and averages are reported. All algorithms return exact results.

A. Results

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

We also compare FreSh against an extended version of MESSI, called MESSI-enh, that allows several threads to concurrently populate the same sub-tree, during tree creation (instead of using a single thread for each subtree). This is implemented using fine-grained locks that are attached on each leaf node of a subtree. MESSI-enh allows to compare the lock-free index creation phase of FreSh against a more efficient blocking one than that of original MESSI.

Figure 3 shows that all algorithms (FreSh, MESSI, and MESSI-enh) continue scaling as the number of threads is increasing, for Seismic 100GB. This is true for all three phases. Moreover, the total execution time of FreSh (Figure 3a 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 (Figure 3c), since FreSh allows subtrees to be populated concurrently by multiple threads, allowing parallelism during this phase, in contrast to MESSI. Interestingly, FreSh achieves better performance than MESSI-enh, in most cases. The results for Seismic are similar and are omitted for brevity.

Considering scalability as the size of the dataset increases, Figure 5 demonstrates that FreSh scales well on all three datasets. In most cases, FreSh is faster than MESSI.

Following previous works [10], [60], 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 6a presents the results for the Seismic dataset, where FreSh performs better than MESSI in most cases.

FreSh vs Baselines. We compare FreSh against several baseline *lock-free* implementations of the different stages of an iSAX-based index. Our results (Figure 6d shows that FreSh performs better than all these implementations.

Summarization Baseline: For buffer creation, we have experimented with three implementations: DoAll-Split, FI-Based, and CAS-Based. All use a single summarization buffer with as many elements as $RawData$. DoAll-Split splits $RawData$ into as many equally-sized chunks as the number of threads. It stores a *done* flag with each data series, which is set after the data series is processed. Each thread traverses $RawData$ (circularly), starting from the first element of its assigned chunk. The thread first checks whether the done flag of a data series is set, and processes it only if not. In FI-Based,
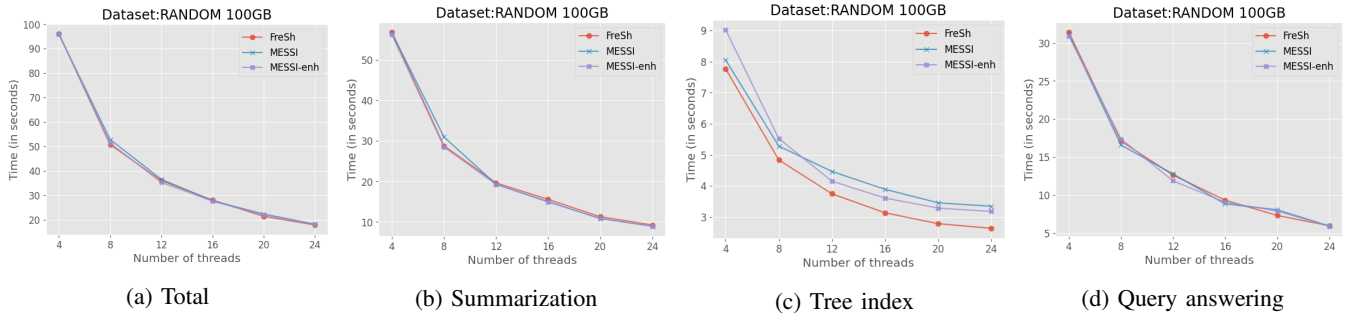
(a) Total   (b) Summarization   (c) Tree index   (d) Query answering

Fig. 3: Comparison of FreSh against MESSI and MESSI-enh on 100GB Random.

(a) Total   (b) Summarization   (c) Tree index   (d) Query answering
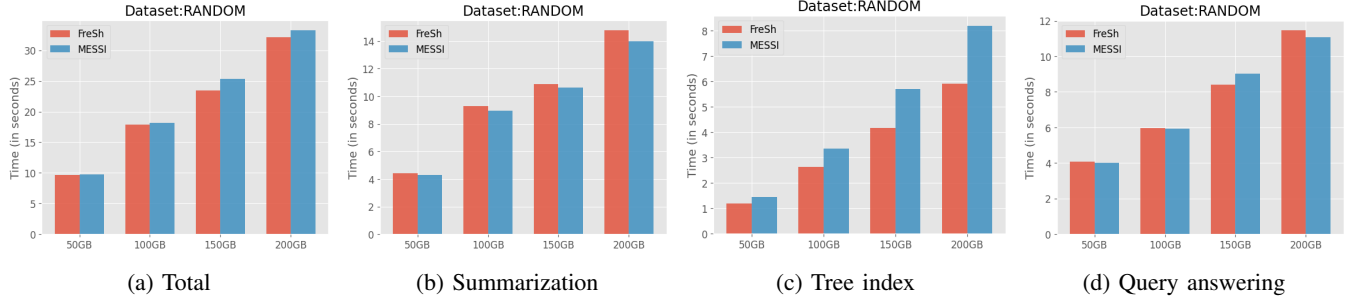
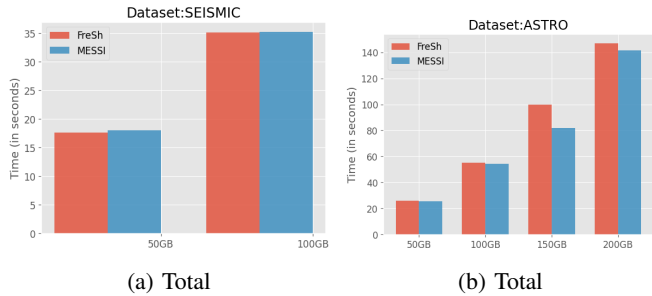Fig. 4: Comparison of FreSh against MESSI: (a-d) on Random, for 24 threads.

(a) Total   (b) Total

Fig. 5: Comparison of FreSh against MESSI: (a) on Seismic, and (b) on Astro, for 24 threads.

threads use *FAI* to get assigned data series from *RawData* to process. When a thread figures out that all *RawData* elements have been assigned, it re-traverses *RawData* to identify data series whose done flag is still `False`, and processes them. `CAS-Based` works similarly to `FI-Based`, while it uses *CAS* instructions, instead of *FAI*. FreSh performs significantly better than all these implementations (Fig. 6d).

Tree Population Baseline: 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 `DoAll-Split`, `FI-Based`, and `CAS-Based` techniques we describe above. To achieve lock-freedom in accessing the tree, we utilize a flagging technique [56], in addition to our new tree implementation. 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*. FreSh performs significantly better than all these implementations

(graph omitted for brevity).

Pruning Baseline: All baselines use a single instance of an existing skip-based lock-free priority queue [52] to store the candidate data series for refinement. Threads uses *FAI* to find the next node to examine in the index tree. When a thread t discovers that all nodes of the tree have been assigned for processing, it re-traverses the tree to find nodes that may still be unprocessed, and processes them.

Refinement Baseline: All threads, repeatedly call DeleteMin to remove elements from the priority queue, and calculate their real distance computation. Our results (graph omitted due to lack of space) show that FreSh performs significantly better than all these implementations, for query answering time (that includes pruning and refinement).

Performance breakdown for index creation phase. 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 of its 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 TreeCopy, a
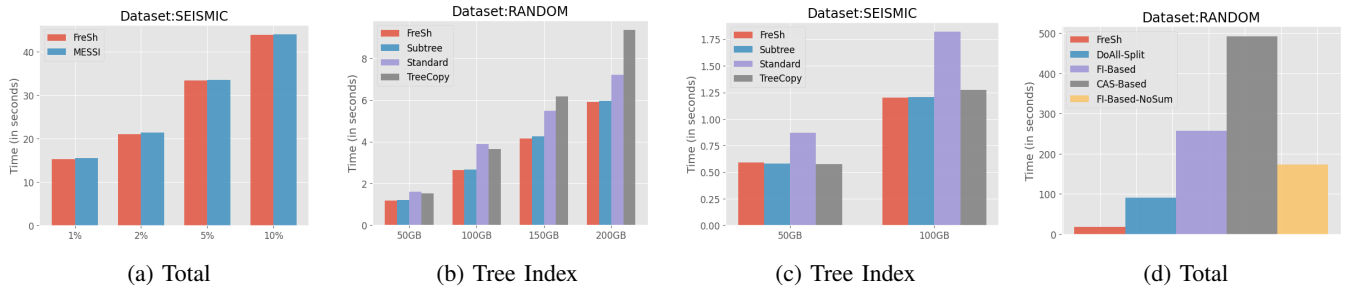
Fig. 6: **(a)** Comparison of FreSh against MESSI on Seismic 100GB with variable query difficulty, where an increasing percentage of noise is added to the original queries. **(b)-(c)** Comparison of FreSh index creation to other tree implementations. **(d)** Comparison of FreSh against baseline implementations on 100GB Random.

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.

Figures 6b-6c compare 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.

Thread Delays. In order to study systems where processes may experience delays (e.g., due to page faults, time sharing, or long phases of updates), we came up with a simplistic benchmark, where we simulate delays at random points of a thread's execution. Figure 7a illustrates that the delay even of a single thread causes a linear overhead on the performance of MESSI, whereas it hardly has any impact in the performance of FreSh. Moreover, Figure 7b shows that MESSI takes (almost) the full performance hit of delayed threads right from the beginning: even a single delayed thread blocks the execution of all other threads, and hence of the entire algorithm. FreSh gracefully adapts to the situation of increasing number of delayed threads, achieving a speedup. Note that these synthetic benchmarks are designed to simply illustrate the impact of lock-freedom on performance when threads may experience delays (or crash), and not to capture some realistic setting. Note that MESSI will not terminate execution even if a single thread fails. In the case of failures (see Figure 8), FreSh always terminates execution, performing almost identical to MESSI with the same number of non-failing threads. This demonstrates that FreSh adapts to dynamic thread environments, maintaining high performance levels.

VII. CONCLUSIONS

Current state-of-the-art data series indexes exploit the parallelism supported by modern multicore machines, yet, their design is lock-based, and therefore, these implementations are blocking. In this paper, we present FreSh, a *lock-free* index,
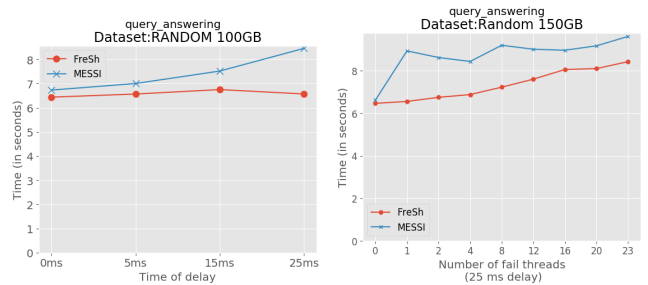
(a) A single thread is delayed. (b) Mutliple threads are delayed.

Fig. 7: Comparison of FreSh against MESSI when varying delay and number of delayed threads.
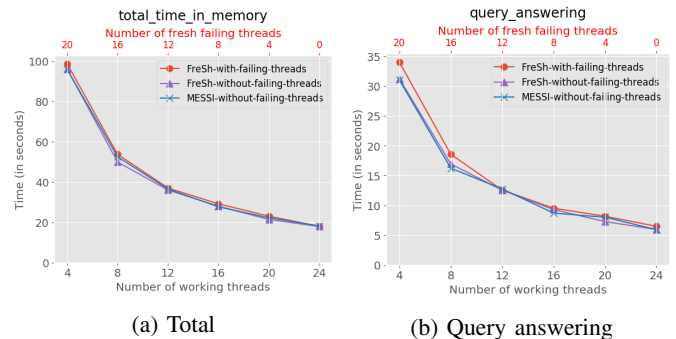
(a) Total (b) Query answering

Fig. 8: Execution time on Random 100GB, when varying the number of threads that permanently fail (FreSh with permanent failures in red circles, FreSh without failures in purple triangles, MESSI without failures in blue crosses).

based on Refresh, our novel generic approach for designing, building and analyzing highly-efficient data series indexes in a modular way. The experimental evaluation demonstrates that FreShperforms as good as the state-of-the-art *blocking* index, thus, adhering to the same locality-aware design principles.

REFERENCES

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

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

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

[4] 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*, vol. 12, no. 2, 2018.

[5] 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*, vol. 39, no. 1, 2014.

[6] 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.

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

[8] ——, "Paris+: Data series indexing on multi-core architectures," *TKDE*, 2020.

[9] ——, "Messi: In-memory data series indexing," in *ICDE*, 2020.

[10] ——, "Fast data series indexing for in-memory data," *The VLDB Journal*, vol. 30, no. 6, pp. 1041–1067, nov 2021.

[11] ——, "Sing: Sequence indexing using gpus," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 1883–1888.

[12] K. Echihabi, P. Fatourou, K. Zoumpatianos, T. Palpanas, and H. Benbrahim, "Hercules against data series similarity search," *Proc. VLDB Endow.*, vol. 15, no. 10, pp. 2005–2018, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p2005-echihabi.pdf

[13] Z. Wang, Q. Wang, P. Wang, T. Palpanas, and W. Wang, "Dumpy: A Compact and Adaptive Index for Large Data Series Collections," in *SIGMOD*, 2023.

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

[15] K. Fraser, "Practical lock-freedom," University of Cambridge Computer Laboratory, Tech. Rep. 579, 2004. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

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

[17] 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*, 2010, pp. 131–140.

[18] 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*, 2014, pp. 332–340.

[19] P. Fatourou and E. Ruppert, "Persistent non-blocking binary search trees supporting wait-free range queries," *CoRR*, vol. abs/1805.04779, 2018. [Online]. Available: http://arxiv.org/abs/1805.04779

[20] 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*, ser. PPoPP '22, 2022, pp. 262–277.

[21] 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*, ser. SPAA '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 111–120. [Online]. Available: https://doi.org/10.1145/3210377.3210408

[22] 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)*, 2016, pp. 43–52.

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

[24] 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.*, vol. 3, no. 3, pp. 263–286, 2001.

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

[26] 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.

[27] Intel, https://www.intel.com/content/www/us/en/support/articles/000005791/processors/intel-core-processors.html, 2023.

[28] 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.*, vol. 13, no. 3, pp. 403–420, 2019.

[29] 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*, 2021, pp. 714–717.

[30] K. Echihabi, T. Palpanas, and K. Zoumpatianos, "New trends in high-d vector similarity search: Ai-driven, progressive, and distributed," *Proc. VLDB Endow.*, vol. 14, no. 12, pp. 3198–3201, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p3198-echihabi.pdf

[31] I. Azizi, K. Echihabi, and T. Palpanas, "Elpis: Graph-based similarity search for scalable data science," *Proc. VLDB Endow.*, vol. 16, no. 6, pp. 1548–1559, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p1548-azizi.pdf

[32] O. Levchenko, B. Kolev, D. E. Yagoubi, R. Akbarinia, F. Masseglia, T. Palpanas, D. E. Shasha, and P. Valduriez, "Bestneighbor: efficient evaluation of knn queries on large time series databases," *Knowl. Inf. Syst.*, vol. 63, no. 2, pp. 349–378, 2021. [Online]. Available: https://doi.org/10.1007/s10115-020-01518-4

[33] A. Gogolou, T. Tsandilas, K. Echihabi, A. Bezerianos, and T. Palpanas, "Data series progressive similarity search with probabilistic quality guarantees," in *SIGMOD*, 2020.

[34] J. Jo, J. Seo, and J. Fekete, "PANENE: A progressive algorithm for indexing and querying approximate k-nearest neighbors," *IEEE Trans. Vis. Comput. Graph.*, vol. 26, no. 2, pp. 1347–1360, 2020.

[35] C. Li, M. Zhang, D. G. Andersen, and Y. He, "Improving approximate nearest neighbor search through learned adaptive early termination," in *SIGMOD*, 2020.

[36] K. Echihabi, T. Tsandilas, A. Gogolou, A. Bezerianos, and T. Palpanas, "Pros: data series progressive k-nn similarity search and classification with probabilistic quality guarantees," *VLDB J.*, vol. 32, no. 4, pp. 763–789, 2023. [Online]. Available: https://doi.org/10.1007/s00778-022-00771-z

[37] M. Chatzakis, P. Fatourou, E. Kosmas, T. Palpanas, and B. Peng, "Odyssey: A Journey in the Land of Distributed Data Series Similarity Search," *PVLDB*, 2023.

[38] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas, "DPiSAX: Massively Distributed Partitioned iSAX," in *ICDM*, 2017.

[39] D.-E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas, "Massively distributed time series indexing and querying," *TKDE*, vol. 32, no. 1, 2020.

[40] 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*, 2014, pp. 329–342.

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

[42] 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*, 2020, pp. 503–505.

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

[44] A. Natarajan, A. Ramachandran, and N. Mittal, "FEAST: a lightweight lock-free concurrent binary search tree," *ACM Transactions on Parallel Computing*, vol. 7, no. 2, May 2020.

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

[46] A. Braginsky and E. Petrank, "A lock-free B+tree," in *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, 2012.

[47] 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*, ser. PPoPP 2015. New York, NY, USA: Association

for Computing Machinery, 2015, pp. 11–20. [Online]. Available: https://doi.org/10.1145/2688500.2688523

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

[49] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-lsm relaxed priority queue," *SIGPLAN Not.*, vol. 50, no. 8, pp. 277–278, jan 2015. [Online]. Available: https://doi.org/10.1145/2858788.2688547

[50] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731504002333

[51] O. Tamir, A. Morrison, and N. Rinetzky, "A Heap-Based Concurrent Priority Queue with Mutable Priorities for Faster Parallel Algorithms," in *OPODIS*, vol. 46, 2016, pp. 1–16.

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

[53] 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*, ser. PPoPP '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 357–368. [Online]. Available: https://doi.org/10.1145/2555243.2555261

[54] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit, "Obstruction-free algorithms can be practically wait-free," in *Distributed Computing*, P. Fraigniaud, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 78–92.

[55] R. Guerraoui, M. Kapałka, and P. Kouznetsov, "The weakest failure detectors to boost obstruction-freedom," in *Distributed Computing*, S. Dolev, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 399–412.

[56] 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*, 2010, pp. 131–140.

[57] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert, "The amortized complexity of non-blocking binary search trees," in *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, 2014, pp. 332–340. [Online]. Available: https://doi.org/10.1145/2611462.2611486

[58] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases," in *SIGMOD*, New York, NY, USA, 1994.

[59] 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*, 2015, pp. 1603–1612. [Online]. Available: http://doi.acm.org/10.1145/2783258.2783382

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

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

[62] 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*, vol. 563, 2014.