DFreSh: A Lock-Free Index for Real-Time Data Series Processing

Paterakis George

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Assistant Prof. *Panagiota Fatourou*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

DFreSh: A Lock-Free Index for Real-Time Data Series Processing

Thesis submitted by
Paterakis George
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Paterakis George

Committee approvals: _____
Panagiota Fatourou
Assistant Professor, Thesis Supervisor

_____
Professor 2
Professor, Committee Member

_____
Professor 3
Professor, Committee Member

Departmental approval: _____
Antonios Argyros
Professor, Director of Graduate Studies

Heraklion, March 2025

TITLE

Abstract

Abstract

In this thesis, we present DFreSh, a real-time lock-free data series index that achieves high performance while maintaining robustness. DFreSh builds upon FreSh, the current state-of-the-art in lock-free data series indexing. FreSh, in turn, is based on Refresh, a generic framework designed to enable efficient lock-freedom in locality-aware data series indexes. Refresh provides a systematic approach to deriving high-performing lock-free versions of other locality-aware blocking data structures. Despite its strengths, FreSh has a critical limitation: it is a static index, incapable of supporting dynamic updates after construction. This limitation renders FreSh unsuitable for real-time environments where data is continuously generated and needs to be inserted-integrated into the index without the need for complete rebuilding. Addressing this shortcoming, DFreSh introduces dynamic update capabilities, making it a robust solution for real-time applications while retaining the performance benefits of its predecessor. We evaluated DFreSh through extensive experiments using both synthetic and real-world datasets. The results demonstrate that DFreSh achieves performance comparable to FreSh, with its lightweight mechanisms for real-time data handling upholding the key principles of performance critical to locality-aware data structures.

o′Ω●o′Ω●o′Ω●o′Ω●o′Ω●o′Ω●

o′Ω●o′Ω●o′Ω●o′Ω●o′Ω●o′Ω●o′Ω●o′Ω●

ο′Ω●ο′Ω●ο′Ω●ο′Ω●ο′Ω●ο′Ω●ο′Ω●ο′Ω●ο′Ω●ο′Ω●

o΄Ω•o΄Ω•o΄Ω•o΄Ω•o΄Ω• o΄Ω•o΄Ω•o΄Ω•o΄Ω•o΄Ω•o΄Ω• o΄Ω•o΄Ω•o΄Ω•

Contents

List of Figures

Chapter 1

Introduction

Processing large collections of data series is critical for a wide range of applications [51, 6, 53]. These applications depend on efficient similarity search, which aims to retrieve the most similar data series to a given query Q. Specifically, a similarity search operation returns a subset of data series from the collection that are closest to Q based on a defined distance metric (or similarity function).

Beyond its standalone utility, similarity search is also a key component in various machine learning tasks, including clustering, classification, and outlier detection. However, performing similarity search at scale is computationally expensive due to two main factors

- The massive size of modern data series collections, which often reach terabyte or petabyte scales.
- The high dimensionality of data series (i.e. length) that modern applications need to analyze.

To overcome these challenges, state-of-the-art data series indexes [17, 9, 68, 54, 56, 55, 57, 58, 13, 69] employ data series summarization. These indexes build tree-based structures where each node stores summarized representations of data series, enabling efficient pruning of the dataset. By filtering out irrelevant series early, these indexes significantly reduce the number of costly distance computations, improving query performance. State-of-the-art such indexes [55, 56, 59] use iSAX summaries [9, 52],thus we call them iSAX-based indexes. Due to their efficiency and scalability, iSAX-based indexes have been widely adopted across diverse application domains. These include operational health monitoring in data centers, vehicles, and manufacturing processes; Internet of Things (IoT) data analysis; environmental and climate monitoring; energy consumption analysis; financial market decision-making; telecommunications traffic analysis; medical diagnostics; web search optimization; and even agriculture, where they help identify pest infestations in crops. To efficiently support these real-world use cases, state-of-the-art data series indexes [17, 52, 54, 55, 57, 58, 13] leverage modern multicore architectures to achieve parallelism. However, most existing approaches rely on lock-based

synchronization mechanisms, which introduce blocking behavior: if a thread holding a lock fails or is delayed, other threads waiting on the lock are forced to stall, leading to degraded performance. While thread failures are relatively rare, they do occur in practice, for example, due to software bugs in applications using the index or issues in memory management systems. This is especially problematic for mission-critical applications, such as nuclear plant monitoring or gravitational-wave detection in astrophysics, where any delay in data processing can have significant consequences. Such latency-sensitive applications would greatly benefit from our proposed approach, which eliminates blocking and ensures continuous progress. While locks provide synchronization, they introduce well-known performance issues, such as priority inversion and convoying [36]. Additionally, depending on how they are used, locks can limit parallelism, leading to suboptimal performance

An alternative approach is lock-free synchronization, a well-studied technique in concurrent tree structures [21, 19, 33, 3] and other data structures [36, 40, 32]. Lock-free algorithms eliminate the need for locks, ensuring that the system as a whole always makes progress, regardless of thread delays or failures. This property is particularly beneficial for highly parallel workloads, where blocking mechanisms can create bottlenecks. In this work, we introduce FreSh, a lock-free data series index that maintains the correctness and strong performance of state-of-the-art lock-based indexes while avoiding their synchronization overheads. Additionally, we present DFreSh, a dynamic version of FreSh that extends its capabilities to support batches of updates while simultaneously answering queries. This enables real-time index updates, making it well-suited for streaming data applications. The relative performance of lock-free vs. lock-based approaches depends on the execution environment. When threads are pinned to distinct cores, algorithms with fine-grained locks perform well. However, in oversubscribed settings (i.e., when there are more threads than available cores), lock-based solutions can suffer performance degradation, as threads holding locks may be de-scheduled, blocking others. Lock-free algorithms avoid such issues but can be more complex and, in some cases, may not always outperform lock-based approaches when delays and failures are absent.

Challenges. Achieving lock-freedom in data series indexes often requires the use of a *helping* mechanism, where threads are made aware of the work other threads perform and can assist them when needed, such as during delays or load imbalances. However, conventional helping mechanisms tend to be costly and introduce significant overhead [36, 40, 44, 70]. This is one of the reasons the vast majority of the software stack still relies on locks. Ensuring lock-freedom while maintaining the good performance of existing data series indexes remains a significant challenge. SState-of-the-art data series indexes are designed to (a) maintain data locality and (b) minimize synchronization, aiming for high parallelism. For example, they often partition the data into disjoint sets and assign a distinct thread to handle each set's data [56, 57, 58]. This design enables parallel and independent thread operations, reducing synchronization and communication costs. These goals are

easier to achieve with locks or barriers [54, 55, 57, 58]. However, the way helping functions in conventional lock-free data structures is inherently incompatible with these principles. Implementing helping in such indexes without compromising these principles is a challenge. Moreover, ensuring lock-freedom while maintaining load balancing and effective data pruning are additional hurdles. State-of-the-art data series indexes involve multiple processing phases, each using different data structures to optimize performance. Designing lock-based versions of these structures is feasible, but creating lock-free versions that adhere to the communication and synchronization principles of existing indexes is more difficult. To develop a generalized approach for supporting lock-freedom in data series indexes systematically, we must first analyze the design decisions of existing state-of-the-art indexes and the performance principles they follow. We need to establish appropriate abstractions for the data series processing stages, as well as design principles that ensure efficiency. Achieving these goals introduces further challenges. Lastly, a significant challenge in the context of dynamic updates, is transforming a static index, like FreSh, into a dynamic one that supports batch insertions of updates while maintaining the same high performance. This dynamic index must ensure that queries continue to return consistent and correct results, while also guaranteeing that each query retrieves all the data seen by previous queries, including any updates made since the last query.

Our approach. We introduce FreSh, a novel static lock-free data series index, and DFreSh, its dynamic counterpart that efficiently addresses all of the above challenges.

Our experimental analysis demonstrate that FreSh matches the performance of MESSI [57], the state-of-the-art concurrent data-series index, which is lock-based. This confirms the efficiency of the helping mechanism we employ in FreSh. In many cases, FreSh even outperforms MESSI by allowing for higher parallelism during the construction of the tree index. It's important to note that if threads crash, MESSI (and other lock-based approaches [54, 57, 58, 13]) will not terminate, which is why we did not provide experiments for this scenario. In contrast, FreSh always terminates successfully and correctly. To achieve FreSh, we developed ReFreSh, a generic approach that can be applied to a range of state-of-the-art blocking indexes to provide lock-freedom without incurring additional costs. ReFreSh introduces the concept of locality-aware lock-freedom, which integrates key properties of data locality, high parallelism, low synchronization costs, and load balancing, qualities found in many existing parallel data series indexes. None of the conventional lock-free techniques we are aware of have been designed with these properties in mind, and our experiments show that using traditional lock-free techniques results in a significant performance drop (Graduate Thesis). ReFreSh ensures that the underlying workload and data separation of the original data series index are preserved, thus not compromising parallelism or load balancing. It provides a mechanism for threads to determine whether a specific workload part has been processed, performing helping only when necessary. ReFreSh operates in two modes:

- Expeditive mode with minimal synchronization costs, which avoids any helping.
- Standard mode, where synchronization is necessary for helpin

A thread initially processes its workload in expeditive mode, and helping is only performed after completing its assigned task. After that, threads synchronize to switch to standard mode. This approach ensures that synchronization and communication costs remain low, similar to the original index. ReFreSh can be applied to any locality-aware algorithm (Chapter 4) to create its lock-free version. FreSh (Chapter 5) follows the design principles of locality-aware iSAX-based indexes [52, 57], but requires replacing the original index data structures with lock-free versions that support both expeditive and standard modes. We introduce lock-free implementations of several concurrent data structures, including binary trees and priority queues (Chapter 5), which provide enhanced parallelism compared to existing solutions. We believe that these lock-free implementations and the ReFreSh framework can be applied to achieve high-performance lock-free versions of various big data processing systems. To enable the systematic application of ReFreSh across all processing stages of an iSAX-based index, we introduce the abstraction of a traverse object (Chapter 3). The traverse object is an abstract data type that offers a generic methodology for designing iSAX-based indexes modularly, abstracting the key operations involved in the index function. This allows us to provide a formal and flexible approach to designing iSAX-based indexes. Specifically, the iSAX-based index can be implemented as a sequence of operations on traverse objects. The novelties of this work go beyond the new techniques we introduce to design the lock-free index tree; in summary:

- Traverse-object abstraction: Provides a mechanism to apply the ReFreSh framework repeatedly to achieve lock-free data series indexes..
- ReFreSh framework: A generic approach that can be applied on top of any locality-aware blocking data structure to make it lock-free.
- FreSh design: first lock-free data series index.
- FreSh implementation: Introduces new lock-free tree and traverse-object implementations to achieve lock-freedom efficiently
- The first lock-free data series index supporting dynamic batch insertions of updates.
- Extends the FreSh implementation to support dynamic updates, addressing the challenge of maintaining lock-freedom during concurrent insertions..

The proposed lock-free tree incorporates several novel ideas that go beyond just engineering improvements. Previous approaches required copying and locally updating a leaf node L upon insertion, followed by replacing it in the shared tree,

which results in poor performance. We designed a novel algorithm for updating leaves that employs concurrent counters, announcements, handshaking, and other techniques to improve efficiency. The expeditive-standard mode transition also posed a challenge, as it requires synchronization when switching between modes.

Contributions. Our contributions are summarized as follows.

• We develop a comprehensive theoretical framework for supporting lock-freedom systematically in highly-efficient data series indexes. This includes introducing key concepts such as the traverse object, locality-awareness, and locality-aware lock-freedom, as well as identifying the desirable principles and properties for achieving optimal performance. This framework forms the basis for ReFreSh, a generic approach that can be applied to any locality-aware data series algorithm to ensure lock-freedom

• Based on ReFreSh, we develop FreSh, the first lock-free, efficient data series index. We present new lock-free implementations of several data structures necessary to support the required functionality, demonstrating the feasibility and performance of a lock-free approach.

• Building on the design of FreSh, we extend it to DFreSh, a lock-free data series index that supports dynamic batch insertions of updates. This is the first solution to maintain lock-freedom while allowing real-time updates, ensuring that queries always return consistent results, including data from previous updates

• Our experiments, using large synthetic and real datasets, show that FreSh performs as well as the state-of-the-art blocking index, without sacrificing performance for lock-freedom. In many cases, it even outperforms existing solutions. Additionally, FreSh significantly outperforms several lock-free baselines, which rely on conventional lock-free techniques, due to its ability to preserve locality-awareness..

• We introduce a theoretical framework using the traverse object, enabling the development of locality-aware data series indexes in a modular way. This framework allows for a systematic application of ReFreSh to transform locality-aware blocking algorithms into efficient lock-free versions.

• TODO: ADD DFreSh contributions.

1.1 Other Related Work

Numerous tree-based techniques for efficient and scalable data series similarity search have been proposed [17, 18, 16, 14], including approximate [4, 47] and progressive [37, 45, 48, 15] solutions. Among these, iSAX-based indexes [52] have proven to be particularly competitive in terms of both index construction and query performance [17, 18, 13, 11, 69]. These indexes also include parallel and distributed solutions that leverage modern hardware (e.g., SIMD, multi-core, multi-socket, GPU), such as ParIS+ [56], MESSI [57], and SING [58], as well as distributed approaches like DPiSAX [72, 73] and Odyssey [11]. The first lock-free concurrent search tree implementation was proposed in [21]. Building on the ideas from that

paper, we develop a baseline algorithm, which we discuss and compare experimentally with *FreSh* in Section 6. Several other non-blocking concurrent search trees have been introduced in the literature [8, 39, 2, 42, 50, 10, 7, 19, 33, 3]. The key novelty of our tree implementation, presented in chapter 5, is its ability to concurrently perform multiple insert operations in a lock-free manner to update the array in a (fat) leaf. Additionally, it supports the expeditive-standard mode of execution. These advancements result in improved parallelism and better performance. Our approach focuses solely on the functionality required for implementing traversal objects, whereas the aforementioned implementations support a variety of other features or are designed for different contexts. Concurrent priority queues have been explored in [1, 61, 71, 64, 65, 49], none of which are based on sorted arrays or support different execution modes. In our baseline lock-free implementations, we use a skip-list-based priority queue [49], which has been shown to perform well. However, our experiments indicate that the priority queue design we implemented for *FreSh* significantly outperforms this approach (Chapter 6). Universal constructions [25, 26, 27, 28, 24, 29, 30, 20, 31] can provide wait-free or non-blocking concurrent versions of any sequential data structure. However, due to their general nature, they are often less efficient than implementations tailored to specific data structures. The algorithms in [25, 27, 31] are highly efficient for small shared objects (e.g., stacks and queues) but are not suitable for our application. The concept of transforming algorithms to achieve different progress guarantees is not new, as seen in [66, 34, 38], though these transformations address different problems. The technique used in *FreSh*, called *ReFreSh* departs from all these prior approaches.

Chapter 2

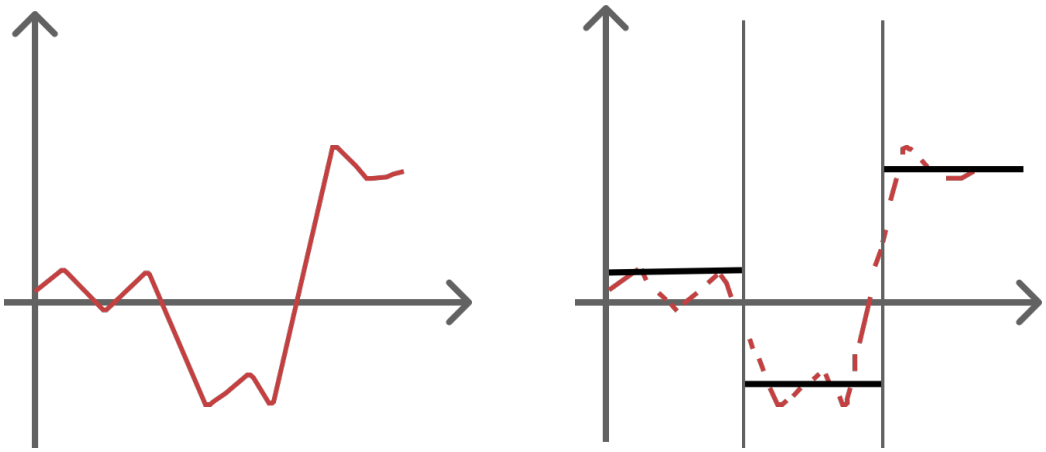
Preliminaries

2.1 Data Series And Similarity Search

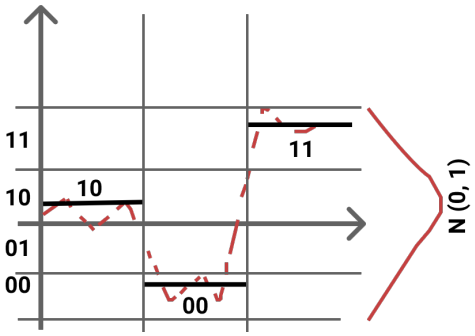
A **data series (DS)** of size (or dimensionality) n is a sequence of n pairs, where each pair consists of a real value and its corresponding dimension. The **Piecewise Aggregate Approximation (PAA)** [46] provides a compact representation of a data series by dividing the x-axis into w equal segments. Each segment is then represented by the **mean value** of the corresponding points, as shown by the black horizontal lines in Figure 2.1b. The PAA representation of a data series $T = (t_1, t_2, \ldots, t_n)$ is computed by dividing it into w segments, each containing $\frac{n}{w}$ data points. The i-th PAA coefficient $\bar{t}_i$ is given by:

$$\bar{t}_i = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} t_j \tag{2.1}$$

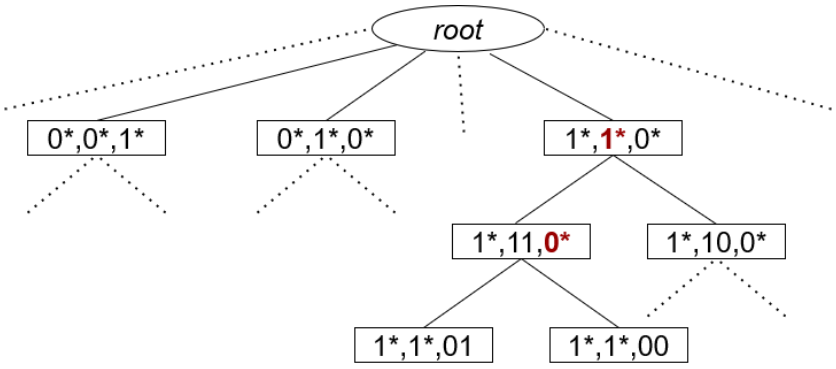
where the summation aggregates all values in the segment, and the factor $\frac{w}{n}$ ensures proper scaling. To compute the **iSAX summary** [62], the y-axis is partitioned into c discrete regions, where c represents the **cardinality** of the representation. Each region is assigned a unique bit pattern, and instead of storing the raw PAA values, iSAX encodes each segment using the bit representation of the region it falls into. This forms a **symbolic representation** of the series, such as the word $10_2 00_2 11_2$ in Figure 2.1c (where subscripts indicate the number of bits used per segment). The number of bits used per region can vary, enabling the construction of a **hierarchical tree index**, known as an **iSAX-based tree index**, as illustrated in Figure 2.1d. The index is a **leaf-oriented tree**, where each leaf stores up to M keys. During insertion, if the target leaf ℓ has available space, the new key is simply added. However, if ℓ is full, it undergoes a **split operation**, where it is replaced by a small subtree consisting of an **internal node** and two new leaves, which inherit the keys from ℓ. If one of the newly created leaves remains empty, the splitting process continues recursively. For further details on iSAX-based indexes, refer to [52].



(a) Data Series (b) PAA Summary



(c) iSAX Summary



(d) iSAX Tree

Figure 2.1: From data series to iSAX index

Similarity Search We focus on *exact similarity search* (also known as exact *1-NN*), which retrieves the data series from a collection that is most similar to a given query series. Similarity is typically measured using **Euclidean Distance (ED)**, but our techniques are general enough to support other widely used *similarity measures*, such as Dynamic Time Warping (DTW) [60]. The **Euclidean distance** between two time series $T = \{t_1, ..., t_n\}$ and $T' = \{t'_1, ..., t'_n\}$ is defined as:

$$ED(T, T') = \sqrt{\sum_{i=1}^{n}(t_i - t'_i)^2}$$

We refer to the distance between the *iSAX summaries* of two data series as the **lower-bound distance**. If we transform the original subsequences into PAA representations, $\overline{T}$ and $\overline{T'}$, using Eq. 2.1, we can then obtain a lower bounding approximation of the Euclidean distance between the original subsequences by:

$$DR(\overline{T}, \overline{T'}) \equiv \sqrt{\frac{n}{w}}\sqrt{\sum_{i=1}^{w}(\bar{t}_i - \overline{t'}_i)^2}$$

The calculation of this distance guarantees the **pruning property**: the lower-bound distance between two data series is always less than or equal to their Euclidean distance, which we refer to as the **real distance**. This property enables efficient pruning of candidates during query processing: a data series can be **pruned** if its lower-bound distance to the query series Q is greater than the real distance of any other data series in the collection from Q.

Leaf-Oriented Trees. In a *leaf-oriented tree*, all data are stored in the leaves, with each leaf capable of holding up to M keys. During an insertion, if the target leaf ℓ has available space, the new key is simply added to ℓ. However, if ℓ is full, it undergoes a **split**: it is replaced by a subtree consisting of an internal node and two new leaves. The keys from ℓ are then redistributed between the new leaves based on their values. If one of the newly created leaves remains empty after redistribution, the splitting process is repeated until both leaves contain keys.

2.2 iSAX-Based Indexing

Concurrent iSAX-based indexes [54, 56, 55, 57, 58] operate in two main phases: the *tree index construction phase* and the *query answering phase*, each utilizing distinct data structures.

Tree Index Construction. During this phase a set of *worker threads* processes a collection of input data series (i.e., *raw data*). Each series is summarized using an iSAX representation and inserted into a *tree index* as a pair of an iSAX summary and a pointer to the corresponding data series. The process is divided into two main stages:

- **Buffers Creation:** iSAX summary pairs are first stored in array-based *summarization buffers*.

- **Tree Population:** Worker threads traverse these buffers and insert their entries into the tree index.

Data series with similar iSAX representations are placed in the same buffer and later within the same subtree of the index tree. This approach ensures **high parallelism**, **good locality**, and **low synchronization overhead** during index construction.

Query Answering. Given a query data series Q, the system follows these steps to find the data series with the smallest distance to Q :

1. The iSAX summary of Q is computed and used to traverse the index tree, leading to a leaf ℓ.
2. The *real distance* between Q and each data series in ℓ is computed.
3. The smallest distance found so far is stored in the **Best-So-Far (BSF)** variable, serving as an initial approximate answer.

Then query answering proceeds by exexuting the following two stages namely the *Prunning Stage* and the *Refinement Stage*:

Pruning Stage: In this stage, a set of *query answering threads* traverse the tree. If the *lower bound distance* from a node to Q exceeds the Best So Far (BSF) value, the node is *pruned* and excluded from further processing. This pruning ensures that no pruned data series can contribute to the final answer.

Refinement Stage: During refinement, the non pruned data series are the candidate series and are stored in one or more *priority queues* [56, 57, 58]. Multiple threads process these candidate data series, calculating their exact distances to Q. The BSF is updated whenever a smaller distance is encountered.

At the end of this phase, the final answer is contained in BSF. *Barriers* synchronize threads at the end of each stage, ensuring correctness, while *locks* handle concurrent access to shared data structures. Figures 2.2 and 2.3 summarize the indexing process.

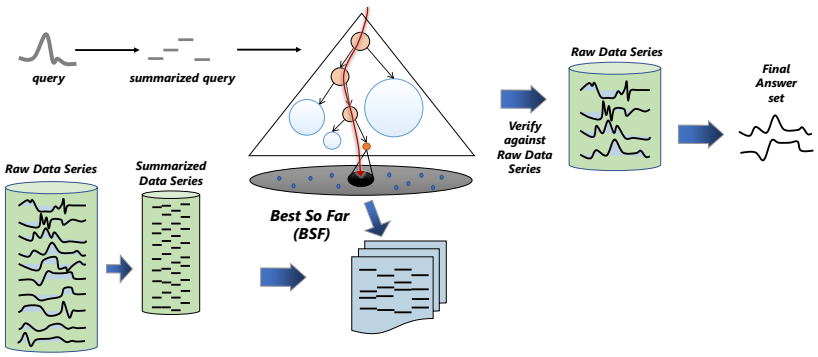


Figure 2.2: Similarity search with the use of a data series index.

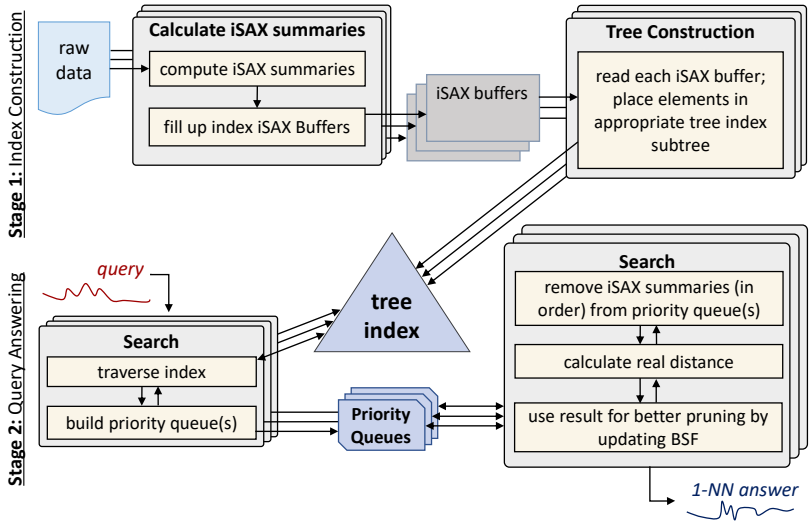


Figure 2.3: Index building and query answering flowchart for the MESSI data series index.

MESSI as an example. MESSI [55] is a state-of-the-art, in-memory iSAX-based index. It uses an array, referred to as *RawData*, to store the raw data. During the *buffers creation stage*, this array is split into a number of fixed-size chunks containing consecutive raw data series. A Worker thread repeatedly *acquires* and processes a chunk, storing the calculated iSAX summaries in the appropriate summarization buffers. Threads determine which chunks to work on using a *FAI* object. Each thread is allocated its own space in each summary buffer to avoid collisions when adding elements, ensuring thread safety. This process continues until all data series in *RawData* have been processed. During the *tree population stage*, worker threads again use *FAI* to *acquire* iSAX buffers to work on. Each subtree of the index tree is a binary leaf-oriented tree with fat leaves. In the *query answering*

phase, a query answering worker repeatedly *acquires* a subtree (using FAI) and *traverses* it by calculating the *lower-bound distance* between the query series Q and the iSAX summary of each node encountered. If the lower-bound distance of a leaf is smaller than the Best So Far (BSF) distance, the leaf is inserted into a *priority queue*, with the distance used as the priority. The algorithm uses a set of *priority queues* and threads insert elements into these queues in a round-robin fashion. Since a priority queue may be concurrently accessed by multiple threads, MESSI employs a coarse-grain *lock* for each queue for synchronization. During the *refinement phase*, each query answering thread t is assigned a priority queue PQ to process. It repeatedly removes the leaf with the minimum priority from PQ and compares its iSAX summary to the BSF. If the leaf's summary is smaller, the real distances between the data series stored in the leaf and the query series are computed. Otherwise, the leaf and all remaining nodes in the queue are pruned. Since multiple threads may process a priority queue concurrently, it is protected by a coarse-grain lock. Once processing of a priority queue is complete, thread t moves on to the next priority queue in a round-robin manner. *Barriers* are used among threads at the end of each stage and before the start of the next to ensure correct synchronization and maintain the integrity of the process.

2.3 System

We consider a shared-memory system with N threads that execute *concurrently and asynchronously* while communicating by accessing shared objects. A shared object O can be atomically read or written. Additionally, the operation FAI(O, v) atomically reads the current value of O, adds the value v to it, and returns the value that was read. The operation $CAS(O, u, v)$ 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 [43]), or they may fail by crashing (e.g., due to software errors). An algorithm is said to be *blocking* if a thread must wait for actions to be performed by other threads in order to make progress. *Lock-freedom* guarantees that some thread makes progress at each point in time, thus the system as a whole continues to make progress independently of the speed of threads or their failures. Lock-Free algorithms do not utilize locks, MESSI is not lock-free; it is blocking.

2.4 Dynamic Data

To simulate dynamicity in data we assume that all data is initially stored in memory, but only a small subset is used to create the initial index. The rest of the data are organized into batches. After the initial index is built, new data arrives dynamically in batches, with the batch size configurable by the user. To simulate real-time data ingestion, we introduce a delay interval between consecutive

batches. This delay, which can be defined by the user, begins as soon as a batch arrives. By appropriately tuning with the delay we simulate different dynamicity patterns in data. The behavior of the system depends on the interplay between batch size and delay interval. If the delay is determined to be long enough, index workers (i.e., threads responsible for building the index) may fully integrate each batch into the index before the next batch arrives, remaining idle for the rest of the time. Conversely, if the delay is determined to be short or the batch size is large, new batches may arrive before previous ones have been fully processed. While index workers are responsible for inserting new data, query workers (i.e., threads responsible for answering queries) simultaneously access this data to perform exact similarity search. Ensuring correctness in such a system requires a robust mechanism to manage concurrent updates and queries. To achieve this, we employ two models inspired by different domains of computer science: a *timestamp-based model* and a *consistency model based on linearizability.*

2.4.1 Timestamp-Based Ordering And Linearizability

A well-known approach for handling dynamic data is the use of timestamps to enforce ordering constraints. This method is widely used in database systems to ensure that data is processed in a meaningful manner [5]. Notable systems employing timestamp-based mechanisms include Apache Kafka and Google Spanner [67, 12]. In our approach, timestamps are implicitly generated using the system's clock. Specifically, every batch is assigned a timestamp corresponding to the time when the processing of the batch starts. This timestamp plays a crucial role in how queries behave. Specifficaly,

- Queries must see all data that previous queries have seen.
- If a query has a later timestamp than an ongoing batch, it must wait for that batch to finish processing before continuing.

By enforcing this ordering, we ensure that queries always operate on a consistent view of the data. Our approach is inspired by the principles outlined in *Models and Issues in Data Stream Systems* [5], which guided the design and implementation of timestamps in our dynamic system.

The second approach we employ is inspired by *linearizability* [41], a key concept in concurrent and distributed systems. Linearizability provides a strong consistency guarantee by ensuring that all operations appear to execute atomically in a single, consistent order.

Formally, a system execution is **linearizable** if:

- For each completed operation p, to insert a linearization point *p somewhere between p's invocation and response in a.
- To select a subset F of the incomplete operations, and for each operation r in F: to select a response, and to insert a linearization point *r somewhere after r's invocation in a.
- These operations and responses should be selected and these linearization

> points should be inserted, so that, in the sequential execution constructed by serially executing each operation at the point that its linearization point has been inserted, the response of each operation is the same as that in a

By leveraging the principles of linearizability, we ensure that concurrent updates and queries interact correctly, preserving system correctness while maintaining high concurrency. The Timestamp and Linearizability models complement each other by addressing different aspects of correctness in a dynamic, concurrent system, reflecting the principles of two distinct domains: the database world and the concurrent computing world. The timestamp model requires queries to wait for ongoing batches to finish before starting their processing. This ensures that any batch arriving before or concurrently with the query will be considered. This approach is generally preferred in the field of databases, as it respects the temporal sequence of batch arrivals and processing. On the other hand, linearizability offers more concurrency, as there is no waiting time. While data arrives, the algorithm is responsible for handling it, ensuring that the execution remains linearizable. This allows for concurrent operation throughout the entire process of insertions.

Chapter 3

Traverse Object

3.1 Traverse Object

In this section, we introduce the **traverse object**, an abstract data type that enables a modular design for iSAX-based indexes. The processing pipeline of an iSAX-based index consists of multiple stages, where each stage processes data produced by the previous one. The first stage handles the original dataset, while subsequent stages refine and structure the data progressively. This structured processing inspired the definition of the **traverse object**, whose sequential specification is given below.

3.1.1 Definition of the Traverse Object

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

- *Put($S, e, param$): Adds an element $e \in U$ to S. The optional parameter param allows implementations to pass additional arguments.*
- *Traverse($S, f, fparam, del$): Traverses S, applying the function f with parameters fparam to each element. If the del flag is set, the elements are deleted from S after being processed. The parameter fparam serves the same purpose as in Put.*

*A traverse object S satisfies the **traversing property**: Each instance of Traverse in a sequential execution applies f at least once to all distinct elements added to S that have not been deleted in a previous invocation of Traverse.*

3.1.2 Traverse Objects in iSAX-Based Indexing

To implement the different stages of an iSAX-based index, we use four instances of a traverse object to store and manipulate different types of data:

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

```
1: procedure InitializeSharedObjects
2:     BC← initially contains all raw data series
3:     TP , PS, RS← initially empty
4:     BSF ← some initial value
5: Code for thread t_i, i ∈ {0, ..., n − 1}:
6: procedure QueryAnswering(QuerySeriesSet SQ) returns int
7:     BC.Traverse(&BufferCreation(), BCParam, False)
8:     TP .Traverse(&TreePopulation(), TPParam, False)
9:     PS.Traverse(&Prunning(), PSParam, False)
10:    RS.Traverse(&Refinement(), RSParam, True)
11:    return BSF
12: procedure BufferCreation(DataSeries ds)
13:    iSAXSummary iSAX ← Calculate the iSAX summary for ds
14:    Index bind ← index to appropriate buffer based on iSAX
15:    TP .Put(⟨iSAX, index of ds ⟩, bind)
16: procedure TreePopulation(Summary iSAX, Index ind, Index bind, Boolean flag)
17:    PS.Put(⟨iSAX, ind⟩, bind, flag)
18: procedure Prunning(DataSeries Q, DataSeriesSet E,
    Boolean flag) returns boolean
19:    iSAXSummary iSAX ← Calculate the iSAX summary for E
20:    int lbDist ← lower bound distance between iSAX and Q
21:    if lbDist < BSF then
22:        RS.Put(⟨E, iSAX⟩, flag) return True
       return False
23: procedure Refinement(DataSeries Q, DataSeriesSet E, Summary iSAX, Function
    *UpdateBSF) returns Boolean
24:    int lbDist, rDist
25:    lbDist ← lower bound distance between iSAX and Q
26:    if lbDist < BSF then
27:        for all pair ⟨iSAX_ds, ind_ds⟩ in E do
28:            lbDist ← lower bound distance between iSAX_ds and Q
29:            if lbDist < BSF then
30:                rDist ← real distance between ds and Q
31:                if rDist < BSF then
32:                    *UpdateBSF(BSF, rDist)          ▷ user-provided routine
           return True
33:    elsereturn False
```

- **Buffer Creation** (BC): Stores the raw data series.
- **Tree Population** (TP): Stores pairs of iSAX summaries and pointers to data series.
- **Pruning** (PS): Organizes these pairs into sets corresponding to the leaf nodes of the tree.
- **Refinement** (RS): Maintains priority queues of candidate series for query answering.

Each of these objects plays a crucial role in the indexing and query answering process. The **buffer creation** phase utilizes an array, *RawData*, to store the raw data series. Thus, elements of BC are stored in *RawData*. The **tree population** phase employs a set of **summarization buffers**, where iSAX summaries and pointers are initially stored. The traverse object TP holds these pairs. The **pruning** stage organizes these pairs within a leaf-oriented tree, where PS manages sets of pairs corresponding to each tree leaf. Finally, the **refinement** phase employs priority queues to manage tree leaves containing candidate series for further evaluation.

3.1.3 Query Answering as a Sequence of Traversals

Query answering in an iSAX-based index follows a structured sequence of four *Traverse* operations, applied in order to the different traverse objects. Algorithm 1 presents the pseudocode for implementing an iSAX-based index using traverse objects.

Since the four stages do not overlap, their execution is typically synchronized using barriers. In Algorithm 1, these barriers (as well as any multithreading aspects) are embedded within the implementations of *Put* and *Traverse*. This ensures that an iSAX-based index satisfies the following key property:

Definition 2 (**Non-Overlapping Property**)**.** *In every concurrent execution of the index, for every traverse object S, an instance of Traverse on S is performed only after all Put operations that add distinct elements to S have completed.*

3.1.4 Ensuring Correctness Through Traversing and Non-Overlapping Properties

Assuming that the **non-overlapping property** holds for BC, TP, PS, and RS, and that *RawData*initially contains all data series in the collection, the **traversing property** guarantees that:

1. The BufferCreationfunction is invoked at least once for each data series in *RawData*.

2. The corresponding iSAX summary is computed, ensuring that at least one appropriate pair is inserted into TP.

3. By the **non-overlapping property**, the TreePopulationfunction processes all these pairs, inserting them into PS (the index tree).

4. The Prunningfunction is applied to all elements in PS, and leaves that cannot be pruned are inserted into RS.

5. The Refinementfunction is applied to every traversed element of RS. Since *Traverse* on RS is invoked with $del = True$, priority queues can be used to implement RS, and DeleteMincan be employed to remove elements as they are processed.

Thus, the system ensures that only relevant candidates are further processed, refining the query results by computing exact distances and updating the **best-so-far** (BSF) value when necessary.

Implementations of *Put* and *Traverse* for BC, TP, PS, and RS in FreSh are provided in chapter 5.

3.2 Adapting the Traverse Object for Dynamic Data Processing

To support dynamic data processing in an iSAX-based index, the *traverse object* requires adaptation. In a dynamic setting, threads execute concurrently and independently across the two distinct phases of the index: *index creation* and *query answering*. Consequently, the BC and TP traverse objects are assigned to the *index workers* (threads responsible for constructing the index), while the PS and RS traverse objects are assigned to the *query workers* (threads responsible for answering queries).

3.2.0.1 Impact of Concurrency on Traverse Objects

The concurrent execution of these phases alters the procedure for answering queries using traverse objects (see Algorithm 2). Unlike the static setting, where the procedure involves four sequential invocations of Traverse (one for each traverse object), in the dynamic setting, the sequence is reduced to only the last two traverse objects, PS and RS.

This change directly affects the *non-overlapping property* (Definition 2), which now applies only to traverse objects within the same phase:

- **Index workers:** BC and TP
- **Query workers:** PS and RS

Since TRAVERSE and PUT operations can occur concurrently across phases, situations arise where an index worker inserts new elements into the iSAX tree while query workers are simultaneously pruning or refining their results. This requires additional mechanisms to maintain correctness.

3.2.0.2 Handling Multiple Batches

In a dynamic environment where data arrives in batches, certain data structures implementing traverse objectsb•"specifically, *BC* and *PS*b•"must be *re-initialized* before processing a new batch. This re-initialization is necessary because:

- Each batch reuses the same buffers, meaning helpers must distinguish between **current** and **obsolete** data.
- Helpers need to **skip outdated data** that no longer belongs to the active batch.
- This differentiation is particularly important during the traversal of TP and *PS* objects.

3.2.0.3 Ensuring Correctness with Sequence Numbers

To address these challenges, our approach introduces *sequence numbers* as a mechanism to track the state of the algorithm.

Definition 3. *A **sequence number** is a monotonically increasing integer used to track the order of operations and ensure consistency in multi-threaded or distributed environments.*

Each *summarization buffer* is associated with a sequence number. During **re-initialization**, an index worker performs the following steps:

1. Resets the buffer size to zero.
2. Increments the buffer's sequence number by one.

This mechanism prevents a thread from processing outdated data. If a worker reads a nonzero buffer size, it indicates the presence of data. However, before proceeding, the worker also checks the **sequence number**:

- If the sequence number **matches** the current batch ID, the data is valid and can be processed.
- If the sequence number is **smaller** than the current batch ID, the data is obsolete and must be ignored.

This ensures that no worker mistakenly processes data from a previous batch (see Algorithm 37).

Algorithm 2 Implementation of a dynamic iSAX-based index using the traverse objects BC, TP .

```
1: procedure InitializeSharedObjects
2:     BC← initially contains all raw data series
3:     TP , PS, RS← initially empty
4:     BSF ← some initial value
5:     GlobalSeq ← 0
6: Code for thread t_i, i ∈ {0, ..., k − 1}:
7: procedure IndexCreation(int TotalUpdates) returns int
8:     int currentBatch ← 1
9:     while currentBatch < TotalBatches do
10:        if model == SystemTS then
11:            batchTS ← getTS()
12:        BC.Traverse(&BufferCreation(), BCParam, False)
13:        TP .Traverse(&TreePopulation(), TPParam, False)
14:        for all summarization buffers that have data do
15:            ReuseBuffers(summBuffer, currentBatch)
16:        if DELAY >= BatchProcessingTime then
17:            sleep(DELAY − BatchProcessingTime)
18:        currentBatch ← currentBatch + 1
19: Code for thread t_i, i ∈ {k, ..., n − 1}:
20: procedure QueryAnswering(QuerySeriesSet SQ) returns int
21:    int localTS ← 0
22:    while !SQ.Empty do
23:        if model == LogicalTS then
24:            if localSeq == GlobalSeq then
25:                CAS(&GlobalSeq, localSeq, localSeq + 1)
26:        else if model == SystemTS then
27:            queryTS ← getTS()
28:            while queryTS >= batchTS do
29:                backoff()
30:        PS.Traverse(&Pruning(), PSParam, False)
31:        TP .Traverse(&Refinement(), RSParam, False)
32:        for all summarization buffers that have data do
33:            ReuseBuffers(summBuffer, currentBatch)
34:        if model == LogicalTS then
35:            localSeq = localSeq + 1
36:    return BSF
37: procedure ReuseSumBuffers(SumBuff * currBuff , int currentBatch)
38:    currBuff− > size[workerID] ← 0
39:    if currBuff− > seq[workerID] < currentBatch then
40:        currBuff− > seq[workerID] ← currentBatch
41:    currBuff− > seq[workerID] + +
```

Chapter 4

Locality-Awareness

Locality-awareness aims at capturing several design principles (Definition 4) for data series indexes which are crucial for achieving good performance. A locality aware implementation respects these principles.

Definition 4. *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 as much as possible. Whenever synchronization cannot be avoided, design the appropriate 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

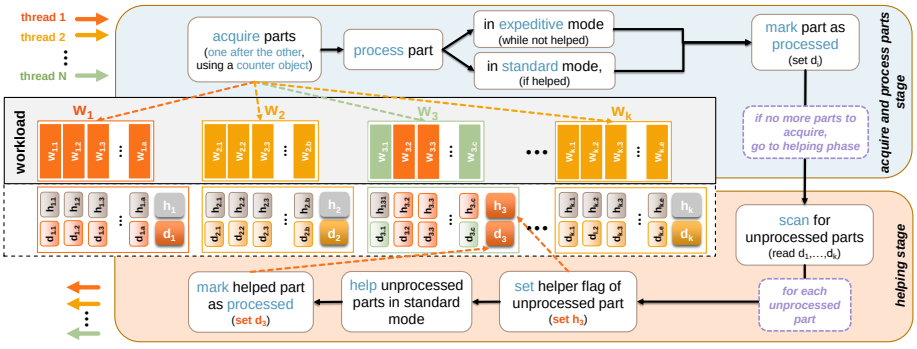


Figure 4.1: Refresh flowchart.

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*. 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. The main idea behind it is to require from each thread that completes processing its own workload,(instead of blocking, waiting for other threads to also finish), to scan for unprocessed workloads (e.g. in an iSAX-based index, unprocessed parts of *RawData* or unprocessed summarization buffers), and help completing their processing.

ReFreSh (Algorithm 3) 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 for each data structure D of $\mathcal{A}$, the following steps (depicted in Figure 4.1):

Algorithm 3 *ReFreSh* - A general approach for transforming a blocking data structure D into a lock-free one.

```
1: procedure InitializeSharedVariables
2:     Workload part W ← [w_1, w_2, ..., w_k]
3:     Boolean array F ← [d_1, d_2, ..., d_k], initially d_i = False
4:     Boolean array H ← [h_1, h_2, ..., h_k], initially h_i = False
5: Code for each thread:
6: procedure Refresh
7:     while W has available parts do
8:         w_i ← Acquire an available part of W
9:         Mark w_i as acquired
10:        if h_i == False then
11:            Process w_i in expeditive mode, while checking that h_i remains False
12:            If h_i == True, switch to standard mode
13:        else
14:            Process w_i in standard mode
15:        d_i ← True
16:    for all d_i ∈ D where d_i == False do
17:        Backoff()                                   ▷ Avoid helping, if possible
18:        if d_i == False then
19:            h_i ← True
20:            Process w_i in standard mode, checking periodically if d_i remains False
21:            If d_i == True, stop processing w_i
22:            d_i ← True
```

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 15).

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 7-15). The thread that acquires a workload is its *owner*.

3. To achieve lock-freedom, every thread t, then, *scans* all the flags (d_i, $1 \leq i \leq k$) to find those parts that are still unfinished (line 16).

4. Thread t *helps* by processing, one after the other, each part found unfinished during scan. For each part w_i, $1 \leq i \leq k$, 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 20). A thread that completes the processing of w_i changes d_i to `True` (line 22).

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 $\mathcal{A}$.

In locality-aware implementations, (e.g., in iSAX-based indexes), 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`) , $1 \leq i \leq k$, is attached with each w_i. , which indicates whether w_i's processing should be performed on expeditive or standard mode. A thread t starts by processing its assigned workload on expeditive mode (lines 4 and 10-11). Before t starts helping some part w_i, it sets h_i to `True` (line 19), to alert w_i's owner thread to start running on standard mode (line 11).

To avoid helping whenever it is not absolutely necessary, i.e. when no thread has failed (or is extremely slow), *RefreSh* provides an optional *backoff scheme* that is used by every thread t (line 17) before it attempts to help other threads (line 18-20). 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. 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. and not the whole w_i. Thus, the redundant work is further decreased. To achieve this, the subparts of w_i should have their own flag and helping-indicator variables.

Lock-freedom is ensured due to the *helping code* (lines 16- 22), 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.

4.0.1 ReFreSh with Dynamic Batches

ReFreSh is designed to be applied once per workload. Threads begin execution in *expeditive mode* and switch to *standard mode* when a helper arrives. However, in an iSAX-based index, *ReFreSh* is applied to multiple phases, including the RawData array, summarization buffers, the iSAX tree, and the sorted arrays. When data arrives dynamically in batches, *ReFreSh* must be applied separately for each phase and its corresponding data structures. For data structures accessed only once per batch (e.g., the RawData array), this does not introduce issues. However, for data structures accessed multiple times (e.g., summarization buffers and the iSAX tree), a challenge arises. The core issue lies in how *ReFreSh* determines when to switch execution modes. Its behavior relies on boolean flags that initially have a value of `False` and transition to `True` during processing. After processing the first batch, subsequent batches will encounter these flags already set to `True`, causing *ReFreSh* to always run in *standard mode*. This is suboptimal because *standard mode* enables helping, which is unnecessary unless a helper has actually arrived. To maintain performance, *ReFreSh* must be modified to support batch processing effectively.

4.0.1.1 Modifications to Support Batches

To ensure *ReFreSh* works correctly with dynamic batches, we introduce the following modifications:

1. **Replacing Boolean Flags with Counters:** Instead of using boolean flags, we replace them with counters. Each part w_i, where $1 \leq i \leq k$, is assigned a counter d_i (initially set to 0). This counter tracks whether w_i has completed processing for a specific batch. Once a thread finishes processing w_i, it sets d_i to $currentBatch + 1$, which corresponds to the ID of the next batch (see Algorithm 4).

2. **Batch-Aware Execution of *ReFreSh*:** *ReFreSh* is executed once per batch. Each time it runs, it requires the ID of the batch it is processing. The counter values are interpreted accordingly:

- A part w_i is considered processed for a batch with ID X when $d_i = X+1$.
- A helper has arrived at part h_i for an update batch with ID X when $h_i = X + 1$.

By implementing these modifications, *ReFreSh* maintains its efficiency across multiple dynamic batches while ensuring that execution mode transitions occur only when necessary.

Algorithm 4 Dynamic Refresh - A general approach for transforming a blocking data structure D of a big-data application $\mathcal{A}$ into a lock-free one that supports dynamic insertions.

```
1: Shared variables:
2: Workload part W := [w_1, w_2, ..., w_k]
3: int F := [d_1, d_2, ..., d_k], initially d_i = 0, ∀i ∈ [1, k]
4: int H := [h_1, h_2, ..., h_k], initially h_i = 0, ∀i ∈ [1, k]
5: Code for each thread:
6: procedure DRefresh(int currentBatch)
7:     while W has available parts do
8:         w_i ← acquire an available part of W
9:         Mark w_i as acquired
10:        if (h_i == currentBatch) or (d_i == currentBatch and h_i < currentBatch) then
11:            Process w_i in expeditive mode, while checking the value of h_i
12:            if h_i == currentBatch + 1 then
13:                Switch to standard mode
14:        else
15:            Process w_i in standard mode
16:        if d_i == currentBatch then
17:            CAS(&d_i, currentBatch, currentBatch + 1)
18:    for all d_i ∈ D where d_i == currentBatch do
19:        Backoff to avoid helping if possible
20:        if d_i == currentBatch then
21:            if h_i == currentBatch then
22:                CAS(&h_i, currentBatch, currentBatch + 1)
23:            else if h_i < currentBatch then
24:                int oldVal ← h_i
25:                CAS(&h_i, oldVal, currentBatch + 1)
26:        Process w_i in standard mode, checking d_i
27:        if d_i == currentBatch then
28:            CAS(&d_i, currentBatch, currentBatch + 1)
```

1. As in *ReFreSh*, threads acquire parts of the batch W for processing until all parts have been claimed. Each thread must determine the appropriate processing mode. A helper has arrived at part w_i if the value of h_i is equal to the current batch ID + 1. The same applies to the processing of d_i. Once a thread finishes processing a part, it must atomically increment d_i using a compare-and-swap (CAS) operation to ensure correctness, as multiple helpers may attempt to update it simultaneously (lines 7- 17).

2. To maintain lock-freedom, every thread t scans all counters (d_i, $1 \leq i \leq k$) to identify unfinished parts (line 18).

3. Thread t *helps* by processing, one after another, each part that remains unfinished after the scan. A part w_i is considered unfinished when its corresponding d_i value is equal to the current batch ID. While processing, thread t periodically checks d_i to determine whether other threads have already completed w_i. For each part w_i that t helps, it must announce that a helper has arrived by updating the value of h_i. In the base case, where helping occurred for this part in the previous batch, h_i is updated atomically using CAS (line 21). However, there are cases where h_i may lag behind for certain parts, as helping is not always required in every batch. For example, if a thread completes processing a part before a helper arrives, the corresponding h_i value remains unchanged (line 23). Once a thread finishes processing w_i, it attempts to update d_i to the current batch ID + 1 (line 28).

Chapter 5

FreSh

We follow the data processing flow, described in Chapter 3, employ *BC*, *TP*, *PS*, and *RS* (from Section 3), and repeatedly apply *ReFreSh* (from Section 4) to come up with *FreSh*, the first lock-free locality-aware data series index.

5.0.1 Counter Object

Not sure if we have to include this!!!! Found it on old versions

Many of our implementations use a *counter object*, which supports the operation NextIndex (Algorithm 5) which returns a positive integer. Let NextIndex be invoked K times. If crashes do not occur, every index in the range $R = \{0, \ldots, K-1\}$ should be returned exactly once; (i.e. by a distinct of the K invocations) each integer is returned exactly once otherwise, as many indexes from R, as the number of crashed threads, are not returned, but the rest are returned exactly once.

Algorithm 5 uses two counters. The first, *cntEM*, is used in the expeditive mode and is updated by simple writes (line 11), whereas the second, *cntSM*, is used in the standard mode and is updated using *FAI* (line 25). As long as no helpers exist ($h =$ `False`), the owner thread t executes a code similar to the sequential code implementing a counter (lines 8- 13). When a helper arrives, it sets *cntSM* to the current value of *cntEM* (lines 14, 20), and uses *FAI* on *cntSM* (line 25) to get assigned new indexes.

Lines 15-25 provide a mechanism for appropriately synchronizing t with the helper threads when they arrive. Assume that t has read the value x in *cntEM*, and it is ready to write $x+1$ in *cntEM*, when two helper threads t_h and t'_h arrive. Thread t_h reads x in *cntEM*, then t writes $x+1$ in *cntEM* and t'_h reads $x+1$ in *cntEM*. Depending on which of the two values will be written in *cntSM* and the order in which the *FAI* instructions will be executed, one of t_h, t'_h may read x in *cntSM* and return it. In this case, t should avoid returning x a second time (to ensure the semantics of a counter object). To achieve this, helpers record in *first* the value with which they initialize *cntSM*. If t discovers that helpers exist (lines 13, 14), it may have to read *first* to figure out which is the first value recorded

Algorithm 5 Pseudocode for NEXTINDEX.

```
1: Shared variables:
2: int cntEM (initially 0), cntSM (initially ⊥)
3: function NEXTINDEX(*h) returns ⟨int, boolean⟩
4:     int pos
5:     int ownerFlag ← False
6:     while True do
7:         if *h = False then
8:             pos ← cntEM
9:             if *h == True then
10:                continue
11:            cntEM ← cntEM + 1
12:            if *h == False then
13:                return ⟨pos, True⟩
14:            ownerFlag ← True
15:        if cntSM = ⊥ then
16:            if ownerFlag ≠ True then
17:                pos ← cntEM
18:                CAS(first, ⊥, pos)
19:                pos ← first
20:            v ← CAS(cntSM, ⊥, pos)
21:            if v == ⊥ and ownerFlag = True then
22:                return ⟨pos, False⟩
23:            else if first < cntEM then
24:                continue
25:        pos ← FAI(cntSM)
26:        return ⟨pos, False⟩
```

in *cntSM*. If it is x, it retries (line 24). Note that helpers should first agree on the value to record in *cntSM* with the *CAS* of line 18 and then all use this value to initialize *cntSM* (lines 19- 20).

5.0.2 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 , $w_1, \ldots, w_k$. This way we have a number of 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 6.

RawData is comprised of k chunks, each containing m groups. Moreover, each group contains r elements (line 2). *FreSh* uses three sets of done flags, *DChunks*, *DGroups*, and *DElements* (line 3), 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 5), to count the chunks, groups and elements, assigned to threads for processing. *FreSh* also employs two sets of *helping* flags (line 4), *HChunks* (for helping chunks) and *HGroups* (for helping groups). For each $1 \leq i \leq k$, $HChunks[i]$ identifies whether there are helpers for chunk i. Similarly, for each $1 \leq j \leq m$, $HGroups[i][j]$ identifies whether there are helpers for group j of chunk i.

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*) on line 15 (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 15 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 19-27) follows the general pattern described in Algorithm 3.

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. This allows an owner thread that has not crashed, to finish the processing of its last chunk, while it still executes in expeditive mode, thus avoiding the cost of switching to and executing a standard mode. *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* (as other iSAX-based indexes) 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.

Algorithm 6 Pseudocode for TRAVERSE in FRESH. Code for thread t.

```
1: Shared variables:
2: Set RawData[1..k][1..m][1..r], initially containing all data series
3: Boolean DChunks[1..k], DGroups[1..k][1..m], DElements[1..k][1..m][1..r], initially all False
4: Boolean HChunks[1..k], HGroups[1..k][1..m], initially all False
5: CounterObject Chunks, Groups[1..k], Elements[1..k][1..m]
6: int Size[1..3] = {k, m, r}
7: Code for each thread:
8: procedure TRAVERSE(Function *BufferCreation, DataSeries RawData[], Boolean D1[],
   Boolean D2[], Boolean D3[], Boolean H1[], Boolean H2[], Boolean h, CounterObject Cnt1,
   CounterObject Cnt2[], CounterObject Cnt3[], int rlevel)
9:     int i
10:    while True do
11:        ⟨i, ∗⟩ ← Cnt1.NEXTINDEX(&h)
12:        if i > Size[rlevel] then break
13:        Mark RawData[i] as acquired
14:        if rlevel < 3 then
15:            Traverse(BufferCreation, RawData[i], D2[i], D3[i], D3[i], H2[i],
       NULL, H1[i], Cnt2[i], Cnt3[i], Cnt3[i], rlevel + 1)
16:        else
17:            ∗BUFFERCREATION(RawData[i])
18:        D1[i] ← True
19:    for all j such that D1[j] is False do
20:        Backoff()                                   ▷ Avoid helping if possible
21:        if D1[j] is False then
22:            H1[j] ← True
23:            if rlevel < 3 then
24:                Traverse(BufferCreation, RawData[j], D2[j], D3[j], D3[j], H2[j],
       NULL, H1[j], Cnt2[j], Cnt3[j], Cnt3[j], rlevel + 1)
25:            else
26:                ∗BUFFERCREATION(RawData[j])
27:            D1[j] ← True
```

Tree Population Stage. Similarly to the buffers creation stage, in tree population, the worker threads have to traverse and process the elements of TP , i.e. all pairs added in the summarization buffers. Processing is now achieved by calling the TREEPOPULATION function (Algorithm 1) for each pair. TREEPOPULATION finds the right subtree of the index tree to place each pair, and then simply calls PS.PUT to add the pair into PS, the next traverse object in the dataflow pipeline. To implement TP .TRAVERSE, we split the elements of TP into 2^w workloads as the number of summarization buffers, and apply *ReFreSh*. Each thread t repeatedly acquires summarization buffers using FAI, and process them to produce the corresponding trees. 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.

5.0.3 Prunning 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. TreePopulation simply transfers the pairs from a summarization buffer to the appropriate subtree of the index tree. To support the concurrent population of a subtree by multiple threads, *FreSh* utilizes Algorithm 9 To implement *PS*.Traverse, *FreSh* (Algorithm 7) 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 Prunning 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.

5.0.3.1 Insert in Leaf-Oriented Tree

Each node of the tree stores a key and the pointers to its left and right children. (refer to Algorithm 9). 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 which summarizes all data series stored in it. 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, (line 21) (i.e. $pos < M$), t stores the new element in $D[pos]$ (line 24). Otherwise, in case the array is full, i.e. $pos \geq M$, then t attempts to split the leaf (line 25). During spliting, D may contain empty positions, since some threads may have acquired positions in D but have not yet stored their elements there (line 24). 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 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 executes the following actions. It first calls a standard *Search* routine to traverse a path of the tree and find an appropriate *leaf* and its *parent* (line 9). Pointer *ptr* is a reference to the appropriate child field of *parent*, which needs to be changed to perform TreeInsert. (lines 10-15). Next, t accesses the counter object to acquire

Algorithm 7 Pseudocode for PUT and TRAVERSE of *PS* in *FreSh*. Code for thread $t \in \{1, \ldots, N-1\}$.

```
1: Shared variables:
2: TreeNode *IndexTree[1..2^w]
3: bool DTree[1..2^w], HTree[1..2^w], initially all false
4: CounterObject TreeCnt[1..2^w]
5: procedure TRAVERSE(Function *PRUNNING, TreeNode *T, CounterObject *Cnt, int x, bool h, int rlevel)
6:     int i
7:     while true do
8:         ⟨i, ∗⟩ = Cnt.NEXTINDEX(&h)
9:         if i > x then
10:            break
11:        if rlevel < 2 then
12:            mark IndexTree[i] as acquired
13:            totalNds ← TOTALNODES(IndexTree[i])
14:            Traverse(Prunning, IndexTree[i], TreeCnt[i], totalNds,
       false, rlevel + 1)
15:            DTree[i] := true
16:        else
17:            nd ← FINDNODE(T, i)
18:            mark nd as acquired
19:            PRUNNING(nd)
20:            mark nd as done
21:    for all j such that DTree[j] is false do
22:        BACKOFF                                  ▷ Avoid helping, if possible
23:        if DTree[j] == false then
24:            HTree[j] := true
25:            HELPTREE(PRUNNING, IndexTree[j])
26:        DTree[j] := true
27: procedure FINDNODE(TreeNode *T, int i)          ▷ Returns TreeNode*
28:    TreeNode *p ← T
29:    int nds ← 0
30:    while p ≠ NULL  &  nds ≠ i do
31:        if nds + p → cnt + 1 < i then
32:            nds ← nds + p → cnt + 1
33:            p ← p → rc
34:        else
35:            p ← p → lc
36:    return p
37: procedure HELPTREE(Function *f, TreeNode *T)
38:    if T == NULL then
39:        return
40:    HELPTREE(f, T → lc)
41:    if ∗T is unprocessed then
42:        ∗f(∗T)
43:    HELPTREE(f, T → rc)
```

Algorithm 8 Type Definitions for the Lock-Free Tree

```
   Type Definitions:
 1: Type Node:
 2:    int key
 3:    {Node, Leaf} *left
 4:    {Node, Leaf} *right
 5:    InsertRec Announce[0..n − 1]
 6:    Boolean helpersExist

 7: Type InsertRec:
 8:    Data data
 9:    int position

10: Type Leaf extends Node:
11:    Data D[0..m − 1]
12:    CounterObject Elements
```

a position in D (line 18) and proceeds to announce the data that it wants to insert in the tree (line 20). Afterwards, it announces this position in *Announce* and stores the data in $D[pos]$ (line 24), if D is not full (line 21). If D is full, it calls SplitLeaf to split *leaf*. If this *CAS* succeeds, then the data have been added and TreeInsert completes. Otherwise, some other thread has successfully split the node.

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 27-31). 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 34). This way, t discovers that the execution mode for l has changed (line 27 and first condition of line 28), and re-attempts its Insert (line 31).

Lemma 5.0.1. *Algor. 9 is a* linearizable, lock-free *implementation of a leaf-oriented tree with fat leaves, supporting only insert operations.*

5.0.4 Refinement

To implement *RS*, *FreSh* uses a set of priorities queues each implemented using an array (Algorithm 10). 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.

During query answering, an application may use the same index tree to answer more than one query. In that case, the done flags of the nodes and other variables need to be reset each time a new query starts. This may require syncrhonization. To avoid this case, *FreSh* implements the *done* flag as a counter (rather than as a boolean). This counter describes the number of queries for which the node has been processed.

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.

Lemma 5.0.2. **(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$(\&$PRUNING$, *,$ `True`$)$ *by t on RS completes, for every leaf ℓ in RS, ℓ either has been processed or it has been pruned.*

Theorem 5.0.1. *FreSh solves the 1-NN problem and provides a lock-free implementation of* QUERYANSWERING *(Alg. 1).*

Algorithm 9 TraverseTree: a lock-free leaf-oriented tree with fat leaves, implementing a traverse object. Code for thread $t \in \{1, \ldots, N-1\}$.

```
    Shared Variables:
 1: Tree ← NULL                                  ▷ Initially points to a Leaf with initialized values
 2: procedure TreeInsert(data, isHelper)
 3:     leaf ← NULL
 4:     parent ← Tree
 5:     ptr ← NULL
 6:     pos, val ← 0
 7:     expeditive ← False
 8:     while True do
 9:         ⟨leaf, parent⟩ ← Search(data, parent)
10:         if parent = NULL then
11:             ptr ← &Tree
12:         else if parent → left = leaf then
13:             ptr ← &parent → left
14:         else
15:             ptr ← &parent → right
16:         if isHelper = True and leaf → helpersExist = False then
17:             leaf → helpersExist ← True
18:         ⟨pos, expeditive⟩ ← NextIndex(&leaf → helpersExist)
19:         if expeditive = False then
20:             leaf → Announce[t] ← ⟨data, ⊥⟩
21:         if pos < M then
22:             if expeditive = False then
23:                 leaf → Announce[t].position ← pos
24:             leaf → D[pos] ← data
25:         else
26:             SplitLeaf(leaf, ptr, expeditive)
27:         if (∗ptr) → helpersExist = True then
28:             if expeditive = False and (∗ptr) → Announce[t].position ≠ ⊥ then
29:                 (∗ptr) → Announce[t] ← ⟨⊥, ⊥⟩
30:             else
31:                 continue
32:         return
33: procedure SplitLeaf(leaf, prt, expeditive)
34:     newNode ← NewNode
35:     newNode → left ← NewLeaf
36:     newNode → right ← NewLeaf
37:     splitBuffer ← ∅
38:     if expeditive = False then
39:         for i ∈ {0, ..., n − 1} where leaf → Announce[i].data ≠ ⊥ do
40:             ldata ← leaf → Announce[i].data
41:             if leaf → Announce[i].position ≠ ⊥ then
42:                 leaf → D[leaf → Announce[i].position] ← ldata
43:             else
44:                 AddToBuffer(ldata, splitBuffer)
45:             newNode → Announce[i] ← ⟨ldata, −1⟩
46:     Distribute(leaf → D, splitBuffer, newNode → left, newNode → right)
47:     CAS(∗prt, leaf, newNode)
```

Algorithm 10 Priority Queue of *FreSh*. Code for thread t.

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

7: procedure INSERT(int priority, Datavalues)
8:     int pos ← FAI(insPos)
9:     A[pos] ← ⟨priority, value⟩

10: procedure INITDELETEPHASE
11:     ⟨int, Data⟩ *sa
12:     sa ← allocate local array of insPos elements
13:     Copy non-⊥ elements of A into sa and sort them
14:     CAS(&SA, NULL, sa)

15: function DELETEMIN(boolean isHelper) returns Data
16:     if isHelper = True and helpersExist = False then
17:         helpersExist ← True
18:     int pos ← Cnt.NEXTINDEX(&helpersExist)
19:     if pos ≥ insPos then
20:         return ⊥
21:     return SA[pos].data
```

Algorithm 11 Type Definitions for TreeInsertion with Dynamic Batches

```
   Type Definitions:
 1: Type Node:
 2:    int key
 3:    {Node, Leaf} *left
 4:    {Node, Leaf} *right
 5:    InsertRec Announce[0..n − 1]
 6:    Int helpersExist

 7: Type InsertRec:
 8:    Data data
 9:    int position

10: Type Leaf extends Node:
11:    DynamicData D[0..m − 1]
12:    CounterObject Elements

13: Type DynamicData extends Data:
14:    int seq
```

5.0.5 FreSh with Dynamic Updates

5.0.6 Linearizable Dynamic Fresh

As discussed in previous chapters, we designed and implemented an extension of FreSh, called *DFreSh*, that supports insertions of dynamic batches while ensuring that queries always return the best possible answer and respect linearizability. In the pseudocode provided, we refer to this version as *LOGICAL*. This approach is completely lock-free, meaning that neither index workers nor query workers need to wait for each other as long as they have available work. For example, index workers can continue inserting batches while query workers process queries concurrently. To ensure correctness, we need a mechanism that guarantees a query with a specific timestamp always returns the same result. To achieve this, we introduce a sequence number mechanism. Specifically, we maintain a global shared counter, *GlobalSeq*, which is accessible to both index and query workers. Query workers read the current value of *GlobalSeq* and attempt to increment it by one before answering a query. A query must be able to see all data series inserted into the index with a timestamp smaller than or equal to its own before it is answered. The insertion process for index workers follows the same general approach as in FreSh, using the `TreeInsert` method (see Algorithm 12). However, the iSAX tree structure in this dynamic setting differs from that of FreSh. Each leaf node now contains an array D, where each element consists of a sequence number, an iSAX summary, and a pointer to the original data series stored in `RawData`. Despite these differences, the core insertion procedure remains similar to what was described in Chapter 5. An index worker first locates the appropriate leaf to insert a new element (line 9). It then attempts to acquire a position in the leaf node. If space is available (line 23), the data series is stored as before. However, for an element to be considered valid,

it must first be assigned a sequence number, which is done using a Compare-And-Swap (CAS) operation (line 32). Since queries are processed concurrently with insertions, a query worker may encounter elements in a candidate leaf during the *Refinement* stage. For each element in the leaf, the query worker first checks its sequence number. If the sequence number is smaller than or equal to the query's timestamp, the element must be considered in the query result. If it has a larger value, the query worker skips it. However, if the sequence number is unassigned (denoted by $\perp$), the query worker cannot ignore the element and must help assign a sequence number using a CAS operation (line 33). Since both index and query workers can assign sequence numbers, the CAS operation ensures correctness while the preceding conditional check optimizes performance by avoiding unnecessary failed CAS attempts. On the other hand, if the leaf is full, the index worker attempts to perform a split. During the split operation, the index worker first examines the announce array for any operations that have been announced but not yet placed into the leaf. If a thread has announced both the data and its position inside the leaf, the thread performing the split rewrites the data into that position, as was done in FreSh. Additionally, if the sequence number is unassigned, the worker helps assign one. However, if the position has not yet been announced, the data is temporarily stored in a buffer called *splitBuffer*. Next, the index worker distributes the elements stored in *splitBuffer*, along with those already in the leaf, to the new left and right child nodes based on their iSAX summaries. During this process, for elements originally from the leaf, the worker checks their sequence numbers and assigns one if necessary. This step is crucial because the leaf remains connected to the tree, making it visible to query workers, which may have included it in their query results. Once all elements are distributed, the worker attempts to finalize the split using a *CAS*operation, following the same approach as in FreSh. However, to fully complete the split, a traversal of the newly created child nodes is required. This is because elements moved from the *splitBuffer* may still have unassigned sequence numbers. Importantly, this traversal must not be performed before the split is established, as announce arrays are not visible to query workers. If these elements receive a sequence number smaller than or equal to that of an ongoing query before the split is finalized, it could lead to correctness violations.

There is one final scenario that can occur during splitting, which we address using the keyword *MARKED*. Consider the following case: an index worker announces its data and acquires a position in the leaf but becomes slow before announcing the position. Meanwhile, another index worker starts a split operation. Since the first worker has not yet announced its position, its operation is treated as incomplete and stored in the *splitBuffer*. While the split is ongoing, the first worker resumes execution and inserts its element into the leaf. At this point, the element becomes visible to queries and may receive a sequence number-either from the inserting thread or from a query worker. This can lead to a correctness violation because the same element might be assigned different sequence numbers before and after the delay. If the thread performing the split has already processed the position where the first worker is placing the element, it will not include it,

meaning the element will only receive a sequence number when the new child nodes are traversed after the split. If this element happens to be the best answer for an ongoing query, the query may or may not include it in its results, violating correctness. To prevent this issue, we introduce a marking technique. Before storing data into the leaf, a thread first checks whether a split is in progress by reading the value of *NextIndex*. If a split is occurring, the thread marks its position by setting the sequence number to *MARKED* using a CAS operation before inserting the data. A query encountering a *MARKED* position treats it as invalid. During the split operation, all marked positions are reset to $\perp$, ensuring that these elements receive a sequence number either from a query worker or when processing the new child nodes after the split.

Algorithm 12 DTraverseTree: a lock-free leaf-oriented tree with fat leaves, implementing a traverse object for dynamic batches. Code for thread $t \in \{0, \ldots, n-1\}$.

```
 1: Shared variables:
 2: {Node,Leaf} *Tree := NULL, initially pointing to a Leaf:
      ⟨key, NULL, NULL, ⟨⊥, ⊥⟩, ..., ⟨⊥, ⊥⟩⟩, False, ⟨⊥, ..., ⊥⟩, ⊥, 0⟩
 3: function TreeInsert(data, isHelper, mode, batchID)
 4:     Leaf *leaf
 5:     {Node,Leaf} *parent := Tree, **ptr
 6:     int pos, val
 7:     Boolean expeditive
 8:     while True do
 9:         ⟨leaf, parent⟩ := Search(data, parent)
10:         if parent = NULL then
11:             ptr := &Tree
12:         else if parent → left = leaf then
13:             ptr := &parent → left
14:         else
15:             ptr := &parent → right
16:         if isHelper = True and leaf → helpersExist ≤ currentUpdate then
17:             int tmp := leaf → helpersExist
18:             if tmp == leaf → helpersExist and tmp ≤ currentUpdate then
19:                 CAS(&leaf → helpersExist, currUpdate, currUpdate + 1)
20:         ⟨pos, expeditive⟩ := Elements.NextIndex(&leaf → helpersExist)
21:         if expeditive = False then
22:             leaf → Announce[t] := ⟨data, ⊥⟩
23:         if pos < M then
24:             if expeditive = False then
25:                 leaf → Announce[t].position := pos
26:             if Elements.GetIndex ≤ M and mode = LOGICAL then
27:                 CAS(&leaf → D[pos].seq, ⊥, MARKED)
28:             leaf → D[pos] := data
29:             if mode == LOGICAL then
30:                 if leaf → D[pos].seq == ⊥ then
31:                     int currSeq := GlobalSeq
32:                     CAS(&leaf → D[pos].seq, ⊥, currSeq)
33:             else if mode == SYSTEM then
34:                 leaf → D[pos].seq := timeStamp[batchID]
35:         else
36:             SplitLeaf(leaf, ptr, expeditive)
37:         if (*ptr) → helpersExist = currentUpdate + 1 then
38:             if expeditive = False and (*ptr) → Announce[t].position ≠ ⊥ then
39:                 (*ptr) → Announce[t] := ⟨⊥, ⊥⟩
40:             else
41:                 continue
       return
```

Algorithm 13 SplitLeaf with batches: a lock-free split operation for a leaf-oriented tree with fat leaves. Code for thread $t \in \{0, \ldots, n-1\}$.

```
1: procedure SplitLeaf(leaf, prt, expeditive, mode)
2:     newNode := new Node initialized with ⟨⊥, NULL, NULL, ⟨⟨⊥, ⊥⟩, ..., ⟨⊥, ⊥⟩⟩, not expeditive⟩
3:     newNode → left := new Leaf initialized with
       ⟨⊥, NULL, NULL, ⟨⟨⊥, ⊥⟩, ..., ⟨⊥, ⊥⟩⟩, False, ⟨⊥, ..., ⊥⟩, ⊥, 0⟩
4:     newNode → right := new Leaf initialized similarly
5:     splitBuffer := ∅
6:     if expeditive = False then
7:         for i ∈ {0, ..., n − 1} with leaf → Announce[i].data ≠ ⊥ do
8:             ldata := leaf → Announce[i].data
9:             if leaf → Announce[i].position ≠ ⊥ then
10:                pos := leaf → Announce[i].position
11:                leaf → D[pos].data := ldata
12:                if mode == LOGICAL then
13:                    if leaf → D[pos].seq == MARKED then
14:                        CAS(&leaf → D[pos].seq, MARKED, ⊥)
15:                    else
16:                        tmstamp := TIMESTAMP
17:                        if leaf → D[pos].seq == ⊥ then
18:                            CAS(&leaf → D[pos].seq, ⊥, tmstamp)
19:                else if mode == SYSTEM then
20:                    tmstamp := timeStamp[batchID]
21:                    leaf → D[pos].seq := tmstamp
22:            else
23:                if mode == LOGICAL then
24:                    Add (ldata, ⊥) to splitBuffer
25:                else
26:                    Add (ldata, timeStamp[batchID]) to splitBuffer
27:            newNode → Announce[i] := ⟨ldata, −1⟩
28:    for each element in leaf → D[i] ∪ splitBuffer do
29:        if element.seq == ⊥ & element ∈ D then
30:            if mode == LOGICAL then
31:                if leaf → D[pos].seq == MARKED then
32:                    CAS(&leaf → D[pos].seq, MARKED, ⊥)
33:                else
34:                    tmstamp := TIMESTAMP
35:                    if leaf → D[pos].seq == ⊥ then
36:                        CAS(&leaf → D[pos].seq, ⊥, tmstamp)
37:            else if mode == SYSTEM then
38:                tmstamp := timeStamp[batchID]
39:                leaf → D[pos].seq := tmstamp
40:        Distribute the element into newNode → left or newNode → right
       based on its key
41:    Fix the key of newNode                    ▷ may result in further leaf splits
42:    CAS(*ptr, leaf, newNode)
43:    if mode == LOGICAL then
44:        Traverse ptr → left and ptr → right
       and for each element with element.seq == ⊥
45:            tmstamp := TIMESTAMP
46:            CAS(&element.pos.seq, ⊥, tmstamp)
```

Algorithm 14 Priority queue of *FreSh*with batches.

```
 1: Code for thread p ∈ {0, ..., n − 1}
 2: Shared variables:
 3: ⟨int, Data⟩ A[0..k − 1][0...n − 1], initially all ⟨⊥, ⊥⟩
 4: CounterObject Cnt, initially 0
 5: Boolean helpersExist, initially False
 6: int insPos, initially 0
 7: ⟨int, Data⟩ *SA, initially NULL
 8: int arrayID, initially p
 9: procedure INSERT(priority, values)
10:     Append values at A[arrayID][p]
11:     arrayID := (arrayID + 1) mod k
12: procedure INITDELETEPHASE
13:     ⟨int, Data⟩ *sa
14:     for each array in A[k] do
15:         if array has valid data then
16:             insPos := insPos + array.size
17:     sa := allocate local array of insPos elements
18:     copy into sa the non-⊥ elements of A and sort them
19:     CAS(&SA, NULL, sa)
20: procedure DELETEMIN(Boolean isHelper, int CurrentUpdate) returns Data
21:     if isHelper = True and helpersExist ≤ CurrentUpdate then
22:         int tmp := helpersExist
23:         if tmp == helpersExist and tmp ≤ CurrentUpdate then
24:             CAS(&helpersExist, CurrentUpdate, CurrentUpdate + 1)
25:     int pos := Cnt.NEXTINDEX(&helpersExist)
26:     if pos ≥ insPos then
27:         return ⊥
28:     if mode == LOGICAL then
29:         if SA[pos].data.seq == MARKED then
30:             return ⊥
31:         if SA[pos].data.seq == ⊥ then
32:             int currSeq := GlobalSeq
33:             CAS(&SA[pos].data.seq, ⊥, currSeq)
34:     return SA[pos].data
```

Chapter 6

Evaluation

6.1 Evaluation of static FreSh

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 [23, 9, 75, 74, 17, 18], and models the distribution of stock market prices [23]. Our real datasets come from the domains of seismology and astronomy. The seismic dataset, *Seismic*, was obtained from the IRIS Seismic Data Access archive [35]. 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 [63]. 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.

Dataset	**Data Series**	**Length (floats)**	**Size (GB)**	**Description**
Seismic	100M	256	100	seismic records
Astro	270M	256	265	astronomical data
Random	100M	256	100	random walks

Table 6.1: Details of datasets used in 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.

6.1.1 Results

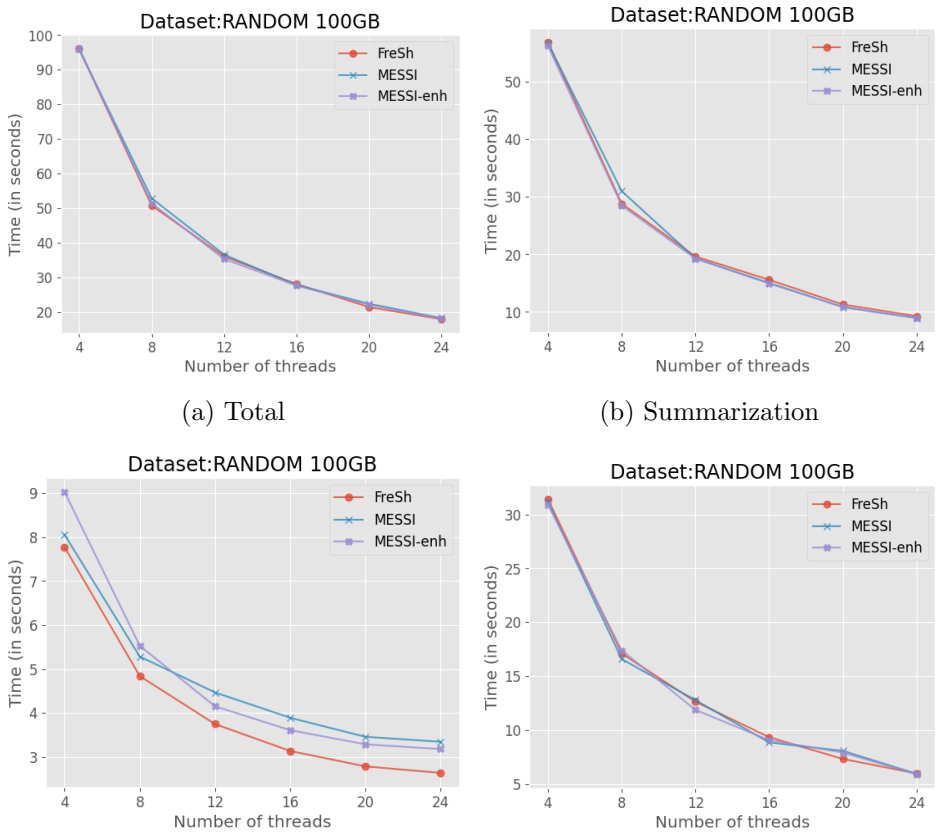
FreSh **vs** *MESSI***.** We compare *FreSh* against *MESSI*, which is the state-of-the-art blocking in-memory data series index.To enable a fair comparison, we use an optimized version of the original *MESSI* implementation, where we have applied all the code enhancements incorporated by *FreSh*. We also compare *FreSh* against an extended version of MESSI, called *MESSIenh*, 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. *MESSIenh* allows to compare the lock-free index creation phase of *FreSh* against a more efficient blocking one than that of original *MESSI*.

Figure 6.1 shows that all algorithms (*FreSh*, *MESSI*, and *MESSIenh*) 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 6.1a) 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 6.1c), 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 *MESSIenh*, in most cases. The results for Seismic are similar and are omitted for brevity. Considering scalability as the size of the dataset increases, Figure 6.2 demonstrates that *FreSh* scales well on all three datasets. In most cases, *FreSh* is faster than *MESSI*. Following previous works [74, 57], 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 6.4a presents the results for the Seismic dataset, where *FreSh* performs better than *MESSI* in most cases.

6.1.2 FreSh vs Baselines.

We compare *FreSh* against several baseline *lock-free* implementations of the different stages of an iSAX-based index. Our results (Figure 6.5a) shows that *FreSh* performs better than all these implementations.

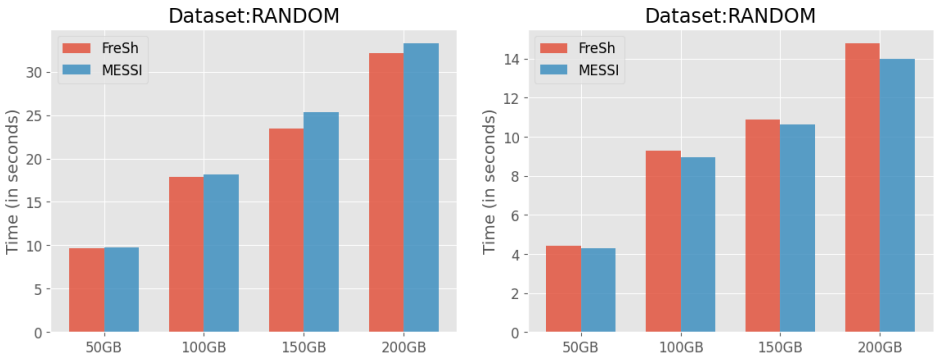
Summarization Baseline: For buffer creation, we have experimented with three implementations: DoAll-Split, FI-Based, and CAS-Based. All use a single summarization buffer with as many elements as *RawData*. DoAll-Split splits *RawData* into



(a) Total (b) Summarization

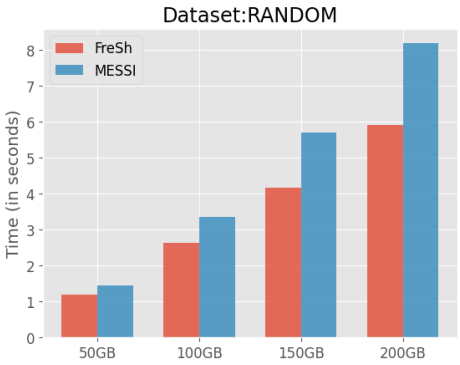
(c) Tree index (d) Query answering

Figure 6.1: Comparison of *FreSh* against *MESSI* and *MESSIenh* on 100GB Random.

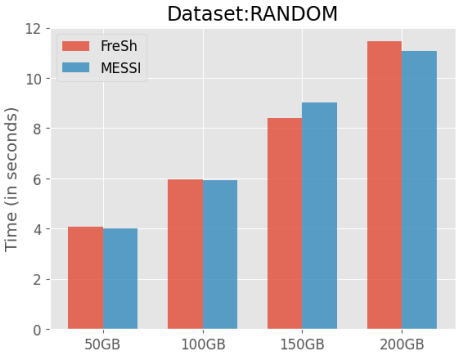


(a) Total

(b) Summarization

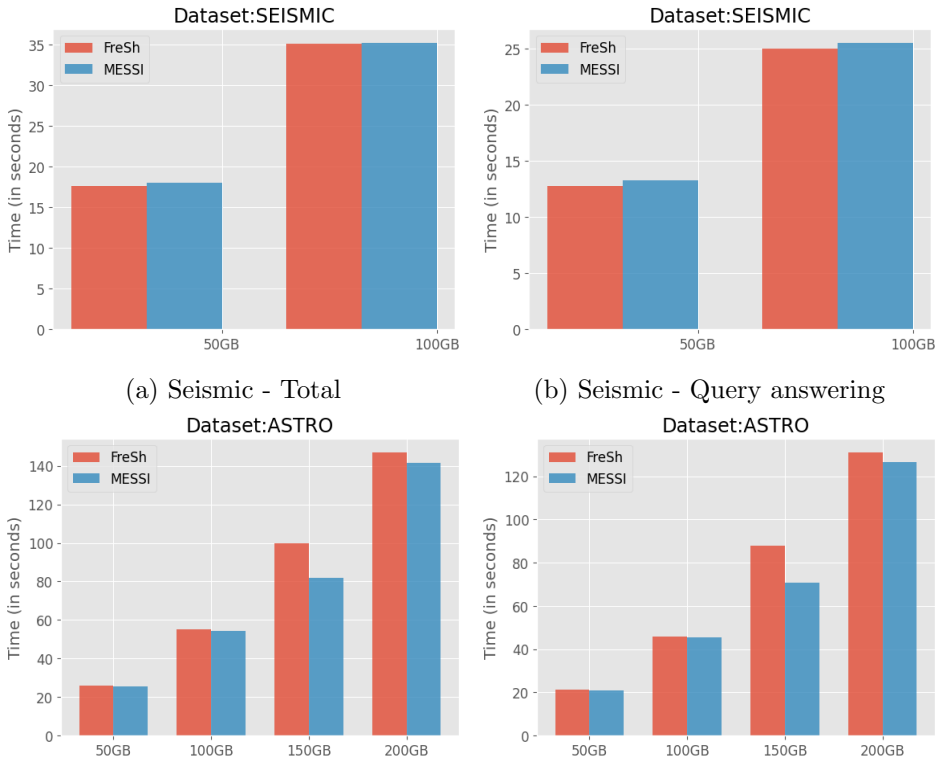


(c) Tree index



(d) Query answering

Figure 6.2: Comparison of *FreSh* against *MESSI*on the Random dataset for 24 threads.



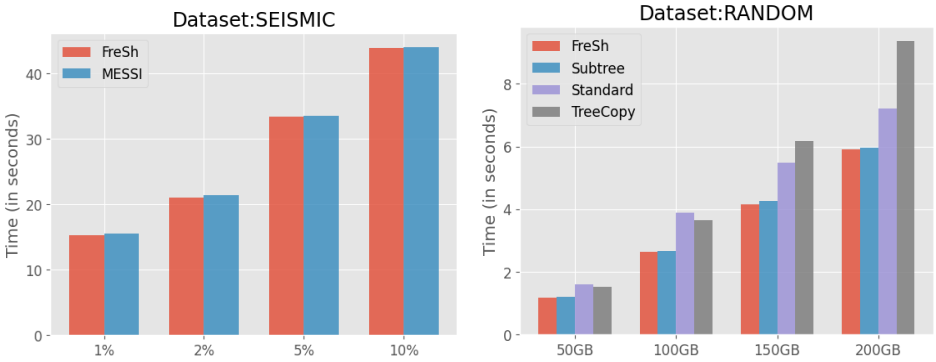
(a) Seismic - Total

(b) Seismic - Query answering

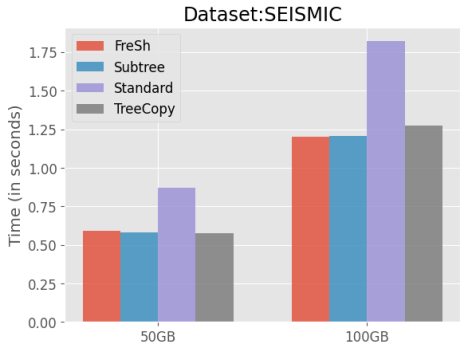
(c) Astro - Total

(d) Astro - Query answering

Figure 6.3: Comparison of *FreSh* against *MESSI* on (a) Seismic and (b) Astro datasets for 24 threads.



(a) Seismic - Total (Query Difficulty) (b) Tree Index - Random



(c) Tree Index - Seismic

Figure 6.4: (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.

as many equaly-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, threads use *FAI* to get assigned data series from *RawData* to process. Each thread t repeatedly performs the following: It executes a *FAI* on O to get a position v of RawData, process the data series stored in this position, and afterwards sets its done flag to `True`. To achieve lock-freedom, FI-Based stores in RawData, a boolean flag (initally false) with each data 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, (i.e., *FAI* returns a value higher than the number of elements of RawData), 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. 6.5a).

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 [22], in addition to our new tree implementation. 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 to an info record in the flagged node. Other threads may use this info record to help the insert complete.

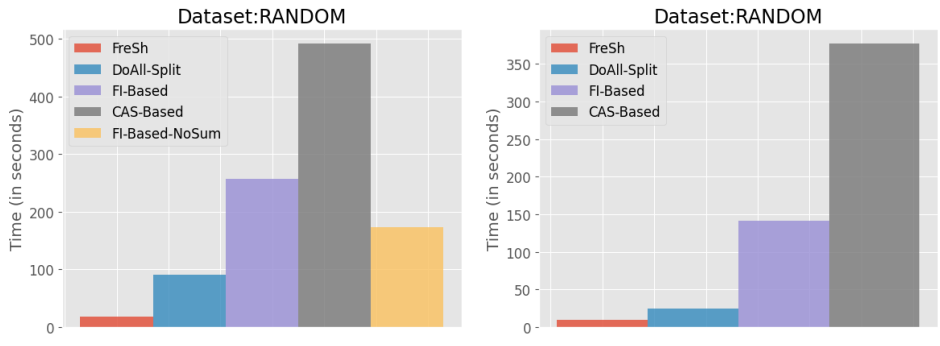
We have also experimented with FI-Based-NoSum, a lock-free implementation that avoids using the summarization buffers and inserts directly iSAX summaries in the index tree, by applying the FI-Based technique on *RawData*. *FreSh* performs significantly better than all these implementations (Figure 6.5c).

Pruning Baseline: All baselines use a single instance of an existing skip-based lock-free priority queue [49] 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. 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 to remove elements from the priority 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

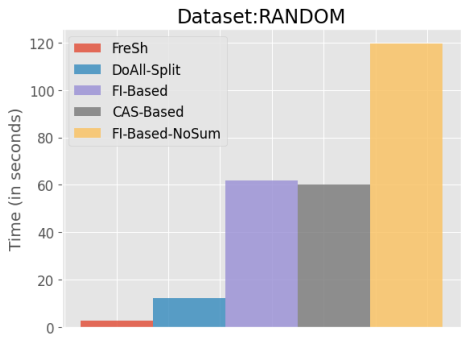
that even the non lock-free simple technique above is quite costly in comparison to our approach.

Our results show that *FreSh* performs significantly better than all these implementations, for query answering time (that includes pruning and refinement, Figure 6.5d),.

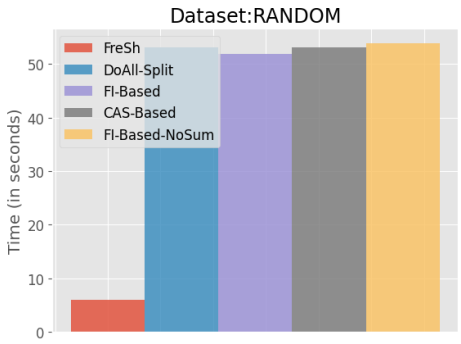


(a) Total

(b) Summarization



(c) Tree index



(d) Query answering

Figure 6.5: Comparison of *FreSh* against baseline implementations on 100GB Random.

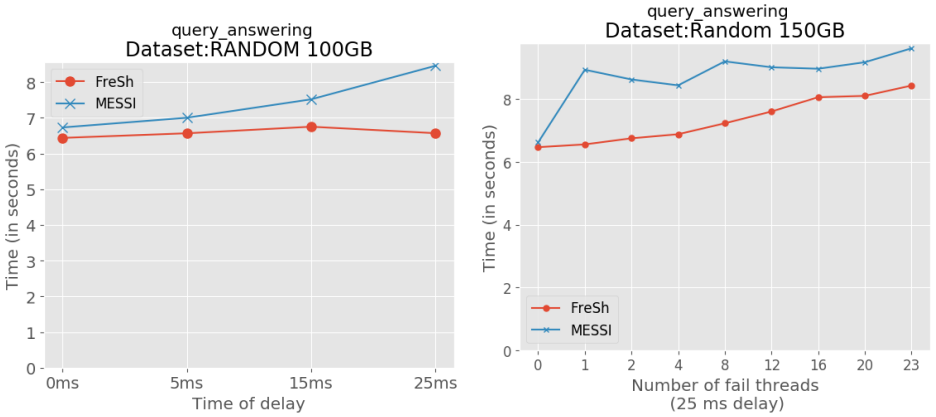
We evaluate the techniques incorporated by *FreSh* to create its tree index by comparing it against three modified versions of it. Recall that in *FreSh* each thread populates each of the subtrees it acquires in expeditive mode, as long as no helper reaches the same leaf of the tree; when this happens 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 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.

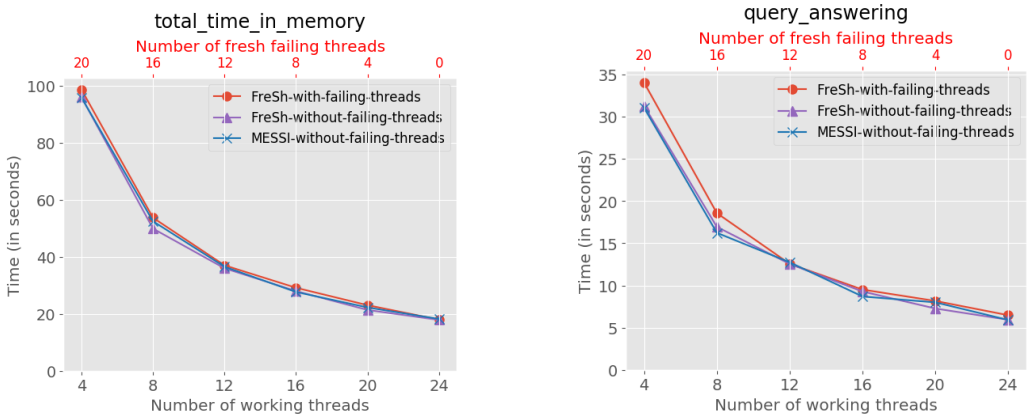
Figures 6.4b-6.4c compare *FreSh* against the modified versions on Random and Seismic with variable dataset sizes and shows that it performs better than them, in all cases. Interestingly, for Seismic 50GB *FreSh* performs similarly to TreeCopy. Recall that each thread works on its own private copy and, 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. We do so by forcing threads to sleep for a specific amount of time, called *delay*. Since *MESSI* (and *MESSIenh*) are lock-based, if a thread crashes, the algorithm will never terminate. Thus, we do not perform experiments for this case. This is due to the barriers that are utilized in their implementation. These barriers are necessary for producing correct outcomes and their ignorance even by one thread could compromise correctness. Figure 6.6a 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*. This is so because in *FreSh*, helper threads undertake the work to be done by the failed thread, efficiently compensating the overhead from the failed thread delay. Moreover, Figure 6.6b 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. When all-but-one threads fail, *FreSh* is still faster than *MESSI* because the single non-failing thread helps the others. 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 6.7), *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.



(a) A single thread is delayed. (b) Multiple threads are delayed.

Figure 6.6: Comparison of *FreSh* against *MESSI* when varying delay and number of delayed threads.



(a) Total (b) Query answering

Figure 6.7: 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).

6.2 Evaluation of Dynamic FreSh

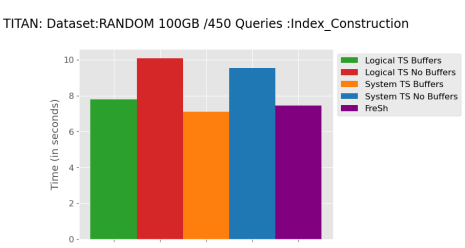
Setup. To evaluate DFreSh we used a different machine equipped with 2 Intel(R) Xeon(R) Gold 5318Y CPU @ 2.10GHz CPUs with 24 cores each, and 36MB L3 cache. The machine runs Ubuntu Linux Ubuntu 24.04.1 LTS and has 256GB of RAM. Code is written in C and compiled using gcc (Ubuntu 13.2.0-23ubuntu4) 13.2.0 with O2 optimizations. **Datasets.** The datasets remain the same. **Evaluation Measures.** We measure (i) the *index construction time* , (ii) the *query answering time* required to answer 50 queries per batch that are not part of the dataset, (iii) we measure the latency of query answering and (iv) we provide a comprehend analysis that shows the performace of DFreSh with different settings, eg different delays between update batches, (different size of batches). Experiments are repeated 5 times and averages are reported. All the algorithms return exact results.

Dynamic FreSh vs FreSh. We compare the dynamic version of *FreSh*, referred to as DFreSh, with *FreSh*, the current state-of-the-art lock-free in-memory data series index. Our implementation includes two versions of DFreSh, each employing a different timestamping algorithm: Logical TS and System TS. Additionally, we implemented variations of both that directly insert data into the iSAX tree, bypassing the summarization buffers used in *FreSh* during its first phase. Unlike DFreSh, which supports insertions of batches dynamically, *FreSh* is a static system designed to process only a single preloaded dataset. To simulate incoming updates in *FreSh*, the entire dataset must be reloaded with each new update batch appended, requiring multiple runs. To ensure a fair comparison, we measure only the time required to incorporate the new update batch, isolating the update performance of both systems.

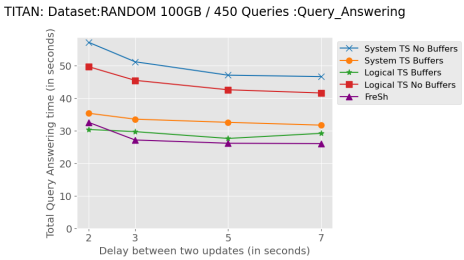
6.2.1 Evaluation using Random Datatasets

All experiments performed begin by preloading the index with 10GB of data, called *initial data* ,followed by processing consecutive batches of XGB until whole dataset is processed. For this experimental analysis we have also decided that each batch is accompanied by 50 new queries. Preloading ensures that even the first queries can find answers. We have also desided to add a delay between consecutive batches. This delay start when a batch arrives and indicates when the next batch is available for processing.

Figure 6.8 illustrates the overall performance of all algorithms when building a 100GB index using random data. Specifically, figure 6.8a compares the actual index construction times of all algorithms, including *FreSh*. As expected, the No Buffers version perform worse due to reduced cache locality caused by bypassing summarization buffers. Despite the overhead of managing dynamic updates (e.g such as handling timestamps), the buffered versions (Logical and System TS) achieve performance comparable to *FreSh*. Figure 6.8b shows the actual query answering time for all algorithms. The buffered versions ouperform No Buffers

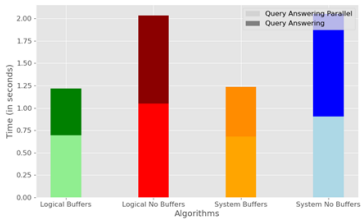


(a) Real Index Construction Time

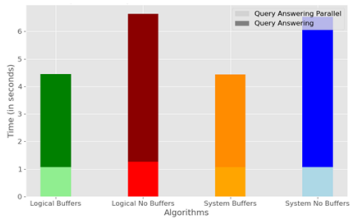


(b) Real Query Answering Time

Figure 6.8: Dynamic FreSh Benchmarks



(a) Minimum Delay

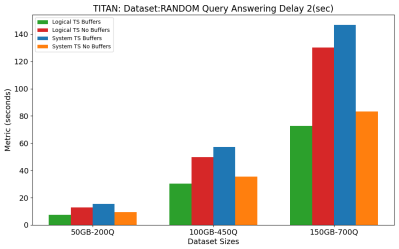


(b) Maximum Delay

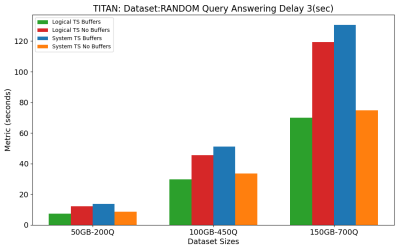
Figure 6.9: Minimum - Longest Delay

versions because the same index worker operates on a subtree, meaning the cache lines of that worker contatin data drom the same subtree leading to improved cache locality. In addition to that *FreSh* avoids concurrent reads and writes, reducing cache misses.

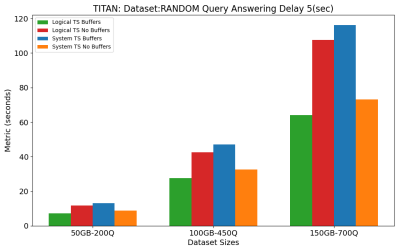
The minimum delay, set at 2 seconds (Figure 6.9a), corresponds to the time required by the fastest algorithm to answer 50 queries on the smallest index size (during the insertion of the first batch). The maximum delay, set at 7 seconds (Figure 6.9b), allows the slowest algorithm to complete 50 queries on the largest index size (last batch). In figure 6.11 the bars represent query answering times, divided into two segments: the light segment indicates concurrent query answering during index construction, while the dark segment represents solo query answering. A dark segment indicates insufficient delay, causing some queries to run during the insertion of other batches. Query answering performance is heavily influenced by the delay. Shorter delays result in poorer performance, as query workers have less time to process the same number of queries while index construction is ongoing. This results in more pending queries, increasing computational effort and response times.



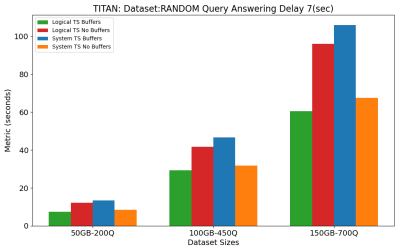
(a) Query Answering Time Delay 2 sec



(b) Query Answering Time Delay 3 sec

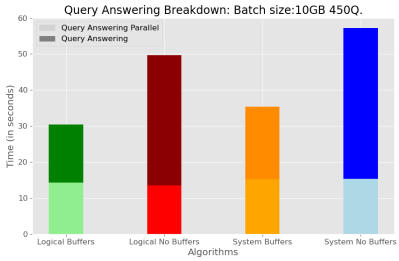


(c) Query Answering Time Delay 5 sec

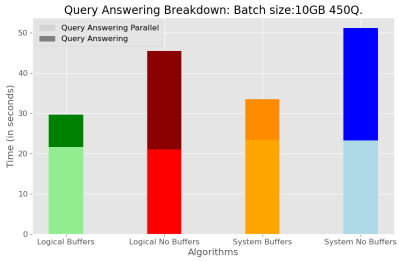


(d) Query Answering Time Delay 7 sec

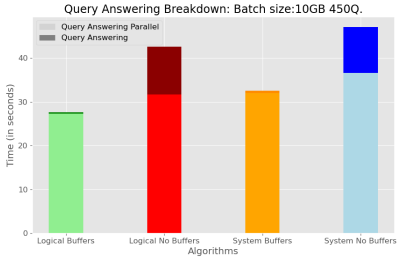
Figure 6.10: Query Answering Time vs. Index Size



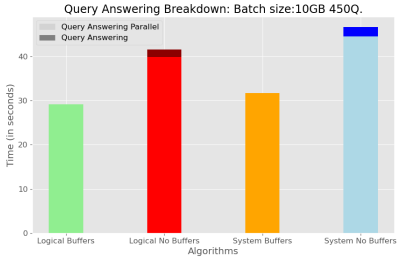
(a) Delay 2 sec



(b) Delay 3 sec

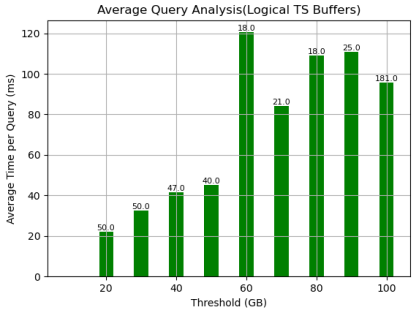


(c) Delay 5 sec

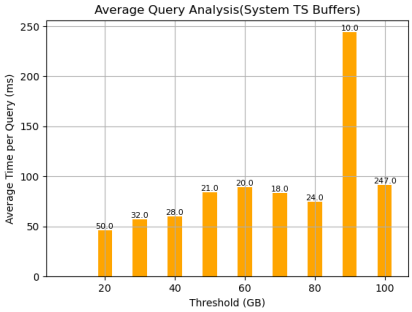


(d) Delay 7 sec

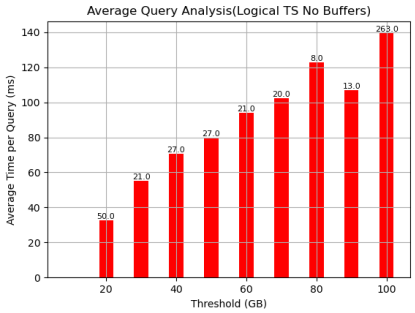
Figure 6.11: Query Answering Breakdown



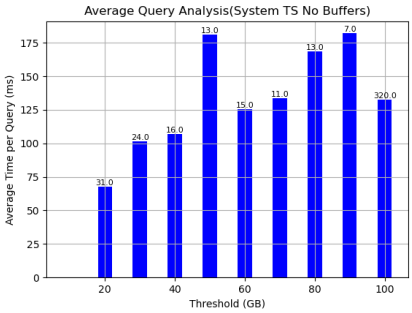
(a) Logical TS buffers



(b) System TS buffers



(c) Logical TS No buffers

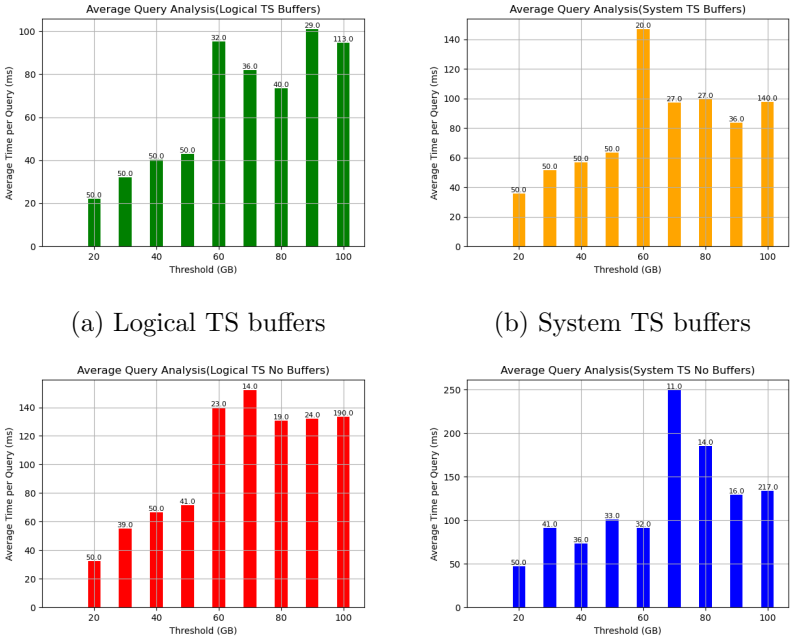


(d) System TS No buffers

Figure 6.12: Query progress with minimum delay 2 sec.

6.2.1.1 Query Progress

Figures 6.12-6.15 illustrate the progress of DFreSh while batches are being inserted. These graphs provide an overview of the number of queries answered as the index grows, with the x-axis representing the index size. For example, in Figure 6.12a, the first bar shows that 50 queries were answered between the initial index state and the insertion of the first batch. This indicates that the implementation of DFreSh using logical timestamps processes all queries associated with that batch before the next insertion. In contrast, Figure 6.12d shows that the implementation without summarization buffers and using system timestamps answers only 31 out of 50 queries in the same timeframe. These graphs also provide insights into the overall query answering performance, as summarized in Figure 6.8b. Notably, as the delay increases, even the slowest implementations,those without summarization buffers, reduce the number of queries carried over to subsequent batches. This suggests that with sufficient delay, even these slower implementations can complete all queries within the available time.



(a) Logical TS buffers (b) System TS buffers

(c) Logical TS No buffers (d) System TS No buffers

Figure 6.13: Query progress with Intermediate delay 3 sec.

6.2.1.2 Latency

Figure 6.16 presents the average latency for each query.

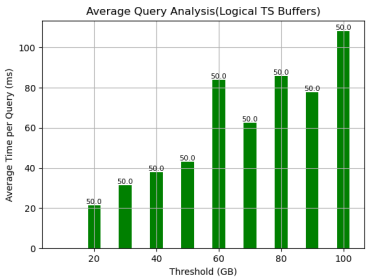
Definition 5 (Query Latency)**.** *Query latency is defined as the time between a query's arrival and the start of its processing.*

As mentioned earlier, queries arrive in batches along with the update batch, meaning that all queries in a batch share the same arrival time. The latency of a query is defined as the time required to process all pending queries that arrived before it. When the delay is long enough to process all queries in a batch, the latency of the i-th query can be calculated as the sum of the times required to answer all previous queries in the same batch:

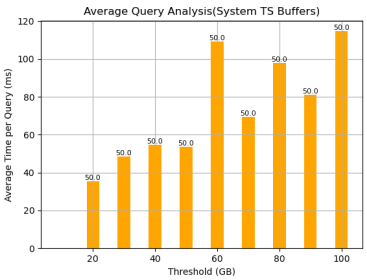
$$\text{Latency}(q[i]) = \sum_{j=0}^{i-1} \text{qa}[j],$$

where qa$[j]$ represents the time needed to answer the j-th query.

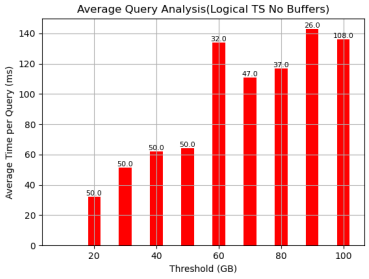
However, if the delay is not sufficient to process all queries within a batch, the latency of the i-th query is calculated differently. It includes the time required to answer the pending queries from previous updates (queries from previous batches)



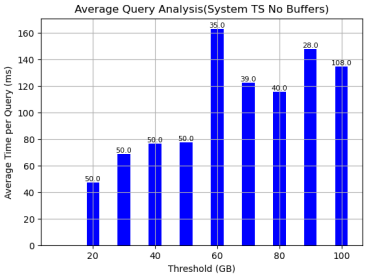
(a) Progress of Queries Logical TS buffers



(b) Progress of Queries System TS buffers

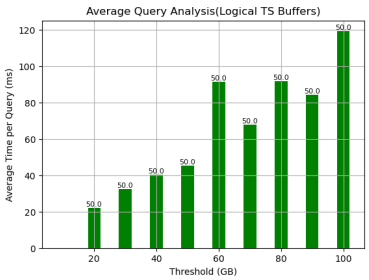


(c) Progress of Queries Logical TS No buffers

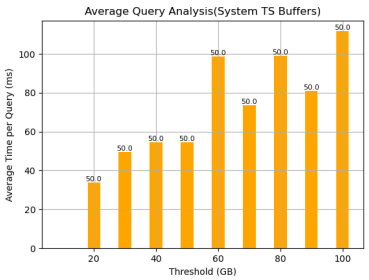


(d) Progress of Queries System TS No buffers

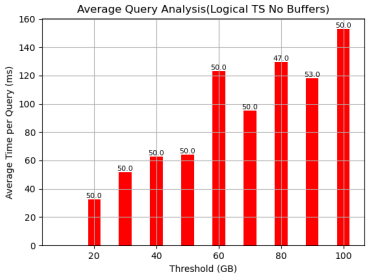
Figure 6.14: Query progress with Intermediate delay 5 sec.



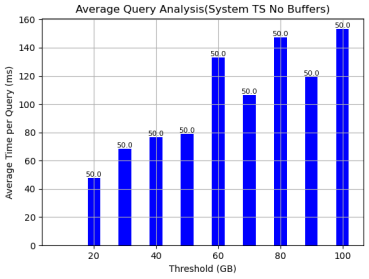
(a) Progress of Queries Logical TS buffers



(b) Progress of Queries System TS buffers



(c) Progress of Queries Logical TS No buffers



(d) Progress of Queries System TS No buffers

Figure 6.15: Query progress with largest delay, 7 sec.

and the time for all the queries that arrived before the i-th query in the current batch:

$$\text{Latency}(q[i]) = \sum_{j=k}^{n-1} \text{qa}[j] + \sum_{j=0}^{i-1} \text{qa}[j],$$

where the first sum accounts for the pending queries from previous updates and the second sum accounts for the queries from the current batch up to the i-th query. For *FreSh* calculating latency is more complicated because it is a static index, meaning query answering begins only after the index construction is complete. The time available for answering queries in a given update batch can be expressed as:

$$\text{AvailableQA}(qa[i]) = \text{DELAY} - \text{IC}[i],$$

where AvailableQA($qa[i]$) represents the available time for answering queries in the i-th update batch, DELAY is the delay interval, and IC[i] is the time required to insert the i-th batch into the index.

If the delay is sufficient to answer all the queries in time, the latency can be calculated as:

$$\text{Latency}(q[i]) = \text{IC}[i] + \sum_{j=0}^{i-1} \text{qa}[j],$$

where IC[i] represents the time to append the i-th batch to the index, and qa[j] is the time required to answer the j-th query. If the delay is insufficient, unanswered queries from the current batch remain pending and carry over to the next query answering period. These pending queries will be processed during the AvailableQA($qa[i+1]$) interval of the subsequent update batch. Thus, the latency can be calculated as:

$$\text{Latency}(q[i]) = \sum_{k=n}^{m} \text{IC}[k] + \sum_{j=0}^{i-1} \text{qa}[j],$$

where the n-th batch contains the queries in question, and the m-th batch represents when query i is being answered. If the current batch also includes pending queries from previous batches, the latency must also account for the time to answer those queries. Thus, the general formula for calculating latency in *FreSh* is:

$$\text{Latency}(q[i]) = \sum_{k=n}^{m} \text{IC}[k] + \sum_{j=p}^{pq} \text{qa}[j] + \sum_{j=0}^{i-1} \text{qa}[j],$$

where n represents the batch that contains the query i,-th m represents the batch the the i-th query is being answered, and queries p to pq represents the pending queries from previous baches.

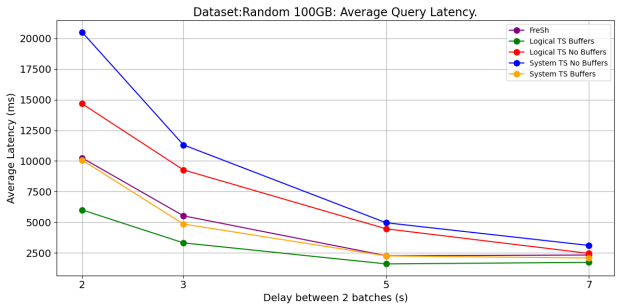


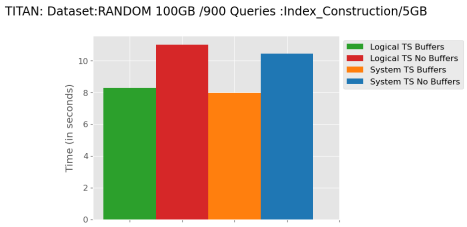
Figure 6.16: Query Latency

In Figure 6.16, System TS Buffers and FreSh exhibit similar latency. The key difference between these two implementations, aside from the fact that System TS Buffers support dynamic updates, is that System TS Buffers allow concurrent query execution with timestamps earlier than the timestamp of the inserting batch. In our implementation, each query receives an incrementing timestamp, meaning that the timestamp of query $i+1$ will be larger than that of query i. As a result, only a small portion of the queries will be fast enough to be answered before the update batch acquires a timestamp. After this point, queries must wait for the index to be completed before they can proceed. In the worst-case scenario for System TS Buffers, queries are never fast enough to be answered before the new batch acquires a timestamp, causing it to behave similarly to FreSh.

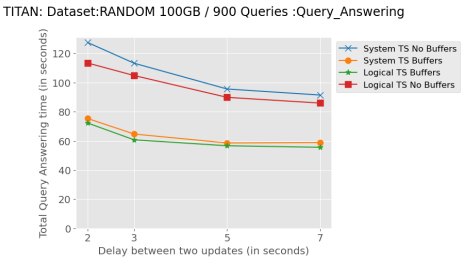
6.2.1.3 DFreSh with different batch sizes

We have also conducted experiments using different batch sizes. In our analysis, we focus on batch sizes of 5GB, 10GB, and 20GB. Each batch corresponds to answering 50 queries. Consequently, for 5GB batches, we process a total of 900 queries over a 100GB dataset, whereas for 20GB batches, we process 250 queries over a 110GB dataset, considering that the initial index size remains 10GB throughout our experiments.

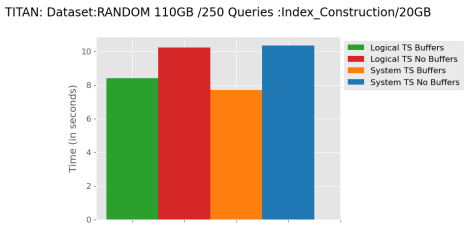
Figure 6.17 presents the overall performance of DFreSh across different batch sizes. The results indicate that the behavior remains consistent, as batch size does not significantly impact either the index construction or query answering performance. In particular, the average query answering time shows only minimal variation across different batch sizes.



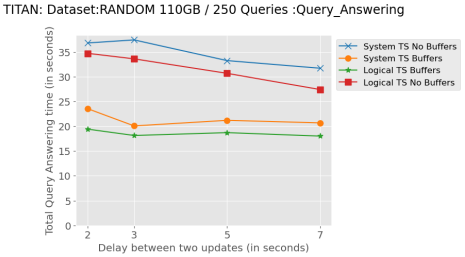
(a) Real Index Construction Time 5GB Batches



(b) Real Query Answering Time 5GB Batches

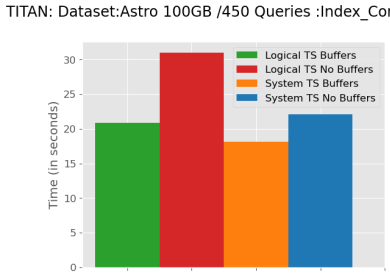


(c) Real Index Construction Time 20GB Batches

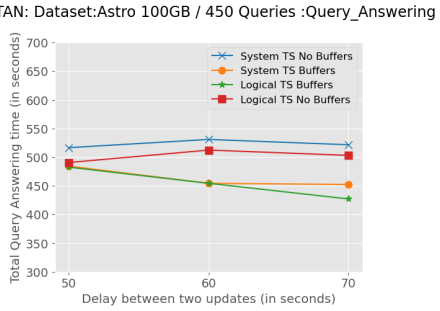


(d) Real Query Answering Time 20GB Batches

Figure 6.17: Dynamic FreSh Benchmarks with different batches



(a) Astro: Real Index Construction Time



(b) Astro: Real Query Answering Time

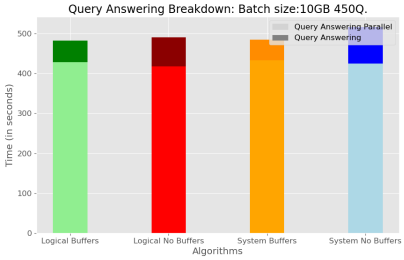
Figure 6.18: Astro: DFreSh Overall Performance

6.2.2 Evaluation using Real Datatasets

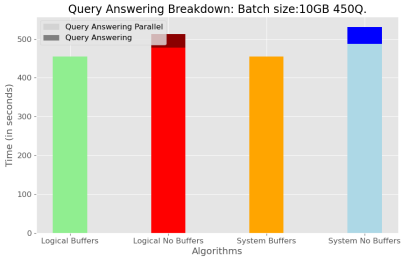
This section focuses on experiments conducted using real datasets. Our experimental analysis of DFreSh requires executing 50 queries per inserted batch. However, the original dataset contained only 100 queries, which were insufficient for our experiments. To generate additional queries, we randomly selected series from the dataset, added Gaussian noise ($\mu = 0$, $sigma = 0.1$) to each point, and used these modified series as queries. Answering queries on real datasets typically requires more time, as observed in the evaluation of FreSh and previous works, since real queries tend to be more challenging. The added noise further increases query difficulty, making the evaluation even more demanding.As a result, the delays used for random datasets cannot be applied to real datasets. Given that query answering is more computationally intensive than index construction in this setting, we adjusted the distribution of worker threads between these tasks. Instead of assigning 36 workers to index construction and 12 to query answering, as in previous experiments, we reversed the allocation, dedicating 36 workers to query answering and 12 to index construction. This ensures that more computational resources are available for the more demanding phase of the process.

Astro Dataset The initial index size is set to 10GB, and we use only the first 100GB of the dataset. This decision is based on two factors: (1) previous experiments have shown that dataset size does not affect the average performance of DFreSh, and (2) the experiment is computationally expensive and time-consuming.

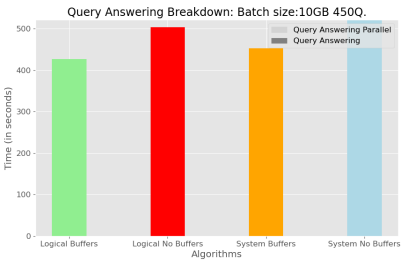
Figure 6.18 presents the overall performance of DFreSh. Specifically, Figure 6.18a shows the index construction time, where, as expected, versions utilizing summarization buffers achieve better performance. We observe that *System Timestamps* outperform *Logical Timestamps*. This is expected, as system timestamps eliminate the need for CAS instructions during timestamp assignment and reduce concurrency in the iSAX tree between workers handling different phases due to the waiting protocol we described, which in turn leads to fewer cache misses. Figure 6.18b illustrates the total query answering time for processing 450 queries. As expected, summarization buffers improve performance. Additionally, as the



(a) Astro: Query Breakdown 50 Delay



(b) Astro: Astro: Query Breakdown 60 Delay



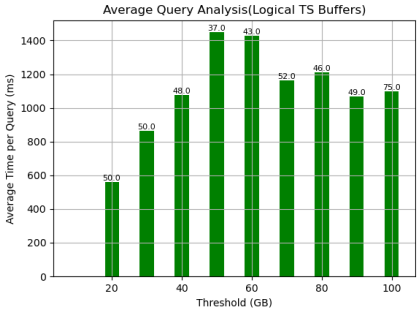
(c) Astro: Astro: Query Breakdown 70 Delay

Figure 6.19: Astro: Query Performance Breakdown

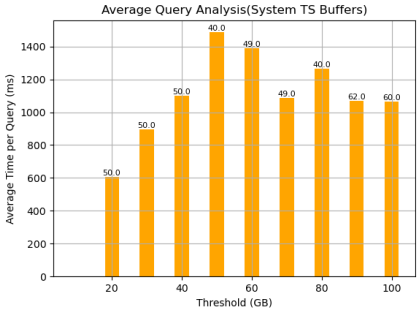
delay increases, different versions tend to perform better because more queries are processed within each batch. Consequently, fewer queries are carried over to subsequent batches, reducing the number of queries that must be answered on an increasingly larger index.

Query Breakdown Figure 6.19 illustrates the distribution of query answering time in DFreSh. It shows how much of the query processing occurs during the concurrent execution phase, including the delay times, and how much takes place after index construction is complete. This distinction is represented by the light and dark segments of the bars, respectively. Additionally, this figure provides insight into the difficulty of query answering. The required delay is 10 times larger than that in the Random dataset.

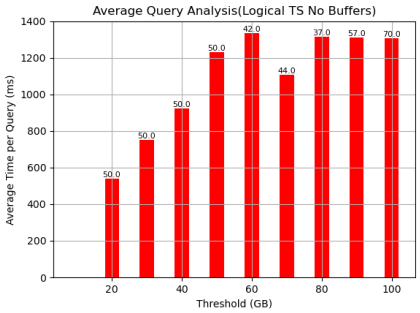
Query Progress Figures 6.20 to 6.22 show the average query answering time as the index size increases. The details of what each bar represents are explained in the same experiment conducted on the random dataset. A few observations can be made from these graphs. As expected, as the delay increases, all versions of DFreSh continue to answer queries in a timely manner. However, in Figures 6.20a to 6.20b, a different pattern emerges between the *Logical* and *System* timestamps. Although *System* timestamps appear fast enough to answer queries promptly in the early stages, they end up perform worse than *Logical*. From these graphs, we observe that when the index is around half its full capacity, the average query answering



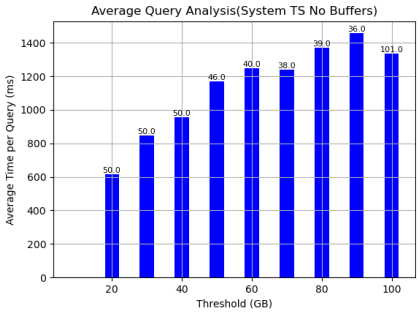
(a) Astro: Logical TS Buffers



(b) Astro: System TS Buffers



(c) Astro: Logical TS No Buffers



(d) Astro: System TS No Buffers

Figure 6.20: Astro: Query Progress 50 delay

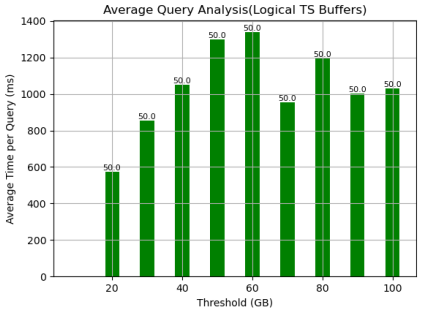
time reaches its peak. This is likely because the queries being answered during this period are the most difficult or the queries cause contention and thus cache misses. Since *Logical* timestamps support more concurrency between workers, they face greater contention, resulting in fewer queries being answered in that phase. For instance, between the 40-60GB range, the *System* version answers 89 queries in total, while the *Logical* version answers only 80. However, in the next range (70-80GB), *Logical* timestamps manage to close the 9-query gap, answering queries in a smaller average time.

Latency Figure 6.23 shows the average query latency for the Astro dataset. As expected, the latency increases as query answering becomes more resource-intensive. The latency is calculated as described in the previous sections, and the observed pattern aligns with expectations.

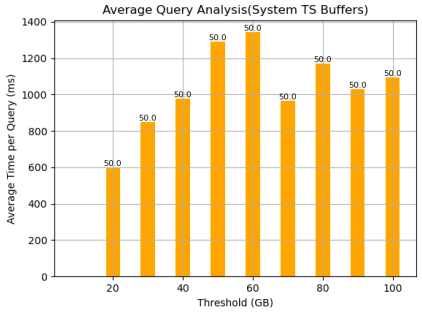
Seismic Dataset

The initial index size is set to 10GB, and the total size of the dataset is limited to 90GB.

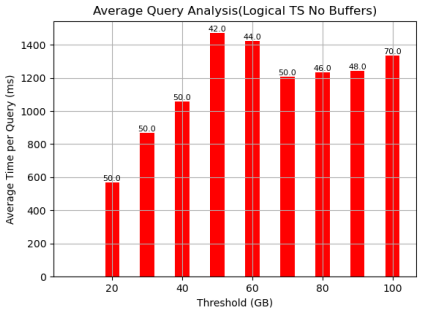
Figure 6.24 presents the overall performance of DFreSh on the Seismic dataset. As with the Astro dataset, Figure 6.24a shows that versions utilizing summarization buffers achieve better index construction performance. Similarly, Figure 6.24b



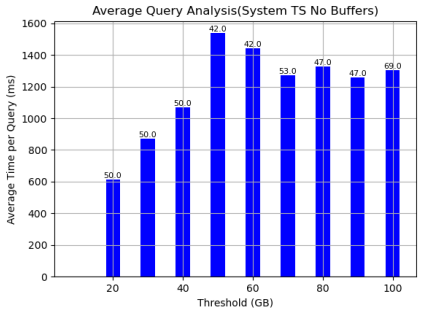
(a) Astro: Logical TS Buffers



(b) Astro: System TS Buffers

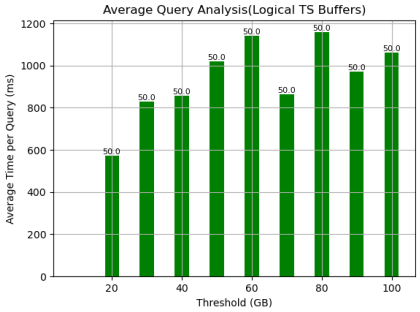


(c) Astro: Logical TS No Buffers

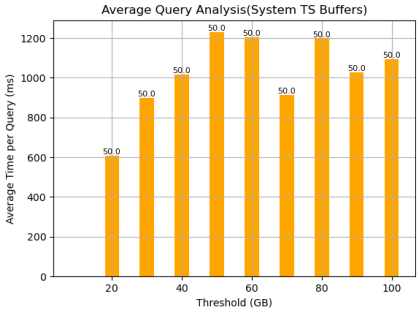


(d) Astro: System TS No Buffers

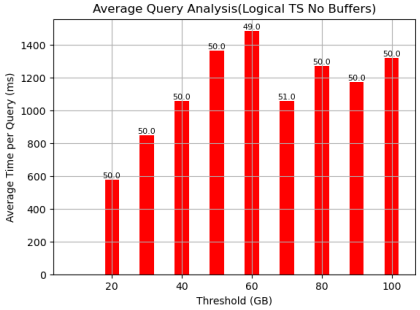
Figure 6.21: Astro: Query Progress 60 delay



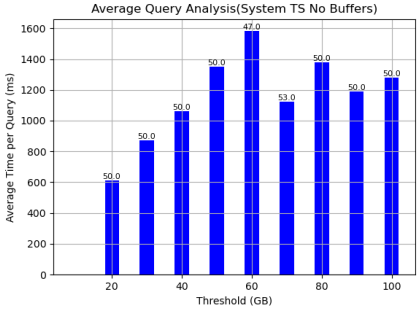
(a) Astro: Logical TS Buffers



(b) Astro: System TS Buffers



(c) Astro: Logical TS No Buffers



(d) Astro: System TS No Buffers

Figure 6.22: Astro: Query Progress 70 delay

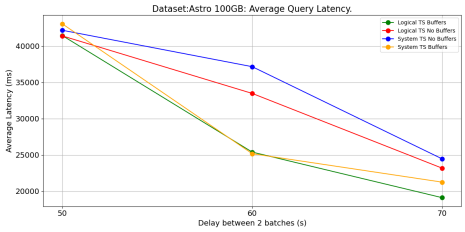
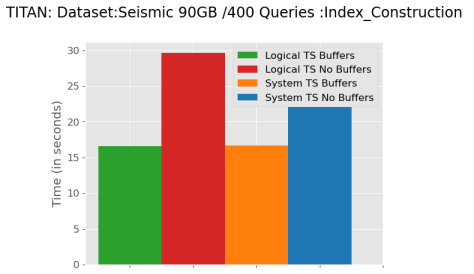
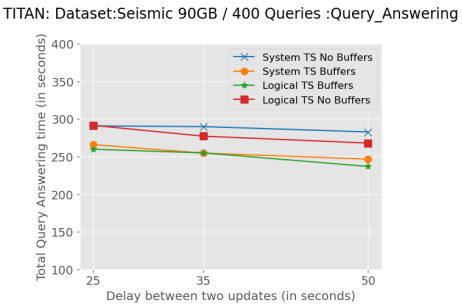


Figure 6.23: Astro: Query Latency



(a) Seismic: Real Index Construction Time



(b) Seismic: Real Query Answering Time

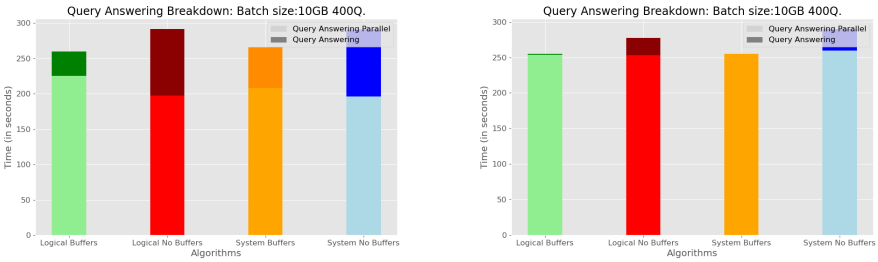
Figure 6.24: Seismic: DFreSh Overall Performance

illustrates the total query answering time for processing 400 queries, where summarization buffers again lead to improved efficiency. Based on our analysis, the chosen delays for this dataset are set to 25, 35, and 50 seconds. The observations made for the *Astro* dataset also apply here, highlighting consistent trends across different real-world datasets.

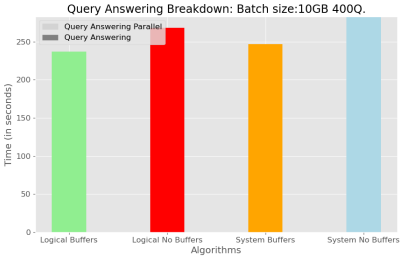
Query Breakdown Figure 6.25 illustrates the distribution of query answering time in DFreSh for the Seismic dataset. As in the Astro dataset, it distinguishes between queries processed during the concurrent execution phase, including delay times, and those answered after index construction is complete, represented by the light and dark segments of the bars, respectively. Once again, we observe the increased difficulty of query answering in real datasets compared to synthetic ones, with the required delay being approximately seven times higher.

Query Progress Figures 6.26 to 6.28 illustrate the average query answering time as the index size grows. The meaning of each bar remains the same as in the corresponding experiment on the random dataset. As expected, increasing the delay allows all versions of DFreSh to maintain timely query answering. Unlike previous cases, this dataset exhibits no unexpected behavior, with patterns aligning well with our observations from earlier experiments.

Latency Figure 6.29 presents the average query latency for the Seismic dataset. As expected, its latency follows a similar trend to that of the Astro dataset, given that both are real datasets. The calculation method remains the same, as described in previous sections, and the observed pattern is consistent with our expectations.

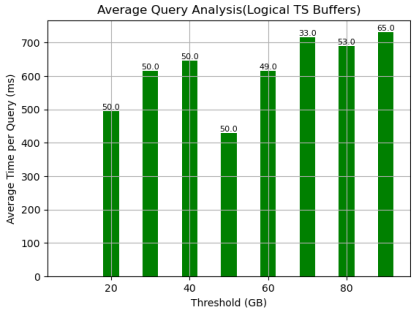


(a) Seismic: Query Breakdown 25 Delay (b) Seismic: Query Breakdown 35 Delay

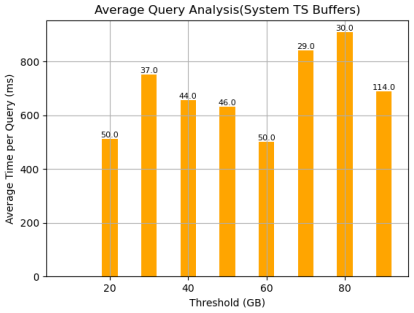


(c) Seismic: Query Breakdown 50 Delay

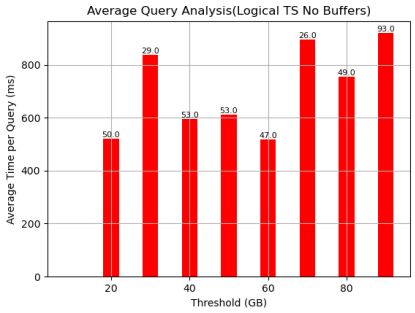
Figure 6.25: Seismic: Query Performance Breakdown



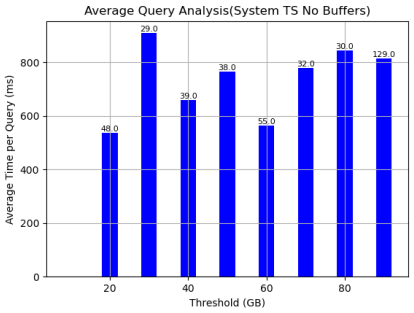
(a) Seismic: Logical TS Buffers



(b) Seismic: System TS Buffers

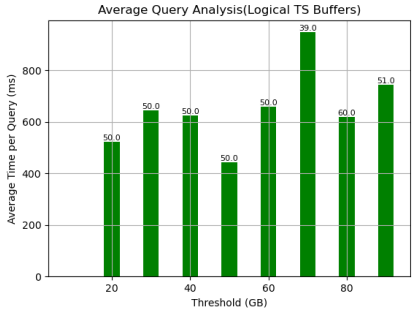


(c) Seismic: Logical TS No Buffers

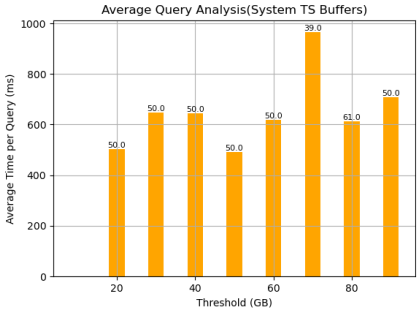


(d) Seismic: System TS No Buffers

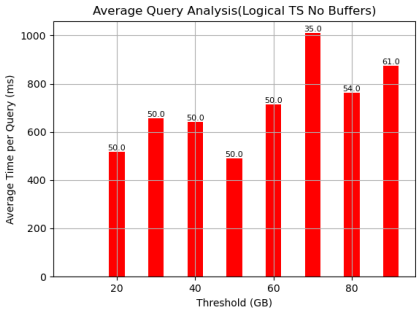
Figure 6.26: Seismic: Query Progress 25 delay



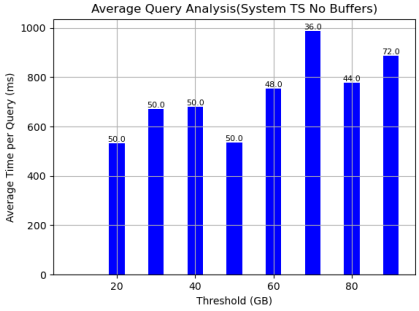
(a) Seismic: Logical TS Buffers



(b) Seismic: System TS Buffers

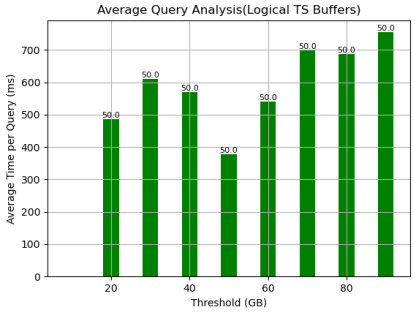


(c) Seismic: Logical TS No Buffers

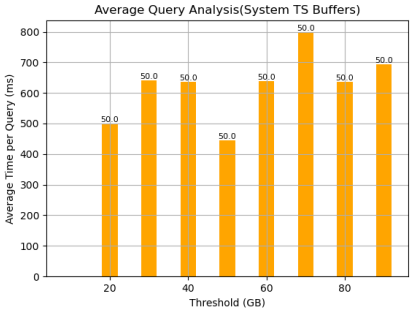


(d) Seismic: System TS No Buffers

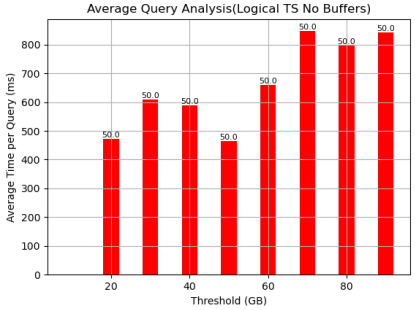
Figure 6.27: Seismic: Query Progress 35 delay



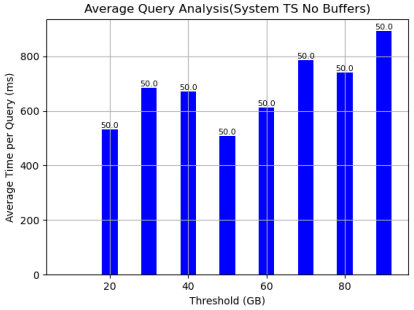
(a) Seismic: Logical TS Buffers



(b) Seismic: System TS Buffers



(c) Seismic: Logical TS No Buffers



(d) Seismic: System TS No Buffers

Figure 6.28: Seismic: Query Progress 50 delay

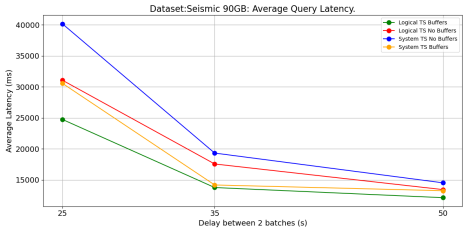


Figure 6.29: Seismic: Query Latency

Bibliography

[1] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir 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] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios 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] Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios 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, 2022.

[4] Ilias Azizi, Karima Echihabi, and Themis Palpanas. Elpis: Graph-based similarity search for scalable data science. *Proc. VLDB Endow.*, 16(6):1548–1559, 2023.

[5] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2002)*, 2002.

[6] Anthony J. Bagnall, Richard L. Cole, Themis Palpanas, and Konstantinos Zoumpatianos. Data series management (dagstuhl seminar 19282). *Dagstuhl Reports*, 9(7), 2019.

[7] Anastasia Braginsky and Erez Petrank. A lock-free B+tree. In *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures*, 2012.

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

[9] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn Keogh. Beyond One Billion Time Series: Indexing and Mining Very Large Time Series Collections with iSAX2+. *KAIS*, 39(1), 2014.

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

[11] Manos Chatzakis, Panagiota Fatourou, Eleftherios Kosmas, Themis Palpanas, and Botao Peng. Odyssey: A Journey in the Land of Distributed Data Series Similarity Search. *PVLDB*, 2023.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Googleb••s globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8:1–8:22, 2013.

[13] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. Hercules against data series similarity search. *Proc. VLDB Endow.*, 15(10):2005–2018, 2022.

[14] Karima Echihabi, Themis Palpanas, and Kostas Zoumpatianos. New trends in high-d vector similarity search: Ai-driven, progressive, and distributed. *Proc. VLDB Endow.*, 14(12):3198–3201, 2021.

[15] Karima Echihabi, Theophanis Tsandilas, Anna Gogolou, Anastasia Bezerianos, and Themis Palpanas. Pros: data series progressive k-nn similarity search and classification with probabilistic quality guarantees. *VLDB J.*, 32(4):763–789, 2023.

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

[17] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art. *PVLDB*, 12(2), 2018.

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

[19] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proc. 33rd ACM Symposium on Principles of Distributed Computing*, pages 332–340, 2014.

[20] Faith Ellen, Panagiota Fatourou, Elefterios Kosmas, Alessia Milani, and Corentin Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. *Distributed Computing*, 29:251–277, 2016.

[21] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing*, pages 131–140, 2010.

[22] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck 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.

[23] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, New York, NY, USA, 1994.

[24] Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue Adaptive Universal Constructions. In *disc09*, pages 127–141, 2009.

[25] Panagiota Fatourou and Nikolaos 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.

[26] Panagiota Fatourou and Nikolaos 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.

[27] Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, 55(3):475–520, October 2014.

[28] Panagiota Fatourou and Nikolaos 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.

[29] Panagiota Fatourou and Nikolaos D. Kallimanis. The RedBlue family of universal constructions. *Distributed Computing*, 33:485–513, 2020.

[30] Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleni 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.

[31] Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleftherios 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.

[32] Panagiota Fatourou, Nikolaos D. Kallimanis, and Thomas 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.

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

[34] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In Pierre Fraigniaud, editor, *Distributed Computing*, pages 78–92, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[35] Incorporated Research Institutions for Seismology with Artificial Intelligence. Seismic Data Access. `http://ds.iris.edu/data/access/`, 2018.

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

[37] Anna Gogolou, Theophanis Tsandilas, Karima Echihabi, Anastasia Bezerianos, and Themis Palpanas. Data series progressive similarity search with probabilistic quality guarantees. In *SIGMOD*, 2020.

[38] Rachid Guerraoui, Michał Kapałka, and Petr Kouznetsov. The weakest failure detectors to boost obstruction-freedom. In Shlomi Dolev, editor, *Distributed Computing*, pages 399–412, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[39] Meng He and Mengdu 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.

[40] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[41] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

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

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

[44] Ramin Izadpanah, Steven Feldman, and Damian 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.

[45] Jaemin Jo, Jinwook Seo, and Jean-Daniel Fekete. PANENE: A progressive algorithm for indexing and querying approximate k-nearest neighbors. *IEEE Trans. Vis. Comput. Graph.*, 26(2):1347–1360, 2020.

[46] Eamonn J. Keogh, Kaushik Chakrabarti, Michael J. Pazzani, and Sharad Mehrotra. Dimensionality reduction for fast similarity search in large time series databases. *Knowl. Inf. Syst.*, 3(3):263–286, 2001.

[47] Oleksandra Levchenko, Boyan Kolev, Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, Themis Palpanas, Dennis E. Shasha, and Patrick Valduriez. Bestneighbor: efficient evaluation of knn queries on large time series databases. *Knowl. Inf. Syst.*, 63(2):349–378, 2021.

[48] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving approximate nearest neighbor search through learned adaptive early termination. In *SIGMOD*, 2020.

[49] Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In Roberto Baldoni, Nicolas Nisse, and Maarten van Steen, editors, *Principles of Distributed Systems*, pages 206–220, Cham, 2013. Springer International Publishing.

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

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

[52] Themis 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.

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

[54] Botao Peng, Panagiota Fatourou, and Themis Palpanas. Paris: The next destination for fast data series indexing and query answering. *IEEE BigData*, 2018.

[55] Botao Peng, Panagiota Fatourou, and Themis Palpanas. Messi: In-memory data series indexing. In *ICDE*, 2020.

[56] Botao Peng, Panagiota Fatourou, and Themis Palpanas. Paris+: Data series indexing on multi-core architectures. *TKDE*, 2020.

[57] Botao Peng, Panagiota Fatourou, and Themis Palpanas. Fast data series indexing for in-memory data. *The VLDB Journal*, 30(6):1041–1067, nov 2021.

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

[59] Botao Peng, Panagiota Fatourou, and Themis Palpanas. SING: Sequence Indexing Using GPUs. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2021.

[60] Thanawin Rakthanmanon, Bilson J. L. Campana, Abdullah Mueen, Gustavo E. A. P. A. Batista, M. Brandon Westover, Qiang Zhu, Jesin Zakaria, and Eamonn J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *SIGKDD*, 2012.

[61] Adones Rukundo and Philippas Tsigas. Tslqueue: An efficient lock-free design for priority queues. In Leonel Sousa, Nuno Roma, and Pedro Tomás, editors, *Euro-Par 2021: Parallel Processing*, pages 385–401, Cham, 2021. Springer International Publishing.

[62] Jin Shieh and Eamonn Keogh. iSAX: Indexing and Mining Terabyte Sized Time Series. In *SIGKDD*, 2008.

[63] S Soldi, Volker Beckmann, WH Baumgartner, Gabriele Ponti, Chris R Shrader, P Lubiński, HA Krimm, F Mattana, and Jack Tueller. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics*, 563, 2014.

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

[65] Orr Tamir, Adam Morrison, and Noam Rinetzky. A Heap-Based Concurrent Priority Queue with Mutable Priorities for Faster Parallel Algorithms. In *OPODIS*, volume 46, pages 1–16, 2016.

[66] Shahar Timnat and Erez 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.

[67] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD 2021)*, pages 2602–2617, 2021.

[68] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. *VLDB*, 2013.

[69] Zeyu Wang, Qitong Wang, Peng Wang, Themis Palpanas, and Wei Wang. Dumpy: A Compact and Adaptive Index for Large Data Series Collections. In *SIGMOD*, 2023.

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

[71] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-lsm relaxed priority queue. *SIGPLAN Not.*, 50(8):277–278, jan 2015.

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

[73] Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. Massively distributed time series indexing and querying. *TKDE*, 32(1), 2020.

[74] Kostas Zoumpatianos, Yin Lou, Ioana Ileana, Themis Palpanas, and Johannes Gehrke. Generating data series query workloads. *VLDB J.*, 2018.

[75] Kostas Zoumpatianos, Yin Lou, Themis Palpanas, and Johannes 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.