Université Paris Cité

Odyssey: A Journey in the Land of Distributed Data Series Similarity Search

Manos Chatzakis, Panagiota Fatourou, Eleftherios Kosmas, Themis Palpanas, Botao Peng

LIPADE-TR-Nº 10

October 15, 2022

Technical Report

LIPADE
Laboratoire d'Informatique PAris DEscartes

Odyssey: A Journey in the Land of Distributed Data Series Similarity Search

Manos Chatzakis
FORTH-ICS &
University of Crete
chatzakis@ics.forth.gr

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

Eleftherios Kosmas
University of Crete & FORTH ICS
ekosmas@csd.uoc.gr

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

Botao Peng
Institute of Computing Technology,
Chinese Academy of Sciences
pengbotao@ict.ac.cn

Abstract

This paper presents Odyssey, a novel *distributed* data-series processing framework that efficiently addresses the critical challenges of exhibiting good speedup and ensuring high scalability in data series processing by taking advantage of the full computational capacity of modern clusters comprised of multi-core servers. Odyssey addresses a number of challenges in designing efficient and highly-scalable *distributed* data series index, including efficient scheduling, and load-balancing without paying the prohibitive cost of moving data around. It also supports a flexible partial replication scheme, which enables Odyssey to navigate through a fundamental trade-off between data scalability and good performance during query answering. Through a wide range of configurations and using several real and synthetic datasets, our experimental analysis demonstrates that Odyssey achieves its challenging goals.

1 Introduction

Motivation. Processing large collections of real-world data series is nowadays one of the most challenging and critical problems for a wide range of diverse application domains, including finance, astrophysics, neuroscience, engineering, and others [35, 67, 38]. Such applications produce big collections of ordered sequences of data points, called *data series*. When data series collections are generated, they need to be analyzed in order to extract useful knowledge [31, 63, 27, 49, 9, 10, 33].

This analysis usually encompasses answering *similarity search* queries [35, 15, 16], which apart from being useful for pattern matching, they also lie at the core of several machine learning methods, such as clustering, classification, motif and outlier detection, etc. Moreover, several applications across domains are very sensitive to the accuracy of the results [38, 5], and therefore, require exact query answering, which is what we focus on in this work.

As the size of the data series collections grows larger [35, 38], recently proposed state-of-the-art data series indexes [36, 43, 44] exploit parallelism through the use of multiple threads [43, 45, 44, 40, 41, 13] and the utilization of the SIMD capabilities of modern hardware. However, the unprecedented growth in size that)data series collections experience nowadays renders even state-of-the-art parallel data series indexes inadequate [36, 15, 16, 38, 5, 25]. Achieving

enhanced performance and high scalability, thus, turns out to be one of the most pressing issues in data series processing. To address these issues, all computing elements of modern computing platforms must be exploited, including not only the multi-core and SIMD capabilities of each system node, but also the full capacity of multiple nodes, which enables fast in-memory processing of large datasets (that would not fit in any single node). This is the problem we address in this work.

Challenges. In the context of data series similarity search, exact query answering is very demanding in terms of resources, even when using a data series index. We need to either prune, or visit every single leaf of the index. Previous works [15, 25] though, have shown that pruning is not particularly effective, especially for some hard datasets (such as Seismic [2], which we discuss and use, among other data sets, in Section 4).

The main goal we need to satisfy is (naturally) *scalability*. That is, increasing the available hardware resources (e.g., the number of nodes) should decrease the time cost, ideally by an equivalent amount, or should enable to process an equivalent amount of additional data (at about the same time cost). In order to meet this goal, we need to ensure that all nodes of the distributed system equally contribute to completing the work, during the entire duration of the execution. In turn, this translates to producing effective solutions to the following two problems: (i) *query scheduling*: given a query workload, decide which queries to assign to each system node; and (ii) *load-balancing*: devise mechanisms so that system nodes that have finished their work can help other system nodes finish theirs.

The challenges in this context are the following. First, to achieve effective query scheduling, we need to come up with mechanisms for estimating the execution cost of data series similarity search queries, which do not currently exist. Second, this observation renders a load balancing scheme necessary, yet, this also means that we need to replicate data in order to make such a mechanism viable, as moving big volumes of data series around would be prohibitively expensive. Data replication works against data scalability and is more costly in whatever regards index creation time, but results in better query answering times, thus leading in interesting trade-offs through which an effective solution should navigate. Third, along with all the above considerations, we also need to ensure that our solutions will still maintain their good parallelization properties for efficient execution in multi-core CPUs inside each system node, and also achieve high pruning power during query answering.

Our Approach. We propose a novel *distributed* data-series (DS) indexing and processing framework, called *Odyssey*, that efficiently addresses the high scalability objective by taking advantage of the full computational capacity of the computing platform. To come up with Odyssey, we undertook a long but fascinating journey of understanding the issues to solve and design the appropriate mechanisms in order to deal with the challenges described above for an efficient and highly-scalable *distributed* data series index.

To come up with an appropriate scheduling scheme for Odyssey, we performed a query analysis that shows correlation between the total execution time and a parameter of the category of the single-node data series indexes we consider. This analysis drove the design of efficient scheduling schemes, by generating an execution time prediction for each query of the input query batch.

To achieve load balancing even in settings where predictions may not be accurate or they cannot be effectively used (e.g., when the queries are produced dynamically and are not known a priory), Odyssey provides a load balancing (LB) mechanism, ensures that nodes sitting idle can take away (or *steal*) work from other nodes which have still work to do (provided that these nodes store similar data). Combining Odyssey scheduler with this LB technique results in very good performance and high scalability for all query batches we experiment with.

Ensuring data scalability and, at the same time, good performance for query answering are kind of contradicting goals. A scheme where data are not replicated would result in the lowest space overhead but experiments show that this technique does not ensure the best per-

formance during query answering. This is so because the absence of data replication disallows the utilization of the power of Odyssey's LB mechanism, resulting in load imbalances.

Odyssey manages to effectively unify these two contradicting goals by supporting a flexible *partial replication scheme*, which works for a spectrum of *replication degrees*. This way, it navigates through the fundamental trade-off between data scalability and good performance during query answering. The degree of replication is one of Odyssey's parameters. By specifying it appropriately, developers can choose the time-space trade-off that best suits their application and setting. For example, for small datasets, they could utilize full replication to get the optimal query answering time, while for large datasets, they could select a smaller *replication* degree or no *replication* at all. Odyssey is built in a modular way by following appropriate design decisions and by employing light-weight mechanisms that can be applied on top of one another to ensure good performance in all cases, without adding any big overheads. Experiments illustrate that Odyssey achieves good performance even when the degree of replication is relatively small.

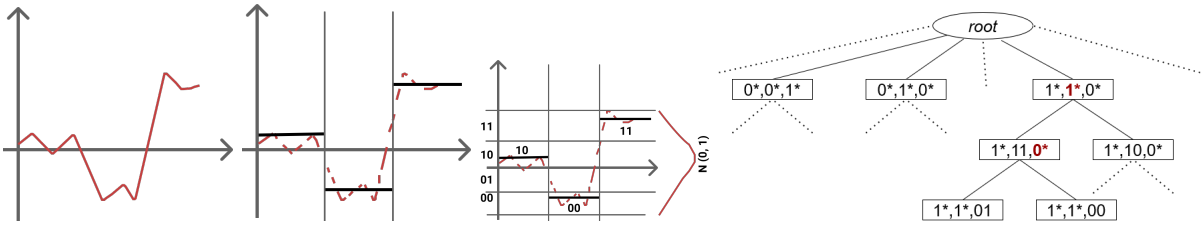
Supporting the components for efficient distributed computation that Odyssey provides, on top of an index that exploits the computation power of a single node as efficiently as state-of-the-art parallel indexes [44, 40], was one more challenging task we undertook while designing Odyssey. A simple approach of using an instance of the state-of-the-art index [44, 40] in each node did not result in good performance mainly for two reasons. First, different data series queries may exhibit variable degrees of locality and their access patterns are only revealed at runtime. This results in low pruning degrees in some of the nodes, having as a consequence severe load balancing problems, which led to performance degradation rather than achieving any speedup (when utilizing more nodes). Second, supporting load-balancing on top of such a simple approach would require moving data around, which is oftenoften prohibitively expensive.

Odyssey *single-node* indexing scheme borrows some techniques from state-of-the-art indexes [43, 45, 44, 40, 41] in order to ensure certain principles that result in low-cost paralellism and high performance within a single node. However, these techniques have been significantly enhanced with new components and mechanisms, to achieve load balancing and come up with a scheme in which work from an overloaded node can be given away to idle nodes without having to pay the prohibited cost of moving any data around. The proposed scheme is innovative in different ways. First, it employs a different pattern of parallelism from all exisitng approaches in travesing the index tree to produce the set of data series that cannot be pruned. Second, it presents new implementations for populating and processing the data structures needed for efficient query answering. To achieve load balancing among the threads, it is critical to choose an appropriate *threshold* on the size of these data structures. After performing a query analysis which revealed a correlation between the median size of the data structures and some parameter of the algorithm, Odyssey proposes an effective mechanism for predicting a good threshold. Additionally, Odyssey provides efficient communication and book-keeping mechanisms, to enable fast exhange of information between the nodes to ensure good pruning degrees in all of them.

Odyssey's best performing algorithm is up to 6.6x faster than its competitors and more than 3.5x better than its best competitor. Additionally, Odyssey's index creation perfectly scales as both the dataset size and the number of nodes increase linearly. Moreover, Odyssey's best performing scheduling strategy is more than 2.5x faster than its initial one.

Contributions. The main contributions of this work are as follows:

- We describe Odyssey, a scalable framework for distributed data series similarity search in clusters with multi-core servers. This makes our approach the first data series solution that exploits parallelization both inside and across system nodes.
- We develop a scheduling algorithm for assigning queries to the nodes of the cluster, which tries to balance the workload across the nodes by computing a (good-enough) estimation of the execution time of each query.
- We present a novel exact search algorithm that supports *work-stealing* between nodes that



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

Figure 1: From data series to iSAX index

share the same index (full replication). Thus, our approach leads to high performance, even when the work is not (or cannot be) equally distributed over the nodes of the cluster. We further extend our solution to work even when only a part of the index is shared among nodes (partial replication).

- Our approach supports different *replication* degrees among the nodes, allowing users to navigate the entire spectrum of solutions, trading space (replication degree) for speed (query answering time).
- We also present a density-aware data partitioning method that can efficiently partition data in a way that improves the work balancing capabilities of our approach.
- Finally, we conduct an experimental evaluation (code and data available online [3]) with a wide range of configurations, using several real and synthetic datasets. The evaluation demonstrates the efficiency of Odyssey, which exhibits an almost linear scale-up, and up to 6.6x times faster exact query answering times than the competitors.

2 Preliminaries and Related Work

Data Series. A *data series*, denoted as $S = \{p_1, ..., p_n\}$, is a sequence of points, where each point p_i is a pair $(u_i, t_i), 1 \leq i \leq n$, of a real value u_i and the position t_i of p_i in the sequence; n is the *size* (or *dimensionality*) of the sequence. When t_i represents time, we talk about *time series*. In several cases, we omit the t_i, e.g., when they are equally spaced, or only play the role of an index for the values u_i [15]; for simplicity, we omit them, as well.

iSAX Summary. The *iSAX summary* [52] of a data series splits the x-axis in equal segments and represents each segment with the mean value of the points of the data series that it contains (see Figure 1). Then it partitions the y axis into regions of sizes determined by the normal distribution and represents each region using a number of bits (*cardinality*). The number of bits can be different for each region, and this enables the creation of a hierarchical tree index (*iSAX-based tree index* [37]; see Figure 1).

Similarity Search. Given a collection of data series $\mathcal{C}$ and an input data series S, *similarity search* is the task of finding the data series in $\mathcal{C}$ which are most similar to S. We focus on finding a single best answer, known as the *1-NN* problem. We also focus on Euclidean Distance (ED). The *euclidean distance* (or *real distance*) between two time series $T = \{t_1, ..., t_n\}$ and $S = \{s_1, ..., s_n\}$ is defined as $ED(T, S) = \sqrt{\sum_{i=1}^{n}(t_i - s_i)^2}$. We call the distance between the *iSAX summaries* of T and S, *lower-bound distance*. The lower-bound distance between any two data series is always smaller than or equal to the real distance between them. We use the term *query* to refer to a similarity search instance.

Single-Node Parallel Summary-Based DS Indexing. Such indexes [43, 45, 44, 40, 42] exploit multiple threads (and SIMD) to create an index tree and answer queries on top of this tree. They are usually comprised of two main phases, the *tree index construction* and the *query*

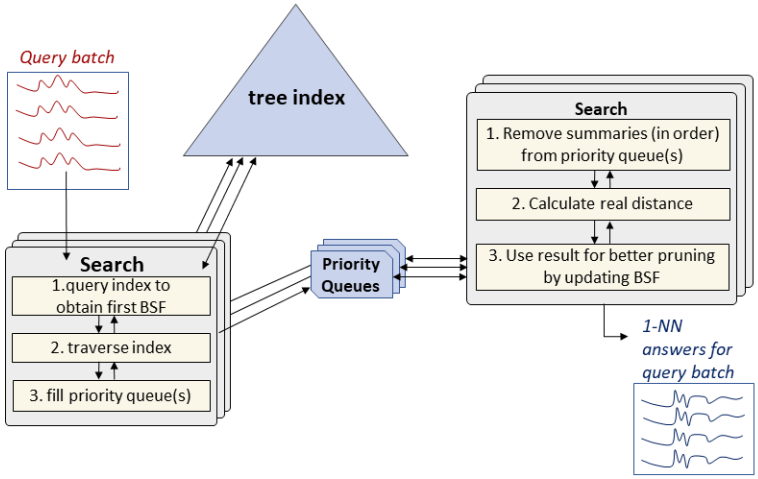


Figure 2: Algorithm Outline of Parallel DS Indexes.

answering phases. In the tree-index construction phase, they first calculate, in parallel, summarizations of all data series in the collection. If the summarizations are iSAX summaries, we talk about *iSAX-based DS indexing*. To achieve a good degree of locality and low synchronization overheads, they store these summaries into a set of *summarization buffers*. Data series that have similar summarizations are placed into the same buffer. Subsequently, the data series of each of these buffers are stored into each of the subtrees of the tree index that they construct. These design decisions allow them to build the tree index in an almost embarrassingly parallel way (thus, without incurring synchronization overheads), and achieve locality in accessing the data during tree construction. They thus respect crucial principles for achieving good performance that should be respected when designing a parallel index.

To answer a query, these indexes first calculate the summarization of the query. Subsequently, they traverse the index tree to find the most appropriate data series based on the iSAX summary lower bound distances. The distance of these data series from the query series is stored in a variable called best-so-far (BSF), and serves as an initial approximate answer to the active query. Then, BSF is used to prune data series from the initial collection. A data series S is pruned when the lower bound distance between S and the query is higher than the current value of the BSF. This process outputs a hopefully small subset of the initial DS collection, containing series that need to be further examined. These series are often stored in (one or more) priority queues [44, 40, 41]. Multiple threads process, concurerntly, the elements of the priority queues, calculating real distance (if needed), and updating the BSF each time a new minimum is met (see Figure 2). Once this process completes, the answer is contained in BSF.

Multi-node Systems and Query Processing. The system consists of a number of asynchronous nodes which communicate by exchanging messages. Each node is a multi-core machine, capable to support multiple threads (and possibly SIMD computation). Threads communicate by accessing shared variables. A shared variable can be atomically read and written. Stronger primitives, such as Fetch&Add may also be provided. Fetch&Add(V, val) atomically adds the value val to the current value of variable V and returns the value that V had before this update.

An arbitrarily large batch of queries is provided in the system as input. The goal is to utilize the system's computational power to execute these queries in a way that minimizes the *makespan*, i.e., the length of time that elapses from the time that any node starts processing a query of the batch to the first point that all nodes have completed their computation. Our techniques can easily be adjusted to work with queries that arrive in the system dynamically.

The data series in the initial collection can be stored in all nodes (*full replication*), or may be scattered to the different nodes so that nodes store disjoint subsets of the data (*no replication*).

A *partial replication* scheme is also possible, where nodes store subsets of the data which are not necessarily pairwise disjoint (e.g., more than one node may store the same subset of data series). A *data partitioning* mechanism determines how to split and distribute the data of the initial data-series collection to nodes.

Query scheduling algorithms aim to schedule the input queries to nodes in a way that each node has approximately the same amount of work to do. Considering full replication, a *Static Query Scheduler (SQS)* partitions the sequence of queries into N subsequences and each node gets one of these subsequences to answer. A *Dynamic Query Scheduler (DQS)* employs a coordinator node, and has other nodes requesting queries to execute from the coordinator. The coordinator may serve requests by assigning the next unprocessed query to a worker when it receives its request, or it may preprocess the sequence of queries (e.g., by re-arranging the queries based on some property) before it starts assigning queries to nodes. To avoid loosing computational power, the coordinator can answer queries itself between serving requests from other nodes.

2.1 Related Work

Data series similarity search queries require the use of specialized index structures in order to be executed fast on very large collections of data sequences. In general, data series indexes operate by pruning the search space based on the summarizations of the series and corresponding lower bounds, and only use the raw data of the series in order to filter out the false positives.

Data Series Indexes. Agrawal et al. [4] presented the first work that argued for the use of a spatial indexing structure for indexing data sequences. This technique works by indexing the first few DFT coefficients of each sequence using an R-Tree [51], and was later optimized by considering the symmetric property of Fourier coefficients [48]. Various indices, specific to data sequences, have been proposed in the literature [14]. Such indices are able to exploit the inherent characteristics of the sequence summarization techniques they employ. DSTree [58] is an index based on the APCA summarization [32]. The DSTree can adaptively perform split operations by increasing the detail of APCA as needed. While this index intelligently adapts its leaf summarizations leading to good data clustering, it is penalized in terms of indexing speed and storage cost. The iSAX index is based on the SAX summarization, and its extension, iSAX [52]. In this case, the data series summarization is bitwise, leading to a concise representation and overall index, as well as enabling fast (bitwise) index operations (including tree traversals and leaf-node split operations). Following the initial iSAX index, several other iSAX-based indices were proposed in the literature [37]. These indexes are among the state-of-the-art solutions in this area [15], including MESSI [40], an in-memory, multi-core and SIMD-enabled version of the iSAX index.

Data Series Management Systems. Several data series management systems have been developed in the last few years [30, 14], several of which operate on top of Apache Spark, or Flink. Beringei [39] has a custom in-memory storage engine. It compresses and organizes data in a series per series scheme. CrateDB [12] partitions data in chunks, stores them in a distributed file system, and indexes them using Apache Lucene. InfluxDB [28] uses Time-Structured Merge Trees (LSM tree variant), logging data on disk as they arrive, and periodically merge-sorting overlapping time-stamps. Prometheus [46] is based on the Beringei ideas. QuasarDB [47] utilizes either RocksDB or Hellium [26]. Riak TS [50] supports both LevelDB or Bitcask, which is a custom log structured hash table. Timescale [55] is a Postgres extension. It partitions time series both in groups of series as well as in distinct time segments. It then provides an abstraction of a single table. IoTDB [57] is geared towards streaming data series. Finally, various systems such as OpenTSDB [34], Timely [54] (concentrated on security) and Warp10 [59] are developed on top of HBase. All the aforementioned systems support range scans in the positions, aggregation functions and filtering. InfluxDB supports queries like moving averages,

prediction, transformations, etc, and Timescale supports gap filling. Nevertheless, none of the above systems supports exact whole-matching similarity search queries.

Distributed Data Series Indexes. KV-Match [60] and its improvement, L-Match [21], are index structures that can support similarity search. These indices can be implemented on top of Apache HBase, and operate in a distributed fashion within Apache Spark. We note that these solutions only support subsequnce similarity search, and not whole-matching [15], which is the focus of this work. TARDIS [64] is an Apache Spark system for similarity search. It supports approximate queries, as well as *exact match* queries, where we want to know if the query appears *exactly the same* within the dataset, or not. This query type is much easier that the exact queries we consider in our work, and cannot be efficiently transformed to exact querying. Finally, DPiSAX [61, 62] is a distributed solution for data series similarity search, developed for Apache Spark using Scala. Even though it was designed for answering batches of approximate search queries, it supports both approximate and exact search queries. DPiSAX exploits the iSAX summaries of a small sample of the dataset, in order to distribute the data to the nodes equally. Then, an iSAX index is built in each node on the local data, and is used to perform query answering. In order to produce the exact search results, all nodes need to send their partial results to the coordinator, which merges them and produces the final, exact answer.

In this study, we compare to DPiSAX, which the only distributed data series index solution in the literature that supports exact search. In order to have a fair comparison, we implemented the data distribution mechanism of DPiSAX in C, and integrated it with the MESSI index [40].

Work-stealing was employed in the Cilk framework [1]. The work-stealing approach was formally studied and analyzed in [8, 19, 20]. A big amount of work has been done on this topic (with [6, 7, 18] being just a few examples).

3 The Odyssey Framework

This section presents the main mechanisms employed by the Odyssey framework. All techniques discussed here are provided in the framework.

3.1 Query Scheduling

To correctly answer a query, it should be forwarded to a set of system nodes which collectively store all the data. Thus, in the no-replication case this set contains all system nodes, thus a scheduling algorithm should forward all queries to all nodes. The full replication case is more interesting as it enables the utilization of different scheduling techniques.

To come up with Odyssey scheduler, we experimented with a collection of scheduling techniques, including the simple static and dynamic schemes (SQS and DQS) for full replication settings, discussed in Section 2. Unfortunately, these schemes suffer from severe load balancing problems for many categories of query batches. For the static case, consider for example, a query sequence which consists of progressively more *difficult* queries (i.e., of queries that each requires less time to run than the next one). SQS will assign to the first system nodes easy queries, which they will answer fast, while the last nodes will get more work to do. The dynamic method (DQS) performs better than SQS for some workloads, but it may also result in load imbalances: even in simple cases where e.g., a query batch includes a single difficult query at the end, most nodes may be sitting idle, while a single node is running the difficult query. This may significantly degrade performance.

Some of these load imbalances could be avoided, if we knew the execution time of each query. Recent work [24] in a different context, illustrated that there exists a correlation between the

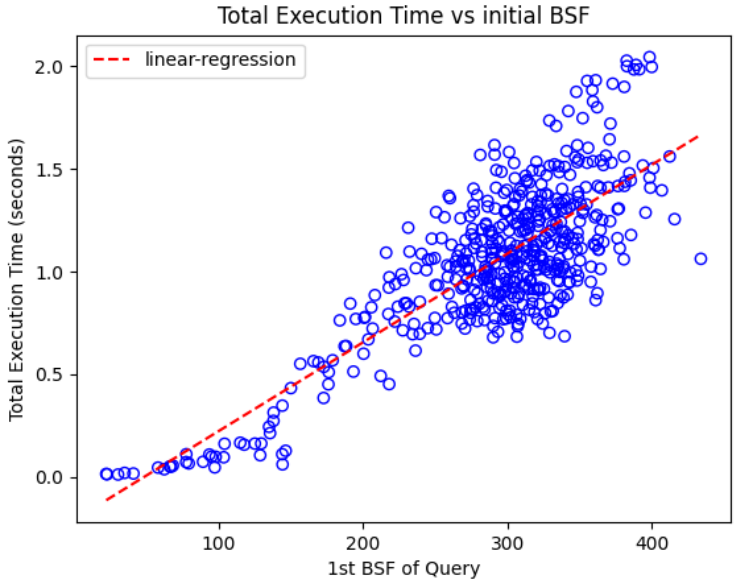


Figure 3: Seismic queries prediction generation using linear regression

initial value of the BSF variable and the number of vertices visited in a single-node tree index. This inspired us to perform a corresponding query analysis which showed correlation between the initial Best-So-Far value and the total execution time. Specifically, similarity search queries, for which the *initial BSF* is high, tend to also have high execution times most of the times.

For the analysis, we used a number of queries for a real dataset. Then, we utilized linear regression to produce the estimation for each query. An example of this outcome (for the Seismic dataset) is shown in Figure 3. Nevertheless, we note that Odyssey can support other, more elaborate prediction schemes, if needed.

These observations led us to design two scheduling algorithms. The first, called *static prediction-based scheduling*, statically allocates the queries to nodes based on their estimations. Each node maintains a *load* variable, which stores the sum of the estimations of the queries that are assigned to it. The algorithm uses a greedy approach to assign queries to nodes so that load balancing is achieved. There are two variations of the algorithm: the first schedules the queries using their order in the sequence, and the second sorts the sequence before starting the process The second, called *dynamic prediction-based scheduling*, is an enhanced version of DQS, where queries as assigned to nodes after sorting the entire query batch, based on estimations (in decreasing order, so that difficult queries are assigned first).

The Odyssey framework supports all of the above algorithms (which are meaningful not only under full replication, but also under partial replication as explained later). The Odyssey index utilizes dynamic prediction-based scheduling which turned out to be the best approach in most cases.

3.2 Load Balancing

Odyssey provides a load balancing (LB) mechanism, which can be applied on top of any of the scheduling schemes described in Section 3.1. Odyssey's LB mechanism ensures that nodes sitting idle can *steal* work from other nodes which still have work to do (provided that they store similar data).

This is necessary as predictions may not always be accurate, or the query batch may be produced dynamically at run time, in which case sorting of the entire query batch is not possible.

The work-stealing mechanism of Odyssey is necessary not only for ensuring good performance but also for achieving high scalability. As the number of utilized nodes increases, the number of queries from the batch that each node has to process becomes smaller and smaller. Thus, problematic scenaria as those described in Section 3.1, may appear, where just one or a few nodes work on difficult queries, while others are sitting idle (without having enough work to do).

Overview of our approach. We performed a number of experiments to get a break-down of the query answering time. This break-down illustrated that the biggest part of the time for query answering goes to priority queue processing. We thus focus on designing a method that will allow nodes to steal work during the execution of that phase.

For simplicity, we first focus on the full-replication case, where the initial collection of data is available in every node. Its extension to partial replication is discussed in Section 3.3.

A simple work-stealing scheme [8, 1] would not work, mainly because moving data (stored in priority queues) around from one node to another is expensive and should be avoided. Thus, the main challenge in our setting is to come up with a mechanism for taking work away from one node and assigning it to another without ever moving any data around.

Odyssey load-balancing mechanism works as follows. An idle system node sn randomly chooses another node sn' and sends it a steal request. If sn' has still work to do, it chooses a number of priority queues to give away to sn. To avoid paying the cost of transferring data around, Odyssey employs a technique that informs sn on how to locally build the priority queues to work on, based on its own index. Node sn traverses the identified part of its index tree and re-constructs these priority queues. As the time to create the priority queues is relatively small in comparison to that for processing them, this scheme works quite well. Experiments show that stealing work in this way, results in performance gains in several cases.

Unfortunately, the approaches followed by existing state-of-the-art indexes [44, 40, 41] for creating and processing the priority queues are too naive to support a work-stealing scheme which requires to avoid moving any data around. The reasons is that existing approaches use simple techniques to populate the priority queues in order to ensure that each queue has about the same number of elements spreading tree leaves from many subtrees to the same priorioty queue (for instance, in [44, 40, 41], each thread follows a round-robin scheme for placing leaves that cannot be pruned in the priorities queues). This disallows the local re-construction of stolen priority queues in an efficient way, because it is hard (or very costly) to track what elements each priority queue contains.

In Odyssey, we propose a new implementation of a single-node, multi-threaded index, which respects the good design principles described for parallel indexes in Section 2, while it simultaneously copes with the problem mentioned above. We describe this implementation in more detail in Section 3.2.1.

3.2.1 Single-Node Query Answering

Consider any system node sn and assume that an iSAX-based index tree has been created and an initial value for the BSF has been computed in sn. An outline of the single-node query answering algorithm of Odyssey is depicted in Figure 4. The pseudocode is provided in Algorithms 1 and 2.

Description. Node sn executes each of the queries in the query batch assigned to it one by one (Algorithm 1, line 3). For each such query Q, it creates a number of search workers to execute it (line 8). FinishFlag[Q] identifies whether the processing of query Q has been completed in sn. As soon as, all queries in sn's query batch have been processed, sn informs other nodes that it has completed by sending a message of type DONE to all other nodes (line 14). Then, it tries to help other active nodes by executing PerformWorkStealing (line 15). Each node allocates a thread to play the role of the work-stealing manager (line 6). This thread simply

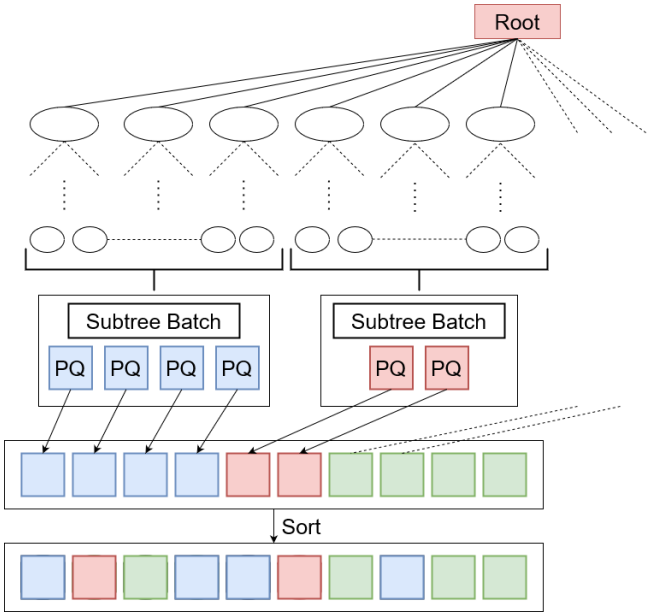


Figure 4: Outline of the Odyssey single-node query-answering process.

process all work-stealing requests that the node will receive (Algorithm 3). (The details of how work-stealing works will be discussed in Section 3.2.2.) We now continue to describe the code of the search workers (Algorithm 2).

The query answering algorithm in *sn* splits the tree into *root subtree (RS) batches*, i.e., sets of consecutive root subtrees, and allocates a number of threads to work on these RS-batches. Each thread begins by getting an RS-batch to work on. Then, the thread begins an RS-batch processing routine (called *ProcessBatch* in Algorithm 2), which traverses the tree recursively and inserts the leaves that cannot be pruned into one of a set of priority queues that belong to the RS-batch. Each tree vertex x has an iSAX summary, which corresponds to the data series stored in the subtree rooted at the vertex. If the lower bound distance between this iSAX summary and the iSAX summary of the query series is higher than BSF, then the entire subtree rooted at x can be pruned, as in this case, we know that we have already seen a data series with a smaller real distance to the query than any element stored in the subtree. Otherwise, if x is a leaf, it is placed in one of the priority queues of the RS-batch, using the lower bound distance between x and the query series, as its priority. For every RS-batch, there exists one active priority queue at each point in time. When the size of this priority queue surpasses a threshold, this queue is abandoned and another one is initialized for the RS-batch.

Each worker is assigned an RS-batch using a Fetch&Add object A *completion flag* for each RS-batch is set when all subtrees of the RS-batch has been traversed and the priority queues of the RS-batch have been populated (lines 27, 32). As soon as an idle thread *th* discovers that all RS-batches have been assigned for processing, it tries to help other still active threads to complete the traversal in their assigned RS-batches (lines 29-34 of Algorithm 2). Thus, it checks the completion flag of RS-batches and if it discovers some RS-batch whose completion flag is still not set, it starts helping the thread th' that has been assigned the RS-batch. Coordination between th' and its helpers is achieved using again a Fetch&Add object. This way, each root subtree of an RS-batch is assigned to just one thread. Synchronization in accessing a priority queue is achieved using locks. To reduce the synchronization cost, there is threshold, $HelpTH$, on the number of threads that may help on each RS-batch (line 30).

This phase ends, when the root subtrees of all RS-batches have been traversed and all priority queues have been populated. Experiments illustrated that we get the best performance when the number of subtree batches, N_{sb}, equals the number of worker threads.

As soon as this *tree traversal phase* is over, we have a set of priority queues for each RS-batch, stored in an array. For performance reasons, this array is sorted by the priority of the top element of each priority queue. This comprises the *priority queue preprocessing* phase (lines 35-42). Specifically, a coordinator thread (which is the thread with $tid = 0$ in Algorithm 2), puts the priority queues of all RS-batches into a shared array (line 38), and then sorts the array using the priorities of the root elements of the priority queues (line 39). This way, the algorithm processes the priority queues with the smallest lower bound distances to the query first. These queues contain data series that are more probable to be in closer real distance to the query, thus enabling further pruning.

Then, the *priority queue processing* phase starts (lines 44-53). All threads perform Fetch&Add to get a priority queue from the PQueues array to process. This processing is performed by a routine, called *ProcessPriorityQueue*. *ProcessPriorityQueue* processes those data series contained in the tree leaves that are stored in the priority queue, which cannot be pruned. Whenever a lower real time distance between any of these series and the query series is calculated, the BSF is updated to contain this distance. The final value of BSF is the response to the query.

Algorithm 1: Odyssey Single-Node Query Answering - Code for node *sn*

```
1  Shared Variables: Shared PointerToArray PQueues = NULL
   Input: QuerySeriesBatch QBatch, Index Index, Integer NThreads
2  Array BSFArray[] ▷ with size |QBatch|
3  for every query series id Q in QBatch do
4  |  iSAX_Q = calculate iSAX summary for Q
5  |  BSF = approxSearch(iSAX_Q, Index)
6  |  create a thread to execute an instance of WorkStealingManager(Q)
7  |  for i ← 0 to NThreads − 1 do
8  |  |  create a thread to execute an instance of SearchWorker(Q, Index, N_sb, i, PQueues )
9  |  end
10 |  Wait for all threads to finish
11 |  FinishFlag[Q] := TRUE
12 |  BSFArray[Q] := BSF;
13 end
14 send(DONE, sn) to all nodes
15 PerformWorkStealing()
16 return (BSFArray)
```

Size of Priority Queues. The size of each priority queue is maintained smaller than or eqaul to a specific threashold, TH. If adding an element in a priority queue results the size of the queue to reach TH, then the thread gives up this priority queue and initiates a new one for the RS-batch. This way, each priority queue does not contain leaves from more than one RS-batch, and contains at most TH leaves from the tree part that corresponds to the RS-batch.

Choosing the appropriate value for the threashold, TH, is important for achieving load balancing among the different threads. Our goal is to develop a method for determining a threshold value which will result with a set of priority queues that have about the same size. The threshold is determined and configured for every dataset we use, based on the queries we run. We explain the process of determining TH for the Seismic real dataset [2], but the process is similar for all other datasets (real or synthetic) we experimented with. After running multiple queries of varying difficulty, we figured out that there exists again a correlation between the initial BSF that is computed for the query and the median size of the priority queues produced for answering it. Then we performed a sigmoid function fitting using the following parameterized formula:

$$f(Z) = m + (M - m)\frac{1}{1 + b \cdot exp(-c(Z - d))}$$

where $M \in [0, 1], m \leq M, b, c \in \mathbf{R}^*, d \in \mathbf{R}$ are the parameters of the sigmoid function (Fig-

Algorithm 2: SearchWorker - Code for thread *tid*

```
17 ▷ Shared Variables
18 Shared Integers BCnt = 0, PQCnt = 0, TotPQ = 0;

   Input: QuerySeries Q, Index Index, Integer N_sb, Integer tid, Queue PQueues[]

19 Integer bindex, pqindex;
20 ▷ Tree Traversal Phase
21 while (TRUE) do
22 |  bindex ← Fetch&Add(BCnt,1);
23 |  if bindex ≥ N_sb then
24 |  |  break;
25 |  end
26 |  ProcessRSBatch(Q, bindex, Index.RSBatches);
27 |  Index.RSBatches[bindex].complete ← TRUE;
28 end
29 for bindex ← 0 to N_sb do
30 |  if !Index.RSBatches[bindex].complete AND
   |    Fetch&Add(Index.RSBatches[bindex].helped, 1) < HelpTH then
31 |  |  ProcessBatch(Q, bindex, Index.RSBatches);
32 |  |  Index.RSBatches[bindex].complete ← TRUE;
33 |  end
34 end
35 Barrier for all threads;
36 ▷ Priority Queue Preprocessing Phase
37 if tid == 0 then
38 |  Traverse all RS-batches and put their priority queues into PQueues[];
39 |  SortByRootPriority(PQueues);
40 |  TotPQ ← number of valid elements of PQueues;
41 end
42 Barrier for all threads;
43 ▷ Priority Queue Processing Phase
44 while (TRUE) do
45 |  pqindex ← Fetch&Add(PQCnt, 1);
46 |  if pqindex ≥ TotPQ then
47 |  |  break;
48 |  end
49 |  if PQueues[pqindex].stolen then
50 |  |  continue;
51 |  end
52 |  ProcessPriorityQueue(PQueues[pqindex]);
53 end
```

ure 5a(a)). The final threshold value is the median value estimation as it comes from the sigmoid function, divided by a factor (e.g. for seismic this factor has to be 16, based on the diagram shown in Figure 5a(b)).

Experiments show that after the tree traversal phase is completed, we end up with a set of RS-batches that have a number of priority queues with most of them being the same size. This results in load balancing among the threads when processing priority queues.

3.2.2 Work-Stealing Algorithm

If a system node *sn* becomes idle, *sn* initiates the work-stealing protocol (Algorithm 4, lines 79-81). It randomly chooses a system node sn' from the set of those nodes that *sn* knows to be still active (line 79) and sends a steal request to it (line 80). The use of the ResponseFlag is

Algorithm 3: WorkstealingManager - Code for node sn

Input: Integer N_B

54 **Upon Receiving a message of type StealingRequest from node** sn'**:**
55 S := Set of at most N_{send} ids of RS-batches that satisfy the Send-Away Property
56 send(S, Q of sn, Q's current BSF) to sn'
57 Mark the priority queues of the RS-batches with ids in S as stolen

58 ▷ Always-enabled event: it is executed repeatedly
59 **Upon receiving no message:**
60 **if** *FinishFlag*[Q] *in sn is set* **then**
61 | Terminate
62 **end**

Algorithm 4: PerformWorkStealing - Code for node sn

Input: Index *index*, **Function** *exact_search_workstealing_func*, **QuerySeries** *queries*[], **Integer** *total_nodes_per_nodegroup*

63 **Upon Receiving a DONE message from node** sn'**:**
64 add sn' in set *DoneNds*
65 **if** *DoneNds contains all system's nodes* **then**
66 | Terminate
67 **end**

68 **Upon Receiving a** $msg = \langle S, Q_s, BSF_s \rangle$ **from node** sn'**:**
69 **if** —S— ¿ *0* **then**
70 | Create threads to traverse the RS-batches with ids in S
71 | Populate and process the corresponding priorities queues
72 | BSFArray[Q_s] := BSF_s; ▷ computed by threads above
73 | Wait all threads to complete
74 **end**
75 ResponseFlag := 0

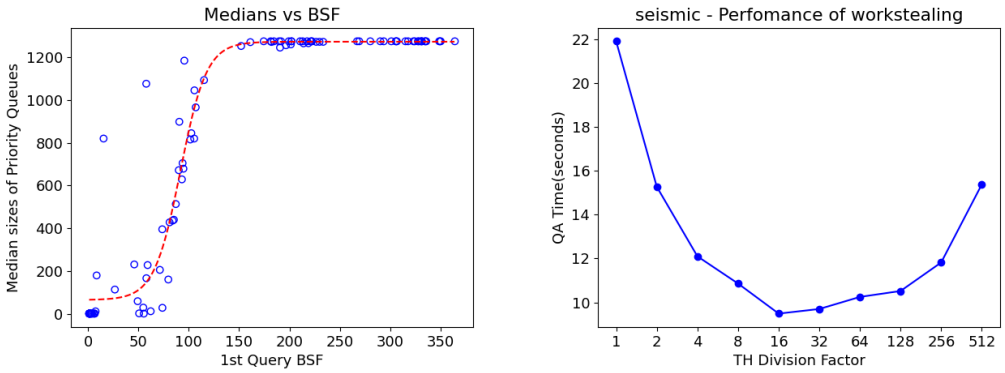
76 ▷ Always-enabled event: it is executed repeatedly
77 **Upon receiving no message:**
78 **if** *!(ResponseFlag)* **then**
79 | sn' := choose randomly a node not in *DoneNds*
80 | send(StealingRequest, sn) to sn'
81 | ResponseFlag := 1
82 **end**

necessary for avoiding sending more than one steal requests to the same target node[1].

A thread in each node acts as the work-stealing manager (Algorithm 3). As soon as the work-stealing manager of sn' receives the request, it tries to give away work to sn (lines 55-57 of Algorithm 3). Recent work in a different context [24], illustrated that there is a gap between the time the correct answer is recorded in BSF and the time when the search algorithm terminates. This is because a large amount of time during the execution of the query answering algorithm is devoted to verifying that there is no better answer in the collection. Specifically, in [24], evidence is provided that the 1-NN is often found in a small amount of time, but it takes the search algorithm much longer to verify that there is no better answer, in order to be able to safely terminate.

Based on these findings, Odyssey work-stealing mechanism, chooses to give away an RS-batch B which satisfies the *Take-Away Property*, namely that B is not yet stolen and its first priority queue is located in the rightmost possible index of the *PQueue* array. This priority

[1]The codes for Algorithms 4 and 3 is written in an event-driven style [**?**, **?**]. The code under each **upon event E takes place** clause is executed atomically each time event E is enabled. The **upon receiving no message** event is always enabled, meaning that the code under it may be executed an arbitrarily large (or even infinite) number of times. The use of ResponseFlag ensures that this code will run only once for each target node.



(a) Sigmoid function fitting for determining TH.

(b) Performance for different Threshold division factors.

Figure 5: Odyssey Single-Node Query-Answering Algorithm Configuration.

queue is then marked as stolen. If more than one batches are to be given away, this process is applied repeatedly to choose additional RS-batches.

Recall that the $PQueue$ array is sorted by the priority of the top element of each priority queue. Thus, by giving away batches in this way, sn' assigns to helpers priority queues that may still contain work. Additionally, it gives away RS-batches that have the highest probability to be unprocessed. Throughout the process, the current BSF is shared among the nodes, every time it is updated, as a helper may steal a priority queue that contains a better answer (or the owner may compute a better value for BSF later).

The number, N_{send}, of RS-bathes that a node gives-away during stealing affects performance. Theoretically, we would like to give away a number of RS-batches which on the one hand, it will enable the stealing node to do a noticeable amount of work, but on the other, the work to be given away should not result in higher query answering times. Experiments shows that fixing the number of RS-batches a node gives away during stealing to 4 was the best choice. Thus, $N_{send} = 4$ in our work.

3.3 Data Replication

Odyssey aims at ensuring data scalability and, at the same time, good performance for query answering. Optimal data scalability requires to follow a no replication approach, but experiments show that the best query answering performance is noticed for fully replicated settings. Odyssey manages to effectively navigate through this trade-off between data scalability and good performance during query answering, by providing a flexible *partial replication scheme*. Figure 6 schematically illustrates how this scheme works in a system with 8 nodes.

The idea is to split the set of nodes into a number of clusters such that i) each node participates to exactly one cluster and ii) the set of data series stored in each node of a cluster are mutually disjoint (within the cluster), while their union forms the entire data series collection. Additionally, a *replication group* is defined as the group of nodes that contain exactly the same set of data series. Specifically, Figure 6 illustrates a configuration with 2 clusters, each one having 4 nodes (of different color), and 4 replication groups, each one having 2 nodes (of same color). Since the nodes of a replication group build their iSAX indices using the same set of data series, we can apply the scheduling and load-balancing schemes, described in Sections 3.1 and 3.2, respectively.

Notice that (for a fixed number of nodes) the number of clusters determines both the number of replication groups and the *replication degree* of the system. In a configuration with d clusters

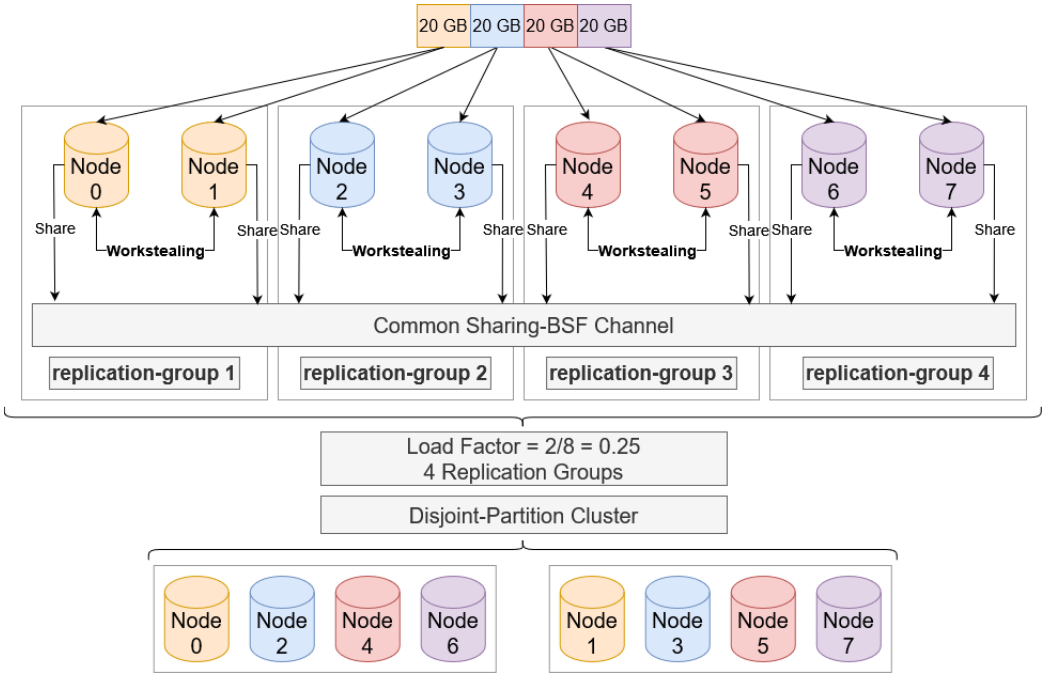


Figure 6: Data Replication with four replication groups.

and N nodes, each replication group contains N/d nodes and each node stores d times more data than in the no-replication case. Fewer clusters lead to smaller space overheads (and thus better data scalability). For example, in a configuration with N clusters, all nodes store the entire data series collection, whereas using only a single cluster corresponds to the no-replication case, thus incurring no space overhead at all. Therefore, Odyssey's data replication scheme allows us to tackle memory limitation problems.

Additionally, fewer clusters lead to better index creation scalability. To better understand this, we define the *load factor* to be the fraction of the number of nodes that share the same data divided by the total number of nodes in the system. This metric represents the percentage of the dataset that each node loads. For example, in full replication, the number of clusters is N and the load factor is 1, meaning that each individual node will load 100% of the dataset.

We experimented with different number of clusters in Section 4 to illustrate that Odyssey nicely navigates through the discussed tradeoffs. Specifically, considering a system with N nodes (where N is a power of 2), Odyssey's data replication scheme supports $1, 2, 4, 8, \ldots, N$ clusters, i.e. $\Theta(\log N)$ different number of clusters, and replication groups of the same size. However, it can easily be adjusted to work for settings with any number of nodes (not necessarily a power of 2) and for replication groups of different sizes.

3.4 Data Partitioning

Odyssey framework supports more than one partitioning schemes. *Equally-split (ES)* is a simple scheme which splits the input data into N equally-sized chunks, where N is the number of system nodes. Each system node is assigned a discrete chunk and builds the corresponding index, resulting in a scheme where each node keeps a local index on its own part of the data. Queries are forwarded to all nodes. Each node produces an answer based on its local index and data. The minimum among them is the final answer to the query. Before distributing the data, *random shuffling (RS)* can be applied to randomly rearrange the series of the initial collection.

To answer a query batch using partial data replication, each query is sent to every system node. Each such node answers the query using its local data, and the partial answers are

gathered in the end to find the smallest answer. Very often for real data, the close answers to a query could be located into a small part of the dataset. The group that has these data will get a good initial answer, it will prune more and it will answer each query really fast, while other groups, will not necessarily compute good initial BSF values. Thus, they will have more work to do leading to imbalances. For this reason, we enhance our distributed index with a book-keeping method that supports BSF sharing. When a node is processing a query and finds an improved value for BSF, it shares this value through a common BSF-Sharing channel (as illustrated in Figure 6). Every node periodically checks this channel to see if an answer for a query has arrived. Because this process runs in parallel, a node may receive a better answer for a query that will be encountered later on. Odyssey book-keeping method solves such synchronization problems. Each node holds an array which stores the improvements received from the channel for the BSF of each query, and before answering a query it checks the data held in this array. This way, each node can have the better answers of a query extracted from all different nodes, and our experimental evaluation shows that the use of this method is critical for performance.

In addition to these simple techniques, Odyssey also provides a sophisticated data partitioning scheme, based on preprocessing of the initial data series collection, which provides a density-aware distribution of the data among the available nodes. The required preprocessing incurs some time overhead. However, it occurs only once for answering as many queries as needed, and thus, as the number of queries to process increases, this overhead is amortized. We dercibe this scheme in Section 3.4.1.

3.4.1 Density-Aware Data Partitioning

We observe that a good partitioning strategy should not assign all similar series to the same system node. In such a case, we risk to create work imbalance for the following reason. Assume that we need to answer a similarity search query, for which all candidate series from the dataset that are similar to the query are stored in one of the system nodes, while all other nodes are storing series that are not similar to the query. Then, during query answering, the node with the similar series will need to perform many (lower bound and real distance) computations in order to determine which of the candidate series is the nearest neighbor to the query, with essentially little pruning (if at all). On the other hand, all the other nodes that store dissimilar series will be able to prune aggressively, and therefore, finish their part of the computations much faster.

The above observations led us to the design of the DENSITY-AWARE partitioning strategy, whose goal is to partition similar series across all system nodes, without incurring a high computational cost. This is achieved by exploiting Gray Code [23] ordering for effectiveness (since it helps us split the similar series), and the summarization buffers of our index for efficiency (since we have to operate at the level of buffers, rather than individual series).

Figure 7 illustrates an example of partitioning the data series in the summarization buffers according to a simple strategy using binary code, and to a strategy based on Gray Code. In the former case, the buffers that end up in the same node contain similar series, since their iSAX represention (the iSAX word of the buffer) are very close to one another; e.g., node 1 stores buffers "000" and "001", so series whose iSAX summaries only differ in one bit. In the latter case, this problem is addressed. The Gray Code ordering places similar buffers close to one another (by definition, two neighboring buffers in this order differ in only one bit), so it is then easy to assign them to different system nodes in a round-robin fashion.

We depict the flowchart of the DENSITY-AWARE partitioning strategy in Figure 8. We start by computing the iSAX summaries of the data series collection, and assigning each summary to the corresponding summarization buffer. These buffers are ordered according to Gray Code, and then the actual data partitioning starts (using round-robin scheduling). We first partition the series inside the λ largest buffers; this is necessary, since often times a small number of buffers will contain an unusually large number of series (that we do not want to assign them

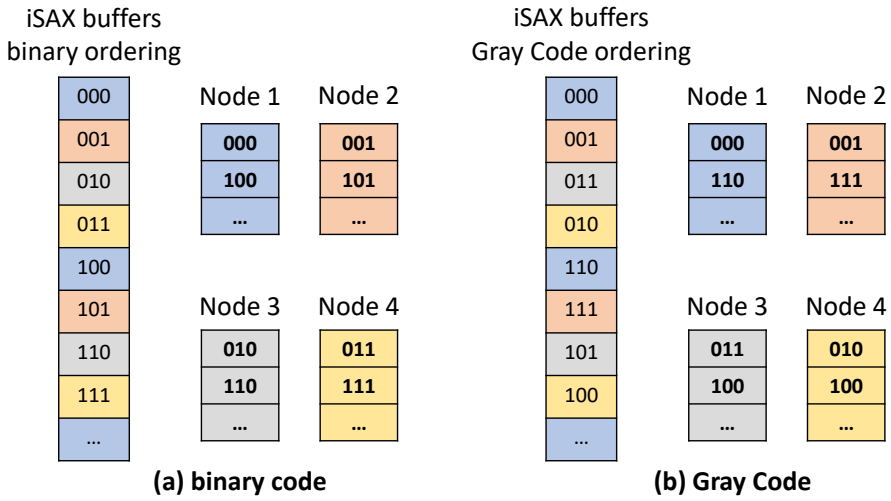


Figure 7: Examples of partitioning the iSAX buffers' data to 4 system nodes, based on (a) simple iSAX and (b) Gray Code.

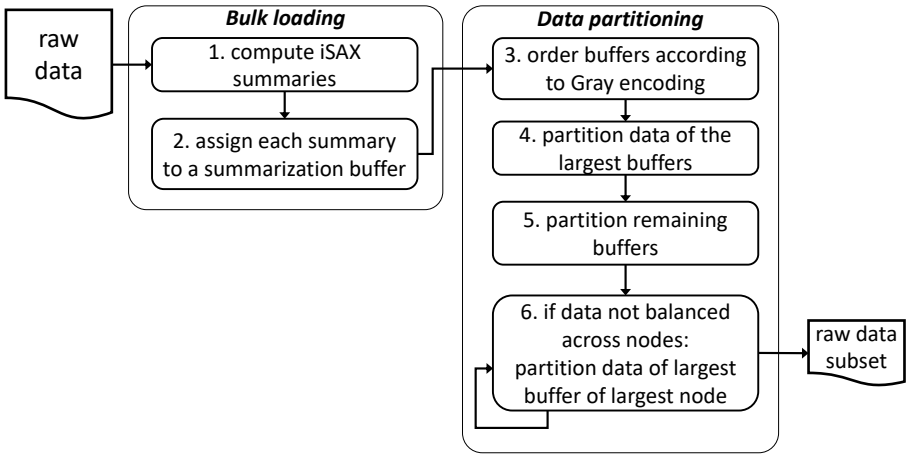


Figure 8: Flowchart of the DENSITY-AWARE data partitioning.

all to the same system node). Then, we partition the remaining buffers, and we check if the partitioning is balanced. If it is not, then we select the largest buffer of the largest node, and we partition the series inside this buffer.

Our experiments with several real datasets (omitted due to lack of space) showed that DENSITY-AWARE exhibits a very stable behavior as we vary λ from a a few hundred to several thousands. In this study, we use $\lambda = 400$.

4 Experimental Evaluation

Setup. Our experiments were conducted on a cluster of 16 SR645 nodes, connected through an HDR 100 Infiniband network. Each node has 128 cores (with no hyper-threading support), 256GB RAM, and runs Red Hat Enterprise Linux release 8.2. All the evaluated algorithms are written in C and compiled using MPICC, Intel(R) MPI Library for Linux OS, Version 2021.2.
Algorithms. In our experimental evaluation, we test the different strategies and algorithms proposed by the Odyssey Framework. Our analysis includes Odyssey's dataset distribution strategies: (i) without replication: distribution of equally-sized chunks (EQUALLY-SPLIT), the

Dataset	# of series	Length (floats)	Size (GB)	Description
Seismic	100M	256	100	seismic records
Astro	270M	256	265	astronomical data
Deep	1B	96	358	deep embeddings
Sift	1B	128	477	image descriptors
Random	100M	256	100	random walks

Table 1: Details of datasets used in experiments.

iSAX summarization-aware distribution algorithm (ISAX-AWARE), the density-aware dataset distribution algorithm (DENSITY-AWARE); and (ii) with replication: full replication of dataset across all nodes (FULL) and partially replicated dataset using $k \in \{1, 2, 4, \ldots, N\}$ replication groups. Note that in the latter case, PARTIAL-N is the same with FULL and PARTIAL-1 is the same with any of the former strategies, i.e. the ones without replication. Additionally, it includes Odyssey's queries scheduling algorithms: (i) static scheduling of equally sized sets of queries (STATIC); (ii) dynamic scheduling using a coordinator node (DYNAMIC); and (iii) predictions-based scheduling, including: static without ordering (PREDICT-ST-UNSORTED), static with ordering (PREDICT-ST), and dynamic (PREDICT-DN). Moreover, we evaluated Odyssey's work-stealing mechanism using both DYNAMIC and PREDICT-DN, with the latter being our best queries scheduling algorithm (as explained later in paragraph "Queries scheduling"); the resulting algorithms are called WORK-STEAL and WORK-STEAL-PREDICT, respectively.

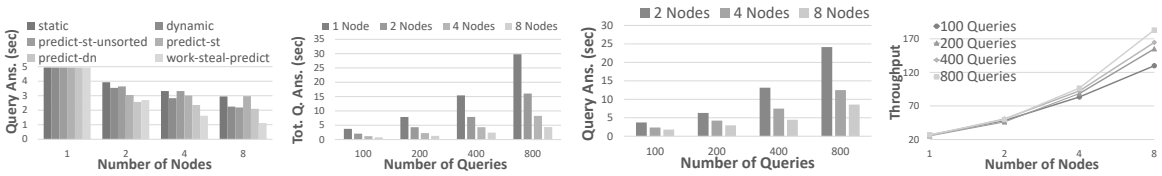
We note that that Odyssey's query scheduling and work-stealing mechanisms can be used together only with the FULL or PARTIAL dataset distribution strategies (which include some dataset replication). distribution strategies that do not include dataset replication.

We compare Odyssey with the following baselines: (i) MESSI, where we run the MESSI index independently in each system node; (ii) MESSI SW BSF, where we extend the previous solution by enabling system-wide sharing of the BSF values; and (iii) DPiSAX, where we implement the DPiSAX data partitioning strategy, and (for fairness of comparison) implement query answering in each node using MESSI.

Datasets. We evaluated Odyssey's strategies and algorithms using both real and synthetic datasets, of varying sizes. The synthetic data series, called *Random*, were generated as random-walks (i.e., cumulative sums) of steps that follow a Gaussian distribution (0,1). This type of data has been extensively used in the past [17, 11, 66, 65, 15, 16], and models the distribution of stock market prices [17]. Our four real datasets come from the domains of seismology, astronomy, deep learning, and image processing. The seismic dataset, *Seismic*, was obtained from the IRIS Seismic Data Access archive [22]. It contains seismic instrument recording from thousands of stations worldwide and consists of 100 million data series of size 256. The astronomy dataset, *Astro*, represents celestial objects and was obtained from [53]. The dataset consists of 100 million data series of size 256. The deep embedding dataset, *Deep*, retrieved from [56], contains 267 million Deep vectors of size 96 extracted from the last layers of a convolutional neural network. Finally, the *Sift* dataset [29] contains image descriptors. Table 1 summarizes the characteristics of these datasets.

Evaluation Measures. During each experiment and for each *node*, we measure (i) the *buffer time* required to calculate the data-series iSAX summarization and fill-in the receive buffers, (ii) the *tree time* required to insert the items of the receive buffers in the tree-index, and (iii) the *query answering time* required to answer the queries assigned to this node. The above times (all together) constitute the *total time* that a *node* works during an experiment; also, buffer and tree times constitute the time required to create the index, called *index time*. To compute the buffer time, tree time, index time, query answering time, and total time of the *system* during an experiment, we take the maximum among the corresponding times of each node participating in this experiment. We report the average times of 10 experiments. We note that all algorithms return in all situations the exact, correct results.

Query scheduling. To compare Odyssey's queries scheduling algorithms, the fully replicated



(a) FULL replication. (b) FULL replication. (c) PARTIAL-2 replication. (d) FULL replication.

Figure 9: (a) Comparison of Odyssey's scheduling algorithms, using Seismic and FULL replication. (b)-(d) Query answering scalability as the number of queries increase, using WORK-STEAL with Random.

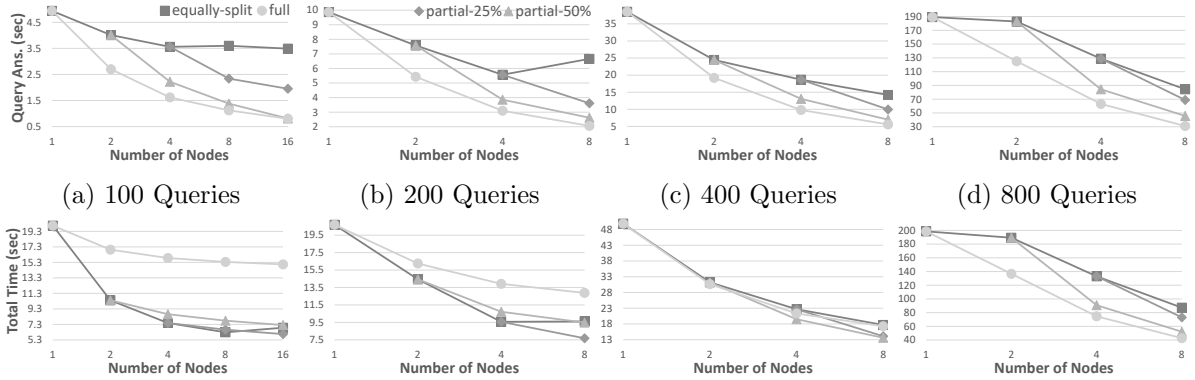
dataset distribution strategy (i.e. FULL) is selected, to avoid measuring any overheads resulting from the partial replicated strategies. Recall that scheduling algorithms can't be used together with the no replication strategies. We experimented with both Random (synthetic dataset) and Seismic (real dataset), and all of our algorithms positively affected performance in comparison with STATIC. Moreover, for the synthetic dataset, we have seen no remarkable differences between all our scheduling algorithms, since the randomness when producing the data series of both the dataset and the queries set, results in queries with almost the same effort to be answered. Therefore, we present only the results for Seismic, where the effort for answering queries varies. Specifically, Figure 9a shows that as the number of nodes increases, PREDICT-DN is the best scheduling policy in all cases and it is up to 150% better than STATIC.

Work-stealing. Figure 9a shows that WORK-STEAL-PREDICT greatly outperforms (up to almost 2x) PREDICT-DN for large number of nodes, i.e. our work-stealing technique positively affects performance on these cases. This happens since (as explained in Section 3.2.2) all the algorithms that do not use the work-stealing technique (i.e. all but WORK-STEAL and WORK-STEAL-PREDICT) suffer from load-imbalance issues. Specifically, when a query set contains a few (significantly less than the number of nodes) queries that require significantly more effort to get answered (than the majority of queries), then as the number of nodes increases more nodes remain idle at the end of the corresponding query answering phase, since no such difficult query is assigned to them. This still holds even when using the best queries scheduling policy.

Query Scalability. To evaluate the scalability of Odyssey's algorithms with increasing number of queries, we conducted experiments with WORK-STEAL using synthetic and real datasets. In Figure 9b we present the results for the Random dataset (results with the other tested datasets are similar) with FULL replication, for a total of 100, 200, 400, and 800 queries. As we can see, WORK-STEAL scales almost perfectly with the increasing number of queries, since the time to execute 100 queries in 1 node is the same with the number to execute 200 queries in 2 nodes, which is the same with the number to execute 400 queries in 4 nodes, which is the same with the number to execute 800 queries in 5 nodes. We have observed exactly the same trend for the PARTIAL scheduling algorithms (Figure 9c). Note that PARTIAL replication can be applied only with two or more nodes. Moreover, we observe that all Odyssey's queries algorithms achieve good scalability as the number of nodes increases. This is better illustrated in Figure 9d, which presents the WORK-STEAL throughput on the Random dataset.

Replication. We study now Odyssy's different replication strategies using the Seismic dataset and WORK-STEAL-PREDICT that is our best scheduling algorithm, to avoid any overhead incurred by load-imbalances between nodes. Specifically, we test EQUALLY-SPLIT, PARTIAL-4, PARTIAL-2 and FULL, replication strategies, for varying number of queries. Figures 10a-10d [[[Lef: change labels of Figure 10b]]] present the query answering time[2], where we observe that the more a dataset is replicated, the less time is required to answer queries, and this is consistent for all

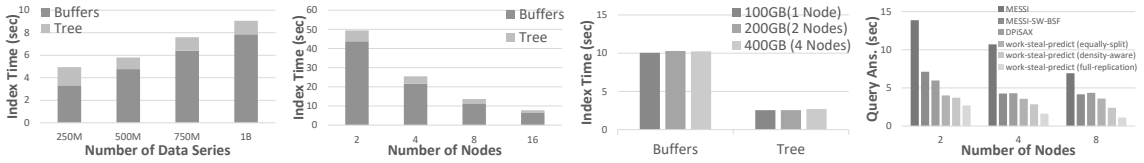
[2]We report results with 16 nodes only for the small workload, because the scheduler of our cluster does not allow long-running jobs on more than 8 nodes.



(a) 100 Queries (b) 200 Queries (c) 400 Queries (d) 800 Queries

(e) 100 Queries (f) 200 Queries (g) 400 Queries (h) 800 Queries

Figure 10: Comparison of Odyssey's replication strategies, using WORK-STEAL-PREDICT with Seismic.



(a) (b) (c) (d)

Figure 11: (a) Index scalability on Deep using EQUALLY-SPLIT, as the dataset size increases, with 16 nodes. (b) Index scalability on Deep using EQUALLY-SPLIT, as the number of nodes increases. (c) Index scalability on the Random dataset as both the dataset size and the number of nodes increase linearly, using EQUALLY-SPLIT. (d) Comparison of WORK-STEAL-PREDICT with Odyssey's different data partitioning schemes, against other implementations, using Seismic.

number of queries. So, the FULL replication strategy has the smaller queries answering time. On the other hand, Figures 10e-10h present the total execution time, which additionally to query answering time, it also includes the time for tree index construction. Interestingly, for small query numbers (100 and 200), we observe exactly the opposite: a larger amount of dataset replication, results in bigger total execution time, with FULL having now the bigger tree index construction time. This happens because the increased tree index construction time dominates in the total time. However, as the number of queries increases, the differences between the total execution time of algorithms become smaller. Remarkably, for large enough number of queries (e.g. 800), the increased tree index construction cost is amortized by the smaller query answering time, having FULL replication strategy performing better than EQUALLY-SPLIT. This analysis revealed an interesting trade-off (regarding the level of replication) between the query answering cost and the tree index construction cost, while the latter can be amortized using a large enough set of queries.

Index Scalability. Figures 11a and 11b illustrate the index creation time of Odyssey for our 1B series Deep dataset using EQUALLY-SPLIT, as the dataset size increases on a system with 16 nodes and as the number of nodes increases (while using the full size datasets), respectively. In both cases, we observe optimal speedup regarding index creation. Additionally, Figure 11c presents the scalability of Odyssey on the Random dataset as both the dataset size and the number of nodes increase linearly, again using EQUALLY-SPLIT. As shown, Odyssey achieves perfect scalability since the corresponding buffer times and index times remain almost constant.

Data partitioning and comparison to competitors. Figure 11d presents (i) a comparison of WORK-STEAL-PREDICT Odyssey's best performing algorithm, against DMESSI, DMESSI-SW-BSF, and DPISAX; and (ii) the performance of Odyssey's different data partitioning schemes, i.e., EQUALLY-SPLIT and DENSITY-AWARE, as well as the FULL replication strategy, using Seismic.

Interestingly, DMESSI performs significantly worse that all the other implementations, showing that by simply executing multiple instances of a state-of-the-art single-node algorithm like MESSI on a multi-node system (in order to scale its applicability on larger dataset sizes) does not perform well on real datasets; thus, more sophisticated approaches are required. On the other hand, Odyssey's WORK-STEAL-PREDICT with FULL replication strategy is significantly better than all its competitors. Specifically, it is up to 6.6x, 3.7x, and 3.8x faster than DMESSI, DMESSI-SW-BSF, and DPISAX, respectively. Moreover, regarding Odyssey's data partitioning techniques, Figure 11d shows that WORK-STEAL-PREDICT with the DENSITY-AWARE partitioning performs better than EQUALLY-SPLIT.

5 Conclusions

In this work, we presented Odyssey, a novel *distributed* data-series processing framework that takes advantage of the full computational capacity of modern clusters comprised of multi-core servers. Odyssey addresses a number of challenges in designing an efficient and highly-scalable *distributed* data series index, including efficient scheduling, load-balancing, and flexible partial replication, and successfully navigates the fundamental trade-off between data scalability and good performance during query answering. Our experimental evaluation on several real and synthetic datasets demontrates that Odyssey achieves its challenging goals.

Acknowledgments: This work has been done while P. Fatourou was working at the LIPADE, Université Paris Cité, as an MSCA Individual Fellow in the context of the PLATON project (MSCA grant agreement #101031688).

References

[1] Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[2] Incorporated Research Institutions for Seismology – Seismic Data Access. http://ds.iris.edu/data/access/, 2016.

[3] Odyssey code and datasets. `https://helios2.mi.parisdescartes.fr/~themisp/odyssey/`, 2022.

[4] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In D. B. Lomet, editor, *FODO*, 1993.

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

[6] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, mar 1999.

[7] G. E. Blelloch, P. B. Gibbons, Y. Matias, and G. J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97. Association for Computing Machinery, 1997.

[8] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[9] P. Boniol, M. Linardi, F. Roncallo, and T. Palpanas. Automated Anomaly Detection in Large Sequences. In *ICDE*, 2020.

[10] P. Boniol and T. Palpanas. Series2Graph: Graph-based Subsequence Anomaly Detection for Time Series. *PVLDB*, 2020.

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

[12] Crate. CrateDB: Real-time SQL Database for Machine Data & IoT, 2018.

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

[14] K. Echihabi, K. Zoumpatianos, and T. Palpanas. Big sequence management: Scaling up and out. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, pages 714–717. OpenProceedings.org, 2021.

[15] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB*, 2018.

[16] K. Echihabi, K. Zoumpatianos, T. Palpanas, and H. Benbrahim. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *PVLDB*, 2019.

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

[18] P. Fatourou. Low-contention depth-first scheduling of parallel computations with write-once synchronization variables. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, New York, NY, USA, 2001. Association for Computing Machinery.

[19] P. Fatourou and P. Spirakis. A new scheduling algorithm for general strict multithreaded computations. In P. Jayanti, editor, *Distributed Computing*, pages 297–311, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[20] P. Fatourou and P. Spirakis. Efficient scheduling of strict multithreaded computations. *Theory of Computing Systems*, 33:173–232, 2000.

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

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

[23] M. Gardner. *Knotted Doughnuts and Other Mathematical Entertainments*. W. H. Freeman, 1986.

[24] A. Gogolou, T. Tsandilas, K. Echihabi, A. Bezerianos, and T. Palpanas. Data series progressive similarity search with probabilistic quality guarantees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020.

[25] A. Gogolou, T. Tsandilas, T. Palpanas, and A. Bezerianos. Progressive similarity search on time series data. In *Proceedings of the Workshops of the EDBT/ICDT 2019 Joint Conference, EDBT/ICDT 2019, Lisbon, Portugal, March 26, 2019*, volume 2322 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.

[26] Hellium. Hellium: Ultra high performance key/value storage, 2018.

[27] P. Huijse, P. A. Estevez, P. Protopapas, J. C. Principe, and P. Zegers. Computational intelligence challenges and applications on large-scale astronomical time series databases. *CIM*, 2014.

[28] InfluxDB. InfluxDB - Open Source Time Series, Metrics, and Analytics Database (http://influxdb.com/), 2018.

[29] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011, May 22-27, 2011, Prague Congress Center, Prague, Czech Republic*, pages 861–864. IEEE, 2011.

[30] S. K. Jensen, T. B. Pedersen, and C. Thomsen. Time series management systems: A survey. *IEEE Trans. Knowl. Data Eng.*, 29(11):2581–2600, 2017.

[31] K. Kashino, G. Smith, and H. Murase. Time-series active search for quick retrieval of audio and video. In *ICASSP*, 1999.

[32] E. J. Keogh, K. Chakrabarti, S. Mehrotra, and M. J. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In S. Mehrotra and T. K. Sellis, editors, *SIGMOD*, 2001.

[33] M. Linardi, Y. Zhu, T. Palpanas, and E. J. Keogh. Matrix Profile Goes MAD: Variable-Length Motif And Discord Discovery in Data Series. In *DAMI*, 2020.

[34] OpenTSDB. OpenTSDB - A Distributed, Scalable Monitoring System (http://opentsdb.net/), 2015.

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

[36] T. Palpanas. The parallel and distributed future of data series mining. In *HPCS*, 2017.

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

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

[39] T. Pelkonen, S. Franklin, P. Cavallaro, Q. Huang, J. Meza, J. Teller, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *VLDB*, 2015.

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

[41] B. Peng, P. Fatourou, and T. Palpanas. SING: Sequence Indexing Using GPUs. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2021.

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

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

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

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

[46] Prometheus. Prometheus – Monitoring system & time series database, 2018.

[47] QuasarDB. QuasarDB: high-performance, distributed, time series database, 2018.

[48] D. Rafiei and A. O. Mendelzon. Efficient retrieval of similar time sequences using DFT. In K. Tanaka and S. Ghandeharizadeh, editors, *FODO*, 1998.

[49] U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco. Practical data prediction for real-world wireless sensor networks. *TKDE*, 2015.

[50] RiakTS. Riak TS – Basho Technologies, 2018.

[51] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, 1987.

[52] J. Shieh and E. Keogh. Isax: Indexing and mining terabyte sized time series. 2008.

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

[54] Timely. Timely – A secure time series database based on Accumulo and Grafana, 2018.

[55] Timescale. Timescale - an open source time series management system, 2018.

[56] S. C. Vision. Deep billion-scale indexing. `http://sites.skoltech.ru/compvision/noimi`, 2018.

[57] C. Wang, X. Huang, J. Qiao, T. Jiang, L. Rui, J. Zhang, R. Kang, J. Feinauer, K. Mcgrail, P. Wang, D. Luo, J. Yuan, J. Wang, and J. Sun. Apache iotdb: Time-series database for internet of things. *Proc. VLDB Endow.*, 13(12):2901–2904, 2020.

[58] Y. Wang, P. Wang, J. Pei, W. Wang, and S. Huang. A data-adaptive and dynamic segmentation index for whole matching on time series. *PVLDB*, 6(10), 2013.

[59] Warp10. Warp 10 – The Most Advanced Time Series Platform., 2018.

[60] J. Wu, P. Wang, N. Pan, C. Wang, W. Wang, and J. Wang. Kv-match: A subsequence matching approach supporting normalization and time warping. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 866–877. IEEE, 2019.

[61] D. E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. Dpisax: Massively distributed partitioned isax. pages 1135–1140, 2017.

[62] D.-E. Yagoubi, R. Akbarinia, F. Masseglia, and T. Palpanas. Massively distributed time series indexing and querying. *TKDE (to appear)*, 2018.

[63] L. Ye and E. Keogh. Time series shapelets: a new primitive for data mining. In *SIGKDD*. ACM, 2009.

[64] L. Zhang, N. Alghamdi, M. Y. Eltabakh, and E. A. Rundensteiner. TARDIS: distributed indexing framework for big time series data. In *ICDE*, 2019.

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

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

[67] K. Zoumpatianos and T. Palpanas. Data series management: Fulfilling the need for big sequence analytics. In *ICDE*, 2018.