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

Manos Chatzakis
EPFL
emmanouil.chatzakis@epfl.ch

Panagiota Fatourou
FORTH, ICS & University of Crete, CSD
faturu@ics.forth.gr

Eleftherios Kosmas
FORTH, ICS & Hellenic Mediterranean University & University of Crete, CSD
ekosmas@csd.uoc.gr

Themis Palpanas
Université Paris Cité & 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 distributed systems 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.

PVLDB Reference Format:
Manos Chatzakis, Panagiota Fatourou, Eleftherios Kosmas, Themis Palpanas, and Botao Peng. Odyssey: A Journey in the Land of Distributed Data Series Similarity Search. PVLDB, 16(5): 1140 - 1153, 2023.
doi:10.14778/3579075.3579087

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://helios2.mi.parisdescartes.fr/~themisp/odyssey/.

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 [44, 47, 81]. 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 [11, 12, 32, 36, 41, 57, 68, 75]. This analysis usually encompasses answering *similarity search* queries [20, 21, 44], which are useful in a variety of downstream analysis tasks [16, 18, 19]. Moreover, several applications across domains are very sensitive to the accuracy of the results [7, 47], and thus, require exact query answering [20], which is our focus.

As the size of the data series collections grows larger [44, 45, 47], recently proposed State-of-the-Art (SotA) data series indexes exploit parallelism through the use of multiple threads and the utilization of the SIMD capabilities of modern hardware [15, 51, 53]. However, the unprecedented growth in size that data series collections experience nowadays, renders even SotA parallel data series indexes inadequate [7, 17, 20, 21, 30, 45, 47], mainly due to the large number of random disk page reads required for exact query answering [20]. To address these issues, fast in-memory solutions have been proposed [49, 50, 52]. However, these solutions do not take advantage of distributed systems, and hence, are limited by the amount of memory of a single machine. This is the limitation we address, thus allowing the above SotA solutions to handle datasets that far exceed the main memory capacity of any single node.

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 leaf of the index. Previous works [20, 30] though, have shown that pruning is not very effective, especially for some hard datasets.

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

Proceedings of the VLDB Endowment, Vol. 16, No. 5 ISSN 2150-8097.
doi:10.14778/3579075.3579087

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 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 (LB) even in settings where predictions may not be accurate, Odyssey provides a LB mechanism, which 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 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 performance during query answering, because no data replication means that Odyssey's LB mechanism cannot be used.

Odyssey manages to effectively unify these two contradicting goals by supporting a flexible *partial replication scheme*. 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, users can choose the time-space trade-off that best suits their application and setting. Experiments show that Odyssey achieves good performance even for small replication degrees.

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 SotA parallel indexes [49, 50, 52], was one more challenging task we undertook while designing Odyssey. A simple approach of using an instance of the SotA MESSI index [49] in each node did not result in good performance mainly due to two reasons. First, different data series queries may exhibit variable degrees of locality (revealed only at runtime), resulting in low pruning in some of the nodes, and thus, in severe load balancing problems and performance degradation. Second, supporting load-balancing on top of such a simple approach would require moving data around, which is often prohibitively expensive. Odyssey *single-node* indexing scheme borrows some techniques from SotA indexes [49–53], and couples those 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.

Odyssey is innovative in different ways. First, it employs a different pattern of parallelism from all existing approaches in traversing 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, and Odyssey proposes an effective mechanism for predicting a good threshold. Additionally, Odyssey provides efficient communication and book-keeping mechanisms, to enable fast exchange of information among nodes to ensure good pruning degrees in all of them.

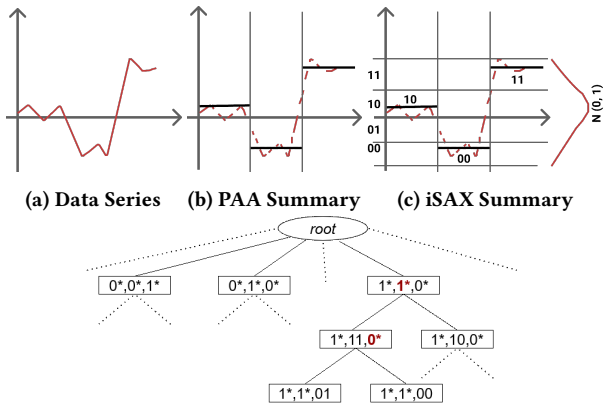
Odyssey is up to 6.6x faster than its competitors and more than 3.5x better than its best competitor. Additionally, Odyssey's index creation perfectly scales with both the dataset size and the number of node. Moreover, Odyssey's best performing scheduling strategy is more than 2.5x faster than its initial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

Contributions. The main contributions of the paper 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 customized 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 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 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 [20]; for simplicity, we omit them, as well.



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

(d) iSAX Tree

Figure 1: From data series to iSAX index

iSAX Summary. The *iSAX summary* [60] 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 index tree (*iSAX-based index tree* [46]; see Figure 1).

Similarity Search. Given a collection of data series C and an input data series S, called the *query*, *similarity search* is the task of finding the data series in 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.

Single-Node Parallel Summary-Based DS Indexing. Such indexes [6, 15, 49–53] 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 *index tree construction* and the *query answering* phases. In the index tree 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 index tree that they construct. These design decisions allow them to build the index tree 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

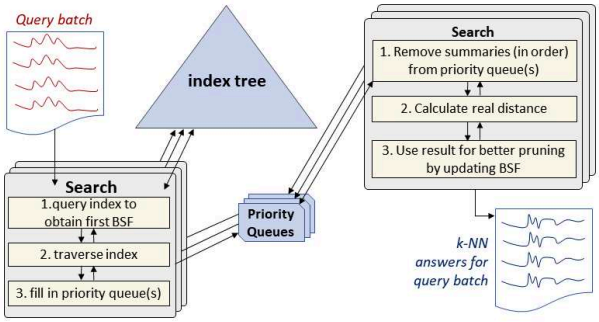


Figure 2: Algorithm Outline of Parallel DS Indexes.

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 [49, 50, 52]. Multiple threads process, concurrently, the elements of the priority queues, calculating real distances (if needed), and updating the BSF each time a new minimum is met (see Figure 2). Once this process completes, the distance to 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, based on the R-Tree [59], and was later optimized [56]. Various indices, specific to data sequences, have been proposed in the literature [19]. DSTree [69] is an index based on the APCA summarization [38]. The DSTree can adaptively perform split operations by increasing the detail of APCA as needed. The iSAX index is based on the SAX summarization, and its extension, iSAX [60]. In this case, the data series summarization is bitwise, leading to a concise representation and overall index. Several other iSAX-based indices have been proposed in the literature [13, 39, 40, 46, 67, 70, 77, 78]. These indexes are among the SotA solutions in this area [20], including MESSI [49, 52], 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 [19, 35].Beringei [48] has a custom in-memory storage engine. It compresses and organizes data in a series per series scheme. CrateDB [14] partitions data in chunks, stores them in a distributed file system, and indexes them using Apache Lucene. InfluxDB [33] uses Time-Structured Merge Trees (LSM tree variant). Prometheus [54] is based on the Beringei ideas. QuasarDB [55] utilizes either RocksDB or Hellium [31]. Riak TS [58] supports both LevelDB or Bitcask, which is a custom log structured hash table. Timescale [64] is a Postgres extension. IoTDB [66] is geared towards streaming data series. Finally, various systems such as OpenTSDB [43], Timely [63] (concentrated on security) and Warp10 [71] 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 [72] and its improvement, L-Match [26], 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 subsequence similarity search, and not whole-matching [20], which is the focus of our paper. TARDIS [76] 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 than the exact queries we consider in our work, and cannot be efficiently transformed to exact querying. Finally, DPiSAX [73, 74] is a distributed solution for data series similarity search, developed for Apache Spark using Scala. It was designed for answering batches of approximate search queries, but also supports exact search. 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. Note that DPiSAX was not explicitly designed for intra-node parallelization, but is the only distributed data series index in the literature that supports exact search.

Work-stealing was employed in the Cilk framework [1]. The work-stealing approach was formally studied and analyzed in [10, 24, 25]. Lots of work has been done on this topic (e.g., [8, 9, 23]).

3 THE ODYSSEY FRAMEWORK

We start with a high level overview of the Odyssey flowchart, which comprises of five stages (see Figure 3).

In the first stage, a *coordinator node* partitions the raw data-series collection to as many chunks as the number of system nodes, and assigns a chunk to each node (including itself). (Section 3.4 details Odyssey's partitioning schemes.) In the second stage, each node (i) loads its chunk of data in memory, (ii) computes their iSAX summaries and stores them into a number of *summarization buffers*, for achieving locality, and (iii) builds its *index tree*. To enhance performance at query answering, Odyssey employs data replication. It forms groups of nodes (*replication groups*, described in Section 3.3), where all nodes of each group store the same chunk of data. Each replication group has a coordinator node, called *group coordinator*, which schedules queries to the group's nodes. A batch of queries (e.g., originating from a *k-NN* classification task) to execute is submitted to all group coordinators (as different groups store different data chunks). In the third stage, the group coordinators start by estimating the execution time of each query, then sort queries in descending order of estimated execution times, and *dynamically* schedule them to the group's nodes (Section 3.1 describes query scheduling). In the fourth stage, each node processes the queries assigned to it. It first calculates an initial BSF, and then prunes the index tree using this BSF, populating the priority queues with leaves that cannot be pruned. Finally, it processes the elements of the priority queues to find the best local answer (corresponding to its data chunk). In this stage, Odyssey supports *BSF-sharing* and *work-stealing* (detailed in Section 3.2). In the last stage, the coordinator node collects the local answers from the group coordinators, and produces the final answers.

3.1 Query Scheduling

To correctly answer a query, it should be forwarded to at least one set of system nodes that collectively store all the data. We call such sets node *clusters* in Section 3.3. Thus, in the no-replication case this set contains all system nodes, so a scheduling algorithm should forward all queries to all nodes. Other replication settings (and especially full replication) are more interesting, as they enable the utilization of different scheduling techniques.

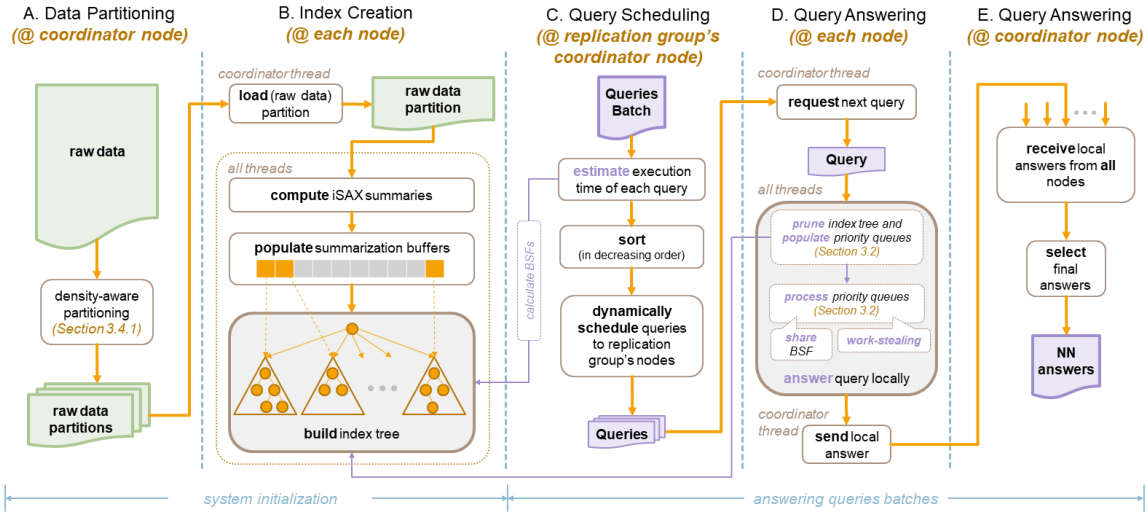


Figure 3: Odyssey flowchart.

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 imbalance 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, while the last nodes will get more work to do. The dynamic method (DQS) 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 [17, 29] illustrated that there exists a correlation between the initial BSF and the number of vertices visited in a single-node index tree. We performed a corresponding query analysis which showed that similarity search queries, for which the *initial BSF* is high, tend to also have high execution times. In this work, we use a linear regression model (other pediction schemes can be used, as well) to produce estimates for each query. An example of this outcome is shown in Figure 4 (for Seismic; we follow the same process for the other datasets).

These observations led us to design two scheduling algorithms. The first, *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 (*unsorted*) schedules the queries using their order in the sequence, and the second (*sorted*) sorts the sequence based on decreasing execution time estimations. . The second scheduling algorithm, called *dynamic prediction-based scheduling*, is an enhanced version of DQS, where queries are assigned to nodes after sorting the entire query batch, based on estimations (in decreasing order).

Figure 4: Linear regression for Seismic queries prediction.

Example 3.1. Consider a system of two nodes, sn_1 and sn_2, and let $Q = \{q_1, q_2, q_3, q_4, q_5\}$ be a query batch to execute. Assume that $\mathcal{ES} = \{100, 50, 200, 250, 80\}$ is the set of the estimated execution times, where the i-th element of $\mathcal{ES}$ is the estimated execution time for q_i, $1 \leq i \leq 5$. Unsorted static prediction-based scheduling, with load variables l_1 and l_2 (for sn_1 and sn_2, respectively), proceeds as follows: q_1 is assigned to sn_1 (so, $l_1 = 100$), and q_2 is assigned to sn_2 (so, $l_2 = 50$). Since $l_2 < l_1$, q_3 is assigned to sn_2 (thus, $l_2 = 250$). Following a similar strategy, q_4 is assigned to sn_1, and q_5 is assigned to sn_2. So, sn_1 receives $\{q_1, q_4\}$ and sn_2 receives $\{q_2, q_3, q_5\}$. In sorted static prediction-based scheduling, the queries of Q are first sorted in decreasing order of their estimated times, resulting in $Q' = \{q_4, q_3, q_1, q_5, q_2\}$ (which corresponds to $\mathcal{ES}' = \{250, 200, 100, 80, 50\}$). After applying the static prediction-based scheduling algorithm (as above) on these sets, $\{q_4, q_5\}$ is assigned to sn_1 and $\{q_3, q_1, q_2\}$ is assigned to sn_2. Finally, dynamic prediction-based scheduling also sorts the queries of Q. In this case, q_4 is assigned to sn_1, q_3 to sn_2, while the rest of the queries are dynamically assigned to nodes (in order) upon request (thus, based on actual execution times).

The Odyssey framework supports all of the above scheduling algorithms. 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. Specifically, idle nodes 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. It is also necessary for achieving high scalability. As the number of utilized nodes increases, the number of batch queries that each node has to process becomes smaller and smaller. Thus, problematic scenarios 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.

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 queues' processing. We thus focus on designing a method that allows 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; partial replication is then discussed in Section 3.3.

A simple work-stealing scheme [1, 10] 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 take work away from one node and assign it to another without ever moving any data around.

Odyssey's 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.

Note that the approaches followed by existing SotA indexes [49, 50, 52] for creating and processing the priority queues are too naive to support work-stealing without moving any data around. In Odyssey, we propose (in Section 3.2.1) 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.

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 5. 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). For each such query Q, it creates a number of search workers to execute it (line 8). As soon as, all queries in sn's query batch have been processed, sn informs other nodes that it has completed (line 12). Then, it tries to help other active nodes by executing PerformWorkStealing (line 13). Each node allocates a thread to play the role of the work-stealing manager (line 6). This thread simply processes all work-stealing

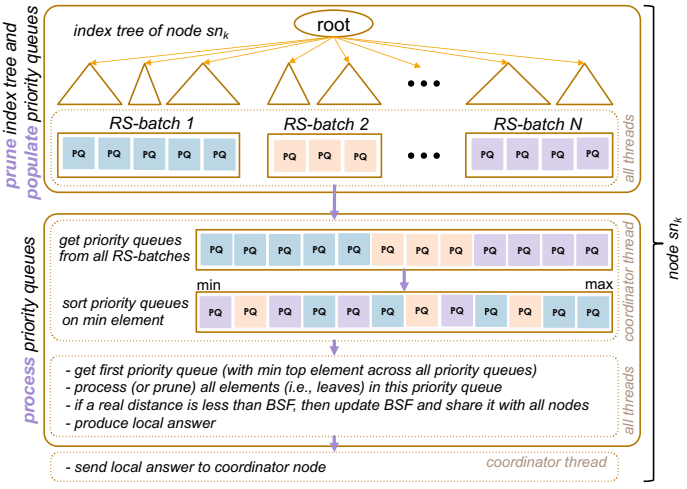


Figure 5: Outline of the Odyssey single-node query-answering process.

requests that the node will receive (Algorithm 3). (Work-stealing is discussed in Section 3.2.2.)

The query answering algorithm in sn splits the tree into *root subtree (RS) batches*, i.e., sets of consecutive root subtrees (see Figure 5), and allocates a number of threads to work on them. Each thread begins by getting an RS-batch to work on using Fetch&Add (Algorithm 2). Then, the thread executes the *ProcessBatch* routine, 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. 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.

As soon as an idle thread th discovers that all RS-batches have been assigned for processing, it tries to help some other still active thread, th', to complete processing its assigned RS-batch (lines 11-14, Algorithm 2). To reduce the synchronization cost, there is a threshold, *HelpTH*, on the number of threads that help on each RS-batch (line 12). This phase ends when the subtrees of all RS-batches have been traversed and all priority queues have been populated. Experiments showed that we get the best performance when the number of RS-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 15-21). 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 23-29). Every thread gets a priority queue from the PQueues array to process (using Fetch&Add). Routine *ProcessPriorityQueue* processes those data series 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. This improved BSF is submitted to all nodes of the system.

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

Shared Variables: Shared PointerToArray $PQueues = NULL$
Input: QuerySeriesBatch $QBatch$, **Index** $Index$, **Integer** $NThreads$
Array $BSFArray[\,]$ ▷ with size $|QBatch|$
for *every query series id* Q *in* $QBatch$ **do**
  $iSAX_Q$ = calculate iSAX summary for Q
  BSF = approxSearch($iSAX_Q$, $Index$)
  create a thread to execute an instance of WorkStealingManager(Q)
  for $i \leftarrow 0$ **to** $NThreads - 1$ **do**
    create a thread to execute an instance of SearchWorker(Q, $Index$, N_{sb}, i, $PQueues$)
  Wait for all threads to finish
  $FinishFlag[Q]$:= TRUE
  BSFArray[Q] := BSF;
send(DONE, sn) to all nodes
PerformWorkStealing()
return ($BSFArray$)

Algorithm 2: SearchWorker - Code for thread tid

▷ Shared Variables
Shared Integers $BCnt$ = 0, $PQCnt$ = 0, $TotPQ$ = 0;

Input: QuerySeries Q, **Index** $Index$, **Integer** N_{sb}, **Integer** tid, **Queue** $PQueues[\,]$

Integer $bindex$, $pqindex$;
▷ Tree Traversal Phase
while ($TRUE$) **do**
  $bindex \leftarrow$ Fetch&Add($BCnt$,1);
  if $bindex \geq N_{sb}$ **then**
    break;
  $ProcessRSBatch(Q, bindex, Index.RSBatches)$;
  $Index.RSBatches[bindex].complete \leftarrow TRUE$;
for $bindex \leftarrow 0$ ***to*** N_{sb} **do**
  if !$Index.RSBatches[bindex].complete$ ***AND*** *Fetch&Add*($Index.RSBatches[bindex].helped$, 1) $< HelpTH$ **then**
    $ProcessBatch(Q, bindex, Index.RSBatches)$;
    $Index.RSBatches[bindex].complete \leftarrow TRUE$;
Barrier for all threads;
▷ Priority Queue Preprocessing Phase
if $tid == 0$ **then**
  Traverse all RS-batches and put their priority queues into $PQueues[\,]$;
  $SortByRootPriority(PQueues)$;
  $TotPQ \leftarrow$ number of valid elements of $PQueues$;
Barrier for all threads;
▷ Priority Queue Processing Phase
while ($TRUE$) **do**
  $pqindex \leftarrow$ Fetch&Add($PQCnt$, 1);
  if $pqindex \geq TotPQ$ **then**
    break;
  if $PQueues[pqindex].stolen$ **then**
    continue;
  $ProcessPriorityQueue(PQueues[pqindex])$;

Finally, all answers are transmitted to the coordinator node, and the globally smallest value of the BSF is the response to the query.

Size of Priority Queues. The size of each priority queue cannot be larger than a specific threshold, 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.

Algorithm 3: WorkstealingManager - Code for node sn

Input: Integer N_B

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

▷ Always-enabled event: it is executed repeatedly
Upon receiving no message:
if $FinishFlag[Q]$ *in* sn *is set* **then**
  Terminate

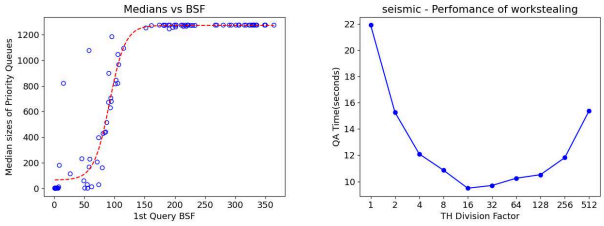
Algorithm 4: PerformWorkStealing - Code for node sn

Input: Index $index$, **Function** $exact\_search\_workstealing\_func$, **QuerySeries** $queries[\,]$, **Integer** $total\_nodes\_per\_nodegroup$

Upon Receiving a DONE message from node sn'**:**
add sn' in set $DoneNds$
if $DoneNds$ *contains all system's nodes* **then**
  Terminate

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

▷ Always-enabled event: it is executed repeatedly
Upon receiving no message:
if !*(ResponseFlag)* **then**
  sn' := choose randomly a node not in $DoneNds$
  send(StealingRequest, sn) to sn'
  ResponseFlag := 1



(a) Sigmoid function fitting for determining TH. **(b) Performance for different Threshold division factors.**

Figure 6: Odyssey Single-Node Query-Answering Algorithm Configuration.

Choosing the appropriate value for the threshold, 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 $M \in [0, 1], m \leq M, b, c \in \mathbf{R}^*$, and $d \in \mathbf{R}$ are the parameters of the sigmoid function (Figure 6a). The final threshold value for each query 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 6b).

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 15-17). It randomly chooses a system node sn' from the set of those nodes that *sn* knows to be still active and sends a steal request to it[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 2-4 of Algorithm 3).

Earlier work has demonstrated that a large amount of the query answering execution time is devoted to verifying that there is no better answer after the correct answer has been processed [17, 29, 30]. Based on these findings, Odyssey's 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 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 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 show that fixing N_{send} to 4 was the best choice (so $N_{send} = 4$ in Odyssey).

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*.

[1]The codes for Algorithms 3 and 4 are written in an event-driven style [5, 42]

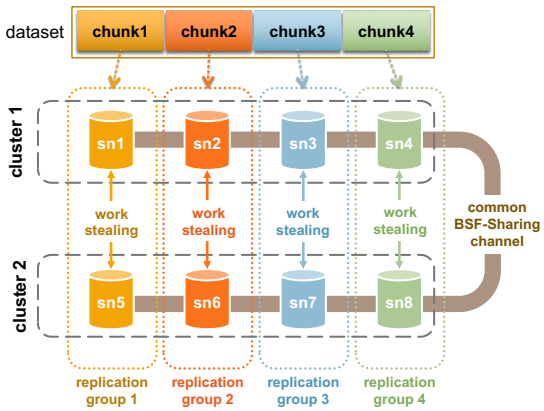


Figure 7: Data replication PARTIAL-4 **($N_{sn} = 8$, data size 80GB).**

The idea is to split the set of system nodes into *clusters*, where each cluster collectively stores the entire dataset (see Figure 7). Each cluster node stores (and indexes) a chunk of the dataset. The chunks stored in each node of a cluster are mutually disjoint. A *replication group* is a group of nodes such that each node stores the same dataset as every other node in the group. (We experimented with replication groups of the same size, but Odyssey can operate with replication groups of different sizes, as well.) The nodes of a replication group build their iSAX indices from the same data chunk. Thus, inside every replication group, we can apply the scheduling and load-balancing schemes described in Sections 3.1 and 3.2, respectively. We call the number of clusters the *replication degree* of the system.

Consider a system with N_{sn} system nodes. We call PARTIAL-k, $k \in \{1, 2, 4, \ldots, N_{sn}\}$, a replication setting with k replication groups and N_{sn}/k clusters. Observe that PARTIAL-N_{sn}, or EQUALLY-SPLIT corresponds to no replication (each node stores a disjoint chunk of the dataset), and PARTIAL-1, or FULL corresponds to full replication (each node stores the full dataset). Note that Odyssey's data replication scheme supports $1 + \log N_{sn}$ different *replication degrees*. Smaller replication degrees lead to smaller space overheads (and thus better data scalability). Thus, Odyssey's data replication scheme allows us to tackle memory limitation problems. Moreover, more replication groups lead to scalability in index creation.

Example 3.2. A system with 8 nodes supports $1 + \log 8 = 4$ different replication degrees: FULL (PARTIAL-1), PARTIAL-2, PARTIAL-4, and EQUALLY-SPLIT (PARTIAL-8). Figure 7 illustrates the case of PARTIAL-4: we have 4 replication groups, organized in 2 clusters; replication degree is 2.

3.4 Data Partitioning

Odyssey framework supports more than one partitioning schemes. Under EQUALLY-SPLIT, 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. 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 (or no replication), each query is sent to every replication group. Each

node answers queries using its local data, and the partial answers for each query 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 7). 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's book-keeping method solves such synchronization problems. Each node holds an array that 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. Thus, each node has the best answers extracted from all 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 describe 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 [28] 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).

Example 3.3. Figure 8 shows 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: their iSAX representations (the iSAX word of the buffer) are very close to one another, e.g., node 1 stores buffers "000" and "100",

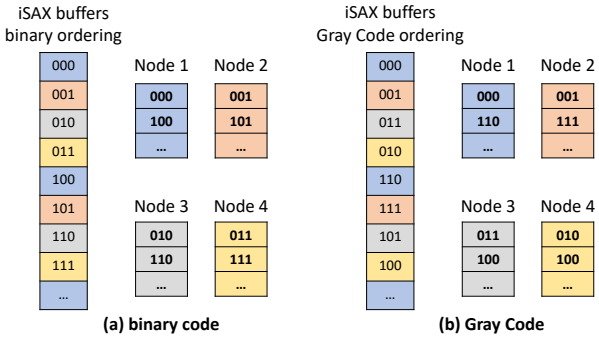


Figure 8: Examples of partitioning the iSAX buffers' data to 4 system nodes, based on (a) simple iSAX and (b) Gray Code.

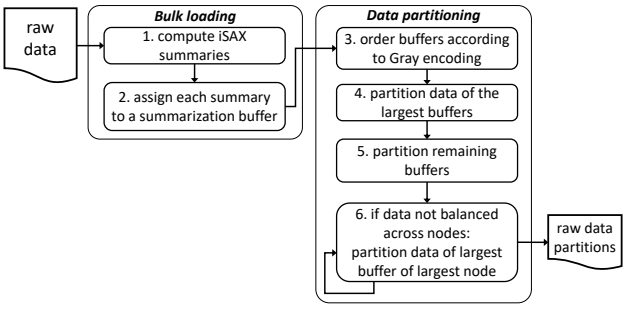


Figure 9: Flowchart of the DENSITY-AWARE data partitioning.

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 9. 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 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 for brevity) showed that DENSITY-AWARE exhibits a very stable behavior as we vary λ from a few hundred to several thousands. In this study, we use $\lambda = 400$.

4 EXTENSIONS

We now discuss two extensions of Odyssey, in order to support *k-NN* search and the Dynamic Time Warping (DTW) distance.

***k-NN* Search.** Extending Odyssey to support *k-NN* similarity search is straight-forward. Instead of computing a single BSF value, we simply need to keep track of the k smallest BSF values.

DTW Distance. We also extend Odyssey to perform similarity search using Dynamic Time Warping (DTW), which is an elastic

distance measure [37]. Note that no changes are required in the index structure for this: the index we build can answer both Euclidean and DTW similarity search queries. Supporting DTW queries requires modifying the query answering algorithm only, and using LB_Keogh [37], which is a tight lower bound of the DTW distance. We note that a lower bound for the DTW distance between the query and a candidate series can be computed by considering the distances between the corresponding points of the candidate series and the points of the LB_Keogh envelope of the query.

5 EXPERIMENTAL EVALUATION

Setup. Experiments conducted on a cluster of 16 SR645 nodes, connected through an HDR 100 Infiniband network. Each node has 128 cores (with no hyper-threading), 200GB RAM (available to users out of the 256GB physical memory), and runs Red Hat Enterprise Linux r8.2. All evaluated algorithms written in C and compiled using MPICC, Intel(R) MPI Library for Linux OS, v2021.2.

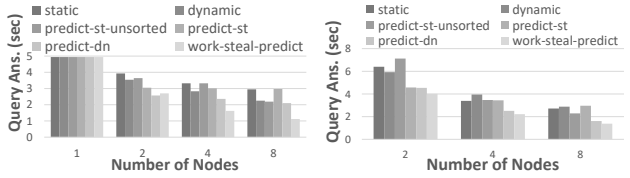
Algorithms. Our experimental analysis includes the entire range of Odyssey's data distribution strategies with k replication groups, PARTIAL-k, $k \in \{1, 2, 4, \ldots, N_{sn}\}$, as well as the density-aware data partitioning algorithm (DENSITY-AWARE). Recall that PARTIAL-N_{sn}, or EQUALLY-SPLIT corresponds to no replication, and PARTIAL-1, or FULL corresponds to full replication. Additionally, our analysis evaluates Odyssey's queries scheduling algorithms: (i) static scheduling assigning equally sized query sets to nodes (STATIC); (ii) dynamic scheduling using a coordinator (DYNAMIC); and (iii) predictions-based scheduling, including: static without ordering (PREDICT-ST-UNSORTED), static with ordering (PREDICT-ST), and dynamic (PREDICT-DN). Moreover, we evaluate Odyssey's work-stealing mechanism using both DYNAMIC and PREDICT-DN, resulting in algorithms WORK-STEAL and WORK-STEAL-PREDICT, respectively. The latter is our best scheduling algorithm (cf. paragraph "Queries scheduling"). We note that that Odyssey's query scheduling and work-stealing mechanisms can be used together only with the FULL or PARTIAL data distribution strategies that provide some replication.

We compare Odyssey to: (i) MESSI [49], 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 [74], where we implement (in C) the DPiSAX data partitioning strategy, and (for fair comparison) implement query answering in each node using MESSI.

Datasets. We evaluated Odyssey's strategies and algorithms using real and synthetic datasets, of varying sizes (refer to Table 1). 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 [13, 20–22, 79, 80], and models the distribution of stock market prices [22]. Our five real datasets come from the domains of seismology (*Seismic*), astronomy (*Astro*), deep learning (*Deep*), image processing (*Sift*), and information retrieval (*Yan-TtI*). *Seismic* contains seismic instrument recordings and consists of 100M data series of size 256 [27]. *Astro* represents celestial objects and consists of 100M data series of size 256 [62]. *Deep* [65] contains 1B Deep vectors of size 96 extracted from the last layers of a convolutional neural network. *Sift* [34] is comprised of image descriptors and Yandex Text-to-Image (*Yan-TtI*) [61] contains 1B vectors that include

Table 1: Details of datasets used in experiments.

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
Yan-TtI	1B	200	800	image and text
Random	100M-1600M	256	100-1600	random walks



(a) FULL replication **(b) PARTIAL-2 replication**

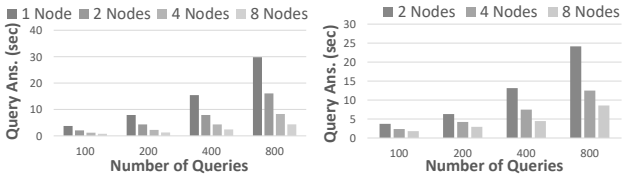
Figure 10: Odyssey's scheduling algorithms (Seismic).

image- and textual-embeddings in the same space; it represents typical cross-modal information retrieval tasks.

Evaluation Measures. During each experiment, E, and for each *node*, *sn*, we measure (i) the *buffer time* required to calculate the iSAX summaries and fill-in the receive buffers, (ii) the *tree time* required to insert the items of the receive buffers in the index tree, and (iii) the *query answering time* required to answer the queries assigned to *sn*. The sum of these times constitute the *total time* that *sn* works during E; also, buffer and tree times constitute the time required to create the index, called *index time*. To compute all the above times during E, we take the maximum among the corresponding times of each node participating in E. We report the average times of 10 experiments.

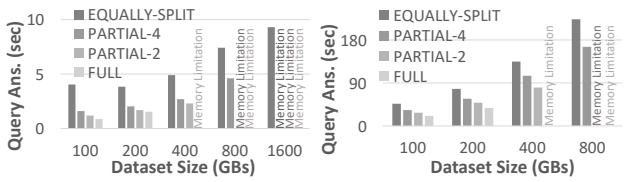
Query scheduling. To compare Odyssey's queries scheduling algorithms, the full replication strategy is selected, to avoid measuring any overheads resulting from the partial replicated strategies. Recall that scheduling algorithms can't be used together with the no replication strategies. We experimented 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. We present the results for Seismic, where the effort for answering queries varies. Specifically, Figure 10 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 10a shows that WORK-STEAL-PREDICT greatly outperforms (up to almost 2x) PREDICT-DN for large number of nodes when using FULL replication, i.e. our work-stealing technique positively affects performance on these cases. The same is true for PARTIAL-2 replication, but to a lesser extent. Recall (from Section 3.2.2) that this happens since all the algorithms that do not use the work-stealing technique 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



(a) FULL replication (b) PARTIAL-2 replication

Figure 11: Query answering scalability as the number of queries increase (Random).



(a) Random (b) Yan-TtI

Figure 12: Query time for 100 queries vs data size (8 nodes).

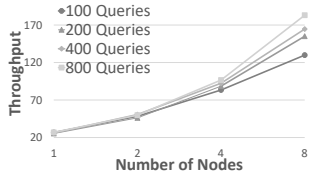


Figure 13: Query throughput (Random, FULL replication).

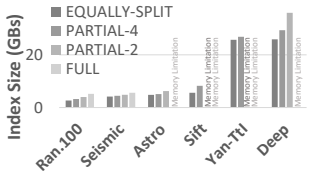
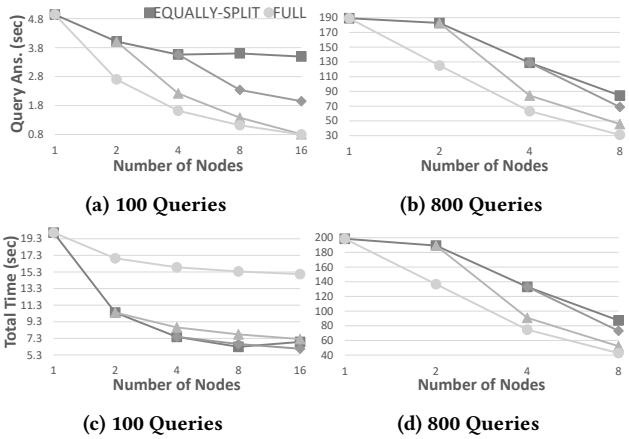


Figure 14: Index size.

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.

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 11a, 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 time to execute $j * 100$ queries in j nodes, $j \in \{2, 4, 8\}$. We have observed the same trend for the PARTIAL scheduling algorithms (Figure 11b). Note that PARTIAL replication can be applied only with two or more nodes. Additionally, we present in Figure 12 scalability experiments, by increasing the dataset size, for Random (between 100-1600GB) and Yan-TtI (between 100-800GB). We measure the total query answering time for 100 queries, when using 8 nodes. Note that we could not execute all replication strategies for all dataset sizes, due to the memory capacity of our nodes. The results show that query answering time scales gracefully as we increase the dataset size, while increasing the replication degree leads to better performance. Moreover, we observe that Odyssey's query answering algorithm achieves good scalability as the number of nodes increases. This is better illustrated in Figure 13, which presents the WORK-STEAL throughput on the Random dataset.



(a) 100 Queries (b) 800 Queries

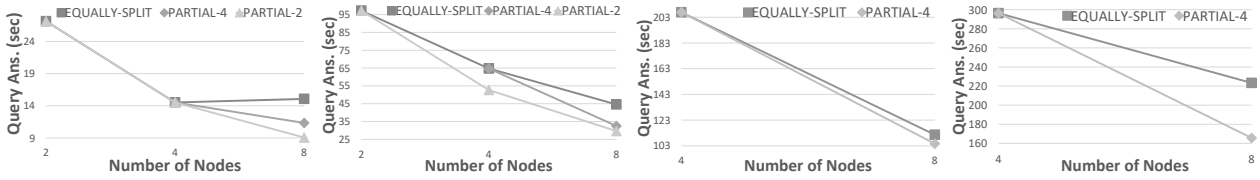
(c) 100 Queries (d) 800 Queries

Figure 15: Comparison of Odyssey's replication strategies, using WORK-STEAL-PREDICT with Seismic.

Replication. We study now Odyssey'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, for varying number of queries. Figures 15a-15b 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 number of queries. So, the FULL replication strategy has the smaller queries answering time. On the other hand, Figures 15c-15d present the total execution time, which includes also the time for index tree construction. Interestingly, for small query numbers (100), we observe exactly the opposite: a larger amount of data replication, results in bigger total time, with FULL having now the bigger index tree construction time. This happens because the increased index tree 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 index tree construction cost is amortized by the smaller query answering time, having FULL replication strategy performing better than EQUALLY-SPLIT. This analysis reveals an interesting trade-off (regarding the level of replication) between the query answering cost and the index tree construction cost, while the latter can be amortized using a large enough set of queries. Figure 16 shows the results of the query answering experiment with 100 queries for the rest of the real datasets. We observe similar trends to those of Seismic (Figure 15a). Overall, when query answering needs to be optimized, we recommend that Odyssey is used with the highest possible replication degree (given the dataset size and compute-cluster characteristics).

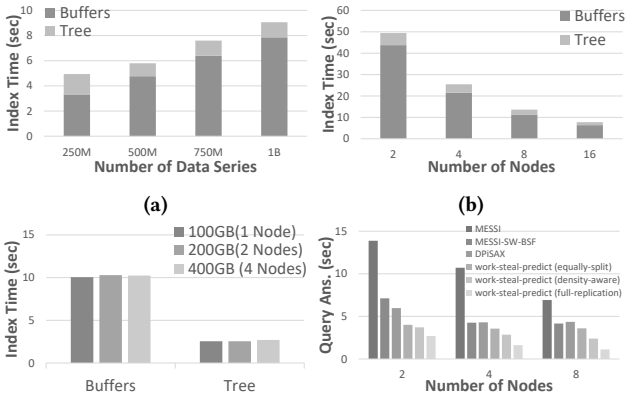
Index Scalability. We present in Figure 14 the total index size in GBs, for every replication strategy when using 8 nodes, for all real datasets we used and for Random 100GB (Ran.100). In all cases, the index size is very small compared to the size of the dataset. Figures 17a and 17b illustrate the index creation time of Odyssey for our 1B series Deep dataset using EQUALLY-SPLIT, as the dataset

[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) Astro (b) Deep (c) Sift (d) Yan-TtI

Figure 16: Comparison of Odyssey's replication strategies, using WORK-STEAL-PREDICT with real datasets, using 100 queries.



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

Figure 17: (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.

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 17c 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 17d 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 SotA 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

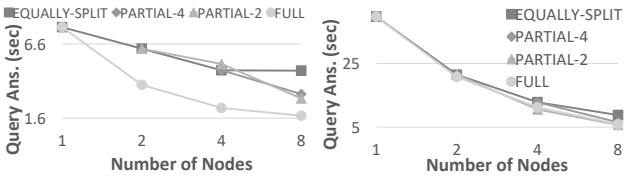


Figure 18: *10-NN* query answering (Random 100GB)

Figure 19: DTW with 5% warping (Random 100GB).

DMESSI, DMESSI-SW-BSF, and DPISAX, respectively. Moreover, regarding Odyssey's data partitioning techniques, Figure 17d shows that WORK-STEAL-PREDICT with the DENSITY-AWARE partitioning performs better than EQUALLY-SPLIT.

Extensions to *k-NN* and DTW. Finally, we present experiments with *k-NN* queries, and the DTW distance, where we measure the query answering time for 100 queries as we increase the number of nodes, when using different replication strategies. We evaluated all replication strategies when varying k between 1 and 20 for *k-NN*, and when varying the warping window size between 1%-15% of the series length for DTW. Figure 18 shows the *k-NN* results for $k = 10$, and Figure 19 shows the DTW results for 5% warping (results with the rest of parameter values are similar). As expected, query answering times are in both cases higher than before, while using more nodes and higher replication degrees improves performance in the same way we have observed in previous experiments. Results with Seismic exhibit similar trends and are omitted for brevity.

6 CONCLUSIONS

We presented Odyssey, a novel *distributed* data-series processing framework that takes advantage 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 trade-off between data scalability and good performance during query answering. In future work, we plan to extend Odyssey to support subsequence similarity search [40], as well as approximate similarity search.

ACKNOWLEDGMENTS

Work supported by NSFC Grant No. 62202450, EU Horizon 2020 Marie Sklodowska-Curie project PLATON No 101031688, and Hellenic Foundation for Research and Innovation (HFRI) project to support Faculty Members and Researchers No 3684. Numerical computations performed on the S-CAPAD/DANTE platform, IPGP, France. Work conducted while Manos Chatzakis was working for the University of Crete, ICS-FORTH and Université Paris Cité.

REFERENCES

[1] 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 55–69.

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

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

[4] Rakesh Agrawal, Christos Faloutsos, and Arun N. Swami. 1993. Efficient Similarity Search In Sequence Databases. In *FODO*, David B. Lomet (Ed.).

[5] Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley and Sons, Inc., Hoboken, NJ, USA.

[6] Ilias Azizi, Karima Echihabi, and Themis Palpana. 2023. Elpis: Graph-Based Similarity Search for Scalable Data Science. *PVLDB* (2023).

[7] Anthony J. Bagnall, Richard L. Cole, Themis Palpanas, and Konstantinos Zoumpatianos. 9(7), 2019. Data Series Management. *Dagstuhl Reports* (9(7), 2019).

[8] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. 1999. Provably Efficient Scheduling for Languages with Fine-Grained Parallelism. *J. ACM* 46, 2 (mar 1999), 281–321. https://doi.org/10.1145/301970.301974

[9] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. 1997. 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.

[10] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (sep 1999), 720–748. https://doi.org/10.1145/324133.324234

[11] Paul Boniol, Michele Linardi, Federico Roncallo, and Themis Palpanas. 2020. Automated Anomaly Detection in Large Sequences. In *ICDE*.

[12] Paul Boniol and Themis Palpanas. 2020. Series2Graph: Graph-based Subsequence Anomaly Detection for Time Series. *PVLDB* (2020).

[13] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn J. Keogh. 2014. Beyond one billion time series: indexing and mining very large time series collections with iSAX2+. *Knowl. Inf. Syst.* 39, 1 (2014), 123–151. http://dblp.uni-trier.de/db/journals/kais/kais39.html#CamerraSPRK14

[14] Crate. 2018. CrateDB: Real-time SQL Database for Machine Data & IoT. http://crate.io/

[15] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2022. Hercules Against Data Series Similarity Search. *PVLDB* (2022).

[16] Karima Echihabi, Themis Palpanas, and Kostas Zoumpatianos. 2021. New Trends in High-D Vector Similarity Search: AI-driven, Progressive, and Distributed. *Proc. VLDB Endow.* 14, 12 (2021), 3198–3201.

[17] Karima Echihabi, Theophanis Tsandilas, Anna Gogolou, Anastasia Bezerianos, and Themis Palpanas. 2023. ProS: Data Series Progressive k-NN Similarity Search and Classification with Probabilistic Quality Guarantees. *VLDBJ* (2023).

[18] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2020. Scalable Machine Learning on High-Dimensional Vectors: From Data Series to Deep Network Embeddings. In *WIMS: The 10th International Conference on Web Intelligence, Mining and Semantics*. ACM, 1–6.

[19] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2021. Big Sequence Management: Scaling up and Out. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT*. OpenProceedings.org, 714–717.

[20] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2018. The Lernaean Hydra of Data Series Similarity Search: An Experimental Evaluation of the State of the Art. *PVLDB* (2018).

[21] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. 2019. Return of the Lernaean Hydra: Experimental Evaluation of Data Series Approximate Similarity Search. *PVLDB* (2019).

[22] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. 1994. Fast subsequence matching in time-series databases. In *SIGMOD*. ACM, New York, NY, USA, 419–429. https://doi.org/10.1145/191839.191925

[23] Panagiota Fatourou. 2001. 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* (Crete Island, Greece) *(SPAA '01)*. Association for Computing Machinery, New York, NY, USA, 10. https://doi.org/10.1145/378580.378639

[24] Panagiota Fatourou and Paul Spirakis. 1999. A New Scheduling Algorithm for General Strict Multithreaded Computations. In *Distributed Computing*, Prasad Jayanti (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 297–311.

[25] Panagiota Fatourou and Paul Spirakis. 2000. Efficient Scheduling of Strict Multithreaded Computations. *Theory of Computing Systems* 33 (2000), 173–232.

[26] Kefeng Feng, Peng Wang, Jiaye Wu, and Wei Wang. 2020. L-Match: A Lightweight and Effective Subsequence Matching Approach. *IEEE Access* 8 (2020), 71572–71583.

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

[28] Martin Gardner. 1986. *Knotted Doughnuts and Other Mathematical Entertainments*. W. H. Freeman.

[29] Anna Gogolou, Theophanis Tsandilas, Karima Echihabi, Anastasia Bezerianos, and Themis Palpanas. 2020. Data Series Progressive Similarity Search with Probabilistic Quality Guarantees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*.

[30] Anna Gogolou, Theophanis Tsandilas, Themis Palpanas, and Anastasia Bezerianos. 2019. Progressive Similarity Search on Time Series Data. In *EDBT*.

[31] Hellium. 2018. Hellium: Ultra high performance key/value storage. https://www.levyx.com/helium

[32] Pablo Huijse, Pablo A Estevez, Pavlos Protopapas, Jose C Principe, and Pablo Zegers. 2014. Computational intelligence challenges and applications on large-scale astronomical time series databases. *CIM* (2014).

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

[34] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. 2011. 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*. IEEE, 861–864. https://doi.org/10.1109/ICASSP.2011.5946540

[35] Søren Kejser Jensen, Torben Bach Pedersen, and Christian Thomsen. 2017. Time Series Management Systems: A Survey. *IEEE Trans. Knowl. Data Eng.* 29, 11 (2017), 2581–2600.

[36] Kunio Kashino, Gavin Smith, and Hiroshi Murase. 1999. Time-series active search for quick retrieval of audio and video. In *ICASSP*.

[37] Eamonn Keogh and Chotirat Ann Ratanamahatana. 2005. Exact indexing of dynamic time warping. *KIS* (2005).

[38] Eamonn J. Keogh, Kaushik Chakrabarti, Sharad Mehrotra, and Michael J. Pazzani. 2001. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. In *SIGMOD*, Sharad Mehrotra and Timos K. Sellis (Eds.).

[39] Michele Linardi and Themis Palpanas. 2019. Scalable, Variable-Length Similarity Search in Data Series: The ULISSE Approach. *PVLDB* (2019).

[40] Michele Linardi and Themis Palpanas. 2020. Scalable Data Series Subsequence Matching with ULISSE. *VLDBJ* (2020).

[41] Michele Linardi, Yan Zhu, Themis Palpanas, and Eamonn J. Keogh. 2020. Matrix Profile Goes MAD: Variable-Length Motif And Discord Discovery in Data Series. In *DAMI*.

[42] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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

[44] Themis Palpanas. 2015. Data Series Management: The Road to Big Sequence Analytics. *SIGMOD Record* (2015).

[45] Themis Palpanas. 2017. The Parallel and Distributed Future of Data Series Mining. In *HPCS*.

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

[47] Themis Palpanas and Volker Beckmann. 48(3), 2019. Report on the First and Second Interdisciplinary Time Series Analysis Workshop (ITISA). *SIGREC* (48(3), 2019).

[48] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *VLDB* (2015).

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

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

[51] Botao Peng, Themis Palpanas, and Panagiota Fatourou. 2018. ParIS: The Next Destination for Series Indexing and Query Answering. *IEEE BigData* (2018).

[52] Botao Peng, Themis Palpanas, and Panagiota Fatourou. 2020. MESSI: In-Memory Data Series Indexing. In *ICDE*.

[53] Botao Peng, Themis Palpanas, and Panagiota Fatourou. 2020. ParIS+: Data Series Indexing on Multi-core Architectures. *TKDE* (2020).

[54] Prometheus. 2018. Prometheus – Monitoring system & time series database. http://prometheus.io/

[55] QuasarDB. 2018. QuasarDB: high-performance, distributed, time series database. https://www.quasardb.net/

[56] Davood Rafiei and Alberto O. Mendelzon. 1998. Efficient Retrieval of Similar Time Sequences Using DFT. In *FODO*, Katsumi Tanaka and Shahram Ghandeharizadeh (Eds.).

[57] Usman Raza, Alessandro Camerra, Amy L Murphy, Themis Palpanas, and Gian Pietro Picco. 2015. Practical data prediction for real-world wireless sensor networks. *TKDE* (2015).

[58] RiakTS. 2018. Riak TS – Basho Technologies. http://basho.com/products/riak-ts/

[59] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB*.

[60] Jin Shieh and Eamonn Keogh. 2008. ISAX: Indexing and Mining Terabyte Sized Time Series. (2008), 9. https://doi.org/10.1145/1401890.1401966

[61] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, Suhas Jayaram Subramanya, and Jingdong Wang. 2022. Results of the NeurIPS?21 Challenge on Billion-Scale Approximate Nearest Neighbor Search. In *Proceedings of the NeurIPS 2021 Competitions and Demonstrations Track*. 177–189.

[62] S Soldi, Volker Beckmann, WH Baumgartner, Gabriele Ponti, Chris R Shrader, P Lubiński, HA Krimm, F Mattana, and Jack Tueller. 2014. Long-term variability of AGN at hard X-rays. *Astronomy & Astrophysics* 563 (2014), A57.

[63] Timely. 2018. Timely – A secure time series database based on Accumulo and Grafana. https://code.nsa.gov/timely/

[64] Timescale. 2018. Timescale - an open source time series management system. http://timescale.com/

[65] Skoltech Computer Vision. 2018. Deep billion-scale indexing. http://sites.skoltech.ru/compvision/noimi.

[66] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin Mcgrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. 2020. Apache IoTDB: Time-series database for Internet of Things. *Proc. VLDB Endow.* 13, 12 (2020), 2901–2904.

[67] Qitong Wang and Themis Palpanas. 2021. Deep Learning Embeddings for Data Series Similarity Search. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*. ACM, 1708–1716.

[68] Qitong Wang, Stephen Whitmarsh, Vincent Navarro, and Themis Palpanas. 2023. iEDeaL: A Deep Learning Framework for Detecting Highly Imbalanced Interictal Epileptiform Discharges. *PVLDB* 16, 2 (2023).

[69] Yang Wang, Peng Wang, Jian Pei, Wei Wang, and Sheng Huang. 2013. A Data-adaptive and Dynamic Segmentation Index for Whole Matching on Time Series. *PVLDB* 6, 10 (2013).

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

[71] Warp10. 2018. Warp 10 – The Most Advanced Time Series Platform. https://www.warp10.io/

[72] Jiaye Wu, Peng Wang, Ningting Pan, Chen Wang, Wei Wang, and Jianmin Wang. 2019. 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*. IEEE, 866–877.

[73] Djamel Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2017. DPiSAX: Massively Distributed Partitioned iSAX. (2017), 1135–1140. https://doi.org/10.1109/ICDM.2017.151

[74] Djamel-Edine Yagoubi, Reza Akbarinia, Florent Masseglia, and Themis Palpanas. 2020. Massively Distributed Time Series Indexing and Querying. *IEEE Transactions on Knowledge and Data Engineering* 32, 1 (2020), 108–120. https://doi.org/10.1109/TKDE.2018.2880215

[75] Lexiang Ye and Eamonn Keogh. 2009. Time series shapelets: a new primitive for data mining. In *SIGKDD*. ACM.

[76] Liang Zhang, Noura Alghamdi, Mohamed Y. Eltabakh, and Elke A. Rundensteiner. 2019. TARDIS: Distributed Indexing Framework for Big Time Series Data. In *ICDE*.

[77] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2014. Indexing for interactive exploration of big data series. In *SIGMOD*.

[78] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. 2016. ADS: the adaptive data series index. *VLDB J.* (2016).

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

[80] Kostas Zoumpatianos, Yin Lou, Themis Palpanas, and Johannes Gehrke. 2015. 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*. 1603–1612. https://doi.org/10.1145/2783258.2783382

[81] Kostas Zoumpatianos and Themis Palpanas. 2018. Data Series Management: Fulfilling the Need for Big Sequence Analytics. In *ICDE*.