Noname manuscript No.
(will be inserted by the editor)

ADS: The Adaptive Data Series Index

Kostas Zoumpatianos · Stratos Idreos · Themis Palpanas

the date of receipt and acceptance should be inserted later

Abstract Numerous applications continuously produce big amounts of data series, and in several time critical scenarios analysts need to be able to query these data as soon as they become available. This, however, is not currently possible with the state-of-the-art indexing methods and for very large data series collections. In this paper, we present the first adaptive indexing mechanism, specifically tailored to solve the problem of indexing and querying very large data series collections. We present a detailed design and evaluation of our method using approximate and exact query algorithms with both synthetic and real datasets. Adaptive indexing significantly outperforms previous solutions, gracefully handling large data series collections, reducing the data to query delay: by the time state-of-the-art indexing techniques finish indexing 1 billion data series (and before answering even a single query), our method has already answered $3 * 10^5$ queries.

1 Introduction

Data produced in the form of sequences of values are omnipresent. Such data can be networking information, web usage data, scientific data (e.g., electrocardiograms, weather data, etc.) as well as financial data (e.g., stock market data), to practically any kind of data series [33, 60, 49, 44, 25]. A common characteristic is that analysts need to examine the sequence of values (i.e., the data series) rather than the individual points independently. However, for reasons that we describe in this section, this type of analysis is particularly expensive, making interactive exploration of data series difficult.

Big Data Series Collections. Informally, a data series is a sequence of values ordered along a dimension (time for time-series). Recent advances in sensing, networking, data processing and storage technologies have significantly eased the process of generating and collecting tremendous amounts of data series at extremely high rates and volumes. In this way, there has been a significant interest in the data management community towards analyzing data series with the least possible processing and storage cost [42, 57, 10, 46, 16, 17].

The Data to Query Gap. For big data exploration, it is prohibitive to rely to full sequential scans for every single query, and therefore, indexing is required. The target of indexing is to make query processing efficient enough, such that the analysts can repeatedly fire several exploratory queries with quick response times.

However, we show in this paper that the amount of time required to build a data series index can be a significant bottleneck; Figure 1 shows that it takes more than a full day to build a state-of-the-art index (iSAX 2.0 [11]) over a data set of 1 billion data series in a modern server machine. The main cost components of indexing are reading the data to be indexed, spilling the indexed data and structures to disk, as well as incurring the computation costs of figuring out where each new data entry belongs to (in the index structure). As the data size grows, the total indexing cost increases dramatically, to a degree where it creates a big and disruptive gap between the time when the data is available and the time when one can actually have access to the data. In fact, as the data grows, the query processing cost (10^5 queries in the case of Figure 1) increasingly

K. Zoumpatianos
University of Trento, Trento, TN, Italy
E-mail: zoumpatianos@disi.unitn.eu

S. Idreos
Harvard University, Cambridge, MA, USA
E-mail: stratos@seas.harvard.edu

T. Palpanas
Paris Descartes University, Paris, France
E-mail: themis@mi.parisdescartes.fr

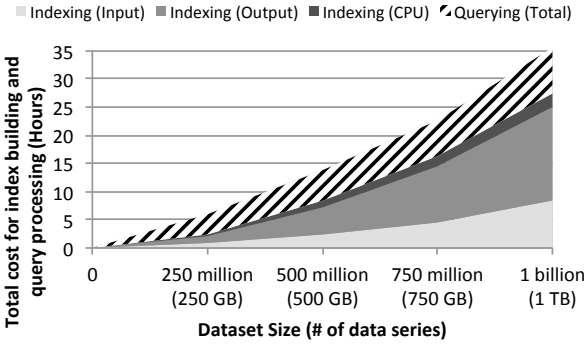


Fig. 1 The data to query gap: building a state-of-the-art index and answering 10^5 queries for big data series collections.

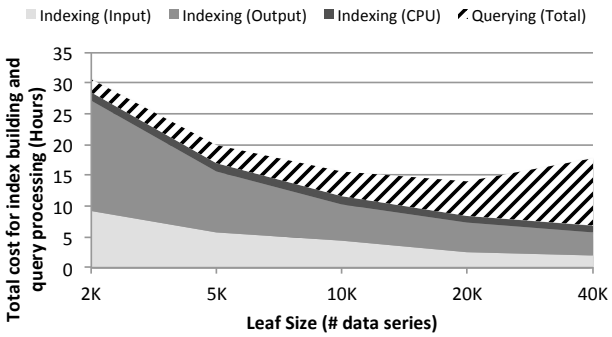


Fig. 2 The indexing to querying trade-off: bigger leaf sizes improve indexing speed, but penalize query answering times.

becomes a smaller fraction of the total cost (indexing & querying). We will discuss this experiment and its set-up in more detail later on, but for now it is interesting to note that the performance shown in Figure 1 is actually the optimal one as we have chosen a leaf size which enables a quick index build time.

As Figure 2 shows (for a 500 million data series set), the smaller the leaf size is the harder it becomes to build an index, while the bigger the leaf size is, the more we penalize query answering times (10^5 queries in this case). Thus, simply choosing a large leaf size does not resolve the data to query problem.

Data Exploration. As data sizes grow even bigger, waiting for several days before posing the first queries can be a major show-stopper for many applications both in businesses and in sciences. For example, this is the case when high velocity financial tick data have to be processed in real-time for computing risks [2], or in vehicle monitoring, where jet airplane engines can generate up to 20TB per hour that needs to be processed for early identification of potentially dangerous situations [47]. Moreover, it is not unusual for various other applications to also involve numbers of sequences in the order of hundreds of millions to billions [1,3]. These data have to be processed and analyzed, in order to identify patterns, gain insights, detect abnormalities, and extract useful knowledge.

In addition, firing exploratory queries, i.e., queries which are not known a priori, is becoming quickly a common scenario. That is, in many cases, analysts and scientists need to explore the data before they can figure out what the next query is, or even which experiment to perform next; the output of one query inspires the formulation of the next query, and drives the experimental process. In such cases, performing tuning and initialization actions up-front suffers from the fact that we do not have enough knowledge about which data parts are of interest [26,30]. Similarly, in many applications, predefined queries are beneficial only if they can track data patterns or events within a given time limit; e.g., traffic monitoring applications for advertisement need to quickly determine user positions and interests.

Adaptive Data Series Indexing. In this work, we study the data to query time gap, and focus on the index creation bottleneck for interactive exploration of very large collections of data series. We propose the first adaptive indexing solution for data series, which minimizes the index creation time, allowing users to query the data soon after its generation, and several times faster compared to state-of-the-art indexing approaches. As more queries are posed, the index is continuously refined and subsequent queries enjoy even better execution times.

During creation time, our Adaptive Data Series index (ADS) performs only a few basic steps, mainly creating the basic skeleton of a tree which contains condensed information on the input data series. Its leaves do not contain any raw data series and remain unmaterialized until relevant queries come. As queries arrive, ADS fetches the relevant data series from the raw data, and moves only those data series inside the index. Future queries may be completely covered by the contents of the index, or alternatively ADS adaptively and incrementally fetches any missing data series directly from the raw data set. When the workload stabilizes, ADS can quickly serve fully contained queries, while as the workload shifts, ADS may temporarily need to perform some extra work to adapt before stabilizing again. In addition, ADS does not require a fixed leaf size; it dynamically and adaptively adjusts the leaf size in hot areas of the index. All leaves start with a reasonably big size to guarantee fast indexing times, but the more a given area is queried, the more the respective leaves are split into smaller ones to enhance query answering times.

The net effect is that users do not have to wait for extended periods of time before getting access to the data. Our results show that by the time state-of-the

art indexing approaches are still in the indexing phase (having answered zero queries), our proposed approach allows users to answer several hundreds of thousands of queries.

Although the concept of adaptive indexing has been studied in the context of column-store databases, there the main goal is to incrementally sort individual arrays (i.e., columns) for point or range queries over 1-dimensional points. In contrast, a data series index is a tree-based index that is tailored to answer similarity search queries over data series collections, thus requiring very different techniques, able to simultaneously index multiple arrays (i.e., data series).

Contributions. Our contributions are summarized as follows.

- We demonstrate the inability of state-of-the-art indexing to cope with exploratory analysis of very large data series collections. We show that the index creation time is a major bottleneck which becomes exponentially worse as data grows.
- We introduce the first adaptive data series index. Adaptive data series indexing minimizes the data to query gap by delaying actions until they are absolutely necessary. Initialization cost is kept at very low levels; only a minimal tree structure based on a summary of the data is built initially. Then, the index structure is continuously enriched as more data and queries arrive and only for the hot part of the data. Each query that is not covered by the current contents of the index, triggers a sequence of actions that have as a side-effect more data to be brought inside the index.
- We demonstrate that no special set-up is required regarding critical low-level details such as leaf size and tree depth. We propose adaptive data series indexing algorithms that start with a rather big leaf size and a shallow tree in order to minimize initialization costs for new data, but then as queries arrive and focus to specific data areas, they adaptively and automatically expand hot subtrees and adjust leaf sizes in the hot branches of the index to minimize querying costs.
- We present algorithms for both approximate and exact query answering. In both cases, we make sure that new data are loaded in the index at a controlled rate (by limiting the number of leaves that are materialized). This is particularly useful as we want to amortize the index creation cost over multiple queries. For the exact search, we describe an algorithm that clearly departs from traditional approaches. Existing exact query answering algorithms suffer from a potentially large number of random disk accesses, because of the need to visit leaves on a most-promising-first order. In contrast, the exact algorithm we propose ensures a sequential disk access pattern. It starts by computing lower bounds based on a summarized version of the data (that fits in main-memory), leading to a skip-sequential access pattern on the raw data on disk.
- We experimentally evaluate our approach using both synthetic and real-world datasets, and demonstrate a drastic reduction in the data to query time. The approximate search algorithm is able to handle several hundreds of thousands of queries by the time that state-of-the-art data series (iSAX 2.0 [11]) and multi-dimensional (R-trees [23], X-trees [9], KD-Trees [8]) techniques are still in the index creation phase. Moreover, we show that our approach is faster than the state of the art, also for the task of full index creation.

Outline. The rest of the paper[1] is organized as follows. In Section 2, we discuss related work and present the necessary background. Section 3 presents ADS in detail. Section 4 presents a novel exact search algorithm that outperforms traditional approaches. Section 6 describes the experimental evaluation, and Section 7 concludes and discusses future work.

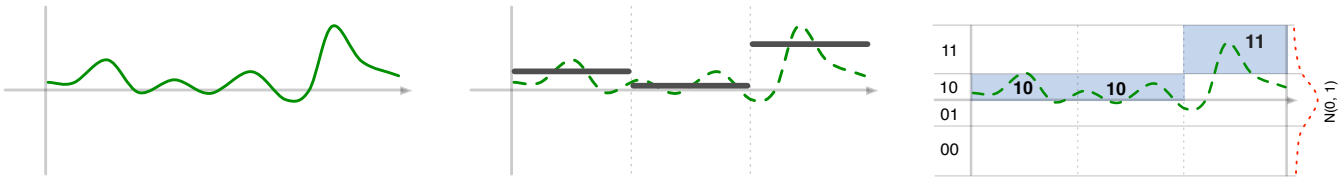
2 Preliminaries and Related Work

In this section, we provide some preliminary definitions and introduce the related work on adaptive indexing and on state-of-the-art data series indexing.

Data Series. Formally, a data series $T = (p_1, ...p_n)$ is defined as a sequence of points $p_i = (v_i, t_i)$ where each point is associated with a value v_i and a position t_i, which defines the order of this point in the sequence (t_i can be time in the case of time series).

Similarity Search in Data Series. One of the most basic data mining tasks is that of finding similar data series in a database [6]. The query comes in the form of a data series X and it says "find me the data series in the database which is most similar to X". Similarity search is an integral part of most data mining procedures, such as clustering [36,58,46,43] classification and deviation detection [10,14]. Similarity is measured using a distance function: Euclidean Distance (ED) [6] is usually chosen for its simplicity, while Dynamic Time Warping (DTW) [42] allows for local time shifts in the two data series. Several other distance functions have been proposed [56], most notably the Edit Distance on Real sequences (EDR) [15], and the Longest Common Subsequence (LCSS) [54].

[1] This paper is an extended version of [64]. It describes an exact search algorithm and a new full index construction method, both outperforming the state-of-the-art. It also includes more detailed discussions and additional experiments.



(a) A raw data series. (b) PAA representation of data series. (c) SAX representation of data series.

Fig. 3 Example PAA/SAX representation for a data-series.

A common approach for answering such queries is to perform a dimensionality reduction technique (the particular choice is of small importance [40,39]), such as *Discrete Fourier Transforms (DFT) [6], Discrete Wavelet Transforms (DWT)* [13], *Discrete Haar Wavelet Transforms (DHWT)Piecewise Aggregate Approximation (PAA)* [35,61], or *Symbolic Aggregate approXimation (SAX)* [37] and then use this representation for indexing. Lower bounding functions in the lower dimensionality spaces can be used to bound the true distances between data series, thus, allowing search algorithms to perform pruning. At the same time, a large set of indexing methods have been proposed for this kind of representations, including traditional multidimensional [23,9] and specialized [50,51,11,7,57] indexes.

Our work follows the same high level principles, but it is the first to introduce an adaptive indexing mechanism for data series in order to assist exploratory similarity search in big data sets. In all previous work, the data series index is built in one step a priori and no queries may be processed until the index is ready. On the contrary, in our work, query processing and index building are interleaved, resulting in a drastically reduced data to query time.

Adaptive Indexing. The concept of adaptive indexing was recently introduced in the context of column-store databases [28,27,29,31,24,48,21,22]. The intuition is that instead of building database indexes upfront, indexes are built during query processing, adapting to the workload. In particular, the algorithms are focused on how to incrementally sort columns in main-memory column-stores. The query predicates are used as pivots during the index refinement steps. Each index refinement step performed during a single query can be seen as a single step of an incremental quick-sort action. As more queries touch a column, this given column reaches closer to a sorted state. The benefit is that adaptive indexing avoids fully sorting columns up front at a high initialization cost, especially when there is no idle time to do so, or no reliable workload knowledge that this is indeed needed. These ideas have also been extended lately for Hadoop-based environments [45].

Even though in this paper we follow the same philosophy, our work is the first to design an adaptive index for data series processing and similarity search queries. Contrary to working with arrays as in column-store relational databases in the case of cracking, our work is based on tree-structures, which are suited for data series indexing, where we index more than one columns at a time (since each data series can be considered an array) via reduced resolutions. One could also consider storing a data series as a row in a column-store, i.e., each point being a separate attribute and then use adaptive indexing. However, then we lose the locality property as accessing one data series would require accessing several different files. Sideways cracking [29] has been proposed in order to handle multiple columns in a column-store, but this is a completely different paradigm, indexing a single relational table across one dimension at a time, and essentially relying in replication to align columns. In addition, contrary to indexing relational data where a global ordering can be imposed, i.e., incrementally creating a range index, in our case a global ordering is not possible and we are answering nearest neighbor queries. Our index introduces several novel techniques for adaptive data series indexing such as creating only a partial tree structure deep enough to not penalize the first queries with a lot of splits, and filling it on demand, as well as adapting leaf sizes on-the-fly and with varying leaf sizes across the index. Some concepts that have appeared in past adaptive indexing work apply here as well, but only as concepts, as the design of the algorithms and data structures is tailored for data series. For example, like in [31,48] we start with a lightweight preparatory step, but without having a global unique ordering of the data. In addition, the notion of adaptively bringing the data inside the index is conceptually similar to partial sideways cracking [29].

Data Series Representations and the iSAX Index. We now discuss the state-of-art data series indexing schemes. In 2000, Yi and Faloutsos [61], as well as Keogh et al. [35], both independently presented the idea of segmented means [61] or Piecewise Aggregate Approximation (PAA) representation [35]. This representation allows for dimensionality reduction in the time

domain, by segmenting the data series in equal parts and calculating the average value for each segment. An example of PAA representations can be seen in Figure 3; in this case the original data series is divided into 3 equal parts. Based on PAA, Lin et al. [37] introduced the Symbolic Aggregate approXimation (SAX) representation. It works by partitioning the value space in segments of sizes that follow the normal distribution. Each PAA value can then be represented by a character (or a small number of bits) that corresponds to the segment that it falls into. This leads to a representation with a very small memory footprint, an important requirement for managing very large data series collections. A segmentation of size 3 can be seen in Figure 3, where the data series is represented with the SAX word "10 10 11".

The SAX representation was later extended to indexable SAX (iSAX) [50]; it considers variable cardinality for each character of a SAX representation, and as a result variable degrees of precision. An iSAX representation is composed of a set of characters that form a word. Each word represents a data series available in the dataset. Each character in a word is accompanied by a number that denotes its cardinality (the number of bits that describe this character). In the case of a binary alphabet, with a word size of 3 characters and a maximum cardinality of 2 bits, we could have a set of data series (two in the following example) represented with the following words: $00_2 10_2 01_2$, $00_2 11_2 01_2$, where each character has a full cardinality of 2 bits and each word corresponds to one data series. If we now reduce the cardinality of the second character in each word, we could represent both of them with a single iSAX representation: $00_2 1_1 01_2$. That is because 1_1 corresponds to both 10 and 11, since the last bit is trailed when the cardinality is reduced. By starting with a cardinality of 1 for each character in the root node and by gradually performing splits by increasing the cardinality by one character at a time, one can build a tree index [50,51]. Such cardinality reductions can be efficiently calculated with bit mask operations.

The state-of-the-art iSAX 2.0 index is also based on this property [11]; it is a data series index that implements fast bulk loading. Figure 4 depicts an example where each iSAX word has 3 segments and each segment a maximum cardinality of 4 (2 bits). The root node has 2^w children (2^3 in Figure 4) while each child node forms a binary sub-tree. Each leaf node corresponds to a split in one dimension and points to a single area of the domain.

A typical data series index, such as iSAX, contains both the summarized representations and the actual, raw data series values. The representations are used as

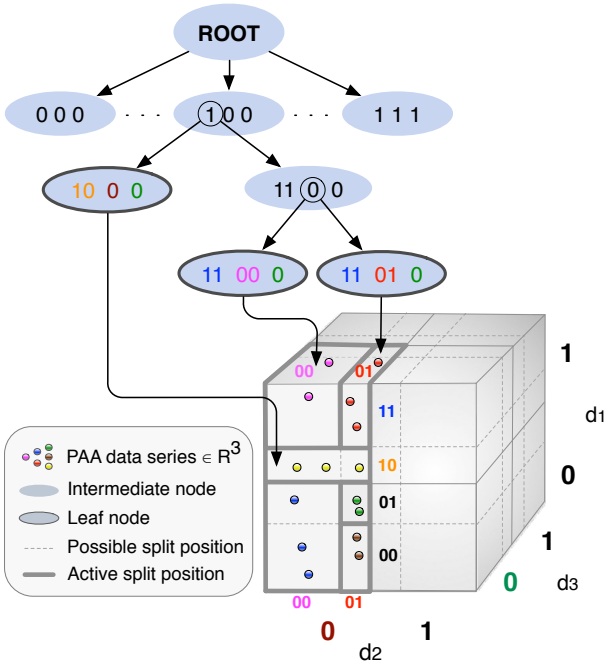


Fig. 4 An example of iSAX and its space partitioning.

index keys to efficiently guide index creation, as well as for answering similarity search queries by pruning the search space, i.e., eliminating candidate data series that cannot possibly be part of the answer (true negatives). The actual data series are also needed in order to eliminate the false positives, and produce the exact, correct answer.

Our contributions build on top of this line of work by enabling adaptive indexing using the state-of-the-art iSAX representations. Contrary to past work, our new adaptive index allows for incremental, continuous and adaptive index creation during query time. Initialization cost is kept low, bringing the ability to query the data set much sooner than in past work. We show both the significant bottleneck faced by state-of-the-art indexing as we grow to large data, as well as the drastic improvement that adaptive indexing brings.

Scans vs Indexing. Even though recent studies have shown that in certain cases sequential scans can be performed very efficiently [42], such techniques are only applicable when the database consists of a single, long data series, and queries are looking for potential matches in small subsequences of this long data series. Such approaches, however, do not bring benefit to the general case of querying a mixed database of several data series, which is the focus of this study. Therefore, indexing is required in order to efficiently support data exploration tasks, which involve ad-hoc queries, i.e., the query workload is not known in advance.

3 The Adaptive Data Series Index

As we discussed earlier, dealing with very large amounts of data series leads to new challenges in data series indexing. Specifically, we stressed the fact that state-of-the-art indexing mechanisms need a prohibitively large amount of time to build a full index: it may take up to several days to create a single index.

In this section, we describe our solution to this problem. We present adaptive data series indexing in detail, and describe how it can reduce the data to query gap by *shifting costly index creation steps from the initialization time to the query processing time*. For ease of presentation, we discuss adaptive data series indexing in two steps; initially we present ADS, a design which introduces the concept of adaptively and incrementally loading data series in the index. Then, we discuss ADS+ which introduces the concept of adaptive splits and adaptive leaf sizes. Finally, we present PADS+, an aggressive variation of ADS+, which is tailored for even better performance in skewed workloads.

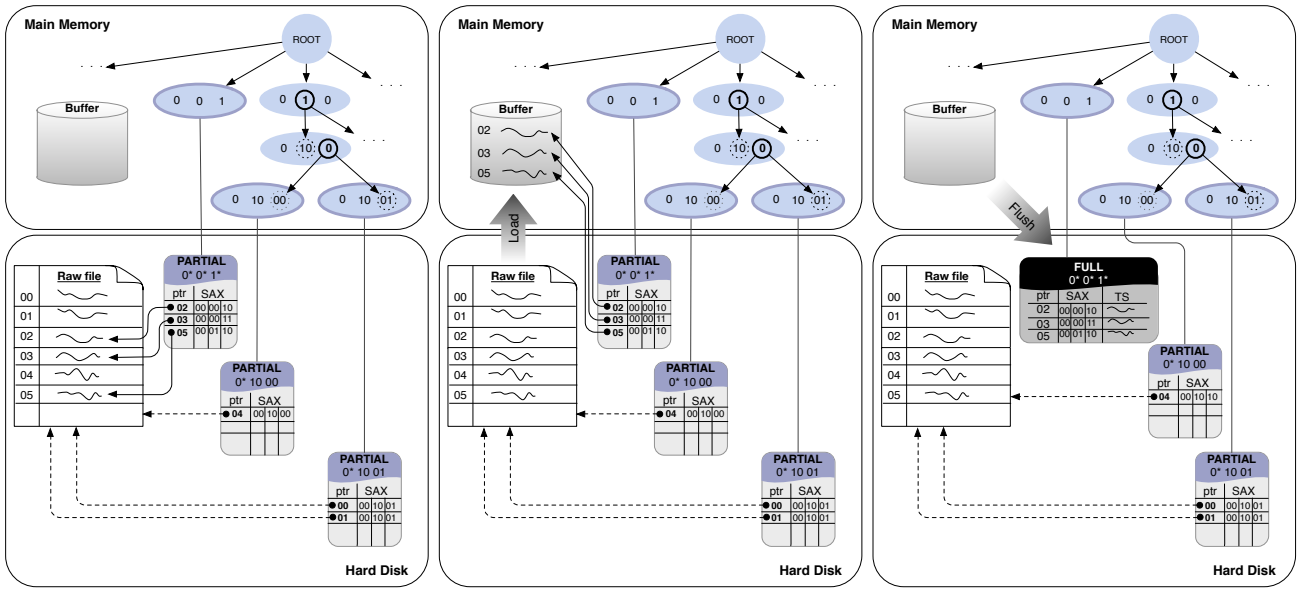
3.1 The ADS Index

In order to increase the exploration ability we need to decrease the data to query time. That is, we need to decrease the amount of time needed until a user can access and query new data at acceptable response times. The main bottleneck is the index construction overhead. ADS attacks the index construction bottleneck by shifting the construction of the leaf nodes of the index (the only nodes that can carry raw values for the data series, and have to be stored on disk) to query time. During the index creation phase, ADS creates a tree which contains only the iSAX representation for each data series; the actual data series remain in the raw files and are only loaded in an adaptive way if a relevant query arrives. On the contrary, state-of-the-art indexes, such as iSAX 2.0, a priori load all raw data series in the index at the leaves of the tree (in order to reduce random I/O during query processing). The analysis of the performance of iSAX 2.0 in Figure 1 motivates our design choice for ADS; it shows that reading from and writing to disk is the main cost component during the indexing phase of iSAX 2.0. The results show that a big part of these read and write costs is due to reading the raw data series from disk and to writing the leaves of the index tree back to disk (after insertions). Motivated by data exploration scenarios where we do not know a priori which data series are relevant for our analysis, ADS avoids these costs completely at initialization time; it pays such costs at query time, only when absolutely necessary, and only for the data which are relevant to the workload. Below we describe ADS in detail.

3.1.1 Index Creation

The index creation phase takes place before queries can be processed but it is kept very lightweight. The process can be seen in Algorithm 1. The input is a raw file which contains all data series in ASCII form. ADS builds a minimal tree during this phase, i.e., a tree which does not contain any data series. The tree contains only iSAX representations. The process starts with a full scan on the raw file to create an iSAX representation for each data series entry. This can be seen in lines 2-5 of Algorithm 1. For data series we also record its offset in the raw data file so future queries can easily retrieve the raw values. To minimize random memory access and random I/O we use a set of buffers in main memory (line 6) to temporarily hold data to be added in the index. When these buffers are full (line 7), we move the data to the appropriate leaf buffer in the index (see discussion in Buffering later on). If necessary, we perform split operations on the way (lines 12-15). The split operation is described in detail in Algorithm 2. Then we sequentially flush each leaf buffer to the disk (Algorithm 1, line 20), set each leaf to be in PARTIAL mode which means that we do not store any raw data series in this leaf (line 21). This process continues until we have indexed all raw data series. We will discuss how we handle new data (updates) later on.

Delaying Leaf Construction. The actual data series are only necessary during query time, i.e., in order to give a complete and correct answer. During the index creation time, the iSAX representations are sufficient to build the index tree. In addition, not all data series are needed to answer a particular set of queries. In this way, ADS first creates all necessary iSAX representations and builds the index tree without inserting any data series and only adaptively inserts data series during query processing (to be discussed later on). There are numerous benefits that come with such a design decision, the most important being the significantly reduced cost to build the index. While it is clear that materializing leaves on demand will incur a large random I/O cost, the main benefit comes from the fact that (a) ADS avoids dealing with the raw data series (i.e., other than the single scan on the raw file to create the iSAX representations), (b) it does not move the raw data series through the tree, and it (c) it does not place the raw data series into the leaf nodes. The data series simply stay in the raw file. This brings benefits in terms of I/O and memory bandwidth used during indexing. Especially when ADS comes to the point of spilling leaf nodes to disk (i.e., all leaves when there is no more free memory), it has a big advantage in that its leaf nodes are very lightweight, containing only iSAX representations, which can be orders of magnitude smaller than



(a) Initial state. (b) Buffered leaf state. (c) Flushed leaf state.

Fig. 5 The ADS index states during query answering.

Algorithm 1: createIndex(**file**, **index**, **n**)

```
1  while not reached end of file do
2      position = current file position;
3      dataSeries = read data series of size n from file;
4      isax = convert dataSeries to iSAX;
5      Move file pointer n points;
6      Add the (isax, position) pair in the index's FBL buffer;
7      if the main memory is full then
8          // Move data from the First Buffers (FBL)
9          // to the appropriate Leaf Buffer (LBL)
10         for every (isax, position) pair ∈ FBL buffer do
11             targetLeaf = Leaf of index for putting (isax,
               position);
12             while targetLeaf is full do
13                 Split(targetLeaf, isax);
14                 targetLeaf = New leaf for putting (isax,
                   position);
15             Insert (isax, position) in targetLeaf's LBL
               buffer;
16         // Flush all Leaf Buffers containing
17         // (isax, position) pairs to the disk, and
18         // set them in PARTIAL mode (no raw data)
19         for every leaf in index do
20             Flush the LBL buffer of this leaf to the disk;
21             Set leaf to be in PARTIAL mode;
22         clear buffers;
```

Algorithm 2: Split(**leaf**)

```
1  diskData = get data from leaf's disk pages;
2  Insert diskData in leaf's buffer (LBL buffer);
3  Split leaf in the best point and create two new children
   leaves;
4  Set leaf as an intermediate node;
5  Set leaf.leftChild in PARTIAL mode;
6  Set leaf.right in PARTIAL mode;
7  for every (isax, position) pair ∈ leaf's LBL buffer do
8      Insert (isax, position) pair in the appropriate child leaf;
```

the data series themselves. For example, a data series of 256 points with a float precision of 4 bytes, can be efficiently summarized with 16 characters of 1 byte each. Moreover, by not inserting the data series in the index, we significantly reduce the cost of splits at the leaf level during the indexing phase; the I/O cost is minimized as only iSAX representations are shuffled between index nodes. All ADS variations maintain the main index tree in memory, while leaf nodes are kept on disk.

Buffering. ADS improves locality when inserting data by buffering data at two levels of the index. Buffering amortizes random access (both in memory and on disk) and is a common practice to improve locality in tree-based indexes, e.g., [62,11], or even in database query plans (which typically have a tree shape) [63]. During index creation, instead of pushing iSAX representations through the index one at a time, ADS initially keeps those in the First Buffer Layer (FBL), a set of buffers corresponding to the children nodes of the index root. Once the FBL is full (i.e., all free memory is consumed), these representations are then passed through the tree and moved to the second layer of buffers corresponding to the leaf nodes of the index, called Leaf Buffer Layer (LBL). Data is then flushed to disk one leaf at a time, ensuring sequential writes. Additionally, every time that a leaf needs to be split and iSAX representations need to be read from disk, we keep them in the LBL, until we run out of space (Algorithm 2, lines 1-2). The leaves are flushed again when there is no more free memory.

Algorithm 3: approxSearchADS(**dataSeries**, **isax**, **index**)

```
1  targetLeaf = leaf of index where this isax should be
   inserted;
2  // Calculate the real leaf distance between the dataSeries
3  // and the raw data series that this leaf refers to or
   contains.
4  bsf = calculateRealLeafDistance(targetLeaf, dataSeries);
5  return bsf;
```

Mapping on the Raw File. ADS reduces the index creation costs by not keeping around the data series. However, the raw data series is needed when queries arrive. For this reason, ADS needs an efficient way to quickly access a given data series entry. To achieve this, ADS maintains a single pointer for each data series entry X in the leaf node where data series X would normally reside. This is a pointer to the raw data file that provides direct access to the raw data series. (As we will discuss later on, the first time the leaf is accessed by a query all pointers are dropped and the corresponding raw data series are loaded.)

Example 1 An example of ADS is shown in Figure 5 which depicts the state of the index after certain events. An index is built on top of a set of iSAX words with a word size of 3 characters and a maximum cardinality for each character of 2 bits. The leaf nodes are depicted as oval shapes with border lines and the intermediate nodes without any border lines. Each intermediate node is split on a single character; the one surrounded by a bold cycle. Each leaf node is connected to a file on disk, where the full cardinality iSAX representations and the corresponding pointers to the raw file are stored. Figure 5(a) shows how the index looks like immediately after the initialization phase and before any query has been processed. In this case, all leaf nodes are in PARTIAL mode, i.e., they do not contain any data series, since no query has been executed yet. Figure 5(b) and Figure 5(c) show what happens when a query arrives and we discuss that in the next subsection.

3.1.2 Querying and Refining ADS

We continue our discussion by describing the process of query answering using ADS. Contrary to static indexes, the querying process in ADS contains a few extra steps. In addition to answering a query q, the query process refines the index during the processing steps of q. These extra index refinement steps do not take place after the query is answered; they develop completely on-the-fly and are necessary in order to answer q. At any given time, ADS contains just enough information in order to handle the current workload. Thus, when new queries arrive, which do not follow the patterns in previous requests, ADS needs to enrich the index with more information.

We provide algorithms for both approximate search and exact search. Approximate search provides answers of good quality (returns a top 100 answer for the nearest neighbor search in 91.5% of the cases for iSAX [50,51]) with very fast response times. On the other hand, exact

Algorithm 4: exactSearchADS(**dataSeries**, **index**)

```
1  isax = convert dataSeries to iSAX;
2  bsf = approxSearchADS(dataSeries, isax, index);
3  bsfDist = Infinite;
4  queue = Initialize a priority queue with the root nodes of
   the index;
5  while node = pop next node from queue do
6      if node is a leaf and MinDist(dataSeries, node) <
       bsfDist then
7          realDist = calculateRealLeafDistance(dataSeries,
           node);
8          if realDist < bsfDist then
9              bsf = node;
10             bsfDist = realDist;
11     else if MinDist(dataSeries, node) ≥ bsfDist then
12         // Found the nearest neighbor, break the loop
13         break;
14     else
15         // It is an intermediate node: push children to the
           queue.
16         minDLeft = MinDist(dataSeries,
           node.leftChild);
17         minDRight = MinDist(dataSeries,
           node.rightChild);
18         if minDLeft < bsfDist then
19             Put node.leftChild in queue with priority
               minDLeft;
20         if minDRight < bsfDist then
21             Put node.rightChild in queue with priority
               minDRight;
22 return bsf;
```

Algorithm 5: calculateRealLeafDistance(**leaf**, **dataSeries**)

```
1  // Check if the raw data have been fetched in the leaf
2  if leaf is in FULL mode then
3      if leaf has raw data series in LBL buffer then
4          bufferBSF = find closest to dataSeries record in
           LBL;
5      if leaf has raw data series on disk then
6          diskBSF = find closest to dataSeries record on
           disk;
7      if diskBSF < bufferBSF then
8          return diskBSF;
9      else
10         return bufferBSF;
11 else if leaf is in PARTIAL mode then
12     // Materialize leaf
13     records = Get all (isax, position) pairs from disk and
       LBL;
14     Sort records based on positions;
15     for every (isax, position) pair ∈ records do
16         Seek position in raw data file;
17         rawDataSeries = Fetch raw data series from raw
           data file;
18         Insert (isax, position, rawDataSeries) tuple in
           LBL buffer;
19         if main memory is full then
20             Flush all LBL buffers on disk;
21     Set leaf to FULL mode;
22     return calculateRealLeafDistance(node, dataSeries);
```

search guarantees that we get the exact answer, but with potentially much higher query execution time.

Approximate Search. When a query arrives (in the form of a data series), it is first converted to an iSAX representation. Then, the index tree is traversed searching for a leaf with an iSAX representation similar to that of the query (Algorithm 3). This is the leaf where the query series would reside if it was a part of the indexed dataset. Whether such a leaf exists already or not, depends not only on the data, but also on past queries. If such a leaf does not exist, then the most similar leaf to the query is used instead. In the case that the leaf node where the search ends is in PARTIAL mode, i.e., it contains only iSAX representations but not any data series, then all missing data series are fetched from the raw file. To enrich a partial leaf, ADS fetches the partial leaf from disk and reads all the positions in the raw file of the data series that belong in this leaf. (A partial leaf holds the iSAX representation for each data series and also its position in the raw file.) Then, it sorts those positions (to ensure sequential access to the raw file) and fetches the raw data series. The new data series are assigned to leaf nodes and kept in memory in the LBL buffers (Figure 5(b)). The corresponding leaf node contains pointers to the buffered data. When there is no more free memory, the LBL buffers are flushed to disk (as seen in Figure 5(c)). The corresponding leaf is then marked as FULL. At this point the leaf data is fully materialized and future queries that need to access the data series for this leaf node, need to fetch the binary leaf data from disk or from the LBL buffer. Once the data series that match the current query are available (either being fetched from the raw file, from the buffer, or from disk) then the real distance from the query is calculated. The minimum distance found in the leaf is used as the approximate answer.

Exact Search. When a query arrives, an approximate search is initially issued in order to get an initial Best So Far answer (BSF). If the BSF is not 0, which means that we did not find a perfect match, then the node with the best possible answer has to be identified. This is done in a recursive way as in the original iSAX index using the MinDistPaaToiSAX [50], and until we are not able to improve BSF any further. The difference is that if a new leaf is needed, which is in partial mode, ADS will enrich this leaf on-the-fly.

The algorithm, described in Algorithm 4, starts by putting all the children of the root in a min-stack ranked using their lower distance bound towards the query (line 4). Then the one with the best minimum distance is popped (line 5) and explored, as long as this distance is better than BSF (lines 6-10). If the currently popped node is an intermediate node (lines 14-21) then its children are pushed into the min-stack for possible future exploration. The process continuous recursively, and stops when the best lower bound is bigger than the BSF distance (lines 11-13), which means that it is not possible to improve the current answer any further.

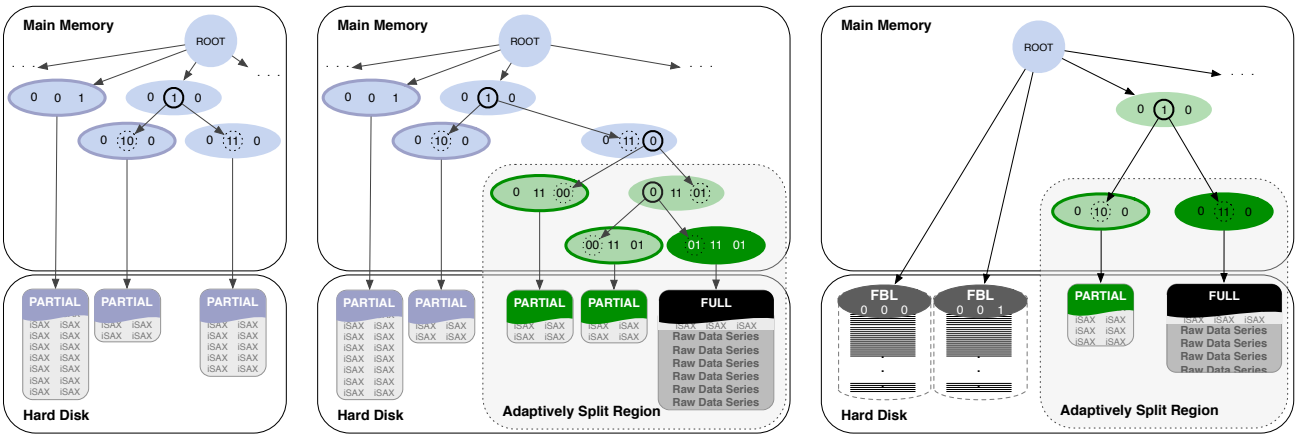
Example 2 Continuing the example of Figure 5, Figure 5(b) and Figure 5(c) show what happens when a query arrives. Figure 5(b) depicts the case when a query reaches a non materialized leaf. The raw data series are fetched in main memory buffers, and the leaf now points to them. If the buffers become full, the raw data series for each leaf are flushed to disk, thus converting them into fully materialized leaves. This can be seen in Figure 5(c); the full leaf contains both the iSAX representations and the raw data series.

3.2 The ADS+ Index (Adaptive Leaf Size)

ADS drastically reduces the index creation time by avoiding the insertion of raw data series in the index until a relevant query arrives. However, there is opportunity for significant further optimizations; by studying the operations that get executed during adaptive index building and refinement we found that the time spent during split operations in the index tree is a major cost component.

Leaf Size and Splits. Splits are expensive as they cause data transfer to and from disk (to update node data). The main parameter that affects split costs is the leaf size, i.e., a tree with a big leaf size has a smaller number of nodes overall, causing less splits. Thus, a big leaf size reduces index creation time. However, as we have shown in Figure 2, big leaves also penalize query costs and vice versa: when reaching a big leaf during a search, we have to scan more data series than with a small leaf. State-of-the-art indexes rely on a fixed leaf size which needs to be set up front, during index creation time, and typically represents a compromise between index creation cost and query cost.

Adaptive Leaf Size. To further optimize the data to query time, we introduce a lightweight variation of ADS, *ADS+*, with a more transparent initialization step. The main intuition is that one can quickly build the index tree using a large leaf size, saving time from very expensive split operations, and rely on queries that are then going to force splits in order to reduce the leaf sizes in the hot areas of the index. ADS+ uses two different leaf sizes: a big build-time leaf size for optimal index construction, and a small query-time leaf size for optimal access costs. This allows us to make future queries benefit from every split operation performed, finding the relevant data by traversing the tree, and not by scanning larger leaves. Initially, the index tree



(a) ADS+ after indexing. (b) ADS+ index after a query. (c) PADS+ index after a query.

Fig. 6 Examples of ADS+ and PADS+ states.

is built as in plain ADS (Algorithm 1), with a constant leaf size, equal to build-time leaf size. In traditional indexes, this leaf size remains the same across the life-time of the index. In our case, when a query that needs to search a partial leaf arrives, ADS+ refines its index structure on-the-fly by recursively splitting the target leaf, until the target sub-leaf becomes smaller or equal to the query-time leaf size. This can be seen in Algorithm 6. Additionally both Approximate and Exact search have been modified to use this policy, a shown in Algorithm 7 (lines 2-5) and Algorithm 8 (lines 7-10), respectively.

Intuitively what happens is that the target leaf is split until it becomes small enough, while all leaves created due to split actions but are not needed for this query are then left untouched and thus with a leaf size which is between the big construction-time leaf size and the small query-time leaf size. If and only if the workload shifts and future queries need to query those leaves, then ADS+ automatically splits those leaves even further to reach a leaf size that gives good query processing times.

Example 3 An example of this process is shown in Figures 6(a) and 6(b). Figure 6(a) depicts the state of ADS+ after initialization and before any query has arrived, while Figure 6(b) shows how a single query results in adaptive splits of the right sub-tree until the target leaf node is fully materialized; intermediate nodes remain in partial mode, with a variable leaf size.

Adaptive and on demand leaf splitting allow ADS+ to have both fast index building and fast query processing. It does not waste time on creating fine-grained versions of each sub-tree of the index, but rather concentrates on the parts that are related to the current workload. When queries focus to a subset of the dataset,

Algorithm 6: SplitADS+(**leaf**, **targetLeafSize**)

```
1 /* If the leaf size is bigger than the target leaf size, split
     node. */
2 if leaf's leaf size > targetLeafSize then
3 |   Split(node);
4 |   SplitADS+(node.leftChild, targetLeafSize);
5 |   SplitADS+(node.rightChild, targetLeafSize);
```

Algorithm 7: approxSearchADS+(**dataSeries**, **isax**, **index**, **queryTimeLeafSize**)

```
1 targetLeaf = leaf of index where this isax should be
    inserted;
2 if targetLeaf's leaf size > queryTimeLeafSize then
3 |   // It can be additionally split
4 |   SplitADS+(targetLeaf, queryTimeLeafSize);
5 |   targetLeaf = targetLeaf's descendant where this isax
  |     should be inserted;
6 // Calculate the real distance between the dataSeries
7 // and the raw data series that this leaf points to.
8 bsf = calculateRealLeafDistance(targetLeaf, dataSeries);
9 return bsf;
```

ADS+ does not need to exhaustively index and optimize all data; it rather concentrates on the most related sub-trees of the index. When the workload shifts and a new area of the index becomes relevant, then the first few queries adaptively optimize the index for the new area as well by expanding the proper sub-trees and adjusting leaf sizes.

Delaying Leaf Materialization. Another optimization that gives ADS+ a lightweight behavior is that it delays leaf materialization even further. In particular, when traversing the tree for query processing, which leads to adaptive leaf splitting, ADS+ does not materialize the initial big leaf, nor all the leaves it creates on its way to the target small leaf. For example, when ADS+ needs to split big leaf X and this results in X

Algorithm 8: exactSearchADS+(**dataSeries**, **index**, **queryTimeLeafSize**)

```
1  isax = convert dataSeries to iSAX;
2  bsf = approxSearchADS+(dataSeries, isax, index,
   queryTimeLeafSize);
3  bsfDist = Infinite;
4  queue = Initialize a priority queue with the root nodes of
   the index;
5  while node = pop next node from queue do
6      if node is a leaf and MinDist(dataSeries, node) <
       bsfDist then
7          if node's leaf size > queryTimeLeafSize then
8              // Need to split this leaf more
9              SplitADS+(node, queryTimeLeafSize);
10             Re-Insert node in queue;
11         else
12             // No need to split any more
13             dist = calculateRealLeafDistance(dataSeries,
               node);
14             if dist < bsfDist then
15                 bsf = node;
16                 bsfDist = dist;
17     else if MinDist(dataSeries, node) ≥ bsfDist then
18         // Found the nearest neighbor, break the loop
19         break;
20     else
21         // It is an intermediate node: push children to the
           queue.
22         minDLeft = MinDist+(dataSeries,
           node.leftChild);
23         minDRight = MinDist+(dataSeries,
           node.rightChild);
24         if minDLeft < bsfDist then
25             Put node.leftChild in queue with priority
               minDLeft;
26         if minDRight < bsfDist then
27             Put node.rightChild in queue with priority
               minDRight;
28 return bsf;
```

Algorithm 9: MinDist+(**dataSeries**, **leaf**)

```
1  if leaf is in FULL mode then
2      /* Use the coarse SAX representation of all the data
       series and calculate the minimum distance .*/
3      return MinDist(dataSeries, leaf);
4  else
5      /* The node is not materialized yet. We can load the
       small iSAX representations file and calculate a tighter
       minimum distance using the iSAX representations of
       all the data series. */
6      isaxValues = Get all isax representations from disk and
       LBL;
7      maxMinDist = 0;
8      for isax ∈ isaxValues do
9          minDist = MinDist(dataSeries, isax);
10         if minDist > maxMinDist then
11             maxMinDist = minDist;
12     return maxMinDist;
```

being split recursively into n new nodes until we reach the target leaf Z with a small leaf size, ADS+ fully materializes only leaf Z. For the rest of the leaves it uses the partial information contained in the leaves to perform the splits, i.e., the iSAX representations. This results in (a) less computation as opposed to having to split based on raw data, (b) less I/O as SAX representations are much smaller, and (c) it enhances the adaptive behavior of ADS+ as it materializes only the truly interesting data that the queries are targeting.

3.3 Partial ADS+ (PADS+)

Although the ADS variations described above help to reduce the indexing cost by omitting the raw data from the index creation process, ADS and ADS+ still need to spend time for creating the basic index structure. This means that users still have to wait until this process finishes, and even though it is a much faster process than full indexing, still certain applications may want *even faster* access to their data. To further optimize the data to query time, we introduce a more lightweight technique which extends ADS+ with an even more transparent initialization step. It is tailored for scenarios where users may want to fire just a few approximate queries, as well as for scenarios with high query workload skew. The new approach is named *Partial ADS+ (PADS+)* and its main intuition is to gradually build parts of the index tree, and only for small subsets of the data as queries arrive. The concept is similar to the idea of partial indexes [53] with the difference that the index is not static, i.e., it is not defined for a pre-decided set of the data; instead it continuously evolves to fit the workload.

Index Initialization. The initialization step of PADS+ is kept as lightweight as possible. PADS+ does not build an index tree at all; there is only a root node with a set of FBL buffers that contain only the iSAX representations. The only step which takes place during the initialization phase is that PADS+ creates the iSAX representations based on the raw data (as in ADS+). This requires a complete scan of the raw data. But then, instead of spending a significant effort using the SAX representations to create a tree as ADS+ does, PADS+ stops at this point and is ready to process queries. The iSAX representations are first kept in in-memory FBL buffers and then (contrary to ADS and ADS+) spilled to disk when the buffers are full. Since these buffers persist on disk, we refer to them as *FBL persistent-buffers (FBL p-buffers)*. All these steps are similar to a subset of the initialization effort that takes place for ADS+. This approach allows PADS+ to significantly reduce the data-to-query time.

Adapting to Queries. PADS+ continuously and incrementally is refined as queries arrive. As the workload shifts and requires new data areas, the nodes in the index tree are adaptively and recursively split to smaller nodes that contain the required data. It follows the same procedure as with ADS+ with the difference that the starting point is an index with a just single root node with no children nodes. In this way, only the parts of the index which are truly relevant for the workload are further developed as queries arrive.

Skewed Workloads. Such an adaptive design favors scenarios where there is high skew in the workload, i.e.,

only part of the dataset is interesting, or when there is periodical skew in the sense that queries focus on a single area of the domain for a given time before the focus shifts to another area.

Querying. When a query is issued, PADS+ converts the query to its iSAX representation and finds the corresponding FBL p-buffer. It then loads the iSAX representations and adaptively splits the buffer data, until the query-time leaf size is reached, at which point it loads the raw time series for that leaf. This process is repeated during query answering, performing adaptive splits every time that algorithm has to calculate the distance to a leaf node that has not yet been split to the query-time leaf size.

When the query answering algorithm needs data that are missing from the tree, it needs to scan the data of the corresponding FBL p-buffer and perform an adaptive split operation on it. In this process, the initial leaf size is set to infinite; thus, adaptive split operations can be performed by splitting the large buffers and creating large leaf files, which are split again *only* if there is a query that asks for them.

Furthermore, using 16 PAA segments (which is common in practice), we initially have 2^{16} FBL buffers. As a result, given a dataset of 1 billion data series, each one of the 65536 FBL buffers will on average contain around 15 thousand iSAX representations. Using a 1 byte representation for each iSAX character (i.e., cardinality 256), and given the fact that we have 16 segments, we would need 16 bytes for representing each data series. This means that the average FBL size would be around 235 KB: a file size that makes it trivial to perform split operations on.

Example 4 An illustration of the PADS+ index can be seen in Figure 6(c). It represents a random instance after a few queries have arrived. The index is not fully built; only a small part of the index is created and only some of the leaves are materialized, following the workload. For example, the two leftmost children of the root point directly to FBL p-buffers on disk; no query has gone through this path. On the contrary, the rightmost child of the root is split, leading to a subtree which reaches down to two leaf nodes. This subtree is created as a side-effect of a query requesting for data series that belong in the leaf that is now marked as FULL in Figure 6(c).

3.4 Updates

Efficiently supporting index updates is an important problem that has gathered a lot of attention [20,5]. ADS has been designed to efficiently support updates (insertions/deletions), as well. This process can be seen in Figure 7, where we depict the status of the index after a series of operations, involving insertions, leaf materializations and deletions.

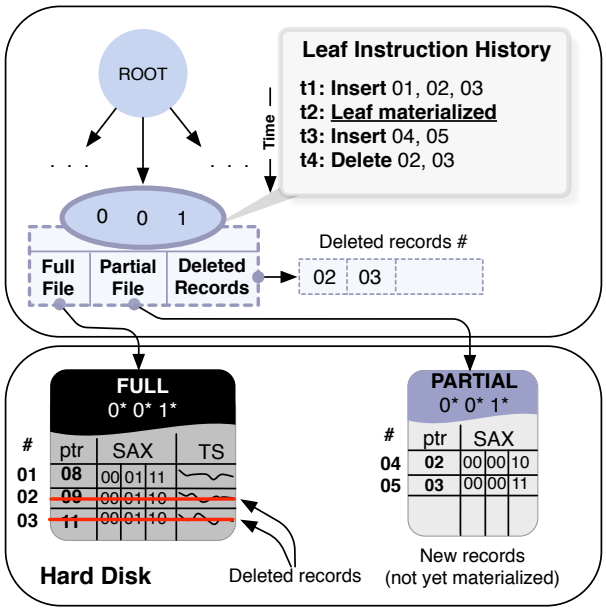


Fig. 7 Insertions and Deletions in ADS.

Inserts. Insertions is the main scenario of interest in data exploration environments, i.e., in a scientific database new data is continuously created, but past data is not discarded. Handling inserts in all ADS variations is done by simply appending the new data series in the raw file, while only its iSAX representation and its position in the raw file is pushed through the index tree.

If the index leaf has already been materialized and is in FULL mode, we create an additional PARTIAL file where the new data series reside. We then set a bit that informs us that the PARTIAL file is not empty. No further actions are needed for partial leaves. If a future query reaches a FULL leaf with pending inserts, then it fetches the new inserts on-the-fly and merges them in the leaf in the same way it is done for PARTIAL leaves (as we discussed earlier).

Example 5 Figure 7 illustrates the example of a single leaf in the presence of several updates. Initially this leaf is empty, and at time $t1$, data series 01, 02 and 03 are inserted. During $t2$ the leaf is materialized (as a result of a query that had to access it). This results in the above three data series to be part of the FULL file after $t2$. In $t3$, two more data series are inserted, and as a result they end up in the PARTIAL file. Since no query has accessed this leaf after $t3$, the new data series is not materialized.

Deletes. When a data series needs to be deleted, we simply mark the data series as deleted in its corresponding leaf (via an in-memory per-leaf bit-vector). Whether the leaf is partial or full does not make a difference. Future queries ignore deleted data series, while future insertions can exploit the space created in this leaf by these ghost entries.

Example 6 This case is also depicted in the example of Figure 7, where at time $t4$, data series 02 and 03 are deleted. We update the in-memory bit-vector of deleted records to include these two data series. Therefore, the data series are marked as deleted and their locations can be overwritten with new data.

In the case where a leaf becomes completely empty, we destroy the leaf and clear the memory that it occupies. If that leaf had no siblings, the parent node is also deleted. This process is propagated upwards until we reach a node that has a non-empty sibling.

3.5 Full Index Construction: ADS-Full

In settings where a complete index is required, i.e., when there is a completely random and very large workload, a full index can also be efficiently constructed with ADS. Our approach, called *ADS-Full*, is comprised of two steps. In the first step, the ADS structure is built by performing a full pass over the raw data file, storing only the iSAX representations at each leaf. In the second step, one more sequential pass over the raw data file is performed, and data series are moved in the correct pages on disk.

The benefit of this process is that it completely skips costly split operations on raw data series: indeed, split operations are performed only on iSAX summarizations, and mostly within the bounds of main memory. The reason is that iSAX summarizations correspond merely to 1.5% of the raw data size, and as a result 1 TB of raw data series can be summarized with 16 GB using iSAX summarizations. This means that a single pass over the raw data file enables the construction of the complete index using the iSAX summaries, entirely in main-memory. In this case, all split operations are performed in main memory, and the data structure is flushed on disk only after the entire process has finished.

During the second step, the raw data file is read again, and their appropriate locations on disk are identified by index lookup operations, as follows: we compare the isax summary of each raw series to the ADS-Full nodes during a single path traversal of the index, until we identify the leaf node in which this series belongs in. These are mostly binary operations, and as a result, extremely fast. Data are then buffered at the LBL level, and when there is no more free main memory, they are sequentially flushed on disk. As we demonstrate in the experiments section, this approach is 40% faster than building the complete index using iSAX 2.0.

4 Exact Query Answering for ADS+

Approximate Search in ADS+ (Algorithm 7) works by visiting the single most promising leaf, and calculating the minimum distance to the raw data series contained in it. This allows us to provide an approximate solution that is close to the actual answer, while at the same time controlling the time spent on reading raw data from the index.

Exact Search on the other hand, requires visiting a much larger part of the dataset, in order to guarantee that the returned answer is truly the closest match to the query in the entire collection. Traditionally, such algorithms (like Algorithm 8 for ADS+) push index nodes into a priority queue, based on their minimum distance estimation. The "closest" ones are the nodes visited first, and the answer is gradually refined as more leaves are visited.

While a large number of raw data may be pruned, the disk accesses involved in this process are random. This is because the raw data-series for each leaf reside on a different page of the disk, and leaves are visited in a most-promising-first fashion. In this way, a significant number of CPU cycles is wasted waiting for data to be fetched from disk.

To overcome this problem, we propose a skip sequential scan algorithm: it employs approximate search as a first step in order to prune the search space, it then accesses the data in a sequential manner, and finally it produces an exact, correct answer. We call this algorithm Scan of In-Memory Summarizations (SIMS). The main intuition is that while the raw data do not fit in main memory, their summarized representations, which can be orders of magnitude smaller, will fit. For example, the size of a 16-segment iSAX representation for a single data series is 16 bytes, while a raw data-series of 256 float points is 1,024 bytes. The iSAX summaries of 1 billion data series occupy merely 16GB in main memory. By keeping these data in-memory and scanning them, we can estimate a bound for every single data series in the dataset.

The algorithm (refer to Algorithm 10) starts by checking if the SAX data are in memory (lines 2-3), and if not it loads them. It then proceeds to create an initial best-so-far (BSF) answer (line 5), using the Approximate Search algorithm of ADS+ (Figure 9(a)). A minimum distance estimation is performed between the query and each in-memory SAX record (lines 7-10),

Algorithm 10: exactSearchSIMS(**dataSeries**, **isax**, **index**, **queryTimeLeafSize**, **file**)

```
1  // If SAX summaries are not in-memory, load them
2  if SAXSummarizations = ∅ then
3  |  SAXSummarizations = loadSAXFromDisk();
4  // Perform an approximate search
5  bsf = approxSearchADS+(dataSeries, isax, index,
   queryTimeLeafSize);
6  // Compute minimum distances for all summaries
7  Initialize mindists[] array;
8  // Start multiple threads & compute bounds in parallel
   parallelMinDistsCompute(mindists,
   SAXSummarizations, dataSeries);
9  // Read raw data for unprunable records
   recordPosition = 0;
10 for mindist ∈ mindists do
11 |  if mindist < bsf then
12 |  |  Move file pointer to recordPosition;
13 |  |  rawData = read raw data series from file;
14 |  |  realDist = Dist(rawData, dataSeries);
15 |  |  if realDist < bsf then
16 |  |  |  bsf = realDist;
17 |  recordPosition++;
18 return bsf
```

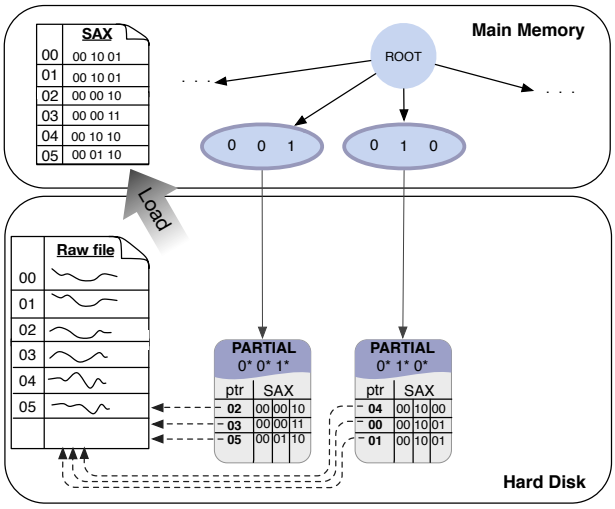


Fig. 8 SIMS initial state.

using multiple parallel threads, operating on different subsets of the data. For each lower bound distance estimation, if it is smaller than the real distance to the BSF, we fetch the complete data series from the raw data file and calculate the real distance (lines 12-14). If the real distance is again smaller than the BSF, we update the BSF value (lines 15-16).

Since the summaries array is aligned to the data on disk, what we essentially do is a synchronized skip sequential scan of the raw (on-disk) data and the (in-memory) mindists array. This property allows us to prune a large amount of data, while ensuring that we do sequential reads in both main memory and on disk, as well as enable modern multi-core CPUs to operate in parallel on the data (the SAX summaries in this case) stored in main memory. The algorithm finally returns the final BSF to the user, which is the exact answer to the query.

The initial state of the index is depicted in Figure 8, where the SAX data can be seen alongside the index in main memory. Initially, SIMS performs an Approximate Search operation, performing adaptive splits and loading data from the raw file as necessary. This can be seen in Figure 9(a). Given a BSF solution produced by Approximate Search a multi-threaded process computes the lower bounds to all in-memory summarizations and a skip-sequential read of the raw file is performed. This is shown in Figure 9(b).

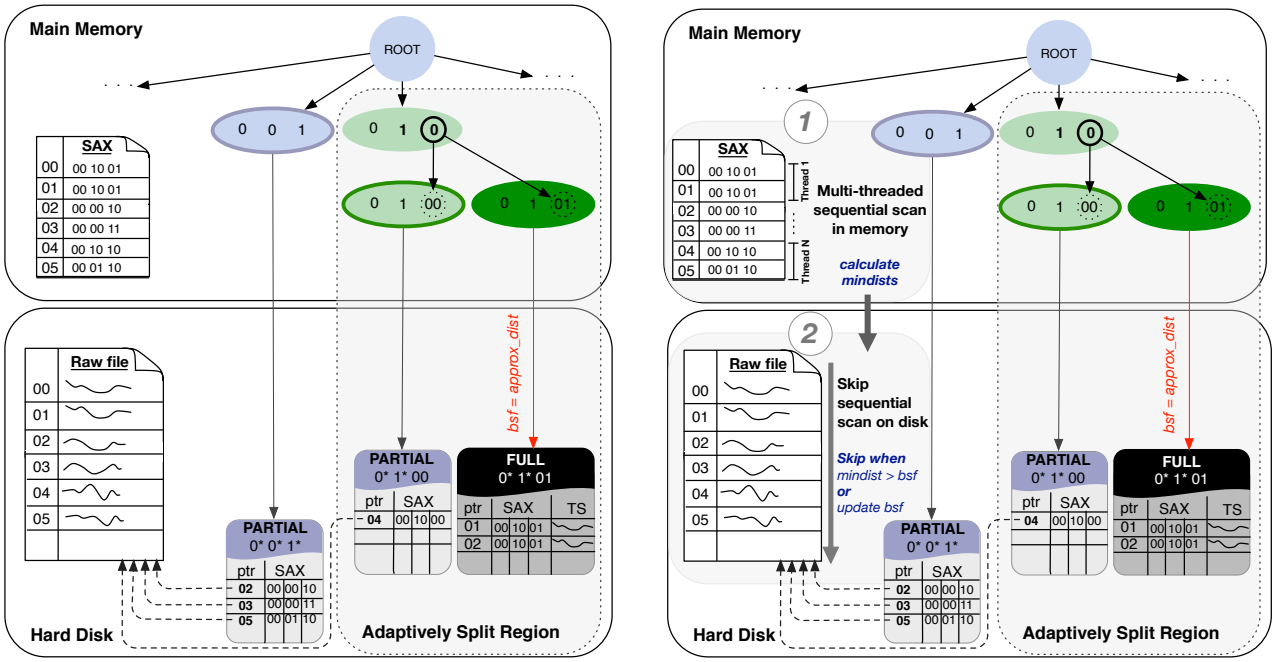
It is important to notice, that SIMS works well even in the degenerate case where our dataset comprises of identical data series. In such a case, even a single exact query could lead to the materialization of the complete index. SIMS avoids this situation, since the rate with which data are materialized is fixed across all queries, and data loading happens only during the approximate part of the algorithm.

5 Complexity Analysis

We now provide a space complexity analysis for ADS, as well as a time complexity analysis for all the search algorithms we have presented. Since the actual size of the index as well as the time needed to answer each query highly depends on the data distribution [66], we concentrate in providing lower and upper bounds for indexing and query answering.

Best Case. The ADS index is the most compact when (after all adaptive split operations) it has the smallest possible number of nodes, and all the leaf nodes are completely full. If we have N data series, and all leaves are full, then we have a total of $l_{min} = \lceil \frac{N}{th} \rceil$ leaves, where th is the query-time leaf size. In order to have the shortest possible tree, every level of the tree must have the highest possible fan-out. If w is the number of iSAX segments used, the root node of ADS has 2^w children that form binary trees. In the best case we have one binary tree for every single root child, with $\lceil 2(\frac{l_{min}}{2^w}) - 1 \rceil$ inner (in-memory) nodes, and $\frac{l_{min}}{2^w}$ leaf (on-disk) nodes each. In total, the smallest possible ADS index will have $n_{min} = 1 + 2^w \left\lceil \left(\frac{\lceil \frac{N}{th} \rceil}{2^{w-1}} \right) - 1 \right\rceil$ nodes (1 root node, and 2^w full binary trees with $\lceil \frac{N}{th} \rceil$ leaves equally distributed among them).

Approximate search in this case requires the traversal of a single path from the root of the index to one



(a) ADS+ Approximate Search.

(b) Skip-sequential Scan using SAX.

Fig. 9 SIMS during query answering.

of the leaves. This is $\Theta\left(log_2\left(\left\lceil\frac{\lceil\frac{N}{th}\rceil}{2^w}\right\rceil\right)\right)$ in-memory accesses and 1 disk read of size $\Theta(th)$ $(th << N)$.

Worst Case. In the worst case, all data series in the ADS index end up in just one of the root children nodes, and all subsequent split operations are unable to separate the data. This would happen only if all the data series were almost identical (in which case using an index would be pointless anyways), and would result in an index with one single leaf (with leaf size $th_{expanded} = N$). The maximum length of the path to this leaf depends on the number of split operations. For w iSAX segments and s bits per segment, the maximum number of split operations is then $w(s-1)$. Approximate search in this case would require $\Theta(w(s-1))$ in-memory accesses and 1 disk read of size $\Theta(N)$.

Exact Search. In the best case, exact search will need to pay the cost of one approximate search and 2^w in-memory accesses for retrieving and pruning all the root level children. This is a constant number of in-memory accesses above approximate search. In the worst case, exact search will access the entire index structure using random disk accesses. Things are different in the case of SIMS, where it is ensured that all disk accesses are sequential. In the best case, SIMS will do just one approximate search and one complete scan over all the iSAX summarizations. In a typical setting, this should be around 1.5% of the raw data size. In the worst case, SIMS will additionally need to perform one full sequential pass over the raw data file as well.

6 Experimental Evaluation

In this section, we present our experimental evaluation. We demonstrate that adaptive data series indexing drastically reduces the initialization time, achieving up to one order of magnitude smaller data-to-query time when compared to state-of-the-art approaches. We show that our algorithms enable users to perform hundreds of thousands of queries faster, while the index creation cost is spread across multiple queries.

Algorithms. We benchmark all indexing methods presented in this paper and we compare all our adaptive indexing variations against the state-of-the-art iSAX 2.0 index [11] that supports bulk loading. We also compare against sequential scan, and two state-of-the-art multi-dimensional indexes: R-Trees [23] and X-Trees [9]. Finally, we implemented several main-memory performance optimizations in the iSAX 2.0 code: we use an LRU buffer for recently queried nodes and also after loading we maintain its last loading buffer in memory.

Infrastructure and Implementation. All the data structures and algorithms presented, as well as an optimized version of iSAX 2.0, are built from scratch in C and compiled with GCC 4.6.3 under Ubuntu Linux 12.04.2. We used an Intel Xeon machine with 64GB of

RAM and 4x 2TB, SATA, 7.2K RPM Hard Drives in RAID0. All algorithms are tuned to make maximum use of all available memory.

Datasets. We use several synthetic datasets for a fine grained analysis, as well as 4 real datasets coming from different domains, in order to demonstrate the usefulness of adaptive data series indexing in real-life scenarios.

For the synthetic datasets, we used a random walk data series generator. This is a generator, where a random number is drawn from a Gaussian distribution $N(0, 1)$, then at each time point a new number is drawn from this distribution and added to the value of the last number. This kind of data generation has been extensively used in the past [6,19,41,7,50,51,11], and has been shown to effectively model real-world financial data [19].

The real datasets are the following. The first dataset (TexMex) [32] contains 1 Billion vectors representing images. The second dataset (DNA) contains 20 Million DNA sequences coming from the Homo Sapiens and Rhesus Macaque genomes [12]. The third dataset (Seismic) contains 100 Million seismic data series collected from the IRIS Seismic Data Access repository [4]. Finally, the fourth dataset (Astro) contains 200 Million astronomical data series representing celestial objects [52]. Each dataset is z-normalized before being indexed. Unless mentioned otherwise, each data series consists of 256 points and each point has a float precision of 4 bytes.

Workloads. The query workloads for every scenario are random. Each query is given in the form of a data series q and the index is trying to locate if this data series or a similar one exist in the database. We study query intensive workloads with various patterns, including skewed workloads, as well as update workloads (the details are provided in the description of the experiments).

6.1 Reducing the Data to Query Time

In our motivation discussion in the introduction section, we discussed Figure 1 as an example that demonstrates the limits of state-of-the-art indexing techniques. For this experiment, we used a synthetic data set of up to 1 billion data series (1 TB) and 10^5 random queries (73% of which need to fetch new data from the raw file). The main observation is that as we try to index more and more data, the initialization time to build a state-of-the-art data series index becomes a prohibitive factor. With 1 billion data series it takes more than a full day in order to index all data using the state-of-the-art iSAX 2.0 index even when a preferable leaf size is used (Figure 1).

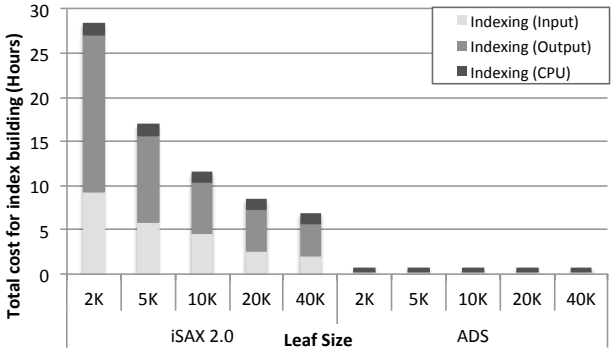


Fig. 10 Reducing indexing costs.

Minimizing Indexing Costs. Let us now see how the adaptive data series indexing ideas can help in reducing the index building costs. In this experiment, we use the same set-up as before, but we now use a constant data size of 500 million data series and we vary the leaf size. We test iSAX 2.0 against ADS.

Figure 10 depicts the results, where we show the total time needed to index all data. ADS drastically reduces the index build time compared to iSAX 2.0 regardless of the leaf size. For example, for the case of a leaf size of 20K data series, which is the best case for iSAX 2.0 (we elaborate on this choice in the following paragraphs), ADS builds the index in only half an hour, while iSAX 2.0 needs 8 hours.

The breakdown of the indexing costs in Figure 10 explains this behavior. *Input* is the time spent reading data from disk. *Output* is the time spent writing data to disk. *CPU* is the time spent doing any kind of computation during indexing. ADS avoids the expensive steps of placing each data series in its corresponding leaf node. The net result is that the Input and Output costs, i.e., the I/O costs, drop drastically compared to iSAX 2.0. At the same time, also the CPU cost drops as ADS does not have to go through the index to place each data series. Overall, reducing the I/O and CPU costs results in a major benefit for ADS during the indexing phase.

The Query Processing Bottleneck of Plain ADS. Having seen that ADS can reduce the indexing costs, let us now see the effect on query processing. Figure 11 shows the results. Using the same set-up as in the previous experiment, it depicts the total time to build the index and to process all 10^5 queries. There are two observations from the behavior seen in Figure 11. First, ADS allows its first few queries to access the data faster than iSAX 2.0. For example, if we take the best leaf size case for ADS (2K) and the best leaf size case for iSAX 2.0 (20K), we see (marked with the red arrow) that

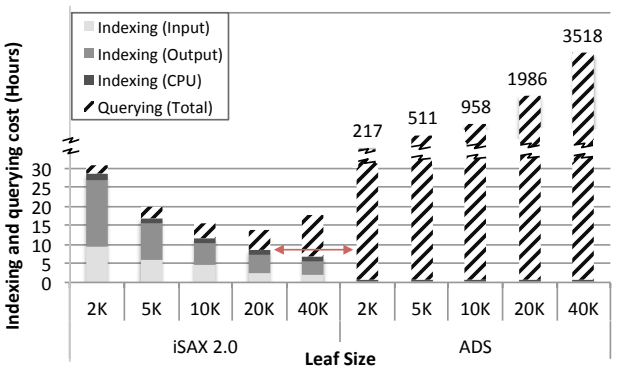


Fig. 11 The query processing bottleneck.

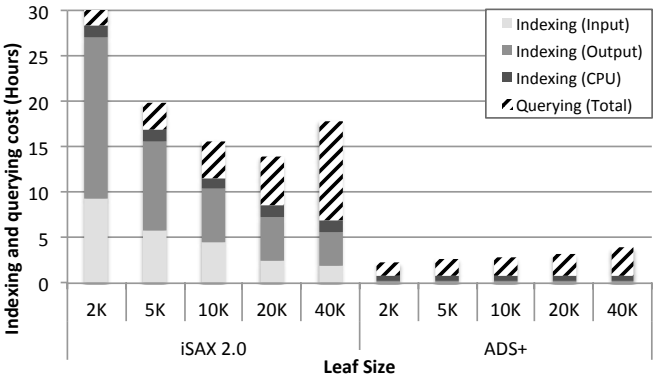


Fig. 12 Reducing the data-to-query time with ADS+.

ADS can answer 12,700 queries by the time iSAX 2.0 is still indexing and has not answered a single query (9 hours). In this way, ADS provides a quick gateway to the data as it was the original intention and motivation. However, as we process more and more queries and regardless of the leaf size, ADS looses its initial advantage; queries take too long to process and overall ADS does not present a feasible solution.

The main reason why ADS suffers is that even a single query might result in fetching a significant amount of raw data series. For example, if a query reaches a leaf which is not yet materialized and the leaf size is set to 2K, then ADS needs to fetch 2K raw data series in order to materialize the leaf. Such costs, significantly penalize queries and in the case of random workloads, as in the example of Figure 11, where each query may hit a completely different area of the index, this brings a significant overall cost. In a more focused workload, i.e., where queries focus on a given part of the index, the overall performance is drastically different as we do not reach the point where we need to fetch extra raw data very often. We discuss such examples later on.

Still though, ADS does not represent a robust solution, i.e., a solution that would be globally applicable in arbitrary workloads.

Robustness with ADS+. This is exactly the motivation for ADS+. ADS+ maintains the adaptive properties of ADS but it is also robust and scalable. To demonstrate this behavior, we repeat the previous experiment, this time using also ADS+. Figure 12 shows that ADS+ significantly outperforms iSAX 2.0 not only during the index building phase but also during the query processing phase. For example, for the best case of iSAX 2.0, i.e., with leaf size 20K, ADS+ can create the index and process all 10^5 queries in only 3 hours while iSAX 2.0 needs roughly 15 hours. In fact, ADS+ can process the queries even faster as it may use even smaller leaf sizes.

Next, we show that ADS+ is robust even when in inferior set-up. Using the same set-up (data and queries)

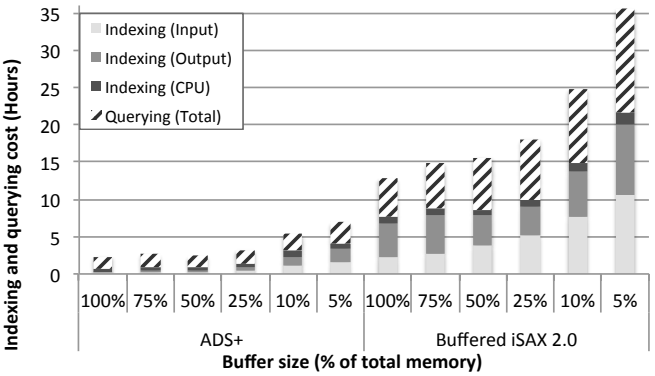


Fig. 13 Total indexing and query answering cost as we increase the buffer size for ADS+ and iSAX 2.0.

as before, we vary the available memory the algorithms can exploit. In addition, for iSAX 2.0 we use a buffer pool with an LRU policy so that it can hold recently visited nodes in memory. In Figure 13, it can be seen that even if we use 10% of the main memory for ADS+, it can still answer all of the 10^5 queries before iSAX 2.0 has finished indexing using 100% of the main memory.

The main novelty in ADS+ is that it can maintain a lightweight index-building step due to only partially building the index but also due to using a large leaf size during this phase. Then, as queries arrive, it adaptively splits leaves in hot areas of the index such that queries in this area may be processed at a smaller cost. In this way, ADS+ solves the robustness and scalability problem of ADS by introducing adaptive node splits, i.e., by being able to adjust the shape of the index based on the workload and only for the areas which are hot and may cause expensive steps for individual queries.

Choosing the Query-Time Leaf Size. The query-time leaf size indicates the finest granularity in which we will split a node with ADS+, and consequently it is directly related to the amount of raw data that we store on disk under each leaf. We have experimented with various query-time leaf sizes ranging from 1 data-

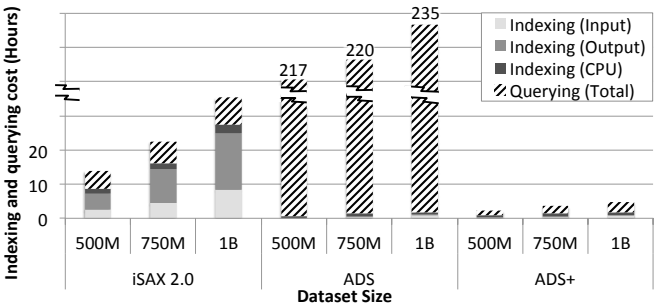


Fig. 14 Scaling to 1 billion data series.

series to 1000 data-series, and measured the average page utilization for 3 different page sizes, as well as the average query answering time. We did this by running 10^5 queries on a dataset of 500 million data-series. As we can see in Table 1, the smaller the query-time leaf size is, the less data we have to fetch from the raw data file, and the faster the materialization of the leaf node is. On the other hand, very small values of query-time leaf size adversely affect space utilization, since page occupancy will be small. As a result, it is important to choose a leaf size that will allow for the maximum page utilization while at the same time offers an acceptable query answering time. For the rest of our experiments we use 10, since when using a page size of 8KB, we maximize page occupancy at around 89% (Table 1 in bold) and the average query answering time remains relatively low at 69 milliseconds.

Query-time leaf size	1	10	100	1000
Query time (millisec.)	11.27	67.64	499.95	4031.68
# of 4KB pages	0.25	1.79	17.23	171.48
# of 8KB pages	0.12	**0.89**	8.61	85.74
# of 16KB pages	0.06	0.45	4.30	42.87

Table 1 Varying query-time leaf size.

Scaling to 1 Billion Data Series. Next, we stress all indexing strategies to study how they can cope with an increasing data set size. We study the behavior up to 1 billion data series and with 10^5 random queries. Regarding leaf sizes, we use the optimal leaf size observed for each index strategy, i.e., 20K for iSAX 2.0, 2K for ADS, and for ADS+ 2K build-time and 10 query-time leaf size. Figure 14 shows the total time to build the index and answer all queries. Across all data sizes, ADS+ consistently outperforms all other strategies by a big margin.

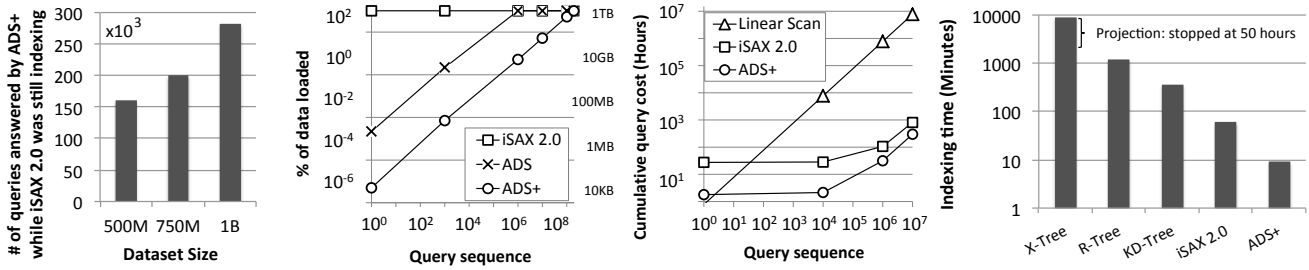
For 1 billion data series, ADS+ answers all 10^5 *queries in less than 5 hours, while iSAX 2.0 needs more than 35 hours.*

By adaptively expanding the tree and adjusting leaf sizes only for the hot workload parts, ADS+ enjoys a $7x$ gain over full indexing in iSAX 2.0. In addition, ADS+ significantly outperforms ADS; even though ADS can significantly reduce indexing costs for all data sizes, as we process more and more queries it suffers due to the high cost of fetching unindexed data series for large leaves during query processing. ADS+ avoids this problem by adaptively splitting its leaves. Also, the rate at which the cost of ADS+ grows is significantly smaller than that of iSAX 2.0; For example, going from 500M to 1B data series, iSAX 2.0 needs more than twice the time, while ADS+ enjoys a sub-linear cost increase.

Figure 15 provides further insights. Figure 15(a) depicts the number of queries that ADS+ can answer within the time that iSAX 2.0 is still indexing. The bigger the data set, the more queries ADS+ can answer before iSAX 2.0 answers even a single query; for the case of 1 Billion data series ADS+ manages to answer nearly $3 * 10^5$ queries while iSAX 2.0 is still indexing. this verifies the fact that ADS+ is more suited towards very large data sets compared to traditional non-adaptive indexing approaches.

Data Touched. In addition, Figure 15(b) shows the amount of data actually touched (indexed) as the query sequence evolves. To see the long term effect, we let a big number of queries run, i.e., 10^7 queries. For iSAX 2.0 the behavior in Figure 15(b) is a flat curve as everything is indexed blindly up front. With ADS and ADS+ though, we index a much smaller percentage of the data; as more queries are processed, more data is indexed and only when needed. While ADS indexes all data by the time it processes 10^6 queries, ADS+ manages to touch even less data; since it splits leaves adaptively to much smaller sizes it needs to materialize much smaller leaves and thus it touches less data overall. In this way, even after 10^7 queries it has touched only 10% of the data, while it needs more than 190M queries in order to touch all the data (i.e., completely build the index). In fact, since this is a random workload, this is the worst case for adaptive indexing as most queries lead to fetching raw data series and enriching the index. This is why ADS has touched all data by query 10^6; most queries will need to materialize a partial leaf and thus they need to fetch $2*10^3$ new data series (its leaf size); $2*10^3*10^6$ adds up to well above 10^9 (the data set size). On the contrary, ADS+ uses a query-time adjustable leaf size of only 10 data series; thus even if all queries need to fetch new data, by query 10^7 we would have fetched at most 10^8 data series which is about the 10% (of the original 10^9 data set) we see in Figure 15(b). By doing less work and only when necessary, ADS+ allows users to have quick access to their data.

Per Query Performance. We continue our study with a discussion that focuses on the individual query performance based on the previous 1 Billion data series



(a) # of queries answered. (b) Percentage of data indexed. (c) Per query response time. (d) ADS+ against spatial indexes.

Fig. 15 Reducing the data-to-query time with ADS+ as we scale to big data.

experiment and 10 Million random queries. Here we also include the scan strategy, i.e., when we do not build an index; instead, every query performs a complete scan over all data series. We will not use ADS from now on as ADS+ consistently outperforms ADS.

Figure 15(c) shows the cumulative per query response time as the query sequence evolves. The scan strategy has a constant but slow response time; every query adds the same cost to the total cumulative costs. Eventually, the scan strategy becomes prohibitive if we want to repeatedly query the same big data; it takes close to 10^5 hours to handle all queries. iSAX 2.0 pays a big cost to build the index (this is included in the cost of Query 1) but then queries are very fast, i.e., the cumulative cost curve is flat as every query adds very little cost. Once the index is built, every iSAX 2.0 query incurs a constant cost; still though there is a big bottleneck to access the data due to the high indexing costs which means that the first query needs to wait for several hours. On the contrary, ADS+ enjoys quick data access time; it finishes building the index and answering all queries by the time iSAX 2.0 is still indexing and has not answered a single query.

In fact, while the crossover point of the scan strategy with iSAX 2.0 is at about 35 queries, for ADS+ it is only at 2 queries. This means that for iSAX 2.0 to be useful we need to fire at least 35 queries while ADS+ starts bringing gains already after the first 2 queries. Moreover, while the average query answering time for ADS+ is about 50 milliseconds, that of iSAX 2.0 is 200 milliseconds. In other words, iSAX 2.0 is never going to amortize its initialization overhead over ADS+ and thus it is always beneficial to use adaptive indexing as opposed to full a priori indexing. This is because of the larger leaf size that is used by iSAX 2.0, in order to reduce the index building time by compromising query times a bit. On the other hand, ADS+ adaptively splits leaves for the hot part of the data and thus it can reduce access times even further. Furthermore, the cost of query answering for ADS+ (essentially, materializing the data of the leaf) increases linearly with leaf utilization. This cost ranges from 20ms when the leaf is already materialized to 160ms when the leaf contains all 10 data-series that need to be loaded from the raw file. When ADS+ needs to perform splits, the query answering times are 129ms for 1-10 splits, 138ms for 10-20 splits, 148ms for 20-30 splits, and 160ms for 30-40 splits. All these times are significantly smaller than the required time to answer a query using serial scan (more than 46min).

6.2 ADS+ vs. Multi-dimensional Indexes

One interesting question is how indexes which are tailored for data series search compare against state-of-the-art spatial indexes. In this experiment, we compare ADS+ and iSAX 2.0 against KD-Tree [8], R-Tree [23], and X-Tree [9], a state-of-the-art adaptive version of R-Tree. X-Tree creates a tree with minimal overlap between nodes and it allows for variable sized nodes in order to accommodate minimum overlapping. Such spatial indexes can be used for indexing data series and performing similarity search; the main idea is that we can use the PAA representations of data series to create a KD-Tree, an R-Tree, or an X-Tree.

Here, we use a set of 100 million data series. In all the cases, the amount of dimensions for the reduced dimensionality PAA representation is set to 16 while the original size of each data series is 256 points. Figure 15(d) depicts the time needed to complete the index building phase for each index. Overall, both data series tailored indexes, iSAX 2.0 and ADS+, significantly outperform the more generic spatial indexes. For example, iSAX 2.0 is one order of magnitude faster than R-Tree while ADS+ is two orders of magnitude faster, and more than an order of magnitude faster than KD-Tree. The raw benefit comes from the fact iSAX 2.0 and ADS+ are tailored to perform efficient comparisons of SAX representations (with bitwise operations). ADS+ being adaptive enjoys further benefits as we discuss

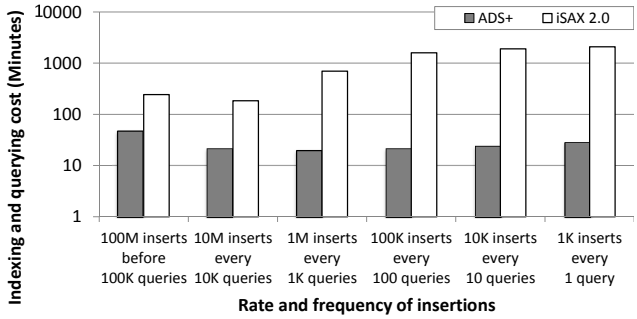
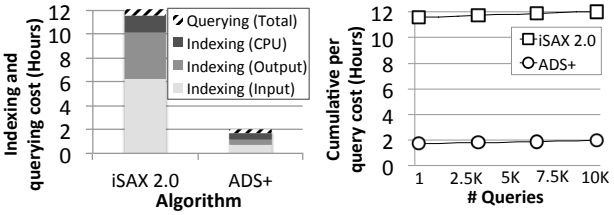


Fig. 16 Updates for 100 million data series and 100 thousand queries in 6 different batch sizes.



(a) Total cost. (b) Cumulative per query costs (Query 1 includes indexing).

Fig. 17 (TexMex) Indexing 1 Billion images (SIFT vectors) and answering 10^4 queries.

in previous experiments as well. X-Tree is significantly slower as a result of its more expensive index building phase which focuses on minimizing overlap between nodes. Naturally, this helps query processing times as less overlap allows queries to focus faster on data of interest. However, as we have shown throughout the analysis in this paper, as we scale to big data, index building is the main bottleneck and thus X-Tree is prohibitively expensive.

6.3 Adaptive Behavior under Updates

In our next experiment we study the behavior of ADS+ and iSAX 2.0 with updates. We use a synthetic data set of 100 million data series and 10^5 random queries. This time, queries are interleaved with updates. In particular, we perform the experiment in 6 steps. Each time a varying number of new data series arrive and at different query intervals. Figure 16 shows the results. The first set of bars represents the case where all data has arrived up front and all queries run afterwards. The second set of bars (10M inserts every 10K queries) represents a scenario where every 10^4 queries 10^7 new data series arrive until we reach a total of 10^8 data series (i.e., the complete data set) and a total of 10^5 queries (i.e, the complete query workload). Similarly, the rest of the bars vary the frequency and the rate of incoming data until the extreme case where we get 1000 new insertions after every single query.

In all cases, ADS+ maintains its drastic performance advantage over iSAX 2.0. When all data arrives up front, the cost is naturally higher; more data has to be queried. For the rest of the cases where data arrives incrementally, interleaving with queries, we observe that when data arrives more frequently the overall cost increases slightly. This is a result of both the fact that merging of updates needs to happen more often and of the fact that more queries need to be processed against more data. However, even in the extreme case where we receive 1000 new data series after every query, ADS+ maintains its adaptive behavior and good performance being able to outperform static iSAX 2.0 by 2 orders of magnitude.

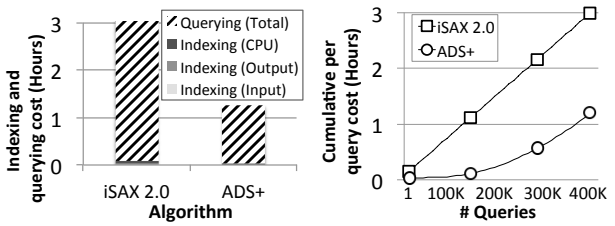
The behavior under deletions is similar. For example in experiments with a data set of 100 million data series, indexed by ADS+, we could perform deletions with an average deletion time at 0.2 milliseconds.

6.4 Reducing the Data-to-query Time in Real-life Workloads

Here, we demonstrate the ability of ADS+ to drastically reduce the data-to-query time in real-life scenarios with real data. In all cases, we use the optimal settings found in the synthetic benchmarks: for iSAX 2.0 uses a leaf size of 20K data series, while ADS+ uses a build time leaf size of 2K data series which adaptively drops down to 10.

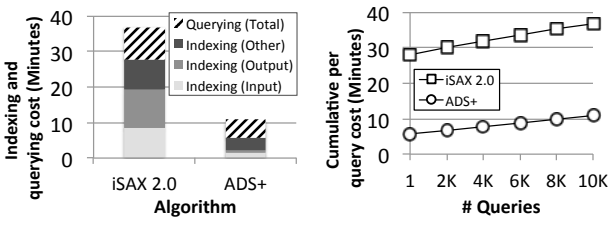
Texmex Corpus (TexMex). The first real-life scenario is an image analysis scenario from the Texmex corpus [32]. This dataset contains 1 Billion images which are translated into a set of 1 Billion data series (SIFT feature vectors) of 128 points each. The scenario is that a user is searching the corpus for images similar to an existing image that they already have. The corpus also contains 10^4 such example queries together with information about which image in the corpus is the nearest neighbor, i.e., the most similar one, for each query.

Figure 17 shows the results. Figure 17(a) shows the total cost to go through the indexing phase and to process all queries. ADS+ maintains its drastic gains as we have seen in the synthetic benchmarks study. Overall, ADS+ finishes answering all queries 6 times faster compared to iSAX 2.0. It is interesting to mention that ADS+ gains not only during the indexing phase but also during the query processing phase, i.e., the time it takes to answer all 10^4 queries is smaller with ADS+. This is because these real-life queries are not completely random, i.e., the workload focuses in specific areas of



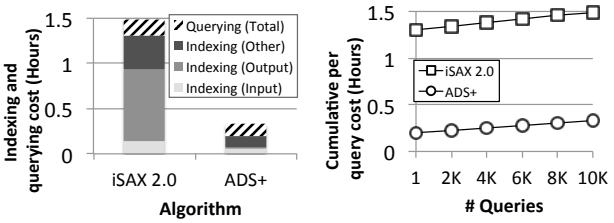
(a) Total cost. (b) Cumulative per query costs (Query 1 includes indexing).

Fig. 18 (DNA) Indexing 20 Million DNA subsequences from the Homo Sapiens genome and answering $4 * 10^5$ queries.



(a) Total cost. (b) Cumulative per query costs (Query 1 includes indexing).

Fig. 19 (Seismic) Indexing 100 Million seismic data series and answering 10^4 queries.



(a) Total cost. (b) Cumulative per query costs (Query 1 includes indexing).

Fig. 20 (Astro) Indexing 200 Million astronomical data series and answering 10^4 queries.

the index. In such cases, ADS+ has the benefit of working on an index which essentially contains less data; it has loaded only the data which are relevant for the hot workload set.

Figure 17(b) helps to understand this behavior even more by demonstrating the evolution of the query processing costs, i.e., the graph shows how the indexing and query processing costs evolve through the query sequence for each indexing strategy. For iSAX 2.0 the first query needs to wait until the whole index is built which takes almost 12 hours. From there on, each query can be processed quite fast. On the contrary, ADS+ allows the first query to access the data in less than 2 hours, while by the time we reach the 2 hours mark all 10^4 queries have been processed. Overall, ADS+ process all queries in just 2 hours, while iSAX 2.0 needs more than 11 hours just for the indexing phase and without processing a single query.

DNA Data (DNA). The second real-life scenario comes from the biology domain. This dataset contains the full genome of the Homo Sapiens (human) which is translated into 20 Million data series of 640 points each, obtained using a sliding window of size 16000, down-sampled by a factor of 25. The scenario is that a user is trying to identify subsequences of the human genome that match subsequences in other genomes. In this way, we create our queries from the genome of the Rhesus Macaque ape which is also translated into 20 Million data series of 640 points each, obtained in the same manner, and each one of these data series can be used as a query against the human genome in search for similar patterns.

Figure 18 shows the results. Similarly to previous experiments, ADS+ brings a significant benefit both in terms of total costs and in terms of per query costs. With ADS+ we can index the data and process all queries 3 times faster, i.e., only after one hour, while with iSAX 2.0 we need to wait for 3 hours. Compared to previous performance examples, it is interesting to note that in this experiment we have a very different data to queries ratio, i.e., we have a relatively small data set of 20 Million data series and a relatively big query set of $4 * 10^5$ queries. Thus, the indexing cost is a much smaller factor of the total cost compared to previous experiments. Still though, ADS+ brings a major benefit and shows a scalable behavior, mainly due to its ability to adapt its shape to workload patterns, by expanding sub-trees and adjusting leaf sizes on-the-fly.

Seismic Data (Seismic). The third real life scenario is one that comes from seismology. We used the IRIS Seismic Data Access repository [4] to gather data series representing seismic waves from various locations. We obtained 100 million data series of size 256 using a sliding window with a resolution of 1 sample per second, sliding every 4 seconds. The complete dataset size was 100GB. We additionally obtained 10,000 data series with the same technique to be used as queries. We used iSAX 2.0 and ADS+ to index the data and answer all the queries in the workload. Figure 19 shows the results. ADS+ can index the data more than 4 times faster than iSAX 2.0. With ADS+ we need to wait just under 6 minutes before we fire our first query, while iSAX 2.0 needs more than 25 minutes. In regards to query answering, we are able to index the data and answer all the 10^4 queries in 11 minutes with ADS+, while iSAX 2.0 requires 37 minutes to complete the same task.

Astronomical Data (Astro) In the last real scenario, we used astronomical data series representing celestial

Workload	Cross-over point (PADS+ over ADS+)
Random	2899 queries
Low skew	2970 queries
Medium skew	3097 queries
High skew	3825 queries

Table 2 Fast access with PADS+ with varying skew.

objects [52]. The dataset comprised of 200 million data series of size 256, obtained using a sliding window with a step of 1. The total dataset size was 200GB. We obtained an additional 10,000 data series from the raw dataset using the same technique to be used as a query workload, and used both iSAX 2.0 and ADS+ to answer the complete workload. Figure 20 shows the results. In this case, ADS+ is more than 6 times faster in indexing time than iSAX 2.0. With ADS+ we need to wait about 12 minutes before we fire our first query, while iSAX 2.0 needs more than 75 minutes. In regards to query answering, we are able to index the data and answer all 10^4 queries in less than 20 minutes with ADS+, while iSAX 2.0 requires 1.5 hours.

6.5 Providing Quick Insights with PADS+

Having shown that it is possible to reduce the user waiting time, without excessively penalizing the query answering time, we now show that we can achieve even faster access to the data for skewed workloads. In this experiment we analyze the performance of ADS+, PADS+ and iSAX 2.0 over a dataset of 1 billion data series and a varying set of query workloads, ranging from completely skewed to completely random queries. In total, we run 10^4 queries. For low skew, 60% of the queries are picked from 40% of the domain. In the medium skew workload, 80% of the queries are picked from 20% of the domain, while for the high skew workload 99.99% of the queries are picked from 0.01% of the domain.

For all workloads both ADS+ and PADS+ significantly outperform iSAX 2.0 being 10 to 20 times faster. iSAX 2.0 needs about 28 hours to index all data and process all queries with the bulk of the time spent in indexing (included in the cost of Query 1). Both ADS+ and PADS+ can do so in less than 1.5 hours for 10^3 queries, and less than 3 hours for 10^4 queries. ADS+ improves slightly as skew increases; less data has to be fetched from outside the index. PADS+, though, as seen in Table 2, manages to improve performance even more as skew increases, being faster than ADS+ and iSAX 2.0 for all skewness levels for the first 2000 queries and for almost 4000 queries in the case of high skewness. When the workload is skewed, this means that PADS+ can focus on certain parts of the index tree and avoid node splits and disk spilling once it optimizes the index for the hot part.

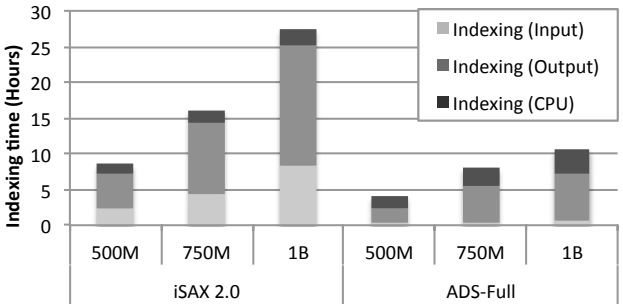


Fig. 21 ADS-Full constructs the complete index in 38% of the time that iSAX 2.0 requires for 1B data series.

While ADS+ provides the best overall solution being both fast and robust, PADS+ provides an attractive solution when we know we want to fire only a few thousands of queries.

6.6 Fast Full Index Construction

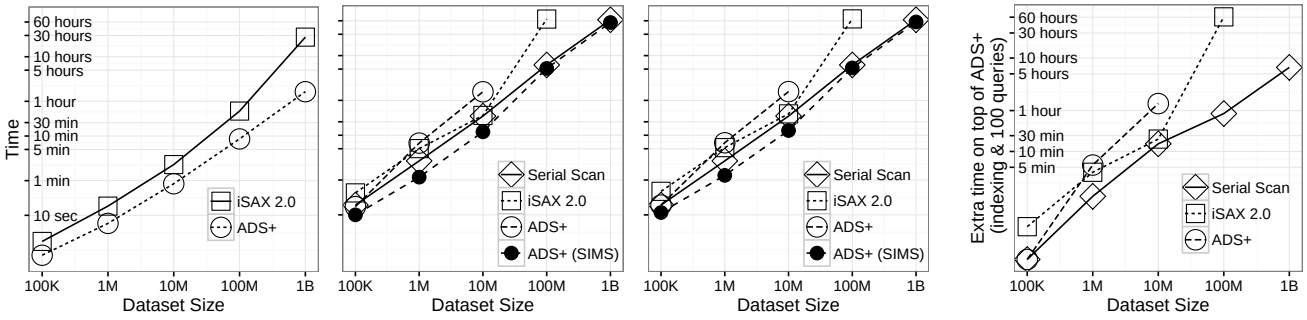
In this subsection, we demonstrate the benefits of ADS-Full for index initialization even when building the whole index in one step. For this experiment, we indexed 500M, 750M and 1B randomwalk generated data series of size 256, and compared ADS-Full with iSAX 2.0. Note that the indexes created by both ADS-Full and iSAX 2.0 are exactly the same: they contain the same inner nodes, and the same leaf nodes (along with the same corresponding raw data series). Following our earlier discussion, for both iSAX 2.0 and ADS-Full we used a leaf size of 20K.

The results are depicted in Figure 21. We observe that for 500M and 750M data series, ADS-Full requires 48% of the time of iSAX 2.0 in order to build the full index, while for the case of 1B data series ADS-Full completes the task in just 38% of the time required by iSAX 2.0. These results demonstrate that our approach outperforms the state of the art, even for the task of building a full index, for which iSAX 2.0 was initially designed.

6.7 Efficient Exact Query Answering using SIMS

In this subsection, we explore the benefits of using the SIMS algorithm for answering exact queries.

Setup. We use both ADS+ and iSAX 2.0 to index 5 random walk generated datasets with sizes of 100K, 1M, 10M, 100M, and 1B data series of length 256. Each data series has a record size of 1024 bytes. We generate queries by adding Gaussian noise to randomly selected



(a) Indexing times. (b) Querying times. (c) Indexing and querying times combined. (d) Extra time on top of ADS+ (with SIMS) for all other methods.

Fig. 22 Indexing 1 billion data series and issuing 100 exact queries.

data-series from the original dataset. The more noise we add, the harder the queries become, as they drift away from their nearest neighbor. We use queries with varying amounts of noise, in order to test the algorithms under different conditions. For each dataset we generate 100 queries, and use the following four methods to answer them.

- **Serial Scan**. This is a baseline approach, which has been shown to outperform the exact search of iSAX 2.0 in several cases [34]. We answer each query by performing a full sequential scan of the raw data file. This method implements the early abandoning technique, where we stop scanning and evaluating the distance for a data series when this distance becomes greater than the best-so-far solution.
- **iSAX 2.0**. This is the exact search algorithm of iSAX 2.0. We use the complete iSAX 2.0 index that we have built beforehand and visit nodes in a most-promising-first fashion. All nodes are pushed in a queue and the one with the minimum lower bound is popped. If this node is a leaf, then we check the full data series, retrieved from disk.
- **ADS+**. This is ADS+ implementing the exact search algorithm of iSAX 2.0. The only difference is that when we visit leaf nodes we first perform adaptive split operations and then load the data from the raw data file in the index.
- **ADS+ (SIMS).** This is the SIMS exact search algorithm. We load all the iSAX representations in main memory and perform a multi-threaded lower bound calculation. We then visit the data on the raw file for only the records with a lower bound less than the best-so-far solution obtained using Approximate Search.

We have removed the square root computation from the Euclidean Distance, for all the above approaches.

Evaluation. In Figure 22(a), we plot the indexing time for both iSAX 2.0 and ADS+. Serial Scan has no initialization cost. ADS+ outperforms iSAX 2.0 by more than an order of magnitude in terms of data-to-query time. In Figure 22(b), we plot the query answering time for all algorithms, and in Figure 22(c), we plot the indexing and query answering times combined, when the dataset varies between 100K and 1B data series.

Dataset size	Cross-over point (Serial Scan over ADS+)
100K	7 queries
1M	4 queries
10M	4 queries
100M	3 queries
1B	3 queries

Table 3 ADS+ outperforms serial scan after a few queries.

ADS+ (SIMS) is the fastest method across the board. The speed-up is more pronounced when the complete dataset fits in main-memory, where we are able to prune at a per data series level. This is because data are transferred from main memory to the CPU in cache-lines, which are much smaller than the data series size. Consequently, we have fine control over the data series that are transferred to the CPU: these are only the data series that need to be processed. On the contrary, in the case of large dataset sizes that exceed the main memory capacity (i.e., 10M and above in our experiments), we are only able to prune at a per disk-page level. Since disk pages fit more than one data series, we end up wasting a considerable amount of time on reading and transferring from disk data series that are not needed.

We observe that the benefit of ADS+ (SIMS) in absolute numbers increases with the dataset size. This is depicted in Figure 22(d), where we plot the amount of additional time that all methods need in order to (index the dataset and) answer all the queries in the workload, when compared to ADS+ (SIMS). For the 10M dataset, Serial Scan, iSAX 2.0, and ADS+ respectively need 2.1x, 2.4x, and 7.4x more time than ADS+ (SIMS), which completes the task within 12.7 minutes.

For the 1B dataset, Serial Scan needs 7 hours more than ADS+ (SIMS) in order to produce the results; iSAX 2.0 and ADS+ required more than 60 extra hours, at which point we stopped their execution.

Our experimental evaluation also shows that, even if Serial Scan has zero initialization cost, ADS+ (SIMS) very quickly outperforms it, after only a few queries (refer to Table 3). With a dataset of 100K data series, Serial Scan becomes slower than ADS+ (SIMS) if we want to answer 7, or more, queries. Moreover, the relative benefit of ADS+ (SIMS) increases with the dataset size: for the datasets with more 100M data series, ADS+ (SIMS) is faster than Serial Scan after answering merely 3 queries. As a result, ADS+ is the best option in all cases, even when analysts need to answer only a few queries.

According to our complexity analysis of Section 5, being able to answer queries faster than the Serial Scan, when including the indexing cost as well, means that we are very far away from the worst case scenario, efficiently pruning large parts of the raw dataset.

7 Conclusions and Future Work

In this work, we show that state-of-the-art data series indexing approaches cannot cope with the data deluge. The time needed to build a data series index becomes prohibitive as the data grows, and may take more than 24 hours to index a collection of 1 billion data series.

We propose the first adaptive indexing approach, where the index is built incrementally and adaptively, resulting in a very fast initialization process. Both the shape of the tree index and the leaf sizes are tuned adaptively and automatically to fit the workload on-the-fly. Using both synthetic and diverse real-life data, we show that our new adaptive indexing approach, ADS+, copes significantly better with the ever growing data series collections, and can answer up several thousands of queries during the time that state-of-the-art indexing approaches are still in the indexing phase. Moreover, we show that the proposed approach can be successfully used even in the case where we need to build the complete index at once.

Our long term goal is to integrate ADS+ into a general data series management system [38] in order to optimize similarity search, and support interactive exploration of big data series [65]. While our current implementation is limited to a single node scenario, ADS can be naturally parallelized, e.g., by distributing different sub-trees to different nodes of a cluster system [55, 59, 18]. Moreover, in order to ensure uniform utilization of the complete infrastructure, each node can host more than one sub-tree, including both hot and cold parts of the index. We also plan to study the performance characteristics of ADS+ using a structured approach for generating query workloads [66].

8 Acknowledgements

We would like to thank Prof. Volker Beckmann for providing us the Astro dataset [52].

References

1. Adhd-200. http://fcon_1000.projects.nitrc.org/indi/adhd200/, 2011.
2. QualiMaster A configurable real-time Data Processing Infrastructure mastering autonomous Quality Adaptation – Deliverable D1.1: Initial Use Cases and Requirements. Technical report, QualiMaster Project, 2014.
3. Sloan digital sky survey. https://www.sdss3.org/dr10/data_access/volume.php, 2015.
4. Incorporated Research Institutions for Seismology – Seismic Data Access. http://ds.iris.edu/data/access/, 2016.
5. D. Achakeev and B. Seeger. Efficient bulk updates on multiversion b-trees. *PVLDB*, 6(14):1834–1845, 2013.
6. R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *FODO Conference*, 1993.
7. I. Assent, R. Krieger, F. Afschari, and T. Seidl. The TS-tree: efficient time series search and retrieval. In *EDBT*, 2008.
8. J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.
9. S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *VLDB*, 1996.
10. Y. Bu, T. wing Leung, A. W. chee Fu, E. Keogh, J. Pei, and S. Meshkin. Wat: Finding top-k discords in time series database. In *SDM*, 2007.
11. A. Camerra, T. Palpanas, J. Shieh, and E. Keogh. iSAX 2.0: Indexing and mining one billion time series. In *ICDM*, 2010.
12. A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. Keogh. Beyond One Billion Time Series: Indexing and Mining Very Large Time Series Collections with iSAX2+. *KAIS*, 39(1):123–151, 2014.
13. K.-P. Chan and A.-C. Fu. Efficient time series matching by wavelets. In *ICDE*, 1999.
14. V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: a survey. *ACM Computing Surveys*, 41(3):1–58, 2009.
15. L. Chen, M. T. Özsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, 2005.
16. M. Dallachiesa, B. Nushi, K. Mirylenka, and T. Palpanas. Uncertain time-series similarity: Return to the basics. *PVLDB*, 5(11):1662–1673, 2012.
17. M. Dallachiesa, T. Palpanas, and I. F. Ilyas. Top-k nearest neighbor search in uncertain data series. *PVLDB*, 8(1):13–24, 2014.
18. C. du Mouza, W. Litwin, and P. Rigaux. SD-Rtree: A scalable distributed rtree. In *ICDE*, 2007.
19. C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, 1994.
20. T. M. Ghanem, R. Shah, M. F. Mokbel, W. G. Aref, and J. S. Vitter. Bulk operations for space-partitioning trees. In *ICDE*, 2004.

21. G. Graefe, F. Halim, S. Idreos, H. A. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *PVLDB*, 5(7):656–667, 2012.
22. G. Graefe, F. Halim, S. Idreos, H. A. Kuno, S. Manegold, and B. Seeger. Transactional support for adaptive indexing. *VLDB J.*, 23(2):303–328, 2014.
23. A. Guttman. R-Trees A Dynamic Structure for Spatial Searching. In *SIGMOD*, 1984.
24. F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5(6):502–513, 2012.
25. P. Huijse, P. A. Estévez, P. Protopapas, J. C. Principe, and P. Zegers. Computational intelligence challenges and applications on large-scale astronomical time series databases. *IEEE Comp. Int. Mag.*, 9(3):27–39, 2014.
26. S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, 2011.
27. S. Idreos, M. L. Kersten, and S. Manegold. Updating a Cracked Database. In *SIGMOD*, pages 413–424, 2007.
28. S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.
29. S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-stores. In *SIGMOD*, 2009.
30. S. Idreos and E. Liarou. dbtouch: Analytics at your fingertips. In *CIDR*, 2013.
31. S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011.
32. H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 33(1):117–128, 2011.
33. K. Kashino, G. Smith, and H. Murase. Time-series active search for quick retrieval of audio and video. In *ICASSP*, 1999.
34. S. Kashyap and P. Karras. Scalable kNN search on vertically stored time series. *KDD*, 2011.
35. E. Keogh, K. Chakrabarti, and M. Pazzani. Dimensionality reduction for fast similarity search in large time series databases. *KAIS*, 3(3):263–286, 2000.
36. E. J. Keogh and M. J. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In *KDD*, 1998.
37. J. Lin, E. Keogh, and S. Lonardi. A symbolic representation of time series, with implications for streaming algorithms. In *DMKD*, pages 2–11, 2003.
38. T. Palpanas. Data series management: The road to big sequence analytics. *SIGMOD Rec.*, 44(2):47–52, 2015.
39. T. Palpanas, M. Vlachos, E. J. Keogh, and D. Gunopulos. Streaming time series summarization using user-defined amnesic functions. *TKDE*, 20(7):992–1006, 2008.
40. T. Palpanas, M. Vlachos, E. J. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *ICDE*, pages 339–349, 2004.
41. D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. In *SIGMOD*, pages 13–25, 1997.
42. T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *KDD*, 2012.
43. T. Rakthanmanon, E. J. Keogh, S. Lonardi, and S. Evans. Time Series Epenthesis: Clustering Time Series Streams Requires Ignoring Some Data. *ICDE*, 2011.
44. U. Raza, A. Camerra, A. L. Murphy, T. Palpanas, and G. P. Picco. Practical data prediction for real-world wireless sensor networks. *IEEE Trans. Knowl. Data Eng.*, accepted for publication, 2015.
45. S. Richter, J.-A. Quiane-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead static and adaptive indexing in hadoop. *VLDBJ*, 2013.
46. P. Rodrigues, J. Gama, and J. Pedroso. Hierarchical clustering of time-series data streams. *Knowledge and Data Engineering, IEEE Transactions on*, 20(5):615–627, 2008.
47. S. Rogers. Big data is scaling bi and analytics, 1 Sep 2011.
48. F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *PVLDB*, 7(2):97–108, 2013.
49. D. Shasha. Tuning time series queries in finance: Case studies and recommendations. *IEEE Data Eng. Bull.*, 22(2):40–46, 1999.
50. J. Shieh and E. Keogh. iSAX: Indexing and Mining Terabyte Sized Time Series. In *KDD*, 2008.
51. J. Shieh and E. Keogh. iSAX: disk-aware mining and indexing of massive time series datasets. *DMKD*, 19(1):24–57, 2009.
52. S. Soldi, V. Beckmann, W. Baumgartner, G. Ponti, C. R. Shrader, P. Lubiński, H. Krimm, F. Mattana, and J. Tueller. Long-term variability of agn at hard x-rays. *Astronomy & Astrophysics*, 563:A57, 2014.
53. M. Stonebraker. The case for partial indexes. *SIGMOD Rec.*, 18(4):4–11, 1989.
54. M. Vlachos, D. Gunopulos, and G. Kollios. Discovering similar multidimensional trajectories. In *ICDE*, 2002.
55. J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi. Indexing multi-dimensional data in a cloud system. In *SIGMOD*, 2010.
56. X. Wang, A. Mueen, H. Ding, G. Trajcevski, P. Scheuermann, and E. J. Keogh. Experimental comparison of representation methods and distance measures for time series data. *DMKD*, 26(2):275–309, 2013.
57. 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):793–804, 2013.
58. T. Warren Liao. Clustering of time series data - a survey. *Pattern Recognition*, 38(11):1857–1874, 2005.
59. Y. Xie, D. Palsetia, G. Trajcevski, A. Agrawal, and A. N. Choudhary. SILVERBACK: scalable association mining for temporal data in columnar probabilistic databases. In *ICDE*, 2014.
60. L. Ye and E. J. Keogh. Time series shapelets: a new primitive for data mining. In *KDD*, 2009.
61. B. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary lp norms. In *VLDB*, 2000.
62. J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, 2003.
63. J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD*, 2004.
64. K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *SIGMOD*, 2014.
65. K. Zoumpatianos, S. Idreos, and T. Palpanas. RINSE: interactive data series exploration with ADS+. *PVLDB*, 8(12):1912–1923, 2015.
66. K. Zoumpatianos, Y. Lou, T. Palpanas, and J. Gehrke. Query workloads for data series indexes. In *KDD*, 2015.