Indexing for Interactive Exploration of Big Data Series

Kostas Zoumpatianos
University of Trento
zoumpatianos@disi.unitn.it

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

Themis Palpanas
Paris Descartes University
themis@mi.parisdescartes.fr

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, which 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. The main idea is that instead of building the complete index over the complete data set up-front and querying only later, we interactively and adaptively build parts of the index, only for the parts of the data on which the users pose queries. The net effect is that instead of waiting for extended periods of time for the index creation, users can immediately start exploring the data series. We present a detailed design and evaluation of adaptive data series indexing over both synthetic data and real-world workloads. The results show that our approach can gracefully handle large data series collections, while drastically 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), adaptive data series indexing has already answered $3*10^5$ queries.

1. INTRODUCTION

Big Data 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. 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 time t_i in which this recording was made. A common characteristic is that analysts need to examine the sequence of values (i.e., the data series) rather than the individual points independently. 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. In this way, there has been a significant interest in the data management community towards analyzing data series in real time with the minimum possible processing and storage overhead.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2610498 .

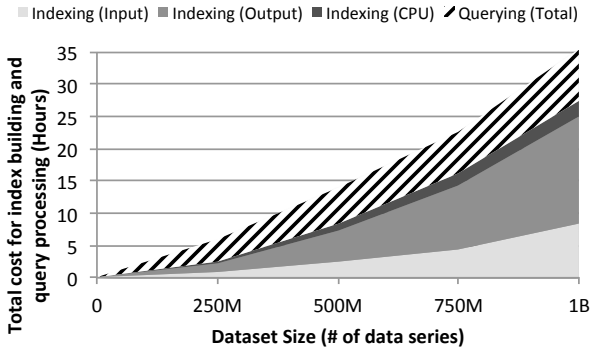


Figure 1: The data to query gap: building a state-of-the-art index and answering 10^5 queries for big data series collections.

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 techniques 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 [6]) 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 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 showstopper for many applications both in businesses and in sciences. 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, e.g., [15, 19]. 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. We propose an adaptive indexing solution 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. 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.
- Furthermore, 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.
- Through a detailed experimental evaluation with both synthetic and diverse real-world workloads, we show that it is possible to drastically reduce the data to query time, being able to handle several hundreds of thousands of queries by the time that state-of-the-art data series (iSAX 2.0 [6]) and multi-dimensional (R-trees [13], X-trees [4], KD-Trees [3]) techniques are still in the index creation phase.

2. RELATED WORK

In this section, we briefly discuss related work and we introduce the necessary background knowledge on adaptive indexing and on state-of-the-art data series indexing.

Similarity Search in Data Series. One of the most basic data mining tasks is that of finding similar data series in a database [1]. 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 [33], classification and deviation detection [5, 9]. A common approach for answering such queries is to perform a dimensionality reduction technique (the particular choice is of small importance [24]) such as *Discrete Fourier Transforms (DFT) [1], Discrete Wavelet Transforms (DWT)* [8], *Discrete Haar Wavelet Transforms (DHWT)*, *Piecewise Aggregate Approximation (PAA)* [22, 34], or *Symbolic Aggregate approXimation (SAX)* [23] and then use this representation for indexing. At the same time, a large set of indexing methods have been proposed for this kind of representations, including traditional multidimensional [13, 4] and specialized [29, 30, 6, 2, 32, 7] indices. Moreover, various distance measures have been presented that work on top of such indexes, e.g., Discrete Time Warping (DTW) and Euclidean Distance (ED).

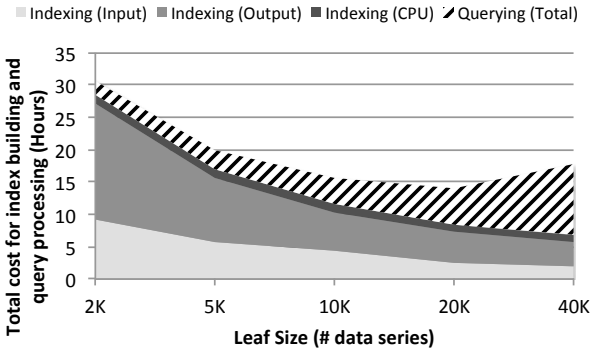


Figure 2: The indexing to querying trade-off: bigger leaf sizes improve indexing speed, but penalize query answering times.

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 [17, 16, 18, 20, 14, 28, 11, 12]. The intuition is that instead of building database indexes up-front, 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 [27].

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 [18] 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 [20, 28] 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 [18].

Data Series Representations and the iSAX Index. We now discuss the state-of-art data series indexing schemes. In 2000, Yi and Faloutsos [34], as well as Keogh et al. [22], both independently presented the idea of segmented means [34] or Piecewise Aggregate Approximation (PAA) representation [22]. 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 4 equal parts. Based on PAA, Lin et al. [23] 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) [29]; 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 [29, 30]. 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 [6]; it is a data series index that implements fast bulk loading. Figure 3 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 3) 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.

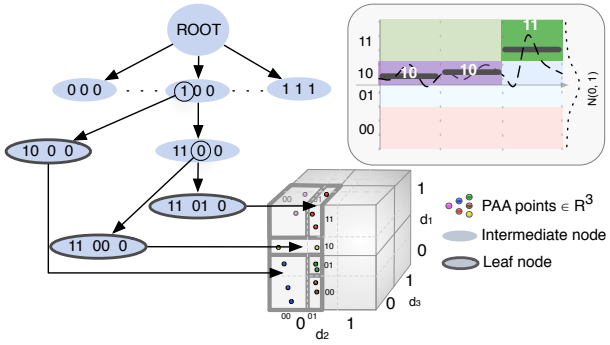


Figure 3: An example of iSAX and SAX representations.

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.

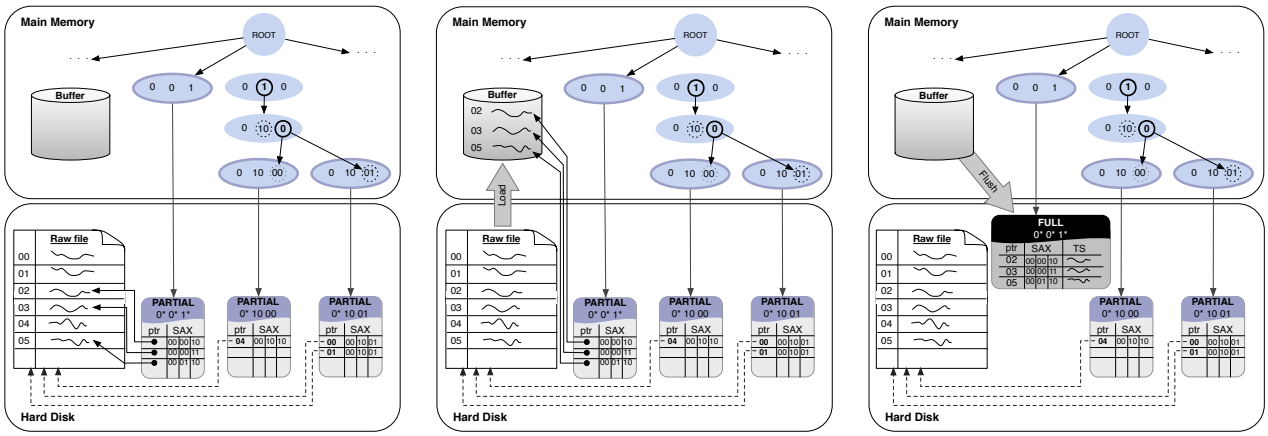
Scans Vs Indexing. Even though recent studies have shown that in certain cases sequential scans can be performed very efficiently [26], such techniques are only applicable when the database consists of a single, long data series, and queries are looking for potential matches in small subsequences of 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

In this section, 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 with good response time. 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



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

Figure 4: The ADS index states.

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 interesting and 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 parts 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 19), set each leaf to be in PARTIAL mode which means that we do not store any raw data series in this leaf (line 20). 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

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 layer)
9          // to the appropriate Leaf Buffer (LBL layer)
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 (isax, position) pairs to
17         // the disk, and set them in PARTIAL mode (no raw data)
18         for every leaf in index do
19             Flush the LBL buffer of this leaf to the disk;
20             Set leaf to be in PARTIAL mode;
21         clear buffers;
```

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 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

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;
```

to improve locality in tree-based indexes, e.g., [35, 6], or even in database query plans (which typically have a tree shape) [36]. 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.

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. An example of ADS is shown in Figure 4; the figure 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 4(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 4(b) and Figure 4(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.

Searching the Index. 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. Whether such a leaf exists already or not, depends not only on the data, but also on past queries. 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.

Enriching the Index. 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 4(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 4(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.

Creating the Answer. 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 Best So Far (BSF) answer. If the BSF is not 0, which means that we did not get an exact match, then the node with potentially the best possible answer has to be identified. This is done in a recursive way as in the original iSAX index using the MinDistPaaToiSAX as described in [29] and until we are not able to improve BSF anymore. The difference is that if a new leaf is needed which is in partial mode, ADS will enrich this leaf on-the-fly.

Example. Continuing the example of Figure 4, Figure 4(b) and Figure 4(c) show what happens when a query arrives. Figure 4(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 4(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 go-

Algorithm 3: 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 4: 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 leaf distance between the dataSeries
7  // and the raw data series that this leaf refers to or contains.
8  bsf = calculateRealLeafDistance(targetLeaf, dataSeries);
9  return bsf;
```

ing 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 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 3. Additionally both Approximate and Exact search have been modified to use this policy, a shown in Algorithm 4 (lines 2-5) and Algorithm 5 (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. An example of this process is shown in Figures 5(a) and 5(b). Figure 5(a) depicts the state of ADS+ after initialization and before any query has arrived, while Figure 5(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 and 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, 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 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.

Algorithm 5: exactSearchADS+(**dataSeries**, **index**, **queryTimeLeafSize**)

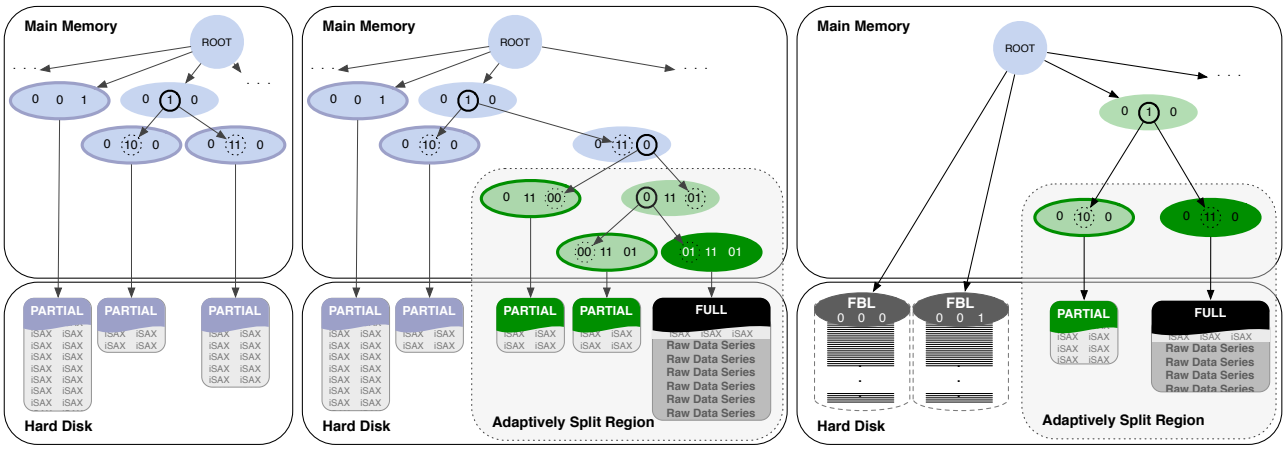
```
1  isax = convert dataSeries to iSAX;
2  bsf = approxSearchADS+(dataSeries, isax, index,
   queryTimeLeafSize);
3  bsfDist = Infinite;
4  queue = Initialize a priority queue with the root nodes of the index;
5  while node = pop next node from queue do
6  |  if node is a leaf and MinDist(dataSeries, 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 6: 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;
```

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



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

Figure 5: Examples of ADS+ and PADS+.

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 [31] 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. 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 buffer stored on disk. It then loads the iSAX representations found in this bucket and adaptively splits it, until the query-time leaf size is reached. It loads the raw time series for the corresponding leaf and answers the query using the approximate search of iSAX [29, 30, 6]. If an approximate answer is not good enough, it follows the regular ADS+ Exact Search algorithm, performing adaptive splits every time that the distance to a leaf node that has not yet been split to the query-time leaf size has to be calculated.

Every time that a query needs data which might be missing from the tree, it needs to scan the appropriate FBL buffer iSAX file and perform an adaptive split operation on it. To optimize this, 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. Further on, 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. An illustration of the PADS+ index can be seen in Figure 5(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 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 5(c).

3.4 Updates

Updates are also handled in an adaptive way.

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 new data series belongs in a leaf which is already in FULL mode, then we simply flip a bit in this leaf so that future queries know that more data exists. In either case, no further actions are needed for partial leaves. If a future query reaches a FULL leave with pending in-

serts, then it fetches the new inserts on-the-fly and it merges them in the leaf in the same way it is done for PARTIAL leaves (as we discussed earlier).

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.

4. EXPERIMENTAL EVALUATION

In this section, we present a detailed experimental evaluation. We demonstrate that adaptive data series indexing for an initialization time that is significantly lower than state-of-the-art approaches, drastically reducing the data-to-query time by one order of magnitude. 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 [6] that supports bulk loading, as well as against sequential scans and state-of-the-art multi-dimensional indexes such as R-Trees [13] and X-Trees [4]. In addition, we performed certain memory usage optimizations for iSAX 2.0; we use an LRU buffer for recently queried nodes and also after loading we maintain its last loading buffer in memory.

Infrastructure. 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 set such as they make maximum use of all available memory.

Benchmarks. We use synthetic benchmarks for a fine grained analysis as well as real-life benchmarks to demonstrate the usefulness of adaptive data series indexing. We generate synthetic datasets using 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 [1, 10, 25, 2, 29, 30, 6], and has been shown to effectively model real-world financial data [10]. 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, while the query workload is 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 both query intensive workloads as well as updates and various workload patterns including skewed workloads.

4.1 Reducing the Data to Query Time

Motivation. In our motivation discussion in the introduction section of the paper, 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 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).

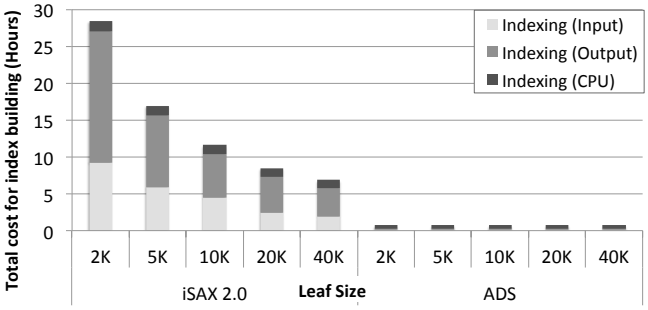


Figure 6: Reducing indexing costs.

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 6 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 6 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 7 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 7. 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 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 7, 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.

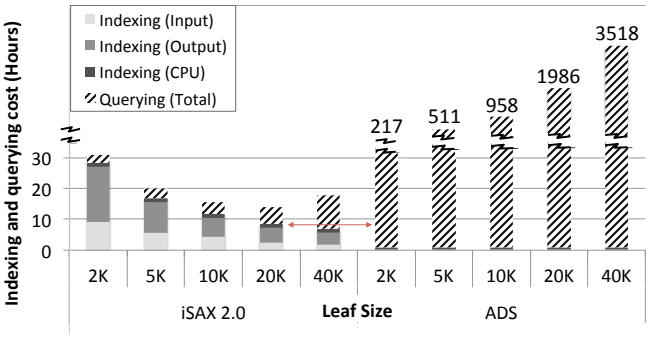


Figure 7: The query processing bottleneck.

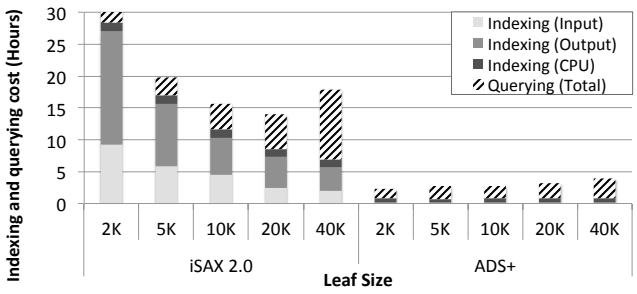


Figure 8: Reducing the data-to-query time with ADS+.

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 8 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) 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 9, 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-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-

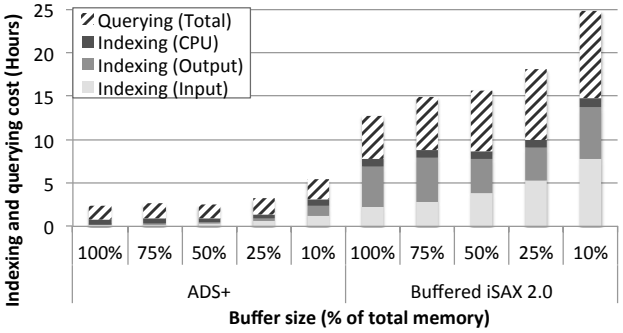


Figure 9: Total indexing and query answering cost as we increase the buffer size for ADS+ and iSAX 2.0.

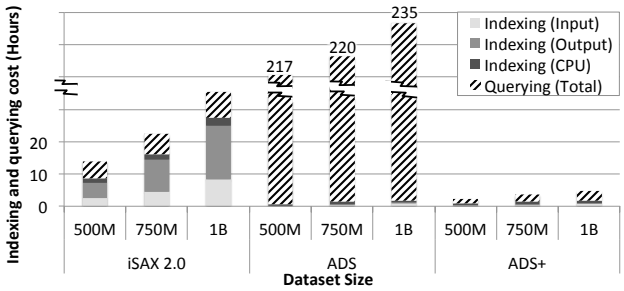


Figure 10: Scaling to 1 billion data series.

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.

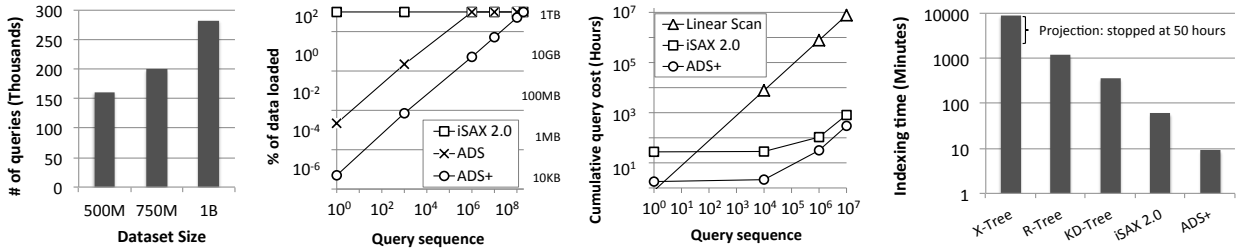
Query-time leaf size	1	10	100	1000
Query time (millisecond)	11.27	67.64	499.95	4031.68
Number of 4KB pages	0.25	1.79	17.23	171.48
Number of 8KB pages	0.12	**0.89**	8.61	85.74
Number of 16KB pages	0.06	0.45	4.30	42.87

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 10 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 7*x* 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



(a) Queries answered by ADS+ while iSAX 2.0 is still indexing. (b) Percentage of data indexed. (c) Per query response time. (d) ADS+ against spatial indexes.

Figure 11: Reducing the data-to-query time with ADS+ as we scale to big data.

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 11 provides further insights. Figure 11(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 11(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 11(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 11(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 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 11(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).

4.2 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 [3], R-Tree [13], and X-Tree [4], 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 11(d) depicts the time needed to complete the index building phase for each index. Overall, both data series

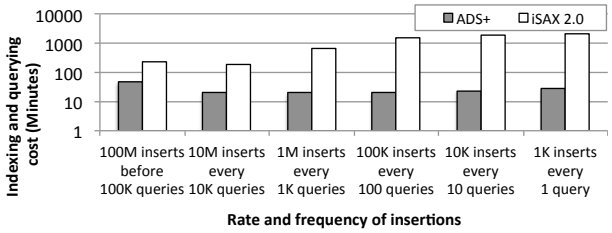


Figure 12: Robust ADS+ performance in update workloads.

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 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 scale to big data, index building is the main bottleneck and thus X-Tree is prohibitively expensive.

4.3 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 12 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.

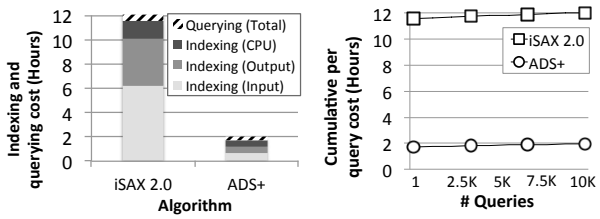
In all cases, ADS+ maintains its drastic performance advantage over iSAX 2.0. When all data arrives up front, the cost is naturally higher; more data has to be queried. For the rest of the cases where data arrives incrementally, interleaving with queries, we observe that when data arrives more frequently 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, ADS+ performs deletions with an average deletion time of only 0.2 milliseconds.

4.4 Real-life Workloads

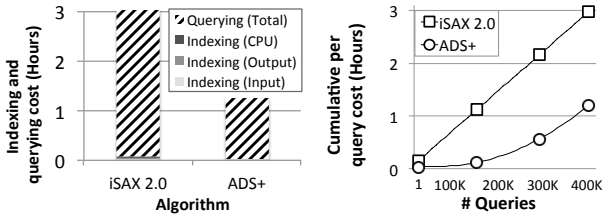
Here, we demonstrate the ability of ADS+ to drastically reduce the data-to-query time in real-life scenarios. In all cases, we use the optimal settings found in the synthetic benchmarks: for iSAX 2.0 uses a leaf size of 20K data series, while ADS+ uses a build time leaf size of 2K data series which adaptively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

Texmex Corpus. The first real-life scenario is an image analysis scenario from the Texmex corpus [21]. This dataset contains 1 Billion images which are translated into a set of 1 Billion data series



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

Figure 13: Indexing 1 Billion images (SIFT vectors) and answering 10^4 queries.



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

Figure 14: Indexing 20 Million DNA subsequences from the Homo Sapiens genome and answering $4 * 10^5$ queries.

(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 13 shows the results. Figure 13(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 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 13(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. 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.

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.

Figure 14 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.

4.5 Providing Fast Insight 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.

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.

5. CONCLUSIONS

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 an adaptive indexing approach, where the index is built incrementally and adaptively. 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 copes significantly better with the ever growing data series collections, and can answer several thousands of queries in the time that state-of-the-art indexing approaches are still in the indexing phase.

6. REFERENCES

[1] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *FODO*, pages 69–84, 1993.

[2] I. Assent, R. Krieger, F. Afschari, and T. Seidl. The TS-tree: efficient time series search and retrieval. In *EDBT*, pages 252–263, 2008.

[3] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[4] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *VLDB*, pages 28–39, 1996.

[5] 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*, pages 449–454, 2007.

[6] A. Camerra, T. Palpanas, J. Shieh, and E. Keogh. iSAX 2.0: Indexing and mining one billion time series. In *ICDM*, pages 58–67, 2010.

[7] 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.

[8] K.-P. Chan and A.-C. Fu. Efficient time series matching by wavelets. In *ICDE*, pages 126–133, 1999.

[9] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: a survey. *ACM Computing Surveys*, 41(3):1–58, 2009.

[10] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.

[11] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *PVLDB*, 5(7):656–667, 2012.

[12] 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.

[13] A. Guttman. R-Trees A Dynamic Structure for Spatial Searching. In *SIGMOD*, pages 47–57, 1984.

[14] 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.

[15] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, pages 57–68, 2011.

[16] S. Idreos, M. L. Kersten, and S. Manegold. Updating a Cracked Database. In *SIGMOD*, pages 413–424, 2007.

[17] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, pages 68–78, 2007.

[18] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-stores. In *SIGMOD*, pages 297–308, 2009.

[19] S. Idreos and E. Liarou. dbtouch: Analytics at your fingertips. In *CIDR*, 2013.

[20] 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.

[21] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 33(1):117–128, 2011.

[22] E. Keogh, K. Chakrabarti, and M. Pazzani. Dimensionality reduction for fast similarity search in large time series databases. *KAIS*, 3(3):263–286, 2000.

[23] J. Lin, E. Keogh, and S. Lonardi. A symbolic representation of time series, with implications for streaming algorithms. In *DMKD*, pages 2–11, 2003.

[24] 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.

[25] D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. In *SIGMOD*, pages 13–25, 1997.

[26] 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 *SIGKDD*, pages 262–270, 2012.

[27] S. Richter, J.-A. Quiane-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead static and adaptive indexing in hadoop. *VLDBJ*, 2013.

[28] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *PVLDB*, 7(2):97–108, 2013.

[29] J. Shieh and E. Keogh. iSAX: Indexing and Mining Terabyte Sized Time Series. In *SIGKDD*, pages 623–631, 2008.

[30] J. Shieh and E. Keogh. iSAX: disk-aware mining and indexing of massive time series datasets. *DMKD*, 19(1):24–57, 2009.

[31] M. Stonebraker. The case for partial indexes. *SIGMOD Rec.*, 18(4):4–11, 1989.

[32] 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.

[33] T. Warren Liao. Clustering of time series data - a survey. *Pattern Recognition*, 38(11):1857–1874, 2005.

[34] B. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary lp norms. In *VLDB*, pages 385–394, 2000.

[35] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, pages 405–416, 2003.

[36] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *SIGMOD*, pages 191–202, 2004.