Refined Indexing for Interactive Exploration of Big Data Series

Xingchi Dai
Harvard University
xdai@seas.harvard.edu

March 2016

1 Overview and Advisors

This proposal is for Xingchi Dai's Master of Engineering degree's thesis. Collaborated with DASLab and IACS, the project is under the guidances of Prof.Stratos Idreos and Dr.Pavlos Protopapas, led by Dr.Niv Dayan at Harvard University. Prof. Margo Selzer, who is my primary advisor, and also gives great academic advices.
In this proposal, I first present the motivation of the entire project. Then I compare different approaches found in the literature and describe potential problems might have occurred. Finally, in the end of this proposal, I give a holistic scientific justification with estimated schedule.

2 Motivation

The increasing need for developing technologies able to index and mine very large collections of data have shown in multiple scientific domains, including biology, astronomy, Internet, finance, weather forecasting etc. Among researches and applications, however, people do not focus on individual data points independently, instead, a sequence of data(such as real-time series) attracts people's attentions and reveals more effective information for us. Examples could be weather data, stock market data, astronomic data, etc.
For a large collection of data, however, it is impossible to sequentially scan the entire data for each single query in the interest of performance. Indexing in this case is required. On the other hand, it becomes a significant bottleneck. For example, to fully build a data set of 1 billion data series indexing, it took over a day[4]. Therefore,elaborated in the later sections, two major problems have occurred: deteriorated query processing performance and challenge of re-indexing when new data are added to given time series.
In recent years, many interesting algorithms and representations have been published. For example, in order to do a similarity time-series search, we can see that from the original ideas of dimensionality reduction techniques, including Discrete Fourier Transform (DFT), Discrete Wavelet Transform(DWT), Symbolic Aggregate Approximation etc., many advanced ideas, such as indexable Symbolic Aggregate Approximation (iSAX) and the most recent algorithm - Adaptive Data Series (ADS) came up many good ideas on effectively representing sequence data and building indexing for those large size data.
However, two problems in these solutions haven't been addressed in any literature. One problem is the hindrance of large size of RAM to achieve higher performance when computing. When buffers are created to accommodate data, a large size of RAM is needed, which is quite expensive. The second problem is that current indexing doesn't support or address the situation of adding new time series. Whenever a new data point is added, how should we rearrange the current index? As such, this project aims to solve these two problems.

3 Background and Related Works

Data Series Representations

1. **Piecewise Aggregate Approximation(PAA) and Symbolic Aggregate approXimation (SAX)**

   In 2000, the introduction of PAA[2] made dimensionality reduction technique available in time-series domain. A simple example from [1] [3]give a good illustration, shown below.

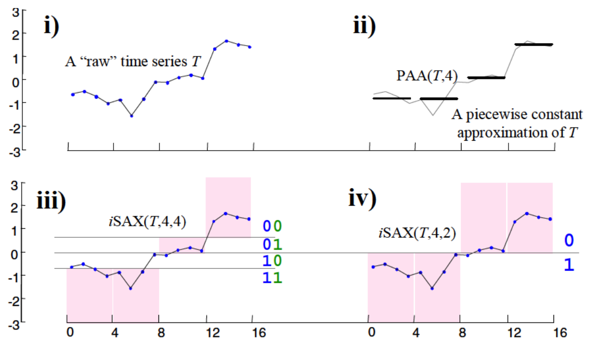


Figure 1: PAA and iSAX

   In the figure above, **i** represents a raw time series while **ii)** is its PAA form. In this example. the PAA form reduces the dimensionality down to 4, each of which is represented by a vector $\vec{t}$, calculated by:

$$\vec{t_i} = \frac{w}{n} \sum_{j=\frac{n}{w}(i-1)+1}^{\frac{n}{w}i} T_j$$

   where T is the time series having length of n, represented in w-dimensional space. The SAX takes the PAA representations as an input and discretize it into a small alphabet of symbols with a cardinality of size **a**, shown in **iii)** and **iv)**, respectively $a = 4$ and $a = 2$. Consequently, [3] uses the following representation to describe a time series.

$$SAX(T, w, a) = \mathbf{T^a}$$

   Binary numbers have been used. In the example above, with different cardinalities, we get $T^4 = (11, 11, 01, 00)$ and $T^2 = (1, 1, 0, 0)$
   It is obvious to see that T^2 is just the significant bits of T^4. Any representation could downgrade to lower cardinality. Euclidean distance then could be represented as the following formula.

$$MINDIST(T^2, S^2) = \sqrt{\frac{n}{w}} \sqrt{\sum_{i=1}^{w} (dist(t_i, s_i))^2}$$

2. **iSAX**

   [3] provides a slight change to the original SAX representation. Each iSAX representation has a superscript of cardinality to avoid ambiguity. It is this small changes that broaden the usage. By using iSAX, it is possible to compare two words of different cardinalities, and also, under iSAX, each word can have mixed cardinalities. This property makes the index construction a lot easier and efficient. [3] also points out the big drawback of the original representation, which caused a skew in the distribution when create index for data, and the largest file contains near 20 percent of the entire dataset, with more than half the files being empty. The solution to this problem is to set up a user defined threshold, *th*, which controls the maximum limit of time series in a file. For example, if our $th = 100$ and there are exactly 100 time series mapped to this word, to avoid the overflow, we split the file. To do this, the first symbol will be promoted to a higher dimension and change the file name accordingly. Based on this, we will get a hierarchical index structure with a controlled fan-out rate.Figure[2] summarizes the idea.

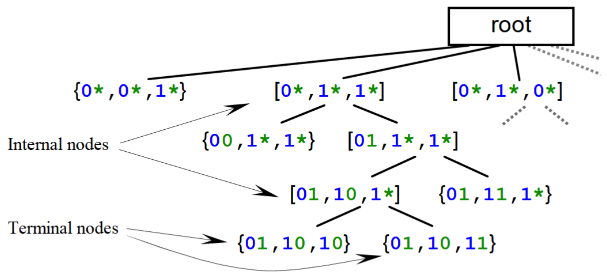


Figure 2: split iSAX by promoting cardinality

3. **iSAX2.0**

iSAX representation endures two major bottlenecks: time complexity of building the index and unnecessarily deep sub-trees due to the poor splitting policy. In [1], the bulk loading schema introduced reduced 72% the index building time by decreasing the number of total disk page accesses. The algorithm behind this bulk loading schema is to group the time series that will end up in a particular subtree and process those time series together. In order to achieve this, First Buffer Layer(FBL) and Leaf Buffer Layer(LBL) are used. The role of the buffers in FBL is to cluster time series that may end up in the same iSAX2.0 subtree, rooted in one of the direct children of the root. In contrast, LBL are used to gather all the time series of leaf nodes, and flush them into disk. How do those two layers work? Firstly, the time series are read and inserted in the corresponding FBL buffer. After this phase, time series stored in FBL will be moved to LBL sequentially. The algorithm creates FBL buffer for each specific leaf node.When all time series of a specific FBL buffer have been moved to the LBL, it will be flushed into disk. The figure below shows the procedural described above. In the figure above(right),

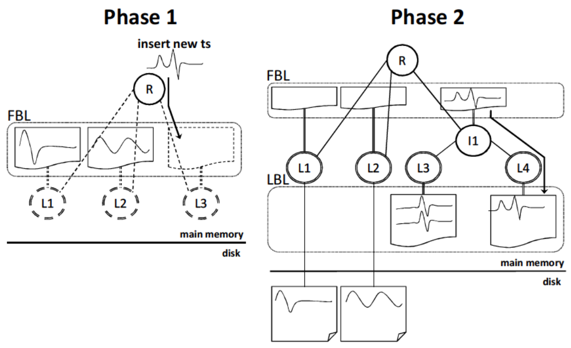


Figure 3: bulk loading algorithm

we can see that the first two nodes' buffers have been flushed into disk, and all available memories are for I1.

Another problem occurred in the iSAX representation is a too deep tree resulting from the undeveloped node splitting policy. in [1], a new node splitting policy based on knowledge of the distribution of the data stored in each node is introduced. Instead of examining for each segment the distributions of the highest cardinality symbols, this algorithm splits the node on the segment for which the distribution of the symbol having the highest probability to divide the time series into the two new nodes. Thus, this avoids the poor utilization of the leaf nodes and unnecessary deep index structure. The following figure explains this policy. For each segment, authors in [1] calculates $\mu \pm 3\sigma$. The algorithm behind is to check each segment the distributions of the highest cardinality symbols across the relevant time series, as much μ and σ here represent their mean and standard deviation. Then it splits the node which the distribution of the symbol indicating the high chance to divide the time series into two new nodes. As the figure[4] shown, the closer the μ is to the break point, the higher chance this segment

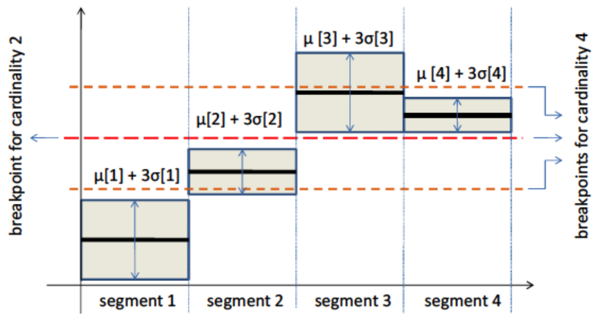


Figure 4: Node splitting policy example

will have equal distribution after splitting. [1] has confirmed the new policy could result in leaf nodes with an average of 54% more occupancy than before.

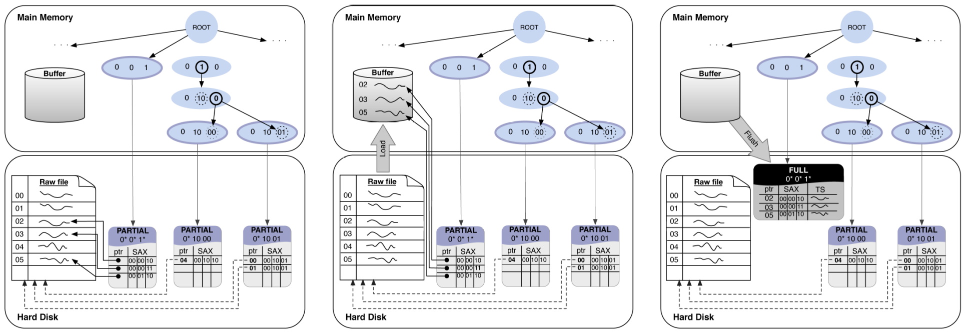
4. **Adaptive Data Series Indexing**

   Adaptive Data Series Indexing proposed in [4] minimizes the index creation time allowing users to query the data after its generation. Users do not need to wait until all index get created to work. As mentioned previously, the inability of current indexing to cope with very large data collections still exists. Users need to wait for a long time before sending query. The new ADS allows incremental, continuous, and adaptive index creation during query time. There are three types of ADS mentioned in [4], which will be summarized in the order decribed in [4].
   The essential idea of ADS is to only create trees containing representations for each data series. The actual data, however, remain in the raw files and are only loaded in an adaptive way when a relevant query arrives. As such, ADS avoids I/O costs at initialization time, and pays such cost at query time. There are two big parts: index creation and index refining.
   During the index creation phase, the leaf construction get delayed. No any data series is inserted. Thus, no splitting cost. During the indexing, I/O and memory bandwidth used is minimized. Also, ADS takes advantages of using buffers to improve locality. For example, during index creation, ADS keeps iSAX representations in the FBL. Only when FBL is full, those representations will be moved to LBL and then flushed into disk, as described in the previous section. No files will be loaded during this phase. So we have to map the raw data by adding a single pointer to the raw file. When leaf nodes contain no data, they are labeled “PARTIAL”.

   During the refining phase, when a new query arrives, it will be converted to an iSAX representation. Then the index tree will look for a similar leaf. If the leaf is in PARTIAL mode, all missing data series will be fetched from the raw data. All data series that belong in this leaf will be read. As before, data series will be kept in the LBL buffers and then flushed to disk. After that, the leaf goes into “FULL” mode. Figure[5] gives a clear visual explanation of the approach stated above.

5. **ADS+ Index and Updates**

   It has been noticed by authors of [4] that sizes of leaves affect index creation and search later on. As such, in ADS+, an adaptive leaf size has been used. ADS+ uses two different sizes of leaf: a big one for index construction and a small one for query time. When a query arrives, ADS+ refines its index structure by recursively splitting the target leaf, until the target sub-leaf becomes smaller or equal to the query-time leaf size.
   Regarding updates, when new data series arrives, we just add new series into raw files and update its iSAX representation and positions into the tree. For deletion, just mark the data series as deleted in its leaf. So the future queries will ignore the data can future insertions can exploit the space.



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

Figure 5: The ADS index states

4 Scientific Justification

There are a couple of questions which haven't been addressed in any literature so far. This project will focus on these three problems stated below.

1. **How to modify the index when new data come in?**

   Many works of literature have addressed how to update the indexing for the situations when new time-series come. In [4], authors give the adaptive way to handle new coming series, just simply appending the new data series in the raw file. However, none of this literature mentioned how to handle the case when new data points are added to the current series. They are two different levels questions, as when new data points come in, the representations we have discussed will need to be modified. Consequently, this re-arranging will become another bottleneck. How to modify as little as possible to the current, established index? This is the first problem the project will need to discuss.

2. **Transformation on time-series to increase the performance of building indexing**

   When we build the indexing, we divided them into segments and found an efficient way to represent and to discretize them. However, in some cases when "time" is not important and people care much more about the shape. We could actually shift time series left or right to find matched time-series.In this project, I will discuss the option of having some basic transformation to time series without affecting the important properties to improve the indexing built.

3. **RAM Size Limitation**

   As in the previous section stated, different kinds of buffers were used in the application. [1][4] both mentioned that the buffer can be flushed into the disk after full. However, the size of RAM is still a potential problem. For example, the size of buffer is 2^w, where w here is dimension. When the dimension increase significantly, we need much more RAM size. In my proposed project, I will also do researches on how to optimize the RAM usage to improve the performance.

5 Schedules

The whole project shall be finished within three semesters, 2016-Spring, 2016-Fall, and 2017-Spring. The major research work should be finished before 2017 Spring semester and the write-up should be done during 2017 Spring semester.

The detailed milestones will be determined after the project is approved by the Committee.

References

[1] Alessandro Camerra et al. "iSAX 2.0: Indexing and mining one billion time series". In: *Data Mining (ICDM), 2010 IEEE 10th International Conference on*. IEEE. 2010, pp. 58–67.

[2] Eamonn Keogh et al. "Dimensionality reduction for fast similarity search in large time series databases". In: *Knowledge and information Systems* 3.3 (2001), pp. 263–286.

[3] Jin Shieh and Eamonn Keogh. "i SAX: indexing and mining terabyte sized time series". In: *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2008, pp. 623–631.

[4] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas. "Indexing for interactive exploration of big data series". In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 1555–1566.