PAST: Processing and Storage of Time series

Eleni Tzirita Zacharatou, Jasmina Malicevic, Nikolaos Kokolakis, Eric Beguet, Puneet Sharma, Saurabh Jain, Mihaela Turcu, Nicolas Tran, Thomas Mühlematter

Introduction

A time series represents a collection of organized data obtained from sequential measurements over time. In order to extract some meaningful information out of that data and interpret the observed values, data mining of time series can be performed, which consists of extracting knowledge from the shape of the time series data. Today, time series data and time series data mining is essential for many applications since both scientific datasets but also data related to demographic, finance etc. is tracked through time.

However efficient mining of such data is a problem mainly because of its high dimensionality. This makes research in this area of particular interest and recently there have been many publications related to efficient mining of time-series data.

The goal of this project was to build a framework that would offer efficient solutions to storing, retrieving and mining time series data.

This document will give a brief overview of the components of the system as well as specific implementation details of individual components and a discussion on the challenges and possibilities for further development.

Framework overview

Being that the system is meant to process large datasets we decided to build it on top of Spark [1] due to its support for fast real-time queries and built-in methods for iterating over the data and mapping objects to programs. Another reason is that Spark supports loading data from an external file and storing it into a RDD object.

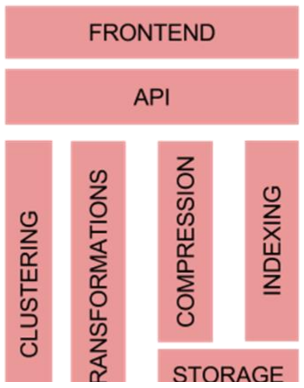


Figure 2 System Components

An overview of the components of our system is shown in Figure 1 and we will describe each of them in the rest of the document.

The system is designed to be user friendly and enable easy application development on top of the framework.

The user loads the data via the command line. Once the data is loaded it is stored in the system and the user can perform transformations, compression algorithms or indexing on such data. After these transformations and data organization, the user can write an application using the underlying structure.

Storage

A time series is defined by a timestamp which identifies one row and corresponding values for that timestamp. Due to the nature of the data, for one time series there is more correlation amongst the consecutive values in a column than the values in the same row. We decided to implement a column store by storing the different values (columns) in separate files and the timestamps in a file of its own.

A diagram of the storage system is provided in Figure 2. A time series is defined with a Schema that enforces constraints on the number of columns and their types and pointers to the files containing the columns.

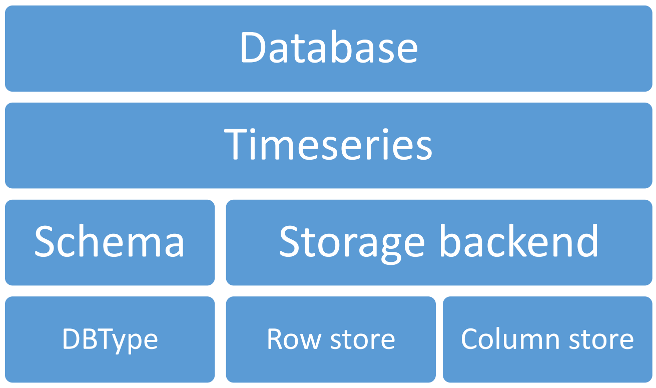


Figure 1 Storage system

The benefit of the data being spatially ordered is that we don't need to read the timestamps every time, all we need is its timeframe defined by start and end timestamps as well as an ε which represents the difference between two consecutive timestamps.

Another built-in feature of the storage system is support for range queries. This is implemented so that parts of the column (range) can be stored in multiple files in order to enable splitting these pieces across multiple machines and operations on such intervals can be done in parallel. In such cases the column files are identified by adhering to the following naming convention: *name{Part_Identifier}* (f.e. “price0”, “price1” etc.)

The main reason for implementing this feature is the way Spark handles range requests. Namely, the function that takes a part of an RDD object*, take(n: int),* converts the entire data to an array before returning the data to the user. By splitting the dataset into intervals, we will need to perform the conversion to an array for the first and last column of the data range in order to perform the *take* function on the data. The same is valid for *drop(n:int).*

Implementation of interval split

How is the split performed and how does it affect operations such as inserting a new time series? The mapping to these files is stored in an interval file whose format is shown in Figure 3. The Identifier mentioned above that is appended to the name of the time series is the value mapped in the interval file to a certain range.

```
Data :
      time ---- value
      1            a
      2            b
      ..
      1000         c
      ..
      2000         d

Index file :
      Interval     Identifier
      1 - 1000           0
      1001 - 2000        1
      2001 - 3000        2
      ...
```

Figure 3 Interval (index) file structure

The splitting range is not done according to some fixed interval but rather trying to balance the number of entries per file.

When loading a time series into the system the values are split by creating the Interval file according to the values of the timestamp column.

A drawback of this approach is slower insertion of new time series into the data store due to the splitting of data.

If the user knows he does not perform range queries, or this optimization is not of importance for him we provide the method *insertWithNoSplit* which does not provide the separation of values into multiple files.

Row Store

As an addition to the system we added support for row-store in case the user finds that for a particular application that will be more beneficial.

This is completely transparent to the user. The type of storage is chosen when the time series is created and the system later on uses the type of store provided for that time series without any further intervention from the user.

Transformations

Due to the high dimensionality of such data there is always a need to minimize the dimension and remove noise in the data. The data is usually preprocessed in order to remove the noise or create a representation of the data that reduces its dimensionality. In addition to this, performing transformations over the data shows gain in storage and speedup. Any new representation seeks to fulfill as much as possible of the following requirements:

- Significant reduction of data dimensionality
- Emphasis on fundamental shape characteristics on both local and global scales
- Low computational cost of the new representation
- Ability to create a good reconstruction from the reduced representation
- Insensitivity to noise in case the representation itself is not already meant to reduce noise

Thus data transformation can be defined as follows:

Given a timeseries T=(t1,…,tn), construct a model $\bar{T}$ of reduced dimensionality $\bar{d}$ ($\bar{d} \ll n$) such that $\bar{T}$ closely approximates T. More formally $|R(\bar{T}) - T| < \varepsilon_r$, $R(\bar{T})$ being the reconstruction function and ε_r being an error threshold. [2]

Our system supports transformations that create a new representation of the time series but also transformations that perform basic computation over the data.

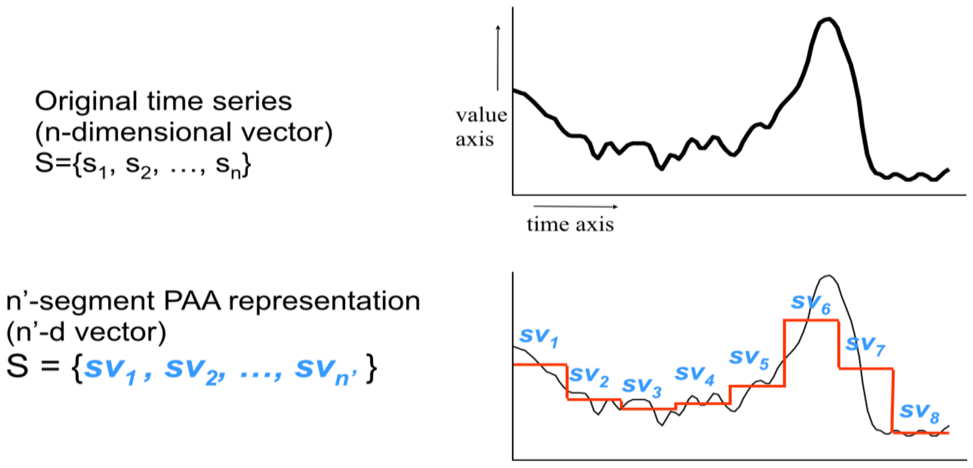


Figure 4 PAA Representation of timeseries

Power Transformations
Even though simple, a power transformation can be very effective as a way of stabilizing the variance across time. Data will become more normally-distributed and less skewed. We are using the square root and the logarithmic transformations.

Mean/Mode/Range/Subtract Mean/Standard Deviation/Normalization
These are necessary functions used to compute information about time series, as their name says, generally used in other transformations as well. The subtract mean filter can be used to remove noises with low frequencies. All these functions and transformations can be performed on complete data, or on partial subsets. As data can be observed periodically, it can present more variance for different periods. Thus it is generally useful to compute the standard deviation on subsets of periods, and create a new, normalized time series.

Moving Average Smoother
A moving average linear filter is generally used in time series to smoothen short-term fluctuations on data, and highlight longer trends and cycles. Having a fixed subset size, each entry in the time series is recomputed by averaging a fixed subset of the series. For an item, the average is computed by using both items before and after the current one, except for the first and last element in the series. Thus, for each element the subset will be different, but the resulting values will be closer.

Given a time series *X*, we create the new time series *Y*. For the new series, its *m*-th value represents the average of *X(m)*, the first k values before m, and the following k values after m, where k is the fixed subset size. The resulting series will be smoother than the original, as consecutive values of Y will have many common values from *X* in their computed average.

Dynamic Time Warping Distance:
For computing the similarity measure, we had to take into account that time series may not overlap exactly with respect to time, and that there can be more similarities if we consider a shifting in time. The dynamic time warping distance is generally used for measuring similarity between two temporal sequences, as it gives better results than a simple distance measure, such as Euclidian distance. When computing the similarity measure of two series, we are allowed to extend time sequences by repeating values.

Shift and Scale
A degree of similarity between two time series can also be achieved if one can be scaled and shifted such that it is shaped in a form closely resembling the other. These transformations can be local, on pairs of subsequences, and in this case, the two time series are similar if they share enough pairs of contiguous similar subsequences. The transformations can as well be directly global.

Piecewise Aggregate Approximation (PAA)
This transformation [3] reduces the dimensionality of time series data". Then you say that the segments are disjoint and modeled using regression. To my knowledge they are not modeled using regression (but they are indeed disjoint). Remove the thing about regression. The idea is to have a fixed frame size, and minimize dimensionality by using the mean values on each frame. If we consider a time series X, and a frame size k, the new time series Y will be composed of length(X) / k values, which represents the average on each of the frames. The segments are disjoint and modeled using regression. Figure 4 shows an example of the approximation of a given time series.

It is considered a non-data adaptive representation being that the intervals are fixed regardless of the type of the data and provides a foundation for the next type of transformation.

Symbolic Aggregate Approximation (SAX)
Taking as input the reduced time series obtained using PAA, it discretizes it into a predefined alphabet of symbols, with a given cardinality. The range obtained from PAA values is split into c subsets, where c is the chosen cardinality. Each subset of values is mapped to a different symbol, where the symbol in our case is a natural number. The SAX [4] representation changes every entry in the time series with its corresponding symbol.

Figure 5 shows the symbolic representation of a time series of length 16 with cardinalities of 4 and 2. This representation will be further explored when building an index based on the symbolic representation of this data.

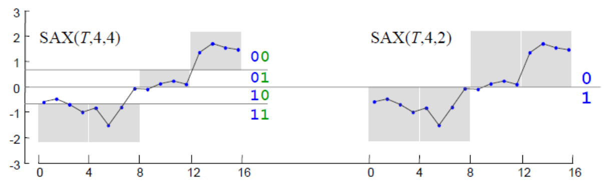


Figure 5 SAX representation of timeseries

Compression
The most obvious step when it comes to processing big data is reducing the amount of data. In the sense of spatial data series, compression is a means of transforming that data such that the total amount of data being stored is reduced. Our goal was to find a compression scheme supporting the characteristics of data obtained from a simulator over time, where data points that are close have similar values. An additional requirement we found crucial for a compression scheme is fast decompression speed as the data will be frequently accessed and we wanted to be able to also perform queries on the compressed data.

Compression using Piecewise Linear Regression (PLR)
The data is divided into a series of disjoint segments such that each segment is a good representative of the containing data and then we try to fit a polynomial model to each segment.

For each of the segments we store:

- Start time, End time
- Minimum and maximum values of the segment
- Model coefficient

After creating this model we don't store raw data anymore but use the stored variables and the model in order to do further computations.

In order to form this data structure we have opted for two approaches: Bottom-up and Sliding Window.

In Figure 6 we show the process of compression when using the *bottom-up approach*. The idea is to treat every two adjacent points as a segment and keep merging adjacent segments in an agglomerative fashion until a threshold is reached. This threshold is defined as a mean absolute error allowed in modelling a segment.

Into this bottom up approach we integrated the Adaptive Piecewise Constant Approximation (APCA) model where we take the average of the time segment as the model. The reason for implementing this is that it enables similarity queries for time.

This is an offline method being that it is aware of the global view of the data while building the model.

In contrary to this, compressing using a *sliding window* approach is an online model where we start with a fixed window size which we keep expanding until a

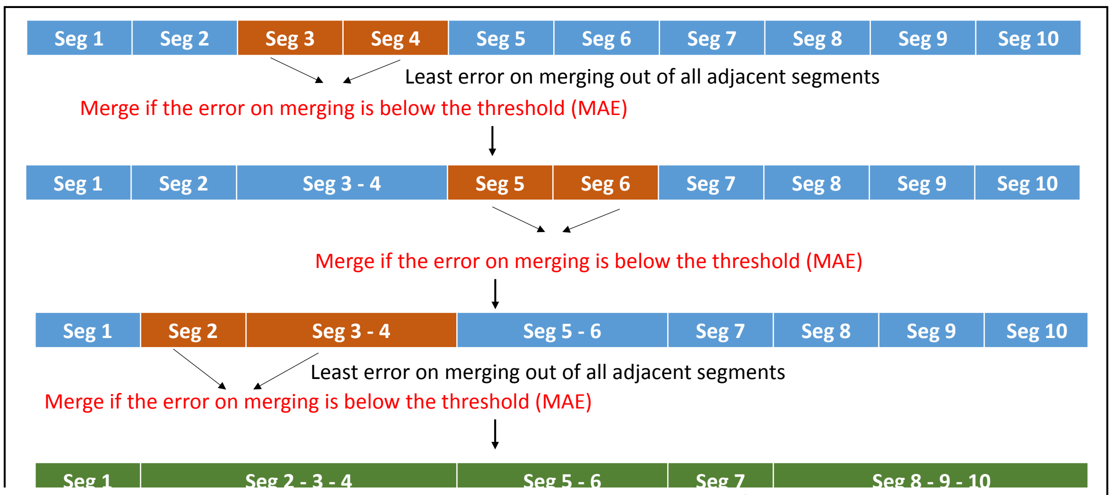

Seg 1
Seg 2
Seg 3
Seg 4
Seg 5
Seg 6
Seg 7
Seg 8
Seg 9
Seg 10
Least error on merging out of all adjacent segments
Merge if the error on merging is below the threshold (MAE)
Seg 1
Seg 2
Seg 3 - 4
Seg 5
Seg 6
Seg 7
Seg 8
Seg 9
Seg 10
Merge if the error on merging is below the threshold (MAE)
Seg 1
Seg 2
Seg 3 - 4
Seg 5 - 6
Seg 7
Seg 8
Seg 9
Seg 10
Least error on merging out of all adjacent segments
Merge if the error on merging is below the threshold (MAE)
Seg 1
Seg 2 - 3 - 4
Seg 5 - 6
Seg 7
Seg 8 - 9 - 10

Figure 6 Compression: Bottom-up approach

certain threshold is reached. The process is aware only about local data.

This technique of creating a polynomial model of the data using a user defined N as the degree of the curve enables us to answer queries without having to worry about the missing values which can be reconstructed.

only its timestamp value. This type of queries are common enough and as a result an interval index is used on the timestamp field.

The index is implemented as a disk interval tree that stores associations of ranges of timestamps to entry row numbers.

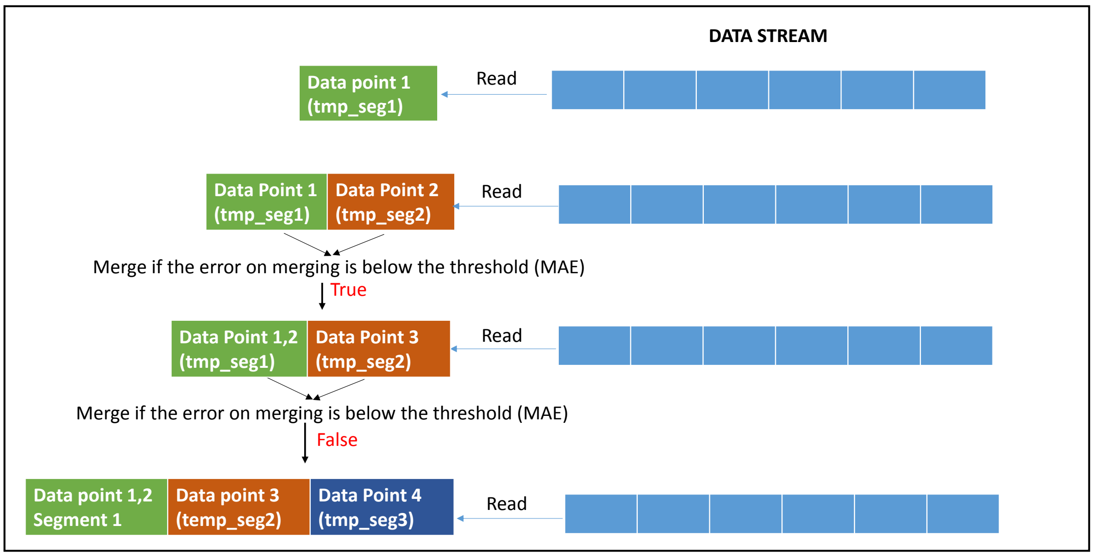


Figure 7Compression: Sliding Window approach

The degree of the curve along with the compression type (regression or APCA) and the maximum mean error can be tuned by the user in order to impact performance. A full list of tunable options can be obtained on the Wiki page of the project.

Query support

The model-view representation of the data enables efficient query processing over the modeled data without the need for accessing raw data. The types of queries we were most interested of supporting are:

- Time point or time range queries: returns the *values* at certain time point or in a time range
- Value point or value range queries: returns *timestamps* for which the values are equal to the query or they match some predefined condition
- Composite queries: returns a key-value pair of timestamp and value and performs a search for a match in both dimensions: time and value

The first step to answering such queries is finding the qualifying segments (segments whose timestamp/value correspond to the range defined in the query). We have implemented an index to support this.

As the timestamp values are not uniformly distributed, it is not possible to retrieve a timeseries entry using

The interval index is optional and it requires that the timestamp of each new entry added is older than any of the timestamps of the existing entries.is process.

Once these segments are identified the data can be extracted in two ways depending on the query: the time point queries can efficiently be performed by just using the model coefficients whereas the other queries can be supported by sampling on all qualifying segments in parallel. In the later case each worker node does sampling and filtering on its own partition and returns the required output.

The sampling granularity can be defined by the user.

Implementation

The underlying framework as for the rest of the system was Spark.

For the bottom-up approach we took advantage that the modelling has a global overview of the system and partitioned the data so that processing of each partition was done in parallel.

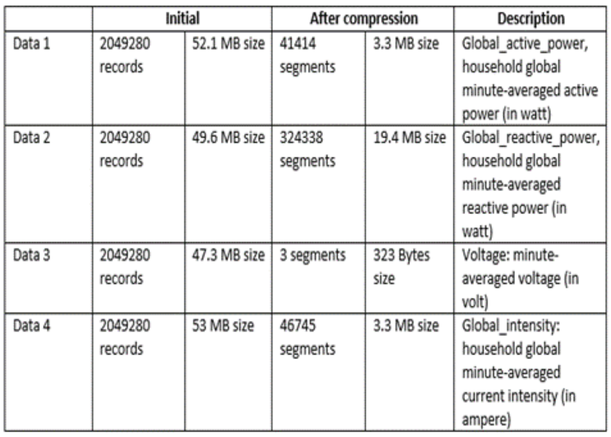
We first opted to use the MLlib [5] but it turned out that it was not compatible with the approach we wanted to implement for bottom-up. The reason was that MLlib requires parallelized objects to work upon but Spark does not allow the creation of parallel objects while working on partitions if the number of partitions is greater than the number of working nodes.

Therefore for the bottom-up approach we used Weka [6], a machine learning library for creating regression models.

In order to support the sliding window model we needed to adapt the algorithm to Spark. Spark allows only extracting the first N elements from a parallelized object hence making arbitrary access to a sub-section of data impossible.

We were able to mitigate this behavior by applying the sliding window approach sequentially, partition by partition followed by converting the subsections of a partition into parallelized objects forwarded to MLlib.

The compression ratio for a test dataset representing individual household electric power consumption for a duration of 4 years using the bottom-up approach is displayed in Table 1.



	Initial		After compression		Description
Data 1	2049280 records	52.1 MB size	41414 segments	3.3 MB size	Global_active_power, household global minute-averaged active power (in watt)
Data 2	2049280 records	49.6 MB size	324338 segments	19.4 MB size	Global_reactive_power, household global minute-averaged reactive power (in watt)
Data 3	2049280 records	47.3 MB size	3 segments	323 Bytes size	Voltage: minute-averaged voltage (in volt)
Data 4	2049280 records	53 MB size	46745 segments	3.3 MB size	Global_intensity: household global minute-averaged current intensity (in ampere)

Table 1 Compression ratio and data size before and after compression

Indexing

A crucial component to every big data mining system is the support of efficient retrieval of the data.

In addition to adding support for efficient range queries in the storage layer of the system we implemented two types of indexes for similarity queries.

Similarity search is of fundamental importance for a variety of time series analysis and data mining tasks. Consider stock time series, one may expect having queries like:

Query1: find all stocks which behave "similar" to stock A.

Or consider marketing time series:

Query2: find past sale patterns that resemble last month.

Or a scientific weather database:

Query3: find past days in which solar wind showed similar patterns to today's.

In order to compare two time series we cannot use exact match like in the case of string matching. We need to use a distance function to compare two time series. The most popular distance function is the L-2 distance.

There are two important kinds of queries that we would like to support in time series database, range queries (e.g., return all sequences whose distance is within an epsilon of the query sequence) and nearest neighbor (e.g., return the k closest sequences to the query sequence). The brute force approach to answering these queries, sequential scanning, requires comparing every time series Y to X. Clearly this approach is unrealistic for large datasets

A traditional approach for indexing time series in order to answer efficiently similarity queries is to use a spatial access method. A time series with n values can be considered as a point in a n-dimensional space. This is why it could be indexed by a Spatial Access Method, such as the R-Tree [7] . These methods partition the space into regions along a hierarchical structure for efficient retrieval. However, typical time series contain over a thousand data points and most SAM approaches are known to degrade quickly as dimensionality increases. As R-Trees and their variants are victims of the phenomenon known as "dimensionality curse", a solution for their usage is first to perform dimensionality reduction.

Even so, in order to perform dimensionality reduction, we cannot simply choose an arbitrary compression algorithm. It requires a technique that produces an indexable representation. For example, many time series can be efficiently compressed by delta encoding, but this representation does not lend itself to indexing. In contrast a representation like DFT lends itself to indexing, with each Fourier coefficient mapping onto one dimension of an index tree. In order to guarantee no false dismissals (i.e. qualifying objects are missed because they appear distant in index space) the distance measure in the index space must satisfy the following condition:

Dindex_space (A,B) <= Dtrue (A,B).

That is to say, we can define a distance measure on the reduced abstraction that is guaranteed to be less than or equal to the true distance measured on the raw data. The tighter the bound, the better. Ideally we would like Dindex_space (A,B) = Dtrue (A,B). Postprocessing is performed by computing the actual distance between sequences in the time domain and discarding any false alarms (i.e. objects that appear to be close in the index but are actually distant).

To sum up, efficient indexing for time series similarity can be achieved with the following three steps:

1) Establish a distance metric from a domain expert (in our case we use Euclidean distance).

2) Produce a dimensionality reduction technique that reduces the dimensionality of the data from n to N, where N can be efficiently handled by a SAM.

3) Produce a distance measure defined on the N-dimensional representation of the data that obeys *Dindex_space(A,B) ≤ Dtrue(A,B).* In this project we implemented the PAA dimensionality reduction of timeseries as well as SAX which is based on PAA.

We created multidimensional index structures based on these representations in order to be able to efficiently support similarity queries.

R-Tree Index

In PAA a time series is divided into equally-sized segments and the mean values of the data points that fall within each segment is recorded. Therefore, as the size of the segments is fixed, only one number per segment is required in order to approximate a time series, the number that records the mean value of all the data points in a segment.

The PAA representation defines a N-dimensional feature space. In other words, the proposed representation maps each time series S = {s1,...,sn} to a point S = {sv1,..., svN} in a N-dimensional space. Each aggregate segment maps onto one dimension of the index tree. We refer to the N-dimensional space as the PAA space and the points in the PAA space as PAA points. An example is give in Figure 4.

Data points in time series tend to be correlated with their neighbors (a phenomenon known as autocorrelation). This is why the PAA representation of the time series is efficient: as data points are correlated with their neighbors, we can efficiently represent a "neighborhood" of data points with their mean value.

A distance measure DR defined in the index space that has the lower bounding property described above and thus can guarantee no false dismissals is the following:

$$DR(\overline{X},\overline{Y}) \equiv \sqrt{\tfrac{n}{N}}\sqrt{\sum_{i=1}^{N}(\overline{x}_i - \overline{y}_i)^2}$$

Building R-Trees with Spark

A MapReduce-based algorithm was adapted for Spark and was used for building the R-Tree index structure in parallel fashion. To be exact, the index structure used is a variant of the classic R-Tree, the Priority R-Tree [8]. Existing Java code which was downloaded from the web (http://khelekore.org/prtree/) was used for building the Priority R-Tree index. As Spark requires the objects it manipulates to be serialisable in order to send them over the network to the Worker nodes, we had to slightly modify the code to make it serialisable.

The definition of the problem is as follows. Let T be a compressed time series data set composed of time series ts_i, i=1,..., |T|. Each time series ts has two attributes <ts.name, ts.values>, where ts.name is a unique identifier for a time series and ts.values is a vector containing the values of the compressed time series. This vector can be interpreted as one point in some multidimensional domain as it was explained in the previous section.

The proposed method consists of two phases which are executed the one after the other. First, the time series are partitioned into groups. Then, each group is processed to create an R-Tree for each partition. These phases are executed on a Spark cluster.

The two main phases of the algorithm are:

1. Computation of a *partitioning* function *f*. The inputs for this phase are the data set *T*, a sample quantity *L* and a positive number *R,* which represents the number of partitions. The purpose of *f* is to assign any object of *T* into one of the *R* possible partitions. The function is computed in such a way that applying *f* on *T* yields *R* (ideally) equally-sized partitions. In practice, minimal variance in sizes is acceptable. At the same time, *f* attempts to put objects that are close in the spatial domain in the same partition. The output of this phase is a function *f* which takes as input a time series' vector of values ts.values and outputs a partition number. Note that no actual partitioning or data moving happens at this point. The next phase utilizes *f* for such purpose.

2. *R-Tree construction*. During this phase, the function *f* calculated in the first phase is used by *Workers* to map each time series to its partition. Then the time series are grouped according to their partition, and *R Workers* build *R* independent R-Tree indices simultaneously on their input partitions. The output of this phase is a set of *R* independent R-Trees.

More details about these phases are provided in the following subsections.

Partitioning Function

The purpose of the partitioning function *f* is to provide a means for assigning objects of *T* to a pre-defined number of *R* partitions. We use the idea of mapping multi-dimensional spaces into an ordered sequence of single-dimensional values via space-filling curves for this purpose. More specifically, the Z-order curve [9] is used. The time series' vector of values ts.values is mapped into a Z-curve. The partition number of a time series ts is determined by *f*(*ts.values*), which evaluates to a value from the set {*1, 2, .., R*}. By using a space-filling curve, the partitioning function *f* achieves two goals:

- Generate *R* (almost) uniformly-sized partitions, and

- Preserve spatial locality. If two distinct time series ts_1 and tsp are close to each other in the spatial domain, then they are likely to be assigned to the same partition, i.e. *f*(ts_1*.values*) = *f*(ts_2*.values*).

The algorithm to define *f is as follows.*

The input data set *is partitioned* via *data sampling*. Given a data set *T* and target number of partitions *R*, the algorithm takes *L* sample time series from *T,* distributes them among *M Workers* (that is, each *Worker* samples L/M objects) and emits their single-dimensional values *S*={*U*(ts_i*.values*), *i=1, .., L*} given a space filling curve *U*. Then S is sorted, and a list *S´* of *R-1* splitting points that split the ordered sequence of samples into *R* equal-sized partitions is determined. Then, in general, a time series ts belongs to partition *j* if *S* [*j-1*] < *U*(ts.*values*) ≤ *S* [*j*]. Thus, *f* utilizes the splitting points in *S´* to assign objects to partitions.

Workers read in total *L* samples at random offsets of their input *T*, and compute their single dimensional value given the space-filling curve *U*. Key-Value pairs (ZCurveValue, C), C being a constant whose value is irrelevant, are produced. Such pairs are produced in order to use Spark capabilities to sort the keys into an auxiliary list *u1 , ..,uL*. Once sorted, the *Master* collects the *L* single-dimensional values generated by *Workers* from which *R-1* elements are taken starting at the (L/R)-*th* element and subsequently at fixed-length offsets L/R to form a list *S´* of splitting points.

The rationale of the splitting points in *S´* is that they provide good enough *boundaries* to sub-divide *T* into *R* partitions since they come from randomly sampled objects. Formally, the function is defined as shown in Figure 8.

R-Tree Construction

In this phase, *R* individual R-Tree indices are built concurrently. *Workers* map each time series to their partition using the partitioning function *f.* Then the time series are grouped by their partition.

$$\text{f(ts.values)} = \begin{cases} 1, U(ts.values) \leq= S'[1] \\ j, S'[j-1] < S', j = 2, \ldots R-1 \quad \text{j,} \quad \text{S'[j-1]} \\ R, U(ts.values) > S'[R-1] \end{cases}$$

Figure 8 Dividing T into R partitions

Subsequently, every partition is passed to a *Worker*, which independently builds an R-Tree on its input. Next, every *Worker* outputs its constructed R-Tree, so *R* R-Trees are written to the file system at the end of this phase.

Since *f* balances partitions, it is expected that all *Workers* will receive a similar amount of objects, thus executing similar amount of work in constructing their R-Trees. However, good balancing depends on the underlying space-filling curve *U* used by *f*, and the number of sampled time series *L*. More samples help in tuning the splitting points, but incur in larger sorting time of *L* elements.

Querying R-Trees with Spark

For retrieving N nearest neighbors of a given query time series q, each R-Tree is independently queried in parallel. The nearest neighbor method from the downloaded Priority-RTree library is invoked in each *Worker.* Each *Worker* will retrieve N nearest neighbors of q that are sent to the *Master*. That is, the *Master* receives N * R nearest neighbors of T. Then the *Master* sorts all the nearest neighbors according to their distance to T, and selects the N nearest neighbors.

iSAX

In order to support indexing over time series represented with SAX words and to support similarity queries the initial notation had to be extended.

Figure 5 denotes a SAX representation of a time series whose equivalent SAX word would be {10,11,01,00} (left) for the cardinality of 4 and {1,1,0,0} (right).

The time series would then be identified by writing the sequence as either strings or integers. ({2,3,1,0} for the first case) and marked as T(4,4) (Timeseries(cardinality, word_length)).

The problem is that two time series might have different cardinalities and this information cannot be retrieved from the integer representation.

The iSAX representation therefore stores the binary representation in order to make the cardinality visible at any point.

The reason for which we need to know the cardinality is that at some point we might have to compare two time series with different cardinalities. As noticed from the example above the values between breakpoints for higher cardinalities are a subset of the values of their lower counterparts. Knowing this allows us to derive the second word from a word represented using higher cardinality by removing the trailing bits of each symbol.

SAX(T,4,16) = T16 = {1100,1101,0110,0001}
SAX(T,4,8) = T8 = {110 ,110 ,011 ,000 }
SAX(T,4,4) = T4 = {11 ,11 ,01 ,00 }
SAX(T,4,2) = T2 = {1 ,1 ,0 ,0 }

Figure 9 Example of SAX words with different cardinalities

Another important reason will be explained in the following paragraphs.

In order to compare two SAX symbols we define a lower bounding approximation of the Euclidian distance as follows:

$$MINDIST(T^2,S^2) = \sqrt{\frac{n}{w}}\sqrt{\sum_{i=0}^{w}(dist(t_i,s_i))^2}$$

Here n represents the length of the time series (16 in this case) and w is the word size. The dist() function represents the distance between two breakpoints and is read-in from a lookup table. The lookup table is generated using code provided on the project webpage of [3]. A sample is given for cardinality of four in Table 2.

Table 2 Lookup table for cardinality 4

	00	01	10	11
00	0	0	0.67	1.34
01	0	0	0	0.67
10	0.67	0	0	0
11	1.37	0.67	0	0

In order to preserve the information about the cardinality, we have extended our initial implementation of SAX by storing it as an additional attribute associated with the integer representation of the symbol and then obtaining the binary representation of the numbers on index insertion.

The intuition for building an index on top of this representation is that similar time series will be represented by the same SAX word. Thus, when issuing a query we first transform it into its equivalent SAX word and search for its matching counterpart in the index.

When adding a file to an index, we simply add all the time series with the same SAX word representation to one index file.

The obvious drawback of this approach is that there may be many time series mapping to one file. If this happens and we issue a query that matches this file, then we still have to sequentially traverse through all the time series mapped to that file. In this case we have no gain with the index.

In order to deal with this situations, the additional information about cardinality is used in order to split the data from one file to two files by promoting the first symbol to a higher cardinality. This way, similar time series will still be close and the index is balanced. The promotion is further explained below.

A split occurs when the maximum amount of data stored in one index file is greater than a user defined threshold variable th.

Index Construction

The algorithm provided in [10] was implemented in Java as a tree index with small modifications to the index structure as well as adaptation to Spark and other components of the system.

As described in [10] there are three types of nodes:

- Internal Node: this node contains pointers to it's children and the initial SAX word before promotion. Every internal node was initially a terminal node with pointers to timeseries
- Root Node: It has the same behavior as the internal node, except it never contained any info about time series
- Terminal Node: The terminal node is the only type of node that contains *pointers* to time series that match the SAX word of it's preceding parent and the binary string obtained after promoting one symbol from it's predecessor.

The difference and benefit of our implementation is that the terminal node does not store the actual data, only the path to the time series that is being indexed. This way the index structure itself is not very large. This could be used for a future optimization of forcing the index structure to be cached in the cluster nodes so the traversal itself will be even faster.

Since our testing applications were comparison between subsequences of time series with a million entries, the index creation included storing an index file with the data of the subsequence. The file was stored in the same path as the original time series with the extension ".sax" and identifier of the part.

It is worth noticing that this causes some redundancy in data but the use case of indexing only a smaller parts of a bigger dataset will perform better if we retrieve that whole file which represents only the subsequence from disk sequentially than randomly reading parts of a larger file.

Ideally, as a future improvement, this should be an optional parameter depending on the application.

The tree nodes are kept as a key-value pair identified by the SAX word of the containing time series. Each terminal node has a list of pointers to the time series matching its id.

When inserting a time series to the index we first obtain its SAX representation. We convert the int symbols using the information about the cardinality to its binary representation. We try to obtain a terminal node with the matching key. In case such a node exist there are two options:

a) The threshold has not been reached yet so we just append information about the current time series to the node
b) The time series causes the node to split. When a split occurs, the first symbol is chosen and moved to higher cardinality by adding a trailing bit which is assigned in a round-robin fashion. All time series pointed to by this node are reassigned to the newly created terminal nodes and this node becomes an internal node.

In case we did not find a node that matches the iSAX representation of the data, we create a new terminal node and add it as a child of the root node.

In order for this functionality to be accessible to the rest of the system and to take advantage of spark, the interface takes RDD objects as input, one object for the timestamps and another for the matching values but the insertion itself is done in a sequential manner for all the entries in one rdd object. This however does not prevent the application user to insert multiple time series represented as RDD objects in parallel.

Querying the iSAX index

For many queries it will not be necessary to read the time series from disk. For example if we want to find the approximate distance between the query and the time series, it is enough to convert the query into a SAX word and if there is not a matching word, calculate the distance using the MINDIST function provided above. The system provides this by implementing a function that returns to the user this distance. This again can be invoked as a spark function over RDD objects and done in parallel.

In case the user has a need for an exact distance to the provided time series there is a more advanced algorithm that uses the fact that SAX was built on top of PAA. In the first step of indexing, instead of obtaining the SAX word for the query, we obtain the PAA representation. Then we calculate the distance among this representation and the node keys for the index tree. This provides us with a tighter lower bound and is calculated according to the following function:

$$MINDIST\_PAA\_SAX(T_{PAA}, S_{SAX}) = \sqrt{\frac{n}{w}} \sqrt{\sum_{i=0}^{w} \begin{cases} (\beta_{Li} - T_{PAAi})^2 \; if \; \beta_{Li} > T_{PPAi} \\ (\beta_{Ui} - T_{PAAi})^2 \; if \; \beta_{Ui} < T_{PPAi} \\ 0 \; otherwise \end{cases}}$$

Where n and w are length and word size respectively and β_{Li} is the lower bound of the particular segment and β_{Ui} the corresponding upper bound. These bounds are obtained from [10] .

After obtaining the node with the smallest distance to the query, all-time series corresponding to that node (or it's children if it is an internal node) are fetched and the Euclidian distances amongst the query and the time series are returned in order to find the minimal.

System parameters

There are certain parameters that can be set by the user and affect performance and result accuracy. The larger the cardinality, the higher will be the granularity of the data providing more accurate maps to symbols and hence increasing the accuracy of approximate searches.

Another important parameter that can be set is the threshold that determines when the node splits. Although, since we are not storing the actual time series in the node but only pointers, this threshold can remain high in order to prevent unnecessary splits.

Benefits of iSAX indexing

By traversing the tree trying to find the most similar word to the query we prune the dataset and perform expensive computation only on a subset of data. If we find a matching terminal node then the distance to the node is 0 and if we need to provide the exact distance then we are guaranteed to find the minimal in the time series mapped to that node. In case it is not the terminal node for which we have a match, we traverse trough the children of the node in order to find the terminal node which matches the most to the upgraded cardinality of the query by calculating the distance using the MINDIST or MINDIST_PAA_SAX functions.

Clustering

Clustering is used to categorize data into groups, which are not predefined, but rather defined by the time series data itself, based on similarity measures between time series. We use the affinity propagation algorithm [11], which considers all data points as possible centers - exemplars.

The algorithm utilizes a similarity matrix *S*, where *S(i,k)* indicates how well the data point with index *k* is suited to be the exemplar for data point *i*. The similarity matrix is constructed by computing the dynamic time warping distance between time series, and set to the negative value of the distance, as the purpose is to minimize the error.

Further, the algorithm mainly consists in updating two matrices: responsibility matrix and availability matrix. Messages passed between points reflect the current affinity that a given data point has for choosing another data point as its exemplar. The responsibility matrix *R* describes how appropriate is a data value to serve as the exemplar for another data. An entry *r(i,k)* of the matrix, shows how representative *k* is to be the exemplar for *i*, in relation to other candidate exemplars for *i.* The availability matrix *A* describes how appropriate would be for a value to pick another value as its representant. For an entry a(i,k) of the matrix, we compute how appropriate it would be for *i* to pick *k* as its exemplar, also considering the preference of other points for *k* as an exemplar.

The responsibility and availability matrices are initialized with zeroes, and further updated according to the formulas:

$$r(i,k) = s(i,k) - \max\{a(i,k') + s(i,k')\}$$

$$a(i,k) = \min\{0, r(k,k) + \mathrm{sum}(\max\{0, r(i',k)\}), \text{for } i \mathrel{!}= k$$

$$a(k,k) = \mathrm{sum}(\max\{0, r(i',k)\})$$

The procedure can be terminated when no change passes a given threshold, or after a selected number of iterations.

Unfortunately this feature has not been fully implemented in our system due to the lack of time. We have started writing a parallel algorithm of affinity propagation using map/reduce, based on [12], it is not yet complete and integrated. The idea was to break down into separate map/reduce jobs for the availability, responsibility and similarity matrices.

User interface

In order for the above components to be available to the user and make the application programming as easy as possible we have implemented a command line utility built by extending the Scala REPL [13] which allows mixing Scala variables into the system.

The idea was to create a friendly user interface providing functions calls such as those implemented in matlab. The User Interface was particularly hard to implement since it included intercommunication between the modules. The complications arose from the decision to code both in Scala and in Java. Even though Scala supports Java functionalities it turned out the opposite is quite difficult to implement. The storage interface was writen entirely in Scala but the rest of the system was done in Java so the command line module had to enable interoperability. We are still working on fully integrating in a user friendly way all the modules in the system.

To issue a query via the command line the user writes a pseudo-sql statement starting with a single quote (‘). The command is then converted to Scala and executed.

Some of the basic commands supported by the system are:

Time series creation

```
scala> 'CREATE timeseriesX (column1 INT, column2
FLOAT) BACKEND ColumnStore
```

Column selection

```
scala> 'SELECT column1, column3 FROM
timeseriesY WHERE column1 > 2 AND column1 < 3
```

Data insertion

```
scala> 'INSERT csv("path/to/file") INTO timeseries;
```

Transformation over a timeseries

```
scala> import past.Transformations
scala> 'SELECT @Transformations.mean(columnX)
FROM timeseriesY
```

The system supports also functions to manipulate the database, start/stop spark etc., find the minimum, maximum or mean value of the time series. We enable scaling, shifting and normalization through the transformations.

Another very interesting feature is allowing users to mix Scala variables into the commands and we will show an application of that in the following section where we describe our experiments.

Experiments

We performed a modest set of tests over the individual modules in order to explore their features.

We decided to test the similarity queries by converting a human DNA string to time series and find the smallest distance to a smaller DNA set belonging to an animal.

Our ideal test case was to use the range index to extract subsequences of both DNAs and then index the data using iSAX and the R-Tree Index. We will provide the description of some of those tests. The tests not included are those who were not entirely integrated into the system and are still in the debugging phase.

The human DNA contained ~1 million entries which were split into subsequences of 1024.

We have implemented an application performing this similarity search both in Hadoop and on top of our system when using no indexing. Then we have implemented this using the iSAX index in order to compare results.

In order to insert the DNA as time series without converting it by an external script we used the functionality of our system to mix Scala variables in the command line.

```
scala> 'CREATE humanDNA (encodedBase BYTE)
BACKEND RowStore

scala> val dna: Iterator[Byte] =
scala.io.Source.fromPath("path/to/dna").map({ case 'A' =>
2; case 'G' => 1; ...})

scala> 'INSERT @dna INTO humanDNA
```

The results we obtained by using our index showed that the execution time was less than when using the map/reduce implementation of the sequential scan but the index creation time added some overhead to the overall runtime.

Finally we noticed that when running with different cardinalities for iSAX we got slightly different results in the sense that the exact distance obtained was smaller than the distance when comparing to use-cases with smaller cardinalities. This is due to the higher granularity of the approximation.

Challenges

The project was a big challenge for all of us especially for our team leader. It was hard to coordinate 9 people and we experienced problems due to bad inter-team communication regarding sub-parts of the project.

Also, perhaps the choice that we made at the beginning to combine Java and Scala was not a very good one. It turned out to be a difficult obstacle to overcome in order to successfully integrate the project components into one system. This has been somewhat mitigated but the progress would have been much better had we taken another direction.

Future work

Building such a complex system is never straight forward and along the way some ideas arose of how it could be improved or even how some novel modules can be added.

One direction that might be interesting to follow is creating bitmap indices for indexing time series values. The idea would be as follows:

Let's say that our goal is to find "interesting" time series and, in them, "interesting" periods of time. More formally, we define a time series as a discrete set of observations $X = x_1, x_2, \ldots x_n$ at consecutive time steps $t = 1,2\ldots n$ (ordered in time). Given a massive set of time series XN and a query q with an upper and a lower

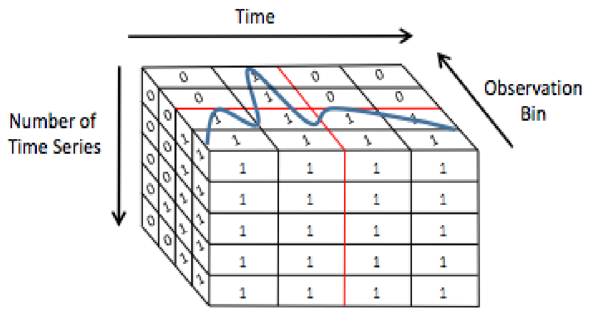


Figure 10 3D bitmap matrix

threshold o_u and o_l (with $o_u \geq o_l$) for the observation as well as an upper and lower bound t_u and t_l (with $t_u \geq t_l$) for time, we want to find all time series $X \in XN$ where x_t is between the two thresholds, i.e., $o_u \geq x_k \geq o_l$ and $t_l \leq t \leq t_u$.

One way to answer such a query is to use a classic bitmap index to index all the observations, then retrieve all the observations that are in the desired range using the index and lastly keep only those observations that fall in the desired time interval. The main limitation of this approach, is that a bitmap index which indexes all the values of all the time series of the data set will have a very large size. In order to control the size of the bitmap index, techniques like binning (i.e. discretization) and compression are typically used. Even so, the size of the index will be large for a time series database containing a huge amount of very long time series.

Time series frequently have considerable similarity between them. Additionally, consecutive observations in one time series tend to be correlated. These properties can be exploited for compressing the bitmap index. The main idea is that once we obtain a bitmap representation for each one of the time series, these representations can be grouped together in order to be jointly compressed. Such a group is presented in the following figure, where the bitmaps of the time series are stacked the one on top of the other to create a 3D bitmap matrix.

The compression is performed by representing a 3D chunk of the matrix where all the bits have the same value with just its position inside the matrix and its bit value, instead of separately storing every bit. This procedure is similar to run-length encoding where a list of consecutive identical bits is represented by its length and its bit value, but in this approach compression is performed along 3 dimensions. The first dimension is the time step, the second dimension is the observation bin, and the third dimension is the time series. To identify 3D chunks where all the bits are identical, the space can be hierarchically subdivided using e.g. quad tree decomposition, until all the bits in one chunk are identical.

This approach is easily parallelizable as time series are first clustered based on their similarity. Compressed bitmap indices, one for each cluster, can be built in parallel and then used to answer queries in parallel.

Another interesting idea is exploring the iSAX logic of indexing the data for a clustering algorithm. It is actually quite intuitive being that similar datasets map to the same SAX word and in that sense, the same way a SAX word identifies a tree node, it can identify a node in the cluster. This could be a data preprocessing step.

Last but not least, as mentioned in the text in the previous chapters, there are many ways to improve our existing components, such as allowing the user flexibility in whether to save the index files on disk or keep pointers to the original file or keep the index structure cached in the cluster nodes.

References

[1] M. C. M. J. F. S. S. I. M. Zaharia, "Spark: Cluster Computing with Working Sets," HotCloud 2010. June 2010..

[2] C. P. Esling, "Time Series Data Mining," *ACM Computing Surveys.*

[3] E. C. K. P. M. a. M. S. Keogh, "Dimensionality reduction for fast similarity search in large time series databases.," 2000.

[4] E. K. S. B. J.Lin, "A Symbolic Representation of Time Series, with Implications for Streaming Algorithms," June 13.

[5] "http://spark.apache.org/mllib/".

[6] "http://www.cs.waikato.ac.nz/ml/weka/".

[7] A. Guttman, "R-Trees: A Dynamic Inex Structure for Spatial Searching," in *SIGMOD*, 1984.

[8] M. d. B. K. Y. L. Arge, "The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree.," in *SIGMOD* , 2004.

[9] G. M. Morton, "A computer Oriented Geodetic Data Base and a New Technique in File Sequencing," IBM Ltd, Ottawa, Canada, 1966.

[10] J. K. E. Shieh, "iSAX: Indexing and Mining Terabyte Sized Time Series," *Proceedings of ACM SIGKDD ,* 2008.

[11] D. D. Brendan J. Frey*, "Clustering by Passing Messages Between Data Points".

[12] "Parallel Hierarchical Affinity Propagation with Mapreduce," [Online]. Available: http://www.academia.edu/6779337/Parallel_Hierarchical_Affinity_Propagation_with_MapReduce.

[13] "Scala REPL," [Online]. Available: http://www.scala-lang.org/old/node/2097.