Searching time series with Hadoop in an electric power company

Alice BERARD
TELECOM PARISTECH
46, Rue Barrault
75013 PARIS, FRANCE
alice.berard@telecom-paristech.org

Georges HEBRAIL
ELECTRICITE DE FRANCE - R&D Division
1, Av. Du Général de Gaulle
92140 CLAMART, FRANCE
georges.hebrail@edf.fr

ABSTRACT

In this paper, we investigate the possibilities offered by the Hadoop eco-system for searching time series in an electric power company (Top-*K* or range-queries based on a similarity measure). There has been much work done on speeding up the search of time series in a large dataset, mainly by designing efficient indexing techniques preceded by reduction techniques. In this paper, we do not follow these approaches but focus on using the brutal force of distributed computations in the Hadoop environment. We propose an implementation of time series search functions in Hadoop and describe experiments on a large database of electric power consumption curves (35M customers observed during 1 month at a 30' sampling rate). We also show that this architecture supports easily the computation of several distances for the same query with a small response time overhead: this is very useful in practice when the end-user does not know very well which distance to use.

Categories and Subject Descriptors
G.3 [**Probability and Statistics**]: Time series analysis;
H.4 [**Information Systems Applications**]: Miscellaneous;
H.2.8 [**Database Management**]: Database applications – *Data Mining.*

General Terms
Algorithms, Performance, Experimentation

Keywords
Time series, Similarity search, Hadoop

1. INTRODUCTION

With the upcoming installation of smart meters, millions of electric power load curves are bound to be recorded every day. Time series data will increase at an unprecedented speed in EDF. In this context, tools to store and process massive time series databases are needed. In particular, searching for similar patterns is a core element of many data mining applications.

However, electric power consumption is not the only use case of time series similarity search in EDF. For example, power plants monitoring via transient behaviors of physical signals provide amount of time series in which patterns research aiming at diagnostic aid is very interesting. Another use case is the grid frequency in power plants, which must be maintained constant. In this case, we are interested in finding drops in frequency in a very large data historian.

To handle these problems, the Hadoop eco-system is both convenient and efficient since it allows inexpensive storage and data-intensive distributed applications. This is well suited for information search. Indeed, Hadoop is based on two key elements, namely the MapReduce programming model and the Hadoop Distributed File System (HDFS), which are derived from Google's computing infrastructure.

Similarity search through dimensionality reduction and indexation has been well discussed in the literature. However, it is not the case for parallelization's brutal force. In this work we show that a parallel sequential similarity search is both fast and flexible. This paper contains the following contributions:

- Firstly, we propose a storage solution and efficient Hadoop user-defined functions with various parameters for time series search
- Secondly, we show their efficiency through a series of experiments with performance results dealing with single but also multiple queries
- Lastly, in case the end-user does not know very well which distance to use, we introduce a novel idea which consists in computing multiple distances and ranking the results according to some aggregation methods.

The rest of this paper is organized as follows. Section 2 defines the problem of similarity search in time series. Section 3 surveys the related work. In Section 4 we describe our general approach in Hadoop. Section 5 focuses on the multi-distances search. Section 6 shows the experimental results. Finally we draw conclusion in Section 7.

2. PROBLEM DEFINITION

Our work in this paper concentrates on univariate time series. This is the reason why we will not consider our datasets regarding transient behaviors of physical signals, as what is chiefly challenging in these datasets is to find various patterns located in several variables simultaneously.

Thus, we use here two main datasets: electric power load (i.e. consumption) curves and grid frequency in power plants. The first one is synthetic and contains daily load curves at a 30' sampling rate of 35M customers during a month, namely around a billion 48-dimensional time series. The second one is a sample subset from one power plant with measurements at a 1'' sampling rate during a month, namely 2.5M values.

Though these are particular examples, they are rather representative and can be easily generalized to other univariate sensor data.

BigMine'13, August 11-14 2013, Chicago, IL, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2324-6/13/08…$15.00.

We now define time series.

Definition 1: A *Time Series S* of length $|S| = n$ is a real numbers sequence: $S = S_1, S_2, \ldots, S_n$. We assume that the sampling rate, that is to say, the time step between each measurement, is regular and the same for all the time series we consider in the database.

Definition 2: A *Subsequence* $S_{i,k}$ of a time series S is a shorter time series of length k starting from position i.

Definition 3: The *Euclidean Distance* (ED) between two time series Q and S where $|Q|=|S|$ is defined as:

$$ED(Q, C) = \sqrt{\sum_{i=1}^{n} (Q_i - S_i)^2}$$

When available time series are very long (for instance the description of electric power consumption of a customer during one month or one year), we split them into consecutive segments of size n (for instance each segment describes one day with *n=48* in the case of a *30'* sampling rate).

The problem we focus on is the design of fast searching methods that search and locate sequences that approximately match a specific query sequence in a time series database.

Before clarifying what a query is, we need to define the dataset which is searched. We consider basically two ways to search such data, as illustrated in Figure 1 and 2:

- *Whole Matching*. The query sequence is compared only once to each candidate sequence. Therefore the query sequence "jumps" from one sequence to the next. All sequences must have the same length, *n*.

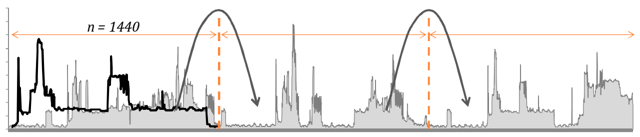


Figure 1: Whole matching (jumping window).

- *Subsequence Matching.* Here we have a smaller query sequence. We look for a subsequence in the large sequence that best matches the query sequence and report its offset. Therefore the query sequence "slides" along every sequence. In the rest of the paper, when considering subsequence matching, we will assume the query length is *l*.

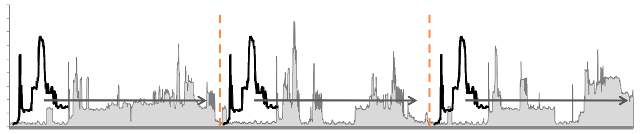


Figure 2: Subsequence matching (sliding window).

We now define a query. Similarity queries can be classified in two categories:

- *Top-K queries*. The *K* closest sequences to the query sequence are returned.
- *Range-queries*. All sequences within an epsilon of the query sequence are returned.

Lastly, we insist on the need for normalization. As stated by the author of [19], "*in order to make meaningful comparisons between two time series, both must be normalized*". This seems intuitive when we are interested in *similarity* in terms of shape but not in absolute values. Indeed, the Euclidean distance between two time series with a high mean discrepancy appear to be high even if the shapes of the time series are similar. It is the same for two time series with a high variance difference.

Definition 4: Let S be a time series, μ its mean and σ its standard deviation. Let i be in $\{1, \ldots, n\}$. The time series can be normalized by dividing by the mean ($\hat{S}_i = \frac{S_i}{\mu}$) or by standardizing ($\hat{S}_i = \frac{S_i - \mu}{\sigma}$).

3. RELATED WORK

Time series are all the more fundamentally high dimensional data since they are very long. Processing and storing large volume of them is very expensive. Thus, with the aim of dealing with large time series databases, many techniques have been proposed in the literature for representing time series with reduced dimensionality, such as *Discrete Fourier Transformation* (DFT) [1][8][23], *Single Value Decomposition* (SVD) [12][15][20], *Discrete Wavelet Transformation* (DWT) [3][22][23], *Piecewise Aggregate Approximation* (PAA) [12], *Adaptive Piecewise Constant Approximation* (APCA) [13], *Symbolic Aggregate approXimation* (SAX) [17][21], etc. In order to measure a *similarity*, these representations should be associated with a distance metric, like classic *Euclidean Distance* (ED) [1][8] or elastic measures such as *Dynamic Time Warping* (DTW) [14].

This dimensionality reduction might be used for two major reasons: firstly, with a view to smooth raw data and thus, to change the query criterion (the distance metric is computed over transformed time series); secondly, in order to create a multidimensional index structure to avoid sequential scans. In this case, dimensionality reduction is performed in order to keep indexing efficient by storing no more than 8-12 coefficients for each time series. Several index structures have been suggested: R-tree [10], R*-tree [2], TV-tree [16], etc. Note that an important property proved by [1] is that, in order to guarantee no false dismissals, the distance measure in the index space must satisfy the following condition:

$$D_{index\ space}(F(A), F(B)) \leq D_{true\ space}(A, B)$$

A and B are two time series, $F(A)$ and $F(B)$ are their transformations after performing dimensionality reduction. This is called the *lower bounding* property and allows range-queries to be performed on the index structure, provided we discard false alarms afterwards.

However, this approach has some drawbacks. On the one hand, we can only deal easily with range-queries since the aim of the index structure is to project the query in the index space and to retrieve all time series in a range ε. On the other hand, subsequence matching queries are quite complex to process with an index. Suppose we have a subsequence matching query of length ω. Faloutsos et al. [8] suggested storing *minimum bounding rectangles* (MBR) containing trails consisting of $n - \omega + 1$ points, one for each possible offset of the sliding window. Obviously, we would like to use the same index to handle queries of length greater than ω. [8] suggested methods based on prefix search, but none of them seems entirely satisfactory.

We opted for another approach: similarity search through a parallel sequential scan. To our knowledge, no one has ever considered this approach in a data mining paper. In [18], Qiao et al. dealt with parallel algorithms for range-queries and Top-*K* search, but only for whole matching, while JaJa et al. [11]

developed indexing techniques to efficiently handle querying multidimensional time series in a parallel implementation.

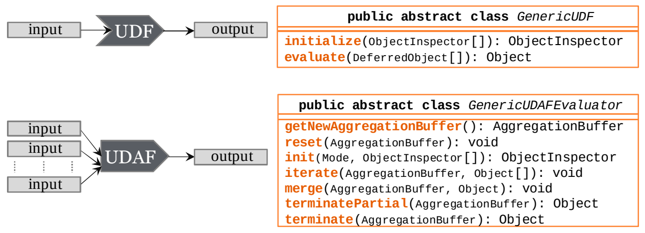
4. TIME SERIES SEARCH IN HADOOP

Our goal is to design and implement a Hadoop based solution to the problem of similarity search in a time series database.

Hadoop is an open-source software framework that supports data-intensive distributed applications. It is based on two key elements: the MapReduce programming model and the HDFS file management system (Hadoop Distributed File System).

We also use Hive, a data warehouse infrastructure built on top of the Hadoop ecosystem and suitable for analytical treatments and ad hoc queries with HiveQL, a SQL-based query language. Hive queries are transparently translated into MapReduce jobs.

Sometimes it might be impossible – or at least difficult – to express a query in HiveQL, especially as it is a simplified SQL version and does not allow all kinds of subqueries, multiple tables in the "where" clause, etc. This problem can be circumvented thanks to custom mappers and reducers that can be plugged in Hive. There are two ways to do so: either by developing streaming mappers and/or reducers scripts or by developing customized Java functions called user defined functions (see Figure 3). These functions can work row by row (User Defined Function, UDF) or they can aggregate several rows (User Defined Aggregation Function, UDAF). Hive allows two varieties of UDAFs: simple and generic. Simple UDAFs are simpler to write but incur performance penalties. Furthermore, they do not allow Generic UDAFs' features such as variable-length argument lists or complex types input and output.



```
public abstract class GenericUDF
initialize(ObjectInspector[]): ObjectInspector
evaluate(DeferredObject[]): Object
```

```
public abstract class GenericUDAFEvaluator
getNewAggregationBuffer(): AggregationBuffer
reset(AggregationBuffer): void
init(Mode, ObjectInspector[]): ObjectInspector
iterate(AggregationBuffer, Object[]): void
merge(AggregationBuffer, Object): void
terminatePartial(AggregationBuffer): Object
terminate(AggregationBuffer): Object
```

Figure 3: UDF and UDAF overridden methods summary.

4.1 Time series storage structure in HDFS

In a previous work at EDF R&D, we showed that Hadoop could be efficiently used to store and manage structured data from smart meters [5]. Different representation models, file formats, and compression methods for the storage were tested.

The default parameters were kept (text file format with default compression codec: .deflate) since they appeared to give good performance while preserving a high compression rate.

An array representation model for storing time series was also designed in [5] and is reused here. It takes advantage of a primitive type defined in Hive that is able to store multiple values of a same type. Thus, it has the advantage of retrieving an entire time series by reading a single record. The array's length is flexible and can be adapted to the sampling rate (i.e. time step): we might store minutely, hourly, daily, monthly, etc. time series.

This representation model is well adapted to any whole matching query of length less than or equal to the array's length. What about subsequence matching? If the time series in which we want to "slide" are of length less than or equal to the array's length, we do not encounter any problem. However, suppose the time series that we want to "slide" in are so long that they were split into several segments. How to "slide" in a split time series? At one point, we inevitably have to "slide" straddling on both a given segment of time series and its following segment. The idea to circumvent this problem is to have an overlap of at least twice the maximum subsequence matching query length. Thus, we need to have access to a new array of length 2*n*.

For example, as shown in Figure 4, by overlapping two days in the database, we are able to deal with any subsequence matching query of length less than or equal to one day.

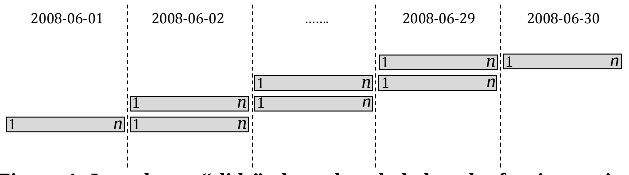


Figure 4: In order to "slide" along the whole length of a time series split up into arrays, we need to overlap at least two arrays.

We propose two ways to obtain a specific segment and its following one. The first one is to operate a self join. This is very efficient in terms of file size, but not in terms of query execution time, especially as partitioning by day would be useless and partitioning by client ID would be impossible (we have more than 35M customers). Besides, this storage allows more general queries since it is possible to add several segments behind the first one. Thus we can handle subsequent matching queries of any length. The other possibility is to adapt the storage by replicating the following segment at the end of each row. This is very efficient in terms of query execution time, but not in terms of file size since it nearly doubles the data.

4.2 Generic search functions

Based on this storage structure, we now describe the algorithms and how we parallelize them. We produce naive algorithms for Top-*K* and range-query in case of whole matching search. They are presented in Table 1 and 2.

Table 1: Range-query search algorithm

```
Algorithm RangeQuery(Q,ε)
for i = 1 to N                        // For each row (record)
  Ci ⟵ (Ci – μ(Ci))/σ(Ci);            // Optional: normalize
  Ci ⟵ SomeTransformation(Ci);        // Optional: transform
  if D(Ci,Q) < ε
    list_id.add(Ci.id);             // Insert the ID of the
  end;                              // current row in the list
end;
```

Ding et al. [4] showed that, "*as the size of the training set increases, the accuracy of elastic measures converges with that of Euclidean distance*". Thus, the computed distance D(Ci,Q) will be Euclidean on raw, normalized or transformed data.

For Top-*K* search, we keep up to date a two-dimension table containing the IDs of the *K* nearest neighbors and their distances. If a row with a distance less than the *best-so-far* maximum distance is found, then the *best-so-far* distance is replaced as well as the corresponding ID.

Table 2: Top-*K* search algorithm

```
Algorithm TopK(Q,K)
for i = 1 to N                        // For each row
  Ci ⟵ (Ci – μ(Ci))/σ(Ci);            // Optional: normalize
  Ci ⟵ SomeTransformation(Ci);        // Optional: transform
  if D(Ci,Q) < max(tab_dist)
    tab_id.update(Ci.id);           // Update the Top-K tab by
    tab_dist.update(D(Ci,Q));       // replacing the previous
  end;                              // max by D(Ci,Q)
end;
```

Note that computing the mean and variance for the normalization step requires a loop iterating *n* times over the values of the current array. This is the same for computing the Euclidean Distance between the query and the current array. As it is a sum of positive terms, we can *early abandon* the calculation as soon as it is higher than ε (for range-queries) or than the *best-so-far* maximum distance (for Top-*K* queries), see [19].

These algorithms are easily adaptable to subsequence matching by adding another loop immediately after the loop over the rows. Indeed, this new loop must examine each subsequence and must normalize each subsequence, as stated in [19]. If the current array is at the beginning or in the middle of a time series, then this new loop iterates *n* times (half the size of the replicated array). Else if the current array is the end (i.e. last segment) of a time series, then data is not replicated and the loop iterates $n - l + 1$ times.

We now want to parallelize these algorithms in Hadoop and Hive. The basic idea is that data are split in blocks replicated three times in HDFS. The previously described naive algorithm is executed on each block, and a local table (Top-*K* or range-query) is returned. Then, these local tables are merged two by two, until obtaining the final table.

For the implementation, we use Generic UDAFs since they outperform any other kind of customization in Hive. Firstly, we have to define an aggregation structure and to initialize it. For Top-*K* search, it is a two-dimension table storing IDs and distances, whereas for range-query, it is a list with only IDs. Note that when dealing with subsequence matching, a new field storing the position is added to the aggregation. Then, we iterate over rows and keep the aggregation up to date. After merging partial aggregations, we get a final aggregation and return it in Hive. The function is plugged in Hive as shown by Table 3.

Table 3: Hive code for executing UD(A)Fs

```
CREATE TEMPORARY FUNCTION TopK as 'com.UDAF.TopK';
SELECT TopK(id, values)
FROM table
WHERE …;
```

One of the most useful aspects of using Hive is that *these functions can be combined with relational algebra*: projections, selections, joins, etc.

4.3 Grouping several queries

We started from whole matching and showed that algorithms, Generic UDAFs, and aggregation structures were also adapted to subsequence matching. Likewise, it is possible to search simultaneous several queries by keeping up to date a similar aggregation structure with an additional dimension in order to refer to the various queries. Moreover, we are still able to use the early abandoning: while at least for one query the current sum is less than the corresponding *best-so-far* maximum distance, we stay on the row and keep doing the calculation by reading next terms of the array. Thus, for each row, we drop one by one the queries as soon as they do not need further terms. It is reasonable to expect that it will be very cost-effective since only one pass on the data is necessary to answer to multiple queries simultaneously.

5. MULTI-DISTANCE SEARCH

In practice, end-users might want to look for particular patterns without knowing precisely which distance would be the best to minimize. The solution we propose is to compute several distances simultaneously for the same query, for example Euclidean Distance or DTW [14] on various representations (normalized and/or transformed data). This is done by keeping up to date one table for each considered distance.

Once again, this is easily implementable in Hive through Generic UDAFs functions using a similar aggregation structure with an additional dimension in order to refer to the tables corresponding to the various distances.

Assuming that distances are Euclidean, early abandoning is still usable: while at least for one distance the current sum is less than the corresponding *best-so-far* maximum distance, we stay on the row and keep doing the calculation by reading next terms of the array. Thus, for each row, we drop one by one the distances as soon as they do not need further terms.

Our approach might be used in order to distinguish between the various distances and to favor one of them, according to some experts' knowledge. But the solution we designed is to run a Top-*K* query for each distance and obtain a combined final ranking (an *inter-ranking*) based on the ranks given by all the Top-*K* of the various distances. This is a rank aggregation problem, a research field of social choice theory dating back to the 18th century which regains interest in the context of machine learning, data and Web mining.

5.1 Rank aggregation problem

The problem is the following:

Problem 1: Given *D* distances and *D* Top-*K* candidates for each distance, the problem is to obtain a combined final ranking of U distinct time series, in reasonable execution time and as *consensual* as possible.

We first define U, the set of results. Let C_D be the set of all time series appearing in the *D* Top-*K* (i.e. in the intersection). Similarly, let $C_{D\text{-}1}$ be a set of all the time series appearing in *D*-1 Top-*K*. Thus, we can recursively define *D* sets of time series, C_1, C_2, … , C_D. As among all the chosen distances, each one optimizes a particular criterion, it is highly unlikely that the *D* Top-*K* contain exactly the same time series ordered differently. However, should this case happens, we would define $U = C_D$ and have only *full lists*: the *D* Top-*K* would be $\tau_1, \tau_2, \dots, \tau_D$ permutations of the set U.

In general, we will have only *partial lists*, which are not permutations yet. Then, we set $U = C_D$ or $U = C_D \cup C_{D\text{-}1}$ for example and we come down to *full lists* like this:

- For each distance, we take (in the corresponding Top-*K*) the relative ranks of the time series in C_D as well as the relative ranks of the time series in $C_{D\text{-}1}$ which are present in this particular Top-*K*.
- For the time series in $C_{D\text{-}1}$ which are not present in this particular Top-*K*, we suggest two solutions: either to add them at the end of the ranking, or to compute their distances and to place them in the ranking according to it.

We now have *D* permutations (or rankings) $\tau_1, \tau_2, \dots, \tau_D$ of |U| time series (or candidates) $c_1, c_2, \dots, c_{|U|}$. The problem now is to obtain a new permutation σ as consensual as possible.

5.2 Optimal rank aggregation

We want to "*minimize the total disagreement between the several input rankings and their aggregation*" [7] by pleasing a majority of individual preferences. Thus, we aim at satisfying the *extended Condorcet criterion*: if the Condorcet winner exists (a candidate that wins the most duels by pairwise simple majority voting), then it should be ranked first. If it does not exist, then a set of candidates preferred by the majority over another set should be ranked higher. Formally, we aim at achieving optimality of the function below:

$$\varphi(\sigma) = \sum_{i=1}^{D} dist(\sigma, \tau_i) \quad \text{by finding} \quad \sigma^* = \operatorname{argmin} \sum_{i=1}^{D} dist(\sigma, \tau_i)$$

where $dist(\sigma, \tau)$ is the *Kendall tau distance* and counts the number of pairwise disagreements between two permutations:

$$K(\sigma, \tau) = \ |\{ i, j \ \in \mathrm{U} \ : \ \sigma(i) < \sigma(j) \text{ but } \tau(i) > \tau(j) \}|$$

The permutation σ^* obtained by optimizing *Kendall tau distance* is called *Kemeny optimal aggregation* and satisfies the extended Condorcet criterion. However, [7] showed that computing the optimum is NP-Hard, even for $D = 4$.

Another popular distance measure between two permutations is the *Spearman footrule distance* which sums over all candidates i in U the absolute difference between the ranks of i according to the two permutations:

$$F(\sigma, \tau) = \sum_{i=1}^{|\mathrm{U}|} |\sigma(i) - \tau(i)|$$

5.3 Heuristic approaches

We now enumerate rank aggregation methods that do not optimize any criteria, but are still effective in practice.

5.3.1 Footrule optimal aggregation

The following relation shows that the *Kendall tau distance* can be approximated via the *Spearman footrule distance*:

$$\forall \ \sigma, \tau \quad K(\sigma, \tau) \le F(\sigma, \tau) \le 2K(\sigma, \tau)$$

Thus, by computing a *footrule optimal aggregation*, we will have a lower bound (by a factor of two) on the optimal *Kendall tau distance*.

[7] investigated *footrule optimal aggregation* and showed that "*it can be computed in polynomial time, specifically, the time to find a minimum cost perfect matching in a bipartite graph*".

Let (*C*, *R*, *W*) be this graph where $C = \{c_1, \dots, c_{|\mathrm{U}|}\}$ is the set of candidates and $R = \{1, \dots, |\mathrm{U}|\}$ is the set of available ranks. The weighs are the following:

$$\forall \ c \in C, \forall \ r \in R, \ W(c, r) = \textstyle\sum_{i=1}^{D} |\tau_i(c) - r|.$$

Thus, for a certain matching, the sum of weighs is $\sum_{c=1}^{|\mathrm{U}|} W(c, r_c)$, which is exactly function φ with $dist(\sigma, \tau) = F(\sigma, \tau)$.

5.3.2 Positional methods

Borda's method was devised by Borda in 1770 but it is an old voting system already used by the Romans. It consists in giving each candidate a certain vector of length D containing its rank in each of these D input rankings. Each rank gives a number of points (in general equal to the rank). Then, these |U| vectors are sorted according to their L_1 or L_2 norms, their means, their medians, etc.

5.3.3 Markov chain methods

The idea is to represent the |U| candidates as vertices in a graph with transition probabilities depending in some particular way on the relative rankings of the candidates. From this state-transition matrix, we get a stationary distribution and thus, an inter-ranking.

Dwork et al. [7] suggested four ways to define the transition probabilities.

6. EXPERIMENTS

As mentioned earlier, two different datasets were used to conduct experiments with the following different goals:

- On the synthetic electric power load curves dataset, we tested the scalability and measured execution time for Top-*K* search, both for whole and subsequence matching.
- In the grid frequency in power plants dataset, we looked for substantial drops in frequency, ideally a 50mHz fall happening in 10 seconds or less, being stable before and after the fall. The end-user, who is an expert, visually knows whether time series resulting from the Top-*K* search match or not. This pattern was defined by the end-user who was not able to define precisely which distance to use. For example, it could be the Euclidean Distance on the whole original pattern, on the drop only, on the derivative of the original pattern, etc. In this experiment, we tested different distances and showed the improvement provided by inter-ranking.

6.1 Hadoop infrastructure

As in [5], the cluster we used in our experiments is composed of 2 racks and a total of 20 nodes with 2 kinds of commodity servers (7 are 1U and 13 are 2U). The 1U servers have 4x1 TB disks while the 2U servers have 8x1 TB disks. Most of the servers have 2 CPUs AMD Opteron(tm) with 8 cores each (2 have 2 x 12 cores). In total, the cluster is composed of 132 TB of storage and 336 cores. The Hadoop distribution installed is the open source CDH3u3 from Cloudera.

6.2 Datasets

6.2.1 Electric power load curves

This first dataset was generated in the context of the proof of concept (POC) carried out by the R&D Division, see [5]. We tested our algorithms on a subset of this dataset: 35,027,502 customers with raw electricity consumption measurements (in Wh) every 30 minutes during one month. This is equivalent to around 3.3 TB of uncompressed data (≈110 GB per day).

For our experiment, we chose as basic length of arrays $n = 48$ and we created two tables: one for daily whole matching queries (of length 48), and one for subsequence matching queries (of length up to 48).

The table used for daily whole matching queries has three columns shown in Table 4.

Table 4: Table structure for whole matching. `id_client` **and** `day` **form the primary key of the table.**

ID_CLIENT (int)	**DAY** (string)	**VALUES** (array<int>)
136630	2008-06-07	[625, 473, 483, 202, 396, 241, …]

The table used for subsequence matching queries concatenates to each row the values of the next day, thus introducing redundancy as suggested in Section 4.1. This approach is illustrated in Table 5 for the second dataset.

Recall that we used a text file format with a deflate compression. With this configuration, the final volume of the whole matching table is around 96 GB (replicated three times in HDFS). It is a considerable saving in terms of storage disk space compared to the original 3.3 TB of uncompressed data. The final volume of the compressed subsequence matching table is around 185 GB.

6.2.2 *Grid frequency in power plants*

This dataset is made up of real electric power network frequency measurements taken at one power plant every second during one month. This makes $29 \times 24 \times 60 \times 60 = 2{,}505{,}600$ values and around 145 MB of uncompressed data. Note that it is only a sample: we expect to process historical data of several years. In this dataset, we were trying to find drops in frequency, defined as follows: a fall greater than or equal to 50 mHz, happening in 10'' or less, stable during 30'' before and 60'' after the fall. This is a subsequence matching problem because the drop may start at any position. Since the length of the query is 100, we used the previously described storage structure and created arrays of 240 values with an overlapping of 120 values. Thus, we can handle any subsequence matching query of maximal length 120 (namely 2 minutes). The table we used contained 2,505,600 / 120 = 20,880 rows. It is illustrated in Table 5.

Table 5: Table structure used for the grid frequency dataset. `id_minute` is the primary key of the table

ID_MINUTE (int)	DAY (string)	TIME (string)	VALUES (array<double>)
0	01/01/2012	00:00	1 120 | 121 240
1	01/01/2012	00:02	1 120 | 121 240
⋮	⋮	⋮	⋮
20879	29/01/2012	23:58	1 120

We also used a text file format with a deflate compression, for a total volume of 3.98 MB. Obviously, the challenge for this dataset is not response time but the fact that experts are not able to define the right distance to use.

Note that raw data is stored and the algorithm normalizes each examined subsequence (by shifting but not scaling since we are interested in the absolute value of the drop).

6.3 Results on scalability

6.3.1 *Simple query Top-K search*

In this section, we measured the time taken by a Top-*K* search on our electric power consumption dataset. We first show how important is the distribution of the data in HDFS blocks. Indeed, the number of blocks is equal to the number of Mappers which depends on the number of nodes in the cluster, but also on the complexity of tasks.

We created several versions of the table mentioned earlier, with various block sizes and thus, various numbers of blocks. Then, we measured the time taken for a Top-*K* search (arbitrarily $K = 5$ since we noticed almost no influence of *K* on the execution time) and averaged it on 100 queries (drawn randomly in the dataset).

We did the same for the subsequence matching table. We recall that we mentioned two ways to handle subsequence matching queries: by a self join or a replicated storage. The self join takes around 50' to be executed: this confirmed what we intuited in Section 4.1. Consequently, we concentrate on the replicated table solution in the following. The results are shown in Figure 5.

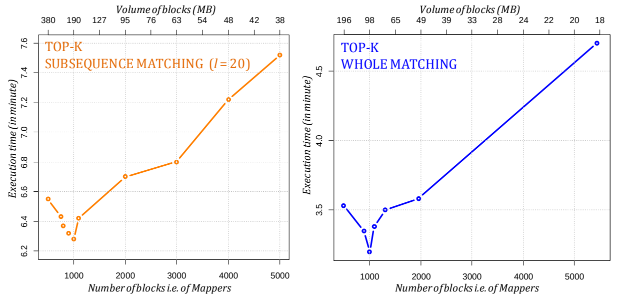


Figure 5: *left*) Time taken by a Top-5 whole matching search in terms of the number of HDFS blocks. *right*) idem for a Top-5 subsequence matching search.

Hadoop indicates that a good level of parallelism is between 10 and 100 Mappers per node. Our cluster containing 20 nodes, it makes between 200 and 2000 Mappers. This result is validated by our experimental results since our 1000 optimal blocks are located between these bounds. Using this new data distribution, we obtained final query execution times shown in Table 6. These response times do not allow interactive search but appear to be satisfactory for the applications we consider.

Table 6: Time taken (in minutes, averaged on 100 queries drawn randomly in the table) by Top-5 queries on two 1000-block tables, one for whole matching and one for subsequence matching.

	Whole Matching	Subsequence Matching		
		$l = 5$	$l = 20$	$l = 45$
Min	3.00	5.62	5.87	6.33
Mean	**3.20**	**5.85**	**6.52**	**7.82**
Max	3.32	6.15	7.08	8.88

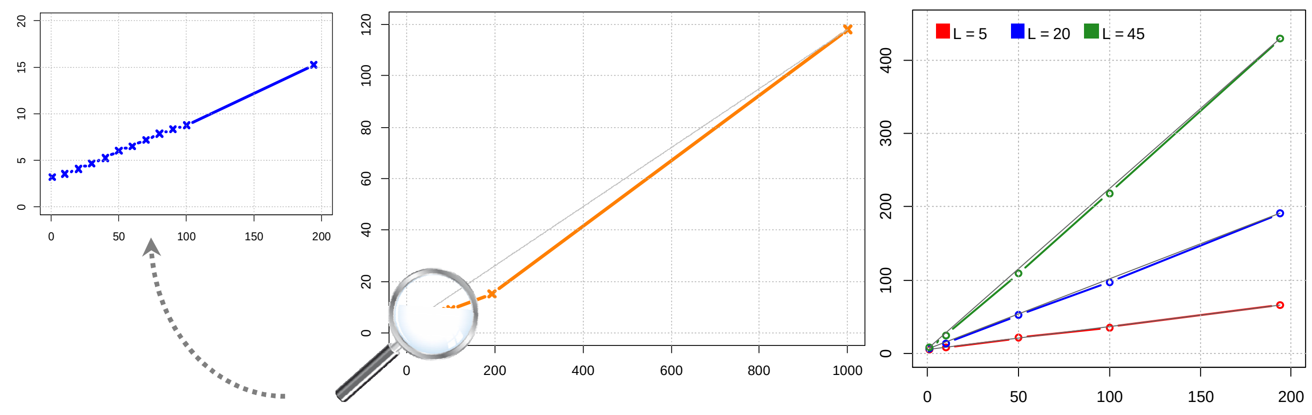


Figure 6: *left*) Time taken (in minutes) by a whole matching Top-5 search in terms of the number of simultaneous queries. *right*) Time taken (in minutes) by subsequence matching queries (of length 5, 20, and 45) Top-5 search in terms of the number of simultaneous queries.

6.3.2 *Multiple queries Top-K search*

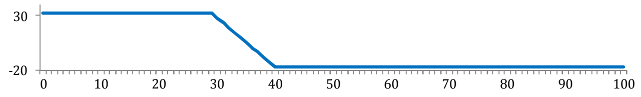
We implemented slightly modified functions allowing up to 1000 simultaneous queries with early-abandon. Clearly, we kept the previously described optimal tables with 1000 blocks. As we expected, our approach is very efficient, as shown by Figure 6, and enables us to process 100 queries in less than 10 minutes.

6.4 Results on multi-distance search

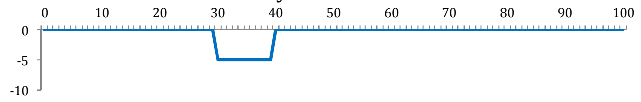
We report here experiments related to the technique described in Section 5, applied to the grid electric power frequency dataset. We are looking for a very precise pattern.

We implemented the following four distance measures:

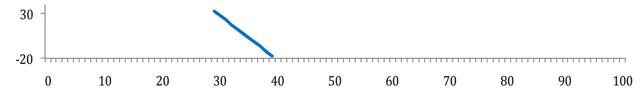
- ***d_1*: Weighted Euclidean Distance on the whole pattern** (illustrated below). We assign higher weighs to the drop in order to balance the trade-off between the absolute value of the drop and the stability before and after.



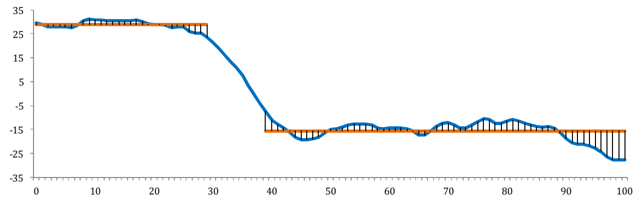
- ***d_2*: Weighted Euclidean Distance on the derivative pattern** (illustrated below). The derivative, computed as the difference between a value and its following one, ensures the stability before and after the fall.



- ***d_3*: Euclidean Distance on the drop only** (illustrated below). This distance guarantees to return only the time series with a substantial linear drop of 50mHz in 10 seconds. However, it does not ensure stability.



- ***d_4*: Variance of the two stability zones** (illustrated below). This is not a distance but rather a quantification of how the examined subsequence matches a model: for each subsequence, we compute $\overline{x_{1..30}}$ (the mean of the first 30 values) and $\overline{x_{41..100}}$ (the mean of the last 60 values). If the difference is higher than a certain threshold (ideally 50), then the measure is defined and it is the sum of the variances of the two stability zones.



We now determine the best K (recall that we execute a Top-K search for each distance) in order to have a set of candidates U for the inter-ranking. We tried various values of K between 1 and 1000 but there does not exist a K beyond which all distances would return the same time series.

For our experiment, we used K = 30 and U = $C_4 \cup C_3$. This choice led us to inter-rank 12 time series A, B, … , L.

We obtained four permutations from which we tested the previously described rank aggregation methods. The results are shown in Figure 7.

Method	Final ranking (or permutation) σ												$\varphi(\sigma) = \sum_{i=1}^{D} K(\sigma, \tau_i)$
Borda – 1	A	B	C	**D**	**E**	**F**	**G**	H	I	J	K	L	83
Borda – 2	A	B	C	**D**	**E**	**G**	**F**	H	I	J	K	L	85
Borda – 3	A	B	C	**E**	**D**	**F**	**G**	H	I	J	K	L	87
Borda – 4	A	B	C	**E**	**D**	**G**	**F**	H	I	J	K	L	89
MC1	A	B	G	C	D	E	F	I	H	J	K	L	99
MC2	A	B	G	I	D	C	E	H	F	J	K	L	125
MC3	A	B	C	G	D	E	F	H	I	J	K	L	93
MC4	A	B	C	D	E	F	G	H	I	J	K	L	83
Footrule optimal aggregation	A	D	E	G	C	H	F	I	B	K	J	L	131

Figure 7: Borda's method (L_1-norm of the D-dimensional vector of the ranks) induces equally placed time series. All possible permutations are listed.

The optimal function was computed for each method. It appears that Borda's method exhibits the best permutation, although it is very simple.

Finally, we can visualize graphical results (see Figures 8 and 9). Experts claimed that it clearly appears that this technique outperforms the classic one using only one distance measure.

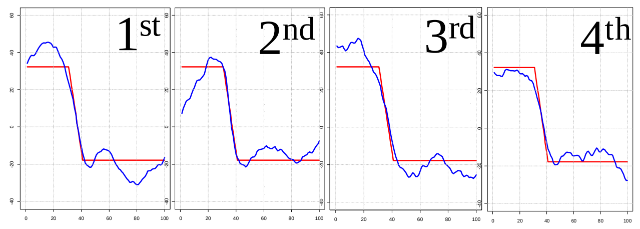


Figure 8: The first four final time series obtained using inter-ranking technique.

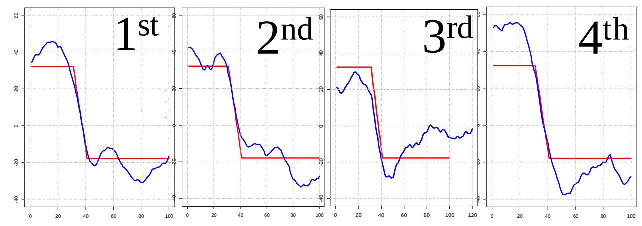


Figure 9: The first four final time series obtained using only one distance, here it is d_2 (on the derivative pattern).

7. CONCLUSION AND FURTHER WORK

In this paper, we proposed an approach for similarity search in a very large time series database, typically containing more than a billion 48-dimensional time series. We used the brutal force of Hadoop and Hive along with its adaptability and flexibility potential to parallelize this search.

On the 336 cores cluster set up in EDF R&D, the result of the Top-K search is obtained in a few minutes. But this is only a glimpse on all the advantages showed by Hadoop's brutal force. In our case for example, its data processing power allows to answer multiple queries in only one pass without reducing much the execution time. We can also carry out normalizations and/or complex transformations on the time series, in order to smooth them or to compute another kind of distance. We evaluated performance on synthetic time series representing smart metering data.

However, as the distance depends a lot on the data contents, it is frequent that the end-user does not know precisely which distance to use. Similarly to multiple queries, we suggested to solve this problem by computing several distances, and thus to build several rankings, one for each considered distance. Then, based on these rankings, we get a final one using heuristics on rank aggregation methods. We implemented this approach with four distances on grid frequency data. We tested various rank aggregation methods.

In future work we intend to improve our ranking approach by using relevance feedback: for each result returned by our algorithms, an expert accesses its relevance. If queries, results, and relevance are recorded, the system can improve by itself. This would be another way to approve or to dismiss a particular distance.

We also plan to search multivariate time series. How to define a multivariate query? This raises several problems, such as the number of queries (should we have a query for each variable or not), the length of the queries (should all the queries be the same length; if not, what would be the limit), the time's position (should a query require the same offset for each variable, or should we allow some offset), the weight of each variable (should some variables be more important than others), etc.

Finally, we project to implement and test indexation techniques in Hadoop, for example R-trees. This has been well discussed in the literature, though not it a parallelized environment, and consists in indexing the first few coefficients of a particular transformation (for example PAA, DFT, DWT, etc).

8. REFERENCES

[1] Agrawal, R., Faloutsos, C. and Swami, A. 1993. Efficient Similarity Search in Sequence Databases. In *Proc. of 4th Conf. on Foundations of Data Organization and Algorithms*

[2] Beckmann, N., Kriegel, H.P., Schneider, R. and Seeger, B. 1990. The R*-tree, an Efficient and Robust Access Method for Points and Rectangles. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 322-331

[3] Chan, K.P. and Fu, A.W.C. 1999. Efficient Time Series Matching by Wavelets. In *Proc. of 15th IEEE Intl. Conf. on Data Engineering*

[4] Ding, H., Trajcevski, G., Scheuermann, P., Wang, X. and Keogh, E. 2008. Querying and Mining of Time Series Data: Experimental Comparison of Representations and Distance Measures. In *PVLDB* 1, 2, pp. 1542-52

[5] D. P. dos Santos, L., Picard, M.-L., G. da Silva, A., Worms, D., Jacquin, B. and Bernard, C. 2012. Massive Smart Meter Data Storage and Processing on top of Hadoop. In *VLDB*

[6] Dutta, H., Kamil, A., Pooleery, M., Sethumadhavan, S. and Demme, J. 2011. Distributed Storage of Large Scale Multidimensional Electroencephalogram Data using Hadoop and HBase. In *Grid and Cloud Database Management*, Springer, Heidelberg

[7] Dwork, C., Kumar, R., Naor, M. and Sivakumar, D. 2001. Rank Aggregation Methods for the Web. In *Proc. of the 10th Intl. World Wide Web Conf.*, New York, pp. 613-622

[8] Faloutsos, C., Ranganathan, M. and Manolopoulos, Y. 1994. Fast Subsequence Matching in Time-Series Databases. In *Proc. ACM SIGMOD Conf.*, Minneapolis

[9] Goldin, D.Q., Millstein, T.D. and Kutlu, A. 2004. Bounded similarity querying for time-series data. In *Information and Computation*, 194(2), pp. 203-241

[10] Guttman, A. 1984. R-trees, a Dynamic Index Structure for Spatial Searching. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 47-57

[11] JaJa, J., Kim, J. and Qin, W. 2004. Efficient Serial and Parallel Algorithms for Querying Large Scale Multidimensional Time Series Data. *http://www.umiacs.umd.edu/publications/efficient-serial-and-parallel-algorithms-querying-large-scale-multidimensional-time*

[12] Keogh, E., Chakrabarti, K., Pazzani, M. and Mehrotra, S. 2001. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. In *Knowl. Inf. Syst.*, 3(3)

[13] Keogh, E., Chakrabarti, K., Pazzani, M. and Mehrotra, S. 2001. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. In *SIGMOD Conf.*

[14] Keogh, E. and Ratanamahatana, C.A. 2002. Exact Indexing of Dynamic Time Warping. In *VLDB*

[15] Korn, F., Jagadish, H.V. and Faloutsos, C. 1997. Efficiently Supporting Ad Hoc Queries in Large Datasets of Time Sequences. In *SIGMOD Conf.*

[16] Lin, K.I., Jagadish, H.V. and Faloutsos, C. 1993. The TV-tree, An Index Structure for High-Dimensional Data. In *VLDB*

[17] Lin, J., Keogh, E., Wei, L. and Lonardi, S. 2007. Experiencing SAX: a novel symbolic representation of time series. In *Data Min. Knowl. Discov.*, 15(2)

[18] Qiao, J., Ye, Y. and Zhang, C. 2006. Parallelization of Similarity Search in Large Time Series Databases. In *Proc. IMSCCS*

[19] Rakthanmanon, T., Campana, B., Mueen, A., Batista, G., Westover, B., Zhu, Q., Zakaria, J. and Keogh, E. 2012. Searching and Mining Trillions of Time Series Subsequences under Dynamic Time Warping. In *VLDB*

[20] Ravi Kanth, K.V., Agrawal, D. and Singh, A. 1998. Dimensionality Reduction for Similarity Searching in Dynamic Databases. In *Proc. ACM SIGMOD Conf.*, pp. 166-176

[21] Shieh, J. and Keogh, E. 2008. iSAX: Indexing and Mining Terabyte Sized Time Series. In *KDD*, pp. 623-631

[22] Stollnitz, E.J., DeRose, T.D. and Salesin, D.H. 1995. Wavelets for Computer Graphics, a Primer. In *IEEE Comput. Graph. Appl.*, 15(3), pp. 76-84

[23] Wu, Y.L., Agrawal, D. and El Abbadi, A. 2000. A Comparison of DFT and DWT Based Similarity Search in Time-Series Databases. In *CIKM*