Storage-Optimizing Clustering Algorithms for High-Dimensional Tick Data

Krisztian Buza[a], Gábor I. Nagy[a], Alexandros Nanopoulos[b]

[a] *Faculty of Electrical Engineering and Informatics*
Budapest University of Technology and Economics, Hungary
buza@cs.bme.hu, nagy.gabor.i@gmail.com
[b] *University of Eichstätt-Ingolstadt, Germany*
alexandros.nanopoulos@ku.de

Abstract

Tick data are used in several applications that need to keep track of values changing over time, like prices on the stock market or meteorological measurements. Due to the possibly very frequent changes, the size of tick data tends to increase rapidly. Therefore, it becomes of paramount importance to reduce the storage space of tick data while, at the same time, allowing queries to be executed efficiently. In this paper, we propose an approach to decompose the original tick data matrix by clustering their attributes using a new clustering algorithm called Storage-Optimizing Hierarchical Agglomerative Clustering (SOHAC). We additionally propose a method for speeding up SOHAC based on a new lower bounding technique that allows SOHAC to be applied to high-dimensional tick data. Our experimental evaluation shows that the proposed approach compares favorably to several baselines in terms of compression. Additionally, it can lead to significant speedup in terms of running time.

Keywords: Tick data, Clustering, Storage

1. Introduction

Representing information about complex objects or phenomena requires a large set of attributes (a.k.a. features). The values of these attributes may change rapidly over time; for instance, prices on the stock market or meteorological measurements. Data with such characteristics, i.e., high complexity/variety, large volume and change velocity, are nowadays being addressed within the paradigm of *big data*. Beyond the challenges pertaining to the management of such data, analytical tasks pose the crucial requirement of investigating their dynamics, i.e., how their attributes change values over time. What is, therefore, necessary is to keep track of the changes of values, which results in very large collections of rapidly changing data. This particular data type is being referred to as *tick data* [13], which is the term we adopt in our study, too.

One of the most important examples for tick data is the record of stock market transactions. This type of data can be considered as a matrix: the columns of the matrix correspond different properties of a transaction, such as price, volume of the trade, the symbol of an asset, etc. Every time a transaction is executed or a quote is given for a stock, a row is appended to this matrix. The speed of a transaction in liquid financial markets can be measured in the order of fractions of milliseconds therefore, this matrix grows extremely rapidly. It is necessary to record all transactions but this magnitude of data flow requires proprietary technology allowing efficient storage and quick retrieval of data. Solutions are often built over column based database technology such as a KDB database.[1] However, as we will describe in more detail, a straightforward application of such technology leads to suboptimal storage of tick data in the sense of occupied storage space on disk or memory.

The primary reason for the sub-optimality of the aforementioned storage is the redundancy of conventional techniques: in case of a straightforward solution, one would use orders of magnitudes more storage space (either disk space or main memory) than required, therefore, storage, access, search and analysis of the data becomes computationally more expensive than necessary. One way to alleviate this problem is the usage of regular data compression methods (see e.g. [16] for an excellent overview). This approach is well-suited for cheap storage of large, historical archives of the data. However, it does not support quick access to the data: if the data is compressed, in order to be able to execute an analytic or search query, a large archive (or at least some parts of that) must be decompressed which might be computationally expensive and therefore the procedure could become inefficient. This problem becomes crucial if *many* queries have to be executed which is usually the case in real-life applications, such as stock market trading.

In this paper, we aim at developing a storage structure for tick data that reduces the storage space required by the straightforward approach with the ability to execute search and analytic queries efficiently in the same manor provided by the straightforward approach. In particular, our approach is based on the decomposition of a large tick data matrix into a number of smaller matrices. We achieve this decomposition by clustering the columns of the matrix. Each resulting smaller matrix contains a subset of the original attributes and the corresponding values of all rows of the original matrix for these attributes. Although, conventional clustering algorithms achieve significant improvements, motivated by hierarchical clustering algorithms, we develop a

[1] http://kx.com/

new clustering algorithm that minimize storage space required for a tick data matrix. We call our approach SOHAC, Storage-Optimizing Hierarchical Agglomerative Clustering.

In this paper, we focus on speeding up SOHAC and propose a new lower bounding technique that allows SOHAC to be applied to high dimensional tick data; we refer to the resulting method as QuickSOHAC. According to our experiments, the proposed lower bounding technique combined with sampling leads to substantial speedup without significant loss of quality. We evaluate the proposed approach on three real-world tick data tables provided by an investment bank. Furthermore, while the primary motivation of our work is to develop a storage schema that allows efficient storage and quick access to the data, we show that conventional compression algorithms may also substantially benefit from the proposed approach.

The remainder of this paper is organized as follows. Section 2 provides an overview of related works. In Section 3 we present the SOHAC algorithm. In Section 4 we focus on efficient implementation of SOHAC and propose a lower bounding technique that allows to speed up the algorithm. We devote Section 5 to the experimental evaluation of our approach. Finally, we conclude in Section 6.

2. Related Work

The data describing transactions on financial markets, especially *tick data* (also known as *tick-by-tick data*) allows thorough analysis of the markets and their dynamics. Recent works focused on statistical properties [14] and analysis [17], [18] of currency exchange rates and stock market tick data [13]. Dionne focused on risk analysis [8], while the dynamics of stock markets were studied in [4], [15] and [21]. Based on tick data, Akram et al. empirically studied the law of one price on different financial markets [2], while Cartea proposed to model stock price tick-by-tick data via a non-explosive marked point process [7]. Blais and Protter studied various algorithms for the recognition of whether a transaction in a tick-data set was initiated by the buyer or seller [5]. Barany et al. [3] aimed to detect market crashes based on high-frequency data. Yabuuchi and Watada studied fuzzy autocorrelation models for forecasting problems related to tick data [23].

Although, storage of tick data is a core component of the systems performing the above analytic tasks, none of the above works focused on how to develop storage structures for high dimensional tick data. More closely related to our work is that of Ahmad et al. who focused on the summarization of tick data time series [1]. However, instead of storing a summarized approximation of the time series, we aim at storing the entire original tick data. As we will demonstrate it in Section 3.1, the storage of tick data is a non-trivial task: conventional techniques result in redundant and therefore suboptimal solutions.

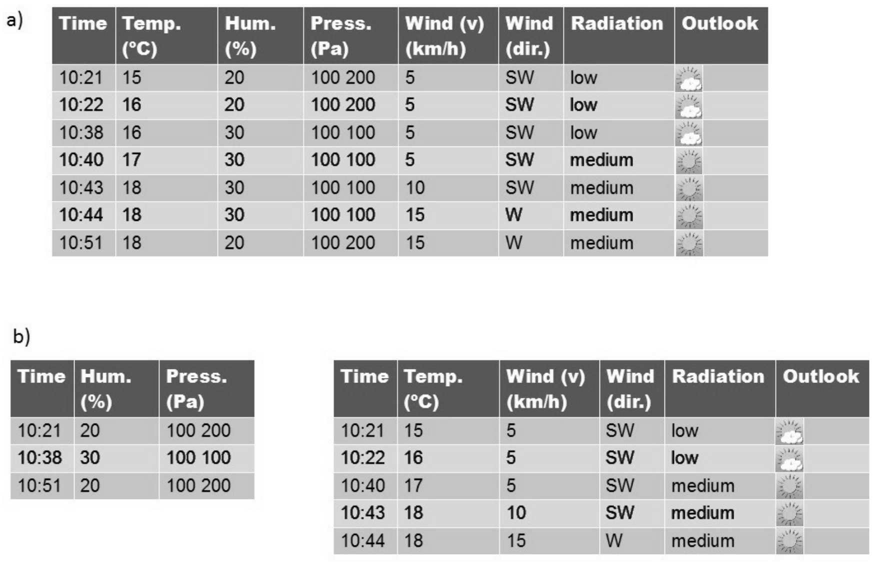
Conventional data compression techniques, see e.g. [16], can efficiently compress the data, and therefore they are well-suited for data archives. For tick data, however, we need storage structures that allow efficient storage and quick access to the data simultaneously while they can be realized in database systems. As we will demonstrate, large tick data tables can usually be decomposed into several smaller ones, the total size of which being substantially less than that of the original table. Simultaneously, this decomposition allows for quick querying of the data as we will discuss in Section 3.5.

Clustering is well-known for its ability to summarize data by grouping similar objects together which allows the user to consider the data at a higher level of abstraction [19]. Therefore, our approach for the decomposition of tick data matrices is based on clustering. In the last decades, very large number of clustering algorithms were developed for various tasks (see e.g. [6], [9], [10] and [12]). We refer to [19] and [22] for excellent surveys of clustering algorithms. We will demonstrate that one can achieve substantial improvements if one uses general-purpose clustering algorithms for the decomposition of tick data matrices. However, such conventional clustering algorithms were originally not designed for storage optimization of tick data and therefore they lead to suboptimal decompositions in most cases.

In contrast, we propose a clustering algorithm, SOHAC, which directly minimize the required storage space and therefore, as shown in large number of extensive experiments, SOHAC substantially outperforms conventional clustering algorithms for the tick data storage problem. An initial, but promising version of our algorithm was presented in [11]. However, due to its computational complexity, the application of SOHAC to high-dimensional tick data still remained a challenge. This is essential, as most of the tick data tables in real-world applications are high dimensional, i.e., contain many columns. Therefore, after reviewing SOHAC, in this article, we focus on speeding up the algorithm. The techniques we propose allow SOHAC to find the decomposition quickly, even in case of high-dimensional tick data tables.

3. Decomposition of Tick Data Matrices based on Clustering

In this section, we explain how the decomposition of tick data matrices may lead to more efficient storage. First, we motivate our approach with an illustrative example, then we review our clustering algorithm, SOHAC, that supports efficient storage of tick data.


a)

Time	Temp. (°C)	Hum. (%)	Press. (Pa)	Wind (v) (km/h)	Wind (dir.)	Radiation	Outlook
10:21	15	20	100 200	5	SW	low	⛅
10:22	16	20	100 200	5	SW	low	⛅
10:38	16	30	100 100	5	SW	low	⛅
10:40	17	30	100 100	5	SW	medium	☀
10:43	18	30	100 100	10	SW	medium	☀
10:44	18	30	100 100	15	W	medium	☀
10:51	18	20	100 200	15	W	medium	☀

b)

Time	Hum. (%)	Press. (Pa)
10:21	20	100 200
10:38	30	100 100
10:51	20	100 200

Time	Temp. (°C)	Wind (v) (km/h)	Wind (dir.)	Radiation	Outlook
10:21	15	5	SW	low	⛅
10:22	16	5	SW	low	⛅
10:40	17	5	SW	medium	☀
10:43	18	10	SW	medium	☀
10:44	18	15	W	medium	☀

Figure 1: An illustrative example for tick data. Features describing the weather are monitored continuously. Whenever the value of one of the features changes, a new row is inserted into the recordings (see the table in the top). Decomposition of such tables by features (columns) that change their values simultaneously may substantially reduce the required storage space (see the tables in the bottom of the figure).

3.1. An Illustrative Example

Suppose that a weather station monitors features of weather conditions. In this example, such features are the temperature, humidity and pressure of the air, the velocity and direction of the wind, the intensity of the radiation of the sun and the overall outlook (such as sunny, cloudy, raining or snowing). These features are monitored continuously over the time. Whenever the value of one of these features changes, a new raw is inserted into the recordings. This new row contains the values of the features as well as a time-stamp indicating *when* the observations were made. See the matrix in the top of Figure 1.

This representation, called tick data, is well-suited for queries: for example, if we are interested for the features of the weather at 10:30 o'clock, we only need to find the raw corresponding the most recent observation *before* 10:30, i.e., we have to consider the raw at 10:22. This raw describes the "state of the world", i.e., it contains the values of all the features that are relevant in the current application. Such queries regarding the "state of the world" at a given time can be effectively supported by indexing techniques.

The only disadvantage of the representation shown in the top of Figure 1 is that the total size of the matrix may become much larger than actually required. In order to illustrate this we stored the same information in two smaller matrices in the bottom of Figure 1. In our approach such decompositions are based on the clustering of columns: in the example, we consider two clusters of columns. One of the clusters contains *Humidity* and *Pressure*, while the other cluster contains the other columns, i.e., *Temperature, Velocity of the wind, Direction of the wind, Radiation* and *Outlook*. As shown in the example, due to the decomposition, we can save storage space: the total number of cells required to store the data was reduced from $7 \times 7 = 49$ to $3 \times 2 + 5 \times 5 = 31$ (without counting the cells in the column *Time* which acts like an index column). This corresponds to a *compression ratio* of $31/49 \approx 0,633$.

While the decomposition reduces the required storage space, in the worst case, the computational complexity of a query may increase moderately: if we are interested for *all* the features describing the weather at 10:30, we have to execute two queries instead of one, however, both queries are executed on much *smaller datasets* (and therefore the overall execution time is expected to grow only moderately compared to the previous case). Whereas if we are only interested for the *temperature and radiation* we have to execute just one query on a dataset of reduced size (and therefore the overall execution time is expected to be reduced).

The example in Figure 1 illustrates the decomposition of a tick data matrix in an intuitive way. Next, we systematically study such decompositions and develop an algorithm that aims at minimizing the storage space required after the decomposition.

3.2. Definitions and Problem Formulation

In general, a *tick data matrix M* is a matrix where columns correspond attributes or features while rows correspond observations of the same features at different moments of time. Rows of the matrix are ordered according to the order of observations, i.e., the values of the i-th row were observed *before* the values of the j-th row if and only if $i < j$. While the observations are made, a new row is added whenever the value of an attribute changes. However, as long as none of the attribute-values changes no new row is added to the matrix, therefore two rows of a tick data matrix differ in the value of at least one attribute. There is an additional column that is used to index the rows of a tick data matrix. This additional *index column* may contain, for example, ascending integer numbers (like the number of the corresponding row) or a time-stamp (see the *Time* column in the example in Section 3.1). We use the term *regular column* for all the columns other than the index column.

With *decomposition* of a tick data matrix M we mean the clustering of the regular columns of

M into k disjoint clusters P_i, $1 \leq i \leq k$, i.e., for each regular column c_j of M:

$$c_j \in P_1 \vee c_j \in P_2 \vee ... \vee c_j \in P_k$$

and for all i, j with $i \neq j$

$$P_i \cap P_j = \emptyset.$$

Note that this clustering refers to the regular columns only, i.e., in this formulation, the index column does not belong to any cluster. Then, for each cluster P_i, a matrix M_i is derived from M by selecting the index column and those columns of M that belong to cluster P_i. Subsequent rows of a derived matrix M_i may contain the same values in all the regular columns. In such cases we only keep the first row. For example, in Figure 1,

$$P_1 = \{\text{Humidity, Pressure}\},$$

$$P_2 = \{\text{Temperature, Wind (velocity), Wind (direction), Radiation, Outlook}\}$$

and the corresponding matrices M_1 and M_2 are shown in the bottom left and bottom right of the Figure 1.

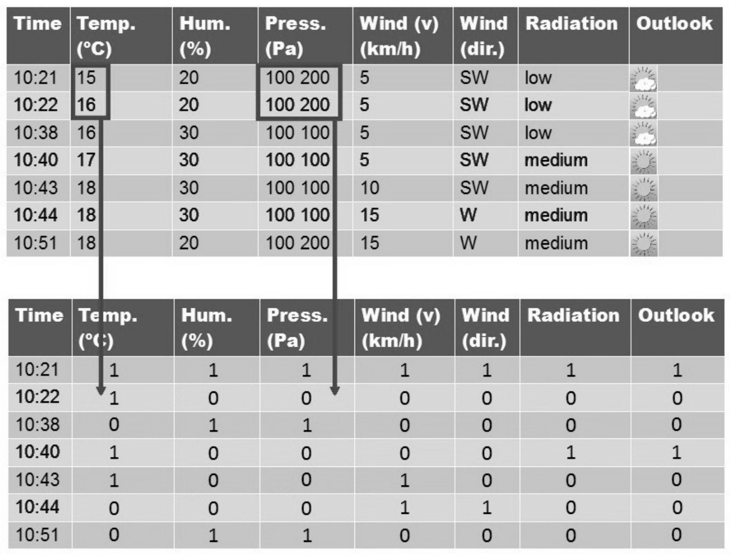
We can easily see that the original matrix can be reconstructed from the decomposition described above, and therefore, instead of the original matrix M, one can use this decomposition to calculate the results of search and analytic queries.

In this paper, we target the problem of finding a decomposition so that the required storage space is minimized. In particular, for a given number of clusters k, we aim at finding a decomposition so that the total number of the cells in all the matrices M_i (without counting the cells in the index column) is minimized. Our approach can simply be adapted for the case of more advanced storage models, where we do not assume uniform storage cost for each cells and/or the storage costs of the index cell is also taken into account.

We note that k is usually relatively small: for example, for the storage of tick data of financial transactions, the user is most interested for the decomposition into $k = 2$ or $k = 3$ clusters (see also Section 3.5).

3.3. Clustering of Columns of Tick Data Matrix

In the literature, there are many clustering algorithms that are able to produce non-overlapping clusters in a way that these clusters together cover all the instances. Therefore, one solution for the problem defined in the previous section is to cluster the columns of a tick data matrix.



Time	Temp. (°C)	Hum. (%)	Press. (Pa)	Wind (v) (km/h)	Wind (dir.)	Radiation	Outlook
10:21	15	20	100 200	5	SW	low	⛅
10:22	16	20	100 200	5	SW	low	⛅
10:38	16	30	100 100	5	SW	low	⛅
10:40	17	30	100 100	5	SW	medium	☀
10:43	18	30	100 100	10	SW	medium	☀
10:44	18	30	100 100	15	W	medium	☀
10:51	18	20	100 200	15	W	medium	☀

Time	Temp. (°C)	Hum. (%)	Press. (Pa)	Wind (v) (km/h)	Wind (dir.)	Radiation	Outlook
10:21	1	1	1	1	1	1	1
10:22	1	0	0	0	0	0	0
10:38	0	1	1	0	0	0	0
10:40	1	0	0	0	0	1	1
10:43	1	0	0	1	0	0	0
10:44	0	0	0	1	1	0	0
10:51	0	1	1	0	0	0	0

Figure 2: Construction of a binary change indicator matrix from a tick data matrix. The tick data matrix is shown in the top of the figure, while the corresponding indicator matrix is shown in the bottom. The index column is the *Time* column in this example.

In the context of our problem, two regular columns are considered to be similar, if they often change values togehter (i.e., in the same rows). Therefore, we define a *binary change indicator matrix* I over a tick data matrix M. Except the entries of the index column, all the entries of the binary change indicator matrix I are either 0 or 1 depending on whether or not the value of a cell in the tick data matrix M is equal to the value of the cell in the same column and the *previous* row of M:

$$I(i,j) = \begin{cases} M(i,j) & \text{if the } j\text{-th column is the index column in } M \\ 0 & \text{if } i > 1 \text{ and } M(i,j) = M(i-1,j) \\ 1 & \text{otherwise} \end{cases}$$

where $M(i,j)$ and $I(i,j)$ denote the entries in the i-th row and j-th column of the tick data matrix M and binary change indicator matrix I respectively.

As an example, Figure 2 shows how the binary change indicator matrix is derived from a tick data matrix. The index column is the *Time* column in this example.

After constructing the binary change indicator matrix I, we can use its regular columns (i.e., all the columns except the index column) as instances in almost any clustering algorithms. Despite the fact that conventional clustering algorithms are not designed to produce optimal clusters in terms of our problem defined in Section 3.2, as we have shown in [11], if we use the binary change indicator matrix representation and a conventional clustering algorithm to find a decomposition of the tick

Algorithm 1 SOHAC: Storage-Optimizing Hierarchical Agglomerative Clustering for Tick Data

```
Require: Tick data matrix $M$, number of clusters $k$
Ensure: Clustering of the columns of $M$
 1: Construct the binary change indicator matrix $I$ from $M$
 2: $P = \{\{c_1\}, \{c_2\}, ..., \{c_n\}\}$
    (Initially, each column $c_j$ of $M$ is a separate cluster)
 3: while $|P| > k$ do
 4:   $s \leftarrow \infty$   (Storage size for the best clustering found so far)
 5:   for all pairs of clusters $(\mathcal{C}_i, \mathcal{C}_j)$, with $\mathcal{C}_i \in P$, $\mathcal{C}_j \in P$ do
 6:     $\mathcal{C}'_i \leftarrow \mathcal{C}_i \cup \mathcal{C}_j$   (Merge clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ into the new cluster $\mathcal{C}'_i$ )
 7:     $P' \leftarrow P \setminus \{\mathcal{C}_i, \mathcal{C}_j\} \cup \{\mathcal{C}'_i\}$
 8:     $s'$ = storage size required to store the decomposition corresponding to $P'$   (This can be
        computed based on $I$.)
 9:     if $s' < s$ then
10:       $P^* \leftarrow P'$   (The best clustering found so far)
11:       $s \leftarrow s'$
12:     end if
13:     $P \leftarrow P^*$
14:   end for
15: end while
16: return $P$
```

data matrix we can already achieve notable improvements in terms of storage space compared to the case of storing the original tick data matrix. In the next section, we review SOHAC, our clustering algorithm that *directly* optimize the storage space and therefore substantially outperforms conventional clustering algorithms for our problem.

3.4. SOHAC: Storage-Optimizing Hierarchical Agglomerative Clustering

SOHAC, Storage-Optimizing Hierarchical Agglomerative Clustering is designed for clustering columns of a tick data matrix. The algorithm builds on the hierarchical agglomerative strategy. Therefore, initially, all the objects belong to separate clusters. Then, clusters are iteratively merged together as long as the current number of clusters is more than k, the user-defined number of clusters. Therefore, at the end of this iterative process, k clusters are produced.

The key feature of our algorithm is that in each iteration it merges those two clusters that lead to minimal storage size of the decomposed matrix. This storage size can simply be calculated based on the binary change indicator matrix. For each examined clustering of the columns, we decompose the binary change indicator matrix. Then, we consider the rows that contain only zeros in the regular columns. The cells of such rows can be eliminated in the examined decomposition without loss of information. Therefore, in order to determine the number of cells required for the storage of the examined decomposition, we only need to count the cells in the rows that contain only zeros in their regular columns. The pseudocode of our algorithm is shown in Algorithm 1.

3.5. Querying Decomposed Tick-Data Matrix

Two common type of queries over a decomposed tick-data matrix are the point queries and the range queries. A point query retrieves the value of one or more features (columns) at a specific time point. A range query retrieves the values of one or more features within a given time range. In this section, we consider the cost to perform point and range queries, and how it is related to the number of clusters k.

Let C_p^s denote the corresponding cost to perform a point query that retrieves the value of a single feature at a specific time point. Also let m denote the number of rows in the original (uncompressed) matrix and r_s the compression ratio of the cluster containing the feature of interest. The compression ratio r_s is in the range $[0, 1]$, i.e., after decomposition, the number of rows in the cluster is $r_s m$ and thus smaller values of r_s indicate better compression. It is easy to see that $C_p^s \propto \log(r_s m)$, because the point query can retrieve the requested value by performing binary search in the cluster that contains the single feature of the query.[2] Each cluster already contains values sorted in time, thus the cost is logarithmic to $r_s m$, i.e., the number of rows in the cluster of interest. An increase of the number of clusters k will tend to improve the compression, thus will tend to reduce r_s. In this case, the cost C_p^s tends to reduce with increasing k.

For a point query that retrieves the values of *all* features at a specific time point, let C_p^a the corresponding cost to perform this query and r_a the compression ratio of the tick data table, i.e., the number of rows in a cluster is $r_a m$ *on average* (i.e., r_a is the average compression ratio of all compression ratios r_s for each individual column). In this case, $C_p^a \propto k \log(r_a m)$, because the query can retrieve the requested values for all features by performing binary search in all clusters

[2]We do not consider that values within clusters are indexed, because the updating of such indices would incur prohibitive overhead costs that are not feasible for rapidly changing tick data.

that contain the values of all features for the given point in time. As before, the query performs binary search in each cluster and there are k clusters in total. Therefore, an increase in the number of clusters k introduces a trade-off for C_p^a: larger k values cause a reduction of r_a and, thus, a reduction of C_p^a by a logarithmic factor (due to the $\log(r_a m)$ factor above); on the other hand, however, larger k values increase C_p^a by a linear factor (due to the k factor above). Based on our empirical results in Section 5.1 about the range of resulting r_a values w.r.t. to increasing k values, it can be derived that the reduction in C_p^a by a logarithmic factor due to smaller r_a is substantially outweighed by the increase by a linear factor due to larger k.

Similar conclusions can be made for range queries. Let C_r^a denote the cost of performing a range query that retrieves the values of all features within a range in time. A range query can be performed by performing binary searching in each cluster, in order to identify the starting point (as done in the case of a point query) and then by performing sequential retrieval of all subsequent values within the query range. If w is the average number of values retrieved during such a sequential searching in each cluster, then $C_r^a \propto k \log(r_a m) + kw$. This results in a similar trade-off as before. Moreover, for range queries with larger time ranges, w will tend to be large, which makes the increase in C_r^a to become larger by an additional factor linear to k due to the second term above (i.e., kw).

Therefore, in real-world applications, where very large matrices need to be decomposed, the number of clusters k should not get high, because this will incur prohibitive query costs in case of range and point queries that retrieve values for more features. As will be described during the presentation of our experimental results, reasonable k values are usually in the range $2 - 5$.

4. Speeding up SOHAC

Algorithm 1 shows SOHAC as we presented it in our previous work [11]. Next, we describe techniques that speed up SOHAC and allow its application to high-dimensional tick data.

4.1. Avoiding the recalculation of storage sizes

Note that, in each iteration, most of the clusters remain unchanged, except the ones that are merged in that iteration. Furthermore, merging two clusters only changes the storage of the columns belonging to the merged clusters. As most of the clusters remain unchanged in each iteration of SOHAC, at the end of an iteration, we do not need to recalculate the storage size of a cluster pair $(\mathcal{C}_i, \mathcal{C}_j)$ if both $\mathcal{C}_i$ and $\mathcal{C}_j$ remained unchanged in that iteration.

4.2. Lower Bounding Storage Size

In each iteration of SOHAC, those two clusters are merged that lead to minimal storage size. However, as we will show, in order to find out which pair of clusters lead to minimal storage size, it is not necessary to calculate the *exact* storage size of *all* pairs of clusters. More importantly, for many of *those* pairs that are affected by the merging steps at the end of the iteration (see line 13 in Algorithm 1), i.e., pairs for which the storage size changes, we can avoid the calculation of the *exact* storage size via *lower bounding* as described below.

In particular, it is unnecessary to calculate the *exact* storage size resulting from merging the clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ if we know that merging $\mathcal{C}_i$ and $\mathcal{C}_j$ results in a storage size of s_0 *at least* and there is an other pair of clusters $\mathcal{C}'_i$ and $\mathcal{C}'_j$ so that merging $\mathcal{C}'_i$ and $\mathcal{C}'_j$ results in storage size of *exactly* s and $s_0 > s$. This is shown in Algorithm 2.

Next, we develop a computationally cheap lower bound of the storage size which allows to speed up the SOHAC algorithm. Suppose we are given two clusters $\mathcal{C}_1$ and $\mathcal{C}_2$ containing columns $c_1, \ldots, c_n$ and $c_{n+1}, \ldots c_{n+m}$ respectively. Assuming uniform storage costs for all the cells of all the columns, the storage size resulting from merging $\mathcal{C}_1$ and $\mathcal{C}_2$ can not be less than $\frac{1}{2}(n+m)$-times the storage size of any pair $(c_i, c_j), 1 \leq i \leq n, n+1 \leq j \leq m$. With storage size of a pair (c_i, c_j) we mean the storage size associated with merging the two clusters that contain the single columns c_i and c_j respectively. According to the above observation, a lower bound of the storage size is

$$\text{StorageSize}(\mathcal{C}_1, \mathcal{C}_2) \geq \max_{c_i \in \mathcal{C}_1, c_j \in \mathcal{C}_2} \frac{1}{2} \cdot (n+m) \cdot \text{StorageSize}(c_i, c_j). \tag{1}$$

The calculation of this lower bound could still be computationally expensive when merging large column clusters $\mathcal{C}_1$ and $\mathcal{C}_2$ because according to (1) we should take into account all the pairs (c_1, c_2), $c_1 \in \mathcal{C}_1$, $c_2 \in \mathcal{C}_2$. In order to avoid it, in our current implementation, for each cluster we maintain a variable that always contains the identifier of the most frequently changing column belonging to that cluster and we calculate the lower bound according to those columns:

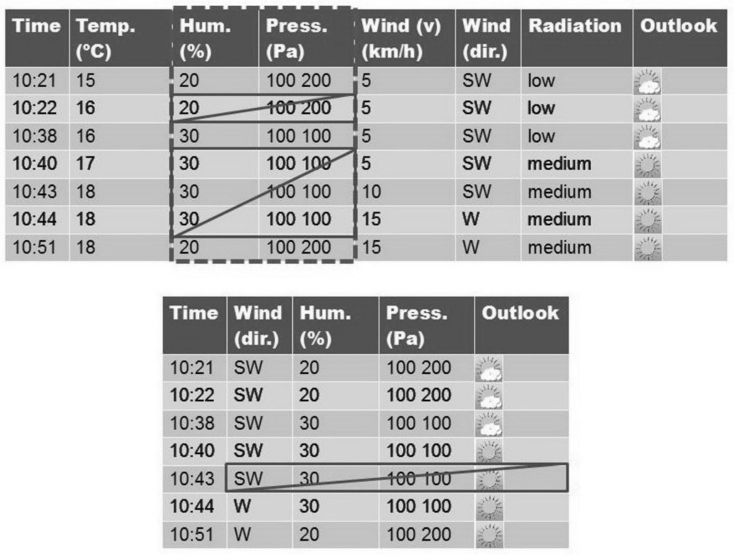
$$\text{StorageSize}(\mathcal{C}_1, \mathcal{C}_2) \geq \frac{1}{2} \cdot (n+m) \cdot \text{StorageSize}(\text{mfcc}(\mathcal{C}_1), \text{mfcc}(\mathcal{C}_2)), \tag{2}$$

where $\text{mfcc}(\mathcal{C})$ denotes the most frequently changing column of cluster $\mathcal{C}$.

This lower bound is illustrated in the example shown in Figure 3. We want to calculate the lower bound for merging the column clusters { Wind (direction), Humidity } and { Pressure, Outlook }. For the cluster { Wind (direction), Humidity }, both columns change twice, therefore, we can treat any of them as the most frequently changing column. For this example we will consider *Humidity*

Algorithm 2 QuickSOHAC: SOHAC with Lower Bounding of Storage Sizes

```
Require: Tick data matrix $M$, number of clusters $k$
Ensure: Clustering of the columns of $M$
 1: Construct the binary indicator matrix $I$ from $M$
 2: $P = \{\{c_1\}, \{c_2\}, ..., \{c_n\}\}$
 3: Initialize all the values $s[i][j]$ of a two-dimensional array as *unknown*, $s[i][j]$ is the storage size resulting
    from merging clusters $\mathcal{C}_i$ and $\mathcal{C}_j$.
 4: while $|P| > k$ do
 5:   $s \leftarrow \infty$    (Storage size for the best clustering found so far)
 6:   for all pairs of clusters $(\mathcal{C}_i, \mathcal{C}_j)$, with $\mathcal{C}_i \in P$, $\mathcal{C}_j \in P$ do
 7:     $\mathcal{C}'_i \leftarrow \mathcal{C}_i \cup \mathcal{C}_j$    (Merge clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ into the new cluster $\mathcal{C}'_i$ )
 8:     $P' \leftarrow P \setminus \{\mathcal{C}_i, \mathcal{C}_j\} \cup \{\mathcal{C}'_i\}$
 9:     if $s[i][j]$ is unknown then
10:       $s_0 \leftarrow$ lower bound of the storage size resulting from merging clusters $\mathcal{C}_i$ and $\mathcal{C}_j$
11:       if $s_0 < s$ then
12:         calculate the exact storage size resulting from merging clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ and set the value of
            $s[i][j]$ to the calculated storage size
13:         if $s[i][j] < s$ then
14:           $P^* \leftarrow P'$,  $s \leftarrow s[i][j]$ ,  $i^* \leftarrow i$ ,  $j^* \leftarrow j$
              (The best clustering found so far)
15:         end if
16:       end if
17:     else
18:       if $s[i][j] < s$ then
19:         $P^* \leftarrow P'$,  $s \leftarrow s[i][j]$ ,  $i^* \leftarrow i$ ,  $j^* \leftarrow j$
            (The best clustering found so far)
20:       end if
21:     end if
22:     $P \leftarrow P^*$
23:     Let the merged cluster be the new $i^*$-th cluster, while the $j^*$-th cluster is not used anymore:
        $\mathcal{C}_{i^*}^{new} = (\mathcal{C}_{i^*} \cup \mathcal{C}_{j^*})$. Set the storage sizes to unknown for all the pairs that include the new cluster:
        $s[i^*][.] \leftarrow$ unknown, $s[.][i^*] \leftarrow$ unknown
24:   end for
25: end while
26: return $P$
```



Time	Temp. (°C)	Hum. (%)	Press. (Pa)	Wind (v) (km/h)	Wind (dir.)	Radiation	Outlook
10:21	15	20	100 200	5	SW	low	⛅
10:22	16	~~20~~	~~100 200~~	5	SW	low	⛅
10:38	16	30	100 100	5	SW	low	⛅
10:40	17	~~30~~	~~100 100~~	5	SW	medium	☀
10:43	18	~~30~~	~~100 100~~	10	SW	medium	☀
10:44	18	~~30~~	~~100 100~~	15	W	medium	☀
10:51	18	20	100 200	15	W	medium	☀

Time	Wind (dir.)	Hum. (%)	Press. (Pa)	Outlook
10:21	SW	20	100 200	⛅
10:22	SW	20	100 200	⛅
10:38	SW	30	100 100	⛅
10:40	SW	30	100 100	☀
10:43	~~SW~~	~~30~~	~~100 100~~	~~☀~~
10:44	W	30	100 100	☀
10:51	W	20	100 200	☀

Figure 3: Example illustrating lower bounding. According to the matrix shown in the top, the storage size for the columns *Humidity* and *Pressure* is $3 \cdot 2 = 6$ (the number of cells that need to be stored when merging the clusters containing these single columns). According to the matrix in the bottom, the actual storage size resulting from merging the clusters { Wind (direction), Humidity } and { Pressure, Outlook } is $6 \cdot 4 = 24$ (without the index column). This actual storage size is more than the resulting lower bound which is $\frac{1}{2} \cdot (2 + 2) \cdot 6 = 12$.

as the most frequently changing column of the cluster { Wind (direction), Humidity }. Regarding the other cluster, *Pressure* changes twice, while *Outlook* changes only once, therefore, *Pressure* is the most frequently changing column of cluster { Pressure, Outlook }. In the top of the Figure 3, we see the storage size resulting from merging the columns *Humidity* and *Pressure* which is $3 \cdot 2 = 6$, i.e., when merging the clusters that contain only the single columns *Humidity* and *Pressure* 6 cells of the original tick data matrix need to be stored. For simplicity, the storage cost of the index column (*Time*) is ignored throughout the example. According to (2), when merging the clusters { Wind (direction), Humidity } and { Pressure, Outlook }, the resulting storage size is at least $\frac{1}{2} \cdot (2 + 2) \cdot 6 = 12$. The actual storage size is 24, as shown in the Figure 3.

Note that in the first iteration of QuickSOHAC, i.e. when each cluster consists of a single column, the lower bounds are exactly the same as the storage sizes. We take this into account in our implementation: in the *first* iteration, we always set the value of $s[i][j]$ in line 12 of Algorithm 2 independently of the outcome of the previous conditional statement.

Table 1: The tick data tables used in our experiments, together with the number of their rows and columns (without the index column).

Data table	Rows	Columns
TD-1	408 042	30
TD-2	554 854	190
TD-3	50 000 000	33

4.3. Sampling

In order to further speed-up the approach, we can run SOHAC on a small sample, such as 5% or 10% of the entire tick data matrix. While this results in substantial speed-up, according to our observations presented in Section 5, the resulting compression ratios are only slightly worse compared to the case of using the entire tick data table.

5. Experiments

We evaluated our algorithm on three real-world tick data tables provided by the world's leading investment bank. The basic properties of these tables, number of columns and rows are shown in Table 1. First, we compare the decomposition produced by our approach and other clustering algorithms in terms of compression ratio. Subsequently, we focus on the runtime of SOHAC and QuickSOHAC, i.e., to which extent the proposed lower bounding technique allows to speed up the algorithm. We also present results for the case when we sampled the data. Finally, we show that our approach can be combined with conventional compression methods: in particular, we show that compression of the tables with gzip[3] may also benefit from the decomposition produced by our approach.

5.1. Compression Ratio

In the first experiment, we compared the decomposition of a tick data matrix resulting from the clusters produced by our approach, SOHAC, to the decomposition of the same tick data matrix using existing and widely used clustering algorithms. We measured the quality of decompositions in terms of compression ratio (CR), i.e., the ratio of the number of cells in regular columns after

[3]http://www.gzip.org/

the decomposition and the number of cells in regular columns in the original matrix:

$$CR = \frac{\text{number of cells in regular columns after decomposition}}{\text{number of cells in regular columns in the original matrix}}$$

An example for the calculation of compression ratio can be found in Section 3.1.

As our approach, SOHAC, is built on hierarchical agglomerative clustering, in our experiments we focused on comparing the clustering produced by SOHAC to the clustering produced by different variants of hierarchical agglomerative clustering algorithms. Additionally, we compared the clustering produced by SOHAC to the clustering produced by k-Means [20]. We tested two variants of k-Means, the first one used Euclidian distance, while the second one used Manhattan distance.

Regarding the variants of hierarchical agglomerative clustering algorithms, we tested Single Linkage, Complete Linkage and Average Linkage with the following proximity measures: Euclidean Distance, Cosine Similarity, Chebychev Distance and Manhattan Distance. We used the Weka-implementations of all of the above baseline algorithms [20]. In total, taking all the examined variants of the baselines into account, we compared our approach to 14 clustering algorithms from the literature. To ensure fair comparison, all of the baselines, as well as our approach, used the binary change indicator matrices as input.

First, we decomposed the entire *TD-2* tick data table into $k = 2$, 3, 4 and 5 clustering using SOHAC and the baselines. Table 2 shows the resulting compression ratios. As we can see, SOHAC clearly outperforms the baselines for all the examined cases.

Subsequently, in order to be able to study whether the improvements are systematic and statistically significant, we split the entire tick data matrix into 10 disjoint sub-matrices, and we repeated all the experiments 10 times. In each of the 10 rounds of the process, we used a different sub-matrix, and clustered the columns of that sub-matrix. Therefore, we could calculate the average and standard deviation of the compression ratio.

In the left of Figure 4, we show the compression ratio and its standard deviation resulting from the decomposition into $k = 2$ clusters using our approach, SOHAC, and the baselines. In the figures, SingleLink, CompleteLink and AverageLink are denoted by H-SL, H-CL and H-AL respectively. For SingleLink, CompleteLink, AverageLink and k-Means, we only show the best-performing variants. In the right of Figure 4, we show the compression ratios as function of the number of clusters. SOHAC achieves the best compression ratio in all cases. Especially for *TD-3*, i.e., the largest among the three data sets, the difference is substantial, which indicates the suitability of SOHAC for tick data of very large size. With increasing k values, as expected, all methods converge to

Table 2: Compression ratios on the entire *TD-2* tick data table for the decompositions using our approach, SOHAC, and the baselines.

Approach	Distance measure	$k = 2$	$k = 3$	$k = 4$	$k = 5$
SingleLink	Euclidean	0.999	0.842	0.694	0.692
	Manhattan	0.999	0.842	0.694	0.692
	Cosine	0.947	0.938	0.313	0.245
	Chebyshev	0.590	0.580	0.574	0.522
CompleteLink	Euclidean	0.999	0.693	0.532	0.528
	Manhattan	0.999	0.693	0.532	0.528
	Cosine	0.947	0.269	0.247	0.167
	Chebyshev	0.590	0.580	0.575	0.522
AverageLink	Euclidean	0.999	0.842	0.694	0.528
	Manhattan	0.999	0.842	0.694	0.528
	Cosine	0.947	0.323	0.311	0.189
	Chebyshev	0.590	0.580	0.574	0.522
k-Means	Euclidean	0.709	0.269	0.232	0.173
	Manhattan	0.999	0.989	0.884	0.474
SOHAC		0.276	0.141	0.122	0.101

comparable compression ratios. Nevertheless, based on the discussion in Section 3.5, larger values of k hinder the efficient execution of queries over the compressed tick data. For this reason, in real-world application the focus is only on small values of k, usually equal to 2 or not much higher (i.e., in the range $2-5$). It is worth to notice that in these cases SOHAC presents a clear advantage compared to all examined baselines.

5.2. Runtime

In order to evaluate the proposed lower bounding technique, we compared the runtimes of two variants of our approach. By SOHAC we denote the variant that is implemented according to Algorithm 1, but avoids the unnecessary recalculation of storage sizes (see Section 4.1). QuickSOHAC additionally uses the proposed lower bounding technique (see Algorithm 2). The runtimes relative to the runtime of QuickSOHAC are shown in Figure 5. We can see that, in terms of runtime, SOHAC is comparable to the baselines even without the proposed lower bounding. The proposed lower bounding leads to 4-6 fold speedup, and therefore QuickSOHAC is faster than the baselines in the vast majority of the examined cases. This means that QuickSOHAC is overall the best performing method, because not only it outperforms all baselines in terms of compression ratios, but it also compares favorable in terms of running time, which makes it suitable for large scale

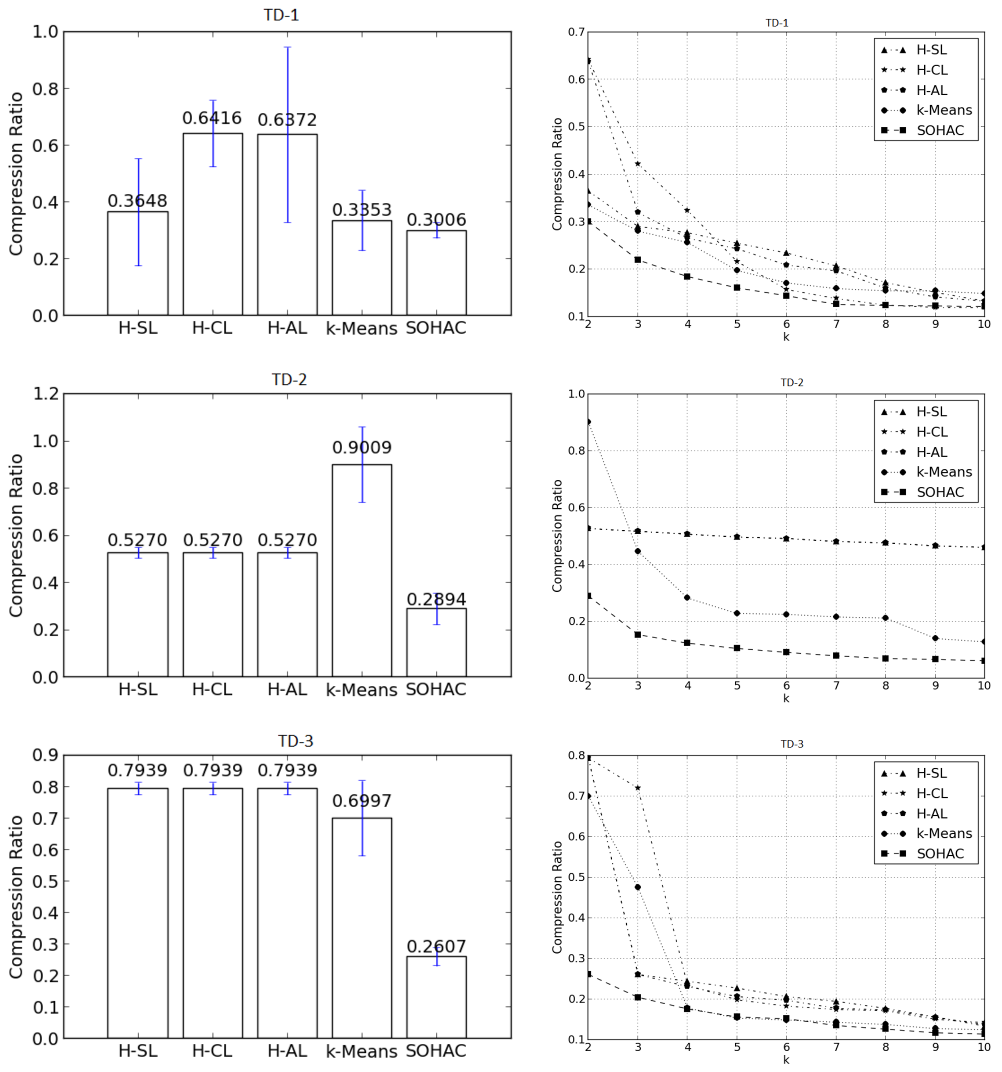


Figure 4: The compression ratio and its standard deviation resulting from the decomposition into $k = 2$ clusters using our approach, SOHAC, and the baselines (in the left). The compression ratio as function of the number of clusters using our approach, SOHAC, and the baselines (in the right). Note that the curves H-SL, H-CL and H-AL overlap for the TD-2 dataset.

applications.

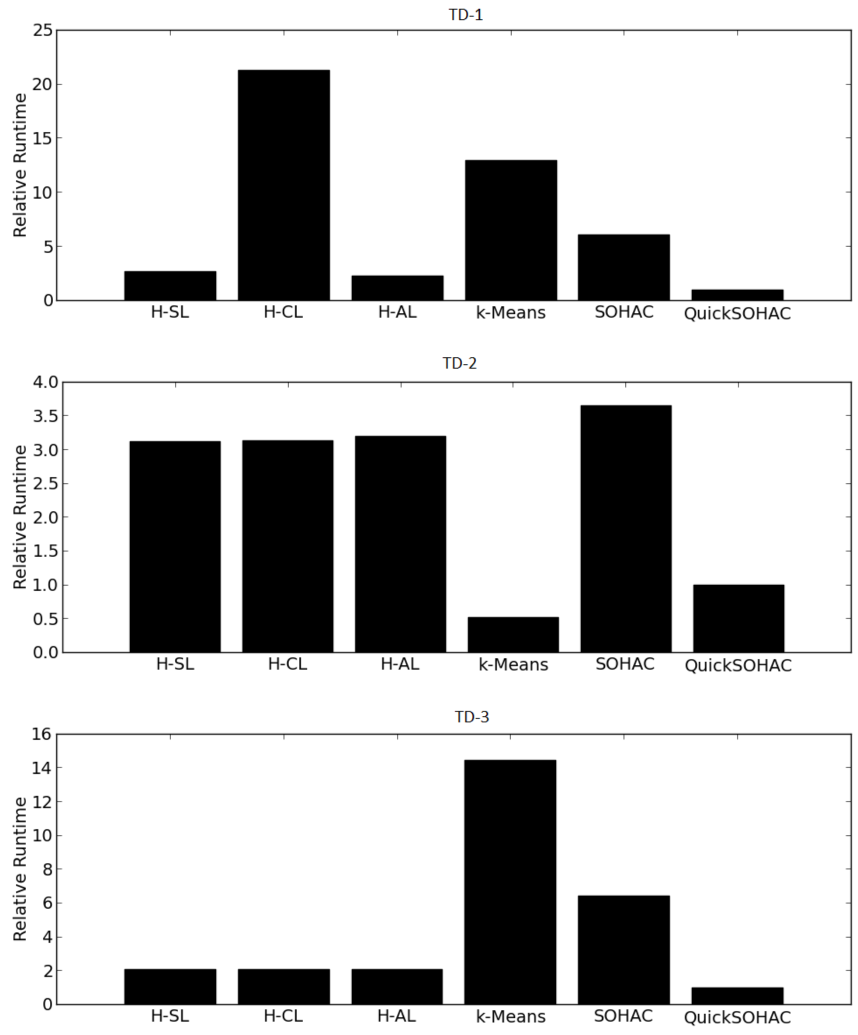


Figure 5: Relative runtimes compared to the runtime of QuickSOHAC. For the baselines, we only show the runtime of the best performing variant w.r.t. compression ratio.

Furthermore, we examined the effect of sampling: in our experiments, we sampled the first 5% and 10% of the tick data matrices, while the effect of sampling in terms of compression ratio is shown in Figure 6. As shown, using a relatively small portion of the entire tick data table, such as 5% or 10%, allows for additional speedup, while maintaining almost as good compression ratios as if we would use the entire tick data table.

5.3. Combination with conventional compression methods

In order to measure physical storage sizes, i.e., the amount of disk space occupied by the tables, we stored both the original binary change indicator matrices and the decomposed ones both in

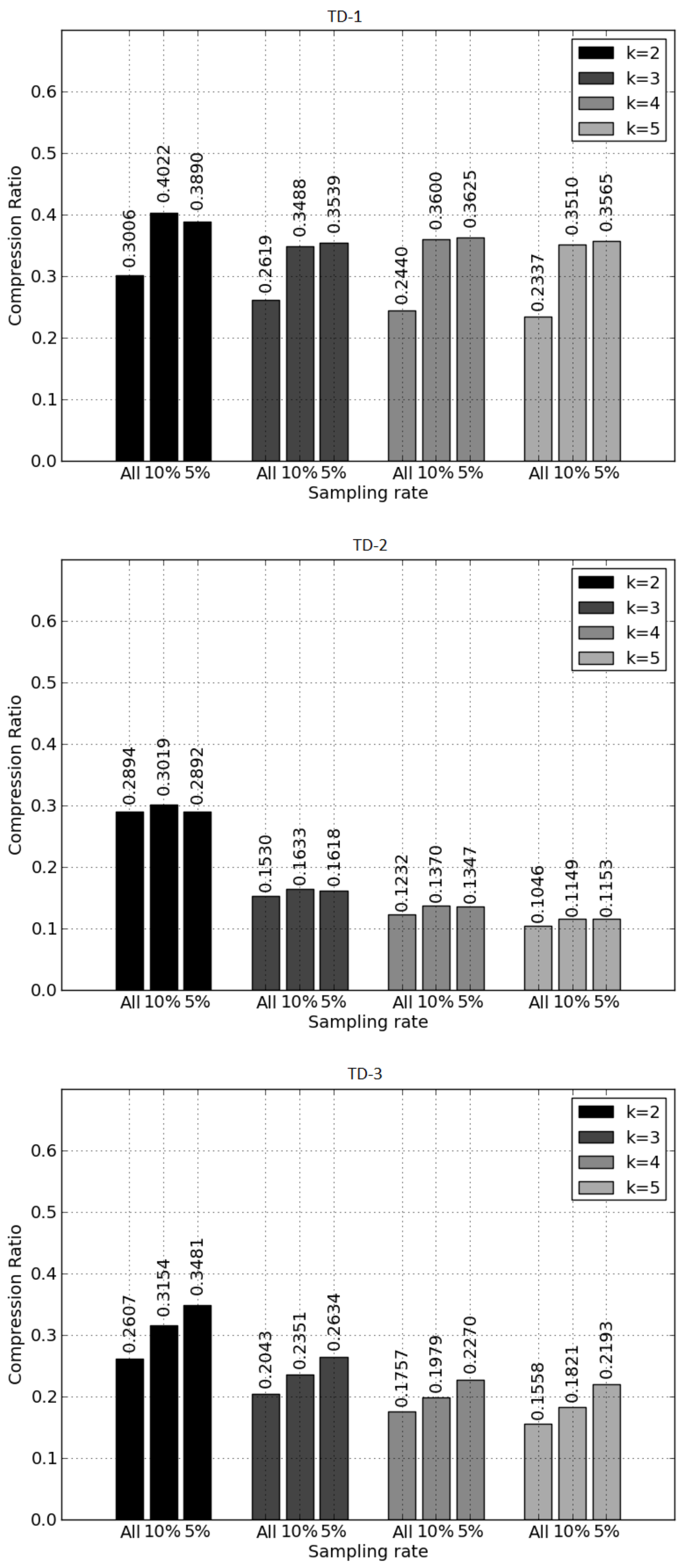


Figure 6: Compression ratio of SOHAC for the cases of using the entire tick data table, sampling 10% and 5% of the table.

simple text files and gzip-compressed text files. The results are shown in Table 3. As we can see, SOHAC outperforms the baselines in both contexts, i.e., storage as simple text file and gzip-compressed storage. Furthermore, we emphasize that SOHAC with gzip leads to substantially reduced physical storage size compared to the case of compressing the entire table with gzip. This indicates that SOHAC may be able to decompose the table along such patterns which could not be identified by gzip.

Table 3: The physical storage sizes (in kBytes) of the binary change indicator matrices and their standard deviation-with and without the usage of gzip.

Data table	Approach	with gzip			without gzip		
TD-1	no decomposition	367	±	20	4849	±	32
	Single Link	349	±	21	3344	±	451
	Complete Link	466	±	32	4743	±	457
	Average Link	372	±	37	4210	±	684
	k-Means	367	±	40	3416	±	427
	SOHAC	<u>351</u>	±	<u>23</u>	<u>3221</u>	±	<u>78</u>
TD-2	no decomposition	691	±	5	23892	±	10
	Single Link	583	±	5	14155	±	2106
	Complete Link	583	±	5	14155	±	2106
	Average Link	583	±	5	14155	±	2106
	k-Means	812	±	61	23330	±	3564
	SOHAC	<u>568</u>	±	<u>25</u>	<u>9617</u>	±	<u>1464</u>
TD-3	no demcomposition	4326	±	75	61436	±	277
	Single Link	4288	±	82	54795	±	340
	Complete Link	4288	±	82	54795	±	340
	Average Link	4288	±	82	54795	±	340
	k-Means	5545	±	366	61149	±	6048
	SOHAC	<u>4045</u>	±	<u>97</u>	<u>37610</u>	±	<u>837</u>

5.4. Discussion

As the experiments show, the proposed algorithm, SOHAC, performs favorably to conventional clustering algorithms on the task it was designed for, i.e., decomposition of tick data matrices. The proposed lower bounding allows to speed up the algorithm by a factor of 4-6. Due to the ever-growing datasets, and the rate of growth in particular, this can be considered relevant even in the light of Moore's law which states that the computational performance doubles every 18 months. Finding the decomposition quickly may also be relevant in real-time applications. With the significant speed up a lower utilization of current computing resources could be achieved. Furthermore, our experiments illustrate that our approach can be successfully used together with conventional compression algorithms, which is relevant in applications where the final compression rate is more important than the execution time of queries.

Regarding the limitations of the proposed approach, we point out that SOHAC is clearly not a general-purpose clustering algorithm, but it was designed for a particular task. While SOHAC (or a modified version of it) might be applied to cluster non-tick data, due to its design, we only expect SOHAC to perform well in cases when the data has at least some relevant aspects in common with tick data, while the adaptation of SOHAC to such new data types is not trivial.

6. Conclusion

In this paper, we focused on the storage of tick data, a type of data appearing in various applications from finance to meteorological observations. Due to the rapid change of the values of the observed features the size of tick data tables tends to increase substantially. We proposed a storage scheme for tick data that reduces the storage space while, at the same time, it allows to efficiently execute queries. The proposed approach performs a decomposition of the original tick data matrix into a number of smaller matrices. We achieve this decomposition by clustering the columns of original matrix based on a new clustering algorithm called Storage-Optimizing Hierarchical Agglomerative Clustering (SOHAC). In this paper, we focused on the efficient implementation of SOHAC. Our observation about the lower bound of storage sizes allowed to speed up SOHAC. This is essential for the application of SOHAC to high-dimensional tick data. Our experimental evaluation demonstrated that the proposed approach compares favorably to several baselines in terms of compression, whereas the lower bounding technique can lead to substantial speedup.

The benefits due to the proposed approach are important, due to the several application domains that are based on the ability to efficiently store and analyse tick data. For instance, major financial institutions record historical stock-market values in the form of tick data, which can be even publicly available and help analysts working in financial services in developing risk management, trading, and quantitative analysis.[4] Recording of such historical data is performed for many years and, thus, is of massive volumes, since in order “to store decades of market data, the volume of disk

[4]See: http://www.xignite.com/market-data/nasdaq-historical-stock-tick-data/

space needed expands geometrically"[5]. Since uncompressed tick data is voluminous, in order to perform useful analysis the proposed compression scheme offers the ability to reduce their storage requirements and to enable their faster processing based on in-memory architectures. Moreover, an appealing option nowadays for storing large volumes of tick data is to follow a cloud-based solution. Storage as a service (STaaS) is increasingly being used by many organizations that manage such large data collections. STaaS allows such organizations to reduce the total cost of ownership by renting storage space from a service provider. However, the costs involved with STaaS usually increase proportionally to the size of the data. Moreover, tick-data are of very large sizes and they grow rapidly. Thus, the compression attained by the proposed scheme can attain significant cost reductions by optimizing the storage of such large and rapidly-changing tick data.

The proposed approach is based on the premise that the "pattern of change" in the tick data remains stable, which allows SOHAC to detect clusters over the entire tick-data matrix. In the long term, however, this "pattern of change" may vary and may require to update the clustering scheme of SOHAC. For this reason, in our future work we plan to investigate extensions of SOHAC that will split the original matrix into clusters, each of which can be effectively represented by a single clustering scheme. The detection of such clusters in a dynamic way, i.e., as new data are arriving, is also an interesting point of future work. Instead of storing the exact signal, in many applications, an approximation of the signal may be sufficient. Therefore, we aim to study SOHAC in context of approximate storage of multivariate signals. Furthermore, we want to explore whether SOHAC can be adapted for the clustering of (multivariate) time-series. The key of this application is a technique that converts time-series into a binary vector reflecting at which positions substantial changes happen. While a naive approach using a fixed threshold may not work well enough, approaches based on machine learning, such as segmentation of the time-series by an appropriate Hidden Markov Model could potentially provide such a representation. We hope to explore the possibilities of the usage of QuickSOHAC clustering algorithm in more general datasets, that are binary in nature, for example document-term matrices in text mining and extend the algorithm to bi-cluster documents and frequently appearing terms in documents. We aim at comparing these results to existing bi-clustering algorithms.

[5]See: http://marketsmedia.com/tick-tick/

Acknowledgment

We thank DAAD and MÖB for financially supporting a researcher exchange program with project number: 39859.

References

[1] S. Ahmad, T. Taskaya-Temizel, K. Ahmad, Summarizing time series: Learning patterns in volatileseries, Intelligent Data Engineering and Automated Learning–IDEAL 2004 (2004) 523–532.

[2] Q. Akram, D. Rime, L. Sarno, Does the law of one price hold in international financial markets? evidence from tick data, Journal of Banking & Finance 33 (2009) 1741–1754.

[3] E. Barany, M.B. Varela, I. Florescu, I. Sengupta, Detecting market crashes by analysing long-memory effects using high-frequency data, Quantitative Finance 12 (2012) 623–634.

[4] R. Bartiromo, Dynamics of stock prices, Physical Review E 69 (2004) 067108.

[5] M. Blais, P. Protter, Signing trades and an evaluation of the leeready algorithm, Annals of Finance 8 (2012) 1–13.

[6] K. Buza, A. Buza, P. Kis, A distributed genetic algorithm for graph-based clustering, Man-Machine Interactions 2 (2011) 323–331.

[7] Á. Cartea, Derivatives pricing with marked point processes using tick-by-tick data, Quantitative Finance 13 (2013) 111–123.

[8] G. Dionne, P. Duchesne, M. Pacurar, Intraday value at risk (ivar) using tick-by-tick data with application to the toronto stock exchange, Journal of Empirical Finance 16 (2009) 777–792.

[9] S. Guha, R. Rastogi, K. Shim, Rock: A robust clustering algorithm for categorical attributes, Information Systems 25 (2000) 345–366.

[10] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman, A. Wu, An efficient k-means clustering algorithm: Analysis and implementation, Pattern Analysis and Machine Intelligence, IEEE Transactions on 24 (2002) 881–892.

[11] G. Nagy, K. Buza, Sohac: Efficient storage of tick data that supports search and analysis, in: P. Perner (Ed.), Advances in Data Mining. Applications and Theoretical Aspects, volume 7377 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2012, pp. 38–51.

[12] A. Nanopoulos, H. Gabriel, M. Spiliopoulou, Spectral clustering in social-tagging systems, Web Information Systems Engineering-WISE 2009 (2009) 87–100.

[13] K. Oh, K. Kim, Analyzing stock market tick data using piecewise nonlinear model, Expert Systems with Applications 22 (2002) 249–255.

[14] T. Ohnishi, T. Mizuno, K. Aihara, M. Takayasu, H. Takayasu, Statistical properties of the moving average price in dollar–yen exchange rates, Physica A: Statistical Mechanics and its Applications 344 (2004) 207–210.

[15] T. Qiu, G. Chen, L.X. Zhong, X.R. Wu, Dynamics of bid–ask spread return and volatility of the chinese stock market, Physica A: Statistical Mechanics and its Applications 391 (2012) 2656–2666.

[16] D. Salomon, Data compression: the complete reference, Springer-Verlag New York Inc, 2004.

[17] N. Sazuka, Analysis of binarized high frequency financial data, The European Physical Journal B-Condensed Matter and Complex Systems 50 (2006) 129–131.

[18] M. Takayasu, H. Takayasu, M. Okazaki, Transaction interval analysis of high resolution foreign exchange data, Empirical Science of Financial Fluctuations-The Advent of Econophysics 18 (2002) 25.

[19] P. Tan, M. Steinbach, V. Kumar, et al., Introduction to data mining, Pearson Addison Wesley Boston, 2006.

[20] I. Witten, E. Frank, Data Mining: Practical machine learning tools and techniques, Morgan Kaufmann, 2011.

[21] Z. Wu, On the intraday periodicity duration adjustment of high-frequency data, Journal of Empirical Finance 19 (2012) 282–291.

[22] R. Xu, D. Wunsch, et al., Survey of clustering algorithms, Neural Networks, IEEE Transactions on 16 (2005) 645–678.

[23] Y. Yabuuchi, J. Watada, Formulation of possibility grade-based fuzzy autocorrelation model and its application to forecasting, International Journal of Intelligent Technologies and Applied Statistics 5 (2012) 321–335.