Contents lists available at ScienceDirect

Computers in Industry

journal homepage: www.elsevier.com/locate/compind

Lossless compression of industrial time series with direct access

Adrián Gómez-Brandón[a,*], José R. Paramá[a], Kevin Villalobos[c], Arantza Illarramendi[b], Nieves R. Brisaboa[a]

[a] Universidade da Coruña, CITIC, A Coruña, Spain
[b] University of the Basque Country UPV/EHU, Donostia – San Sebastián, Spain
[c] Ikerlan Technology Research Centre, Basque Research and Technology Alliance (BRTA), Arrasate-Mondragón, Spain

ARTICLE INFO

Article history:
Received 12 March 2021
Received in revised form 8 June 2021
Accepted 14 June 2021
Available online 25 June 2021

Keywords:
Compression
Smart manufacturing
Time series

ABSTRACT

The new opportunities generated by the data-driven economy in the manufacturing industry have caused many companies opt for it. However, the size of time series data that need to be captured creates the problem of having to assume high storage costs. Moreover, these costs, which are constantly growing, begin to have an impact on the profitability of companies. Thus, in this scenario, the need arises to develop techniques that allow obtaining reduced representations of the time series. In this paper, we present a lossless compression method for industrial time series that allows an efficient access. That is, our aim goes beyond pure compression, where the usual way to access the data requires a complete decompression of the dataset before processing it. Instead, our method allows decompressing portions of the dataset, and moreover, it allows direct querying the compressed data. Thus, the proposed method combines the efficient access, typical of lossy methods, with the lossless compression.

1. Introduction

The application of data processing and exploitation technologies in different sectors, aided by the widespread promotion of Big Data tools and other synergistic technologies like Cloud Computing and the Internet of Things (IoT), has led to the concept of a data-driven economy (European Commission, 2014) being coined as one of the cornerstones of global economic development. Furthermore, the manufacturing industry is one of the key targets where this data-driven economy is being deployed globally, leading to the emergence of the idea of Smart Manufacturing as a global-scale overarching term encompassing numerous initiatives and strategies (e.g., Industry 4.0 (Kagermann et al., 2013)) addressing the use of data exploitation for optimizing and transforming manufacturing businesses. In this way, data has emerged as a significant facilitator for boosting manufacturing competitiveness around the world in recent years, and companies have begun to appreciate the strategic relevance of data (Tao et al., 2018).

In Smart Manufacturing scenarios, the majority of the acquired data are time series generated by large-scale sensor networks monitoring the continuous functioning of the manufacturing processes or equipment to be analyzed. As a result, the volume of acquired data (i.e. time series data from sensors collected 24 hours a day, 7 days a week) is rapidly increasing (Risse, 2018; Zhou, 2019). Such is the growth pace that in 2015 the manufacturing industry was already generating more than 1000 EB of data annually and it is expected to increase by 20 times in the next 10 years (Yin and Kaynak, 2015). As a consequence, problems are arising related to the considerable costs associated with the resources needed to store them (Villalobos et al., 2020, 2019).

In order to reduce data storage costs, the oldest data is routinely eliminated to make room for new data, resulting in the loss of valuable historical data for further analytic processes (Amazon, 2020). Long-term data, on the other hand, is acknowledged as critical in Smart Manufacturing scenarios (Kusiak, 2017).

Time series reduction techniques and time series compression techniques are the two primary types of approaches that can be used to generate reduced representations of time-series data. The first type are often lossy techniques, which means that when rebuilding a time series from its reduced representations, some data is lost. These techniques are primarily aimed at lowering the dimensionality of time-series data in order to facilitate further data analysis. The second type are lossless compression techniques, which focus on encoding data in a much more compact format that saves storage space without loosing data. One problem of these

* Corresponding author at: Facultade de Informática, Campus de Elviña s/n, 15071 A Coruña, Spain
E-mail addresses: adrian.gbrandon@udc.es (A. Gómez-Brandón), jose.parama@udc.es (J.R. Paramá), kvillalobos@ikerlan.es (K. Villalobos), a.illarramendi@ehu.eus (A. Illarramendi), brisaboa@udc.es (N.R. Brisaboa).

https://doi.org/10.1016/j.compind.2021.103503

final types of techniques is that they usually require decompressing the entire dataset in order to access the data (which can be a slow process that hinders the management of the data for the analysis).

In this work, we present *Direct Access Compression of time series* (DACTS), a new lossless compression method designed for time series from industrial environments. DACTS is based on a grammar-based compressor, RePair (Larsson and Moffat, 2000). Grammar compressors use a grammar to replace frequent sequences of original symbols with new shorter symbols. Thus, although the application of the proposed method (DACTS) could be interesting for different types of time series, it is in environments such as Smart Manufacturing where this type of method acquires a greater relevance. Some reasons for this are: (1) industrial time series, are usually large time series captured by sensor networks operating almost uninterruptedly, that contain numeric measurements of a variety of equipment parameters and physical magnitudes. In those type of series, repeating patterns can be found that could benefit the reduction potential of DACTS method. (2) In many cases, industrial machine controllers are programmed in an inefficient way, in terms of capturing data for analytical purposes. For example, sometimes, they may be sending a constant value for several hours, to indicate that the machine is operating in manual mode and these data are stored anyway, occupying an unnecessarily space that increases data storage costs. Therefore, the reduction potential of DACTS method can also be interesting in this situation.

However, the normal RePair does not allow direct access, that is, given a position of the original file, we do not know where it is represented in the compressed file, and thus, if we want to find it, we have to decompress from the beginning until finding that position. In order to overcome this limitation, we added additional data structures to achieve that capability of direct access. Moreover, to be able to efficiently query the compressed data, simple direct access is not enough, therefore we have also included other data structures and designed efficient query algorithms. In summary, the main characteristics of the proposed technique, are: it performs an efficient lossless compression of time-series, it allows decompressing portions of the time series without the need for decompressing from the beginning and, it is capable of efficiently querying the compressed data. This opens the way to run analysis queries directly on lossless compressed data, breaking in this way, the traditional dichotomy in the field of industrial time series storage, where if a lossless compression technique is used, analysis procedures could require too much time to be feasible, whereas if a lossy type reduction technique is used, some data are lost.

In order to show the behaviour of the proposed technique we present two different experiments; first, we compare our method to the state of the art in pure compression. These methods have to decompress the complete dataset in advance to query them. In this case, at the price of slightly worse compression where data are highly repetitive, DACTS is up to 20 times faster when querying. In the second experiment, we compared DACTS with a real time series database management system. In this case, our method was up to 1300 times faster when answering the queries.

The outline of the paper is as follows. In Section 2 some related work that consider on the one hand, techniques to generate reduced representations of time series data; and on the other hand, several previous data structures defined to deal with compressed information, are presented. In Section 3 the main technical details related to DACTS are explained. In Section 4 an experimental evaluation is presented. Finally, Section 5 shows our conclusions and directions for future work.

2. Related work

In this section, we present first some works that have considered data reduction and compression techniques for time series, and then some proposals of data structures defined to deal with compressed information.

2.1. Techniques to generate reduced representations of time series

Data reduction and compression techniques represent a resource with the potential to reduce the cost associated with the storage of time series, by obtaining a reduced or compressed representation of the data which is much smaller in volume than the original data, while maintaining the information as complete as possible. If some information is lost, then, the compression will be lossy, whereas if the original information can be recovered from the reduced representation, the compression will be lossless.

With regard to time series reduction techniques, besides allowing to obtain reduced representations of the time series, they also make it possible to optimize some data analysis techniques associated with time series processing (Aghabozorgi et al., 2015; Lin et al., 2003), such as time series similarity search, time series clustering, and time series data mining, where most of the algorithms scale poorly to high-dimensional data. Various studies have addressed the representation of time series through the application of reduction or approximation techniques to time-series data. For example, in Fu (2011), a very thorough classification of different techniques for the reduced representation of time-series data is provided, grouping them in families and identifying the most representative techniques in each family. Indeed, the selection of reduction and approximation techniques that are analyzed and compared is similar across various references discussing time series data mining (Fu, 2011; Wang et al., 2013; Esling and Agon, 2012; Palpanas et al., 2004; Gordevičius et al., 2012).

Moreover, it is worth mentioning that this type of techniques is mainly designed to preserve enough information about the time series to support indexing or specific data mining algorithms over pre-processed datasets, rather than compressing the raw time series (Blalock et al., 2018). As a result, they are techniques that generate loss (which could hamper the analysis of the data) and usually targeted for a particular analysis scenario (e.g., a particular level of down-sampling may not work well for all kind of analysis that are susceptible to be applied over a time series). For that reason, other approaches are advocating for compression techniques that preserve what it is given (i.e., lossless compression) and leave pre-processing (e.g., dimensionality reduction or frequency filtering) for further processes.

With regard to lossless compression techniques for time-series data, different approaches have been appearing during the last years that allow representing the time-series data in a much more compact representation without losing data. In (Blalock et al., 2018), these approaches are reviewed together with the most representative techniques for each for them. The reviewed approaches include, among others, XOR differentiation, directly applying general-purpose compressors, delta encoding and then applying integer compressors, or predictive coding and byte-packaging. Moreover, *Sprintz* is also presented as the state-of-the-art algorithm developed for the Internet of Things time series, tending to achieve higher compression ratios and decompression speed (Blalock et al., 2018).

However, most of those techniques do not support random access, and thus, all the data must be decompressed even when accessing a small portion of it. Compression techniques that allow accessing the data without requiring to decompress all the data are thus desirable, as they allow data to be stored in a compressed format while also allowing applications to perform random

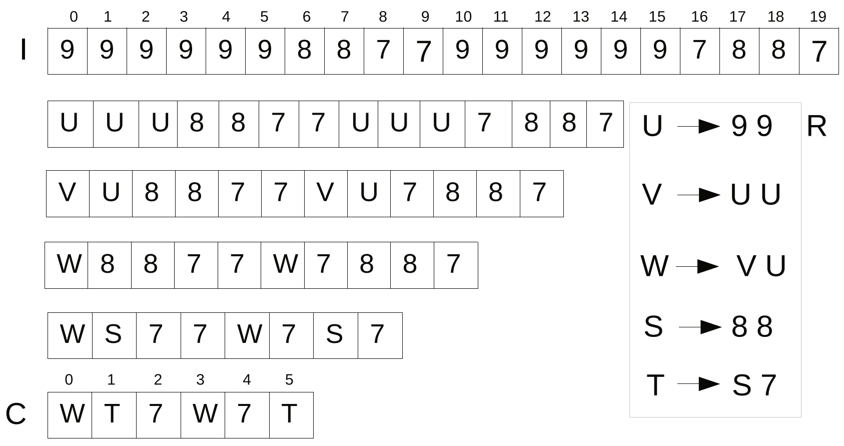


Fig. 1. Example of Re-Pair compression.

data access decompression (Silva de Moura et al., 2000; Brisaboa et al., 2007, 2008, 2013; Klein and Shapira, 2016; Külekci, 2014; Vestergaard et al., 2020). Specifically for time series, in (Vestergaard et al., 2020), it is shown a compression technique with random access, that is, it is capable of recovering portions of the original data, that later should be processed with a normal algorithm for uncompressed data. Instead, there is a new family of methods providing *direct access* that are equipped with specific algorithms to solve queries directly on the compressed data, that is, without a decompression procedure (Silva de Moura et al., 2000; Navarro, 2016). In this work, we present a compression method for time series with that capability.

2.2. *Compact data structures*

In the last years, a significant research effort has been made to design data structures that not only store compressed information with random access, but are capable to run complex queries directly on the compressed data. In order to approach, and sometimes overcome, the querying performance on uncompressed data, these methods do not always decompress the affected zone and then execute the query on the decompressed data, which could be clearly slower than querying the uncompressed data, but they are able to understand the compressed data and therefore execute the query on them. Moreover, in many cases, indexes or other types of data structures are added to the compressed data to speed up the querying process. These methods are called *compact data structures* (Navarro, 2016).

DACTS makes use of two previous techniques that are shown next.

2.2.1. *Re-Pair*

Re-Pair is a compression method based on creating a grammar and using it to replace pairs of symbols of the original sequence with a new symbol defined by the grammar. In our case, we consider a sequence of integers I (called *terminals*). The compression proceeds as follows:

1. It computes the most frequent pair of integers ab in I;
2. a new rule $s \rightarrow ab$ is created and added to a dictionary R. s is a new symbol not present in I;
3. every occurrence of ab in I is replaced by s; and
4. steps 1–3 are repeated until all pairs in I appear only once (see Fig. 1).

The resulting compressed sequence is called C. Observe C in Fig. 1, every symbol represents a phrase (a sequence of one or more of the integers in I), for example, a W represents the sequence 999999.

However, original symbols that were not compressed may remain in I, those symbols are called *terminal* symbols; otherwise, the symbol is defined in R and is called *non-terminal* symbol. Compressing a sequence with Re-Pair can be computed in linear time and a phrase may be recursively decompressed in optimal time (i.e., proportional to its length).

2.2.2. *DACs*

If we have a sequence of numbers, one way to compress them is to try to use the exact number of bits to represent the number ($\lceil \log_2(number) \rceil$ bits). However, by doing this, we lost the ability to access the ith number in the sequence. Several methods have been devised to be able to use variable-length representations of numbers that allow accessing to the ith number (Brisaboa et al., 2008; Klein and Shapira, 2016; Külekci, 2014) without the need to decompress the entire sequence.

This approach opens the way to keep the data always compressed and decompress exclusively the portions of the data needed for a certain purpose.

In this work, we use Directly Addressable Codes (DACs) (Brisaboa et al., 2013), which have been shown to be able to compress a sequence of integers with very fast direct access to the ith number.

Given a sequence of integers $X = x_1, x_2, \ldots, x_n$, DACs take the binary representation of those numbers and rearrange them into a level-shaped structure as follows: the first level, B_1, contains a sequence of n numbers. Each number of that level has n_1 bits, which are the n_1 least significant bits of the binary representation of each original symbol. A bitmap C_1 indicates for each position $1, 2, \ldots, n$ whether the binary representation of each integer requires more than n_1 bits (1) or not (0). In the second level, B_2 stores the next n_2 bits of the integers which have a 1 in B_1. A bitmap C_2 marks the positions that need more than $n_1 + n_2$ bits, and so on until all numbers in X have all their bits represented. The number of levels ℓ and the number n_l of bits at each level is computed in order to obtain the best compression.

Observe in the example of Fig. 2 that the 6th number of the original sequence is a 5. Below X, we can see the DAC representation using chunks of 2 bits in all levels. Therefore, the 6th chunk of 2 bits of B_1 is 01, the least significant 2 bits of the binary representation of 5. Obviously, we need an additional chunk to represent a 5, thus the 6th bit of C_1 is set to 1 to indicate this. In the second level, there are only two chunks of 2 bits, in order to know which of those two correspond to the original symbol at position 6, we count the

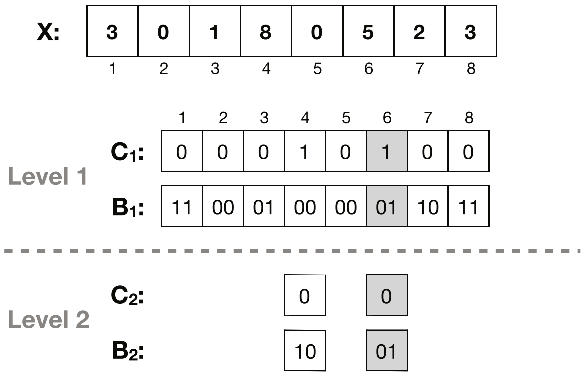


Fig. 2. Example Dac.

number of 1s bits until the 6th position of B_1, which is 2, then our number is the second chunk of B_2.

In compact data structures, counting the number of 1 bits (or 0 bits) until a given position is a very common operation called *rank*. More formally $rank_b(B, p)$ counts the number of occurrences of bit b in bitmap B up to position p. This operation can be solved in constant time (see (Munro, 1996), for example), using $n + o(n)$ bits of total space (in practice, approximately 5% extra space over the original bitmap).

3. A detailed description of DACTS

In this section, we present how we combine Re-Pair and DAC with additional data structures to make DACTS more suitable for industrial time series compression, and the algorithms to run queries on it.

Let us consider that we have a sequence of numbers N, if they are floating-point numbers, we multiply them to transform them into a sequence of integers I. Compression is achieved in DACTS by compressing I with Re-Pair.

Nevertheless, as shown in Fig. 1, in C we do not have a direct way to obtain the ith original symbol. Observe that, each symbol in C represents an undetermined number of symbols from the original sequence. This means that Re-Pair is not a compression method with direct access, and thus, for example, to access the integer at position 11, we have to decompress from the beginning.

The typical solution to this is to add a directory indicating, every d integers, the position of those integers in the compressed data. Notice that we are representing time series, thus the positions on the sequence I correspond with time instants. Therefore, for those time instants that are multiple of d, we can directly obtain their positions in the compressed data, and from that position on, we decompress normally until reaching the information of the desired time instant.

However, in Fig. 3, observe that position 12 of I is represented within the symbol at position 3 of C and, that position represents the integers of the original sequence from position 10 to position 15. Therefore a pointer to the symbol in C including a time instant i is not enough; instead, that pointer must have two parts: (1) the position in C of the symbol including x_i, and (2) an offset inside that symbol. Therefore the pointer to position 12 must be 3 : 2, that is, the symbol at position 3 of C, and within it, an offset of 2 from its beginning (see Fig. 3).

Let us suppose that we want to obtain the value at position 10. In the directory, we obtain the entry pointing to the closest time instant lesser or equal to 10, in our case, time instant 5: $\langle 0 : 5\rangle$. Typically, now we simply continue decompressing until reaching position 10. However, observe that we are now located at symbol W (position 0), and the next one is T, but we do not even know how many time instants T covers, so we lost the synchronism achieved by accessing a sampled position, and therefore, the only solution is to decompress all symbols from W onwards, until reaching time instant 10.

In our example, we have to decompress W, which requires 3 steps (rules $W \rightarrow VU$, $V \rightarrow UU$, and $U \rightarrow 99$), we obtain the sequence 999999, and we know that the time instant 5 is the sixth 9. Then, we process T, which requires 2 steps (rules $T \rightarrow S7$ and $S \rightarrow 88$) and we obtain the sequence 887, so we know that the 7 is at time instant 8. We continue reading the terminal 7, which corresponds to time instant 9. Hence, we proceed with the next symbol (W). Therefore, after decompressing W, we know that the first value of 999999 is our target.

This process would be very slow. To solve this, for each rule of R, we add the number of time instants that it covers, as shown in Fig. 3 under the title *span*.

Now, the process is much simpler. The pointer takes us to position 6 of the first W, but from R, we know that W spans 6 time instants, so that position is the last one of W, so we do not need to decompress it. Then, we read the T, and we check in R that T

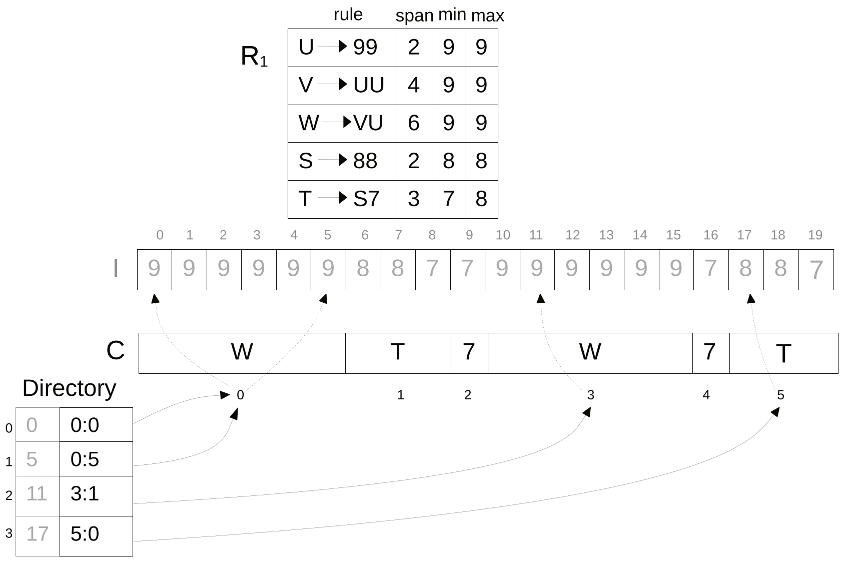


Fig. 3. Original sequence and its representation with repair.

spans 3 time instants, so it reaches time instant 8. Since it does not contain the desired time instant (10), we do not decompress it. The next symbol 7 is a terminal, so we know that it is at time instant 9. Finally, when we reach the second W, we know that we have to obtain the first symbol of W. Hence, we have to decompress W to obtain its original sequence 999999, and then we have obtained our target. Observe, that we have only decompressed the last symbol.

With this, we have seen the method to obtain the integer from a given position, but the typical query requires extracting a portion of several contiguous time instants. The obvious solution is that, once we have reached the first time instant of the query, we simply decompress symbols of C until reaching the last time instant of the queried period.

In industrial environments, as seen, typical queries also include summarizing or even analysis queries. In this work, we implemented two additional queries:

1. Given a time series $\mathcal{S}[1...n]$ and a time interval $[b, e]$ within $1...n$, return the minimum and maximum values within $[b, e]$.
2. Given a set of time series $\mathcal{S}_1, \ldots, \mathcal{S}_m$, of size n and a time interval $[b, e]$, where $b \leq e \leq n$, compute the Euclidean distance between $\mathcal{S}_1$ and $\mathcal{S}_2, \ldots, \mathcal{S}_m$ in the time interval $[b, e]$ and sort $\mathcal{S}_2, \ldots, \mathcal{S}_m$ according to that distance.

The naive solution to both queries is to decompress the time interval $[b, e]$ and proceed accordingly with the processed query.

To improve the speed, we add to each rule in R the maximum and minimum values of the symbols covered by the rule. Observe in Fig. 3 that the non-terminal T corresponds to the original sequence 887, so in R we set the minimum value to 7 and the maximum to 8.

Now, let us suppose that we want to obtain the maximum and minimum values in the range [6, 11] of the sequence I. By using the same process to access a position, we reach the symbol T that covers from time instant 6 until time instant 8. Instead of decompressing it, in the rule at R, we already have the maximum and minimum values in that stretch. Therefore, we know that the maximum value is 8 and the minimum is 7 without decompressing that symbol. Next, we process the terminal 7 corresponding to time instant 9, so the minimum and the maximum do not change. Then, we reach a W, which covers from time instant 10 to time instant 15. The maximum and minimum of symbol W is 9, so that means all values between 10 and 15 are 9, thus we can conclude that the maximum value is 9 and the minimum is 7 without decompressing any of the symbols.

Moreover, the use of those minimum and maximum values in the rules are not only useful for solving the min/max query. This can also help in the rest of the operations when the minimum and the maximum values are equal, we know that all the symbols covered by that rule are the same without decompressing it. For example, let us suppose that we want to recover the sequence between positions 2 and 5. We access the first symbol W, and from the information in R, we know that W covers 6 time instants, so it is enough to solve our query, and, in addition, since min = max = 9, all those 6 time instants are 9s, so we can solve the query simply outputting 9999. Observe that again, we can solve the query, without decompressing anything.

The *span*, *min*, and *max* fields of the entries of R are compressed with DACs.

Next, we show the algorithms of the three queries more formally.

3.1. Extract

Algorithm 1 shows the pseudocode for extracting the integers between time instant t_b and t_e. Line 1 obtains a reference to the symbol in C including the time instant t_b. For this, a call to the function *First* returns the tuple $\langle currS, ptrC, t_i, t_j \rangle$, where *currS* is that symbol, *ptrC* is a pointer to its position in C, and t_i and t_j are the time instants covered by that symbol.

The loop of Line 3 simply decompresses the symbols of C one by one from that obtained after the call to *First* until reaching t_e.

Algorithm 1. Extract(t_b, t_e)

```
Algorithm 1: Extract($t_b, t_e$)
1  $\langle currS, ptrC, t_i, t_j \rangle \leftarrow First(t_b)$;
2  $Result \leftarrow \emptyset$;
3  while $t_j \leq t_e$ do
4      $Result.add(Decompress(currS, t_b, t_e, t_i, t_j))$;
5      $ptrC \leftarrow ptrC + 1$;
6      $currS \leftarrow C[ptrC]$;
7      $t_i \leftarrow t_j + 1$;
8      $t_j \leftarrow t_j + R[currS].span$;
9  if $t_j > t_e$ then
10     $Result.add(Decompress(currS, t_b, t_e, t_i, t_j))$;
```

Algorithm 2 shows *First*. It starts by accessing the directory corresponding to the closest sampled position previous to t_b. t'_b is adjusted to the time instant indicated by the offset of the directory entry, and t'_e points to the last time instant covered by the currently processed symbol of C (*currS*). The loop of Line 6 simply follows C until reaching t_b, adjusting always t'_b and t'_e to the time instants covered by the current symbol (*currS*).

Algorithm 2. First(t_b)

```
Algorithm 2: First($t_b$)
1  $entry \leftarrow t_b + 1/d$;// $d$ is the sample rate of the
     directory, so entry is the entry in the directory
     corresponding to the previous position to $t_b$
2  $\langle ptrC, offset\rangle \leftarrow Directory(entry)$;
3  $currS \leftarrow C[ptrC]$;
4  $t'_b \leftarrow t_b - offset$;
5  $t'_e \leftarrow t'_b + R[currS].span - 1$;
6  while $t'_e < t_b$ do
7      $ptrC \leftarrow ptrC + 1$;
8      $currS \leftarrow C[ptrC]$;
9      $t'_b \leftarrow t'_e + 1$;
10     $t'_e \leftarrow t'_e + R[currS].span$;
11 return $\langle currS, ptrC, t'_b, t'_e\rangle$
```

Algorithm 3 shows the decompression of a symbol from *C*. Observe that, we do not check if the minimum and maximum values of a symbol are equal, in order to avoid decompressing the rule. The reason is that in practice, in this query, the costs of obtaining those values from the DACs storing them and the check outweigh the benefits.

Algorithm 3. Decompress(*currS*, t_b, t_e, t_i,t_j)

```
Algorithm 3: Decompress($currS$, $t_b$, $t_e$, $t_i$, $t_j$)
1  $Result \leftarrow \emptyset$;
2  if currS is non-terminal then
3      $leftS \leftarrow R[currS].leftS$;
4      $rightS \leftarrow R[currS].rightS$;
5      $t_m \leftarrow t_i + R[leftS].span$;
6      if $[t_i, t_m - 1] \cap [t_b, t_e] \neq \emptyset$ then
7          $Decompress(leftS,\ t_b,\ t_e,\ t_i,\ t_m - 1)$;
8      if $[t_m, t_j] \cap [t_b, t_e] \neq \emptyset$ then
9          $Decompress(rightS,\ t_b,\ t_e,\ t_m,\ t_j)$;
10 else
11     if $t_i \in [t_b, t_e]$ then
12         $Result.add(currS)$;
13 return $Result$
```

3.2. Minimum and maximum

The computation of the minimum and maximum is basically the same process as the extract query. Algorithm 4 shows the pseudocode. With respect to Algorithm 1, the only difference is that instead of using the function *Decompress*, it uses a different function called *MinMax*.

Algorithm 4. **Extremes**(t_b, t_e)

```
Algorithm 4: Extremes($t_b, t_e$)
$min \leftarrow \infty$;
$max \leftarrow -\infty$;
$\langle currS, ptrC, t_i, t_j \rangle \leftarrow First(t_b)$;
while $t_j \leq t_e$ do
    $\langle minS, maxS \rangle \leftarrow (MinMax(currS,\ t_b,\ t_e,\ t_i,\ t_j))$;
    if $minS < min$ then $min \leftarrow minS$;
    if $maxS > max$ then $max \leftarrow maxS$;
    $ptrC \leftarrow ptrC + 1$;
    $currS \leftarrow C[ptrC]$;
    $t_i \leftarrow t_j + 1$;
    $t_j \leftarrow t_j + R[currS].span$;
if $t_j > t_e$ then
    $\langle minS, maxS \rangle \leftarrow (MinMax(currS,\ t_b,\ t_e,\ t_i,\ t_j))$;
    if $minS < min$ then $min \leftarrow minS$;
    if $maxS > max$ then $max \leftarrow maxS$;
```

In turn, *MinMax* (shown in Algorithm 5) is basically the same as *Decompression*, but now, it takes advantage of having stretches with the same min/max value. This can be seen in Lines 2–9, where if the minimum and maximum values stored in *R* for the current symbol are equal, those values are used and then the decompression is not needed.

Algorithm 5. **MinMax**(*currS*, t_b, t_e, t_i,t_j)

```
Algorithm 5: MinMax($currS,\ t_b, t_e,\ t_i, t_j$)
1  if $currS$ is non-terminal then
2      if $R[currS].min = R[currS].max$ then // All symbols are the same
3          $t'_b \leftarrow \max(t_b, t_i)$;
4          $t'_e \leftarrow \min(t_e, t_j)$;
5          $min \leftarrow R[currS].min$;
6          $max \leftarrow R[currS].max$;
7      else if $t_i \geq t_b$ and $t_j \leq t_e$ then // The whole rule is within $[t_b, t_e]$
8          $min \leftarrow R[currS].min$;
9          $max \leftarrow R[currS].max$;
10     else
11         $leftS \leftarrow R[currS].leftS$;
12         $rightS \leftarrow R[currS].rightS$;
13         $t_m \leftarrow t_i + R[leftS].span$;
14         if $[t_i, t_m - 1] \cap [t_b, t_e] \neq \emptyset$ then
15             $MinMax(leftS,\ t_b,\ t_e,\ t_i,\ t_m - 1)$;
16         if $[t_m, t_j] \cap [t_b, t_e] \neq \emptyset$ then
17             $MinMax(rightS,\ t_b,\ t_e,\ t_m,\ t_j)$;
18 else
19     if $t_i \in [t_b, t_e]$ then
20         $min \leftarrow currS$;
21         $max \leftarrow currS$;
22 return $\langle min, max \rangle$
```

Table 1
Trace of the stacks.

	$Stack_1$	$Stack_2$
Step 1	<T,1,6,8>	<X,3,6,7>
Step 2	<S,1,6,7> <7,1,8,8>	
Step 3	<7,1,8,8>	<A,4,8,10>

3.3. *Similarity*

Algorithm 7 shows the function *Similarity* that given two time series ($\mathcal{S}_1$ and $\mathcal{S}_2$) and a time interval $[t_b, t_e]$, it computes the Euclidean distance of the time series in that time interval.

By using the function *First*, it obtains the symbol in *C* of each time series covering t_b. Then, it adds each of those symbols to its corresponding stack: $Stack_1$ and $Stack_2$ are initialized with the symbol from $\mathcal{S}_1$ and $\mathcal{S}_2$, respectively. The processed symbol at any given step is obtained from those stacks.

In t_1 and t_2, the algorithm keeps account of the last time instant that has already been processed in each time series.

In order to take advantage of Re-Pair and the minimum and maximum values stored at the rules, the algorithm uses the concept of *run* to denote a time interval of a time series that has the same value. In Lines 6 and 7, the algorithm uses the function *NextRun* to obtain the next *run* from the currently processed symbol.

Lines 10 and 11 compute the parts of the current runs of both time series that overlap, that is, the time interval where both time series have the same value, and then, in Lines 12 and 13, the Euclidean distance is computed for that stretch. This process is repeated until reaching the last time instant of the queried time interval.

Algorithm 6. **NextRun**(*Stack*, t_b, t_e)

Algorithm 6: NextRun$(Stack,\ t_b, t_e)$

```
1  do
2      $\langle currS, ptrC, t_i, t_j\rangle \leftarrow Stack.pop()$;
3      if $R[currS].min = R[currS].max$ then // All symbols are the same
4          $sol \leftarrow \langle R[currS].min, t_i, t_j\rangle$;
5          break;
6      else
7          $\langle leftS, rightS\rangle \leftarrow decompress[currS]$;
8          $t_m \leftarrow t_i + R[leftS].span$;
9          if $[t_m, t_j] \cap [t_b, t_e] \neq \emptyset$ then
10             $Stack.add(\langle rightS, ptrC, t_m, t_j\rangle)$
11         if $[t_i, t_m - 1] \cap [t_b, t_e] \neq \emptyset$ then
12             $Stack.add(\langle leftS, ptrC, t_i, t_m - 1\rangle)$
13 while $Stack$ is not empty;
14 if $Stack$ is empty then // $currS$ was completely processed
15     $ptrC = ptrC + 1$;
16     $currS \leftarrow C[ptrC]$;
17     $t_i = t_j + 1$; $t_j = t_j + R[currS].span$
18     if $[t_i, t_j] \cap [t_b, t_e] \neq \emptyset$ then
19         $Stack.add(\langle currS, ptrC, t_i, t_j\rangle)$
20 return $sol$
```

Algorithm 7. **Similarity**$(\mathcal{S}_1, \mathcal{S}_2, t_b, t_e)$

```
Algorithm 7: Similarity($\mathcal{S}_1$, $\mathcal{S}_2$, $t_b$, $t_e$)
1  $e_1 \leftarrow \mathcal{S}_1.First(t_b)$;// $e_1$ and $e_2$ are two tuples that
     contain $\langle currS, ptrC, t_i, t_j \rangle$
2  $e_2 \leftarrow \mathcal{S}_2.First(t_b)$;
3  $Stack_1.add(e_1)$; $Stack_2.add(e_2)$;
4  $t_1 \leftarrow 0$; $t_2 \leftarrow 0$; $sim \leftarrow 0$;
5  while $Stack_1$ is not empty or $Stack_2$ is not empty do
       // Each run is a triplet $\langle val, t_i, t_j \rangle$
6      if $t_1 \leq t_2$ then
7          $run_1 \leftarrow \mathcal{S}_1.NextRun(Stack_1, t_b, t_e)$
8      if $t_2 \leq t_1$ then
9          $run_2 \leftarrow \mathcal{S}_2.NextRun(Stack_2, t_b, t_e)$
10     $i \leftarrow \max(run_1.t_i, run_2.t_i)$;
11     $j \leftarrow \min(run_1.t_j, run_2.t_j)$;
12     $\Delta \leftarrow |run_1.val - run_2.val|$;
13     $sim \leftarrow sim + \Delta \times (j - i + 1)$;
14     $t_1 \leftarrow run_1.t_j$; $t_2 \leftarrow run_2.t_j$;
15 return $sim$
```

Let us illustrate the algorithm with the two time series in Fig. 4, and assuming that we want to compute the euclidean distance between time instants 6 and 8.

Lines 1 and 2 of Algorithm 7 obtain a pointer to the first symbol of each time series containing the time instant 6. Line 3 adds them to the stacks. The result can be seen in Table 1 in *Step 1*. The symbol of C_1 containing the time instant 6 is the symbol T at position 1 and, in the case of C_2, it is the symbol X at position 3. Observe that the tuples pointing to those symbols also contain two additional values, t_i and t_j, that indicate the time interval corresponding to those symbols, that is, T spans between time instants 6 and 8, and X spans between time instants 6 and 7.

Line 7 triggers the call $\mathcal{S}_1.nextRun(Stack_1, 6, 8)$. In Table 3, we can see its trace under the label $(1)\mathcal{S}_1.nextRun(Stack_1, 6, 8)$. Line 2 pops the top of $Stack_1$. $CurrS$ is T, so the *if* of Line 3 is false and then the flow jumps to Line 7, where T is decompressed obtaining the pair $\langle S, 7\rangle$ (see columns *LeftS* and *RightS* in *Step 1* of $(1)\mathcal{S}_1.nextRun(Stack_1, 6, 8)$ in Table 3). t_m is the time instant where the right symbol starts, in our case, 7 starts at time instant 8. Since the time interval $[t_m, t_j] = [8, 8]$ intersects our query time interval ([6, 8]), the symbol 7 is added to $Stack_1$, as seen under *Step 2* of Table 1. Next, the *RightS* symbol (S) is treated, it covers from time instant $t_i = 6$ until time instant $t_m - 1 = 7$, which intersects [6, 8], so it is also added to $Stack_1$ (see *Step 2* of Table 1). Next, a new iteration of the *do-while* loop extracts the top of $Stack_1$, which is a S that covers from time instant $t_i = 6$ until time instant $t_j = 7$. Since, $R_1[S].\min = R_1[S].\max = 8$, then Line 4 sets *sol* to $\langle 8, 6, 7\rangle$ (see *Step 2* of $(1)\mathcal{S}_1.nextRun(Stack_1, 6, 8)$ in Table 3). This ends $(1)\mathcal{S}_1.nextRun(Stack_1, 6, 8)$.

Next, in Line 9, the algorithm issues $\mathcal{S}_2.nextRun(Stack_2, 6, 8)$. We can see in Table 3 under the label $(1)\mathcal{S}_2.nextRun(Stack_2, 6, 8)$, that it processes the top of $Stack_2$, which includes the symbol X of C_2. Observe in R_2, that X has the same maximum and minimum value (9), so the *if* of Line 3 of Algorithm 6 is true and thus, in the variable *sol*, it is stored a 9 with the time instants ([6, 7]) covered by that symbol (see *Step 1* of $(1)\mathcal{S}_2.nextRun(Stack_2, 6, 8)$ in Table 3), and the *do-while* loop is broken.

Since after removing the top of $Stack_2$ the stack was empty, the flow reaches Line 15 of Algorithm 6, where the next symbols of C_2 is read, that is, $ptrC = 4$ and $currS = A$ adjusting $t_i = 8$ and $t_j = 10$. Since that interval intersects [6, 8], the tuple $\langle A, 4, 8, 10\rangle$ is added to $Stack_2$, as seen in Step 3 of Table 1.

Therefore, after the two calls to the *nextRun* function, run_1 and run_2 have the values shown in Step 1 of Table 2. Both are runs of two elements, in fact, in this example, they cover exactly the same time instants. Lines 10 and 11 of Algorithm 7adjust the overlapping time instants of both runs, which in our example are obviously $[i, j] = [6, 7]$, and then the distance of that interval is stored in *sim*. Observe that we are adding the values of a period of time (in this example of 2 time instants) in just one step. This speeds up the computation.

Next, Line 14 of Algorithm 7 sets $t_1 = 7$ and $t_2 = 7$ the last time instant processed so far in the time series, and the loop returns to Line 6. Line 7 issues again $\mathcal{S}_1.nextRun(Stack_1, 6, 8)$. That call is shown in Table 3 as $(2)\mathcal{S}_1.nextRun(Stack_1, 6, 8)$. It extracts the top of the $Stack_1$, which is $\langle 7, 1, 8, 8\rangle$. Given 7 is a terminal, its minimum and maximum values are the same, and thus the variable *sol* is set to $\langle 7, 8, 8\rangle$ (see *Step 1* of $(2)\mathcal{S}_1.nextRun(Stack_1, 6, 8)$ in Table 3). After removing $\langle 7, 1, 8, 8\rangle$ from $Stack_1$, it is empty, so the *if* of Line 14 takes the flow to line 15, where the pointer to C_1 is moved to position 9, and then *CurrS* is set to 7. However, that symbol corresponds to a time instant outside of our queried time interval [6, 8], and then nothing is added to the stack.

The flow returns to the similarity function, to Line 8, and thus a second call $(2)\mathcal{S}_2.nextRun(Stack_2, 6, 8)$ is issued. This call pops the tuple $< A, 4, 8, 10 >$ from $Stack_2$. Since $R_2[A].\min = R_2[A].\max = 8$, *sol* is set to $< 8, 8, 10 >$ and the *do-while* loop is broken. Since $Stack_2$ is empty, Lines 15 and 16 of *nextRun* move the pointer to C_2 to position 5, but that position surpasses the time instant 8, and thus nothing is added.

Returning to the similarity function, we have the state shown in the *Step 2* of Table 2. $\Delta = 1$, $i = 8$ and $j = 8$, so a 1 is added to *sim*. Since both stacks are empty, then the process ends, and the obtained result is $sim = 3$.

Table 2
Trace of similarity function.

	t_b	t_e	t_1	t_2	*sim*	run_1	run_2	Δ	*i*	*j*
Step 1	6	8	0	0	0	<8,6,7>	<9,6,7>	1	6	7
Step 2	6	8	7	7	2	<7,8,8>	<8,8,10>	1	8	8

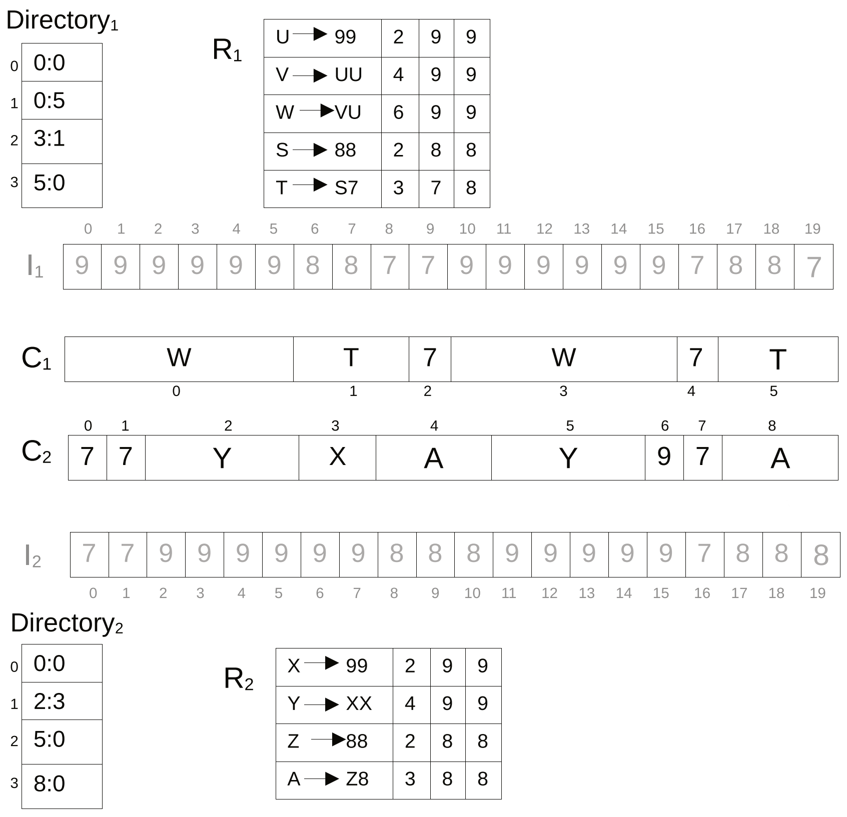


Fig. 4. Two time series.

Table 3
Trace of NextRun function.

Call		*CurrS*	*ptrC*	t_i	t_j	*LeftS*	*RightS*	t_m	*sol*
$(1)\mathcal{S}_1.nextRun(Stack_1, 6, 8)$	Step 1	T	1	6	8	S	7	8	
	Step 2	S	1	6	7				<8,6,7>
$(1)\mathcal{S}_2.nextRun(Stack_2, 6, 8)$	Step 1	X	3	6	7	9	9		<9,6,7>
	Step 2	A	4	8	10				
$(2)\mathcal{S}_1.nextRun(Stack_1, 6, 8)$	Step 1	7	1	8	8				<7,8,8>
	Step 2	7	2	9	9				
$(2)\mathcal{S}_2.nextRun(Stack_2, 6, 8)$	Step 1	A	4	8	10				<8,8,10>
	Step 2	Y	5	11	14				

4. Experimental evaluation

For our experimental evaluation, we implemented DACTS in C++, using components from the SDSL library[1] (Gog et al., 2014). Our implementation also uses a balanced version of Re-Pair by G. Navarro[2] to build the grammar, and represents the extra information on non-terminals using DACs with an unlimited number of levels and without a predefined chunk size.

As baselines, we also included in our experiments three well-known general purpose compressors. Gnu `gzip`,[3] a Ziv-Lempel-based compressor, the powerful `p7zip`[4] compressor, which is an LZMA compressor with a dictionary of up to 4 Gigabytes, and `snappy`,[5] which is used, among others, by Influx, MongoDB, or Cassandra.

For running queries on these baselines, the compressed time series is completely decompressed before running a C program over the uncompressed data.

Finally, we also included a compression method with random access, the plain `dac`, using also the implementation of SDSL library.

In all experiments, the data are initially stored on disk. We measured elapsed (or clock) time. The times of extract and minimum/maximum queries are resulting from running 500 queries with random time intervals. In the case of similarity queries, we run 100 different random intervals.

4.1. Datasets

The datasets are from two different origins. First, real data from extrusion machines of a factory that wants to keep its name confidential, these data were used since this company is part of the R&D project that finances this work. The datasets gather the data of three different sensors during 100 days and in 7 different machines. Depending on the sensor, the data distribution is different.

These are the datasets used in extract and minimum and maximum queries (see the details of these datasets in Table 4):

[1] https://github.com/simongog/sdsl-lite
[2] http://www.dcc.uchile.cl/gnavarro/software/repair.tgz
[3] http://www.gzip.org/
[4] http://p7zip.sourceforge.net/
[5] http://google.github.io/snappy/

Table 4
Details of the datasets.

Dataset	Description	Machine type	Sensor type	Avg Interval Extract MinMax	Avg Interval Similarity	#Points	Samp Freq
M1-7ZE6BE	Accumulated machine stop time in the last 24 h	Extruder Machine	Counter	1949259	1883	7675823	1 Hz
M2-PS4EK1	Temperature of an extruder machine's tilting servo	Extruder Machine	Thermal	1949259	1653	7553234	1 Hz
M2-VMTKD6	Melting temperature of an extruder machine	Extruder Machine	Thermal	1949259	9474	7553234	1 Hz
7ZE6BE	Accumulated machine stop time in the last 24 h	Extruder Machine	Counter	11293184	1301750	52454444	1 Hz
PS4EK1	Temperature of an extruder machine's tilting servo	Extruder Machine	Thermal	11293184	1614791	44249009	1 Hz
VMTKD6	Melting temperature of an extruder machine	Extruder Machine	Thermal	11293184	1733745	52454444	1 Hz
all	Mixture	Extruder Machine	Mixture	38804747	n/a	149157897	1 Hz
PHM-10	Ion current impacting the beam grid determining the amount of ions accelerated through the grid assembly to the wafer	Ion mill etch tools	Counter	19059987	472380	82189440	1 Hz
PHM-18	Wafer rotation speed setting	Ion mill etch tools	Accelerometer	19059987	472380	82189440	1 Hz

- `M1-7ZE6BE` joins 100 days of sensor `7ZE6BE` in machine `I_JKH_JJAHTT`.
- `M2-PS4EK1` joins 100 days of sensor `PS4EK1` in machine `I_DXR_RSQESL`.
- `M2-VMTKD6` joins 100 days of sensor `VMTKD6` in machine `I_DXR_RSQESL`.

- `7ZE6BE` joins the 100 days of sensor `7ZE6BE` in 7 different machines.
- `PS4EK1` joins the 100 days of sensor `PS4EK1` in 7 different machines.
- `VMTKD6` joins the 100 days of sensor `VMTKD6` in 7 different machines.

- `all` joins `7ZE6BE`, `PS4EK1`, and `VMTKD6`.

To provide values on a public domain dataset, we used the dataset *PHM DATA Challenge 18: Etching tool fault detection (PdM)*.[6] This dataset contains 24 time series coming from different sensors. The time series are divided into 20 files, given their size.

To select the time series used in our experiments, we run gzip, p7zip, snappy and DACTS on 22 columns (the other 2 were the timestamp and the identifier of the tool) of just one file (01_M01_DC_train). Each column corresponds to the time series of one sensor.

As seen in Table 5, the best compression is achieved by `p7zip`, followed by `gzip`, since they are typical compressors that aim at obtaining good compression power. DACTS is close to `gzip` in some columns, like 6, 7, 17, 18, 19, and 20. In others, DACTS is still clearly better than `snappy`, whereas there are a set of columns where they are on a par (4, 5, 9, 10, 11, 12, and 14). However, there are three columns were DACTS perform worse than all the rest, columns 21, 22, and 23. DACTS requires datasets where there are repetitions of the same sequences of numbers. Therefore, it is likely that in small files this may be harder to find, as in this experiment, where the original data were only 12 Mbytes. In the rest of experiments, we join the 20 files, so the data of one sensor are 314 Mbytes. In those larger files, DACTS clearly improves the compression achieved by `snappy` in columns 21, 22, and 23, where DACTS achieves a 8.22% 7.03%, and 8.68%, whereas `snappy` obtains a 9.75%, 7.99%, and 9.25%, respectively.

Table 5
Compression ratio of 22 columns of file 01_M01_DC_train.

Column	gzip	p7zip	DACTS	Snappy
3	0.35%	0.30%	0.74%	4.97%
4	0.40%	0.35%	5.78%	5.15%
5	0.21%	0.13%	4.58%	4.86%
6	0.36%	0.26%	0.54%	4.90%
7	0.94%	0.83%	1.26%	5.94%
8	12.00%	7.20%	18.94%	21.48%
9	13.46%	9.22%	25.79%	28.75%
10	30.47%	19.91%	52.97%	51.10%
11	16.18%	10.86%	29.34%	32.96%
12	24.77%	16.47%	49.41%	40.21%
13	7.63%	5.29%	11.72%	23.04%
14	11.54%	7.74%	22.80%	23.26%
15	7.16%	5.13%	11.49%	21.96%
16	8.66%	6.14%	13.79%	23.14%
17	0.69%	0.59%	0.94%	5.24%
18	0.14%	0.06%	0.17%	4.73%
19	0.30%	0.22%	0.59%	4.85%
20	0.68%	0.64%	0.97%	5.57%
21	1.58%	1.00%	26.50%	8.74%
22	1.47%	0.89%	29.95%	8.59%
23	2.51%	1.42%	28.05%	10.46%
24	0.47%	0.38%	1.21%	5.06%

For the rest of experiments, we chose columns 10 and 18. Column 10 represents those sensors where DACTS has more problems, since in this column, DACTS obtained the worst results. Column 18 is a representative of the sensors where DACTS nearly meets or surpasses the compression performance of `gzip`.

For the similarity queries, we used three sets of datasets:

- *Extrusion daily datasets*: formed by the datasets of one day of one sensor of one machine.
- *Extrusion 100 days datasets*: formed by the datasets joining 100 days of one sensor of one machine.
- *PHM datasets*: the 20 original *train* files.

We preprocessed all datasets to transform them into integers. If the original numbers were floating-point numbers, in the extrusion machines, we took the first two decimal digits and multiply them by 100 to obtain integer numbers; in the case of the PHM, since the

[6] https://github.com/makinarocks/awesome-industrial-machine-datasets

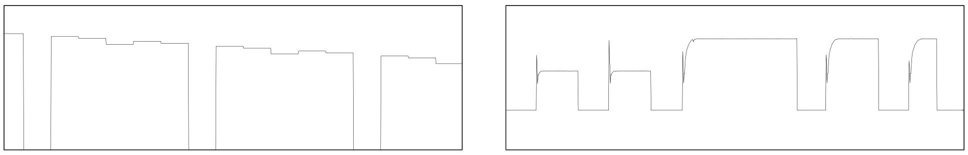
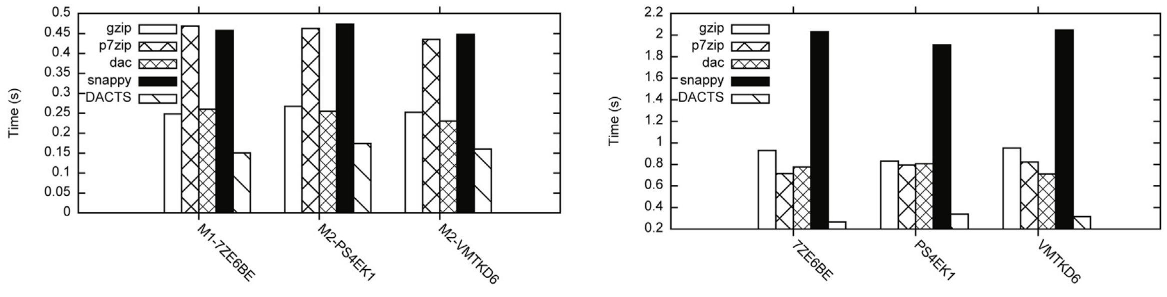
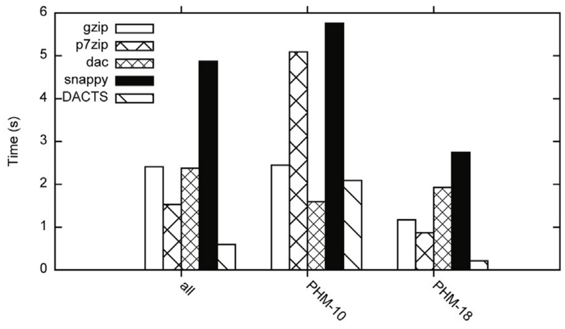

(a) M1-7ZE6BE (b) PHM-10

Fig. 5. Two portions of the time series M1-7ZE6BE and PHM-10.



(a) Small datasets (b) Medium datasets



(c) Large datasets

Fig. 6. Extract time (sec.).

numbers were smaller, we took the four first decimal digits and multiply them by 1000. Fig. 5 shows the shape of two portions of the datasets.

Column *Avg Interval Extract MinMax* of Table 4 gives the average size of the query intervals used in the queries *Extract* and *Minimum/Maximum* and column *Avg Interval Similarity* gives the average size of the query intervals used in the *Similarity* queries.

4.2. Compression

Table 6 shows the original size (in Megabytes) of the datasets and the compression ratio[7] achieved by the compression methods of this study.

In the case of the extrusion machines, in the shorter files, DACTS behaves worse, the differences with `gzip` are between 0.38 and 3.73 percentage points and with `p7zip` between 0.5 and 4.03 percentage points. As the datasets increase in size, repetitions increase, and then Re-Pair performs better. In midsize datasets, DACTS performs better than `gzip` in `7ZE6BE`, and the differences in the other two datasets are 0.68 and 2.4 percentage points, in the case of `p7zip`, differences are between 0.1 and 2.72 percentage points. In the largest dataset, differences are small, 0.90 percentage points with `gzip` and 1.09 percentage points with `p7zip`.

[7] The size of the compressed file as a percentage of the original file.

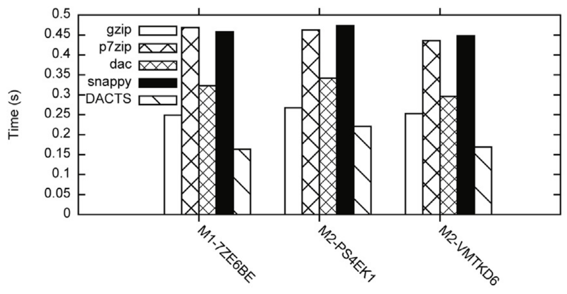
Table 6
Compression ratios.

	Size (Mb)	gzip	p7zip	dac	DACTS	Snappy
M1-7ZE6BE	29.28	0.29%	0.17%	30.73%	0.67%	4.99%
M2-PS4EK1	28.81	2.20%	1.90%	44.53%	5.93%	9.16%
M2-VMTKD6	28.81	0.78%	0.63%	28.91%	1.55%	5.87%
7ZE6BE	200.10	0.27%	0.15%	30.07%	0.25%	4.96%
PS4EK1	168.80	2.19%	1.87%	44.53%	4.59%	9.17%
VMTKD6	200.10	0.71%	0.55%	25.39%	1.23%	5.87%
all	568.99	0.99%	0.80%	35.73%	1.89%	6.53%
PHM-10	313.50	35.29%	23.06%	60.16%	58.63%	55.85%
PHM-18	313.50	0.12%	0.03%	78.91%	0.10%	4.72%

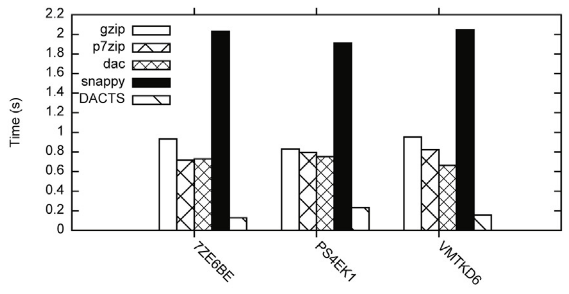
Mention apart is `dac`, with a typical compression of "symbol by symbol", that is, for each original symbol, in the compressed file there is also one (shorter) symbol, it is not able to compress to the same level as the rest. The main feature of `dac` is its capability of decompressing from any given point.

In the case of the PHM datasets, as explained, in column 10, the results of DACTS are poor. Re-Pair requires datasets with long sections that are equal to others. This is also the basis of `gzip` and `p7zip`, but the direct access of DACTS has a price, DACTS considers the original sequence of data as 32-bit integers, whereas `gzip` and `p7zip` considers the original data as a sequence of bytes. Therefore, `gzip` and `p7zip` find more repetitive sections.

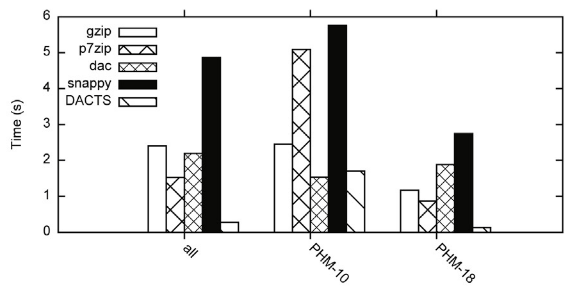
However, as seen in the results of column 18, in the industrial environment, it is likely that the repetitiveness of the data may yield almost the performance of `gzip`, indeed, in this column, DACTS



(a) Small datasets

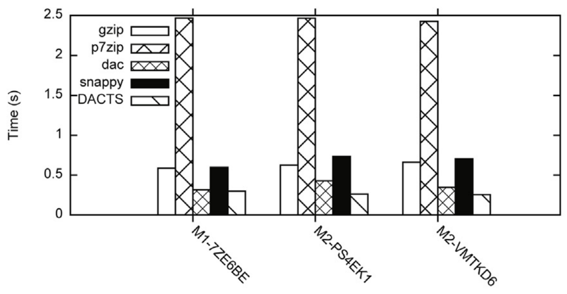


(b) Medium datasets

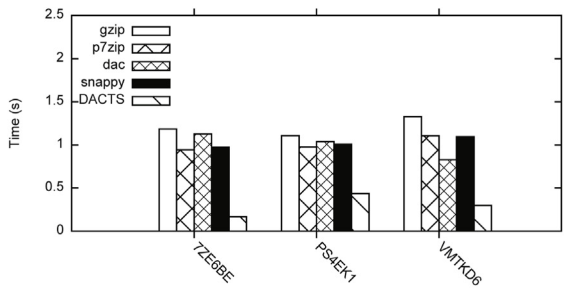


(c) Large datasets

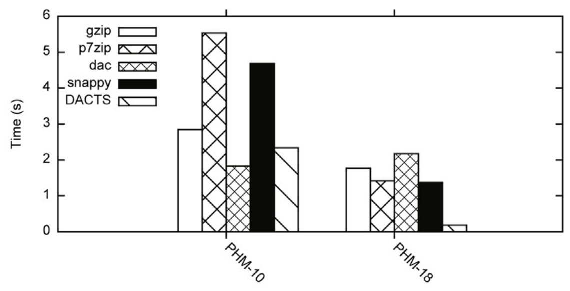
Fig. 7. Time to obtain min/max (sec.).



(a) Daily datasets



(b) 100 days datasets



(c) PHM datasets

Fig. 8. Time to sort by Euclidean distance (sec.).

obtains better compression than `gzip`. In any case, in our experiments, in the worst case, DACTS obtains compression ratios around those of `snappy`, a widely used compressor.

This can also be seen in the improvements in small vs medium and big datasets. In small files, the chances of finding repeating sections are lower. This can be seen by comparing the small and medium files of the same sensor, for example, in the dataset `M2-PS4EK1` the compression ratio of DACTS is 5.93% with a gap of 4.03 percentage points with `p7zip`. However, in the dataset `PS4EK1`, DACTS improves to 4.59% with a gap of 2.72 percentage points with `p7zip`.

4.3. Extract

Fig. 6 shows the extract time. Here it is where we can see the benefits of having the ability of starting the decompression from intermediate points of the compressed file. DACTS is between 1.17 and 5.47 times faster than *gzip*, between 2.33 and 4.07 times faster than *p7zip*, and between 2,72 and 12.86 times faster than *snappy*. Obviously, the reason is that in classic compressors, a complete decompression of the dataset is required before performing the extract operation.

As in the rest of the experiments, `PHM-10` is the worst scenario for DACTS, observe, for example, that DACTS is only 1.17 times

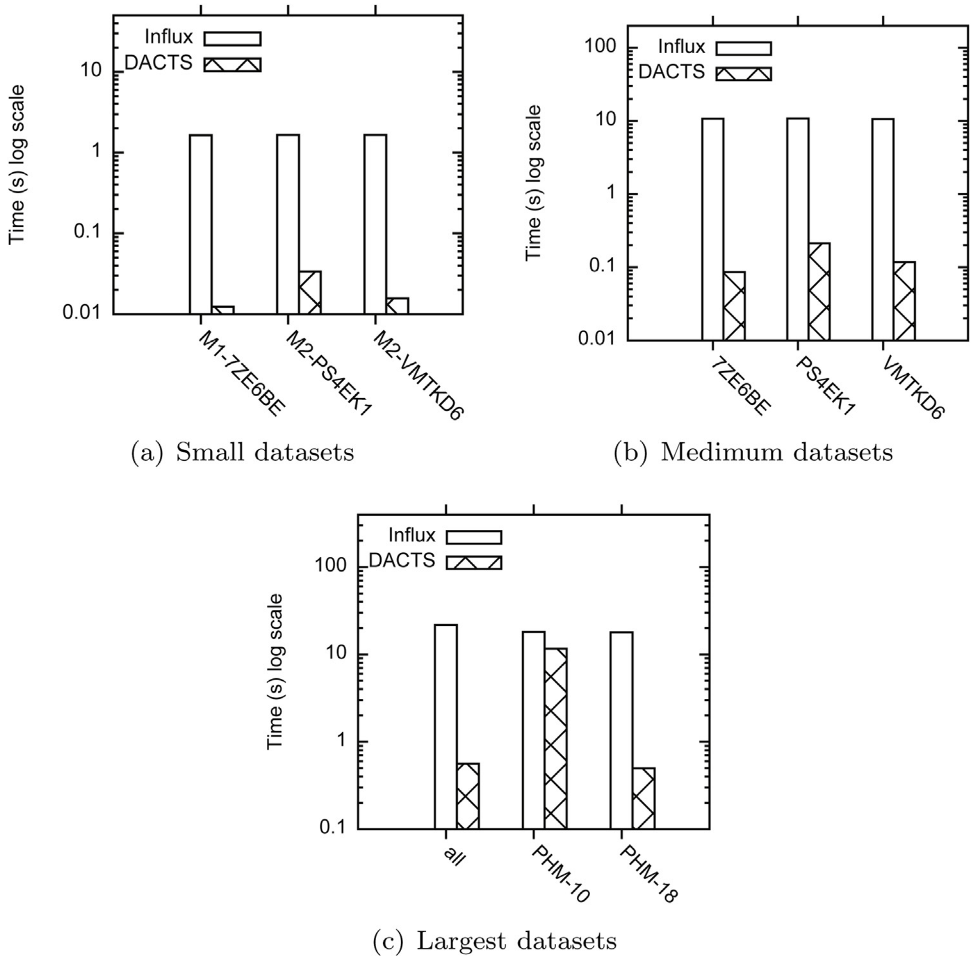


Fig. 9. Extract time (seconds in logarithmic scale) in the InfluxDB experiment.

faster than `gzip`. The reasons are that the input file is larger (only 58.63% of compression) and the non-terminal symbols (repetitive sections) are smaller. The additional information of the rules representing the non-terminal symbols is one of the key factors to speed up the queries on compressed data, and thus, the smaller the sections covered with non-terminals, the lower the acceleration.

Comparing with a native direct access method (`dac`), we still have better results: between 1.46 and 9.02 times faster, if we exclude the `PHM-10` dataset, where `dac` is 1.30 times faster. Although `dac` only has to decompress the exact portion of the dataset needed to solve the query, the problem is that its compression ratio is really poor, and thus it has to process (including reading) much more data. This is especially noticeable in `PHM-18` and `all`, where its performance is close to the naive general purpose compressors.

4.4. Minimum and maximum

Fig. 7 shows the time to obtain the minimum and maximum. The improvements in this query are even better since, as explained, DACTS can obtain from R the minimum and maximum values of the non-terminals used in the compressed sequence C, thus without needing to decompress them.

DACTS is between 1.21 and 8.83 times faster than `gzip`, between 2.09 and 6.56 faster than `p7zip`, and between 2.14 and 20.75 times faster than `snappy`. Except in `PHM-10`, where `dac` is 1.11 times faster, `dac` shows mediocre results, it is between 2.14 and 20.75 times slower than DACTS.

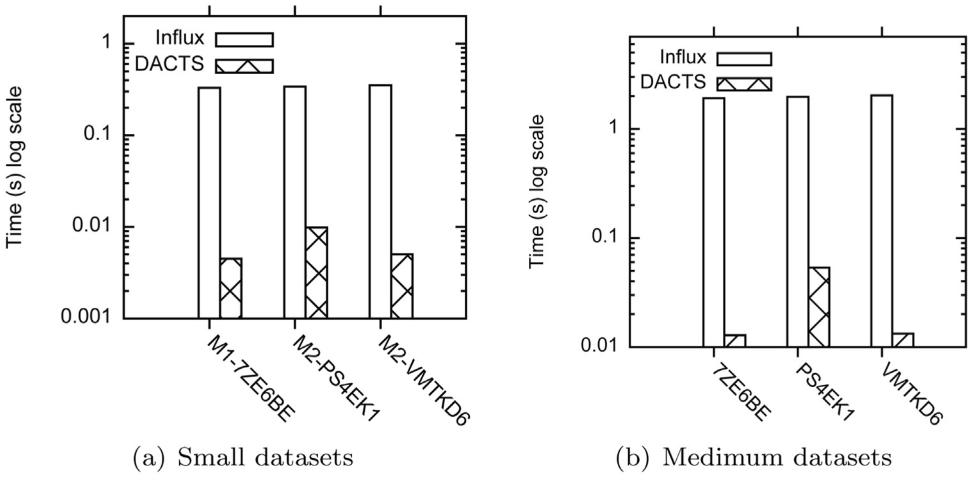
4.5. Similarity

In the case of the daily datasets, the experiments take the time series of the first day and compute the Euclidean distance in a random time interval between that time series and the remaining 99 days of the same sensor and machine, and then, it sorts the 99 days according to that distance. In the case of the 100 days datasets, the experiment takes the 100 days of one sensor and machine, and computes the Euclidean distance with respect to the 100 days of the same sensor in the other 6 machines. Again, once this was computed, the machines are sorted according to the computed distance. Finally, in the case of the PHM, the experiment takes one of the 20 original files and computes the Euclidean distance with the 19 remaining files.

Fig. 8 shows the results. In the daily datasets, `gzip` is between 1.95 and 2.58 times slower and `snappy` between 1.98 and 2.79 times slower. `p7zip` is penalized by its slower decompression speed, especially in this experiment, where it has to decompress 100 relatively small files, and thus it is between 8.2 and 9.44 times slower. Finally, `dac` is almost on a par in the `7ZE6BE` and, 1.35 and 1.63 times slower in the other two.

In the 100 days datasets, `p7zip` improves with longer files, whereas the rest worsens, especially `dac`. `gzip` is between 2.53 and 7.14 times slower, `snappy` between 2.31 and 5.87 times slower, `dac` between 2.38 and 6.79 times slower, and `p7zip` is between 2.24 and 5.69 times slower.

Finally, in the PHM datasets, again `PHM-10` shows the difficulties of DACTS with this dataset, it is only 1.21 times faster than



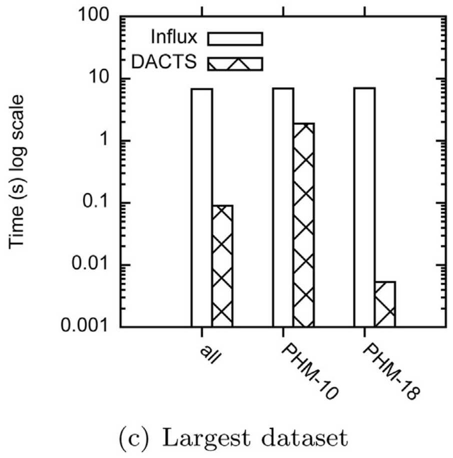


Fig. 10. Time to obtain min/max (seconds in logarithmic scale).

gzip, 2 times than snappy, 2.36 than p7zip, while dac is 1.27 times faster. However, the case of PHM-18 shows the other side of the coin; DACTS is 9.5 times faster than gzip, 7.35 times than snappy, 7.62 than p7zip, and 11.67 times faster than dac.

The reasons for the improvements in this query are, in part, also related to the additional information in the rules, which also helps in this query.

4.6. A brief comparison between DACTS and InfluxDb

In this experiment, we compare DACTS with a real Time Series Data Base (TSDB) management system, InfluxBD, to show the differences in the behaviour of both systems considering the same scenario. For the experiment, the datasets considered in Section 4.1 have been formatted to the *line protocol*[8] required by InfluxDb to represent each point of data (i.e., each measurement), and then, they have been inserted into an InfluxDb instance.

We are aware that both systems are different in nature and that therefore, their comparison may not be fair. However, we believe that it is interesting to compare our novel proposal against such a widespread system in the area of time series management. InfluxDB is a real TSDB management system that, to answer a query, needs to access data on the disk, and also search the data by timestamp, while in DACTS data is accessed directly by position in memory. This comparison has been included just to give a general idea of the differences between InfluxDB and DACTS.

Moreover, it is worth also mentioning that in DACTS we have omitted the timestamps of the time series (since we are working with periodic time series sampled at 1Hz (i.e., a measurement per second) and we already store the initial timestamp) and the standard data representation in InfluxDb (i.e., the line protocol) requires the timestamp of each data point for inserting the data into the database.

Fig. 9 shows the time required by the extract operation. The improvement of DACTS in time is between 49 and 132 times faster.

Fig. 10 shows the results for the computation of the minimum and maximum values. Here the improvements of DACTS range from 35 times to 715 times faster.

Finally, in the similarity experiment shown in Fig. 11, DACTS is between 3.7 and 1306 times faster.

The experiments were conducted on an Intel® Core^{TM} i7-3820 CPU @ 3.60GHz (4 cores) with 10MB of cache and 64 GB of RAM, running Debian GNU/Linux 9 with kernel 4.9.0-8 (64 bits), gcc version 6.3.0 with -O9 optimization and on a Google Compute Engine instance for the InfluxDb evaluation. The used machine for this instance has been a *n1-highmem-4*[9] (*4 vCPUs, 26 GB RAM*) with a standard persistent disk.

[8] InfluxDb Line Protocol Syntax: https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_tutorial/#syntax.

[9] Machine types in Google Compute Engine: https://cloud.google.com/compute/docs/machine-types.

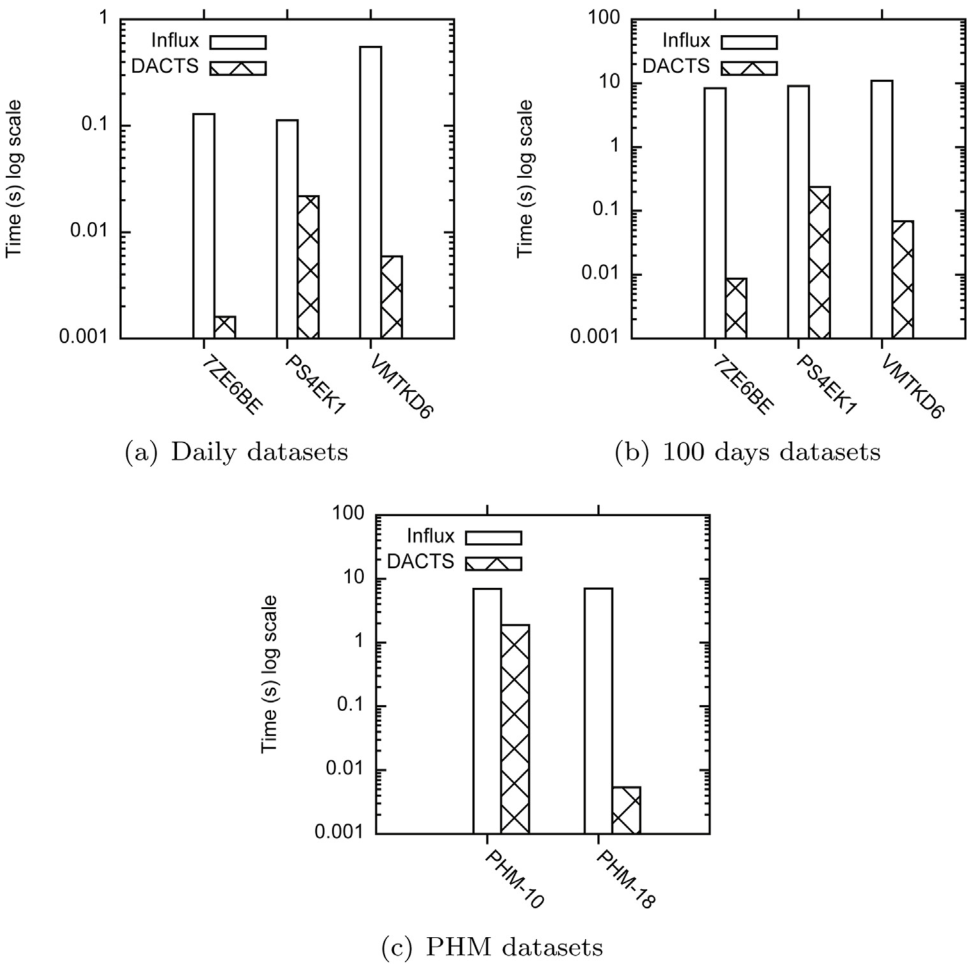


Fig. 11. Time to sort by Euclidean distance (seconds in logarithmic scale).

5. Conclusions

In this work, we have presented DACTS a lossless compressor for time series captured in industrial scenarios. We took Re-Pair, a compression method especially suited for datasets with a high level of repetitiveness, and we modified it in two aspects. First, Re-Pair does not provide random access to a given position without decompressing from the beginning. The typical solution to this problem is to sample regular positions, however, a sampled position might point to a position of the compressed sequence that contains a non-terminal, and thus, it represents several original terminals. Therefore, to solve that problem, we added an offset to the pointers. The second improvement is to use additional information in the rules of the grammar in order to speed up the queries over the compressed data.

DACTS shown in the experiments that it is well suited for repetitive datasets, where it can obtain a compression power close to `gzip` and even better in some cases. In the cases where the dataset does not contain repetitiveness, the compression, and also the access times, worsens. Still, it maintains, even in the worst cases, a compression power around that of `snappy`, a widely used compressor.

The main feature of DACTS is the improvement in speed. DACTS is up to 12.86 times faster than its competitors in extraction time. Thanks to the additional information in the rules, the improvement during the computation of the minimum and maximum values of a time interval are even better, up to 20.75 times faster. In the case of the similarity queries, the improvement is up to 11.67 times faster. The only exception is the hardest dataset for DACTS, `PHM-10`, where `dac` is 1.30 times faster when extracting, 1.11 when obtaining the minimum and the maximum, and 1.27 times faster when running the similarity queries. In this dataset, there are less repetitive sequences, this worsens the compression, and as explained, this decrease of repetitiveness affects DACTS to a greater extent than `gzip` and `p7zip`. But not only compression is affected, with less repetitiveness the sequences of symbols covered by rules are shorter. DACTS is faster when it deals with longer sequences of symbols therefore when the sequences of symbols covered by rules are shorter the answering time of queries increases.

As future work, we will work on improving compression, and developing new analysis queries.

Conflicts of interest

None.

Authors' contribution

Adrián Gómez-Brandón: Conceptualization, Software, Validation, Writing-Review & Editing. **José R. Paramá:** Conceptualization, Methodology, Software, Validation, Investigation, Writing-Original Draft, Writing-Review & Editing. **Kevin Villalobos:** Software, Validation, Investigation, Writing-Original Draft, Writing-Review & Editing. **Arantza Illarramendi:** Conceptualization, Writing-Original Draft, Writing-Review & Editing, Supervision. **Nieves R. Brisaboa:** Conceptualization, Supervision.

Acknowledgements

For the A Coruña team: This work was supported by CITIC, as Research Center accredited by Galician University System, is funded by "Consellería de Cultura, Educación e Universidade from Xunta de Galicia", supported in an 80% through ERDF Funds, ERDF Operational Programme Galicia 2014-2020, and the remaining 20% by "Secretaría Xeral de Universidades" (Grant ED431G 2019/01), Xunta de Galicia/FEDER-UE under Grants [IG240.2020.1.185; IN852A 2018/14] and Ministerio de Ciencia, Innovación under Grants [TIN2016-78011-C4-1-R; RTC-2017-5908-7]. For the Basque team: Ministerio de Ciencia, Innovación y Universidades under Grant [FEDER/TIN2016-78011-C4-2-R] and the Basque Government under Grant No. [IT1330-19]. Funding for open access charge: Universidade da Coruña/CISUG.

References

Aghabozorgi, S., Shirkhorshidi, A.S., Wah, T.Y., 2015. Time-series clustering – a decade review. Inf. Syst. 53, 16–38 http://www.sciencedirect.com/science/article/pii/S0306437915000733.

Amazon, 2020. AWS Resource & Custom Metrics Monitoring (accessed 28.04.20) https://aws.amazon.com/cloudwatch/faqs/?nc1=h_ls.

Blalock, D., Madden, S., Guttag, J., 2018. Sprintz: time series compression for the internet of things. Proc. ACM Interact. Mobile Wear. Ubiquit. Technol. 2 (3), 1–23.

Brisaboa, N., Ladra, S., Navarro, G., 2013. DACs: bringing direct access to variable-length codes. Inf. Process. Manage. 49 (1), 392–404.

Brisaboa, N.R., Fariña, A., Ladra, S., Navarro, G., 2008. Reorganizing compressed text. Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08), Singapore, 139–146.

Brisaboa, N.R., Farina, A., Navarro, G., Paramá, J.R., 2007. Lightweight natural language text compression. Inf. Retrieval 10 (1), 1–33.

Esling, P., Agon, C., 2012. Time-series data mining. ACM Comput. Surv. 45 (1), http://dx.doi.org/10.1145/2379776.2379788.

European Commission, 2014. Towards a Thriving Data-Driven Economy. Technical Report. https://ec.europa.eu/digital-singlemarket/news/communication-data-driven-economy.

Fu, T.-c., 2011. A review on time series data mining. Eng. Appl. Artif. Intell. 24 (1), 164–181 http://www.sciencedirect.com/science/article/pii/S0952197610001727.

Gog, S., Beller, T., Moffat, A., Petri, M., 2014. From theory to practice: plug and play with succinct data structures. Proc. 13th International Symposium on Experimental Algorithms (SEA), 326–337.

Gordevičius, J., Gamper, J., B&ldquo;ohlen, M., 2012 Jun. Parsimonious temporal aggregation. VLDB J. 21 (3), 309–332, http://dx.doi.org/10.1007/s00778-011-0243-9.

Kagermann, H., Wahlster, W., Helbig, J., 2013. Recommendations for implementing the strategic initiative Industrie 4.0: Final report of the Industrie 4.0 Working Group. Technical Report.

Klein, S.T., Shapira, D., 2016. Random access to fibonacci encoded files. Discrete Appl. Math. 212, 115–128, stringology Algorithms.

Külekci, M.O., 2014. Enhanced variable-length codes: Improved compression with efficient random access. Proceedings Data Compression Conference (DCC'14), Snowbird, UT, USA, 26–28 March, 2014, 362–371.

Kusiak, A., 2017 04. Smart manufacturing must embrace big data. Nat. News 544, 23–25.

Larsson, N.J., Moffat, A., 2000. Off-line dictionary-based compression. Proc. IEEE 88 (11), 1722–1732.

Lin, J., Keogh, E., Lonardi, S., Chiu, B., 2003. A symbolic representation of time series, with implications for streaming algorithms. Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery. DMKD '03. Association for Computing Machinery, New York, NY, USA, 2–11, http://dx.doi.org/10.1145/882082.882086.

Munro, J.I., 1996. Tables. Proc. 16th Conference Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 37–42.

Navarro, G., 2016. Compact Data Structures – A Practical Approach. Cambridge University Press.

Palpanas, T., Vlachos, M., Keogh, E., Gunopulos, D., Truppel, W., 2004 April. Online amnesic approximation of streaming time series. Proceedings of the 20th International Conference on Data Engineering, 339–349.

Risse, M., 2018. The new rise of time-series databases. https://www.smartindustry.com/blog/smart-industry-connect/the-new-rise-of-time-series-databases/.

Silva de Moura, E., Navarro, G., Ziviani, N., Baeza-Yates, R., 2000. Fast and flexible word searching on compressed text. ACM Trans. Inf. Syst. 18 (2), 113–139.

Tao, F., Qi, Q., Liu, A., Kusiak, A., 2018. Data-driven smart manufacturing. J. Manuf. Syst. 48, 157–169, Special Issue on Smart Manufacturing. http://www.sciencedirect.com/science/article/pii/S0278612518300062.

Vestergaard, R., Lucani, D.E., Zhang, Q., 2020. A randomly accessible lossless compression scheme for time-series data. IEEE INFOCOM 2020 – IEEE Conference on Computer Communications, 2145–2154.

Villalobos, K., Ramírez, V.J., Diez, B., Blanco, J.M., Goñi, A., Illarramendi, A., 2019. A hierarchical storage system for industrial time-series data. 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), vol. 1, 699–705.

Villalobos, K., Ramírez-Durán, V., Diez, B., Blanco, J., Goñi, A., Illarramendi, A., 2020. A three level hierarchical architecture for an efficient storage of industry 4.0 data. Comput. Ind. 121, 103257 https://www.sciencedirect.com/science/article/pii/S0166361520304917.

Wang, X., Mueen, A., Ding, H., Trajcevski, G., Scheuermann, P., Keogh, E., 2013. Experimental comparison of representation methods and distance measures for time series data. Data Mining Knowl. Discov. 26 (2), 275–309, http://dx.doi.org/10.1007/s10618-012-0250-5.

Yin, S., Kaynak, O., 2015. Big data for modern industry: challenges and trends [point of view]. Proc. IEEE 103 (2), 143–146.

Zhou, Z., 2019 May. Table Store Time Series Data Storage – Architecture. https://www.alibabacloud.com/blog/table-store-time-series-data-storage-architecture_594779.