An APCA-Enhanced Compression Method on Large-Scale Time-Series Data

Haiquan Wang
Department of Computer Science and Technology, Tsinghua University, Beijing, China
whq15@mails.tsinghua.edu.cn

ABSTRACT

With the widespread use of sensors, large-scale time series data becomes ubiquitous. As a result, it is important to provide efficient compression storage and retrieval algorithms, which are the base of subsequent data mining analysis, on these data. One of the most widely used data compression algorithms on time series data is Adaptive Piecewise Constant Approximation (APCA). However, APCA's compression storage overhead is still great for large-scale time series data. In this paper, we present a novel APCA-Enhanced algorithm. Extensive experiments over large real-world data sets demonstrate our algorithm's better performance of compression ratio and query latency compared with run length encoding and APCA.

1. INTRODUCTION

With the widespread use of sensors, large-scale time series data becomes ubiquitous. There has been much interest in data mining on time series data, such as clustering [2], classification [12][20] and mining of association rules [1]. As with all data mining problems, the key to effective and scalable algorithms is choosing a suitable representation of the data.

The most promising solutions involve performing dimensionality reduction, or referred to as compression, on the data in advance, then indexing the reduced data with an effective index structure. In choosing a dimensionality reduction technique, what is required is a technique that produces an indexable representation. For example, many time series can be efficiently compressed by delta encoding, but this representation does not lend itself to indexing. On the contrary, SVD, DFT, DWT,Piecewise Aggregate Approximation [13] (PAA) and Adaptive Piecewise Constant Approximation [11] (APCA) all lend themselves naturally to indexing.

In this work, we introduce a novel technique, namely APCA-E, which is based on APCA. As we will show, our APCA-E representation has several important advantages over existing technique. Specially, our algorithm, maintains lower compression ratio over several existing algorithms through grouping adjacent data points in time series, creating observation value encoding dictionary and generating new observation values with format of bits array. Compared to APCA, our APCA-E algorithm achieves several times improvement of compression ratio. In addition, an index tree, e.g. a B-Tree, is directly built on compressed series to support point query and range query, therefore it provides convenience for subsequent data mining algorithms.

The main contributions of this paper are the following:

- We propose a new APCA-enhanced compression and query method, maintaining several times improvement of compression ratio compared to APCA algorithm.
- An index structure is built on compressed data to achieve better performance of point and range query.

The rest of the paper is organized as follows. We survey several compression strategies for time series in Section 2. In Section 3, we give the representation of time series, APCA, compression ratio and query types offered in this work. The details of APCA-E algorithm are presented in Section 4. We describe how to build an index on compressed series to support point and range query in Section 5. Comprehensive experimental results on large real and artificial data sets are reported in Section 6. The paper is concluded in Section 7.

2. RELATED WORK

RLE (Run Length Encoding) substitutes consecutive symbols which are the same value with a symbolic value or string length to reduce the storing length of data. RLE was employed in the transmission of television signals as far back as 1967 [22]. Nowadays, RLE is also used in many applications to compress series [10]. The advantage is that the amount of data can be compressed into small units of high repeatability, however, the disadvantage is that if the data frequency is not high, the compressed data's size may be larger than the original data.

PAA (Piecewise Aggregate Approximation) [13] [28] method slides a fixed size sliding window in time series and calculates the mean of the data in the sliding window. The disadvantages of PAA are as follows: (1) The size of the sliding window is a key factor; (2) Smoothing the series by means of the averaging method may lost some characteristic information (3) Not taking the characteristic that the reference value of past time series for future series value grows over time into account, the algorithm treats each segment of the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM TUR-C '17, May 12-14, 2017, Shanghai, China

© 2017 ACM. ISBN 978-1-4503-4873-7/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3063955.3063975

time series equally, thus not fitting the pump-style sliding window in stream sequence processing.

PLA (Piecewise Linear Approximation) [6] [23] method first segments the time series and treats each segment as a piece of data that has a linear relationship, and then uses linear functions to fit each segment of data with minimal least squares error. The PLA method can fit the morphological similarity sequence to the original data, and keep the mutation point in the sequence. The main drawback of this method is the high computational complexity. For time series with length n (n>0), the process to obtain the optimal segmentation fitting with N(0<N<n) segments costs time $O(Nn^2)$.

APCA (Adaptive Piecewise Constant Approximation) [11] method segments the time series into a series of variable-length sub-segments, each of which is represented by a pair of the mean value and the rightmost time-scale value of the segment. Compared to the PAA method, it overcomes the limitation of the requirement for a fixed sliding window size, but it suffers from the following drawbacks: (1) Under the same conditions, it requires twice as much storage as the PAA method; (2) To complete the N-state (0<N<n) APCA representation of time series with length n (n > 0), APCA algorithm's time complexity is $O(Nn^2)$. Despite some optimal strategies, the algorithm's time complexity is also $O(nlog(n))$.

There are also some similarity-based methods [19, 25, 16, 27, 8, 9, 18, 3, 4, 7, 26, 17, 19].

3. PRELIMINARIES

3.1 The time series representation

A time series is represented as $C = (t_1, v_1), (t_2, v_2)$,..., ($t_n$, v_n), where t_i is the sampling time and v_i the corresponding observation value. The difference between t_i and t_{i+1} denotes the sample time interval. In this paper, we assume the sampling time is represented as the UNIX timestamp. The observation value type differs in application scenarios and can be integer, boolean or float type and so on.

3.2 The APCA Representation

Given a time series $C = (t_1, v_1), (t_2, v_2), ..., (t_n, v_n)$, we are able to produce an APCA representation, which we will represent as $C = (t_{1'}, v_{1'}), (t_{2'}, v_{2'}), ..., (t_{m'}, v_{m'})$, where $v_{i'}$ is the mean value of data points in ith segment. The length of the ith segment can be calculated as $t_{(i+1)'} - t_{i'}$.

3.3 Compression ratio

To evaluate the effect of compression algorithms, we introduce the concept of compression ratio, which is the ratio of the compressed file size to the pre-compressed file size. The formula definition is as follows:

$$compression\ ratio = \frac{compressed\ file\ size}{pre-compressed\ file\ size}.$$

Obviously, for compression ratio, the smaller is the better.

3.4 Query type

In this paper, we focus on the query operations over time series data. Formally, consider a time series data set C with N records, where each record c of C has two properties, i.e. sampling time t and observation value v. We consider the following two main query operations, due to the their wide application scenarios.

Definition 1 (Point Query) Given a query sampling time instant t_q and a compressed time series data set C, a point query (denoted as $point(C, t_q)$) asks for the observation value corresponding to the query sampling time instant t_q in C. Formally, $point(C, t_q) = c$, where $c.t = t_q$ and $c \in C$.

Definition 2 (Range Query) Given a query sampling time interval Q and a compressed time series data set C, a range query(denoted as $range(C, Q)$) asks for all the sampling records, whose sampling time instant is in the interval Q, from C. Formally, $range(C, Q) = \{c | c \in C, c.t \in Q\}$.

4. THE APCA-ENHANCED COMPRESSION ALGORITHMS

4.1 Generating APCA results

The first step of APCA-Enhanced compression algorithm is to convert original time series data to APCA results, which has the representation described in section 3.2. In general, finding the optimal piecewise polynomial representation of a time series requires a $O(Nn^2)$ dynamic programming algorithm [5] [21]. In most cases, however, an optimal representation is not necessary. Many researchers, therefore, use a greedy suboptimal approach instead [23][14][24]. Another strategy proposed in [11] works by first converting the problem into a wavelet compression problem, for which there are well known optimal solutions, then converting the solution back to the APCA representation and making minor modifications. This strategy produces high quality approximations in $O(nlog(n))$. All of the algorithms above achieve a good performance on small datasets. As the scale of dataset increases, however, the performance of pre-compression process decreases.

[15] presents a method, which finishes in linear time, to fetch the approximate segmentation of APCA. In this section, we use the method proposed in [15] to generate APCA results in O(n).

4.2 The enhanced compression strategy (APCA-E)

After generating APCA results, our enhanced compression strategy is applied to obtain final results. The basic idea of this process is to merge multiple tuples, which have the format (t, v), into another new tuple, whose format remains the same. Data manipulation contained in the entire process is lossless and compressed data can be easily and efficiently decompressed. Before describing the detail, we here define some parameters used in the algorithm:

- **P** The maximum number of tuples in each group.
- **JUMP** The maximum sampling time interval between adjacent tuples.
- **JNB** The number of bits to store $JUMP$. $JNB = \lceil log_2(JNB) \rceil$
- **V** The total number of possible observations.
- **VNB** The number of bits to encode all possible observations. $VNB = \lceil log_2(VNB) \rceil$

We need to give each possible observation a binary encoding in advance, thus a hash table is built. In the later algorithm execution step, the binary encoding of each observation is stored rather than the original value.

The algorithm reads the first P tuples to be processed in the input sequence. These P tuples are scaned from the beginning to the end and the first $k(k <= P)$ tuples, which

satisfy that each sampling time interval of adjacent tuples is less than the maximum value $JUMP$ predefined, are extracted. Obviously, if k is less than P, the time interval of the kth tuple and the $(k+1)th$ tuple must be greater than $JUMP$.

These first k tuples will be merged into another new tuple, which has the notation *new_tuple* (we use the notation *tuple* to represent that tuples from the original APCA results). The sampling time property of *new_tuple*, i.e. *new_tuple.t*, equals to that of the first one in the first k tuples.

Observation value property of *new_tuple*, i.e. *new_tuple.v* is generated from the first k tuples. The *new_tuple.v* is stored using a bit array, which contains two segments logically. One segment is used to store the k-1 time intervals belonging to the first k tuples, while another contains all the observation value of the first k tuples.

All the observation value of the first k tuples is stored in the second segment. The content is actually, however, the binary hash encoding obtained from the hash table pre-created rather than the original observation value. Similar to the first segment, the length of the second segment is $P \times VNB$ rather than $k \times VNB$. The last $(P-k) \times VNB$ bits are also filled with zero. By joining the two segments, we obtain the *new_tuple.v*, which contains $(P-1) \times JNB + P \times VNB$ bits. What we need to do next is just to repeat the construction process on the APCA results until the sequence ends, while the whole enhanced compression process is finished.

Algorithm 2 gives more detail of this process. We use the notation $C[i]$ to access the ith element of APCA results time series. Furthermore we use $C[i].t$ and $C[i].v$ to fetch the sampling time and the observation value. Our algorithm scans the input sequence only once to generate the final results, therefore, the enhanced compression strategy can be finished in O(n). There is, however, an additional space overhead to store the hash table of all possible observation value. In fact, this additional space overhead is negligible.

Algorithm 2: Generate APCA-E results

Input:
- $D[1:n]$ - APCA results array; each element d is a tuple $< t, v >$; $d.t$ represents the segment start timestamp; $d.v$ represents the segment mean value;
- P - the maximum number of tuples in each group;
- $JUMP$ - the maximum timestamp interval between adjacent tuples;
- V - the total number of possible observations;
- h_t - the hash table storing all possible observations.

Output: the APCA-E representation results E

```
1  JNB = ⌈log2(JUMP + 1)⌉;
2  VNB = ⌈log2V⌉;
3  left ← 1; right ← 2; num ← right − left;
4  last_tuple = C[left];
5  tmp_diff = 0; tmp_value = h_t(D[left].v);
6  while right ≤ n do
7      num ← right − left;
8      if (num > P) OR
9      (num < P AND C[right].t−last_tuple.t > JUMP)
       then
10         store_value =
11           (tmp_diff <<
12                (P∗VNB+(P−1−(num−2))∗JNB)) |
13           (tmp_value << (P − (num − 1)) ∗ VNB);
14         E.append(< C[left].t, store_value >);
15         left ← right;
16         tmp_diff ← 0;
17         tmp_value ← h_t(D[left].v);
18         num ← 1;
19         last_tuple ← C[left];
20     else
21         tmp_diff = (tmp_diff << JNB) |
22                    (C[right].t − last_tuple.t);
23         tmp_value = (tmp_value << VNB) |
24                    h_t(C[right].v);
25         last_tuple = C[right];
26     right ← right + 1;
27 store_value =
28   (tmp_diff <<
29       (P ∗ VNB + (P − 1 − (num − 1)) ∗ JNB)) |
30   (tmp_value << (P − num) ∗ VNB);
31 E.append(< C[left].t, store_value >);
32 return E;
```

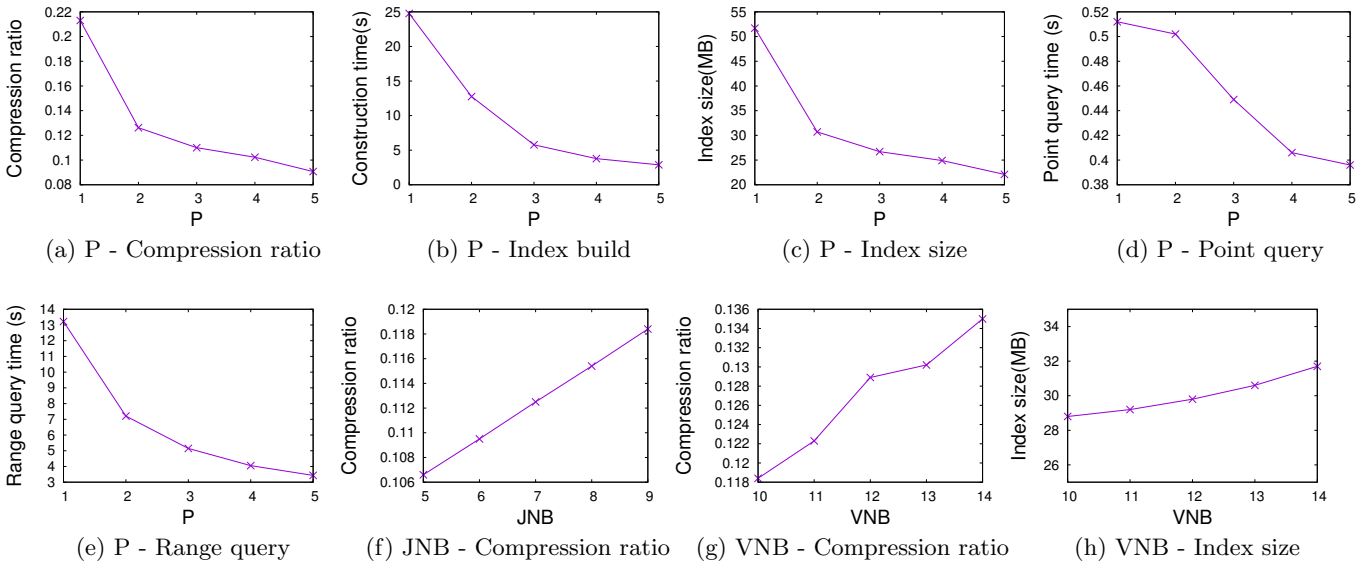
Figure 1: Generate APCA-E results

5. BUILDING INDEX AND QUERYING

Through the compression procedures described in section 4, a raw time series is converted by APCA at first and the enhanced strategy, into final compression results. Data compression strategies achieve good storage performance, meanwhile some basic query operation types on data stored are necessary for further data analysis. To avoid the additional cost of decompression, we build an index on the compressed data directly. The sampling time of the compressed series is chosen as the key of a B-Tree leaf node, whose corresponding value is the merged binary bits array described in section 4.2. Here the index built on the compressed data supports two query operation: point query and range query.

Point Query A point query asks for the observation value corresponding to the query sampling time in time series. We query a sampling point in two steps: (1) find the biggest index key that is not bigger than the query sampling time utilizing the index built on the compressed time series and obtain the key's value; (2) decompress the binary bits array stored in the value to fetch the APCA results and generate the approximation of the query time point. Obviously, this solution can be finished in $O(log(N)+P)$, where P is defined in section 4.2.

Range Query A range query returns all the observation value corresponding to the query time range in time series. We process the range query on compressed time series in two steps: (1) execute two point queries corresponding to the beginning and end of the query time range utilizing the B-Tree index built on the compressed time series to fetch the beginning leaf node and end leaf node, then take all the leaf nodes between the beginning and end via the linked list of all index leaf nodes; (2) decompress the leaf nodes extracted in (1) to obtain the APCA results, which is ordered according to the feature of B-Tree, and scan all the APCA results to generate all the approximation observation value corresponding to the sampling time in query time range. The process is finished in $O(log(N)+k+P)$, where k is the length of query range and P is defined in section 4.2.



(a) P - Compression ratio (b) P - Index build (c) P - Index size (d) P - Point query

(e) P - Range query (f) JNB - Compression ratio (g) VNB - Compression ratio (h) VNB - Index size

Figure 2: Effect of three parameters in APCA-E.

6. EXPERIMENT

6.1 Experiment Setup

All experiments were conducted on a machine with a 32-core Intel Xeon E5-2630 v3 2.40GHz processor and 64GB RAM. This machine runs Ubuntu 14.04.5 LTS with gcc 4.8.4.

We used the following real and synthetic datasets:

- **SGCC-FD** stands for STATE GRID Corporation of China frequency datasets, which contain almost 100 million real-time frequency records monitored per second in the last three years.
- **RD**: We generated synthetic time series datasets of various size(10 million to 50 million records) using random way.

Specially, we generated 100 thousand cases of point query and 30 thousand cases of range query respectively to evaluate the total time cost, thus improving the accuracy of the results.

We designed a series of experiments to evaluate the six performance criteria: compression time, compression ratio, index construction time, index storage size, point query and range query, of our algorithm. We also compare our algorithm with other 2 typical compression algorithms: run length and APCA, to demonstrate our algorithm is more effective. In comparison experiments, we set the three parameters defined in section 4.2 as: $P = 3, JNB = 7, VNB = 10$, and the error bound of APCA algorithm is set to 0.005.

6.2 Effect of three parameters in APCA-E

In this section, we elaborate the effect of three parameters: P, JNB, VNB, which are defined in section 4.2, on total compression time, compression ratio, index construction time, index storage space, point query and range query. We selected SGCC-FD dataset with 20 million records as the experiment dataset. A B-tree is chosen as the index structure thus index construction time representing the time consuming of building the whole B-tree. Point query was executed 100 thousand times and range query was executed 30 thousand times respectively.

6.2.1 *Effect of P*

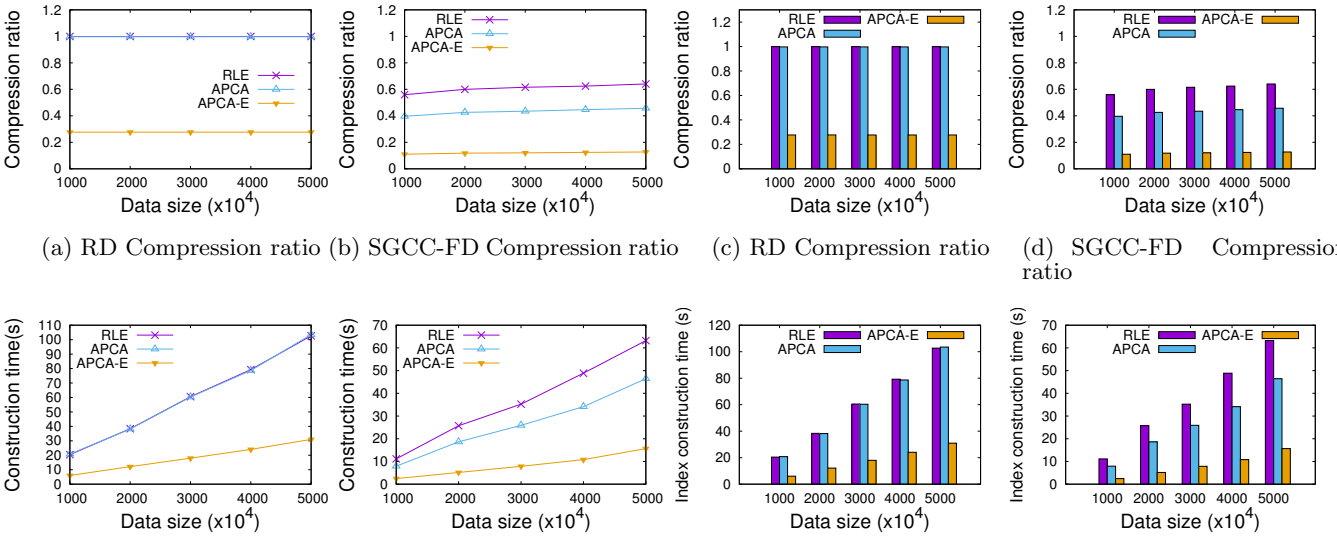
Figure 2.(a)-(e) represent the effect of parameter P on compression ratio, index construction time, index storage space and performance of point query and range query. The figure 2.(a) depicts that compression ratio defined in section 3.3 decreases with the increase of parameter P. If parameter P increases, the number of sample points stored in one group grows. Therefore, the algorithm APCA-E compresses more data to store, thus reducing the compression ratio. We can find that the index construction process costs less time in figure 2.(b) and the index size occupies less storage space in figure 2.(c) as parameter P grows bigger. This result is related to the compression ratio. The less compression ratio is, the smaller number of total new sample points is. It is, therefore, easy to understand that index construction process costs less time and index size is smaller if the compression ratio is lower. The latency of point query and range query becomes lower and lower as parameter P grows bigger and bigger, which is depicted in figure 2.(d) and 2.(e). The performance of query operation is much relevant to the index size which determines the height of index structure (e.g. B-Tree). The shorter the index structure is, the faster the query process is. We have found that the index size grows low as the parameter P increases, as a result the query operation becomes faster.

6.2.2 *Effect of JNB*

The figure 2.(f) reflects the effect of parameter JNB on compression ratio. From the figure, we can conclude that compression grows high as the parameter JNB increases. The parameter JNB represents the binary bits count of the maximum time step between neighboring sample points compressed in one group from APCA results. There are $(P-1) \times JNB$ binary bits in the value corresponding to the key generated by APCA-E from APCA results . Therefore, if the parameter JNB grows high, there need more binary bits to store the new sample value, thus worsening the performance of compression.

6.2.3 *Effect of VNB*

The effect of parameter VNB on compression ratio and index size is shown in figure 2.(g)-(h). The increasing of parameter VNB means that there are more different kinds of sample value. Therefore, the binary bits count needed



(a) RD Compression ratio (b) SGCC-FD Compression ratio (c) RD Compression ratio (d) SGCC-FD Compression ratio

(e) RD Index build (f) SGCC-FD Index build (g) RD Index build (h) SGCC-FD Index build

Figure 3: Performance of compression ratio and index build time.

to store the sample value grows, thus extending the length of each so-called sample value. It is not surprising to find that compression performance decreases. However, the degree of compression performance's reducing is very limited. As compression ratio increases, the compressed data size increases. Therefore, index size increases, for index is built directly on compressed data. It is the same as the change of compression ratio that extra storage space size caused by the increasing of parameter VNB is very small.

6.3 Comparison with existing algorithms

In this section, we compare APCA-E algorithm with two existing compression algorithms, that is run length encoding(RLE) and APCA. A series of experiments are conducted to examine this three algorithms' performance of compression time, compression ratio, index construction time, index size, point query and range query, on two different datasets, that is $SGCC - FD$ and RD described in section 6.1. To test the impact of data scale, each dataset's size varies from 10 million to 50 million records. The parameters used in APCA-E here are as follows: $P = 3, JNB = 7, VNB = 10$. We give analysis on the sixth performance criteria next.

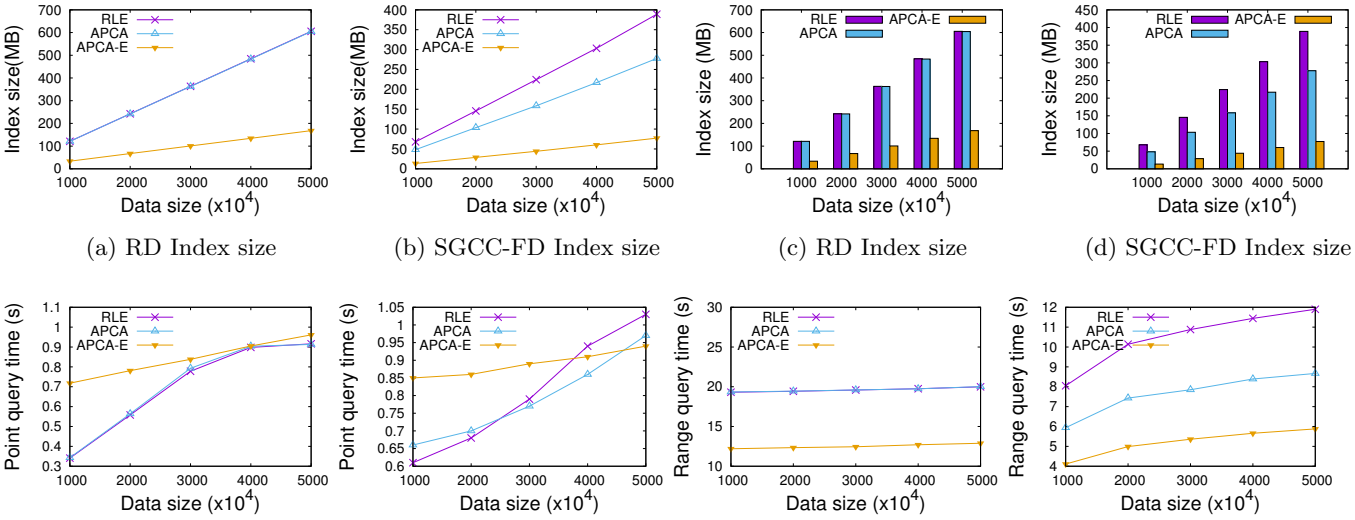
Compression ratio The compression ratio results of three algorithms on different scale of datasets RD and $SGCC - FD$ are shown in figure 3.(a) and figure 3.(b). We can find that all compression ratio of the three algorithms almost remains the same though the dataset scale grows. This result proves that our algorithm APCA-E, like two other algorithms, can maintain consistent compression performance as the dataset scale changes. Another important result APCA-E achieves the lowest compression ratio among the three algorithms. We show the compression ratio histogram results of three algorithms on datasets RD and $SGCC - FD$ with 30 million records in figure 3.(c)-(d). It is intuitive to find that APCA-E achieves at least 5x and 3x improvement of compression ratio than RLE and APCA respectively.

Index construction time The figure 3.(e)-(f) show the results of index construction time on two datasets. We can find that index construction time of three algorithms increases as dataset scale grows as index construction time is directly related to compressed data size. The relation between total index construction time and dataset scale presents linear, because index construction process can finish in O(n). In addition, APCA-E algorithm's index construction process costs the least time among the three. To show it intuitively, we also plot the histogram results on datasets with 30 million records in figure 3.(g) and 4.(h). Because APCA-E algorithm's compression ratio is the lowest, index construction time is the least naturally.

Index size The correlation results of three algorithms between index size and dataset scale are shown in figure 4.(a)-(b). It is intuitive that index size and dataset scale is linearly related. From the figures, we can find that APCA-E's index size is the lowest among the three. We also show the histogram results on datasets with 30 million records in figure 4.(c)-(d). The index size of APCA-E is just 20% and 25% of that of RLE and APCA. The reason is that the compression ratio of APCA-E is the least and index is built on the compressed data.

Point Query Point query experiment results are shown in figure 4.(e) and 4.(f). It is obvious that point query latency increases as dataset scale grows. With the growing of datasets scale, the index size built on the compressed data increases thus raising the height of index and increasing the cost. The figures show that APCA-E's point query performance is worse than the other two, which is mainly because APCA-E stores more points in one group thus taking extra partial decompression cost.

Range Query The range query latency results of three algorithms are shown in figure 4.(g)-(h). The two figures depict that range query latency increases as dataset scale grows. With the growing of dataset scale, the index increases naturally thus raising the height of index tree, therefore, range query's latency increases. It is intuitive to find that APCA-E achieves the best range query performance among the three. The reason is related to range query execution procedure. The procedure determines two leaf nodes corresponding to the two end points of the range at first, and then takes out all leaf nodes between the starting leaf node and the terminal. As APCA-E achieves the lowest compression ratio thus building the lowest index tree, as a result, the number of total leaf nodes taken out from the index is the least. Therefore, APCA-E has the lowest latency when executing a range query.



(a) RD Index size (b) SGCC-FD Index size (c) RD Index size (d) SGCC-FD Index size

(e) RD Point query (f) SGCC-FD Point query (g) RD Range query (h) SGCC-FD Range query

Figure 4: Performance of Index size and Query latency.

7. CONCLUSIONS

The main contribution of this paper is to propose a new APCA enhanced dimensionality reduction method, namely APCA-E. Comparing with APCA and RLE, APCA-E has several times improvement of compression ratio. Comprehensive experiments also show that APCA-E's query performance is excellent. Specially, its range query performance is better than APCA and RLE. Future directions include further enhancing the performance of APCA-E, while applying it to an actual system.

Acknowledgment

This work was supported by 973 Program of China(2015CB358700), NSF of China (61373024,61422205,61472198).

8. REFERENCES

[1] G. Das, K.-I. Lin, H. Mannila, G. Renganathan, and P. Smyth. Rule discovery from time series. In *KDD*, volume 98, pages 16–22, 1998.

[2] A. Debrégeas and G. Hébrail. Interactive interpretation of kohonen maps applied to curves. In *KDD*, volume 1998, pages 179–183, 1998.

[3] D. Deng, G. Li, and J. Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD*, pages 673–684, 2014.

[4] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*, pages 340–351, 2014.

[5] C. Faloutsos, H. Jagadish, A. O. Mendelzon, and T. Milo. A signature technique for similarity-based queries. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 2–20. IEEE, 1997.

[6] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In R. T. Snodgrass and M. Winslett, editors, *SIGMOD*, pages 419–429. ACM Press, 1994.

[7] J. Fan, G. Li, L. Zhou, S. Chen, and J. Hu. SEAL: spatio-textual similarity search. *PVLDB*, 5(9):824–835, 2012.

[8] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.

[9] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[10] T. Kawano. Run-length encoding graphic rules, biochemically editable designs and steganographical numeric data embedment for dna-based cryptographical coding system. *Communicative & integrative biology*, 6(2):e23478, 2013.

[11] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM SIGMOD Record*, 30(2):151–162, 2001.

[12] E. J. Keogh and M. J. Pazzani. An enhanced representation of time series which allows fast and accurate classification, clustering and relevance feedback. In *KDD*, volume 98, pages 239–243, 1998.

[13] E. J. Keogh and M. J. Pazzani. A simple dimensionality reduction technique for fast similarity search in large time series databases. In *PAKDD*, pages 122–133. Springer, 2000.

[14] E. J. Keogh and P. Smyth. A probabilistic approach to fast pattern matching in time series databases. In *KDD*, volume 1997, pages 24–30, 1997.

[15] D. Li. *Studies on Time-series Data*. PhD thesis, 2008.

[16] G. Li, D. Deng, J. Wang, and J. Feng. PASS-JOIN: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[17] G. Li, J. He, D. Deng, and J. Li. Efficient similarity join and search on multi-attribute data. In *SIGMOD*, pages 1137–1151, 2015.

[18] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a TASTIER approach. In *SIGMOD*, pages 695–706, 2009.

[19] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914, 2008.

[20] M. K. Ng, Z. Huang, and M. Hegland. Data-mining massive time series astronomical data setsâĂŤa case study. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 401–402. Springer, 1998.

[21] T. Pavlidis. Waveform segmentation through functional approximation. *IEEE Transactions on Computers*, 100(7):689–697, 1973.

[22] A. Robinson and C. Cherry. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3):356–364, 1967.

[23] H. Shatkay and S. B. Zdonik. Approximate queries and representations for large data sequences. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 536–545. IEEE, 1996.

[24] C. Wang and X. S. Wang. Supporting content-based searches on time series via approximation. In *SSDBM*, pages 69–81. IEEE, 2000.

[25] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.

[26] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

[27] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD*, pages 85–96, 2012.

[28] B.-K. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary lp norms. VLDB, 2000.