RpDelta: Supporting UCR-Suite on Multi-versioning Time Series Data

Xiaoyu Han[1], Fei Ye[1], Zhenying He[1,2](✉), X. Sean Wang[1], Yingze Song[3], and Clement Liu[4]

[1] School of Computer Science, Fudan University, Shanghai, China
{xyhan22,fye21}@m.fudan.edu.cn, {zhenying,xywangCS}@fudan.edu.cn
[2] Shanghai Key Laboratory of Data Science, Shanghai, China
[3] University of Liverpool, Liverpool, UK
sgyson16@liverpool.ac.uk
[4] Chase Grammar School, Cannock, UK
clement.liu@chasegrammar.com

Abstract. In real applications, various cleaning strategies are adopted to repair a specific time series several times for better effects. These multiple versions of the repaired time series, along with the raw time series, are often stored directly in the system for the users. However, as the scale of data explodes, high storage cost becomes a non-negligible problem. To address this problem, we propose RpDelta, a repaired time series storage strategy, under which a repaired time series can be represented as the combination of the raw time series and a differential file to use the storage space more efficiently. Meanwhile, we design a sequential reading strategy based on a finite state machine to make RpDelta adaptive to practical uses, which will almost not introduce additional time and space overheads. We also take the UCR-Suite algorithm as an example to introduce our optimizations on a simultaneous-operation circumstance with the help of RpDelta's properties. The extensive experiments show the effectiveness and efficiency of our work.

Keywords: Repaired time series · Multiple versions · Differential file

1 Introduction

Data quality for time series is critical in time series analysis and forecasting. The credibility of the results depends on assumptions that collected time series are reliable, which only sometimes hold in reality. For example, in finance, the correct rate of stock information on Yahoo Finance is 93% [11]. Moreover, in manufacturing, the collected data may be partially noisy or missing due to the unreliability of the physical sensor devices [4]. Therefore, before operating on time series, to avoid the degradation of analysis and forecasting caused by poor data quality, the data quality of time series should be checked from several aspects, such as data validity, completeness, and consistency [9].

X. Wang et al. (Eds.): DASFAA 2023, LNCS 13943, pp. 205–220, 2023.
https://doi.org/10.1007/978-3-031-30637-2_14

Recent works have proposed many data-cleaning algorithms to repair time series data to improve data quality. However, as we discuss below, it is difficult for a single data-cleaning algorithm to meet the requirements of real-world applications. Therefore, we will adopt different cleaning algorithms or parameters under one specific cleaning algorithm for better effects according to our comprehension of the collected data as time passes. In this way, we cannot overwrite the raw time series after we get a repaired one, which means that the raw one is non-tamperable [8]. Actually, Apache IoTDB integrates data-cleaning algorithms as functions into SQL operations, which keeps the raw time series unchanged unless we use update operation. So, a better way is to store multiple repaired time series separately from the original time series. Figure 1(a) shows the different versions of repaired time series derived from the same raw time series.

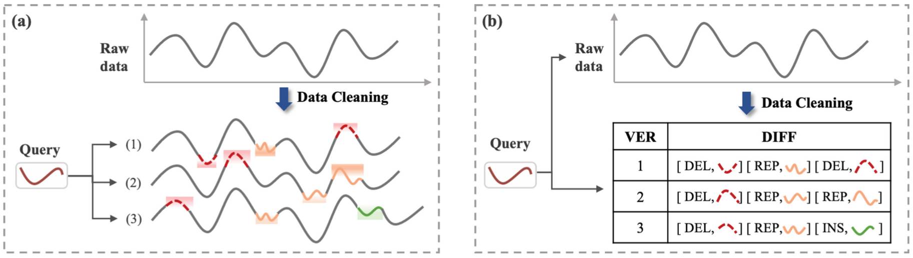
(a)
Raw data
Data Cleaning
Query
(1)
(2)
(3)
(b)
Raw data
Data Cleaning
Query
VER
DIFF
1
[DEL,] [REP,] [DEL,]
2
[DEL,] [REP,] [REP,]
3
[DEL,] [REP,] [INS,]

Fig. 1. (a) Subsequence matching on traditional multiple versions of repaired time series.(b) An optimization of subsequence matching under RpDelta storage strategy.

Under the above constraint of storing both the original time series and multiple repaired time series, there are several fundamental problems:

1. **Unnecessary Storage Use.** What is required to be repaired only accounts for a small part of the raw time series, generally at most 15%, because anomalies are often rare [2] or the collected data is untrusted. In other words, the repaired time series stores a lot of data identical to the raw one, which causes an enormous waste of space resources.
2. **High Disk I/O.** Sometimes we need to conduct the same operation(e.g., subsequence match) on all the repaired time series to analyze the effects of the data-cleaning algorithms and ensure robustness. To achieve this, we must read all the repaired time series, thus leading to a great deal of disk I/O, including many redundancies and jeopardizing the life of the disks.
3. **Extra Operation Time Overheads.** Considering the similarities between different versions of repaired time series, we will perform many repetitive calculations during the operation, which incurs extra time overheads.

To solve the above problem, we design a new storage strategy named RpDelta for multiple versions of repaired time series. Besides the raw time series, RpDelta stores all the repair operations from the raw to a repaired time series in a differential file called delta file. The number of delta files corresponds to the number of repaired versions. In this way, we can lower the storage cost from a wholly repaired time series to a delta file containing only necessary repairs. Since we can quickly restore the repaired time series according to its corresponding delta file and the raw time series, this will not harm the data integrity. Moreover, we redesign the sequential reading strategy under RpDelta based on the finite state machine approach and use the subsequence matching algorithm UCR-Suite [6] as an example to introduce the way to lower the disk I/O and repetitive calculations. Figure 1(b). shows the basic idea of RpDelta. It can be seen as extracting the unique part of each repaired time series to organize different delta files to replace the complete series. Meanwhile, the database engine can encapsulate all operations modifications under the RpDelta structure. Users do not need to do additional operations by themselves and can directly call the given interface.

To sum up, the main contributions of our paper are summarized as follows:

– We propose RpDelta, a storage strategy aimed at multiple versions of the repaired time series, which can significantly lower storage cost while maintaining data integrity.
– We further propose a sequential reading strategy under RpDelta based on the finite state machine approach. This approach possesses the same time complexity $O(n)$ and space complexity $O(1)$ as reading the complete time series directly, without prior restoration of the complete repaired time series.
– We take the UCR-Suite subsequence searching algorithm as an example to introduce the optimization of the circumstance about performing the same operations on all repaired time series. This optimization can efficiently reduce the disk I/O and unnecessary time overhead.
– We conduct extensive experiments on the UCR Time Series Archive. They demonstrate our optimizations of storage use, disk I/O and time overhead.

2 Preliminaries

2.1 Basic Concepts

Definition 1 (Time Series). A *time series* is a sequence of data points ordered by the time that can be denoted as the form $T = (t_1, t_2, \dots, t_n)$, where $n = |T|$ stands for the length of time series T. A subsequence of T with length k is part of the whole time series that can be denoted as the form $T_{i,k}$, where i stands for its start position and satisfies $1 \le i \le n - k + 1$. For q time series, we use $T^1, T^2, \dots, T^q$ to distinguish them and the same for subsequences $T^1_{i,k}, T^2_{i,k}, ...T^q_{i,k}$.

Definition 2 (Distance Metrics). Here we give the definition of two existing distance metrics used for time series calculation in our work, *Euclidean distance* (ED) and *Dynamic Time Warping* (DTW).

Given two time series T and T' with length n:

$$ED(T,T') = \sqrt{\sum_{i=1}^{n}(t_i - t'_i)^2} \tag{1}$$

$$\begin{aligned} &DTW(O,O) = 0, DTW(O,T) = DTW(T,O) = +\infty \\ &DTW(T,T') = \sqrt{min \begin{cases} DTW(T_{1,n-1}, T'_{1,n-1}) \\ DTW(T, T'_{1,n-1}) \\ DTW(T_{1,n-1}, T') \end{cases} + (t_n - t'_n)^2} \end{aligned} \tag{2}$$

where O stands for empty series, $T_{1,0} = T'_{1,0} = O$.

We use $dist(T,T')$ to signal the distance between T and T$'$ in this paper.

2.2 Problem Statement

Definition 3 (Subsequence Matching Problem). Given a time series T with length n and a query time series Q with length m(usually $m \ll n$), find a subsequence of T starting from position i which minimizes $dist(T_{i,m}, Q)$.

3 Repaired Time Series Storage

3.1 RpDelta Construction

The design for the new storage strategy RpDelta is inspired by the properties of the repaired time series. We investigated that the content needed to be repaired in a time series often accounts for only about 3% to 5% of the whole, at most no more than 15%, so most of the time series data after the repair remains the same as the raw series. In this case, considering the immutability of the raw series, we can use a differential file to record all the repair operations needed from the raw series to the repaired time series, which is similar to the binlog in MySQL. Here we denoted this differential file as a delta file. In Fig. 2, the delta file records three different operations that are above the arrow. We can quickly obtain a complete repaired time series by sequentially implementing those operations on the raw time series, as the Fig. 2 shows. This strategy helps us to ensure data integrity while lowering the storage costs. Our later experiments prove this point.

3.2 Delta File StorageFormat

We next define the exact form of the delta files. Here how to represent a repair operation is the foremost thing. After we research the current data cleaning algorithms, we classify the operations into three basic types: insert, delete and replace. We may use insertion in interpolation, deletion in removing repetitive samplings, and replacement in fixing the noises. The other complicated operations can be decomposed into these three types.

Moreover, we involve three parameters: start position, operation length, and data points. All the operations need start position and operation length to describe their scopes. The start position of insertion means we will insert a segment before the position in the raw series. The start position of deletion means we will delete a segment with this position in the raw series as the first data point. Only insertion and replacement need data points to determine their operation contents. In this way, we can define an operation as *"Type Length Position Data(Array)"*. Take "INS 2 1 [9, 6]" as an example, it means that we will insert an array [9, 6] whose length is 2 into the position before the time point 1. We can see this example and other operations in Fig. 2.

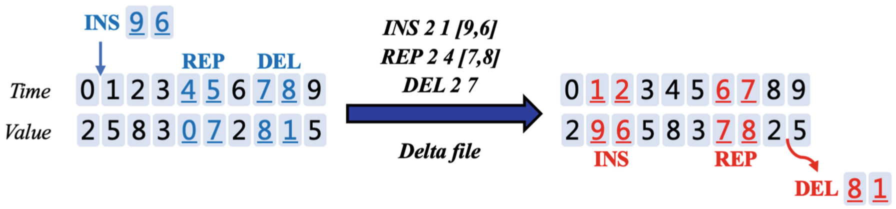

Fig. 2. Using a raw time series and a delta file to restore the repaired time series.

Besides, the data points in different time series have distinct types for various uses. Suppose in a situation requiring high precision, treating double data as float in the delta files is a disaster. So we should be informed of the data points' type when recording the repaired operations. At the beginning of the delta file, we can include the type information matching with the data points in the time series. Of course, we can add other information as well if necessary.

In short, a delta file involves two parts, the first part is some basic information about the repair operations and the corresponding time series, and the second part is a set of operations stored in the above format.

3.3 Delta File Constraints

To avoid ambiguity in different organizational forms in delta files, here we make some reasonable constraints to make our processing and explanation clearer and more convenient.

Constraint 1 (Position Constraint). A repair operation's start position refers to the raw series's corresponding position

For insertion, it may cause the index position in the raw series to change, so we need to make this constraint to ensure a delta file is free from duality of position. Sometimes the repair positions are off the raw series, such as inserting first and then restricting the speed of the inserted segment. However, we can merge them into one operation of inserting an already speed-restricted segment, performed on the raw series and equivalent to the previous two operations. The above shows the rationality of this constraint.

Constraint 2 (Overlap Constraint). There is no overlap between the two repair operations
According to constraint 1, we will not perform any operation on the inserted or deleted segments. If two insertion or deletion operations are performed in the same position, we can merge them according to their order. If an insertion operation shares the same position with a deletion or replacement operation, that has no problem because the inserted segment will not be affected. Some illegal situations, such as replacing or deleting some parts of a deleted subsequence, will not be allowed in our delta files. In this way, this constraint is reasonable.

Constraint 3 (Order Constraint). The repair operations are sorted by the position order in the delta file
Combining constraint one with constraint two, we can conclude that each repair operation is independent of the other. So swapping the order between any two operations in the delta file will not affect the correctness of the whole repair process. On this occasion, to deal with RpDelta more conveniently, we sort the repair operations in the delta files by position in ascending order.

4 Operations and Optimizations on RpDelta

When implementing analysis on time series, we are likely to perform operations on them. Sometimes we need to find the best-matching subsequence, while other times, we need to cluster multiple different time series. The existing algorithms are based on the input of a complete time series, however, under RpDelta each time series consists of two parts, thus making us fail to perform the algorithms directly. We will take the subsequence searching algorithm UCR-Suite as an example to introduce how to support some basic operations under RpDelta. Moreover, we will also utilize RpDelta's properties to accelerate some circumstances.

4.1 Sequential Reading

Sequential reading (reading the data points one by one) is the most basic operation on time series, and we can find it in many prevalent time series algorithms. Most of the algorithms read the data points of a time series one by one for calculations, even if in a batch process(data will be put into the buffer at first). On this occasion, we need to implement it on RpDelta first for further implementation.

A simple and intuitive approach is to restore the repaired time series according to the raw series and the corresponding delta file and then read it sequentially. Nevertheless, we will encounter two problems using this method. First, to restore, we need to put the entire raw series into the memory, which is a large burden. Second, no matter what data structure we use, the time complexity to restore a time series with length n and m operations in its delta file will be $O(mn)$, since insertion and deletion need $O(n)$ time to complete. The restoration process itself will likely take longer than the time series algorithms. Under this circumstance, we need to design a more efficient and clever way to read sequentially in real time on RpDelta. Previously, we read the raw series from start to end directly, so our approach reads both the raw series and the delta file only once.

Problem Simplification. Here, we refer to where we can get the data as the **data source**. In traditional complete storage, the data source is the raw time series, and it is unique, so it is easy to conduct each read. While under RpDelta, all the data points in repaired time series can be obtained from either the raw series or the corresponding delta file, which means that we have two data sources. So considering both data sources at the same time is necessary. In this way, we can reduce the sequential reading problem to the strategy of deciding the data source of the next read according to the current reading status.

FSM (Finite State Machine) Method. We use FSM to solve the problem of data source choice. FSM is a computational model with a finite number of states, which can transfer from one to another in response to some inputs.

Here we use ST_Raw, ST_Ins, ST_Del, and ST_Rep to stand for the four different reading states, and we use ST_Cmp to stand for a checking state.

While dealing with a repaired time series, we first check the position of the following repair operation in ST_Cmp. Before arriving at this position, the repaired time series is the same as the raw. In this way, we can choose the raw series as the data source in ST_Raw and then return to the ST_Cmp state for the next check. While encountering such a position, we move to different states according to their corresponding operation types and return to ST_Cmp when the operation is over after several cycles. Table 1 shows the change between different states.

Table 1. Transfer between different states

Now	ST_Cmp		$ST\_Ins/Del/Rep$		ST_Raw
Input	(Pos) matched	not matched	End of read	Not end of read	End of read
Next	$ST\_Ins/Del/Rep$	ST_Raw	ST_Cmp	Remain unchanged	ST_Cmp

In the insertion state, the delta file is the data source, and we will directly return to ST_Cmp when finished. In the deletion state, we take raw series as the data source and skip a specific length of raw series before reading the point we want. Finally, in the replace state, the delta file is the data source, and we also need to skip the corresponding subsequence replaced in the raw series. This method allows us to read the repaired time series on RpDelta sequentially, almost without extra time or space complexity. Figure 3 exemplifies the reading process.

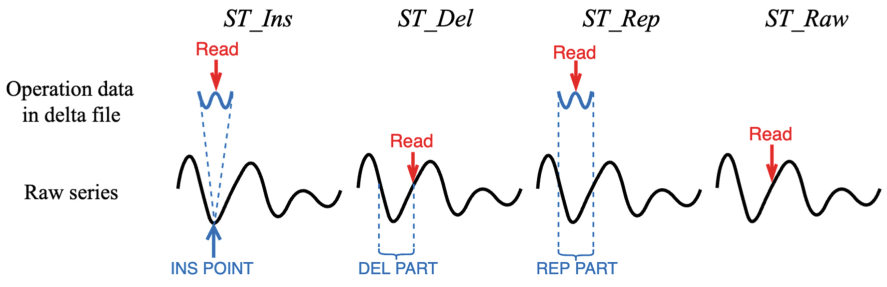

Fig. 3. Reading process in different reading states.

With the help of the above state machine, the space complexity can reach $O(1)$, and we only need to read the raw series and Delta File once to obtain the repaired time series sequentially on RpDelta.

4.2 Fundamental Subsequence Search

Here we take the UCR-Suite algorithm, a prevalent subsequence searching algorithm, as an example to illustrate how to implement such kind of operations on RpDelta.

The time series addressed by UCR-Suite are stored as .txt format files in the file system. The algorithm uses file pointers to read them sequentially, reflecting in the code as *fscanf*. Under the RpDelta strategy, we only need to replace it with a reading interface implemented according to the FSM method, and the rest of the algorithm remains unchanged.

The introductions of delta files and FSM reading strategy will bring about extra time overhead in the read-in of the UCR-Suite. However, it is small enough to be ignored, especially when the cost of the calculations is high. Our experiments displayed in Sect. 5.3 prove this point in detail.

Through the example of UCR-Suite, we can see that for all operations similar to it, in which we need to read the data points in time series one by one, the previous reading method(e.g., file pointer or others under different storage form) can be modified to the interface based on FSM method without any pain. This also proves the practical feasibility of RpDelta.

4.3 Subsequence Search on Multiple Repaired Time Series

Multi-versioning time series data is used to provide robustness for downstream tasks. To avoid potential errors caused by a single cleaning algorithm, we provide multi-versioning results or select the most reliable version for a given task. In this process, comparison and analysis between different versions of time series will bring unnecessary I/Os and repetitive calculations. For example, in UCR-Suite, we will calculate the distance between the query series and a repaired time series subsequence to find the best match. We are likely to read and take the same subsequence to calculate many times because this part does not need to be repaired and thus appears in many repaired time series, which causes much unnecessary extra time overhead.

Given that our RpDelta solves the repetitive storage by recording duplicates in one raw series and storing the exclusive parts in corresponding delta files, we will use this property to eliminate the repetitive calculations. Here we still take UCR-Suite as an example to illustrate how to optimize such occasion. Our optimization includes three steps.

Parallel Read Processing. We use a parallel approach to read multiple series at the same time. We should note that this approach is parallel abstractly and does not mean multiple cores or threads here(though it can be realized in such ways). In the past, we read in a data point, calculated the metrics, compared the distance with the best-so-far, and then read in the following data point and repeated this process. Now we read in an array directly, which consists of many data points from different time series. After that, we take the same procedure above on each of them. Our core idea of optimization is to reuse the calculations. If we deal with the time series one by one in this situation, we need to pay a lot for storing multiplexing value. It is because we must record the calculations of all the possible repetitive parts during the first repaired time series process. When reading them parallel, we just need to store them temporarily because they will soon be consumed by the others and can be thrown.

Moreover, we only need to access the raw series on the disk once in parallel reading, lowering the I/O cost. This is because we will put a data point into the buffer in the memory when first met, and while it has been used by all the sequential readings of the repaired time series(FSM method), it will be removed from the buffer. In this way, we can also lower time overhead since there is no need to access the disk for the raw series frequently.

Repetitive Subsequence Judgment. Moreover, we must decide on which circumstance we should reuse the calculations. As mentioned above, to the identical subsequences appearing many times, we just need to calculate once and store the value for later use. Therefore, one of the most important things is to propose a strategy that allows us to judge repetitive subsequences in different repaired time series. Here, we propose a coarse-grained method to determine them: if a subsequence has not been repaired by any data-cleaning algorithm, it is our target because it will appear in all of the repaired time series. In this method, we deliberately abandon many possible repetitive subsequences, such as those that appear $l-1$ times in the total l repaired time series. This makes the optimization more manageable and less costly in terms of implementation, maintenance and judgment. On the other hand, if we insist on finding all of the repetitive subsequences, chances are that its time cost will outweigh what we save on the calculation reuse.

In the concrete implementation, we scan all the delta files and mark all the operation positions during preprocessing. After this, we can get an array B in which $B[i]$ stands for the total number of operations on raw series $T_{i-m+1,m}$(m is the query length). Here we should notice that the influence of an operation is in the range of its length, not the single start data point. By examining $B[i] = 0$ or not, we can quickly learn whether a subsequence is our target during runtime with the time cost of $O(1)$.

Algorithm 1: Optimizations in UCR-Suite under DTW metric.

```
Input: $n = |T^k|$, $m = |Q|$, $q$ is the series number
Output: The best-match subsequence in each time series
for $p \leftarrow 1$ to $q$ do
    for $i \leftarrow 1$ to $n - m + 1$ do
        if $T^p_{i,m}$ belongs to repetitive subsequence then
            if meet $T^p_{i,m}$ first time then
                Calculate $lb\_kim$ and put into cache(may early abandon);
            else
                Get $lb\_kim$ from cache;
                if $lb\_kim < bsf^p$ and is not completely calculated then
                    Calculate $lb\_kim$ more exactly and refresh it in cache;
                end
            end
        else
            Calculate $lb\_kim$;
        end
        if $lb\_kim < bsf^p$ then
            Continue to calculate $lb\_k$, $lb\_k2$, $dtw$ in order like above;
        else
            Early abandon;
        end
    end
end
```

Repetitive Calculation Elimination. In UCR-Suite under DTW metric, the main computations are *lb_kim* lower bound generated by the LB_KimFL algorithm, *lb_k* lower bound generated by the LB_KeoghEQ algorithm, *lb_k*2 lower bound generated by the LB_KeoghEC algorithm and the DTW itself. All the above lower bound are served for early abandoning in UCR-Suite. We should point out that what we get in the calculation may be different from the exact value of them because of early abandoning. When first meeting a repetitive subsequence, we will manage it as usual and save the main computations in a buffer. Next time we process it again, all the saved computations will be reused. Suppose they are enough for early abandoning or distance calculation. In that case, we can eliminate repetition and adopt them; if not, we will calculate again for more exact values and update them in a buffer. Algorithm 1 shows part of this procedure related to *lb_kim*, and other computations are similar, so we omit them here. In this way, most calculations of repetitive subsequences can be combined into nearly one or, at most several times, thus significantly improving the subsequence-searching efficiency on multi-versioning time series.

5 Experiments

5.1 Experimental Setup

Datasets. We conduct extensive experiments on the latest UCR Time Series Classification Archive Dataset [3], which contains plenty of real-world datasets, such as ECG and Electric Devices data.

Baselines. Our baseline stores every repaired time series entirely on the disk for the storage experiment. In addition, to the operation experiment, since our optimization is introduced with the example of the UCR-Suite algorithm, here we use the UCR-Suite algorithm under ED and DTW respectively as our baselines to show our optimization. We get the implementation of UCR-Suite from its official website. Moreover, we identify that UCR-Suite uses the idea of early abandoning to accelerate the matching process, which makes our optimization of repetitive calculation less influential, but this is not always the case. Therefore, to make the results of the experiments more apparent to us, we modify UCR-Suite under the ED metric by removing the early abandoning strategy and use it as a baseline as well.

Our experiments use **ED** and **DTW** to stand for the raw UCR-Suite algorithm under the corresponding distance metric. $\mathbf{ED^-}$ stands for UCR-Suite under ED without early abandoning. $\mathbf{ED_p}$ uses parallel read processing under ED, and **ED-M** uses repetitive subsequences elimination.

The reason why we do not choose the latest subsequence searching algorithms, such as KV-Match [12] or ULISSE [5], as the baselines is that what we have done is not a searching algorithm but an idea of optimization under RpDelta. Meanwhile, such algorithms need many times the space we require because they use indexes to accelerate.

Implementation Details. We use a 10^7 long time series with the data type of double as the raw time series. This time series is concatenated by the time series in UCR Archive. We obtain query series of the desired length from the subsequence of the raw time series. In terms of repaired time series, we generate them artificially under the guidance of the Pareto principle. All of our data-cleaning strategies randomly put 80% of the repairs on 20% of the raw series and 20% of the repairs on 80% of the raw series. Replacement operations account for 60%, while insertion and deletion account for 20% respectively. The average length of each repair operation is 10, and the value of data points in operation is between $[\mu - \sigma, \mu + \sigma]$, where μ is the mean and σ is the standard deviation of the raw time series.

In the storage experiment, all the time series are stored as .txt files in the file system. For convenience in observing, we directly use the space of the raw time series to approximate the space of the repaired time series, which is reasonable in terms of mathematical expectations and will not influence our conclusions.

In the operation experiment, we have 3 key parameters: series number, repair rate, and query length. Here the series number reflects the number of data-cleaning strategies used to achieve multi-versioning time series. Each data-cleaning strategy corresponds to a version. We set the standard value of the series number to 6, the repair rate to 4%, and the query length to 128. We take the standard value if we do not specify the parameters' values. In the different experiments, we will adjust the value of the corresponding parameter. The Sakoe-Chiba Band [7] used on DTW calculations is 0.05(ratio to query series).

Testbed. Our algorithms are implemented in C++ and compiled by g++ 7.5.0 on Ubuntu 18.04 system with a 4.15.0-189-generic Linux kernel. We conduct experiments on an Intel(R) Xeon(R) Silver 4208 CPU @ 2.10 GHz machine with 64 GB RAM.

5.2 Storage Performance

We compare the storage use of RpDelta with the complete storage baseline. Table 2 shows RpDelta's storage performance. We find out from Table 2 that the storage use ascends as the repair rate and series number increases, but is much slower than the baseline, with less than 20% of the baseline at 8 repaired time series.

Table 2. Storage use of multi-versioning repaired time series under different repair rates and series numbers and the value of θ to each repair rate.

Rate	Number					
	1	2	4	6	8	θ
Baseline	182.02M	273.03M	455.04M	637.06M	819.08M	1
1%	91.92M	92.83M	94.67M	96.48M	98.30M	0.01001
2%	92.84M	94.67M	98.30M	101.95M	105.61M	0.02006
4%	94.66M	98.30M	105.56M	112.86M	120.12M	0.03999
8%	98.32M	105.60M	120.18M	134.77M	149.29M	0.08005

Experiments show that as the number of time series increases, the ratio of space to the baseline will decrease, but it will not converge to 0. Here we assume that the size of the delta file under a specific repair rate is a fixed value γ, and the size of a repaired time series or the raw one is a fixed value λ. The total number of repaired time series is n. We find that the ratio will converge to the value θ:

$$\theta = \lim_{n \to +\infty} \frac{n \cdot \gamma + \lambda}{(n+1)\lambda} = \frac{\gamma}{\lambda} \tag{3}$$

The last column of the Table 2 shows our experiments' value of θ at different repair rates. It illustrates the limits of RpDelta's space-saving capabilities.

This experiment demonstrates that in the situation of multi-versioning time series, RpDelta can effectively lower the space cost to a low level and solve the problem of unnecessary disk storage occupation.

5.3 Operation and Optimization Performance

We conduct extensive experiments to demonstrate the effectiveness of our optimization under different circumstances. Experiments show that we usually have a 60%–100% speed improvement under the ED metric, and under the DTW metric, a 30%–50% one. Moreover, they also display the good scalability of our method. The specific experiments are as follows.

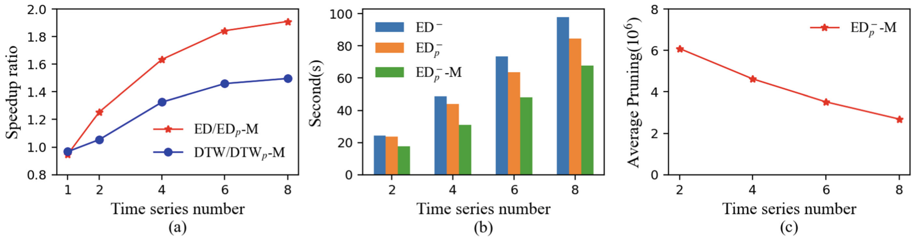

Fig. 4. (a) The speedup ratio of post-optimized algorithm to the baseline under different series numbers. (b) The running time of $\mathbf{ED^-}$, $\mathbf{ED_p^-}$, $\mathbf{ED_p^-}$**-M** under different series numbers. (c) The average number of eliminated repetitive subsequences in a repaired time series under different series numbers (divide by series number - 1 when calculating).

Effects of Series Number. In this experiment, we explore the impact of the series number on our optimization. Figure 4(a) shows the speedup ratio under different series numbers. When the series number is 1, we only apply the FSM method described in Sect. 4.1. At this point, the sequential reading time under the RpDelta strategy is almost the same as that of the baseline. Only maintaining the state machine to select the data source makes it slightly slower, but it does not matter.

As the series number increases, the speedup ratio gradually ascends, but the growth rate becomes smaller. We can get the reason from Fig. 4(b) Experiments on UCR-Suite without early abandoning show that despite the more I/O time saved by parallel read processing in the situation of more series, the effect of repetitive calculation elimination also descends. This is because the random distribution of the repair part leads to a decreasing number of repetitive subsequences per series, as Fig. 4(c) exhibits. These two factors together give rise to the results in Fig. 4(a).

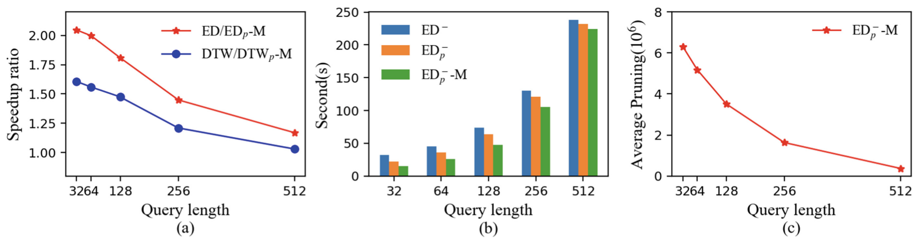

Fig. 5. (a) The speedup ratio of post-optimized algorithm to the baseline under different query length. (b) The running time of $\mathbf{ED^-}$, $\mathbf{ED_p^-}$, $\mathbf{ED_p^-}$**-M** under different query length. (c) The average number of eliminated repetitive subsequences in a repaired time series under different query length (divide by series number - 1 when calculating).

Effects of Query Length. In this experiment, we explore the impact of the query length on our optimization. Figure 5(a) shows the speedup ratio under different query lengths. The speedup ratio remains at a high level of more than 1.8 under ED and 1.5 under DTW when the query length is less than 128 and gradually descends to a comparatively low level (nearly 1) as the query length increases. This is because it is more difficult to find such long repetitive subsequences on this occasion, as Fig. 5(c) exhibits. Meanwhile, the increase in query length leads to a longer running time, thus making the optimization of parallel read processing inconspicuous, which is displayed in Fig. 5(b).

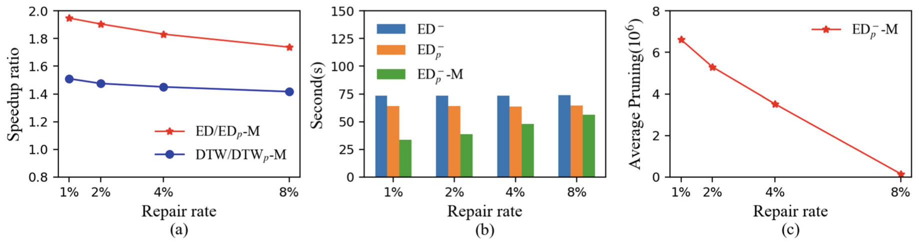

Fig. 6. (a) The speedup ratio of post-optimized algorithm to the baseline under different repair rates. (b) The running time of $\mathbf{ED^-}$, $\mathbf{ED_p^-}$, $\mathbf{ED_p^-}$**-M** under different repair rates. (c) The average number of eliminated repetitive subsequences in a repaired time series under different repair rates (divide by series number - 1 when calculating).

Effects of Repair Rate. In this experiment, we explore the impact of the repair rate on our optimization. Figure 6(a) shows the speedup ratio under different repair rates. The speedup ratio declines slowly when the repair rate ascends but still maintains a high level. Generally speaking, the higher the repair rate, the fewer the repetitive subsequences, but the speedup ratio is not seriously affected here. We can get the reason for this phenomenon from the experiment on $\mathbf{ED}^-$. Through Fig. 6(b), we can find that the running time of **ED** and $\mathbf{ED_p}$ is nearly unchanged as the repair rate ascends. Nevertheless, repetitive subsequence elimination has little effect at a high repair rate because few repetitive subsequences exist, as Fig. 6(c) exhibits. Parallel read processing plays a more significant role in speedup when the query length is 128, and the series number is 6, which explains Fig. 6(a).

To sum up, our optimization of operations on RpDelta owns a significant effect. The parallel read processing can provide stable improvement, and the repetitive subsequences elimination is considerably helpful, especially when the series number, query length, and repair rate are relatively small and still has sound effects when they increase.

6 Related Work

There are many studies on time series data cleaning and time series operations. Xi Wang pointed out that the current data-cleaning algorithms can be divided into three categories [11]. The first is a smoothing-based cleaning algorithm, such as the interpolation method used by S. Xu [13]. The second is a constraint-based cleaning algorithm. For example, Shaoxu Song proposed SCREEN [10], which uses the speed restrictions on value changes in a given interval. The third is a statistics-based cleaning algorithm, such as a method based on the HMM for RFID data cleaning put forward by Baba et al. [1]. However, all the above algorithms just focus on how to repair the time series more effectively and ignore the high storage costs of repaired time series.

In time series operations, there exist many kinds, like time series subsequence searching, time series clustering, and so on. Our works focus on time series subsequence searching. UCR-Suite is a state-of-the-art approach to solving the normalized subsequence searching problem, while KV-Match [12] and ULISSE [5] are the latest searching algorithms based on indexes. However, current operations are performed on the complete time series and fail to deal with the circumstances of many repetitive subsequences between sequential input time series.

7 Conclusion

In this paper, we propose a repaired time series storage strategy called RpDelta for efficient use of storage space. We first introduce the basic idea and the storage format of RpDelta. Then, we impose three constraints on the delta files in RpDelta to avoid possible ambiguities. In order to put RpDelta into practical use, we design a sequential reading strategy based on a finite state machine and take UCR-Suite as an example to illustrate how it works in a concrete operation of time series. Moreover, we also optimize the simultaneous-operation circumstance with the help of the RpDelta's properties. The experiments performed on

UCR 2018 archive show that we can save more than 80% of the storage space by using RpDelta when the series number is 8. Meanwhile, we can usually obtain a 60%–100% speed improvement under the ED metric and 30%–50% under the DTW metric, which shows the effectiveness and efficiency of our approach.

Acknowledgements. The authors would like to thank all the anonymous reviewers for their insightful comments and suggestions. This work was supported by the National Key R&D Program of China (No. 2021YFB3300502).

References

1. Baba, A.I., Jaeger, M., Lu, H., et al.: Learning-based cleansing for indoor RFID data. In: SIGMOD Conference, pp. 925–936. ACM (2016)
2. Benkő, Z., Bábel, T., Somogyvári, Z.: Model-free detection of unique events in time series. Sci. Rep. **12**(1), 227 (2022)
3. Dau, H.A., Keogh, E., Kamgar, K., et al.: The UCR time series classification archive (2018). https://www.cs.ucr.edu/~eamonn/time_series_data_2018/
4. Jeffery, S.R., Alonso, G., Franklin, M.J., Hong, W., Widom, J.: Declarative support for sensor data cleaning. In: Fishkin, K.P., Schiele, B., Nixon, P., Quigley, A. (eds.) Pervasive 2006. LNCS, vol. 3968, pp. 83–100. Springer, Heidelberg (2006). https://doi.org/10.1007/11748625_6
5. Linardi, M., Palpanas, T.: Scalable data series subsequence matching with ULISSE. VLDB J. **29**(6), 1449–1474 (2020). https://doi.org/10.1007/s00778-020-00619-4
6. Rakthanmanon, T., Campana, B.J.L., Mueen, A., et al.: Searching and mining trillions of time series subsequences under dynamic time warping. In: KDD, pp. 262–270. ACM (2012)
7. Sakoe, H., Chiba, S.: Dynamic programming algorithm optimization for spoken word recognition. IEEE Trans. Acoust. Speech Signal Process. **26**(1), 43–49 (1978)
8. Sathe, S., Papaioannou, T.G., Jeung, H., Aberer, K.: A survey of model-based sensor data acquisition and management. In: Aggarwal, C. (ed.) Managing and Mining Sensor Data, pp. 9–50. Springer, Boston (2013). https://doi.org/10.1007/978-1-4614-6309-2_2
9. Song, S., Zhang, A.: IoT data quality. In: CIKM, pp. 3517–3518. ACM (2020)
10. Song, S., Zhang, A., Wang, J., Yu, P.S.: SCREEN: stream data cleaning under speed constraints. In: SIGMOD Conference, pp. 827–841. ACM (2015)
11. Wang, X., Wang, C.: Time series data cleaning: a survey. IEEE Access **8**, 1866–1881 (2020)
12. Wu, J., Wang, P., Pan, N., et al.: KV-match: a subsequence matching approach supporting normalization and time warping. In: ICDE, pp. 866–877. IEEE (2019)
13. Xu, S., Lu, B., Baldea, M., et al.: Data cleaning in the process industries. Rev. Chem. Eng. **31**, 453–490 (2015)