A Feature and Similarity Based Grouping Algorithm for Efficient Time Series Storage

Yao Wang
School of Mechanical Engineering
Tongji University
Shanghai, China
1910425@tongji.edu.cn

Jiong Zhao
School of Mechanical Engineering
Tongji University
Shanghai, China
jiong.zhao@tongji.edu.cn

Jinhai Pan
School of Mechanical Engineering
Tongji University
Shanghai, China
2330380@tongji.edu.cn

Zhenkuo Kang
School of Mechanical Engineering
Tongji University
Shanghai, China
2210420@tongji.edu.cn

Qicai Zhou
School of Mechanical Engineering
Tongji University
Shanghai, China
qczhou@tongji.edu.cn

***Abstract*—For time series databases, single-column storage and single-column-group storage are common storage schemes. The former stores each time series independently but results in redundant storage of timestamps. The latter stores a batch of time series by sharing a time column but requires extra disk space for recording null values. This paper aims to design an algorithm that finds a compromise storage scheme between the two. We divide a batch of time series into multiple column-groups to minimize the overall space cost. We first model the time series grouping problem as an optimal set partitioning problem. Subsequently, we propose a two-phase algorithm to solve this problem. In Phase 1, we propose a similarity metric based on the time overlap degree between time series, and we employ a clustering algorithm to generate initial solutions. In Phase 2, based on the genetic algorithm (GA), we propose genetic operators derived from the features of time series to improve the quality of solutions obtained in Phase 1. Finally, we validate the effectiveness of the proposed algorithm through experiments.**

Keywords— time series, genetic algorithm, feature, clustering, operators, similarity

I. INTRODUCTION

Most time series databases adopt columnar storage, which can be further divided into single-column storage and column-group storage [1]-[2]. The single-column storage scheme stores the time column and the value column for each time series independently. Fig. 1(a) shows an example of 5 time series with single-column storage. All time columns T_1~T_5 and all value columns V_1~V_5 are stored in disk files. Obviously, this scheme leads to timestamp redundancy. Time series databases such as Prometheus [3] and InfluxDB [4] adopt the single-column storage.

To avoid storing duplicate timestamps, some time series databases like TDengine [5], IoTDB [6], and Heracles [7] adopt a column-group storage scheme. This scheme stores multiple time series by sharing a single time column. Taking IoTDB as an example, if the 5 time series are defined under the same *Device* [8], they will be stored as a single-column-group, as shown in Fig. 1(b). Only one time column T_C is stored in the file, which is the union of the timestamps of all time series. These 5 time series share one time column and are

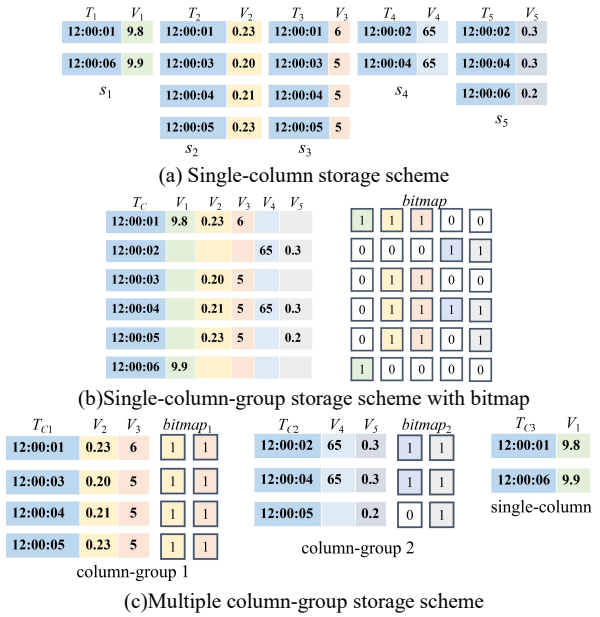

Fig. 1. Illustrations of time series and storage schemes

defined to belong to the same column-group. We use a single set C to represent a column-group. Elements in set C are the time series belonging to that group, so the column-group in Fig. 1(b) can be expressed as $C = \{s_1, s_2, s_3, s_4, s_5\}$.

If the time series can be well aligned on timestamps, the column-group storage will save a lot of space. However, especially in industrial environments, the sensors deployed on electromechanical equipment that generate time series data are independent of each other, and the data collection commands may also be unsynchronized [9]. These situations result in time series within a column-group failing to align at every timestamp. In the worst case, the collection frequencies of the two sensors are completely inconsistent, and the time series can hardly be aligned. To address this issue, disk files using column-group storage record the matching relationship between timestamps and values with some marker information, which occupies additional disk space. For example, IoTDB and TDengine use bitmaps. In Fig. 1(b), a "0" in the bitmap indicates that there is no corresponding value at that position.

So, is there a storage scheme between single-column and single-column-group storage that occupies less disk space? For s_1~s_5 in Fig. 1(a), there may be such a multiple column-groups storage scheme that requires less storage space. As shown in Fig. 1(c), this scheme constructs column-group 1 with s_2 and s_3 and column-group 2 with s_4 and s_5, while s_1 remains in a separate column-group. This scheme enables time series within a column-group to fully share timestamps, minimizes timestamp redundancy, and reduces the space cost by bitmaps.

Therefore, this paper aims to find an appropriate multiple column-groups storage scheme with minimal storage space cost for a batch of time series, especially those generated from industrial scenarios and electromechanical equipment. Fang et al. [10] first studied this problem and proposed a greedy algorithm based on maximum gain. However, their method relies on repeated calculations of storage space gain, lacks global search capability, and fails to utilize the inherent features of time columns. The grouping problem is similar to time series matching or similarity joins. Existing studies [11]-[12] on time series matching have adopted index structures to handle matching between series, but these methods are not well-adapted to the time columns of time series. Thus, alternative methods capable of finding appropriate column-group storage schemes are still worthy of further investigation.

To find a better scheme, we propose a Column-Grouping algorithm based on Similarity and Features using Clustering and GA (CGSF) from the perspective of time series features and similarity. The contributions of our work are as follows: (1) We model the time series grouping problem as an optimal set partitioning problem, which is a classic combinatorial optimization problem. (2) We propose the CGSF algorithm to solve the problem in two phases. In the first phase, we use a clustering method to pre-group time series and propose a similarity metric based on timestamp overlap. In the second phase, we propose a feature-based GA to improve the quality of the solution obtained in the first phase. (3) We implement the proposed algorithm in Apache IoTDB to conduct experiments. The results verify that CGSF can effectively reduce space cost and outperform existing algorithms.

II. Problem of Grouping Time series

A. Space Cost of Column Storage

In this section, we first present the space cost models for time series in columnar storage. Fang et al. [10] have proposed space cost models, which are adopted in our work. Given a set of n time series $S = \{s_1, s_2, s_3, ..., s_n\}$, each time series s_i consists of a time column T_i and a value column V_i, as illustrated in Fig. 1(a).

1) Single-column storage. For single-column storage, both the time column and value column of each time series are stored, and its space cost is denoted as $CostS(S)$:

$$CostS(S) = a \cdot \sum_{i=1}^{n} |T_i| + V_s \tag{1}$$

where $|T_i|$ represents the length of time series s_i. The parameter a denotes the space cost of a single timestamp. V_s denotes the total space cost of all value columns of S.

2) Single-column-group storage. As shown in Fig. 1(b), when adopting the single-column-group storage scheme, the n time series in set S are assigned to the same column-group C. All time series share a common time column. The space cost can be denoted as follows:

$$CostG(C) = a|T_C| + V_s + n \cdot |T_C| \cdot b \tag{2}$$

where the first term represents the space cost of the shared time column T_C. $|T_C|$ denotes the length of T_C. The parameters a and V_s have the same meaning as those in Equation (1). b is the space cost of storing a single bit in a bitmap.

B. Mathematical model of Time Series Grouping Problem

We aim to find a multiple column-groups storage scheme with the minimum space cost for a given set of time series. We define a column-group storage scheme as a family of sets $\mathcal{C}$, which is described as follows:

Definition 1. Column-Group Storage Scheme. Let $S = \{s_1, s_2, s_3, ..., s_n\}$ be a set of n time series. A column-group storage scheme is a partition of set S, denoted as $\mathcal{C} = \{C_1, C_2, C_3, ..., C_k\}$. Each valid partition of S corresponds to a distinct storage scheme. Each partition subset C_i is a set of time series with the following rules:

If $|C_i|>1$, the time series within C_i belong to the same column-group and are stored using the single-column storage.

If $|C_i|=1$, C_i contains only one time series, which is stored stored using the single-column-group storage.

Based on the aforementioned Definition 1, we model the time series grouping problem as follows:

Problem 1. Time Series Grouping Problem. Given a set of time series $S = \{s_1, s_2, s_3, ..., s_n\}$, find an optimal partition that partitions set S into m subsets $C_1, C_2, ..., C_m$, such that the objective function $f(C_1, C_2, ...C_m)$ is minimized. The objective function is:

$$\min f(C_1,...C_m) = \sum_{i=0}^{m} (1_{\{|C_i|=1\}} \cdot CostS(C_i) + 1_{\{|C_i|>1\}} \cdot CostG(C_i)) \tag{3}$$

Problem 1 also needs to meet the following constraints:

(1) $\forall i$, $C_i \neq \emptyset$, and $C_i \cap C_j = \emptyset$ $(i \neq j)$; (2) $\bigcup_{i=1}^{m} C_i = S$.

The objective is to find an optimal partition that divides all time series into m subsets. Under this partition scheme, the total space cost of storing all time series is minimized. The constraints require that the set partition must satisfy non-emptiness, coverage, and mutual exclusivity. In Equation (3), $1_{\{\cdot\}}$ is an indicator function that takes the value of 1 if the specified condition is satisfied, and 0 otherwise. Formally, this function is defined as follows:

$$1_{\{|G_i|=1\}} = \begin{cases} 1 & if |C_i| = 1 \\ 0 & otherwise \end{cases}, \quad 1_{\{|G_i|>1\}} = \begin{cases} 1 & if |C_i| > 1 \\ 0 & otherwise \end{cases} \tag{4}$$

where C_i is a subset of the set S. If the number of series in C_i is 1, according to Definition 1, we use Equation (1) to calculate the space cost of storing a single time series. If the number of series in C_i is greater than 1, all time series in C_i are stored as a

single-column-group, and the space cost of this column-group is calculated using Equation (2).

To solve Problem 1, we propose the CGSF algorithm. We decompose Problem 1 into two subproblems and solve this problem in two phases. (1) **Phase 1** (Pre-grouping): We use a clustering method with a dynamic similarity metric to pre-group time series and obtain an initial column-group storage scheme. (2) **Phase 2** (Optimization): Building upon the initial scheme, we propose the feature-based GA to improve the quality of the initial solution obtained from Phase 1.

III. Phase 1: The Pre-grouping Algorithm Based on the Similarity of Time Series

In fact, in the initial stage of our study, we attempted to use genetic algorithms (GA) [13] directly to solve Problem 1. When using GA, it is necessary to initialize some solutions. However, random initialization will cause the algorithm to fail to solve Problem 1. We analyze the reasons for algorithm failure as follows: time series with different lengths are often grouped together, while time series with high similarity cannot be initialized into the same group. The randomly initialized solutions are consistently poor, which further leads to the issue that GA always outputs poor schemes and the selection operators always choose inferior parent individuals. Therefore, Phase 1 serves to generate high-quality initial solutions.

Combining Equations (1) and (2), we can see that since the space cost of V_s is fixed, the length of the time column is the direct factor affecting the space cost. The total length of the time column in a column-group depended on the union of the time columns of each time series. Thus, the total length is related to the number of duplicate timestamps (referred to as the overlap degree) between time series. A higher overlap degree results in a shorter time column length, which in turn leads to lower space cost. For stably generated time series, particularly those originating from sensors on industrial equipment, a relatively high overlap degree implies similarity between time columns.

Therefore, in Phase 1, we cluster time series with high similarity into the same column-group. Meanwhile, to accurately measure the similarity, we also design a dynamical similarity metric based on the overlap degree of time columns.

A. Pre-Grouping Time Series Based on Clustering

If the time columns of two time series are highly similar, we can intuitively conclude that they can share more timestamps. DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [14] is a density-based clustering algorithm that forms clusters where samples within the same cluster exhibit high similarity, while samples from different clusters exhibit low similarity.

In Phase 1, inspired by DBSCAN, we propose a clustering algorithm for pre-grouping time series based on the similarity of time columns. The input to Algorithm 1 is the set of all candidate time series. The output is a family of sets $\mathcal{C}$, which represents a storage scheme as specified in Definition 1. Since the time columns are the only factor that affects the grouping results, we cluster the input time series based on the similarity of time columns.

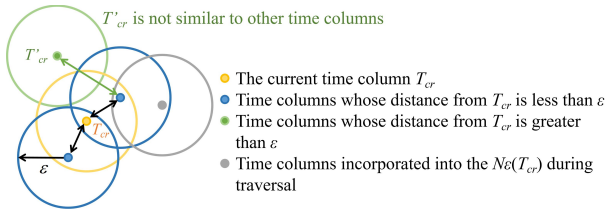

Fig. 2. Pre-grouping time series in Phase 1

We borrow several key concepts from DBSCAN [14]. We treat the time columns of the input time series as samples for the clustering process. The neighborhood radius ε, shown by the radius of the circle in Fig. 2, acts as the similarity threshold. Given a sample T_{cr}, other samples with similarity meeting the ε threshold to T_{cr} are collected into a set, denoted as $N\varepsilon(T_{cr})$. If the number of samples in $N\varepsilon(T_{cr})$ exceeds the predefined parameter *MinPts*, T_{cr} is identified as a core point. A core point forms the basis of a cluster. A cluster is incrementally constructed by including other core points or border points in its neighborhood.

Initially, Algorithm 1 derives time columns from each time series and extracts features from each derived time column. These features are used to compute similarity in Phase 1 and will be applied in Phase 2 of our CGSF algorithm. Additionally, Algorithm 1 initializes a hash map variable, denoted as *TCluster*, to record the cluster membership of each time series, where the keys correspond to cluster IDs and the values store the time series belonging to the corresponding clusters. Concurrently, an integer variable *CNum* is initialized to track the current cluster ID being processed (refer to Lines 1-3 of Algorithm 1). Algorithm 1 employs a breadth-first traversal (BFS) strategy, with the specific steps detailed as follows:

1) Traverse all time columns. For any time column T_i, if it has not been assigned to a cluster, add it to *AuxiliaryQueue*. This queue records the neighbors of core points during the iteration (refer to Lines 4-6 of Algorithm 1).

2) Traverse each time column in *AuxiliaryQueue*. Retrieve a time column from the queue, denoted as T_{cr}. Subsequently, the *SearchSimilarSeries* function is employed to find time columns similar to T_{cr}. Simultaneously, all time columns with a distance to T_{cr} less than ε are saved into the set $N\varepsilon(T_{cr})$ (refer to Lines 7-12 of Algorithm 1).

3) If T_{cr} has no similar time columns, i.e., $|N\varepsilon(T_{cr})|=0$, the time series corresponding to T_{cr} is marked as an outlier. It will be stored according to the single-column storage scheme (refer to Lines 13-14). As shown in Fig. 2, the green point T'_{cr} has no similar samples and is thus identified as an outlier.

4) If T_{cr} has similar time columns but the quantity is insufficient, i.e., $|N\varepsilon(T_{cr})|<MinPts$, the time series within $N\varepsilon(T_{cr})$ are recorded into the cluster numbered *CNum*. Furthermore, we stop searching for other similar time columns (refer to Lines 15-17).

5) If T_{cr} has similar time columns and the quantity is sufficient, i.e., $|N\varepsilon(T_{cr})|\geq MinPts$, then T_{cr} is identified as a core point. Subsequently, both T_{cr} and all time columns within $N\varepsilon(T_{cr})$ are recorded into the cluster numbered *CNum*. As

shown in Fig. 2, if *MinPts* is 2, the yellow point T_{cr} is a typical core point with abundant similar samples. Meanwhile, the time columns within $N\varepsilon(T_{cr})$ (e.g., blue points in Fig. 2) are re-enqueued into the *AuxiliaryQueue* to further check whether their neighbors can be incorporated into the current cluster (refer to Lines 18–21).

6) Repeat the above process until all time columns are assigned to a certain cluster or identified as outliers.

7) Finally, we transformed the clustering result into a set family $\mathcal{C}$(as defined in Definition 1), which is then returned.

Algorithm 1. Pre-grouping time series based on clustering

Input: Time series set $S = \{s_1, s_2, s_3, ..., s_n\}$
Output: A family of sets $\mathcal{C}$
1. *TSet* ← Extract the time column of each time series in *S*;
2. *CNum* ← 1; //Initialize cluster index
3. *TCluster* ← Initialize empty *hash map*;
4. *AuxiliaryQueue*← Initialize empty *queue*;
5. **for** T_i **in** *TSet* **do**
6. Add T_i to the auxiliary queue if it belongs to no cluster;
7. **while** *AuxiliaryQueue* **is not** empty **do**
8. T_{cr} ← Get the head element of *AuxiliaryQueue*;
9. **if** T_{cr} belongs to any cluster **then**
10. **continue**;
11. **end if**
12. $N\varepsilon(T_{cr})$ ← ***SearchSimilarSeries***(T_{cr}, *TSet*);
13. **if** $|N\varepsilon(T_{cr})| == 0$ **then**
14. Label T_{cr} as an outlier within *TCluster*;
15. **else if** $|N\varepsilon(T_{cr})| < MinPts$ **then**
16. Assign T_{cr} to the cluster numbered *CNum* and record it in *TCluster*;
17. Assign the time series in $N\varepsilon(T_{cr})$ to the cluster numbered *CNum*;
18. **else**
19. Assign T_{cr} to the cluster numbered *CNum* and record it in *TCluster*;
20. Assign the time series in $N\varepsilon(T_{cr})$ to the cluster numbered *CNum*;
21. Add all sample within $N\varepsilon(T_{cr})$ to *AuxiliaryQueue*;
22. **end if**
23. **end while**
24. *CNum* ← *CNum*+ 1; //prepare for next cluster
25.**end for**
26.$\mathcal{C}$ ← Transform *TCluster* into a family of subsets;
27.**return** $\mathcal{C}$;

B. Similarity Metric Based on Overlap Degree

In the previous section, we addressed that the *SearchSimilarSeries* function is used to identify neighbors of a given time column. The function operates as follows: given a time series set $TSet = \{T_1, T_2, ...\}$, a target time column T_{cr}, a similarity metric function $D_t(T_x, T_y)$, and a similarity threshold ε, it searches for all time columns similar to T_{cr}. The result is denoted as $N\varepsilon(T_{cr}) = \{T_i \in TSet \mid D_t(T_i, T_{cr}) < \varepsilon\}$. $D_t(T_x, T_y)$ is the similarity metric function, which compute the similarity between two time series T_x and T_y.

To identify which time columns are similar to T_{cr}, we can introduce the concept of distance [15]. However, a key question arises: how to quantify the distance between two time columns? Pioneering studies suggest that the distance between two samples can be quantified by computing their *Lp*-norm distance. Consequently, we can extract features from each time column. Next, the distance between two time columns can be computed using these features, thereby providing a quantitative metric for their similarity. For a single time column, previous work [10] proposes that the following features can be extracted: 1) Interval. The interval feature refers to the difference between two consecutive timestamps in a time series. The average of all such time differences is used as the interval feature of the time series, denoted as δ. 2) Length. The length feature refers to the number of data points in a time series, denoted as *len*. 3) Starting time refers to the first timestamp of a time series, denoted as t^s.

Using these features, we can compute the L_2 distance (Euclidean distance) between two time columns as a measure of their similarity. However, the determination of the similarity threshold ε is inherently subjective. Since similarity metrics based on *Lp*-norm distance are independent of data semantics, these metrics fail to accurately capture the semantic similarity between time series, thereby exacerbating the difficulty in determining ε. In the context of the time series grouping problem, the more mutually overlapping timestamps two time series share, the higher their similarity degree. Consequently, they are considered more similar. Thus, we propose using the number of overlapping timestamps to measure similarity. We first present the definition of overlap degree of time series:

Definition 2. Overlap Degree of Time Series. Let T_1 and T_2 be the timestamp sets of two time series s_1 and s_2, respectively. The overlap degree between s_1 and s_2 is defined as the size of their intersection, i.e., $|T_1 \cap T_2|$, denoted as θ.

Instead of calculating the L_2 distance using extracted features, we propose quantifying similarity based on the overlap degree of two time series. Accordingly, we define a similarity threshold based on the overlap degree.

Definition 3. Similarity Threshold Based on Overlap Degree. Let T_1 and T_2 be the timestamp sets of two time series s_1 and s_2. The similarity threshold, denoted as ε, is the critical overlap degree that ensures the overlap between s_1 and s_2 is sufficient to achieve space savings. It is calculated as follows:

$$\varepsilon = \frac{2(|T_1| + |T_2|)b}{a + 2b} \tag{5}$$

where the parameters are defined as in Equation (1).

Equation (5) can be derived based on the Lemma of Column-Column Merging [10]. According to Equations (1) and (2), if storing two time series s_1 and s_2 in a single-column-group reduces the space cost, the following condition should be satisfied:

$$\begin{aligned} &CostS(s_1) + CostS(s_2) - CostG(C_{(s_1, s_2)}) > 0 \\ &\Rightarrow (|T_1| + |T_2|) \cdot a - [(|T_1| + |T_2| - D_\theta(s_1, s_2)) \cdot a + D_\theta(s_1, s_2) \cdot b \cdot 2] > 0 \end{aligned} \tag{6}$$

where $C_{(s_1, s_2)}$ denotes the column-group comprising s_1 and s_2. $D_\theta(s_1, s_2)$ is the similarity metric function based on the overlap degree of time columns.

Rearranging Equation (6), we obtain the minimum overlap degree [10] required for storing s_1 and s_2 as a single-column-group to achieve space savings:

$$D_\theta(s_1, s_2) \geq \varepsilon = \frac{2(|T_1| + |T_2|)b}{a + 2b} \tag{7}$$

where the parameters are defined as in Equation (1).

So, only when the overlap degree of s_1 and s_2 satisfies the condition (7) are they deemed to be similar. During the search

for time columns similar to T_{cr}, the threshold ε is not a static value; instead, it varies dynamically based on the input T_{cr} and the current candidate time column T_u. Therefore, ε serves as a dynamic threshold. However, the direct calculation of the timestamp overlap between two time series incurs non-negligible computational overhead. Fortunately, previous work [10] has provided an approximate calculation method for the overlap degree based on the Gaussian distribution probability density model.

In summary, Algorithm 2 outlines the process of constructing the result set of similar time series for the input time series T_{cr}. The search result is denoted as $N\varepsilon(T_{cr})$. The specific steps are as follows: (1) Traverse the input time columns set. Let the current time column under traversal be denoted as T_u. (2) Calculate the threshold ε between T_{cr} and T_u in accordance with Definition 3. (3) Compute the overlap degree θ between T_{cr} and T_u. We adopt the Gaussian distribution probability density model to perform an approximate calculation. (4) If the overlap degree θ exceeds the threshold ε, T_{cr} and T_u are deemed to be similar. The time column T_u is then added to $N\varepsilon(T_{cr})$ as a similar sample of T_{cr}.

Algorithm 2. Search for similar time columns

Input: Time column set *TSet* and target time column T_{cr}
Output: The result set $N\varepsilon(T_{cr})$

```
1. Nε(T_cr) ← ∅;
2. for T_u in TSet do
3.   if T_cr != T_u then
4.     ε ← (2·(|T_cr|+ |T_u|)b) / (a+2b);//Using Equation (7)
5.     θ ← Calculate the overlap degree between T_cr and T_u;
6.     if θ ≥ ε then
7.       Put T_u into the result set Nε(T_cr);
8.     end if
9.   end if
10.end for
11.return Nε(T_cr);
```

IV. Phase 2: Search for Better Solutions with Genetic Algorithm Based on Features

Phase 1 assigns each time series to a specific column-group based on similarity and effectively mitigates the issues caused by the random initialization in genetic algorithms (GA). The ultimate objective of Problem 1 is to find a storage scheme that minimizes the total space cost. Therefore, based on the output of Phase 1, Phase 2 further refines the storage scheme using genetic algorithms, with the objective of minimizing space cost. The problem addressed in Phase 2 is as follows:

Problem 2. Optimize Column-Group Storage Schemes. Given a set of time series $S = \{s_1, s_2, s_3, ..., s_n\}$ and a family of sets $\mathcal{C} = \{C_1, C_2, C_3, ..., C_K\}$ representing the column-group storage scheme for S, our task is to adjust the time series contained in each subset of $\mathcal{C}$ such that the objective function defined in Equation (3) is minimized.

Problem 2 can be viewed as a combinatorial optimization problem with an initial solution. Adjusting the column-group memberships of time series is analogous to the variation and exchange of genes in GA. However, using classical random genetic operators still struggles to generate high-quality individuals. Thus, based on GA, we proposed feature-and-similarity-based genetic operators to iteratively search for a near-optimal column-group scheme.

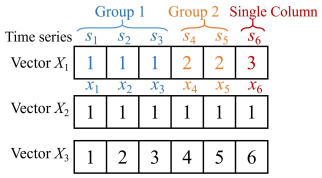

Fig. 3. Encoding the column-groups with integer vector

A. Encoding of Time Series Group

Integer vector encoding is adopted to encode a column-group storage scheme. In a vector, each index corresponds to a specific time series, while the value at that index denotes the column-group to which the time series is assigned. Suppose there are six time series (denoted as s_1 to s_6). Fig. 3 presents four examples. As shown by Vector X_1, time series s_1, s_2, and s_3 are assigned to Group 1. s_4 and s_5 are assigned to Group 2. s_6 is assigned to Group 3, indicating that s_6 is stored using the single-column storage scheme. Vector X_2 shows that all time series are assigned into a single group, which corresponds to the single-column-group storage scheme. Vector X_3 shows that each time series is assigned to a distinct group of its own, which corresponds to the single-column storage scheme.

B. Initialization, Fitness, and Selection Operator

Phase 1 effectively mitigates the problems of instability and algorithmic failure caused by random initialization in GA. When generating the initial population, at least half of the individuals are derived from the output of Phase 1. To maintain population diversity, the rest of the individuals are randomly initialized. After initialization, we calculate the fitness of each individual using Equation (3). The input to the fitness function is an encoded column-group scheme. And the output is the space cost of the corresponding scheme, which is taken as the fitness value. Tournament selection is adopted to select high-quality individuals from the current population. In the offspring generation process, the elitism strategy is also integrated.

C. Crossover and Mutation Operators Based on Time Series Features and Similarity

Crossover operators exchange specific segments in the encoded vectors to generate new offspring. Classic crossover operators, such as single-point crossover, are characterized by high randomness. Randomly exchanging segments of two parent individuals usually fails to generate superior offspring or requires a significant number of iterations to converge. Intuitively, time series with similar features are more likely to be assigned to the same column-group. Based on this intuition, we propose a crossover operator that leverages the features and pairwise similarity of time series.

Definition 4. Crossover Operator based on Time Series Features Similarity. Let $S = \{s_1, s_2, s_3, ..., s_n\}$ be a set of n time series. Let $X^{(1)} = x_1^{(1)}, x_2^{(1)}, ..., x_n^{(1)}$ and $X^{(2)} = x_1^{(2)}, x_2^{(2)}, ..., x_n^{(2)}$ be two parent individuals encoded using the integer vector scheme described in Section 4A. Given m features $F = \{f_1, f_2, f_3, ..., f_m\}$ for each time series, a feature-specific similarity metric function $\text{sim}_f(s_a, s_b)$ for each feature f, and a feature-specific similarity threshold θ_f, the crossover operator is defined as follows:

1) Randomly select an index i in $X^{(1)}$ to obtain the corresponding anchor time series s_i; then randomly select a feature subset $F_z \subseteq F$, with the constraint that $|F_z| \geq 1$.

2) From parent individual $X^{(1)}$, find all time series that are similar to s_i with respect to all features in F_z. Record the indices of these similar series to an index set, denoted as I_{sim}:

$$I_{sim} = \{j \in \{1, 2, ..., n\} \mid sim_f(s_i, s_j) < \theta_f, \forall f \in F_z\} \quad (8)$$

3) Exchange the gene segments corresponding to all indices in I_{sim} between the two parent individuals $X^{(1)}$ and $X^{(2)}$ to generate the new offspring individuals.

4) Return the optimal individual from all offspring and parent individuals.

As shown in Fig. 4, we present examples of the crossover operator. The feature set F adopts the 3 features extracted in Section 3B. The feature similarity threshold θ_f is defined as a relative difference of no more than 50% between the feature values of two time series, compared to the smaller feature value. In Fig. 4(a), we randomly select an anchor time series from $X^{(1)}$; for instance, s_6 is chosen. Subsequently, a subset of features from F is randomly selected; for example, only the length feature is chosen, i.e., $F_z = \{len\}$. Next, we identify all time series in $X^{(1)}$ that are similar to s_6 in terms of the length feature. Assuming that the relative length difference among s_4, s_5, and s_6 is no more than 50%, we deem that s_4, s_5, and s_6 to be similar in terms of the length feature. The indices of these similar series are denoted as $I_{sim} = \{4, 5, 6\}$, as indicated by the orange dashed circle in Fig. 4(a). Subsequently, the gene values $x_4^{(1)}, x_5^{(1)}$, and $x_6^{(1)}$ are exchanged with the corresponding gene values $x_4^{(2)}, x_5^{(2)}$, and $x_6^{(2)}$ at the corresponding positions in $X^{(2)}$, as indicated by the blue dashed circle in Fig.4(a). Thus, two new offspring individuals $newX^{(1)}$ and $newX^{(2)}$ are generated. $newX^{(2)}$ indicates that the number of column groups is decreased by 1 and that s_2, s_3, s_4, s_5, and s_6 are reassigned to Group 2. The case illustrated in Fig. 4(b) follows a similar process. We randomly select s_5 from $X^{(3)}$. We then randomly select a subset of features from F—for instance, the length and interval features. If s_5 and s_4 are similar with respect to these two features, the indices are denoted as $I_{sim} = \{4, 5\}$. Next, the gene values $x_4^{(3)}$ and $x_5^{(3)}$ in $X^{(3)}$ are exchanged with the gene values at the corresponding positions in $X^{(4)}$. Finally, we select the optimal individual from the newly generated offspring and the original parents and incorporate it into the next population.

The mutation operator operates on a single individual by modifying specific gene segments, which increases the diversity of solutions. Similar to traditional crossover operators, mutation operators based on random strategies typically fail to generate superior individuals for Problem 1. Building on the aforementioned intuition that time series with similar features are more likely to be grouped together, we propose a mutation operator that leverages the features and pairwise similarity of time series.

Definition 5. Mutation Operator based on the Similarity of Time Series Features. Let $S = \{s_1, s_2, s_3, ..., s_n\}$ be a set of n time series. Let $X = (x_1, x_2, ..., x_n)$ be a individual encoded using the integer vector scheme described in Section 4A. Given m features $F = \{f_1, f_2, f_3, ..., f_m\}$ for each time series, a

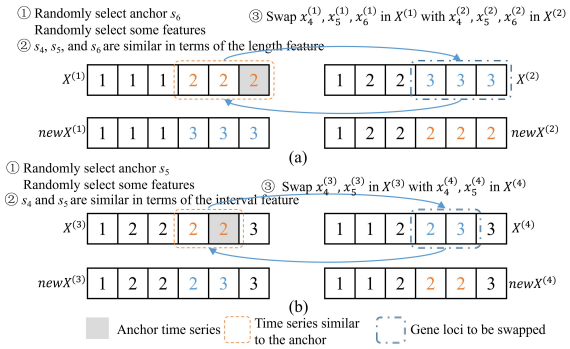

Fig. 4. Crossover operator based on time series features

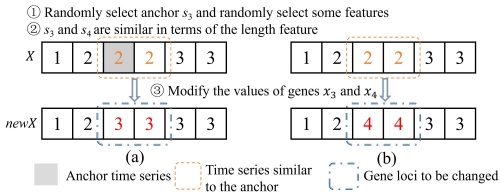

Fig. 5. Mutation operator based on time series features

feature-specific similarity metric function $sim_f(s_a, s_b)$ for each feature f, and a feature-specific similarity threshold θ_f, the mutation operator is defined as follows:

1) Randomly select an index i in X to obtain the anchor time series s_i; then randomly select a feature subset $F_z \subseteq F$.

2) From parent individual X, identify all time series that are similar to s_i with respect to all features in F_z. Record the indices of these similar time series to the index set I_{sim}:

$$I_{sim} = \{j \in \{1, 2, ..., n\} \mid sim_f(s_i, s_j) < \theta_f, \forall f \in F_z\} \quad (9)$$

3) Generate a new index, denoted as x', where x' is sampled from a uniform distribution over the set $\{1, 2, 3, ..., |\mathcal{C}|+1\}$. Subsequently, for all $j \in I_{sim}$, set the gene value of x_j in X to x', where $|\mathcal{C}|$ denotes the number of groups in the current column-group storage scheme. The upper limit of x' is set to $|\mathcal{C}|+1$, which enables the corresponding time series to be assigned to a newly created column-group.

As shown in Fig. 5(a), we randomly select s_3 as the anchor time series from the given individual X. Next, a subset of F is randomly selected. For instance, the feature length is selected. We then identify that s_3 and s_4 are similar with respect to the length feature. Subsequently, we reassign the column groups of s_3 and s_4 based on the uniform distribution. This results in s_3, s_4, s_5, and s_6 being assigned to the same Group 3. Another example, shown in Fig. 5(b), illustrates that s_3 and s_4 are reassigned to a newly created Group 4.

D. CGSF Algorithm Overview

To summarize, we formally define the time series grouping problem (Problem 1) and solve it in two phases. Algorithm 3 integrates and outlines the entire workflow: (1) Extract features from all input time series. (2) Pre-group the input time series using Algorithm 1 detailed in Phase 1. (3) Initialize the population based on the output of Phase 1. (4) Calculate the fitness of all individuals in the population. (5) Apply

tournament selection strategy to select parents. (6) Generate offspring individuals using the crossover operator as defined in Definition 3. (7) Update offspring individuals using the mutation operator as defined in Definition 4. (8) Check whether the termination condition is satisfied; if not, return to Step 4. (9) When Algorithm 3 terminates, it outputs the optimal individual X_b obtained during the iteration and decodes it into a column-group storage scheme.

Algorithm 3. Overview of CGSF

Input: Time series set $S = \{s_1, s_2, s_3, ..., s_n\}$
Output: A family of sets $\mathcal{C}$
1. $TSet \leftarrow$ Extract the time column of each time series in S;
2. $TFea \leftarrow$ Extract features from the time columns in $TSet$;
3. $\mathcal{C} \leftarrow$ Pre-grouping the time series in S using **Algorithm** 1;
4. $X_b \leftarrow X_{init} \leftarrow$ According to Section 4A, encode $\mathcal{C}$;
5. Initialize the population P according to Section 4B using X_{init};
6. **while** maximum iterations not satisfied **do**:
7. Calculate the fitness of all individuals in P according to Equation (3) find the optimal individual X_b in $P(t)$;
8. **while** the size of new population < N:
9. X_1 , $X_2 \leftarrow$ Select parents from P;
10. new$X_1 \leftarrow$Using the crossover operator in **Definition** 3;
11. new$X_2 \leftarrow$Using the mutation operator in **Definition** 4;
12. Put newX_1 and newX_2 into new population P';
13. **end while**
14. $P \leftarrow P'$;
15.**end while**
16.Decode the best X_b into a family of sets $\mathcal{C}$;
17. **return** $\mathcal{C}$;

V. Experiments

A. Method integration and experimental setup

In this section, we integrate CGSF into the time series database Apache IoTDB to conduct experiments. IoTDB is based on LSM-Tree [16], which supports both single-column storage and single-column-group storage. When new data is ingested into IoTDB, it is written into Memtable and waits to be flushed to a disk. CGSF is integrated into the flushing phase. Before the time series in the Memtable are flushed, CGSF outputs a grouping scheme for these time series. In industrial environments, sensors typically collect data at a stable frequency. Therefore, we use the time series of limited length in the static Memtable to run the CGSF algorithm once and obtain the column-groups scheme. To avoid repeating the calculation of the overlap degree, we also cache this metric for reuse during the execution. The server configuration in our experiment is as follows: Intel Core i5-2320 CPU, 32GB DDR3, and 1TB HDD. All code is implemented in Java 1.8 and is available on the GitHub repository [17]. Four datasets from industrial scenarios are used in our experiments, which are detailed in Table 1. All datasets contain the status data of industrial machines.

We compare multiple storage strategies and algorithms: (1) Automatic Column-Grouping for time series (ACG) [10], the state-of-the-art grouping algorithm in this field, which derives the storage scheme in a bottom-up manner by analyzing space gains. (2) Automatic Column-Grouping with Overlap Estimation algorithm (ACGOE) [10], which estimates the overlap degree and conducts column-grouping based on ACG. (3) Single-column storage (SC). (4) Single-column-group storage (SG) [2][6].

TABLE I. Data Used in Experiment

DataSet	Points	Rows	Sensors	Null rate
TBM	259,428	9,978	26	15.07%
Tunnel	3,491,919	120,411	29	31.20%
Vehicle	850,000	50,000	17	27.06%
Engine	1,120,000	70,000	16	23.20%

TABLE II. Comparison of Space Cost (Unit: KB)

DataSet	SC	SG	APGOE	ACG	CGSF
TBM	782.89	507.30	507.32	507.22	507.22
Tunnel	2302.87	2173.18	2172.86	2172.86	2158.69
Vehicle	1151.38	1252.08	1250.01	1250.01	1126.83
Engine	706.60	757.59	757.59	757.59	688.33

TABLE III. Comparison of Time Consumption (Unit: ms)

DataSet	APGOE	ACG	CGSF
TBM	34.2	25.8	23.6
Tunnel	188	197	64
Vehicle	13	17.6	14.2
Engine	17	21.3	14.6

TABLE IV. Space Cost With Different Metric (Unit: KB)

Metric	TBM	Tunnel	Vehicle	Engine
OD	757.50	1142.00	693.44	2209.73
ED	507.32	1250.01	688.326	2195.11

In our experiment, the threshold *MinPts* is set to 3. The number of iterations of Algorithm 3 is set to 20. The population size is set to 10. All configuration parameters of the experimental database are kept at their default values. Two metrics are adopted to evaluate different methods: (1) Space cost: the total disk space occupied by all data after being persisted to the database in accordance with the corresponding strategy or algorithm. (2) Computational time consumption: the total time required to calculate a column-group storage scheme for a batch of time series.

B. Space Cost and Time Consumption

Table 2 presents the results under different storage schemes. On TBM dataset, SC yields the highest space cost, while CGSF and ACG both yield the lowest space cost. On other datasets, CGSF achieves the lowest space cost. The experiments in this section indicate that the CGSF can effectively reduce the total space cost by data. Table 3 presents the computational time consumption. SG and CG are natively integrated into IoTDB, thus incurring no computational overhead. ACGOE and CGSF yield higher computational efficiency, whereas ACG incurs higher computational time consumption due to its need to calculate the overlap degree of time series accurately. On other datasets, considering space cost and computational time consumption, CGSF achieves superior comprehensive performance.

C. Performance Comparison Under Varying Data Volumes

To verify the robustness of the CGSF algorithm, we vary the volume of data written to the database. Figs. 6(a) and (b) present that CGSF performs better than other algorithms across different data volumes. Figs. 6(c) and (d) illustrate the time consumption under varying data volumes. CGSF and ACGOE yield the low grouping computation time, while ACG incurs a much longer computational time consumption.

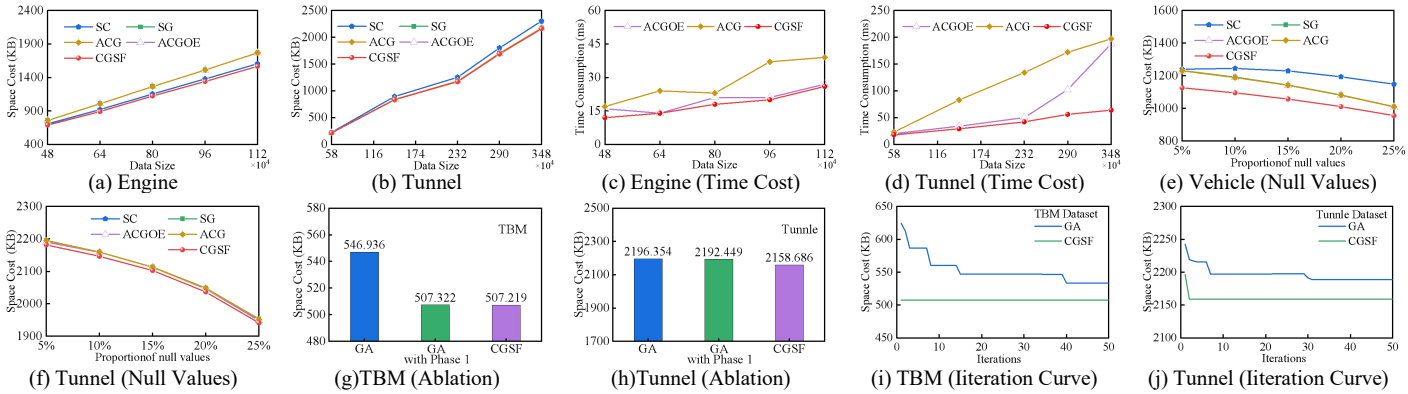

Fig. 6. Results under different experimental conditions

D. Comparison under Varying Proportion of Null Values

Increasing the proportion of null values of the datasets will further exacerbate the misalignment degree of time series, which increases the difficulty that all algorithms face to find better solutions. As shown in Figs. 6(e)(f), when the proportion of null values increases, CGSF can still find a scheme with the lowest space cost among all algorithms.

E. Comparison of Similarity Metrics

In Phase 1 of CGSF, we propose a similarity metric based on the overlap degree of time columns. In this section, we use the proposed overlap degree (OD) and Euclidean distance (ED) as similarity metrics respectively to group time series. Table 4 presents the results indicating that, except for the Vehicle dataset, the grouping results using the overlap degree-based similarity metric achieve superior performance to those derived from the Euclidean distance metric.

F. Ablation Experiment: the Impact of Initialization and Feature-based Operators on Algorithm Performance

In this section, we focus on CGSF itself and evaluate the impact of its two phases. Taking the TBM and Tunnel datasets as examples, Figs. 6(g)(h) present the results. The blue bar represents the basic GA. The green bar represents the method that first performs Phase 1 and then uses random operators. As observed from the results, compared with the random initialization, Phase 1 can generate superior solutions. The purple bar represents CGSF, which integrates both Phase 1 and Phase 2. Figs. 6(i)(j) indicate that the proposed genetic operators can further improve the quality of solutions and accelerate the convergence of the algorithms.

VI. Conclusion

We model the time series grouping problem as a set partitioning problem and propose the CGSF algorithm, which solves it through a two-phase framework. First, we introduce an overlap degree-based pre-grouping algorithm to obtain an initial partition scheme, enabling the GA to initiate its iterations from a high-quality initial solution. Next, we propose genetic operators based on time series features and their similarities to further refine the quality of the initial solution and accelerate the convergence rate of our algorithm. Finally, our experimental results demonstrate that the proposed CGSF algorithm achieves superior performance in terms of both solution quality and computational efficiency.

References

[1] X. Zhao, J. Qiao, X. Huang, et al., "Apache tsfile: An iot-native time series file format," Proc. VLDB Endow., vol. 17, no. 12, pp. 4064–4076, 2024.

[2] Z. Q. Wang and Z. L. Shao, "TimeUnion: An efficient architecture with unified data model for timeseries management systems on hybrid cloud storage," in Proc. 2022 Int. Conf. Manag. Data (SIGMOD), New York, NY, USA, 2022, pp. 1418–1432.

[3] Prometheus. [Online]. Available: https://prometheus.io/docs

[4] InfluxDB. [Online]. Available: https://www.influxdata.com

[5] TDengine. [Online]. Available: https://docs.taosdata.com

[6] C. Wang, J. Qiao, X. Huang, et al., "Apache IoTDB: A time series database for IoT applications," Proc. ACM Manag. Data, vol. 1, no. 2, pp. 1–27, 2023.

[7] Z. Wang, J. Xue, and Z. L. Shao, "Heracles: An efficient storage model and data flushing for performance monitoring timeseries," Proc. VLDB Endow., vol. 14, no. 6, pp. 1080–1092, 2021.

[8] Aligned Timeseries. [Online]. Available: https://iotdb.apache.org/zh/UserGuide/latest/Basic-Concept/Operate-Metadata_apache.html

[9] Y. Kang et al., "Separation or not: On handling out-of-order time-series data in leveled LSM-tree," in Proc. 2022 IEEE 38th Int. Conf. Data Eng. (ICDE), Kuala Lumpur, Malaysia, 2022, pp. 3340–3352.

[10] C. Fang, S. Song, H. Guan, et al., "Grouping time series for efficient columnar storage," Proc. ACM Manag. Data, New York, NY, USA, 2023, pp. 1–26.

[11] Y. Mei, S. Song, Y. Lee, et al., "Representing temporal attributes for schema matching," in Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discov. Data Min., CA, USA, 2020, pp. 709–719.

[12] F. Gao, S. Song, L. Chen, et al., "Efficient set-correlation operator inside databases," J. Comput. Sci. Technol., vol. 31, no. 4, pp. 683–701, 2016.

[13] J. H. Holland, "Genetic algorithms," Sci. Am., vol. 267, no. 1, pp. 66–73, 1992.

[14] M. Ester, H. P. Kriegel, J. Sander, et al., "A density-based algorithm for discovering clusters in large spatial databases with noise," in Proc. 2nd Int. Conf. Knowl. Discov. Data Min., Portland, OR, USA, 1996, pp. 226–231.

[15] Y. Liu, Y. Zhang, M. Zeng, et al., "A novel distance measure based on dynamic time warping to improve time series classification," Inf. Sci., vol. 656, pp. 119921, 2024.

[16] P. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (LSM-tree)," Acta Inform., vol. 33, no. 4, pp. 351–385, 1996.

[17] Data and code, GitHub Repository, [Online]. Available: https://github.com/wangyao2/iotdb-0-13-4-ColumnGroupingResearch