Grouping Time Series for Efficient Columnar Storage

CHENGUANG FANG, BNRist, Tsinghua University, China
SHAOXU SONG, BNRist, Tsinghua University, China
HAOQUAN GUAN, BNRist, Tsinghua University, China
XIANGDONG HUANG, BNRist, Tsinghua University, China
CHEN WANG, BNRist, Tsinghua University, China
JIANMIN WANG, BNRist, Tsinghua University, China

Columnar storage is now an industry standard design in most open-source or commercial time series database products, making them HTAP systems. The time column of a time series serves as the key for identifying the other value column, namely *single-column storage* scheme. When multiple time series share a similar set of timestamps, very likely in a module of multiple sensors, it is natural to group them together, i.e., one time column identifies multiple value columns in a *single-group storage* scheme. While multiple value columns sharing the same time column reduce the space cost of repeating timestamps, it may introduce extra space cost for recording null values. The reason is that time series may not be exactly aligned on each timestamp, owing to missing values, distinct data collection frequencies, unsynchronized clocks and so on. The *column-groups storage* scheme is thus to divide columns into multiple groups, within which the value columns share the same time column. Unfortunately, the problem of finding the optimal column groups for the minimum space cost is highly challenging, NP-hard according to our analysis. Thereby, we propose a heuristic algorithm for automatically grouping time series for efficient columnar storage. The column groups storage has been deployed in Apache IoTDB, an open-source time series database. The extensive performance analysis, over real-world data from our industrial partners, demonstrates that the proposed column groups achieve near optimal storage, more concise than the storage of single-column or single-group schemes. Interestingly, both the flushing and querying time costs of column groups are comparable to those of single-column or single-group, i.e., without incurring extra time cost.

CCS Concepts: • **Information systems** → **Stream management**.

Additional Key Words and Phrases: time series, grouping, columnar storage

ACM Reference Format:
Chenguang Fang, Shaoxu Song, Haoquan Guan, Xiangdong Huang, Chen Wang, and Jianmin Wang. 2023. Grouping Time Series for Efficient Columnar Storage. *Proc. ACM Manag. Data* 1, 1, Article 23 (May 2023), 26 pages. https://doi.org/10.1145/3588703

This work is supported in part by National Key Research and Development Plan (2021YFB3300500), the National Natural Science Foundation of China (62021002, 62072265, 62232005), Civil aircraft research project (MZJ3-2N21), Beijing National Research Center for Information Science and Technology (BNR2022RC01011), and Alibaba Group through Alibaba Innovative Research (AIR) Program. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.
Authors' addresses: Chenguang Fang, BNRist, Tsinghua University, Beijing, China, fcg19@mails.tsinghua.edu.cn; Shaoxu Song, BNRist, Tsinghua University, Beijing, China, sxsong@tsinghua.edu.cn; Haoquan Guan, BNRist, Tsinghua University, Beijing, China, ghq18@mails.tsinghua.edu.cn; Xiangdong Huang, BNRist, Tsinghua University, Beijing, China, huangxdong@tsinghua.edu.cn; Chen Wang, BNRist, Tsinghua University, Beijing, China, wang_chen@tsinghua.edu.cn; Jianmin Wang, BNRist, Tsinghua University, Beijing, China, jimwang@tsinghua.edu.cn.

2836-6573/2023/5-ART23
https://doi.org/10.1145/3588703

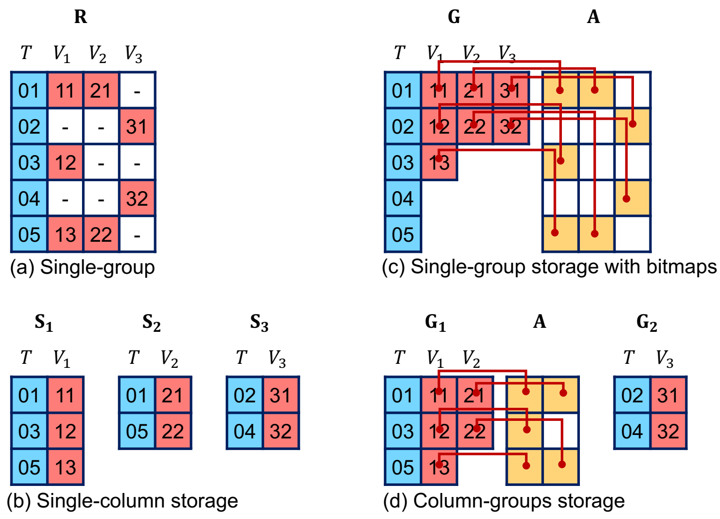

Fig. 1. Motivation example of various storage schemes with time column T, value columns V_i, and bitmap **A**, the red lines are the positions of the values marked by the bitmap.

1 INTRODUCTION

Most time series database management systems, open source or commercial, such as Apache IoTDB [3], InfluxDB [6], OpenTSDB [7], Prometheus [8], and TDengine [10], are designed with columnar storage. Such scheme enables highly efficient hybrid transaction/analytical processing (HTAP), where each time series is collected, compressed and queried individually. Even for multi-dimensional analysis, the time-ordered time series could be aligned efficiently with a merge sort. For example, Figure 1(a) illustrates a multi-dimensional time series, $\mathbf{R}(T, V_1, V_2, V_3)$, where T is the time column, and V_1, V_2, V_3 are the value columns, corresponding to three unary time series.

1.1 Motivation

To efficiently store the data, for each unary time series, the *single-column storage* scheme uses the time column as the key for identifying the other value column. Encoding and compression techniques [22, 26] are applied to both the time and value columns. For the example in Figure 1(a), the corresponding single-column scheme is $\mathrm{S}_1(T, V_1)$, $\mathrm{S}_2(T, V_2)$, $\mathrm{S}_3(T, V_3)$. Obviously, the repeated time column T incurs unnecessary space overhead, as shown in Figure 1(b).

Alternatively, the *single-group storage* scheme aligns the multiple time series together, for example, $\mathbf{R}(T, V_1, V_2, V_3)$ as shown in Figure 1(a), where the value columns V_1, V_2, V_3 share the same time column T. This design would be effective when multiple time series share a similar set of timestamps, very likely in a module of multiple sensors. However, one needs to specify this very carefully, as time series may not be exactly aligned on each timestamp, owing to various issues like missing values, distinct data collection frequencies, unsynchronized clocks and so on [31]. That is, there are many null values (denoted by $-$) as illustrated in Figure 1(a), which also occupy extra space in value columns compared to the single-column storage scheme. As presented in Figure 1(c), we use a bitmap **A** to record the positions of null values, which is not necessary for single-column storage. In other words, while multiple value columns sharing the same time column reduce the space cost of repeating timestamps, it may introduce extra space cost for recording null values.

In this sense, the *column-groups storage* scheme is thus to divide columns into multiple groups. In each group, the value columns share the same time column. For instance, the time series in Figure 1(a) could be split into two column groups $\mathbf{G}_1(T, V_1, V_2)$ and $\mathbf{G}_2(T, V_3)$. Since the data in each group are compactly aligned without many null values and the time column is shared, as illustrated in Figure 1(d), its space cost is more efficient than both the aforesaid single-column and single-group storage schemes.

The query, evaluated on such a representation of the database, indeed returns the exact results. The bitmap indicates how the value columns should be aligned with the time column in a group, while different groups are aligned by their time columns. For instance, in Figure 1(d), to query the entire time series, the bitmap $\mathbf{A}$ first indicates that the value `22` in the value column V_2 should be aligned to the row with time `05` in T. Next, the value columns V_1 and V_2 in group $\mathbf{G}_1$ are aligned with V_3 in $\mathbf{G}_2$ on the time column T, leading to the query result of multi-dimensional time series as illustrated in Figure 1(a).

1.2 Solution

Our solution is thus to find the optimal column groups that can minimize space cost. It is worth noting that the column grouping solution may also output single-column or single-group schemes as special column groups, depending on the data features.

Efficiently finding the optimal column groups for the minimum space cost, however, is highly challenging, mainly in two aspects. (i) *Deciding a group is difficult.* Given the various combinations of columns as possible groups, it is not surprising that the problem is generally hard, according to our analysis in Theorem 1 in Section 2.4. Thereby, we propose a heuristic algorithm for automatically grouping time series of efficient columnar storage in Section 3. (ii) *Evaluating a group is also costly.* Precisely evaluating the degree of two time series sharing the same timestamps needs to costly traverse all the points in the series. To this end, we propose to extract the timestamp features of time series, and evaluate their overlap on timestamps over the constant size features rather than the entire raw series in Section 4.

Our major contributions are summarized as follows.

(1) We formalize the column-grouping problem to determine the optimal storage strategy for given time series (Section 2), and analyze NP-hardness of the problem (Theorem 1).

(2) We devise column-grouping algorithm in a bottom-up grouping strategy, greedily merging column groups (Section 3). Efficient merging pruning strategies are developed based on the bounds of timestamp overlaps.

(3) We develop an approximate evaluation strategy to efficiently estimate the overlaps on timestamps of time series. It is based on the constant size features extracted from the timestamps, thus avoiding time-consuming traversals of all the points. (Section 4).

(4) We have deployed the column groups storage in Apache IoTDB, an open-source time series database [33] (Section 5). The source code of automatic grouping is available in the GitHub repository of Apache IoTDB [9]. Users may add a keyword `autoaligned` to declare the function in the SQL statement of creating time series. When the time series is flushed from memory for the first time, it automatically calls the function of column grouping.

(5) We conduct extensive experiments over real-world data from our industrial partners (Section 6). The results demonstrate that the proposed column groups achieve near optimal storage, more concise than the storage of single-column or single-group schemes. Interestingly, both the flushing and querying time costs of column groups are comparable to those of single-column or single-group, i.e., without incurring extra time cost.

Table 1 lists the frequently used notations. Section 7 discusses related work, and Section 8 identifies some future directions.

Table 1. Notations

Symbol	Description
S	a set of time series
S_i	the i-th time series of S
α	the space cost of a timestamp
β	the space cost of one bit in bitmap
V_{S}	the space cost of storing all the values of S
$\mathcal{G}$	a set of time series groups in column-groups storage
G	a time series group in $\mathcal{G}$ with column-groups storage
$\mathbf{A}$	a bitmap of size $n \times m$ recording the alignment of S
m	the number of distinct timestamps in time series
n	the number of time series
$P, \hat{P}$	exact/approximate overlap evaluation function

2 COLUMN-ORIENTED STORAGE

Let us first introduce the single-column and single-group storage of time series with a formal definition. Column-groups storage is then studied to combine the advantages of both storage strategies.

2.1 Single-Column Storage

Let $\mathrm{S} = \{S_1, S_2, \ldots, S_n\}$ be a set of n time series, where each $S_i \in \mathrm{S}$ has a time column T_i of timestamps and a value column V_i of data values, i.e., $S_i = (T_i, V_i)$. An example is illustrated in Figure 1(b).

To serialize the time series, single-column storage stores each time series separately, including a timestamp column and a value column. The space cost of the single-column storage is thus composed of serializing the series of timestamps and values.

Definition 1 (single-column storage). *Given a set of n time series S, the single-column storage of S has space cost*

$$cost_c(\mathrm{S}) = \sum_{i=1}^{n} \alpha m_{S_i} + V_{\mathrm{S}} \tag{1}$$

where m_{S_i} denotes the length of S_i, α denotes the space cost of storing a timestamp, V_{S} denotes the space cost of storing all the values of S.

Note that α is different with various compression strategies, such as delta-of-delta of timestamps, which only stores the maximum bit length of the delta-of-delta timestamps for each timestamp [28]. Such a weight could be observed from data. It is also notable that the value storage V_{S} is the same in various storage schemes studies in this paper, and could be ignored in comparison.

2.2 Single-Group Storage

To share the timestamp column among multiple time series, we can store n time series in one group, with one common time column, by aligning the timestamps of them.

Therefore, we first give the definition of the aligned time column, which identifies the timestamps from aligned time series in a common timestamp column, and shared by all time series in the group.

Definition 2 (aligned time column). *Given a set of time series* S*, the aligned time column* T_{S} *is the union of the time column of each time series* $S_i \in \mathrm{S}$*.*

$$T_{\mathrm{S}} = \bigcup_{S_i=(T_i,V_i)\in \mathrm{S}} T_i \tag{2}$$

The length of T_{S} is denoted by m_{S}, i.e., $m_{\mathrm{S}} = |T_{\mathrm{S}}|$. For simplicity, if the context is clear, we will abbreviate m_{S} as m.

For the set S of n time series, we use an alignment bitmap $\mathbf{A} = (a_{ij})_{m\times n}$ that records the alignment of the timestamps S (i.e., records the positions for null values), where m is the total aligned length of all the time series in S, and n is the time series number in S, $a_{ij} = 1$ denote the value of S_j has the corresponding value in the i-th aligned timestamp. Figure 1(c) illustrates an example for **A**, where **A** has the same shape with S if we align S by timestamps.

Based on the aligned time column, we then introduce the definition of the column group and the single-group storage. According to the form of the column group, it includes a common time column, a list of value columns and a bitmap recording the missing values. The single-group storage is thus defined over the column group, which serializes the data of a column group according to its form. We then discuss the space cost of the single-group storage.

Definition 3 (single-group storage). *A column group for a set of n time series* S *is to group the time series* $S_i \in \mathrm{S}$ *together, to share an aligned time column* T_{S} *and use a bitmap* **A** *to record the positions of null values, denoted by* $\mathbf{G} = (T_{\mathrm{S}}, V_1, \ldots, V_n, \mathbf{A})$*. The single-group storage of* S *is to serialize all the time series according to* **G***. The space cost of* **G** *is the sum of the space cost for each part,*

$$cost_g(\mathbf{G}) = (\alpha + n * \beta)m_{\mathrm{S}} + V_{\mathrm{S}} \tag{3}$$

where α *and* β *denote the space cost of storing a timestamp and a bit in the bitmap, respectively, and* V_{S} *denotes the space cost of storing all the values of* S*.*

Note that, the alignment of the timestamps might result in extra space cost for storing the bitmap to record missing values owing to the alignment of different time series. It is even worse when the time series are highly unaligned, i.e., having large differences in timestamps among the time series. In such a scenario, large disk space is wasted for recording the missing values in bitmap **A**.

2.3 Column-Groups Storage

As illustrated in Definitions 1 and 3, the space cost V_{S} for storing value columns are the same in the single-column and single-group storage schemes. Their difference in space cost is between the duplicated timestamp columns in single-column storage and the extra bitmap **A** in the single-group storage. When the time series could be densely aligned with highly similar timestamps, the single-group storage is preferred. In contrast, if the time series can barely be aligned with distinct timestamps, i.e., no much sharing, the single-column storage performs. In practice, some time series may share similar timestamps, while others not. Intuitively, we could divide the time series into multiple groups, within which timestamps are similar and shared.

Definition 4 (column-groups storage). *Given a set of time series groups* $\mathcal{G} = \{\mathbf{G}_1, \mathbf{G}_2, \cdots, \mathbf{G}_K\}$ *of size K, each with single-group storage, and a time series set* $\mathbf{R} = \{R_1, R_2, \cdots, R_{|\mathbf{R}|}\}$ *of the other time series with single-column storage, the column-groups storage serializes the time series in each* $\mathbf{G}_i$ *with single-group storage, and each time series* $R_i \in \mathbf{R}$ *with single-column storage separately according to the grouping scheme, Formally, the total space cost is:*

$$cost_{cg}(\mathcal{G}, \mathbf{R}) = cost_c(\mathbf{R}) + \sum_{\mathbf{G}_i \in \mathcal{G}} cost_g(\mathbf{G}_i) \tag{4}$$

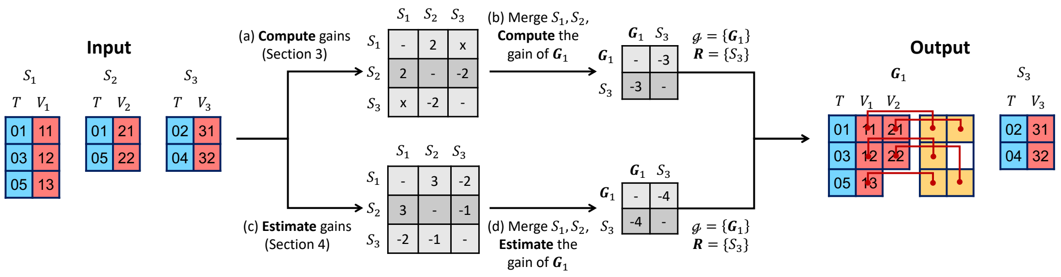
Input
(a) Compute gains (Section 3)
(b) Merge S_1, S_2, Compute the gain of G_1
(c) Estimate gains (Section 4)
(d) Merge S_1, S_2, Estimate the gain of G_1
$\mathcal{G} = \{G_1\}$
$R = \{S_3\}$
Output

Fig. 2. A running example of the proposed grouping approaches. (a) The algorithm in Section 3 first computes the gains of possible merging referring to an overlap function P of columns and groups, and prunes the impossible merges. (b) It then searches for the largest merging gain and iteratively merges the groups till all the gains are negative. (c) Alternatively, Section 4 presents an approximate approach $\hat{P}$ to estimate gains, and then (d) merges the column-groups in the same way as Section 3.

where $cost_g$ and $cost_c$ denote the space costs for single-column storage and single-group storage, respectively.

We will next define the problem of deciding the storage strategy for each time series. Intuitively, for time series with highly aligned timestamps, i.e., the timestamps are similar, we choose single-group storage for them. For the rest of the time series, we serialize them with single-column storage.

2.4 Column-Grouping Problem

2.4.1 Problem Definition. In this section, we will give a formal definition of the column-grouping problem. Following the idea of the column-groups storage in Section 2.3, the problem is to decide the storage strategy for each time series. That is to say, we need to decide which time series are stored in the same group with single-group storage, and which time series are stored separately with single-column storage. We thus formalize the following problem.

Problem 1 (column-grouping problem). *Given a set of n time series $\mathbf{S} = \{S_1, S_2, \cdots, S_n\}$, the column-grouping problem is to find a set of column groups $\mathcal{G}$ with single-group storage and a set of $\mathbf{R}$ with single-column storage, such that $\mathbf{G}_1 \cup \mathbf{G}_2 \cdots \mathbf{G}_K \cup \mathbf{R} = \mathbf{S}$ and the total space cost $cost_{cg}(\mathcal{G}, \mathbf{R})$ is minimized.*

As the example illustrated in Figure 2 below, the inputs are the data columns cached in memory, namely MemTable, to be flushed to disk. The outputs are the identified groups of columns, i.e., the strategy used to flush the data in disk.

In general, we do not have any assumption about the data. However, the approximate overlap estimation proposed in Section 4 works better with (nearly) regular time intervals.

In an LSM-tree store, considered in this paper, the updates of time series data will be processed in MemTable before flushing to disk, or delayed to another new MemTable. One may run the grouping algorithm again when flushing that new MemTable.

In other words, the streaming sensor data are cached in an active MemTable and flushed when full. To handle drastic change, one may call the grouping algorithm for each MemTable flush.

2.4.2 Hardness. Unfortunately, the problem is generally hard.

Theorem 1. *For a set of n time series, Problem 1 of finding a grouping scheme with the minimum $cost_{cg}(\mathcal{G}, \mathbf{R})$ is NP-hard.*

Proof sketch. To show NP-hardness, we build a reduction from the k-set packing problem, one of the Karp's 21 NP-complete problems [15]. The complete proof of NP-hardness is available in [4].

In practice, a device is often attached with tens or hundreds of sensors (columns). The data are first cached in memory with a size usually no greater than millions. Before flushing the data from memory to disk, we determine the grouping scheme. Referring to the NP-hardness w.r.t. the number of columns (series), we present an evaluation on finding the optimal grouping scheme, to show the challenging column sizes, in Section 6.3.

2.4.3 Problem Focus and Solution Applicability. Remarkably, for univariate time series that are almost time-aligned and are sampled at near-regular intervals, the grouping methods would perform better (with more efficient pruning in Section 3.1 and more accurate estimation in Section 4 below). Nevertheless, our proposal is generally applicable to any type of time series, i.e., not limited to univariate time series.

3 COLUMN-GROUPING ALGORITHM

In this section, we devise the column-grouping algorithm for the problem. Owing to the hardness of the problem and the infeasibility of enumerating all the possible grouping strategies, we propose to find the solution with a bottom-up structure, following a greedy strategy that merges the current groups with the most reduction of cost (namely merging gains) in each step. Figure 2 illustrates a running example of performing the column-grouping algorithm. As shown in Figure 2(a), the merging strategy first computes the gains of possible merging, and prunes the impossible merges. Next, in Figure 2(b), the algorithm searches for the largest merging gain and iteratively merges the groups till all the gains are negative.

3.1 Merging Strategy

As aforementioned, in each step of the grouping algorithm, we aim to merge the groups or the columns. Recall that the column-groups storage is to balance the repeated timestamps and the bitmaps for a smaller total cost. We therefore discuss the gain of reducing the space cost when we merge different groups or columns. Next, we transform the merging gains into the overlaps to efficiently compute the space cost. Based on the transformation, pruning strategies are further studied for efficiency. Examples of computing merging gains and pruning are illustrated in Figure 2(a). In the (gray) gain matrices, the numbers are the computed merging gains, as defined in Section 3.1.1 below. We denote 'x' the merging gain pruned by the merging strategy, and stored as *null* values in the implementation.

3.1.1 Merging Gain. To compute the merging gain, i.e., the space cost reduction of merging two groups or columns, we discuss different scenarios including column-column merging, column-group merging and group-group merging with their corresponding reduction in space costs, namely **merging gain**. For column-column merging, let S_1, S_2 be two columns to be merged, the merging gain $\Delta(S_1, S_2)$ of merging S_1 and S_2 should be:

$$\begin{aligned}\Delta(S_1, S_2) &= cost_c(\{S_1\}) + cost_c(\{S_2\}) - cost_g(\{S_1, S_2\}) \\ &= (m_{S_1} + m_{S_2} - m_{\{S_1,S_2\}})\alpha - 2m_{\{S_1,S_2\}}\beta.\end{aligned} \tag{5}$$

For column-group merging, let $n_{\mathbf{G}}$ denote the number of the time series in group $\mathbf{G}$, and let $S_1, \mathbf{G}_1$ be the column and the group to be merged, the merging gain $\Delta(S_1, \mathbf{G}_1)$ of merging S_1 and

$\mathbf{G}_1$ is also related to the number of time series in $\mathbf{G}$:

$$\begin{aligned}\Delta(S_1, \mathbf{G}_1) &= cost_c(\{S_1\}) + cost_g(\mathbf{G}_1) - cost_g(\{S_1\} \cup \mathbf{G}_1) \\ &= (m_{S_1} + m_{\mathbf{G}_1} - m_{\{S_1\}\cup\mathbf{G}_1})\alpha \\ &+ (n_{\mathbf{G}_1} m_{\mathbf{G}_1} - (n_{\mathbf{G}_1} + 1) m_{\{S_1\}\cup\mathbf{G}_1})\beta.\end{aligned} \tag{6}$$

For group-group merging, let $\mathbf{G}_1, \mathbf{G}_2$ be the groups to be merged, the merging gain $\Delta(\mathbf{G}_1, \mathbf{G}_2)$ of merging $\mathbf{G}_1$ and $\mathbf{G}_2$ should be

$$\begin{aligned}\Delta(\mathbf{G}_1, \mathbf{G}_2) &= cost_g(\mathbf{G}_1) + cost_g(\mathbf{G}_2) - cost_g(\mathbf{G}_1 \cup \mathbf{G}_1) \\ &= (m_{\mathbf{G}_1} + m_{\mathbf{G}_2} - m_{\mathbf{G}_1\cup\mathbf{G}_2})\alpha \\ &+ (n_{\mathbf{G}_1} m_{\mathbf{G}_1} + n_{\mathbf{G}_2} m_{\mathbf{G}_2} - (n_{\mathbf{G}_1} + n_{\mathbf{G}_2}) m_{\mathbf{G}_1\cup\mathbf{G}_2})\beta.\end{aligned} \tag{7}$$

If the merging gain is great than 0, it indicates the total space cost could be reduced by the merging, which could guide the merging of the columns and groups. For example, in Figure 2(a), S_1 and S_2 have merging gain $2 > 0$, whereas merging S_2 and S_3 leads to negative gain -2.

3.1.2 Overlapping Bounds. The computation cost of the merging gain in Section 3.1.1 mainly comes from the computation of aligning the timestamps. In this section, we first consider transforming the computation of the merging gain to overlapping, i.e., computing the overlapping of two time series to efficiently merge the groups. The bounds of the overlaps are given to further filter the merging candidates.

Let $P(x, y)$ denote the number of the overlapping timestamps of T_x and T_y, where x and y could be any group or single time series. The intuition is that, the aligned timestamps of x and y could be derived by $P(x, y)$. We could thus transform the merging gains in Section 3.1.1 with the following lemmas.

Lemma 2 (column-column merging). *Let S_1, S_2 denote the columns to be merged. If $\Delta(S_1, S_2) > 0$, we have*

$$P(S_1, S_2) > \frac{2(m_{S_1} + m_{S_2})\beta}{\alpha + 2\beta}. \tag{8}$$

The column-column merging deals with the merging of columns, which is the basic scenario. A more complex scenario is to merge a column and a group, which is related to not only the overlaps but also the aligned timestamps of the group and the current time series in the group (i.e., $n_{\mathbf{G}_1}$).

Lemma 3 (column-group merging). *Let $S_1, \mathbf{G}_1$ denote the column and the group to be merged. If $\Delta(S_1, \mathbf{G}_1) > 0$, we have*

$$P(S_1, \mathbf{G}_1) > \frac{(n_{\mathbf{G}_1} m_{S_1} + m_{\mathbf{G}_1} + m_{S_1})\beta}{\alpha + (n_{\mathbf{G}_1} + 1)\beta}. \tag{9}$$

For group-group merging, similar bound could be proved.

Lemma 4 (group-group merging). *Let $\mathbf{G}_1, \mathbf{G}_2$ denote the groups to be merged. If $\Delta(\mathbf{G}_1, \mathbf{G}_2) > 0$, we have*

$$P(\mathbf{G}_1, \mathbf{G}_2) > \frac{(n_{\mathbf{G}_1} m_{\mathbf{G}_2} + n_{\mathbf{G}_2} m_{\mathbf{G}_1})\beta}{\alpha + (n_{\mathbf{G}_1} + n_{\mathbf{G}_2})\beta}. \tag{10}$$

According to the above lemmas, we successfully transform the merging gains into the overlaps with overlapping bound, which could enable further pruning. A brief summary of the merging gains and the overlapping functions is shown in Table 2.

Table 2. Merging gains and overlapping bounds

Merging gain	Overlapping bound
$\Delta(S_1, S_2) > 0$	$P(S_1, S_2) > \frac{2(m_{S_1}+m_{S_2})\beta}{\alpha+2\beta}$
$\Delta(S_1, \mathbf{G}_1) > 0$	$P(S_1, \mathbf{G}_1) > \frac{(n_{\mathbf{G}_1} m_{S_1}+m_{\mathbf{G}_1}+m_{S_1})\beta}{\alpha+(n_{\mathbf{G}_1}+1)\beta}$
$\Delta(\mathbf{G}_1, \mathbf{G}_2) > 0$	$P(\mathbf{G}_1, \mathbf{G}_2) > \frac{(n_{\mathbf{G}_1} m_{\mathbf{G}_2}+n_{\mathbf{G}_2} m_{\mathbf{G}_1})\beta}{\alpha+(n_{\mathbf{G}_1}+n_{\mathbf{G}_2})\beta}$

3.2 Bottom-Up Grouping Algorithm

In this section, we propose the bottom-up grouping algorithm to group the given time series, based on the merging strategy in Section 3.1. According to Problem 1, the input is a set of n time series $\mathbf{S} = \{S_1, S_2, \cdots, S_n\}$.

Due to the hardness of the problem introduced in Section 2.4, we devise a greedy strategy to merge the columns into groups and locally minimize the overall cost. A collection of column groups $\mathcal{G} = \{\mathbf{G}_1, \mathbf{G}_2, \cdots, \mathbf{G}_K\}$, and a gain matrix $\mathbf{B}$ is maintained to record the current grouping scheme and the corresponding merging gains. Note that if $|\mathbf{G}_i| = 1$, $\mathbf{G}_i$ could be stored with single-column storage, since there is no need for $\mathbf{G}_i$ with size 1 to use bitmap for marking null values.

In the initialization stage, $\mathcal{G}$ is initialized by $\mathbf{G}_i \leftarrow \{S_i\}, 1 \leq i \leq n$. Following Section 3.1, the algorithm then computes the merging gains of all the pairs of the time series to initialize the gain matrix $\mathbf{B}$, where $b_{ij} = \Delta(\mathbf{G}_i, \mathbf{G}_j)$.

In each iteration, if there exists $b_{ij} > 0$, i.e., a pair with positive merging gain, it indicates that there is still an opportunity for reducing the space cost by merging. We thereby take the greedy strategy to merge the pair with the maximal gain. The processing is divided into two phases: the **merging phase** and the **updating phase**. Figure 2 illustrates a running example of the iteration.

(1) Merging phase. The merging phase aims to find the maximal merging gain in the gain matrix $\mathbf{B}$, and then conduct the merging of the group. The algorithm first seeks for the maximal merging gain in $\mathbf{B}$, denoted by b^*_{ij}. The corresponding groups or columns $\mathbf{G}_i, \mathbf{G}_j$ in $\mathcal{G}$ are thus selected for merging. After determining $\mathbf{G}_i, \mathbf{G}_j$, a new group $\mathbf{G}_{new}$ is created by merging $\mathbf{G}_i, \mathbf{G}_j$, i.e., including all the time series of $\mathbf{G}_i, \mathbf{G}_j$. We then replace $\mathbf{G}_i, \mathbf{G}_j$ in $\mathcal{G}$ with $\mathbf{G}_{new}$, and reorganize $\mathbf{B}$.

(2) Updating phase. The updating phase further updates the gain matrix $\mathbf{B}$ after $\mathbf{G}_i, \mathbf{G}_j$ are merged. According to the merging phase, in each iteration, only the gains between the updated group $\mathbf{G}_{new}$ and other elements in $\mathcal{G}$ should be recomputed, while the other gains are reserved.

Both phases are conducted iteratively until there does not exist $b_{ij} > 0$. Then the algorithm outputs the current grouping scheme. The number of the groups is exactly the parameter K defined in Problem 1, which could be determined by the algorithm. Algorithm 1 outlines the bottom-up grouping algorithm.

For instance, Figure 2(a) provides an example of the updating phase, where the gain matrix is fully computed (or pruned). Figure 2(b) shows an example of the merging phase, where S_1 and S_2 with the maximal gain are merged into $\mathbf{G}_1$ to reduce space cost.

4 APPROXIMATE OVERLAP ESTIMATION

In this section, we propose to approximately evaluate the overlaps without traversing, i.e., estimate the overlaps, denoted by function $\hat{P}$. Rather than going through the whole time series, we propose to extract a fixed-length feature from the time column, to represent the time column (Section 4.1).

Algorithm 1: Bottom-Up Grouping (S, n)

```
   Input: a set of time series $\mathrm{S} = \{S_1, S_2, \cdots, S_n\}$
   Output: the grouping scheme $\mathcal{G}$
 1 $k \leftarrow 0$;
 2 $\mathcal{G} \leftarrow \{\{S\} | S \in \mathrm{S}\}$;
 3 initialize $\mathbf{B}_{n \times n}$ with $null$;
 4 for $i = 1, 2, \cdots, n$ do
 5     for $j = i+1, i+2, \cdots, n$ do
 6         $b_{ij} = \Delta(S_i, S_j)$;
 7 while $\exists b_{ij} > 0$ do
 8     $b^*_{ij} = \max(\mathbf{B})$;
 9     $\mathcal{G} \leftarrow (\mathcal{G} - \{\mathrm{G}_i, \mathrm{G}_j\}) \cup \{\mathrm{G}_i \cup \mathrm{G}_j\}$;
10     update $\mathbf{B}$ corresponds to $\mathcal{G}$;
11 return $\mathcal{G}$;
```

In the meantime, we estimate the overlaps approximately based on the extracted features, following the probability distribution of the timestamps. Then the estimation of the overlaps between columns could be conducted efficiently (Section 4.2). Moreover, following the same line, we extend the estimation for single columns to column-groups (Section 4.3), defining $\hat{P}$ for all three cases as P. Such overlap estimation function $\hat{P}$ could replace P, which significantly reduces the running time of the algorithm. Figure 2 presents a running example of the approximate method. In Figure 2(c), the approximate approach $\hat{P}$ estimates the merging gains, as presented in Sections 4.2 and 4.3. It then merges the columns in the same way as Section 3 in Figure 2(d).

4.1 Feature Extraction

In this section, we propose the time column feature extraction, which extracts a fixed-length feature from the time column to represent its characteristics. The extracted features could enable more efficient overlap estimation.

As we only consider the feature extraction of the time column, i.e., the timestamps, two intuitions of the time column are first considered: (1) The time column is monotonically increasing; (2) In IoT scenarios, the time series are common with regular interval (i.e., collected with fixed or similar frequencies).

Based on the aforementioned intuitions, the straightforward idea is to model the time column in the manner of regular interval time series. Let $T = (t_1, t_2, \ldots, t_n)$ denote the time column of a time series with length n, to extract the feature from T, three features of a regular interval time series are considered first.

4.1.1 Interval. We first consider the interval of a regular interval time series, i.e., the differences between consecutive timestamps. For a regular interval time series, the interval is constant. Motivated by the robustness of the median, we take the median of the timestamp differences as the interval ϵ:

$$\epsilon = median(t_2 - t_1, t_3 - t_2, \ldots, t_n - t_{n-1}). \tag{11}$$

4.1.2 Length. The length of the regular interval time series is important to overlap estimation. We use n (length of T) as the feature.

4.1.3 Start Timestamp. The start timestamp of the regular interval time series should also be a feature to locate the timestamps. Although t_1 could be directly used as the start timestamp, it could also be erroneous, thus making the whole feature extraction biased. Since we treat the time series as a regular interval time series, we could first compute virtual timestamps for all timestamps in T (by subtracting a specific number of intervals), and then take the median. Formally, the start timestamp t^s is computed as follows:

$$t^s = median(t_1, t_2 - \epsilon, t_3 - 2\epsilon, \ldots, t_n - (n-1)\epsilon). \tag{12}$$

4.1.4 Variance. While the aforesaid three features can represent a regular interval time series, for an arbitrary time series (i.e., with irregular intervals), they are not sufficient to represent. We further include a feature to represent how the time series conforms to the extracted regular interval time series, namely variance.

For an arbitrary time series, as we extract the interval, the length and the start timestamp from it, we model a regular interval time series from the original time series. The regular time series T^r extracted from T satisfies $t_i^r = t^s + (i-1)\epsilon$. Variance is defined based on the corresponding timestamps of T^r and T, to denote the deviation of the T to its regular representation T^r:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(t_i^r - t_i)^2}{n}}. \tag{13}$$

If $\sigma = 0$, it is exactly a regular interval time series, i.e., $T = T^r$.

In summary, the feature of given T is extracted as $[\epsilon, n, t^s, \sigma]$.

4.2 Overlap Estimation for Columns

After extracting the features, we discuss the overlap estimation by utilizing the proposed features. We start from the overlap estimation for columns. Since we abstract the time columns as regular interval time series, the idea is to model the timestamps as distributions, and estimate the overlaps based on the distributions. Therefore, we take the following steps to estimate the overlaps for columns. For instance, in Figure 2(c), the gains of merging pairs of $S1, S2, S3$ are all estimated in the gain matrix. It is not surprising that the estimation may be slightly different from the exact evaluation in Figure 2(a), as in Section 3.

4.2.1 Range Normalization. First, we notice that only overlapping ranges could generate overlapping timestamps. We therefore normalize the ranges of both time series into their overlapping range, thus filtering the ranges impossible to overlap. Let T_a and T_b be the input time series, and $[\epsilon_a, n_a, t_a^s, \sigma_a]$, $[\epsilon_b, n_b, t_b^s, \sigma_b]$ be their features, respectively. Let T_a^r and T_b^r denote the regular interval time series of T_a and T_b, identified by their features (i.e., ϵ, n, t^r). To normalize the ranges, we find the range of T_a^r, i.e., $R(T_a^r) = (t_a^s, t_a^s + (n_a - 1)\epsilon_a)$, and the range of T_b^r, i.e., $R(T_b^r) = (t_b^s, t_b^s + (n_b - 1)\epsilon_b)$. Their overlapping range is denoted by $R = R(T_a^r) \cap R(T_b^r) = (R_{start}, R_{end})$. If $R = \emptyset$, the time series do not overlap, and the algorithm returns 0 as a result. We then consider the scenarios when $R \neq \emptyset$.

Let t'_a is the latest timestamp of T_a^r before R, i.e.,

$$t'_a = \max t, t \leq R_{start}.$$

Analogously, t'_b is the latest timestamp of T_b^r before R. Then, let n'_a be the length of the normalized range of T_a, having

$$n'_a = \min n, t'_a + (n-1)\epsilon_a \geq R_{end},$$

and n'_b is defined analogously. We thus normalize their ranges into $(t'_a, t'_a + (n'_a - 1)\epsilon_a)$ and $(t'_b, t'_b + (n'_b - 1)\epsilon_b)$, where they could overlap. Figure 3(a) gives an example.

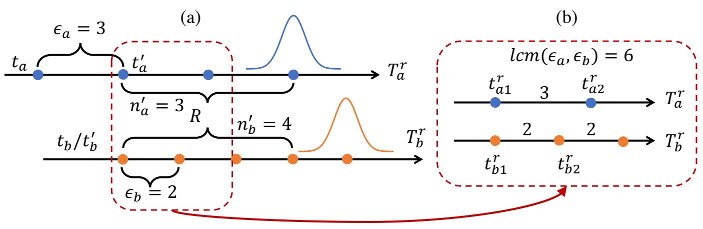

Fig. 3. An example of estimating the overlaps.

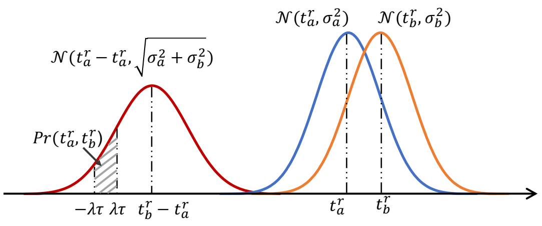

Fig. 4. Computing overlapping area of the distributions

4.2.2 Overlap Estimation. To estimate the overlaps for columns, we model each point of T_a and T_b with Gaussian distribution. For instance, given any $t^r_a \in T^r_a, t^r_b \in T^r_b$, their Probability Density Functions (PDFs) are denoted by $f_1(x; t^r_a, \sigma_a) = \mathcal{N}(t^r_a, \sigma^2_a), f_2(x; t^r_b, \sigma_b) = \mathcal{N}(t^r_b, \sigma^2_b)$. For t^r_a, t^r_b, their probability to be equal is given by the difference of their distributions, which also follows Gaussian distribution of $f(x; t^r_b - t^r_a, \sigma_a, \sigma_b) = \mathcal{N}(t^r_b - t^r_a, \sigma^2_a + \sigma^2_b)$. Due to the granularity of the timestamps, denoted by τ (e.g., 1h, 1s, 1ms), if $t^r_b - t^r_a$ falls into a discrete range $(-\lambda\tau, \lambda\tau)$, we could regard $t^r_a = t^r_b$, i.e., an overlap, where λ is a constant. Indeed, λ could control the deviations that we tolerate for overlaps. The probability of $t^r_a = t^r_b$, denoted by $Pr(t^r_a, t^r_b)$, is thus computed as:

$$\begin{aligned} Pr(t^r_a, t^r_b) &= \int_{-\lambda\tau}^{\lambda\tau} f(x; t^r_b - t^r_a, \sigma_a, \sigma_b)dx \\ &= \int_{-\lambda\tau}^{\lambda\tau} \frac{1}{\sqrt{2\pi(\sigma^2_a + \sigma^2_b)}} e^{-\frac{1}{2}\left(\frac{x-(t^r_b - t^r_a)}{\sqrt{\sigma^2_a + \sigma^2_b}}\right)^2} dx \\ &= \frac{1}{2}\operatorname{erf}\left(\frac{\lambda\tau - (t^r_b - t^r_a)}{\sqrt{2(\sigma^2_a + \sigma^2_b)}}\right) - \frac{1}{2}\operatorname{erf}\left(\frac{-\lambda\tau - (t^r_b - t^r_a)}{\sqrt{2(\sigma^2_a + \sigma^2_b)}}\right), \end{aligned} \tag{14}$$

where

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-\eta^2} d\eta$$

is the error function of a normal distribution. Figure 4 illustrates an example of computing overlapping area of the distributions.

The formula gives the probability of overlapping, given any two timestamps from T^r_a and T^r_b. Indeed, considering $t^r_a \in T^r_a$, according to the distributions of timestamps in T^r_b, it is true that any $t^r_b \in T^r_b$ is possible to be equal to T^r_a according to Formula 14. However, in terms of the complexity of the problem, we simplify the estimation by only considering the probability of t_a^r and its nearest neighbor in T_b^r to be equal. This falls into two scenarios based on the relationship between ϵ_a, ϵ_b.

(1) If $\epsilon_a = \epsilon_b$, i.e., they have the same frequencies, it is obvious that for $t_a^r \in T_a^r$, the distance $|t_a^r - t_b^r|$ between t_a^r and its nearest neighbor t_b^r is fixed. Therefore, for each $t_a^r \in T_a^r$, we can directly find its nearest $t_b^r \in T_b^r$. The overlaps of T_a and T_b, i.e., $\hat{P}(T_a, T_b)$, are thus estimated as

$$\hat{P}(T_a, T_b) = Pr(t_a^r, t_b^r) * \min(n_a', n_b').$$

(2) If $\epsilon_a \neq \epsilon_b$, i.e., they have different frequencies, the periodicity should be taken into consideration. Intuitively, the least common multiple of ϵ_a and ϵ_b (denoted by $lcm(\epsilon_a, \epsilon_b)$) should be a period, where all possible differences of t_a^r and t_b^r appear. Without loss of generality, let $\epsilon_a > \epsilon_b$, i.e., T_a^r contains fewer timestamps. For all timestamps of T_a^r in a period, there are $\kappa = \frac{lcm(\epsilon_a,\epsilon_b)}{\epsilon_a}$ scenarios of the differences between t_a^r and its nearest t_b^r. We use $\{t_{a1}^r, t_{a2}^r, \cdots, t_{a\kappa}^r\}$ to denote t_a^r corresponding to the κ scenarios, and their corresponding nearest timestamps in T_b^r is denoted by $\{t_{b1}^r, t_{b2}^r, \cdots, t_{b\kappa}^r\}$, respectively. We can compute the overlaps of T_a and T_b according to their proportions in one period:

$$\hat{P}(T_a, T_b) = \sum_{i=1}^{\kappa} \rho_i * Pr(t_{ai}^r, t_{bi}^r), \tag{15}$$

where κ denotes the scenario number in a period, ρ_i is the number of points following the i-th scenario in the overlapping range R.

Example 4.1. Figure 3 illustrates an example of estimating the overlaps, given T_a^r and T_b^r with $\epsilon_a = 3$ and $\epsilon_b = 2$ respectively. Figure 3(a) first normalizes the ranges of T_a^r and T_b^r into R, with $n_a' = 3$ and $n_b' = 4$. Figure 3(b) then illustrates the timestamps in one period, with the length of $lcm(\epsilon_a, \epsilon_b) = 6$, and thus $\kappa = \frac{lcm(\epsilon_a,\epsilon_b)}{\epsilon_a} = 2$. In the period, t_{a1}^r, t_{a2}^r with the interval of 3 are illustrated, and their corresponding nearest neighbors t_{b1}^r, t_{b2}^r are found, with $t_{b1}^r - t_{a1}^r = 0$ and $t_{b2}^r - t_{a2}^r = -1$. Following the definition of ρ_i, $t_{b1}^r - t_{a1}^r = 0$ appears twice in R and $t_{b2}^r - t_{a2}^r = -1$ appears once, i.e., $\rho_1 = 2$, $\rho_2 = 1$. The overlaps are thus estimated by $\hat{P}(T_a, T_b) = 2Pr(t_{a1}^r, t_{b1}^r) + Pr(t_{a2}^r, t_{b2}^r)$, following Formula 15.

4.3 Overlap Estimation for Column-Groups

In this section, we further discuss how to estimate overlaps for column-groups. When the columns are merged into one column-group, a set of features for the column-groups will be maintained, including all the features of the columns in the group. We then propose to estimate the overlaps between a column and a column-group. That is to say, we do not merge the features directly, which might lose more information. Instead, we maintain all the features in a set and further propose to estimate overlaps over a set of features. A range map is first constructed for the column-group, to store all overlapping ranges for the time columns in the group. For example, in Figure 2(d), the gain of merging $\mathbf{G}_1$ and S_3 is estimated by the approximate approach as -4, again, a bit different from -3 in Figure 2(b) computed by the exact method in Section 3.

4.3.1 Column-Group Estimation. Given a column-group $\mathbf{G}$, let $\mathbf{T}_\mathbf{G} = \{T_1, T_2, \ldots, T_{|\mathbf{G}|}\}$ denote all the time columns in $\mathbf{G}$, and let T_i^r denote the regular interval time series of $T_i \in \mathbf{T}_\mathbf{G}$. First, we find all distinct timestamps from all T_i^r, denoted as T_{all}, and then sort T_{all} in time order, denoted by $\{t_1, t_2, \ldots, t_{|T_{all}|}\}$. Then, we use $\mathcal{R} = \{R_i | R_i = (t_i, t_{i+1}), t_i \in T_{all}\}$ to denote all the consecutive ranges in $\mathbf{G}$, divided by T_{all}. The range map is thus denoted by $M = \{(R_j, \mathbf{T}_j)\}$ to store the ranges with their overlapping time columns, where $R_j \in \mathcal{R}$ is a range, and $\mathbf{T}_j$ is the set of

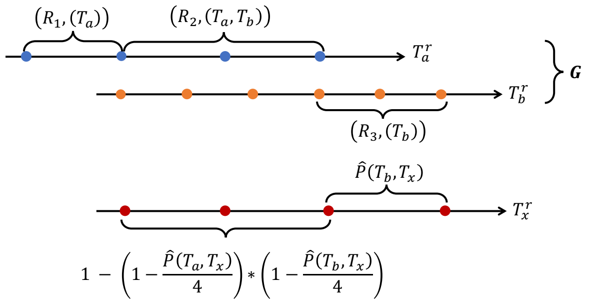
$(R_1, (T_a))$
$(R_2, (T_a, T_b))$
T_a^r
G
T_b^r
$(R_3, (T_b))$
$\hat{P}(T_b, T_x)$
T_x^r
$1 - \left(1 - \frac{\hat{P}(T_a, T_x)}{4}\right) * \left(1 - \frac{\hat{P}(T_b, T_x)}{4}\right)$

Fig. 5. Estimating the overlaps with a column-group

the time columns that overlap at R_j. Figure 5 shows the example of a range map, where $M = \{(R_1, (T_a)), (R_2, (T_a, T_b)), (R_3, (T_b))\}$.

By storing the features and the range map, the question is how to estimate the overlaps between a column-group (i.e., a set of features) and a column. Let T_x denote the input column, and let $M = \{(R_j, \mathbf{T}_j)\}$ denote the range map, following the Range Normalization step in Section 4.2.1, we find the overlapping ranges of R_x (i.e., the range of T_x) and R_j, having $(R_j, \mathbf{T}_j) \in M$.

Recall that $\hat{P}(T_i, T_x)$ denotes the overlaps of T_i and T_x, $\frac{\hat{P}(T_i,T_x)}{|T_x|}$ thus represents the proportion of T_x that might overlap T_i. The probability of their overlaps could be computed as follows:

$$\hat{P}(T_x, (R_j, \mathbf{T}_j)) = (1 - \prod_{T_i \in \mathbf{T}_j} (1 - \frac{\hat{P}(T_i, T_x)}{|T_x|})) * \mu_{xj}, \tag{16}$$

where μ_{xj} is the number of timestamps of T_x in the range R_j.

Therefore, based on Formula 16, the overlaps between a column-group $\mathbf{G}$ and a column T_x could be estimated by $\hat{P}(\mathbf{G}, T_x)$:

$$\hat{P}(\mathbf{G}, T_x) = \sum_{R_j \in \mathcal{R}} \hat{P}(T_x, (R_j, \mathbf{T}_j)),$$

where $R_j \in \mathcal{R}$, $\mathcal{R}$ is the set of consecutive ranges in $\mathbf{G}$.

Example 4.2. Figure 5 illustrates an example of estimating the overlaps between $\mathbf{G}$ and T_x, where $T_\mathbf{G} = \{T_a, T_b\}$. The range map is built as $\{(R_1, \{T_a\}), (R_2, \{T_a, T_b\}), (R_3, \{T_b\})\}$. Given an input time column T_x, which overlaps R_2, R_3, we could thus estimate the overlaps as:

$$\begin{aligned}\hat{P}(\mathbf{G}, T_x) &= \hat{P}(T_x, (R_2, \{T_a, T_b\})) + \hat{P}(T_x, (R_3, \{T_b\})) \\ &= 3\left(1 - \left(1 - \frac{\hat{P}\,(T_a, T_x)}{4}\right) * \left(1 - \frac{\hat{P}\,(T_b, T_x)}{4}\right)\right) + \hat{P}(T_b, T_x).\end{aligned}$$

4.3.2 Group-Group Estimation. Finally, given two column groups $\mathbf{G}_1, \mathbf{G}_2$, we first consider the points that each $T \in \mathbf{T}_{\mathbf{G}_2}$ overlaps with $\mathbf{G}_1$, and sum up them based on the proportions, denoted by an auxiliary function $\hat{P}^A(\mathbf{G}_1, \mathbf{G}_2)$:

$$\hat{P}^A(\mathbf{G}_1, \mathbf{G}_2) = \sum_{T \in \mathbf{T}_{\mathbf{G}_2}} \hat{P}(\mathbf{G}_1, T) * \frac{|T|}{\hat{m}_{\mathbf{G}_2}}. \tag{17}$$

Note that $\hat{m}_\mathbf{G}$ denotes the estimated length of $\mathbf{G}$, i.e., estimation of $m_\mathbf{G}$. We will discuss the evaluation of $\hat{m}_\mathbf{G}$ later.

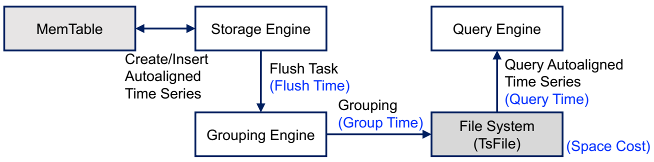

Fig. 6. System architecture for the column-groups storage. The metrics used in evaluations are in blue.

Thereby, $\hat{P}(\mathbf{G}_1, \mathbf{G}_2)$ is defined based on Formula 17. Since the overlaps should be possible in both groups, we then choose the smaller one of $\hat{P}^A(\mathbf{G}_1, \mathbf{G}_2)$ and $\hat{P}^A(\mathbf{G}_2, \mathbf{G}_1)$:

$$\hat{P}(\mathbf{G}_1, \mathbf{G}_2) = \min\left(\hat{P}^A(\mathbf{G}_1, \mathbf{G}_2), \hat{P}^A(\mathbf{G}_2, \mathbf{G}_1)\right).$$

Finally, as for $\hat{m}_{\mathbf{G}}$, it is stored with each column-group during the grouping. In the initialization stage, for all columns, we set $\hat{m}_T = m_T = |T|$. Whenever two column-groups are merged, for instance $\mathbf{G}_1$ is merged with $\mathbf{G}_2$, we update it as:

$$\hat{m}_{\mathbf{G}_1 \cup \mathbf{G}_2} = \hat{m}_{\mathbf{G}_1} + \hat{m}_{\mathbf{G}_2} - \hat{P}(\mathbf{G}_1, \mathbf{G}_2).$$

In this way, we maintain an estimated length for each $\mathbf{G}$ in the processing of grouping, and thus avoiding traversals of the whole time columns for the length.

5 SYSTEM DEPLOYMENT

The column-groups storage has been included [9] in Apache IoTDB [3], an open-source time series database. Figure 6 illustrates the architecture of the system related to the column-groups storage.

To enable the column-groups storage strategy in database, a keyword `autoaligned` is devised to declare the function in the SQL statement of creating time series and inserting data. The corresponding SQL statement for creating time series is:

```
create autoaligned timeseries root.sg1.d1(s1 INT32, s2 DOUBLE, s3 FLOAT)
```

which corresponds to the definition of $\mathbf{S} = \{S_1, S_2, S_3\}$ in Section 2.1. The create statement initializes the `autoaligned` time series in one MemTable (a sorted buffer in memory [17]) and defines its schema. The database engine thus maintains the MemTable in memory, till the flush process is called. All the time series are assigned to one group when initializing. The SQL statement of inserting data into the autoaligned time series is as follows.

```
insert into root.sg1.d1(time, s1, s2, s3) autoaligned values(1640966400000,1,2,3)
```

Flush task is executed when the memory buffer MemTable reaches a certain threshold, to serialize the data into the disk. FlushManager in the storage engine handles the flush task for asynchronous persistence without blocking normal writes.

The system supports metadata tightly coupled with time series data. In short, metadata in time series database with column-groups include: (1) Group info of time and value columns are new and stored together with other metadata below. (2) Encoding and compression schemes for each column are the same as single-column, since they can still be applied to the grouped columns. (3) Statistics of value columns in column-groups are the same as single-column, while statistics of timestamps are over the aligned time column.

6 EXPERIMENTS

The experiments are conducted over ten real-world datasets. While the source code of automatic column-grouping is available in the GitHub repository of Apache IoTDB [9], the experiment related code and data are in [5].

Table 3. Dataset summary that counts the number of points (t, v) collected in each time series, and the number of rows $(t, v_1, v_2, \ldots)$ aligned on timestamps of different time series.

	points	sensors	devices	rows	null rate
Campus	841,878	10	1	100,000	15.81%
CSSC	2,880,000	48	3	63,977	6.22%
GW	6,100,000	244	2	50,000	50.00%
TY	1,750,000	70	5	123,214	79.71%
WC	1,286,734	16	1	100,000	19.58%
WH	5,963,683	39	1	160,000	4.43%
CRRC	4,935,587	10,000	2	10,000	58.87%
APM	3,031,800	3	2	1,010,626	0.01%
Pistachio [30]	63,785	60	2	2,148	1.02%
Zomato [13]	550,000	22	2	50,000	50.00%

6.1 Experimental Settings

6.1.1 Datasets and Pre-Processing. We conduct the experiments over ten datasets. The datasets are listed in Table 3, and are described in detail as follows respectively.

(1) Campus contains climate data such as wind speed and temperature.
(2) CSSC contains ship engine data such as water temperature and oil pressure.
(3) GW contains data of wind turbines such as wind speed and direction.
(4) TY contains vehicle engine data such as vehicle speed and torque.
(5) WC contains vehicle data such as air input, engine torque and fuel liquid level.
(6) WH contains chemistry data such as high-speed pump operation used for fault diagnosis.
(7) CRRC contains train data such as train working conditions.
(8) APM contains log-type records such as cloud application performance monitoring.
(9) Pistachio [30] contains images (i.e., binary blobs type) of Pistachio and features.
(10) Zomato [13] is a non-time-series dataset of 50k records.

6.1.2 Metrics. Four metrics are employed to evaluate the methods, including **space cost**, **flush time**, **group time** and **query time**. Space cost denotes the storage space of time series in the file system, after they are flushed from MemTable into disk. Flush time is the time for the database to execute the flush tasks, and group time is the time for computing column-groups by our proposed grouping algorithms. Note that while the grouping algorithm is conducted during the flush process, to better evaluate the methods, we record group time and flush time separately. Query time is the time for querying autoaligned time series. The relationships of the metrics in terms of system architecture are illustrated in Figure 6.

6.1.3 Methods. We compare our proposed column-groups scheme based on exact evaluation (Section 3) and approximate evaluation (Section 4) with single-column and single-group schemes.

6.2 Evaluation over Different Datasets

To test scalability and stability of the proposed column-groups storage, we compare the column-groups storage with single-column storage and single-group storage on the four metrics introduced in Section 6.1.2. We evaluate the methods in two dimensions of scalability: the data size (i.e., number of records) and the time series number (i.e., number of sensors).

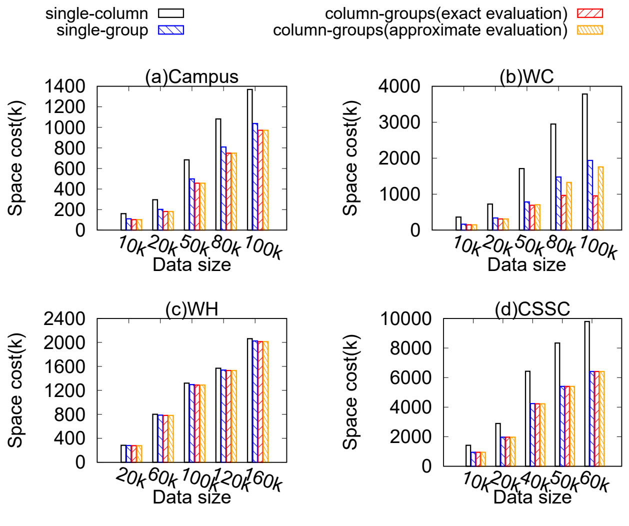

Fig. 7. Space cost by varying data size

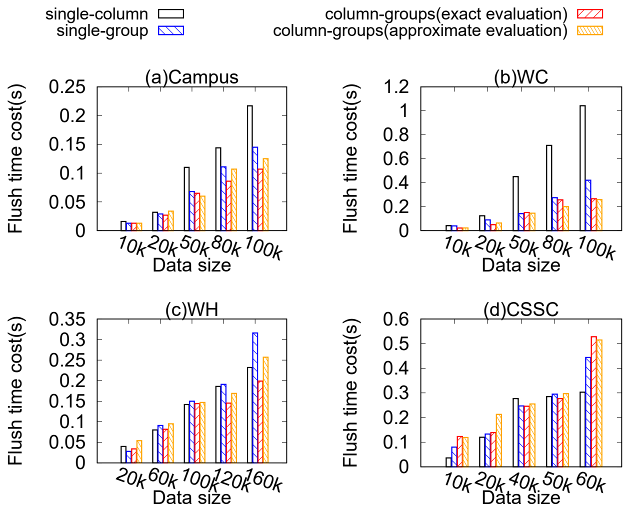

Fig. 8. Flush time by varying data size

6.2.1 Varying Data Size. Figures 7-10 report the results by varying the data size, i.e., the number of rows (aligned and unaligned).

As shown in Figure 7, it is not surprising that, due to the ability of column-groups storage to find a better grouping strategy for lower space cost, column-groups storage always shows the lowest space cost, compared to single-column storage and single-group storage. For the datasets that are highly aligned (e.g., CSSC and WH), column-groups storage has close but no worse performance than single-group storage. Overall, column-groups storage with approximate evaluation shows close results to exact evaluation, which demonstrates the accuracy of the overlap estimation.

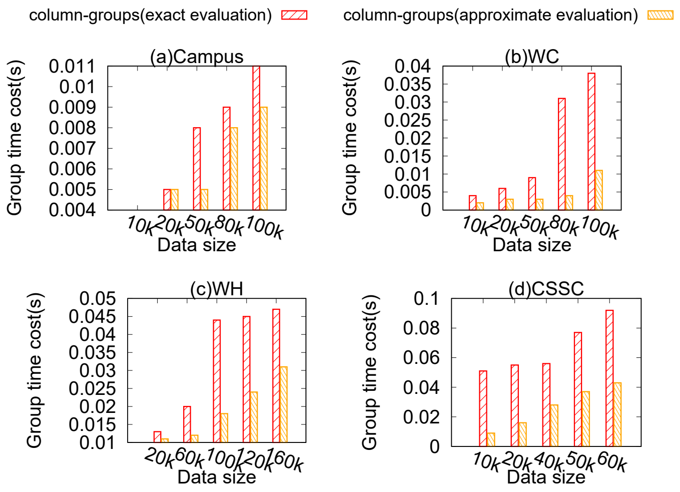

Fig. 9. Group time by varying data size

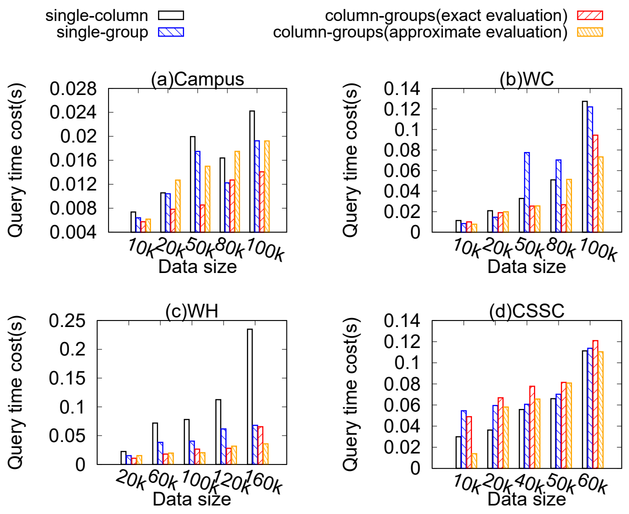

Fig. 10. Query time by varying data size

Figure 8 reports the flush time cost, which is the time for the flush thread to serialize the MemTable into file system. In all datasets, the flush time costs of the column-groups storage are comparable to those of single-column or single-group.

Figure 9 illustrates the group time of column-groups storage with exact and approximate evaluations. It is not surprising that approximate evaluation always shows lower group time cost.

For query time cost in Figure 10, it is related to the space cost. In most datasets, the column-groups storage shows comparable results in query time, which verifies the stability of the proposal.

Since WH is multivariate, the grouping algorithm suggests single-group scheme. Moreover, the dataset is with regular time interval, which can be effectively compressed. Thereby, the timestamps

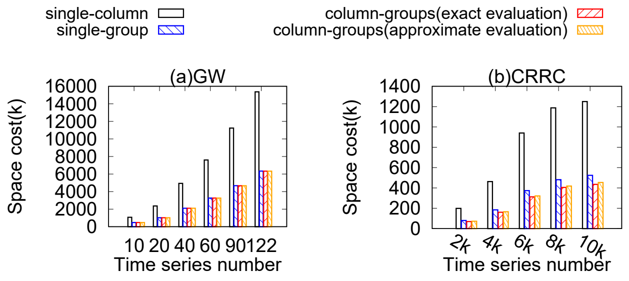

Fig. 11. Space cost by varying time series number

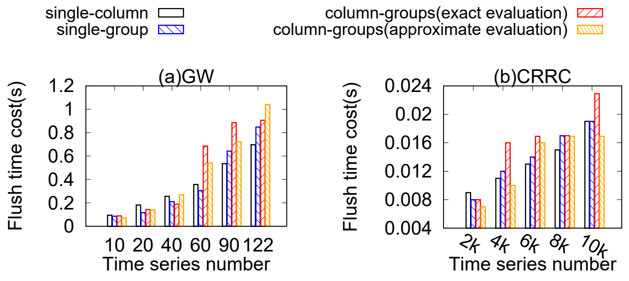

Fig. 12. Flush time by varying time series number

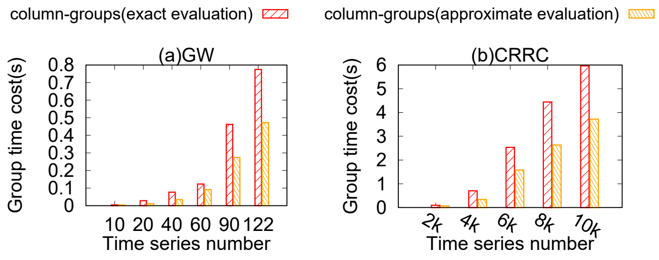

Fig. 13. Group time by varying time series number

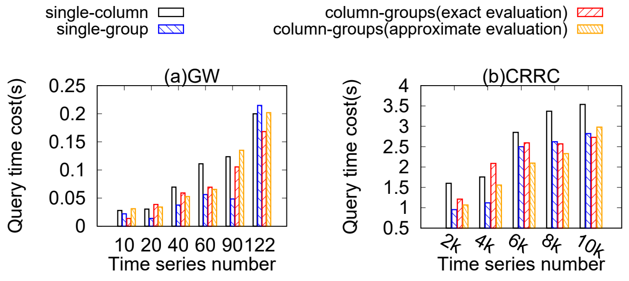

Fig. 14. Query time by varying time series number

do not take much space, i.e., the difference between single-column and single-group is not large. The column-groups scheme thus shows only a slight improvement, as shown in Figure 7(c).

Note that the target of column grouping, as stated in Problem 1, is to reduce the space cost (not query). The space cost is reduced with more column groups, i.e., lower I/O time as well. However,

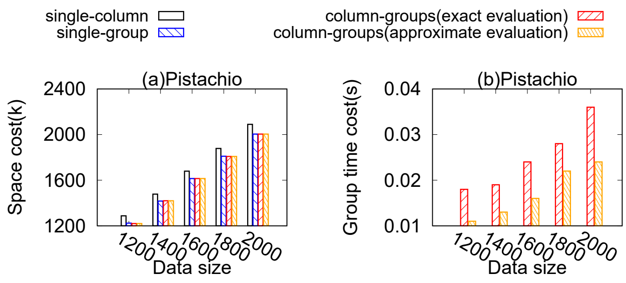

Fig. 15. Performance over Pistachio (binary blobs) dataset

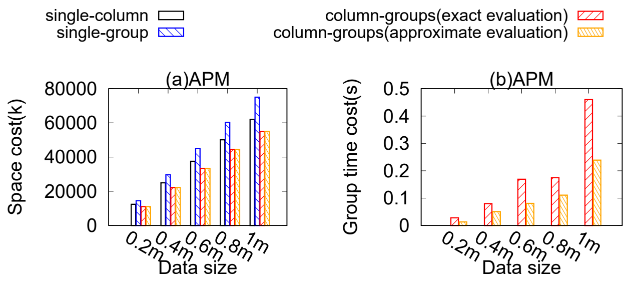

Fig. 16. Performance over APM (logs) dataset

the corresponding CPU time of aligning more groups may increase in query processing. Thereby, the exact evaluation (for better space cost in Figure 7) does not necessarily lead to lower query time than the approximate one, e.g., in WC and CSSC in Figure 10.

6.2.2 Varying Time Series Number. Another dimension of the data is the time series number, i.e., the number of sensors. Generally, more time series increase the difficulty for column grouping. We thus vary the time series number over the datasets in Figures 11-14.

The column-groups storage stably shows the lowest space cost in Figure 11. The time costs of flushing and querying are also comparable. Since the query selects all time series, the query cost increases with the number of series, in Figure 14.

In particular, we conduct experiments over CRRC, having a large number of time series (about 10k). The large number of columns brings more opportunities to optimize the grouping schemes. Therefore, the column-groups scheme shows clearer improvement, compared to single-column and single-group, in Figure 11(b).

6.2.3 Evaluation on Different Types of Datasets. We also conduct experiments on more diverse and larger time series data to show the effectiveness of the proposed approaches. For multivariate data, WH is exactly a dataset of multivariate time series, where a time column is shared by all value columns. It is not surprising that the grouping algorithm suggests single-group in Figure 7.

For binary blobs, Pistachio [30] is a dataset of Pistachio images. For logs, APM is a dataset of cloud application performance monitoring, including 1m log-type records. Figures 15 and 16 present the results over the Pistachio and APM datasets. The performances of our proposal over these datasets are consistent with the results in Figure 7, since it applies to any value type. In Pistachio, the space costs are dominated by the binary blobs due to their large sizes. Therefore, the spaces saved by our proposals are not obvious.

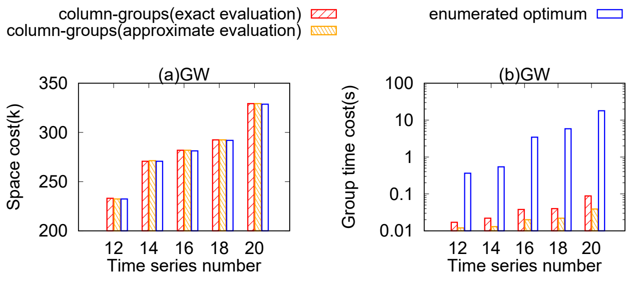

Fig. 17. Comparison with the optimum

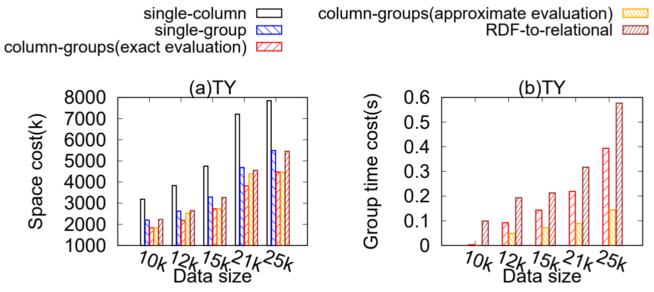

Fig. 18. Comparison with similar methods

6.3 Comparison with the Optimum

The grouping algorithm in Section 3 locally minimizes the space cost, i.e., a heuristic solution. The question is how the results of the algorithm are close to the optimal solution. Referring to the NP-hardness w.r.t. the number of columns (series) in Theorem 1, we implement a method based on enumeration to find the optimum grouping. The results are reported in Figure 17. Both column-grouping algorithms show almost the same space cost with the optimal results. The enumeration method takes about 20s for only 20 columns, i.e., 100× time cost of flush in Figure 12, too costly for grouping during flushing.

6.4 Comparison with Similar Methods

By interpreting subjects in RDF as timestamps and predicates as time series (columns), the problem of clustering predicates for transferring RDF to relational schemas [27] is similar to our problem of grouping columns for aligned time series. However, the unique features of timestamps, such as numeric with regular intervals, are different from subjects in RDF and not considered in the RDF-to-relational work. Figure 18 illustrates the comparison results. Owing to the pruning and estimation by the unique features of timestamps, our proposed column-groups show lower space cost and time cost than the RDF-to-relational approach.

6.5 Implementation in Other Systems

6.5.1 Implementation in TDengine. To show the applicability of the proposed grouping algorithms, we implement the proposals in another open-source time series database, TDengine [10]. TDengine is also based on LSM-tree in a columnar format. We thus build a prototype of our proposals over TDengine in C, compiled using GCC 9.4.0 in Linux. Analogous to the implementation in Section 5,

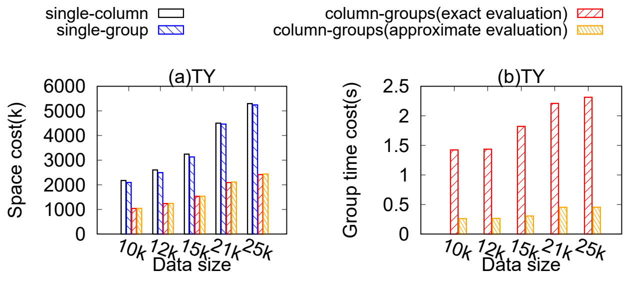

Fig. 19. Implementation in TDengine [10]

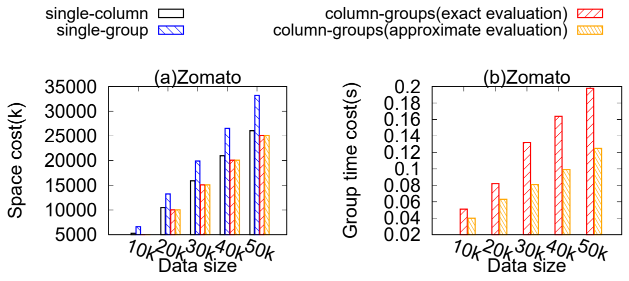

Fig. 20. Implementation in HBase [1]

the grouping algorithms are invoked during the procedure of flushing data into disk (namely "commit" in TDengine). The source code of the TDengine implementation is available in [11].

The results are reported in Figure 19, generally consistent with the evaluation in IoTDB in Figure 7. As shown, column-groups storage with exact and approximate evaluations shows lower space cost than single-column or single-group in TDengine as well. Again, approximate evaluation always shows lower grouping time costs.

6.5.2 Implementation in HBase. In general, our proposal is applicable to any data types with numerical key columns. Therefore, we implement the method in a more general columnar store, Apache HBase [1], for non-time-series data platforms. HBase also adopts an LSM-tree structure for columnar storage. Multi-column tables are also supported with a shared key. We thus implement the proposals at the stage of flushing regions (the data storage and management unit in HBase). The source code of the HBase implementation in Java is available in [2]. We conduct experiments over Zomato [13], a non-time-series dataset with numerical keys of 50k tuples. The results in Figure 20 are generally similar to those in time series databases, such as Figure 16 on the APM log data.

6.6 Combination with Compression

Note that the compression techniques can still be applied to grouped columns, i.e., complementary to our grouping algorithm. We evaluate how different grouping strategies work well with the compression techniques. The widely used encoding and compression methods for time series are employed, including TS2DIFF (an encoding method based on delta-of-delta [28]) and SNAPPY (a compression method [29]). For each grouping strategy, we combine it with different compression schemes, including Plain (without encoding or compression), TS2DIFF (with encoding), SNAPPY (with compression) and Mixed (with both encoding and compression). The results are reported in

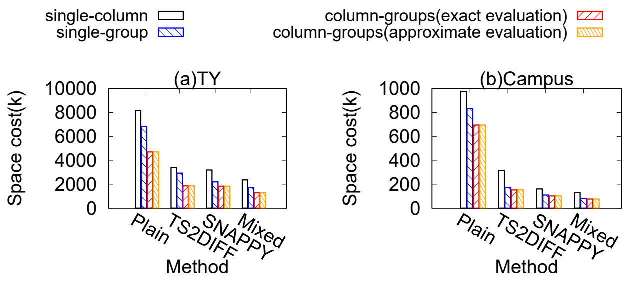

Fig. 21. Combination with compression

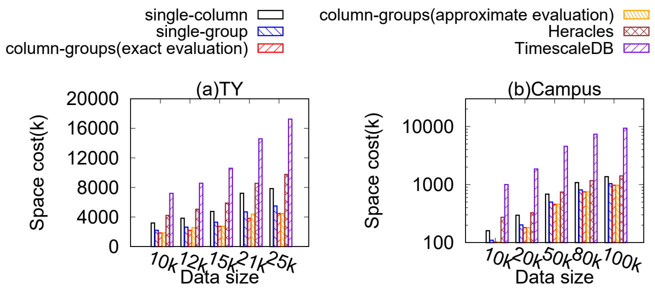

Fig. 22. Comparison with alternative systems

Figure 21. It is not surprising that the mixed compression strategy shows the lowest space cost. Besides, our proposed grouping algorithms cooperate well with all compression and encoding techniques, showing even lower space costs.

6.7 Comparison with Alternative Systems

Figure 22 compares our proposal with competing systems, including columnar Heracles [35] and non-columnar TimescaleDB [12].

Heracles [35] is a time series database based on Prometheus [8]. It uses a shared timestamp column for multiple time series, similar to the idea of our single-group. Heracles uses offset blocks to skip null values, while single-group scheme employs a bitmap to mark the null values. Remarkably, besides the reduced bitmap by grouping columns (not considered in Heracles), a full bitmap could be omitted for multivariate data with exactly aligned time. Thereby, the space cost of our single-column is lower than Heracles. By further considering column-groups, the space cost is even lower.

TimescaleDB [12] is built on PostgreSQL, supporting multivariate data. Owing to the row-oriented nature, it is not effective in handling null values, and thereby shows the highest space costs.

In summary, compared to the non-columnar strategy, e.g., in TimescaleDB, the columnar storage supports efficient compression of the series growing over time as illustrated in Figure 21, thus showing lower space cost in Figure 22. Moreover, compared to the columnar storage alternatives such as Heracles, our proposal handles better the null values together with non-null ones by column-groups, and thereby has less space cost.

7 RELATED WORK

7.1 Columnar Storage

Columnar storage is an industry standard design in most open-source or commercial time series database products, making them HTAP systems. Existing studies of columnar storage mainly work for distributed databases [16, 19, 23], relational databases [21, 24] and graph database systems[20]. In this study, we propose a novel grouping method for efficient columnar storage in time series database. Concerning the single-column storage and single-group storage in time series databases, we propose column-groups storage to save the space based on grouping algorithm.

7.2 Similarity Join

To compute the merging gain of merging the groups, the key is to compute the overlapping timestamps, which is similar to the idea of similarity join. [14, 18] study the set-similarity join (SSJoin) by signature-based methods, which mainly rely on the bound of the similarity functions. [34] proposes prefix-based method for adaptively adjusting the prefix, and employ inverted index for storing the prefixes. However, these approaches are devised mainly for string sets, due to the properties of the time series database, the prefixes are hard to define and it is too expensive to construct the inverted index. For similar reasons, the set-correlation method [32] is not applicable. Inspired by the properties of the time series, in this study, we devise pruning strategies for evaluating the merging gains and the overlaps.

7.3 Time Series Matching

As for matching approaches devised for time series, time series similarity matching method [36] proposes an index structure to deal with subsequence matching problem, which aims to find the most similar subsequence. While in our column-grouping problem, the overlaps of the complete time series should be computed. Existing study on matching temporal attributes [25] leverages timestamps and the values for time series schema matching. However, the method mainly works for matching the schema, and thus is not applicable to address our problem, which focuses on the overlaps of timestamps.

8 CONCLUSIONS

Among the design choices of grouping or not for efficiently storing multiple time series, in this study, we propose to first analyze the space cost of different storage schemes. The problem is thus to find the proper groups of time series that can minimize the cost, where no-grouping is also considered as a special column grouping scheme. Recognizing the NP-hardness of the problem, we turn to a more practical solution of a heuristic algorithm. It achieves near optimal space cost, without introducing much extra time cost. Indeed, both the flushing and querying time costs of column groups are comparable to those of single-column or single-group storage schemes. That is, while leading to more concise space cost, the extra column grouping step does not always increase the corresponding time cost. As the IO cost could be reduced with a more efficient column-groups storage, the total cost of flushing and querying may also reduce, as observed in Section 6.

ACKNOWLEDGMENTS

This work is supported in part by National Key Research and Development Plan (2021YFB3300500), the National Natural Science Foundation of China (62021002, 62072265, 62232005), Civil aircraft research project (MZJ3-2N21), Beijing National Research Center for Information Science and Technology (BNR2022RC01011), and Alibaba Group through Alibaba Innovative Research (AIR) Program. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] Apache HBase. https://hbase.apache.org/.

[2] Apache HBase Implementation. https://github.com/iotdbColumnGroup/HBase.

[3] Apache IoTDB. http://iotdb.apache.org.

[4] Appendix. https://iotdbcolumngroup.github.io/iotdbColumnGroup/appendix.pdf.

[5] Code and Data. https://github.com/iotdbColumnGroup/iotdbColumnGroup.

[6] InfluxDB. https://www.influxdata.com/.

[7] OpenTSDB. http://opentsdb.net/.

[8] Prometheus. https://prometheus.io.

[9] Source Code. https://github.com/apache/iotdb/tree/research/auto-aligned.

[10] TDengine. https://github.com/taosdata/TDengine/.

[11] TDengine Implementation. https://github.com/iotdbColumnGroup/TDengine.

[12] TimescaleDB. https://www.timescale.com.

[13] Zomato Dataset. https://www.kaggle.com/himanshupoddar/zomato-bangalore-restaurants.

[14] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-Similarity Joins. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006.* ACM, 918–929. http://dl.acm.org/citation.cfm?id=1164206

[15] Esther M Arkin and Refael Hassin. 1998. On local search for weighted k-set packing. *Mathematics of Operations Research* 23, 3 (1998), 640–648.

[16] Haoqiong Bian, Ying Yan, Wenbo Tao, Liang Jeff Chen, Yueguo Chen, Xiaoyong Du, and Thomas Moscibroda. 2017. Wide Table Layout Optimization based on Column Ordering and Duplication. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017.* ACM, 299–314. https://doi.org/10.1145/3035918.3035930

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 205–218. http://www.usenix.org/events/osdi06/tech/chang.html

[18] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA.* IEEE Computer Society, 5. https://doi.org/10.1109/ICDE.2006.9

[19] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. 2011. Column-Oriented Storage Techniques for MapReduce. *Proc. VLDB Endow.* 4, 7 (2011), 419–429. https://doi.org/10.14778/1988776.1988778

[20] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar Storage and List-based Processing for Graph Database Management Systems. *Proc. VLDB Endow.* 14, 11 (2021), 2491–2504. http://www.vldb.org/pvldb/vol14/p2491-gupta.pdf

[21] Muon Ha and Yulia A. Shichkina. 2022. Translating a Distributed Relational Database to a Document Database. *Data Sci. Eng.* 7, 2 (2022), 136–155. https://doi.org/10.1007/s41019-022-00181-9

[22] Shuai Han, Mingxia Liu, and Jian-Zhong Li. 2022. Efficient Partitioning Method for Optimizing the Compression on Array Data. *J. Comput. Sci. Technol.* 37, 5 (2022), 1049–1067. https://doi.org/10.1007/s11390-022-2371-7

[23] Donghe Kang, Ruochen Jiang, and Spyros Blanas. 2021. Jigsaw: A Data Storage and Query Processing Engine for Irregular Table Partitioning. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021.* 898–911. https://doi.org/10.1145/3448016.3457547

[24] Per-Åke Larson, Eric N. Hanson, and Susan L. Price. 2012. Columnar Storage in SQL Server 2012. *IEEE Data Eng. Bull.* 35, 1 (2012), 15–20. http://sites.computer.org/debull/A12mar/apollo.pdf

[25] Yinan Mei, Shaoxu Song, Yunsu Lee, Jungho Park, Soo-Hyung Kim, and Sungmin Yi. 2020. Representing Temporal Attributes for Schema Matching. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020.* ACM, 709–719. https://doi.org/10.1145/3394486.3403115

[26] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. 1997. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proceedings of the ACM SIGCOMM 1997 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, September 14-18, 1997, Cannes, France.* ACM, 181–194. https://doi.org/10.1145/263105.263162

[27] M. Tamer Özsu. 2016. A survey of RDF data management systems. *Frontiers Comput. Sci.* 10, 3 (2016), 418–432. https://doi.org/10.1007/s11704-016-5554-y

[28] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (2015), 1816–1827. https://doi.org/10.14778/2824032.2824078

[29] Horst Samulowitz, Chandra Reddy, Ashish Sabharwal, and Meinolf Sellmann. 2013. Snappy: A Simple Algorithm Portfolio. In *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7962)*, Matti Järvisalo and Allen Van Gelder (Eds.). Springer, 422–428. https://doi.org/10.1007/978-3-642-39071-5_33

[30] Dilbag Singh, Yavuz Selim Taspinar, Ramazan Kursun, Ilkay Cinar, Murat Koklu, Ilker Ali Ozkan, and Heung-No Lee. 2022. Classification and Analysis of Pistachio Species with Pre-Trained Deep Learning Models. *Electronics* 11, 7 (2022), 981.

[31] Shaoxu Song, Yue Cao, and Jianmin Wang. 2016. Cleaning Timestamps with Temporal Constraints. *Proc. VLDB Endow.* 9, 10 (2016), 708–719. https://doi.org/10.14778/2977797.2977798

[32] Shaoxu Song and Lei Chen. 2010. Efficient set-correlation operator inside databases. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010.* ACM, 139–148. https://doi.org/10.1145/1871437.1871459

[33] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin Mcgrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. 2020. Apache IoTDB: Time-series database for Internet of Things. *Proc. VLDB Endow.* 13, 12 (2020), 2901–2904. https://doi.org/10.14778/3415478.3415504

[34] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012.* ACM, 85–96. https://doi.org/10.1145/2213836.2213847

[35] Zhiqi Wang, Jin Xue, and Zili Shao. 2021. Heracles: An Efficient Storage Model And Data Flushing For Performance Monitoring Timeseries. *Proc. VLDB Endow.* 14, 6 (2021), 1080–1092. https://doi.org/10.14778/3447689.3447710

[36] Jiaye Wu, Peng Wang, Ningting Pan, Chen Wang, Wei Wang, and Jianmin Wang. 2019. KV-Match: A Subsequence Matching Approach Supporting Normalization and Time Warping. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019.* IEEE, 866–877. https://doi.org/10.1109/ICDE.2019.00082

Received April 2022; revised July 2022; accepted August 2022