RESPONSE LETTER

We thank the (meta) reviewers for insightful comments and constructive suggestions. The comments and changes are highlighted in different colors (violet for Reviewer 1, red for Reviewer 2, blue for Reviewer 3, and cyan for Reviewer 4).

Meta Reviewer

Thank you very much for your comments. We have addressed the reviewers' concerns and revision items in the revised manuscript. Please find our response to your comments as follows.

Required Changes: R1O1: Provide a more comprehensive discussion of how the proposed ideas relate to existing work.

Reply: For R1O1, we provide a more comprehensive discussion and experiments of how the proposed ideas relate to existing work.

R1O2: Clarify on approach's assumptions and include point queries in the experimental evaluation.

Reply: For R1O2, we clarify on our approach's assumptions and include point queries, compression performance on sorting columns, queries on NULL values, filter processing strategy in the experimental evaluation.

R1O3: Provide a more detailed explanation for the experiments.

Reply: For R1O3, we provide a more detailed explanation for setting, unit and compression operation of the experiments in reply to R1D9-D11.

R1O4: Improve on the paper's presentation.

Reply: For R1O4, we have revised proposed typos in R1D12-R1D21 and improve the presentation of the paper.

Required Changes: R2O1: Provide a discussion of how data updates are handled.

Reply: For R2O1, we provide a discussion and evaluation of how data updates are handled.

R2O2: Provide a more elaborate discussion of the determination of the sub-column width.

Reply: For R2O2, we discuss the determination of the sub-column width, and we add how to use the additional compression algorithms to compress sub-columns, such as dictionary encoding, in Section VI-E2, Page 12.

Required Changes:R3O1: Clarify on the paper's novelty and contributions.

Reply: For R3O1, we clarify on the paper's novelty and contribution, and we also compare query performance of our method with "Selection Pushdown in Column Stores using Bit Manipulation Instructions" in Section VI-D1, Page 11.

R3O2: Strength the motivation for the proposed cost-model based heuristic.

Reply: For R3O2, we strengthen the motivation and challenging for the proposed cost-model in Section I, Page **??**. Then, we introduce a heuristic method Sub-column Determination with PSO to solve the problem in the new Section A, Page 16, and discuss the compression performance of exact Sub-column and Sub-column with PSO in Section VI-E3, Page 12.

Required Changes: R4O1: Strengthen the motivation for finding the optimal bit width in the paper's introduction.

Reply: For R4O1, we strengthen the motivation for finding the optimal bit width in the paper's introduction in Section I, Page 1.

R4O2: Provide more information about the datasets in Section 1-A.

Reply: For R4O2, we provide more information about the datasets in Section I-A.

R4O3: Simplify and clarify the notations in the paper.

Reply: For R4O3, we simplify and clarify the notations in Section II-A, Page 3, and Section 1, Page 4.

R4O5: Improve on experimental evaluation by considering datasets with large bit width.

Reply: For R4O5, we improve on experimental evaluation by adding a dataset with large bit width.

R4O6: Include an ablation study for the experimental evaluation.

Reply: For R4O6, we include an ablation study in Section VI-E4 of Page 12.

Comments on the Revised Submission (if Appropriate): R4O4: Provide a theoretical analysis of the compression ratio.

Reply: For R4O4, we provide a theoretical analysis of the compression ratio for a special case, the normal distribution in Section III-D, Page 6.

Reviewer #1

R1O1: The analysis and comparison is limited to time series use cases while compression and query processing on compressed columns is widely adopted for columnar databases. Some of these systems have a wider range of compression techniques. Others, e.g. Apache Parquet supports a very similar set of compression techniques as IoTDB. A more comprehensive discussion how the presented ideas relates to these systems is important.

why mention series everywh in the paper?

Reply: Thank you for your suggestions about the scope of our algorithm. The analysis and comparison of our algorithm Sub-column is based on columns, with not only time-series data but also non-time-series datasets. For example, Arade, EE, Gov and WT are non-time-series datasets as described in Section VI-A2. In Figure 7, Sub-column has also the best compression ratio on these datasets. Hence, Sub-columns could also compress columns for columnar databases.

In response to R1D1-D4, we also provide more detailed comparative analysis with other systems including Apache Parquet, C-store and SAP HANA in the revised manuscript.

For R1D1, we add the discussion and evaluation about delta encoding and dictionary encoding in Apache Parquet in Section VI-D1, Page 10.

impleme and evaluatio in Parqu

For R1D2, we have provided a more detailed discussion of their ideas and limitations to better elucidate the contributions of our algorithm in Section VII-C, Page 13.

For R1D3, we add the discussion about the tuple materialization strategy, and we add an experiment to compare four materialization strategies based on our sub-column algorithms

and materialization strategies in Bitweaving [25] in the new Section VI-D4, Page 11.

For R1D4, we compare our Sub-columns algorithm with compression method of SAP HANA on an equidistant piece-wise table [37] in Section VI-D1, Page 10.

R1O2: The narrow scope of the paper mentioned in O1 naturally limits also the solution space making the contribution of the paper quite narrow.

Reply: Thanks for your feedback about contributions of the paper. Our overall contributions are not narrow, but we employ Prop 1-3 for pruning to optimize compression time. Additionally, optimizations of Sub-column are not only effective in storage and query performance within real-world database Apache IoTDB and but also other columns storage such as Apache Parquet as shown in reply to R1O1. In response to R1D5-D8, we provide a detailed comparative analysis of our algorithm's scope and contribution in the revised manuscript.

For R1D5, we add discussion and evaluation about point queries to new Section VI-E4, Page 13.

For R1D6, we add an experiment and discussion about sorting the columns and compare methods of the paper R3 [35] in Section VI-E1, Page 12.

For R1D7, we provide a discussion and experimental analysis on aggregation on columns with NULL values in Section VI-D3, Page 11.

For R1D8, we discuss the more explanation about filter processing strategy in Section VI-D2, Page 11.

R1O3: The explanation of the experiments requires more detailed explanation.

Reply: Thanks for your suggestion about the explanation of the experiments in D9-D11, and we have thoroughly revised the manuscript to provide a more detailed and clear explanation of the experiments, as outlined below point by point.

For R1D9, we have further clarified the experimental settings in Section VI-A, Page 9.

For R1D10, we add more detailed explanation about the unit of throughput in Section VI-A3, Page 9.

For R1D11, we provided further details on compression and decompression operation in Section VI-A3, Page 9.

R1O4: Several presentation issues of the paper need to be fixed.

Reply: We have corrected these presentation issues in the revision version according to comments in R1D12-D21.

R1D1: Note that delta encoding (available, e.g. in Parquet) or dictionary encoding (available, e.g. in Parquet) can address the issue of leading zeros. Please discuss.

Reply: Thanks for your positive feedback about compression techniques in Parquet. We add the discussion and evaluation about delta encoding and dictionary encoding in Parquet in Section VI-D1, Page 10.

We evaluate compression ratio of delta encoding and dictionary encoding in Apache Parquet in Figure 9(a). As described in Parquet [5], delta encoding decreases the magnitude of the values to be stored, further adding leading zeros. The compression in dictionary encoding is achieved by replacing each repeated value with a key, which has more leading zeros than the original values. Hence, delta encoding has better compression ratio than Sub-columns on some datasets in Figure 9. But Sub-column has bigger compression ratio by adding more zeros of sub-columns, thus, the compression ratio of Sub-column is better than those of Parquet (Delta) and Parquet (Dictionary) as presented in Figure 9.

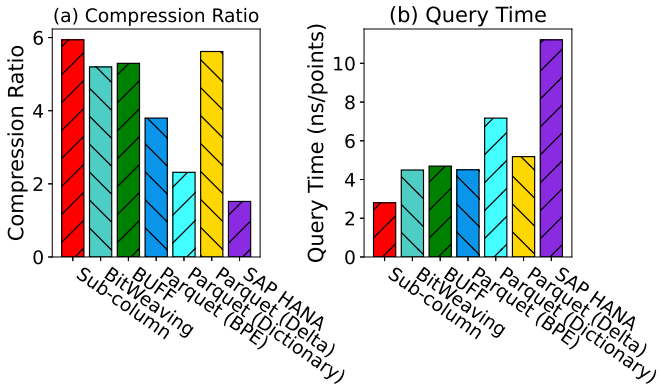


Fig. 9: Comparing compression ratio of compressions with adding leading zeros in Parquet

R1D2: Similar ideas presented in this paper were also presented in, e.g.

R1:https://dl.acm.org/doi/pdf/10.14778/1921071.1921077 - query processing on groups of columns

R2:https://dl.acm.org/doi/10.1145/2463676.2465322 and various follow-up work

Reference [1] of the paper and that work should be discussed in more detail.

Reply: Thank you for your suggestion about papers with similar ideas. Among them, R1 is to partition data into chunks and compresses each column using dictionary and run-length encoding in Hyrise. R2 is to organize sub-column bits into contiguous groups to optimize CPU cache utilization during scans in BitWeaving [25]. Reference [1] is to apply applies RLE and dictionary encoding to compress columns in C-Store. and we provide a more detailed discussion of their ideas and limitations to better elucidate the contributions of our algorithm in Section VII-C, Page 13.

Hyrise [15] partitions data into chunks and compresses each column using dictionary and run-length encoding. Its query engine processes predicates directly on compressed blocks to minimize data access and late tuple reconstruction. C-Store [1] applies RLE and dictionary encoding to compress data. But Hyrise and C-Store neither split the column into sub-columns for compression nor perform sub-column-level query processing.

BitWeaving [25]organizes sub-column bits into contiguous groups to optimize CPU cache utilization during scans. However, it does not explore how different bit widths affect compression and query performance.

R1D3: The paper should explain the tuple materialization strategy and its impact on performance. For

align names of Parquet

mention R1 is ..., R2 is ... Referenc [1] is ....

too word indicate differenc directly ¿what is the differenc

simply apply?

which is the focus

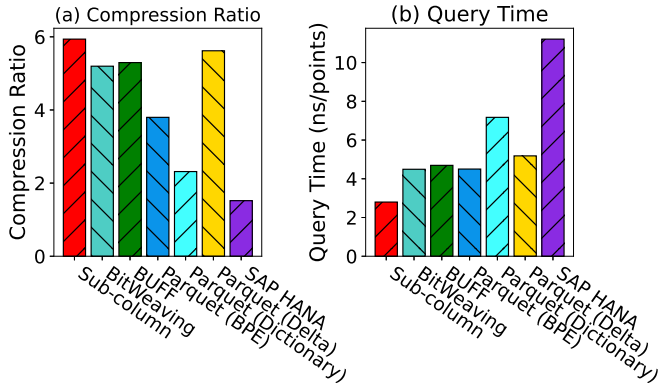


Fig. 9: Compression and query performance of Sub-column and other baselines

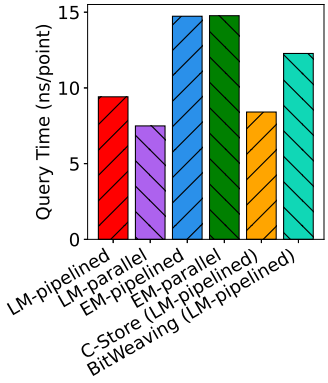


Fig. 10:Impact on query performance of Sub-column Algorithm by different tuple materialization strategies

example, the method in R2 above is not very effective in materializing tuples. Also see reference [1] and https://www.cs.umd.edu/~abadi/papers/abadiicde2007.pdf for further discussions.

Reply: Thanks for your advice about the tuple materialization strategy. We add the discussion about the tuple materialization strategy and its impact on query performance, and we conduct an experiment to compare query performance on four materialization strategies based on our sub-column algorithms and materialization strategies in Bitweaving [25] (i.e., R2) in a new Section VI-D4 of Page 11.

Materialization is the process of adding columns to intermediate results [2], i.e., https://www.cs.umd.edu/~abadi/papers/abadiicde2007.pdf. [2] notes that the point at which columns are materialized into tuples is a key performance factor: early materialization may inflate I/O and CPU costs by assembling tuples before filtering is complete, whereas late materialization can avoid unnecessary work and thus improve query execution efficiency. Authors mention four materialization strategies including two early materialization strategies (EM-pipelined and EM-parallel) and two late materialization ones (LM-pipelined and LM-parallel) [2]. R2 (BitWeaving) [25] uses LM-pipelined, which performs well only for highly selective queries but becomes inefficient when many tuples or columns must be reconstructed. Its conversion from bit-vector to tuples adds significant overhead, making materialization very low effective. C-Store [1] also uses LM-pipelined and performs worse for lowly selective queries. BitWeaving and C-Store do not filter with sub-columns, thus it is not effective for them to reconstruct materialization tuples.

We conduct an experiment to compare the average query time of four materialization strategies based on our sub-column algorithm and materialization strategy LM-pipelined in BitWeaving [25] and C-Store [1] in Figure 10. The LM-parallel strategy employed by our algorithm demonstrates better query time performance than alternative database materialization strategies for the real-world dataset.

R1D4: Some systems, e.g. SAP HANA, also support compression on columns of timeseries data, see https://help.sap.com/docs/SAP_HANA_PLATFORM/b2f4bdf7b83f4444bfab5564e9ff6aee/a17e909dfd144b2ab8f6a411eee91cb4.html?locale=en-US.

Reply: Thank you for your suggestion about SAP HANA, which also supports compression on columns of timeseries data. In Section VI-D1, Page 10, we compare our Sub-columns algorithm with compression method of SAP HANA based on an equidistant piecewise table in [37], i.e., https://help.sap.com/docs/SAP_HANA_PLATFORM/b2f4bdf7b83f4444bfab5564e9ff6aee/a17e909dfd144b2ab8f6a411eee91cb4.html?locale=en-US.

The algorithm in SAP HANA for compressing piecewise equidistant tables employs arithmetic encoding of timestamp columns using base values, incremental units, and nanosecond offsets to achieve high compression efficiency while maintaining query performance [37]. Our Sub-column algorithm could also compress timestamp columns with piecewise, and we compare compression and query performance of Sub-column and SAP HANA in Figure 9. Since our Sub-column could remove more leading zeros, Sub-column has better compression ratio than method of SAP HANA as shown in Figure 9(a). The query time of our algorithm outperforms compressions of SAP HANA, as demonstrated in Figure 9(b), due to (1) accelerated range filtering via compression-aware data skipping, and (2) enhanced aggregation performance by leveraging run-length in RLE-compressed sub-columns.

R1D5: Note that RLE compression is inefficient for point queries. Hence, point queries should also be part of the evaluation.

Reply: We are grateful to the reviewer for suggestions about point queries, and we add discussion and evaluation about point queries to new Section VI-E, Page 13.

To evaluate the effectiveness of our algorithm for point queries and the impact of RLE on point query performance, we measured the point query time for Sub-column-only-BPE, Sub-column-only-RLE and our proposed approach Sub-column. Point query with RLE requires cumulative calculation of run-lengths to determine positions, whereas BPE allows direct jumping to the point position. As a result, the point

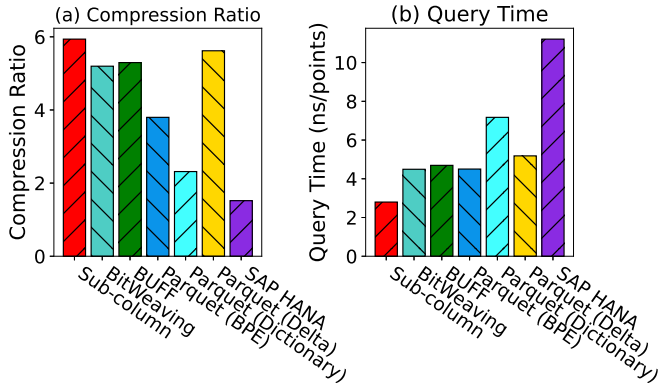


Fig. 9: Compression and query performance of Sub-column and other baselines

query time of Sub-column is higher than that of Sub-column with BPE, but better than that of Sub-column with only RLE, as demonstrated in Figure 17(d).

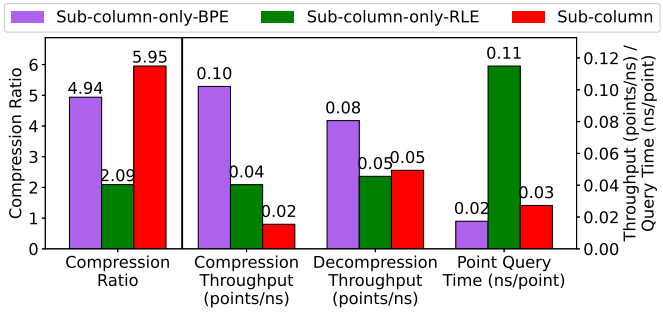


Fig. 17: Point query of Sub-column, Sub-column with only BPE or RLE

R1D6: Note that RLE - and also other compression schemes can benefit from sorting the columns. It seems an implicit assumption of the paper that sorting columns is not allowed. Please elaborate why this is not an option? Also see R3: https://core.ac.uk/download/pdf/539565665.pdf.

Reply:

We appreciate the reviewer's comments about benefiting from sorting the columns. And we add an experiment and discussion about sorting the columns in Section VI-E1, Page 12.

The paper R3 [35] proposes an entropy-based algorithm for detecting soft functional dependencies, allowing columns with similar entropy to share a combined dictionary and improve compression when sorted. This method was implemented in SAP Net Weaver Business Warehouse Accelerator (BWA), which uses dictionary and run-length encoding (RLE) on value blocks. Sorting increases consecutive repeated values, reducing both the number of RLE pairs and unique values per block, thereby enhancing compression efficiency.

In Figure 13, we present the compression ratio of Dictionary, RLE and our Sub-column on columns sorted by key or value, since the most important functional dependencies are those where a candidate key (or the primary key) functionally determines other attributes [39]. Sorting improves the compression ratio of RLE as shown in Figure 13. However, data can only be ordered by the values of one column, values of the other column can appear in random order as said in [35]. As a result, the compression ratio of Dictionary and Sub-column deteriorates. Hence, sorting value columns is not always an effective option to improve compression ratio. Notably, even though sorting columns according to value yielded a noticeable improvement in Figure 13, RLE are still worse than the compression ratio achieved by our algorithm Sub-column.

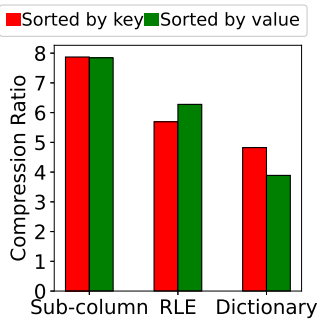


Fig. 13: Compression ratio on columns sorted by keys or values

R1D7: Looking at algorithm 3 and based on the text, it seems that columns may not contain NULL values. At least, the algorithm is not correct in the presence of NULL values or empty input.

Reply: We sincerely appreciate the reviewer's the helpful comment about processing of of NULL values, and we add the discussion and experiments about aggregation on columns with NULL values in Section VI-D3, Page 11.

In Apache IoTDB, the positions of null values are recorded with a bitmap, while non-null values are stored in a compressed format. This allows aggregation such as MAX(X) and SUM(X) to be computed directly over the compressed non-null values, whereas COUNT(X) can be efficiently derived from the bitmap. We evaluated the execution time of the aggregation queries, MAX(X), SUM(X), and COUNT(X) using Algorithm 3. The results, presented in Figure 11, show that the average time per point for MAX(X) and SUM(X) remains nearly constant regardless of the null values, since Algorithm 3 processes non-null compressed values directly. Since COUNT(X) could be efficiently derived from the bitmap, the time of COUNT(X) is very low in Figure 11.

R1D8: The filter processing strategy in section IV.A assumes that the first filters applied are very selective. Also, this may not be an effective strategy, e.g. when filters with many matching rows on two columns are anti-correlated. Then filtering these columns in parallel may be faster.

Reply: We thank the reviewer for their insightful comments on filter processing strategy on two columns, and we add the more explanation about filter processing strategy in Section VI-D2, Page 11.

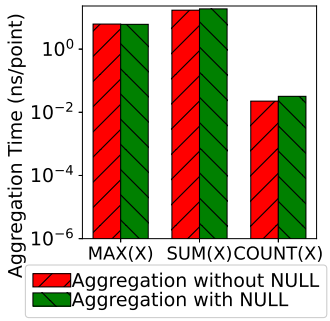


Fig. 11: Aggregation time on columns with NULL

We present the time of filter processing in pipeline and parallel on datasets and the selectivity and correlation of these datasets in Figure 12. Given the number of total rows S, the number of filtered rows A and B, and the number of match rows F, the selectivity τ [38] and Matthews correlation coefficient ϕ [29] are as follows,

$$\tau = \frac{\frac{F}{S}}{\frac{A}{S}\frac{B}{S}} = \frac{FS}{AB},$$

and

$$\phi = \frac{\frac{F}{S} - \frac{A}{S}\frac{B}{S}}{\sqrt{\frac{S-A}{S}\frac{S-B}{S}}} = \frac{FS - AB}{S\sqrt{(S-A)(S-B)}}.$$

As noted by the reviewer, the first filters applied are very low selective, thus, the filtering time in pipeline of BM and BP is better than that in parallel. When the correlation coefficient is negative or low positive, two columns are anti-correlated or weak correlated, and the filter processing in pipeline is not an effective strategy. Hence, the filter processing in parallel is significantly better than that in pipeline on datasets Arade, CT, and WS, as depicted in Figure 12.

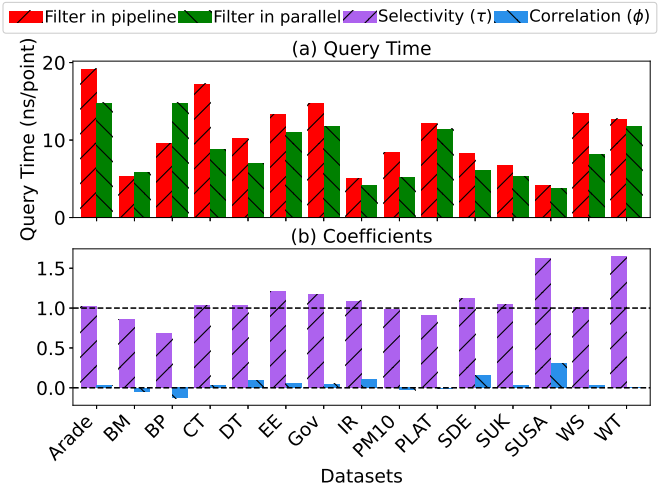


Fig. 12: The time of filter processing in pipeline and parallel on datasets and the selectivity and correlation of these datasets

R1D9: The system used in the experiments has a GPU. Is the GPU used in IoTDB? If yes, please explain how.

Reply: Thank you for this question. IoTDB does not utilize the GPU for the experiments conducted in this paper. To avoid misunderstanding, this setting has been removed from Section VI-A, Page 9.

R1D10: I do not understand the unit "points/ns". What are the points?

Reply: The term "points" refers to an array of value points to be uncompressed. Each value point is a value consisting of 8 bytes. The aforementioned explanation has been added to Section VI-A3, Page 9.

R1D11: For the compression and decompression experiments, I do not understand which operation is performed. Is compression basically a data loading? and decompression a full scan?

Reply: Yes, that is exactly correct. And we add the explanation about compression and decompression operation in Section VI-A3, Page 9.

In our experiments, compression refers to the operation of loading and transforming the data into a compressed byte stream, and decompression refers to the full scan and restoration of the compressed data to its original form.

R1D12-R1D21:

Reply: Thanks for many detailed comments. We have corrected these presentation issues in the revision version according to comments in R1D12-D21.

Reviewer #2

R2O1: The authors do not discuss the case of updates in the data. This might be challenging in the case of bit-packing encoding. Blocks limit the impact for append-only data, but not for general updates.

Reply: We appreciate the reviewer's comments and suggestions about updates in the data. As the reviewer said, blocks limit the impact for append-only data. For general 'update', we could directly change each sub-column of updated value v if the bit width is not larger than the maximum bit width. Otherwise, we have to add new sub-columns. We further discuss about update operations and conduct experiments about the update time in Section VI-D5, Page 11.

The following discussion about updates in the Sub-column compressed data is divided into 2 parts, append-only and general update. For 'append-only', blocks limit the impact for append-only data. And we add a new block for the new values appended, similar to appending data without compression. Thus, the time to handle append-only data is essentially its compression time, which is faster than general update operations. For general 'update', we could directly change each sub-column of updated value v if the bit width is not larger than the maximum bit width. Otherwise, we could add several new sub-columns for the portion of v that exceeds the maximum bit width.

We test the update time of Sub-column algorithm in Figure 14. We can observe that updating larger values is slower than

each rep
usually
has two
parts, (1)
discussio
with
review a
revision
place, (2
revision
paragrap
exactly.
check all
replies

updating smaller ones, and both are slower than the time of append-only data.

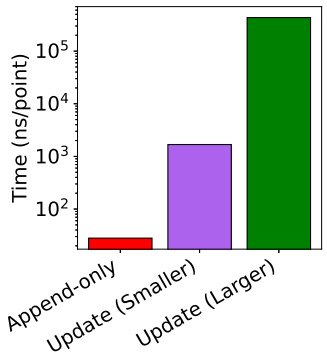


Fig. 14: Comparing update time based on Sub-column

R2O2: The authors discuss the determination of the sub-column width for run-length and bit-packing encoding and while they mention it is not limited to these two, they do not elaborate on this further.

Reply: Due to space constraints, two compression algorithms with better performance in compressing sub-columns, run-length and bit-packing encoding, were initially discussed. As follows, we further discuss how to use the additional compression algorithms to compress sub-columns, such as dictionary encoding, and we add the content in Section VI-E2, Page 12.

The determination of the sub-column width includes but not limited to bit-packing and run-length encoding. If a sub-column has smaller cardinality, the sub-column could be compressed by Dictionary Encoding (DE).

For the given series X and the bit width of sub-columns β, the j-th sub-column cost with bit-packing is,

$$DE(X, j, \beta) = \lceil \log(\omega_j + 1) \rceil n + 2(\beta + \omega_j),$$

where ω_j is cardinality of the j-th sub-column in the series, i.e., the number of unique values in the j-th sub-column.

If the bit width of cardinality ω_j is larger than that of sub-columns, i.e., $\lceil \log(\omega_j + 1) \rceil$ $b_\beta(j)$, the sub-column cost with dictionary encoding is greater than that with bit-packing encoding, and we could not compress the sub-column with dictionary encoding. Otherwise, when $DE(X, j, \beta)$ is smaller than $\min\{BPE(X, j, \beta), RLE(X, j, \beta)\}$, we could compress the j-th sub-column with dictionary encoding.

We evaluate the compression performance of Sub-columns with DE in Figure 15. As shown in Figures 15(a) and (b), the incorporation of DE results in a modest improvement in compression ratio but leads to degraded compression throughput. The improvement of decompression throughput is attributed to the fact that DE decompresses faster than RLE, as DE replaces RLE to compress several sub-columns.

R2Q9. Inclusive writing: Fig. 8 might be difficult to read for people with color impairment

Reply: Thank you for your feedback about inclusive writing. For clarity, the solid points represent the optimal bit width,

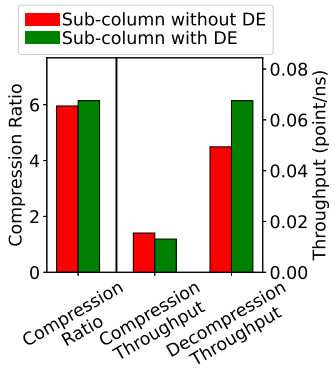


Fig. 15: Comparing compression ratio of Sub-column with and without DE

while the other points are displayed as hollow markers. And the legend is relocated to the right-hand side of the plot, with all points enlarged to improve visibility in Figure 8.

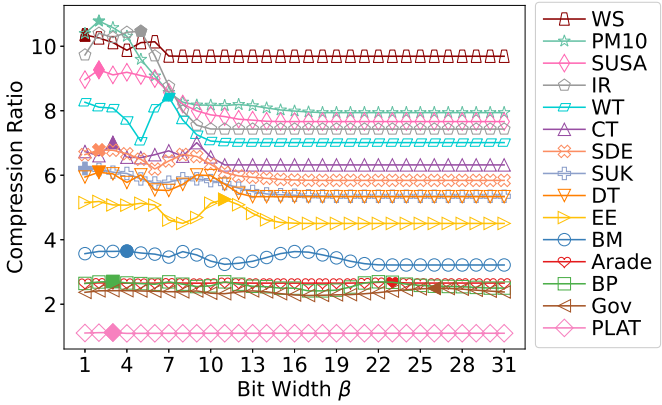


Fig. 8: Compression and decompression performance of our algorithm with various bit widths β of sub-columns

R2Q11. Additional remarks: There are many writing mistakes, formulas, text and links that go beyond the border and in Fig. 1 the x-label and sub-caption are overlapping.

Reply: Thank you for your feedback, and I have corrected the formatting issues, including the overlapping text in Fig. 1 and the content extending beyond the borders.

rectangle can be smaller, to reduce the heigh

Reviewer #3

R3O1: My main concern is that the novelty and originality of the work are not clearly demonstrated. For instance, the material in Section 4 appears quite similar to predicate pushdown techniques. Given the existing work in this area, I am not convinced that BUFF is the only or most distinctive approach to predicate pushdown. A good starting point could be "Selection Pushdown in Column Stores using Bit Manipulation Instructions".

Reply: Thanks for your suggestion about novelty and originality of the work. We add more detailed novelty and originality, and we also compare our method with "Selection

Pushdown in Column Stores using Bit Manipulation Instructions" in Section VI-D1, Page 11.

The novelty of Sub-column algorithm lies in its optimization of query time at the algorithmic level, rather than through hardware utilization. The algorithm Parquet-select in [24] leverages Bit Manipulation Instructions (BMI) to enable efficient selection pushdown on encoded columnar data, significantly reducing decoding overhead and improving query performance by up to an order of magnitude, even for complex predicates and nested structures. However, the approach is highly dependent on specific hardware support (e.g., Intel/AMD BMI instructions), limiting its portability to ARM-based architectures, and may incur additional overhead when processing mixed RLE and BPE data due to frequent context switching. We test compression ratio and query time of Parquet-select in Figure 9. The most queries of Parquet-Select is slower than Sub-column without BMI, and its has the worst compression ratio as depicted in Figure 9.

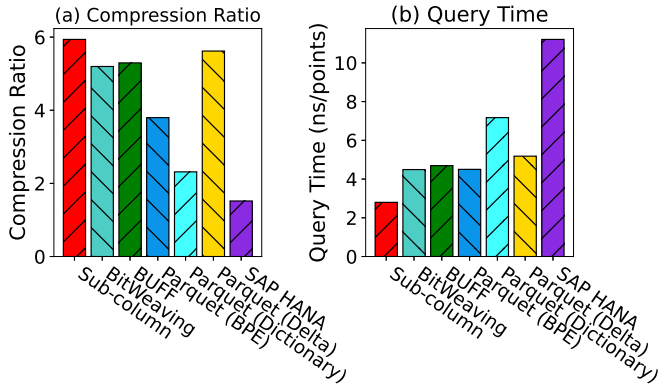


Fig. 9: Compression and query performance of Sub-column and other baselines

R3O2: Using a straightforward method—exhaustively traversing each possible beta—to motivate a cost-model-based heuristic is unconvincing. The problem is essentially one of optimal parameter search. It would be helpful to include a discussion of the unique challenges specific to this problem that cannot be effectively addressed by existing solutions, such as Particle Swarm Optimization.

Reply: Thanks for your constructive suggestions about the method. We strength the motivation and challenge for the proposed cost-model in Section VI-E3, Page 12. Then, we briefly introduce a heuristic method to solve Sub-column Determination problem using PSO, and discuss the compression performance of exact Sub-column and Sub-column (PSO) in Section VI-E3, Page 12. And we leave the details in online appendix [?].

The Sub-column determination problem is essentially one of bit width β search. Our proposed algorithm is a cost-model-based algorithm to find optimal bit width of sub-columns. The unique challenges specific to this problem is to find optimal storage cost of sub-columns with less compression time. Furthermore, we propose a discussion and experiments

Algorithm 4: Sub-column Determination with PSO

Input: Series $X = (x_1, x_2, \ldots, x_n)$, PSO hyperparams: swarmSize s, maxIter $t_{\max}$, inertia w, cognitive coefficients c_1, social coefficients c_2, localRadius R
Output: Bit width β'

1 $M = \lceil \log(x_{\max} - x_{\min} + 1) \rceil$
2 **for** $i \leftarrow 1$ **to** s **do**
3 $\quad p_i = U(1, M)$
4 $\quad v_i = U(-(M-1)/2, (M-1)/2)$
5 $\quad p_{best,i} = p_i$
6 $\quad Cost_{\min,i} = Cost(X, round(p_i))$
7 $p_{best} = \arg\min_i Cost_{\min,i}$
8 $C_{\min} = \min Cost_{\min,i}$
9 $v_{max} = M$
10 **for** $t \leftarrow 1$ **to** $t_{\max}$ **do**
11 $\quad$ **for** $i \leftarrow 1$ **to** s **do**
12 $\quad\quad r_1 = U(0,1), r_2 = U(0,1), v_i = w \cdot v_i + c_1 r_1 (p_{best,i} - p_i) + c_2 r_2 (p_{best} - p_i)$, clamp v_i to $[-v_{\max}, v_{\max}]$
13 $\quad\quad p_i = p_i + v_i$, clamp p_i to $[1, M]$ $\beta = round(p_i)$
14 $\quad\quad$ **if** $C(X, \beta) < p_{best,i}$ **then**
15 $\quad\quad\quad Cost_{\min,i} = C(X, \beta)$, $p_{best,i} = p_i$
16 $\quad\quad\quad$ **if** $C(X, \beta) < C_{\min}$ **then**
17 $\quad\quad\quad\quad C_{\min} = C(X, \beta)$, $p_{best} = p_i$
18 $\beta' = round(p_{best})$
19 **for** $b \leftarrow \beta' - R$ **to** $\beta' + R$ **do**
20 $\quad$ **if** $1 \le b \le M$ **then**
21 $\quad\quad$ **if** $Cost(X, b) < C_{\min}$ **then**
22 $\quad\quad\quad C_{\min} = Cost(X, b)$, $\beta' = b$
23 **return** β'

comparing the exact Sub-column Determination algorithm with the Sub-column Determination algorithm using PSO in Section VI-E3, Page 12.

We replace the exhaustive search for the optimal sub-column bit width with a Particle-Swarm-Optimization (PSO) based procedure in Algorithm 4. In Lines 4-9, algorithm initial i-th particle's position p_i, its velocity v_i, the best i-th position $p_{best,i}$, the minimum cost $Cost_{\min,i}$, the global best position p_{best}, and the global minimum cost $C_{\min}$. Then, after $t_{\max}$ iterations, the algorithm updates the minimum cost for each particle and the global minimum cost. Particles update their velocities and positions using the standard PSO rule with inertia w and cognitive/social coefficients c_1, c_2 in Lines 12-13. The global minimum cost $C_{\min}$ is updated whenever any particle finds a lower cost in Lines 14-18. Finally, in the vicinity of the optimal solution β' found by the global search of PSO, another local refinement is performed in Lines 19-23.

Since Algorithm 4 updates s particles with $t_{\max}$ iterations, the total runtime is approximately $s * t_{\max} * n$ and the time complexity is $O(n)$. The parameters s and $t_{\max}$ could be tuned to trade off computational cost against solution quality, which influences compression ratio. This method requires far fewer time cost evaluations than a full exhaustive search for large M in Algorithm 1, when $s * t_{\max}$ is far smaller than M.

We perform the evaluation about the compression performance of Sub-column (Exact) and Sub-column (PSO) in Section VI-E3. Note that Sub-column (PSO) is a bit worse than the Sub-column algorithm in Figure 16, since Sub-column (PSO) could only find an approximate solution to the problem. Nonetheless, due to fewer iterations, Sub-column (PSO) has better throughput than the Sub-column algorithm as depicted in Figure 16.

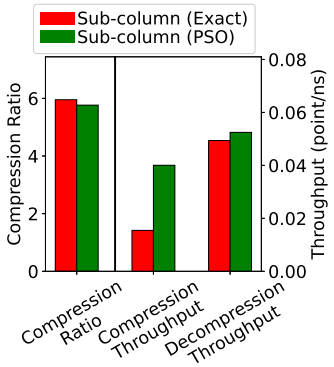


Fig. 16: Comparing compression ratio of exact Sub-column and Sub-column with PSO

Reviewer #4

R4O1: This paper spends a large space describing how to find the optimal bit width. However, the introduction lacks sufficient motivation or discussion on this aspect. Why is finding the optimal bit width challenging? What are the underlying trade-offs? It is necessary to include an intuitive explanation or a brief summary of the challenges.

Reply: Thanks for your valuable feedback about motivation of challenging and trade-offs. And we add the intuitive explanation and a brief summary of the challenges in Section I, Page 1.

To find the optimal bit width could improve compression ratio of sub-columns, but it takes too much time to traverse all possible bit width from 1 to 64, which could not be acceptable for compression algorithm. Then, it is challenging to take less time to find the optimal bit width to compress data.

Then, the optimal bit width β for sub-columns requires careful consideration and the underlying trade-offs: while larger β may reduce run-length to add the storage cost, smaller β could lead to excessive sub-columns and increased meta data storage overhead of sub-columns. Therefore, we need to propose a solution to determine the optimal bit width of sub-columns with smallest storage cost.

R4O2: In Section 1-A, please provide basic information about the real-world datasets.

Reply: In Section I-A, real-world datasets are all datasets in Table II, and the basic information of these datasets is shown in Section VI-A2. And we revise the caption of Figure 1 to explain about basic information, in Page 1.

R4O3: Some notations used in the paper are quite confusing and complex. Try to simplify these notations and make them clear.

E.g., in Definition 2.1, xcp+1;

E.g., in Proposition 1: n(Bj), Bj;

Reply: Thanks for your suggestions about notations in the paper. In Definition II.1 in Page 3, we have simplified x_{c_p} to x_α. Similarly, in Proposition 1 in Page 4, we have replaced βj with θ and $\eta(\beta j)$ is revised to $\eta(\theta)$.

R4O4: Section 3-D provides a time complexity analysis. I am curious if it is possible to include a theoretical analysis of the compression ratio. I understand this is hard. It is just a suggestion.

Reply: Thank you for your comments about a theoretical analysis of compression ratio. Although theoretical analysis of the compression ratio may not be feasible in all cases, we provide a theoretical analysis of the compression ratio for a special case, the normal distribution in Section III-D, Page 6.

a propos
tion?[add

Proposition 4. When the given series X with n values complies with i.i.d. $N(\mu, \sigma^2)$, the compression ratio of Sub-column is less than or equal to $\frac{64}{\lceil \log(\mathbb{E}[x_{\max}] - \mathbb{E}[x_{\min}] + 1) \rceil}$.

Proof. When the given series X with n values complies with i.i.d. $N(\mu, \sigma^2)$ [16], the expected minimum and maximum values are,

$$\mathbb{E}[x_{\max}] = \mu + \sigma \left[b_n + \frac{\gamma}{b_n} + o\left(\frac{1}{b_n}\right) \right],$$

and

$$\mathbb{E}[x_{\min}] = \mu - \sigma \left[b_n + \frac{\gamma}{b_n} + o\left(\frac{1}{b_n}\right) \right],$$

where b_n is $\sqrt{2 \ln n} - \frac{\ln \ln n + \ln(4\pi)}{2\sqrt{2 \ln n}}$. Then compression ratio is less than or equal to $\frac{64}{\lceil \log(\mathbb{E}[x_{\max}] - \mathbb{E}[x_{\min}] + 1) \rceil}$.

R4O5: The experiments show that the optimal bit width is around 1-4. It raises the question of whether this is because the datasets only have small bit widths. To validate, please consider including datasets with larger bit widths.

Reply: Thanks for your suggestion about the diversity of datasets, and we add a new dataset POI-lat in Table II and add the description about POI-lat (PLAT) in Section VI-A2 in Page 9. Then, we add the following discussion about optimal bit width of PLAT in Section VI-C in Page 10.

For datasets with larger bit width, the compression difficulty is greater, resulting in lower compression ratios that close to 1. Nevertheless, the optimal bit width of sub-columns is still within the 1-4 bit range. The bit widths of maximum value in PLAT is very large, i.e., 64 bits, but the optimal bit width for PLAT is 3 bits, as shown in Figure 8(a). The fact that the optimal bit width is consistently low (1-4 bits) suggests that this result is not attributable to the datasets having small bit widths.

R4O6: The experimental results would benefit from an ablation study. Please try to evaluate the performance when using only BPE or only RLE on sub-columns.

Reply: Thanks for your suggestion about the experimental results, and we add an ablation study about the compression

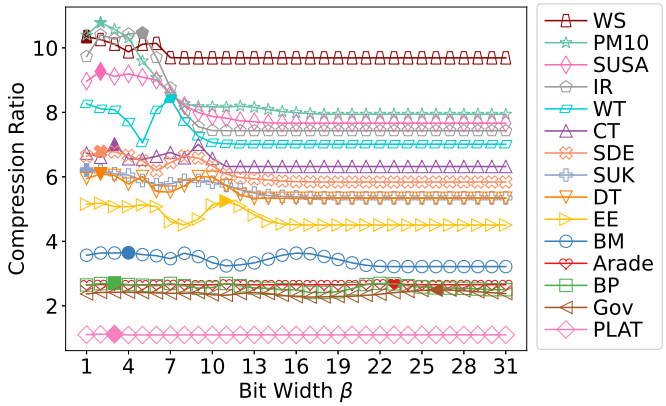


Fig. 8: Compression and decompression performance of our algorithm with various bit widths β of sub-columns

when using only BPE or only RLE on sub-columns in Section VI-E4 of Page 12.

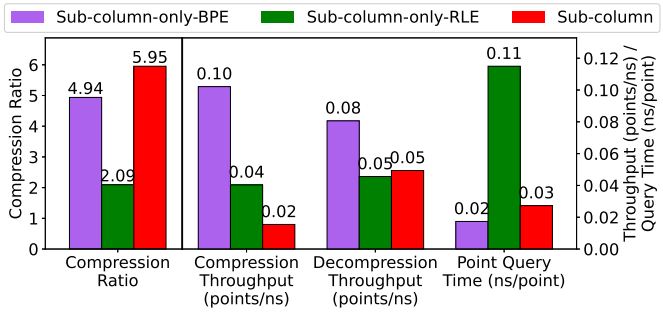


Fig. 17: Compression and query performance of ablation study on Sub-columns

Figure 17(a) shows the average compression ratio, compression throughput, and decompression throughput of Sub-column, Sub-column with only BPE, and Sub-column with only RLE across all datasets presented in Table II. The compression ratio of Sub-column with only BPE and Sub-column with only RLE is worse than that of our Sub-column algorithm, since BPE and RLE could reduce storage cost of sub-columns. However, it takes more time for RLE to compute the run-length series than for BPE to compress and decompress data. RLE has a worse compression and decompression time than BPE, hence, Sub-column algorithm has worse compression and decompression throughput than Sub-column with only BPE but better than Sub-column with only RLE.

Sub-column Compression for Query Processing

Jinzhao Xiao
Tsinghua University
xjc22@mails.tsinghua.edu.cn

Nan Yang
Tsinghua University
yangn21@mails.tsinghua.edu.cn

Shaoxu Song*
Tsinghua University
sxsong@tsinghua.edu.cn

Jianmin Wang
Tsinghua University
jimwang@tsinghua.edu.cn

***Abstract*—Different compression techniques may have distinct application scenarios. For instance, while Bit-packing encoding compresses values by representing them with a fixed bit width, run-length encoding reduces storage costs by storing run lengths of consecutive repeats for the data exhibiting high repeatability. However, when values are very large with seldom leading zeros or data has lower repeatability, bit-packing or run-length encoding will perform worse on compression. Intuitively, to improve their performances, we may divide values into several sub-columns such that leading zeros and repetition are increased for better bit-packing and run-length encoding. In addition, the sub-column compression could further improve query processing by enabling data skipping and filtering at the sub-column level. In this paper, we propose a sub-column framework, which dynamically determines the compression and bit width of sub-columns, enhancing the overall compression ratio. We also demonstrate how sub-column approaches can optimize query processing, particularly for range filtering and aggregation tasks. The experimental comparison over real-world datasets demonstrates the improvement of compression ratio and query time. Our sub-column compression and query processing methods have been deployed in Apache TsFile and Apache IoTDB.**

***Index Terms*—query, sub-columns, compression**

I. Introduction

Many compression algorithms [8], [49], [14], [4] use bit-packing encoding (BPE) [49] and run-length encoding (RLE) [14] as fundamental operations. Bit-packing encoding allocates a fixed number of bits to represent each value within a series, while run-length encoding stores value and its consecutive repeated times (i.e., run length). For example, a series $X = (0, 2, 2, 2, 2, 7, 7, 7)$ has the maximum value 7. The value 7 needs a bit width of 3 after eliminating leading zeros. Therefore, each value of the series can be efficiently stored using 3 bits with bit-packing encoding. For run-length encoding, the series is represented by three pairs: [1,0], [4,2], and [3,7]. These denote a single 0, four consecutive values 2s, and three consecutive values 7s, respectively.

However, when values are very large with few leading zeros or have low repeatability, BPE and RLE perform poorly on compression. Taking four values $(10791147, 10792951, 10803331, 10819218)$ of a series as an example in Figure 1, the bit width of the maximum value 10819218 is 24 bits. Notably, the series has already been normalized by subtracting its minimum value of the series, resulting in a zero within the series. Consequently, these four values cannot be further compressed through additional minimum subtraction and bit-packing. Thus, the cost of the series after bit-packing encoding is 96 bits. Since there are no repeated values, run-length encoding requires only 4 additional bits to store the run length "1" for each of the four values. Consequently, while only BPE or RLE still requires substantial bit storage for compressed series due to large values and low repeatability, we introduce a sub-column operation to enhance compressibility. By dividing the binary representations of these four values into 3-bit groups (sub-columns). we enable to enhance RLE and BPE by improving leading zeros and repeatability. Subsequently, for each sub-column, we dynamically select RLE or BPE based on their respective storage costs. For the fifth and sixth sub-columns BPE can reduce storage cost to 16 bits by removing leading zeros, while RLE compresses the seventh and eighth sub-columns into 6 bits each (3 bits for run length and 3 bits for value). The sub-column approach achieves a total of 76 bits, outperforming only BPE or RLE.

To find the optimal bit width could improve compression ratio of sub-columns, but it takes too much time to traverse all possible bit width from 1 to 64, and determine whether to use BPE or RLE to compress each sub-column, which could not be acceptable for compression algorithm. Then, it is challenging to take less time to find the optimal bit width to compress

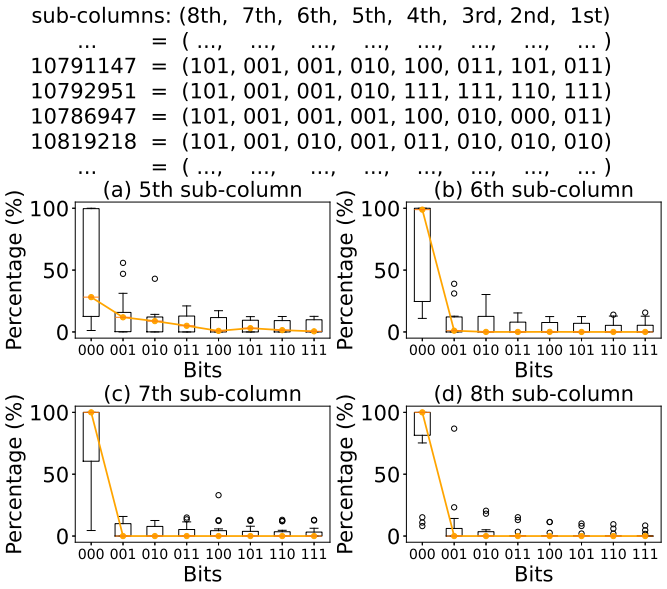


Fig. 1: The distribution of 3-bits data, in the 5th to 8th sub-columns of values from all real world datasets in Table II.

*Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

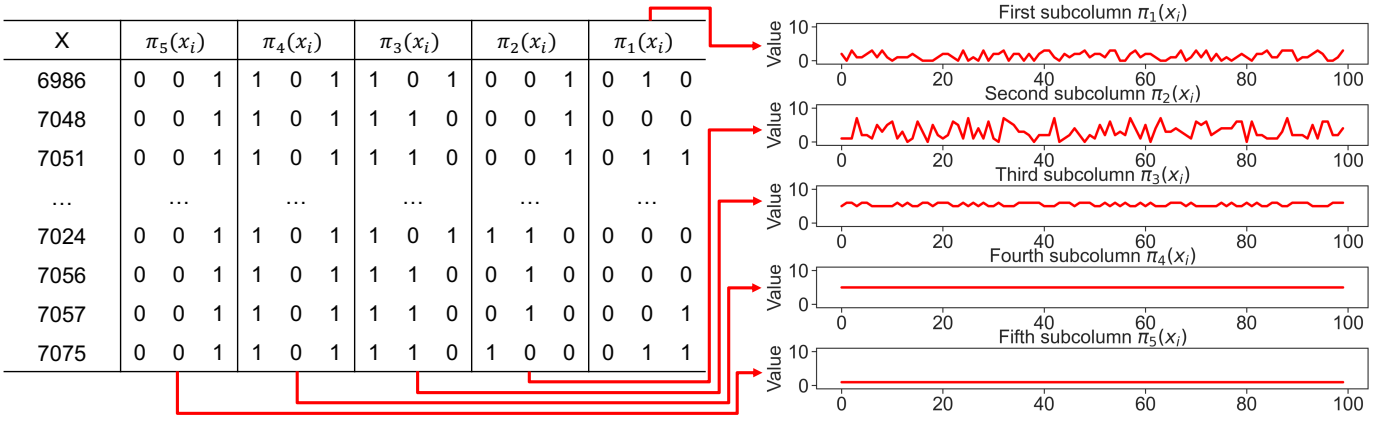


Fig. 2: Sub-column representation of a series in dataset Dewpoint-temp.

data.

Then, the optimal bit width β for sub-columns requires careful consideration and the underlying trade-offs: while larger β may reduce run-length to add the storage cost, smaller β could lead to excessive sub-columns and increased meta data storage overhead of sub-columns. Therefore, we need to propose a solution to determine the optimal bit width of sub-columns with smallest storage cost.

A. Compression with Sub-columns

From an intuitive perspective, despite the presence of numerous large values and low repeated values in the datasets, we can divide the binary representation of the values into multiple sub-columns. And these sub-columns do not all demonstrate uniform distribution. This division bring more leading zeros and value repetitions of sub-columns, thereby enhancing the effectiveness of both bit-packing and run-length encoding techniques.

We study the distribution of sub-columns with 3-bit width in real world datasets, and present the distribution of the fifth to eighth sub-columns in Figure 1. Specifically, the fifth sub-column generally displays skewed distributions and has many leading zeros in these datasets, making bit-packing encoding a more suitable compression method for these sub-columns. Sub-columns with more leading zeros could be effectively compressed by bit-packing encoding, thereby optimizing storage of values.

Likewise, the seventh and eighth sub-columns predominantly contain one or two distinct 3-bit data across most datasets. For these sub-columns, run-length encoding can be effectively applied to compress the data by storing only the run-lengths of consecutive repeated values.

B. Query with Sub-columns

Compressing values in sub-columns not only reduces the storage cost, but also could be beneficial to the query processing. Intuitively, a query executor could filter values at the sub-columns level, thus reducing the number of values that need to be processed and enhancing overall query performance.

In BUFF [27], each input value is divided into 8-bit (i.e., one-byte) sub-columns to optimize query processing by skipping data. However, BUFF does not exploit query optimization opportunities provided by run-length encoding for sub-columns. Notably, run-length encoding (RLE) optimizes range filtering performance by avoiding full scans of sub-columns containing repeated values. Simultaneously, RLE enhances aggregation efficiency for operations such as SUM and COUNT by using run length.

C. Contribution

In the paper, we propose a Sub-column algorithm for data compression and query processing, including but not limited to bit-packing and run-length encoding. Our major contributions are as follows.

(1) We formalize the optimization problem of determining sub-columns for data compression in Section II. We formally define how the sub-column storage cost determination problem depends on the bit widths of the sub-columns.

(2) We propose the optimal solution of sub-column determination problem in Section III. Our method efficiently computes the optimal sub-column bit width for minimal storage cost by employing incremental BPE computation and strategic pruning of both BPE and RLE candidates.

(3) We introduce a novel query optimization technique that exploits BPE and RLE compressed sub-columns to accelerate both range filtering and aggregation operations in Section IV. The proposed technique optimizes query in two key ways: (1) accelerating range filtering through compression-aware data skipping and direct predicate evaluation on compressed sub-columns, and (2) enhancing aggregation performance by exploiting run-length in sub-columns compressed by RLE.

(4) We perform extensive experiments on real world datasets in Section VI. Our algorithm achieves superior compression performance and query efficiency compared to state-of-the-art methods, demonstrating 10-60% higher compression ratio, and 30-50% faster query processing across various datasets.

Our sub-column determination framework has been implemented in the file format Apache TsFile [48] to reduce the

TABLE I: Notations

Notation	Description
X,n	a series, the number of values in the series
M	bit width of the maximum value in series
$\pi_j(x_i)$	the j-th sub-column of the i-th values in series
m	the number of sub-columns
r_s	the s-th run-length of values after RLE
α	the cumulative count of values after RLE
l	the number of run-lengths in a series
l_j	the number of run-lengths in j-th sub-column
R	the run-length series of a series
$R(j)$	the run-length series of the j-th sub-column
$BPE(X, j, \beta)$	the sub-column cost of BPE with bit width β
$RLE(X, j, \beta)$	the sub-column cost of RLE with bit width β
$C(X, \beta)$	the storage cost of sub-columns with β
$\#, A$	the operator of range filtering and aggregation

storage cost, and in the database engine Apache IoTDB [42] for query optimization. The implementation of the framework is available in the official GitHub repositories of Apache IoTDB [19] and Apache TsFile [41]. The experiment related code and data are available in [40] for reproducibility.

II. Problem Statement

In this section, we provide a detailed explanation of the problem statement. In Section II-A, we formalize the definitions of bit-packing (BPE) and run-length encoding (RLE), providing a foundational understanding of these compression techniques. Section II-B introduces the sub-column representation of a series, our proposed approach for optimizing data compression. Finally, Section II-C formalizes the definition of sub-column determination problem. Table I presents the commonly used notations.

A. Bit-packing and Run-length Encoding

Bit-packing encoding [25] assigns a fixed bit width to all values in a series, ensuring consistent representation. The resulting storage cost could be calculated as follows.

Definition II.1 (Storage Cost of Bit-packing Encoding)**.** *For a series $X = (x_1, \ldots, x_n)$, its storage cost of BPE is*

$$BPE(X) = \lceil \log(x_{\max} - x_{\min} + 1) \rceil n \tag{1}$$

where $x_{\max} = \max X$ and $x_{\min} = \min X$ are the maximum and minimum values in series X. By default, the logarithm is in base 2.

In real-world datasets, the changes of values are relatively small in a series, leading to many consecutive repeat values. RLE performs better when series has vast consecutive repeats [43], thus RLE has better compression ratio for bits.

We define the run-length series of X as

$$R = (r_1, r_2, \ldots, r_s, \ldots, r_l)$$

having

$$x_\alpha \neq x_{\alpha+1}, x_{\alpha+1} \eqdots x_{\alpha+r_s}, x_{\alpha+r_s} \neq x_{\alpha+r_s+1},$$

where cumulative consecutive count α is $\sum_{i=1}^{s-1} r_i$ and l is the number of [run-length, value] pair. According to the run-length series R, we could get the storage cost of series compressed by run-length encoding.

Definition II.2 (Storage Cost of Run-length Encoding)**.** *For the given series X, the storage cost of RLE is,*

$$RLE(X) = l(\lceil \log(x_{\max} - x_{\min} + 1) \rceil + \lceil \log(\max r + 1) \rceil),$$

where l is the number of run-lengths.

B. Sub-column Representation

A series can be divided into multiple sub-columns, each with a bit width β, allowing further compression using bit-packing or run-length encoding. We define the sub-columns for series as follows.

Definition II.3 (Sub-column for Series)**.** *For the given series $X = (x_1, \ldots, x_i, \ldots, x_n)$ and the bit width β of sub-columns, the j-th sub-column is $\pi_j(X)_\beta$, and the j-th sub-column value of x_i is $\pi_j(x_i)_\beta$. Therefore, x_i could be represented as follows,*

$$x_i = (\pi_m(x_i), \ldots, \pi_j(x_i), \ldots, \pi_1(x_i))_\beta,$$

where $m = \lceil \frac{\lceil \log(\max x_i) \rceil}{\beta} \rceil$.

Example II.1. *As shown in Figure 2, the series could be split into five sub-columns, each with a 3-bit width. Taking* 6986 *as an example, its binary representation is* (1101101001010) *and the sub-column representation is* $(001, 101, 101, 001, 010)_3$ *according to Definition II.3. Among them, the sub-column $\pi_1(x_i)$ can be compressed from 3 bits to 2 bits by bit-packing encoding.*

For sub-columns with RLE, we define the consecutive repeat series of the j-th sub-column $\pi_j(x_i)$ as follows.

Definition II.4 (Run-length Series of Sub-column)**.** *The run-length series $R(j)$ of the j-th sub-column is,*

$$R(j) = (r_1, r_2, ..., r_{l_j}),$$

having

$$\pi_j(x_\alpha) \neq \pi_j(x_{\alpha+1}), \pi_j(x_{\alpha+1}) \eqdots \pi_j(x_{\alpha+r_s}),$$
$$\pi_j(x_{\alpha+r_s}) \neq \pi_j(x_{\alpha+r_s+1}),$$

where the cumulative consecutive count α is $\sum_{i=1}^{s-1} r_i$ and l_j is the number of run-lengths of the j-th sub-column.

Example II.2. *In Figure 2, $\pi_4(x_i)_3$ of all the values is the same in the series, consisting of 100 instances of '101'. Thus, the run-length series R(4) of the fourth sub-columns are* (1100100) *in binary representation (decimal: 100), and l_4 are 1. Then, we store the [run-length, value] pair as [1100100,101] for the fourth sub-columns compressed by RLE.*

While we consider integer values by default, float or double values can be naturally supported by scaling. That is, we convert float or double values into integer by scaling 10^p, where p is the precision of the original floating-point series.

you rking on E+BPE? s or no?

O3

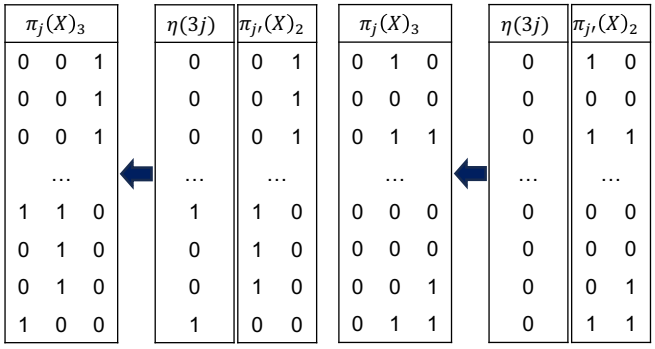
R1D13

β is 3?[yes] ¿explain it in the example

rephrase this paragrap While w consider integer values by

	$\pi_5(x_i)$	$\pi_4(x_i)$	$\pi_3(x_i)$	$\pi_2(x_i)$	$\pi_1(x_i)$
RLE/BPE	RLE	RLE	BPE	BPE	BPE
6986			1 0 1	0 0 1	1 0
7048			1 1 0	0 0 1	0 0
7051			1 1 0	0 0 1	1 1
...	[1100100,	[1100100,	...	...	...
7024	001]	101]	1 0 1	1 1 0	0 0
7056			1 1 0	0 1 0	0 0
7057			1 1 0	0 1 0	0 1
7075			1 1 0	1 0 0	1 1

Fig. 3: Sub-column representation of the example series in Figure 2 with bit-packing and run-length encoding.



(a) Case 1: $\eta(\beta j) = 1$ (b) Case 2: $\eta(\beta j) = 0$

Fig. 4: Storage cost with BPE for a 3-bit sub-column can be incrementally derived from its 2-bit sub-column and a single-bit sub-column, as indicated by arrows.

C. Sub-column Storage Cost

According to Definition II.3, we formally define storage cost of sub-columns with bit-packing and run-length encoding as follows.

Definition II.5 (Sub-column Cost with Bit-packing Encoding)**.** *For the given series X and the bit width of sub-columns β, the j-th sub-column cost with bit-packing is,*

$$BPE(X, j, \beta) = \lceil \log(\max_{1 \le i \le n} \pi_j(x_i)_\beta + 1) \rceil n,$$

where $\pi_j(x_i)_\beta$ is the j-th sub-column of i-th value x_i in the series.

According to Definition II.5, the storage cost $BPE(X, 1, 3)$ of the first sub-column $\pi_1(x_i)$ is 16 bits in Figure 2. Likewise, we could calculate the storage cost of sub-column compressed by run-length encoding as follows.

Definition II.6 (Sub-column Cost with Run-length Encoding)**.** *For the given series X and the bit width of sub-columns β, the j-th sub-column cost of run-length encoding is,*

$$RLE(X, j, \beta) = l_j(\beta + \lceil \log(n + 1) \rceil).$$

According to Definition II.6, we could calculate that the storage cost $RLE(X, 4, 3)$ of the fourth sub-columns $\pi_4(x_i)$ with RLE is 10 bits. And $RLE(X, 5, 3)$ is also 10 bits as illustrated in Figure 3. By comparing sub-column cost of BPE and RLE, we could get j-th sub-column cost $C(X, j, \beta) = \min\{BPE(X, j, \beta), RLE(X, j, \beta)\}$. Then, the storage cost of all sub-columns is as follows.

Definition II.7 (Sub-column Cost)**.** *The sub-column storage cost $C(X, \beta)$ of series X is the sum of each sub-column cost,*

$$C(X, \beta) = \sum_{j=1}^{m} (\min\{BPE(X, j, \beta), RLE(X, j, \beta)\}).$$

Example II.3. *For the series in Figure 3, we could conclude that the better compressions of all the sub-columns are RLE, RLE, BPE, BPE and BPE, and their storage cost are 10, 10, 300, 300 and 200 bits. Thus, the total sub-column cost is 820 bits, which presents a 37% reduction compared to the storage cost 1300 bits of bit-packing encoding.*

According to the storage cost $C(X, \beta)$ of sub-columns, we could define the sub-columns determination problem in Definition II.8.

Definition II.8 (Sub-column Determination Problem)**.** *For a given series X, the sub-columns determination problem is to determine the sub-column bit width β with minimal sub-column cost,*

$$\arg\min_{\beta} C(X, \beta).$$

III. Sub-column Determination

To determine the optimal solution of sub-columns determination problem, a straightforward idea is to traverse each possible β and calculate sub-column cost in Definition II.7. In this section, we propose an algorithm to reduce the computation time to identify the optimal β. To optimize time of calculating sub-column storage cost with BPE, we describe incremental computation of bit-packing encoding in Section III-A. Then, we introduce the method of pruning bit-packing encoding and run-length encoding in Sections III-B and III-C. Finally, we propose our sub-columns determination algorithm and analyze its complexity in Section III-D.

A. Incremental Computation of Bit-packing

For sub-column cost with BPE, we firstly calculate whether every 1-bit sub-column is all 0s with $\beta = 1$, i.e., $\eta(j) = \max_{1 \le i \le n} \pi_j(x_i)_1$ is 0 or 1. According to the array $\eta(j)$ and the following proposition, we could incrementally compute the bit width of each sub-column with other β for bit-packing encoding.

Proposition 1. *For bit width $b_\beta(j)$ of j-th sub-column $\pi_j(x_i)_\beta$ after bit-packing, we could get*

$$b_\beta(j) = b_{\beta-1}(j') + \eta(\theta),$$

R1D14

restate to reflect th relations of sect 3.1-3.3, how the previous used in t below

R2O5

R4O3

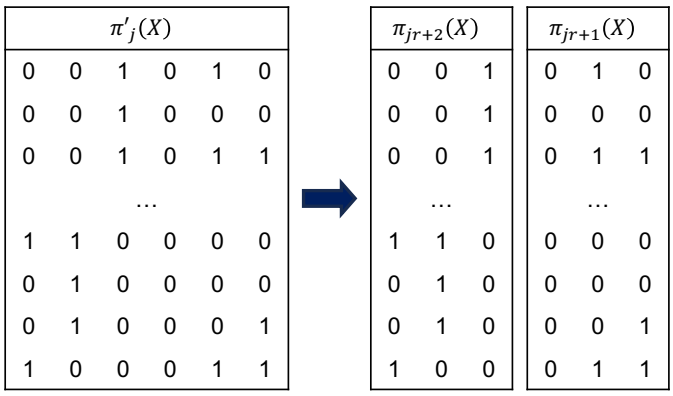


Fig. 5: The left sub-column (with $\beta' = 6$) could be decomposed into the right two sub-columns (each with $\beta = 3$), indicated by the arrow. And the storage cost of the right two sub-columns (500 bits) is smaller than that the left sub-column (600 bits) compressed by BPE.

where $\theta = \beta j$ and $j' = \frac{\theta}{\beta-1}$.

Proof. When j' is $\frac{\theta}{\beta-1}$, we could conclude that $(\beta-1)j'$ equals to θ and j'-th sub-column (with $\beta-1$) is the front $(\beta-1)$ bits of j-th sub-column. If $\eta(\theta)$ is 0, i.e., θ-th bits in all the values are 0, the bit width of j-th sub-column is that of j'-th sub-column (with $\beta-1$) after bit-packing encoding. □

Intuitively, the β-bit sub-column cost with BPE can be incrementally derived from the (β-1)-bit sub-column and a single-bit sub-column.

Example III.1. *Taking the case in Figure 4a as an example, the bit width $b_3(j)$ of the BPE compressed sub-column $\pi_j(x_i)_\beta$ is 3 bits, which is $b_{\beta-1}(j') + \eta(\theta)$ with $\beta = 3$. But for the case in Figure 4b, bit width $b_3(j)$ of the sub-column $\pi_j(x_i)_\beta$ compressed by BPE is 2 bits, since $\eta(\theta)$ is 0.*

B. Pruning Bit-packing Encoding

For bit-packing encoding, the relationship between the storage cost of different bit widths is outlined in the below proposition.

Proposition 2. *For the given X, if β' is divisible by β, the sub-column cost with bit-packing of β is smaller,*

$$BPE(X, j, \beta') \geq \sum_{k=1}^{q} BPE(X, jq + k, \beta),$$

where $\beta' = q\beta$, and q is a positive integer.

Proof. For the given series $X = (x_1, \ldots, x_i, \ldots, x_n)$ and the sub-columns with β' are

$$(\pi'_m(X) \ldots \pi'_j(X) \ldots \pi'_1(X))_{\beta'},$$

then $\pi_j(X)$ is $(\pi_{jq}(X)\pi_{jq-1}(X) \ldots \pi_{jq-(q-1)}(X))$. Then, the difference ΔC between the sub-column cost with bit-packing encoding of β and β' is,

$$\begin{aligned}\Delta C &= (BPE(X, j, \beta') - \sum_{k=1}^{q} BPE(X, jq + k, \beta))n \\ &= ((\lceil \log(\max_{1 \leq i \leq n} \pi'_j(x_i) + 1) \rceil) \\ &\quad - \sum_{k=1}^{q} (\lceil \log(\max_{1 \leq i \leq n} \pi_{(j-1)q+k}(x_i) + 1) \rceil))n\end{aligned}$$

If $\max_{1 \leq i \leq n} \pi_{(j-1)q+l_j}(x_i)$ is 0 and $\max_{1 \leq i \leq n} \pi_{(j-1)q+h_j}(x_i)$ is not 0 with $l_j > h_j$, we could conclude $\lceil \log(\max_{1 \leq i \leq n} \pi'_j(x_i) + 1) \rceil = h_j\beta$ and $\sum_{k=1}^{j} (\lceil \log(\max_{1 \leq i \leq n} \pi_{j(q-1)+k}(x_i) + 1) \rceil) \leq h_j\beta$.

Therefore, it can be inferred that

$$\Delta C \geq (h_j\beta - h_j\beta)n = 0.$$

□

As depicted in Figure 5, the storage cost with BPE of these two 3-bit-width sub-columns is less than the sub-column with bit width 6. To evaluate the storage cost of the j-th sub-column, $C(X, j, \beta)$, we have to compare sub-column cost with RLE with BPE. To improve the efficiency of determining bit width β, the algorithm could prune some comparison process following Proposition 2. When bit width β of sub-column is a composite number, we only need to compare the sum of sub-column cost with BPE (i.e., $\sum_{k=1}^{q} BPE(X, jq + k, \frac{\beta}{q})$) to sub-column cost with the run-length encoding.

C. Pruning Run-length Encoding

When the bit width β of a sub-column is a prime number, we can further prune the run-length encoding computation. The following proposition demonstrates that this pruning is achievable by using a threshold for the number of run-lengths (l_j) in the j-th sub-column.

Proposition 3. *If the number of run-lengths of j-th sub-column satisfies $l_j \geq \frac{BPE(X,j,\beta)}{\beta + \lceil \log(n+1) \rceil}$, we can infer that*

$$RLE(X, j, \beta) \geq BPE(X, j, \beta).$$

Proof. According to $l_j \geq \frac{BPE(X,j,\beta)}{\beta + \lceil \log(n+1) \rceil}$ and Definition II.6, we could infer that

$$\begin{aligned}RLE(X, j, \beta) &= l_j(\beta + \lceil \log(n + 1) \rceil) \\ &\geq BPE(X, j, \beta).\end{aligned}$$

□

When we calculate the storage cost with run-length encoding, the variable l_j is counted up sequentially from 1. Hence, when $l_j \geq \frac{BPE(X,j,\beta)}{\beta + \lceil \log(n+1) \rceil}$ holds, we stop the processing of counting and the storage cost $C(X, j, \beta)$ of j-th sub-columns is that of bit-packing encoding according to Proposition 3.

Algorithm 1: Sub-column Determination

Input: Series $X = (x_1, x_2, \ldots, x_n)$
Output: Bit width β'

1 $x_{\max} = \max_{1 \le i \le n} x_i$;
2 $x_{\min} = \min_{1 \le i \le n} x_i$;
3 $C_{\min} = (\lceil \log(x_{\max} - x_{\min} + 1) \rceil) n$;
4 Calculate η_j;
5 $\beta = 1$;
6 **while** $\beta \le \lceil \log(x_{\max} - x_{\min} + 1) \rceil$ **do**
7 $j = 1$;
8 **while** $j \le \frac{\lceil \log(x_{\max} - x_{\min} + 1) \rceil}{\beta}$ **do**
9 **if** β *is a composite number* **then**
10 $BPECost = \sum_{k=1}^{r} BPE(X, jr + k, \frac{\beta}{r})\}$;
11 **if** $RLE(X, j, \beta) < BPECost$ **then**
12 $C(X, j, \beta) = RLE(X, j, \beta)$ with Proposition 2;
13 **else**
14 $C(X, j, \beta) = \min\{RLE(X, j, \beta), BPE(X, j, \beta)\}$;
15 **else**
16 Calculate $BPE(X, j, \beta)$ by Proposition 1;
17 **if** $l_j \ge \frac{BPE(X,j,\beta)}{\beta + \lceil \log(n+1) \rceil}$ **then**
18 $C(X, j, \beta) = BPE(X, j, \beta)$;
19 **else**
20 $C(X, j, \beta) = RLE(X, j, \beta)$ by Proposition 3;
21 $C(X, \beta) = C(X, \beta) + C(X, j, \beta)$;
22 $j = j + 1$;
23 **if** $C(X, \beta) < C_{\min}$ **then**
24 $\beta' = \beta$;
25 $C_{\min} = C(X, \beta)$;
26 β += 1;
27 **return** β';

D. Sub-column Determination Algorithm

Algorithm 1 presents the pseudo code of sub-columns for data compression. In Lines 1-3, the algorithm could get the storage cost of bit-packing with the maximum and minimum of the series. Then, with Proposition 1, the algorithm calculates η_j in Line 4, i.e., whether the j-th sub-column is all zeros with $\beta = 1$. Furthermore, we enumerate each β to calculate $C(X, \beta)$, and find the optimal β with the minimum storage cost. According to Propositions 2 and 3, we compare run-length encoding cost to bit-packing cost of each sub-column in Lines 8-21. Finally, the algorithm compares the storage cost of all the sub-columns with the minimum cost and determines the optimal bit width.

In Algorithm 1, the maximum bit width M is a $\lceil \log(x_{\max} - x_{\min} + 1) \rceil$ for a given series. And it takes at most constant M loops to enumerate β. For each β, it takes at most $O(n)$ to calculate the sub-columns cost $C(X, \beta)$. Therefore, the time complexity analysis of the sub-column determination algorithm is $O(n)$.

Algorithm 2: Range Filter Query with Sub-columns

Input: j-th sub-column $\pi_j(X)$, Predicate $\#$, Series $X_\#$, Index $I_\#$
Output: New Series $X'_\#$, Index $I'_\#$

1 **if** *j=m+1* **then**
2 **return** $X, (1, 2, \ldots, n)$;
3 $X_\#$ = Range-Filtering($\pi_{j+1}(X)$, $\#$, $X_\#$,$I_\#$);
4 **if** *Compression of j-th sub-column is BPE* **then**
5 **for** i *in* $I_\#$ **do**
6 $\pi_j(x_i)$ is from the $((i-1)\beta + 1)$-th bit to the $(i\beta)$-th bit;
7 **if** $\pi_j(x_i) = \pi_j(v)$ **then**
8 $I'_\#$ add i;
9 **else if** $\pi_j(x_i) \ \# \ \pi_j(v)$ **then**
10 $X'_\#$ add $(x_i << \beta + \pi_j(x_i))$;
11 **else if** *Compression of j-th sub-column is RLE* **then**
12 **while** $s < k$ *and* $i \in I_\#$ **do**
13 **if** $\alpha + r_s < i$ **then**
14 s += 1;
15 **continue**;
16 **else**
17 **if** $\pi_j(x_\alpha) = \pi_j(v)$ **then**
18 $I'_\#$ add $[\alpha, \alpha + r_s]$;
19 **else if** $\pi_j(x_\alpha) \# \pi_j(v)$ **then**
20 $X'_\#$ add $(x_i << \beta + \pi_j(x_\alpha))$
21 **return** $X'_\#, I'_\#$;

R4O4

Proposition 4. *When the given series X with n values complies with i.i.d. $N(\mu, \sigma^2)$, the compression ratio of Sub-column is less than or equal to* $\frac{64}{\lceil \log(\mathbb{E}[x_{\max}] - \mathbb{E}[x_{\min}] + 1) \rceil}$.

Proof. When the given series X with n values complies with i.i.d. $N(\mu, \sigma^2)$ [16], the expected minimum and maximum values are,

$$\mathbb{E}[x_{\max}] = \mu + \sigma \left[b_n + \frac{\gamma}{b_n} + o\left(\frac{1}{b_n}\right) \right],$$

and

$$\mathbb{E}[x_{\min}] = \mu - \sigma \left[b_n + \frac{\gamma}{b_n} + o\left(\frac{1}{b_n}\right) \right],$$

where b_n is $\sqrt{2 \ln n} - \frac{\ln \ln n + \ln(4\pi)}{2\sqrt{2 \ln n}}$. Then compression ratio is less than or equal to $\frac{64}{\lceil \log(\mathbb{E}[x_{\max}] - \mathbb{E}[x_{\min}] + 1) \rceil}$. □

IV. Query Processing with Sub-columns

In addition to the compression benefits, sub-column determination algorithm could also enhance the query execution. In the section, we redesign the query algorithm for both range filtering and aggregation to leverage sub-column compression.

A. Range Filtering

The algebra of range filtering with sub-columns could be written as follows.

$$
\begin{aligned}
&\sigma_{x_i \# v}(X) \\
&\iff (\sigma_{X.m \# v.m}(\pi_m(X)) \bowtie \pi_{m-1}(X) \bowtie \ldots \bowtie \pi_1(X)) \\
&\quad \cup (\sigma_{X.m=v.m}(\pi_m(X)) \bowtie \sigma_{X.m-1 \# v.m-1}(\pi_{m-1}(X)) \\
&\quad\quad \bowtie \ldots \bowtie \pi_1(X)) \cup \ldots \\
&\quad \cup (\sigma_{X.m=v.m}(\pi_m(X)) \bowtie \sigma_{X.m-1=v.m-1}(\pi_{m-1}(X)) \\
&\quad\quad \bowtie \ldots \bowtie \sigma_{X.2=v.2}(\pi_2(X)) \bowtie \sigma_{X.1 \# v.1}(\pi_1(X))),
\end{aligned}
$$

where the operator of range filtering $\#$ is one of $\{>, <, =, \leq, \geq, \in\}$.

The processing of range filtering based on sub-columns is as follows. First, we select the values whose m-th sub-column is larger than the m-th sub-column of v. Then, we examine the $(m-1)$-th sub-column of candidate values, i.e., $\sigma_{X.m=v.m}(\pi_m(X))$. If the $(m-1)$-th sub-column is compressed by bit-packing encoding, we could directly access the position of $(m-1)$-th sub-columns by the index of candidate values. If the $(m-1)$-th sub-column is compressed by RLE, we could get multiple candidate values once according to the cumulative count.

We rewrite the range filtering query in Algorithm 2. In line 3, the code recursively calls the range filtering function. It performs range filtering sequentially from the m-th sub-column down to the first sub-column, continuing until there are no remaining values to filter. If the j-th sub-column is compressed by BPE, we could avoid decompressing all values and wasting query time by directly accessing the $(i\beta - \beta + 1)$-th bit to retrieve $\pi_j(x_i)$ in Line 6 of Algorithm 2. If the j-th sub-column is compressed by RLE, the algorithm finds the cumulative count of $\pi_j(x_i)$ in the candidate index sets $I_\#$ and simultaneously evaluates r_p instances of $\pi_j(x_i)$ in Lines 12-15. If $\pi_j(x_i)$ equals $\pi_j(v)$, we put i into $I'_\#$ to compare x_i with v using the $(j-1)$-th sub-column. When $\pi_j(x_i) \# \pi_j(v)$ satisfies, we add x_i to the filtered values set $X'_\#$. For multi-predicates on a column (e.g. where $X < a$ and $X > b$), we could replace '#' with '$\in$ (a,b)' in Algorithm 2.

B. Materialization

For multi-predicate range filters on multi-columns (e.g., WHERE $X < a$ AND $Y < b$), the indices retrieved from $X < a$ are passed to the Y column, and indices are further filtered using $Y < b$.

C. Aggregation

We also optimize aggregations on values compressed by sub-column determination algorithm. Algorithm 3 describes how the aggregation query is executed based on sub-columns. Similar to Algorithm 2, the algorithm recursively calls the aggregation function, and it performs aggregation sequentially from the m-th sub-column down to the first sub-column. In Lines 8-12 and Lines 19-23, the algorithm performs aggregation calculation on $\pi_j(x_i)$.

Algorithm 3: Aggregation Query with Sub-columns

Input: j-th sub-column $\pi_j(X)$, Aggregation Operation A, Filtered Series X_A, Aggregation x_A
Output: New Aggregation x_A

```
1  if j = m + 1 then
2      return x_A;
3  x_A = Aggregation(π_{j+1}(X), A, X_A, x_A);
4  x'_A = 0;
5  if The compression of j-th sub-column is BPE then
6      for i in X_A do
7          π_j(x_i) is from the ((i − 1)β + 1)-th bit to the
             (iβ)-th bit in π_j(X);
8          if A = Max then
9              x'_A = max{π_j(x_i), x'_A};
10             Update X'_A;
11         else if A = Sum then
12             x'_A = x'_A + π_j(x_i);
13 else if The compression of j-th sub-column is RLE
     then
14     while s < k and i < n do
15         if α < i then
16             s += 1;
17             continue;
18         else
19             if A = Max then
20                 x'_A = max{π_j(x_α), x'_A};
21                 Update X'_A with [α, α + r_s];
22             else if A = Sum then
23                 x'_A = x'_A + π_j(x_α) r_s;
24         i += 1;
25 x_A = x_A << β;
26 x_A = x_A + x'_A;
27 return x_A;
```

Besides the query optimization on data compressed by bit-packing encoding in Line 7, run-length encoding can further improve the performance of sum queries. Specifically, RLE-compressed sub-columns improve SUM efficiency by performing a single multiplication (run-length r_p * value) instead of r_p additions, as shown in Line 23.

V. System Deployment

We implement our sub-column determination algorithm in Apache TsFile [48], [6] and Apache IoTDB [42], [4]. In Section V-A, we describe the storage deployment of data compressed by our sub-columns determination algorithm in Apache TsFile, which is a columnar storage file format designed for time series data with efficient compression. We further explain the query processing based on our algorithm in Apache IoTDB, which is an IoT native database with high performance for data management and analysis.

A. Storage Layout in Apache TsFile

Apache TsFile organizes data into chunk groups containing device time series data, each with multiple chunks for indi-

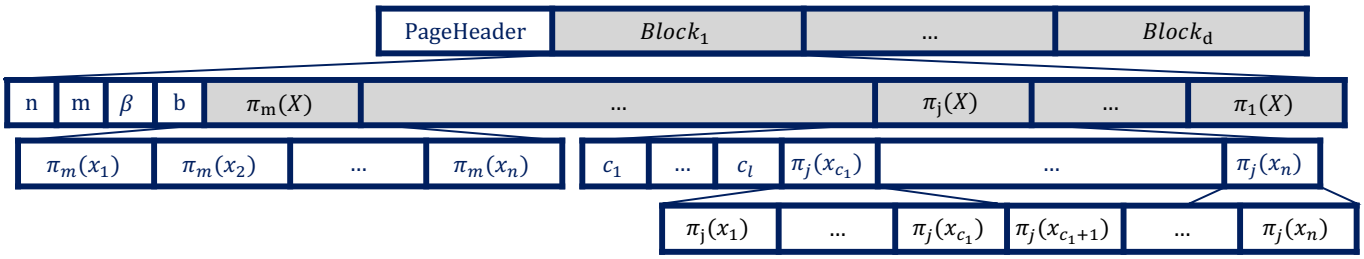


Fig. 6: Deployment of sub-columns in Apache TsFile

vidual sensors. Chunks consist of headers and pages, where pages are the basic serialization units. In each page, as shown in Figure 6, we divide the data into blocks. In each block, we use our algorithm to compress and store data.

The section outlines the storage layout of data block compressed by the algorithm in the TsFile system. Firstly, the compressed data stores several meta data of the series, including the number of values n, the bit width of the maximum value m, and the bit width β of sub-columns in Figure 6. Then, we store the actual bit widths b of each sub-column after bit-packing (BPE) or run-length encoding (RLE). Finally, the algorithm stores each sub-column compressed by BPE or RLE. For the sub-column compressed by RLE, the algorithm stores the cumulative count α of run-length and values x_α.

B. Query Processing in Apache IoTDB

In this section, we will introduce how to improve query processing including range filtering, count, max/min and sum by using Algorithms 2 and 3 implemented in Apache IoTDB.

1) Range Filtering: Sub-columns can improve query speed of the following SQL statement about range filtering with Algorithm 2.

```
IoTDB > select X from root where X # v;
```

Example V.1. *Taking an example as follows, we execute the following statement in the series of Figure 3.*

```
IoTDB > select X from root where X > 7046;
```

By executing m-th and $(m-1)$-th sub-columns predicates $\sigma_{X.m>v.m}(X)$ and $\sigma_{X.m-1>v.m-1}(X)$, we could not filter any values where m is 5. Then, according to $\sigma_{X.3>v.3}(X)$, we could filter out all other values, keeping only five: 7048, 7051, 7056, 7057, and 7075. Finally, we could get 7075 with range filtering of the second sub-column, i.e., the second sub-column of value 7075 is larger than that of 7046.

2) Count: For the given series $X = (x_1, \ldots, x_i, \ldots, x_n)$, we could improve the query executor to calculate the count $COUNT(X)$ in the series with Algorithm 3.

```
IoTDB > select COUNT(X) from root;
```

For a column X with no NULL values, the $COUNT(X)$ operation can be implemented by directly returning the count n stored within the compressed data.

3) Max and Min: For the given series $X = (x_1, \ldots, x_i, \ldots, x_n)$ and range filtering $X \# v$, the query statement to find the maximum $x_{\max}$ (similarly for minimum value $x_{\min}$) is presented below.

```
IoTDB > select MAX(X) from root where X # v;
```

The aggregation predicate $MAX(X)$, similarly for $MIN(X)$, could be decomposed as:

$$\max(X) \iff \gamma_{\max(1)}(\pi_1(X) \bowtie \ldots \bowtie \gamma_{\max(m-1)}(\pi_{m-1}(X) \bowtie \gamma_{\max(m)}(\pi_m(X)))).$$

Starting with the m-th sub-column, the query executor finds several values with the greatest/smallest m-th sub-column in the values with $X.m \# v.m$. The query executor subsequently processes the $(m-1)$-th sub-column of these values. It continues to identify the maximum or minimum value until only a single value remains or the first sub-column has been fully processed. During the determination of the maximum or minimum value, each sub-column incrementally narrows the search and decompression range and reduces query time. And Lines 8-10 and Lines 19-21 of Algorithm 3 describe the procedure for finding the maximum value.

4) Sum: For the given series $X = (x_1, \ldots, x_i, \ldots, x_n)$ and range filtering $X \# v$, the query statement of the sum $SUM(X)$ of the series after range filtering is described below.

```
IoTDB > select SUM(X) from root where X # v;
```

After Algorithm 2, we could calculate the sum of each sub-column of values filtered by $X \# v$. Note that for the sub-column compressed by RLE, the sum is calculated by multiplying run-length and values, which reduces the time required for summation. The processing of calculating the sum is presented in Lines 11-12 and Lines 22-23 of Algorithm 3.

VI. Experiment

In this section, we conduct experiments about our sub-column determination algorithm. In Section VI-A, we introduce baselines, real-world datasets and metrics of the experiments. In Section VI-B, we experimentally compare the compression ratio and throughput of our sub-column determination algorithm with other existing algorithms. In Section VI-C, we present the compression performance of our algorithm varying bit width and block size, and we conduct the evaluation about query time in Section VI-D.

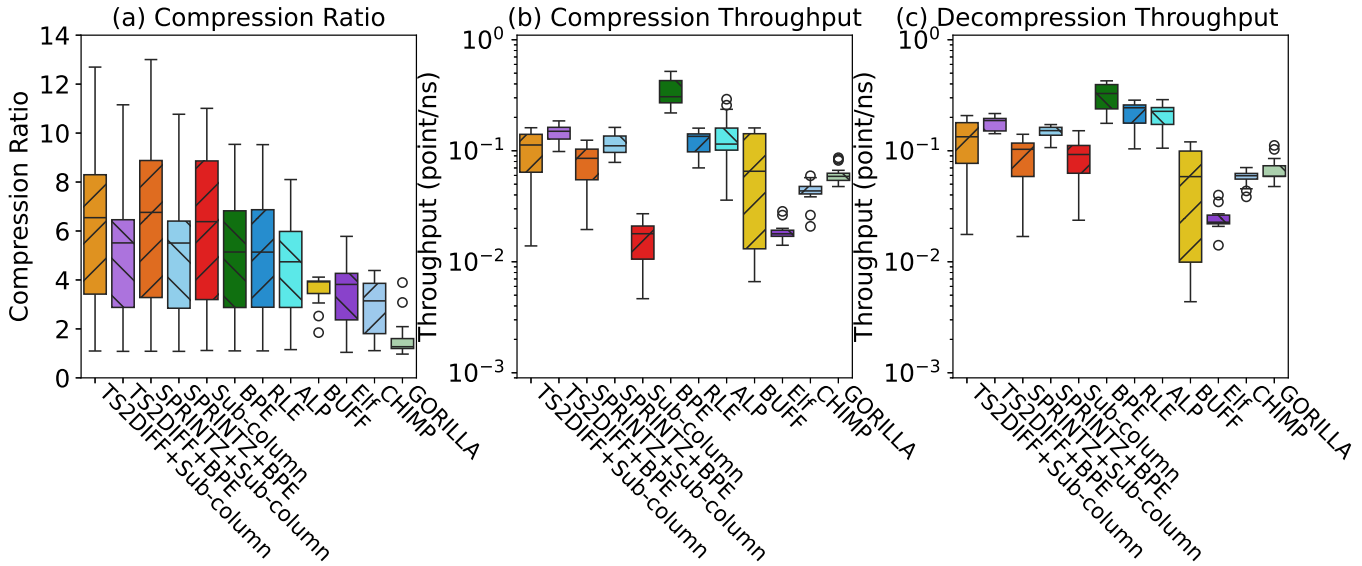


Fig. 7: Compression performance on various datasets, where our sub-column technique can work together with other methods

TABLE II: Real-world datasets

Dataset	Abbr.	Source	# Values	Type	Bit width
Arade	Arade	[7]	9,888,775	Float	25 bits
Bird-migration	BM	[18]	17,964	Float	23 bits
Bitcoin-price	BP	[18]	2,741	Float	33 bits
City-temp	CT	[34]	2,905,887	Float	11 bits
Dewpoint-temp	DT	[33]	5,413,914	Float	13 bits
EPM-Education	EE	[13]	900,000	Integer	31 bits
Government	Gov	[7]	141,123,827	Float	35 bits
IR-bio-temp	IR	[32]	380,817,839	Float	11 bits
PM10-dust	PM10	[31]	222,911	Float	18 bits
POI-lat	PLAT	[17]	424,205	Float	64 bits
Stocks-DE	SDE	[10]	45,403,710	Float	18 bits
Stocks-UK	SUK	[10]	115,146,731	Float	20 bits
Stocks-USA	SUSA	[10]	374,428,996	Float	15 bits
Wind-speed	WS	[30]	199,570,396	Float	8 bits
Wine-Tasting	WT	[21]	120,975	Integer	16 bits

D9

A. Experimental Setting

The experiments were conducted on an Apple M1 Pro chip with 8 CPU cores,complemented by 16GB of unified memory.

1) Baselines: We compare our sub-column determination algorithm (Sub-column) with the state-of-the-art lossless series compression algorithms, including GORILLA [36], CHIMP [26], Elf [23], BPE [49], RLE [14], SPRINTZ+BPE [8], TS2DIFF+BPE [45], BUFF [27], and ALP [3]. Note that complementary to the existing algorithms, we also replace BPE in SPRINTZ and TS2DIFF with our algorithm, i.e., SPRINTZ+Sub-column and TS2DIFF+Sub-column.

2) Datasets: To verify the performance of sub-columns compression algorithm, we adopt 14 real world datasets encompassing both time-series and non-time-series data, including float and integer values in various scenarios. Table II shows the number of values, data type and bit width of maximum in these datasets. Arade and Government refer to the energy dataset and the government monetary dataset (USD) from the Public BI benchmark. Bird-migration is a dataset tracking animal migration, recording the position of birds. Bitcoin-price contains the exchange rate of Bitcoin against the US dollar and other transaction data. City-temp records the temperatures of major cities around the world. Dewpoint-temp records the relative dew point temperature on rivers and lakes. EPM-Education is built from the recordings through an application while learning with an educational simulator. IR-bio-temp exhibits the changes in the temperature of infrared organisms. POI-lat dataset has the latitude information of the coordinates in radian of Position-of-Interests (POI) extracted from Wikipedia. PM10-dust records near real-time measurements of PM10 particles in the atmosphere. Stocks-UK, Stocks-USA, and Stocks-DE contain the stock exchange prices for the UK, USA, and Germany, respectively. Wind-speed describes wind speed data. Wine-Tasting contains the prices of wines, sourced from around 130,000 reviews published on WineEnthusiast.

R4O5

3) Metrics: We compare compression ratio with other methods, which measures the ratio of compressed data size to uncompressed one, $compressionRatio = \frac{uncompressedSize}{compressedSize}$. We also evaluate the throughput of compression and decompression, measured in the number of value points per CPU nanoseconds (point/ns). Note that each value point is a value consisting of 8 bytes. Meanwhile, we also measured the query performance of different algorithms (measured in CPU time per value point, ns/point). In our experiments, compression refers to the operation of loading and transforming the data into a compressed byte stream, and decompression refers to the full scan and restoration of the compressed data to its original form. Each experiment is conducted 500 times and the average is reported.

R1D10

R1D11

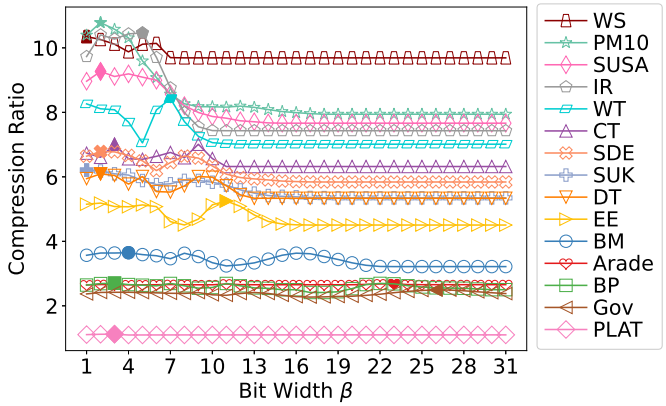


Fig. 8: Compression ratio of our algorithm with various bit widths β of sub-columns

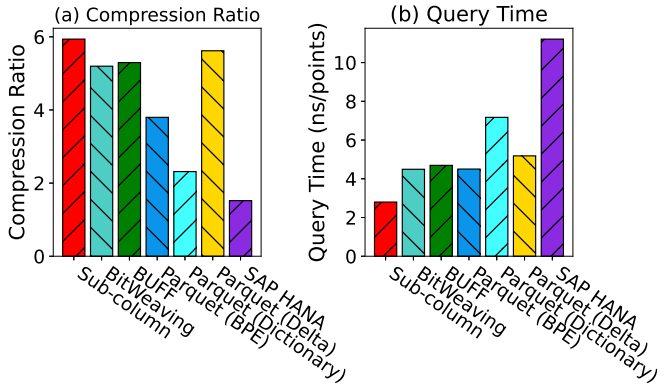


Fig. 9: Compression and query performance of Sub-column and other baselines

B. Compression Evaluation

In the section, we compare the compression performance of our proposal algorithm Sub-column with other baselines in Figure 7.

1) Compression Ratio: Figure 7(a) demonstrates that the sub-column determination algorithm achieves a 20% to 60% higher compression ratio compared to run-length and bit-packing encoding. Our algorithm performs the best on almost all the datasets, except for several datasets (worse than SPRINTZ+BPE and TS2DIFF+BPE). Furthermore, the sub-column determination algorithm could improve the compression ratio of SPRINTZ and TS2DIFF, that is to say, the compression ratio of SPRINTZ+Sub-column and TS2DIFF+Sub-column is better than that of SPRINTZ+BPE and TS2DIFF+BPE on all datasets.

2) Compression and Decompression Throughput: Since bit-packing encoding only removes leading zero of each value, the compression and decompression throughput of bit-packing is the best on all the datasets as depicted in Figures 7(b) and (c). As shown, the sub-column determination algorithm achieves no much worse than compression and decompression throughput of other methods, since it does not take much time to determine β in our algorithm.

3) Trade-off between Compression Ratio and Throughput: Compared to other algorithms, the compression ratio benefits of sub-columns determination algorithm can be categorized into 2 types: 'replacement' and 'comparative' relationships. For the 'replacement' relationship, when sub-columns determination algorithm is replaced BPE in both TS2DIFF and SPRINTZ, the compression throughput increases, but the compression ratio improves in both algorithms. For the 'comparative' relationship, sub-column determination algorithm spends worse throughput than Gorilla, Chimp, and Elf, while also achieving a better ratio.

C. Parameter Evaluation

We evaluate the compression ratio of the sub-column determination algorithm by varying bit width β of sub-columns in this section. The result reported in Figure 8 represents the average compression ratio across all datasets in Table II.

As depicted in Figure 8(a), the optimal bit width β for the compression ratio varies across different datasets, while for most datasets (13 out of 15 datasets), the best β is between 1 and 4 bits. Even if PLAT has large bit width of maximum value, i.e. 64 bits, the optimal bit width is 2 bits. Thus, the algorithm could optimize the compression throughput by only finding the best β between 1 and 4 bits. With respect to Algorithm 1, the loop over β only needs to iterate from 1 to 4 in line 5.

D. Query Evaluation

We perform the query experiment about range filtering, aggregation, materialization and update time in this section.

1) Comparison of Query and Compression Ratio: We compare our algorithm's query time against the query time of Parquet-Select [24], BitWeaving [25] and BUFF [27] in Figure 9.

We evaluate compression ratio of delta encoding and dictionary encoding in Apache Parquet in Figure 9(a). As described in Parquet [5], delta encoding decreases the magnitude of the values to be stored, further adding leading zeros. The compression in dictionary encoding is achieved by replacing each repeated value with a key, which has more leading zeros than the original values. Hence, delta encoding has better compression ratio than Sub-columns on some datasets in Figure 9. But Sub-column has bigger compression ratio by adding more zeros of sub-columns, thus, the compression ratio of Sub-column is better than those of Parquet (Delta) and Parquet (Dictionary) as presented in Figure 9.

The algorithm in SAP HANA for compressing piecewise equidistant tables employs arithmetic encoding of timestamp columns using base values, incremental units, and nanosecond offsets to achieve high compression efficiency while maintaining query performance [37]. Our Sub-column algorithm could also compress timestamp columns with piecewise, and we compare compression and query performance of Sub-column and SAP HANA in Figure 9. Since our Sub-column

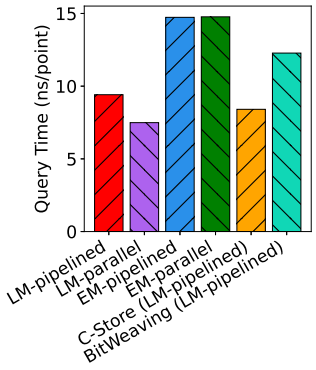


Fig. 10: Impact on query performance of Sub-column algorithm by different tuple materialization strategies

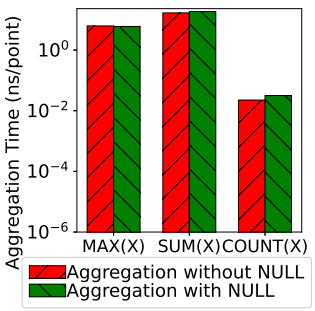


Fig. 11: Aggregation time including $MAX(X)$, $SUM(X)$ and $COUNT(X)$ on columns with NULL values

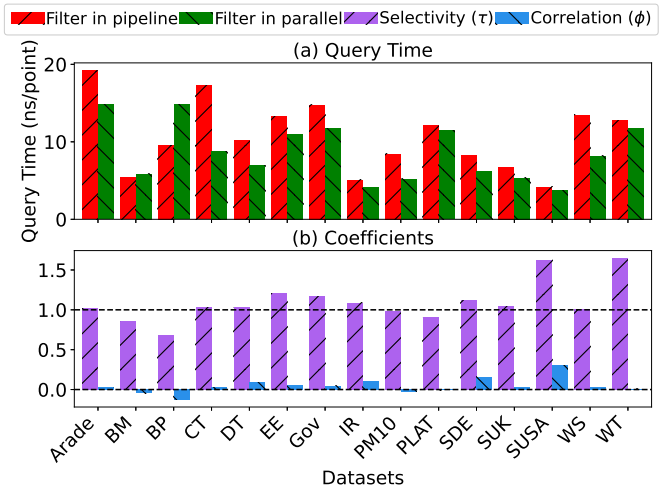


Fig. 12: The time of filter processing in pipeline and parallel on two Anti-correlated or Correlated columns

could remove more leading zeros, Sub-column has better compression ratio than method of SAP HANA as shown in Figure 9(a). Our algorithm outperforms compressions of SAP HANA, as demonstrated in Figure 9(b), due to (1) accelerated range filtering via compression-aware data skipping, and (2) enhanced aggregation performance by leveraging run-length in RLE-compressed sub-columns.

The novelty of Sub-column algorithm lies in its optimization of query time at the algorithmic level, rather than through hardware utilization. The algorithm Parquet-select in [24] leverages Bit Manipulation Instructions (BMI) to enable efficient selection pushdown on encoded columnar data, significantly reducing decoding overhead and improving query performance by up to an order of magnitude, even for complex predicates and nested structures. However, the approach is highly dependent on specific hardware support (e.g., Intel/AMD BMI instructions), limiting its portability to ARM-based architectures, and may incur additional overhead when processing mixed RLE and BPE data due to frequent context switching. We test compression ratio and query time of Parquet-select in Figure 9. The most queries of Parquet-Select is slower than Sub-column without BMI, and its has the worse compression ratio as depicted in Figure 9.

2) Filter Processing Strategy: We present the time of filter processing in pipeline and parallel on datasets and the selectivity and correlation of these datasets in Figure 12. Given the number of total rows S, the number of filtered rows A and B, and the number of match rows F, the selectivity τ [38] and Matthews correlation coefficient ϕ [29] are as follows, $\tau = \frac{\frac{F}{S}}{\frac{A}{S}\frac{B}{S}} = \frac{FS}{AB}$, and $\phi = \frac{\frac{F}{S} - \frac{A}{S}\frac{B}{S}}{\sqrt{\frac{S-A}{S}\frac{S-B}{S}}} = \frac{FS-AB}{S\sqrt{(S-A)(S-B)}}$. The first filters applied are very low selective, thus, the filtering time in pipeline of BM and BP is better than that in parallel. When the correlation coefficient is negative or low positive, two columns are anti-correlated or weak correlated, and the filter processing in pipeline is not an effective strategy. Hence, the filter processing in parallel is significantly better than that in pipeline on datasets Arade, CT, and WS, as depicted in Figure 12.

3) Aggregation on Columns with NULL: In Apache IoTDB, the positions of null values are recorded with a bitmap, while non-null values are stored in a compressed format. This allows aggregate functions such as MAX(X) and SUM(X) to be computed directly over the compressed non-null values, whereas COUNT(X) can be efficiently derived from the bitmap. We evaluated the execution time of the aggregation queries—MAX(X), SUM(X), and COUNT(X)— varying proportions of null values using Algorithm 3. The results, presented in Figure 11, show that the average time per point for MAX(X) and SUM(X) remains nearly constant regardless of the null rate, since Algorithm 3 processes non-null values directly. Since COUNT(X) could be efficiently derived from the bitmap, the time of COUNT(X) is very low in Figure 11.

4) Materialization: We conduct an experiment to compare impact on query time based on our sub-column algorithm by four materialization strategies and materialization strategies in Bitweaving [25] and C-Store [1] in this section.

5) Update Time: The following discussion about updates in the Sub-column compressed data is divided into 2 parts, append-only and general update. For 'append-only', blocks limit the impact for append-only data. And we add a new block for the new values appended, similar to appending data without compression. Thus, the time to handle append-only data is essentially its compression time, which is faster than general update operations. For general 'update', we could directly change each sub-column of updated value v if the bit width is not larger than the maximum bit width. Otherwise, we could add several new sub-columns for the portion of v that exceeds the maximum bit width.

We test the update time of Sub-column algorithm in Figure 14. We can observe that updating larger values is slower than updating smaller ones, and both are slower than the time of append-only data.

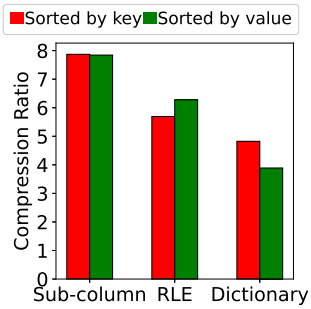


Fig. 13: Compression ratio on columns sorted by key or value

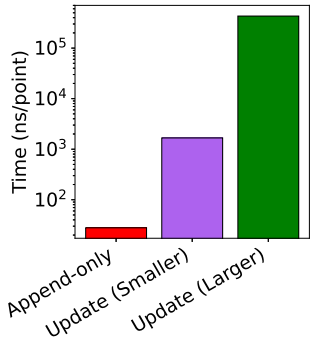


Fig. 14: Comparing update time on Sub-column compressed values

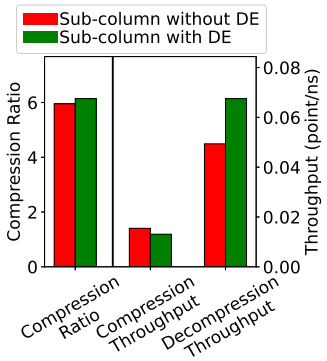


Fig. 15: Comparing compression ratio of Sub-column with and without DE

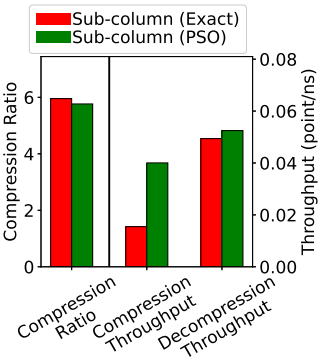


Fig. 16: Comparing compression ratio of exact Sub-column and Sub-column with PSO

E. Variants of Sub-column

1) Compression Ratio on Sorted Columns: The paper [35] proposes an entropy-based algorithm for detecting soft functional dependencies, allowing columns with similar entropy to share a combined dictionary and improve compression when sorted. This method was implemented in SAP Net Weaver Business Warehouse Accelerator (BWA), which uses dictionary and run-length encoding (RLE) on value blocks. Sorting increases consecutive repeated values, reducing both the number of RLE pairs and unique values per block, thereby enhancing compression effectiveness.

In Figure 13, we present the compression ratio of Dictionary, RLE and our Sub-column on columns sorted by key or value, since the most important functional dependencies are those where a candidate key (or the primary key) functionally determines other attributes [39]. Sorting improves the compression ratio of RLE as shown in Figure 13. However, data can only be ordered by the values of one column, values of the other column can appear in random order as said in [35]. As a result, the compression ratio of Dictionary and Sub-column deteriorates. Hence, sorting value columns is not always an effective option to improve compression ratio. Notably, even though sorting columns according to value yielded a noticeable improvement in Figure 13, RLE are still worse than the compression ratio achieved by our algorithm Sub-column.

2) Add Dictionary Encoding to Sub-column: The determination of the sub-column width includes but not limited to bit-packing and run-length encoding. If a sub-column has smaller cardinality, the sub-column could be compressed by Dictionary Encoding (DE). For the given series X and the bit width of sub-columns β, the j-th sub-column cost with bit-packing is,

$$DE(X, j, \beta) = \lceil \log(\omega_j + 1) \rceil n + 2(\beta + \omega_j),$$

where ω_j is cardinality of the j-th sub-column in the series, i.e., the number of unique values in the j-th sub-column. If the bit width of cardinality ω_j is larger than that of sub-columns, i.e., $\lceil \log(\omega_j + 1) \rceil \ b_\beta(j)$, the sub-column cost with dictionary encoding is greater than that with bit-packing encoding, and we could not compress the sub-column with dictionary encoding. Otherwise, when $DE(X, j, \beta)$ is smaller than $\min\{BPE(X, j, \beta), RLE(X, j, \beta)\}$, we could compress the j-th sub-column with dictionary encoding.

We evaluate the compression performance of Sub-columns with and without Dictionary Encoding (DE) in Figure 15. As shown in Figure 15, the incorporation of DE results in a modest improvement in compression ratio but leads to degraded compression throughput. The improvement in decompression throughput, however, is attributed to the fact that DE decompresses faster than RLE, as DE replaces RLE on several sub-columns.

3) Comparing with Sub-column (PSO): The Sub-column determination problem is essentially one of bit width β search. Our proposed algorithm is a cost-model-based algorithm to find optimal bit width of sub-columns. The unique challenges specific to this problem is to find optimal storage cost of sub-columns with less compression time. Of course, we propose an heuristic method to find an approximate optimal solution with less time. And we propose discussion and experiments about the algorithm Sub-column Determination Sub-column Determination with PSO to solve the problem.

We perform the evaluation about the compression performance of Sub-column (Exact) and Sub-column (PSO) in this section. Note that Sub-column (PSO) is a bit worse than the Sub-column algorithm in Figure 16, since Sub-column (PSO) could only find an approximate solution to the problem. Nonetheless, due to fewer iterations, Sub-column (PSO) has better throughput than the Sub-column algorithm as depicted in Figure 16.

4) Ablation Study: We conduct an ablation study on the compression and query performance when using only BPE or only RLE on sub-columns. Figure 17 shows the average compression ratio, compression throughput, decompression throughput and query time of Sub-column, Sub-column with only BPE (Sub-column-only-BPE), and Sub-column with only RLE (Sub-column-only-RLE) across all datasets presented in Table II. The compression ratio of Sub-column-only-BPE and Sub-column-only-RLE is worse than that of our Sub-

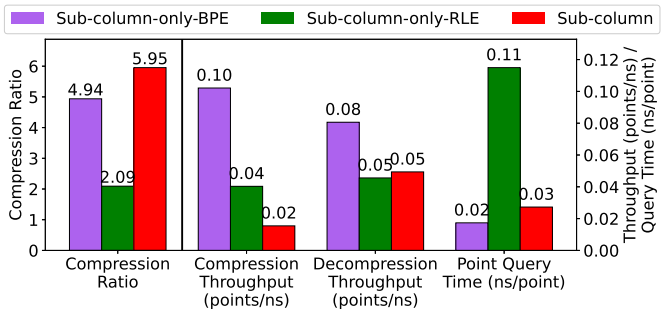


Fig. 17: Compression and query performance of ablation study on Sub-columns

column algorithm, since BPE and RLE could reduce storage cost of sub-columns. However, it takes more time for RLE to compute the run-length series than for BPE to compress and decompress data. RLE has a worse compression and decompression time than BPE, hence, Sub-column algorithm has worse compression and decompression throughput than Sub-column-only-BPE but better than Sub-column-only-RLE.

To evaluate the effectiveness of our algorithm for point queries and the impact of RLE on point query performance, we measured the point query time for Sub-column-only-BPE, Sub-column-only-RLE and our proposed approach Sub-column. Point query with RLE requires cumulative calculation of run-lengths to determine positions, whereas BPE allows direct jumping to the point position. As a result, the point query time of Sub-column is higher than that of Sub-column with BPE, but better than that of Sub-column with only RLE, as demonstrated in Figure 17(d).

VII. Related Work

There are numerous algorithms available for compressing series data [9], [43], [47]. Some of these algorithms are lossy, meaning they sacrifice data precision to achieve higher compression rates, as demonstrated in studies like [12], [11], [22], [46], [28]. Nonetheless, lossless compression algorithms aim to preserve the exact original data during compression, which is crucial in many scenarios such as industrial internet of things and financial records. Recent works have proposed various lossless algorithms, such as those presented in [43], [45], [44], offering improved efficiency and compression performance. It takes too much time for several compressions to compress data [1], [20]. And we focus more on lightweight lossless compressions such as XOR-based compression, delta-based compression and compression with sub-columns.

A. XOR-based Encoding

GORILLA [36] compresses floating-point values by computing the XOR of the current and previous decimals based on the IEEE 754 standard, followed by compressing these XOR values. Then, GORILLA uses control bits to indicate leading and trailing zeros are in the XOR values. CHIMP [26] enhances control bits of GORILLA by exploiting the distribution of leading and trailing zeros. Elf [23] further refines the compression process of CHIMP. Elf increases trailing zeros through floating-point precision before performing the XOR operation. However, the exponent part of float with the IEEE 754 standard introduces many ones in the leading bits, which significantly increases the bit width of bit-packing encoding.

B. Delta-based Encoding

When series data is smooth, delta-based encoding techniques, such as TS2DIFF [45] and SPRINTZ [8], subtract the previous data from the current data to reduce the absolute value. TS2DIFF [45] regards delta-of-delta of values as residuals. SPRINTZ [8] uses an adaptive filter Fast Integer REgression (FIRE) to predict series and stores the residuals between the predicted values and the real values. FIRE learns the autoregressive model with the coefficient a power of $\frac{1}{2}$, in order to calculate residuals with bit shift. Then, these algorithms use bit-packing to compress the residuals by removing leading zeros. However, these methods further fail to compress the space occupied by the residuals after bit-packing.

C. Query with Compressed Sub-columns

BUFF [27] decomposes values into integer and fractional parts, and splits both parts into sub-columns with 8 bits (a byte) width for query. However, BUFF does not consider further compression and query of sub-columns compressed by bit-packing (BPE) [25] or run-length encoding (RLE) [14] with different bit width. BPE employs a fixed bit width for each value and achieves compression by removing leading zeros of values. Bit-packing encoding could optimize the query of range filtering and aggregation by data skipping and filtered sub-columns. And RLE could improve the storage cost of the sub-columns by recording run-lengths and values. Some aggregation operators such as sum and count could become quicker by decompressing run-lengths and values directly. Hyrise [15] partitions data into chunks and compresses each column using dictionary and run-length encoding. Its query engine processes predicates directly on compressed blocks to minimize data access and late tuple reconstruction. C-Store [1] applies RLE and dictionary encoding to compress data. But Hyrise and C-Store neither split the column into sub-columns for compression nor perform sub-column-level query processing. BitWeaving [25], i.e., R2, organizes sub-column bits into contiguous groups to optimize CPU cache utilization during scans. However, it does not explore how different bit widths affect compression and query performance.

VIII. Conclusion

In this paper, we propose a novel sub-column compression framework, which enhances compression ratio and query performance by dividing a series into sub-columns. Our proposal automatically and efficiently determines the optimal bit width of sub-columns with incremental computation and pruning. Remarkably, the range filtering and aggregation queries could benefit from our sub-column compression. The proposed sub-column compression and its query optimization have been

implemented in Apache TsFile and Apache IoTDB. We conduct experiments comparing our method with other existing algorithms on real-world datasets. The results demonstrate that our proposal achieves a significantly higher compression ratio and less query time.

ACKNOWLEDGMENT

This work is supported in part by the National Natural Science Foundation of China (92267203, 62021002, 62232005), the National Key Research and Development Plan (2021YFB3300500), Beijing National Research Center for Information Science and Technology (BNR2025RC01011), and Beijing Key Laboratory of Industrial Big Data System and Application. Shaoxu Song (https://sxsong.github.io/) is the corresponding author.

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682. ACM, 2006.

[2] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization strategies in a column-oriented DBMS. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 466–475. IEEE Computer Society, 2007.

[3] Azim Afroozeh, Leonardo X. Kuffo, and Peter A. Boncz. ALP: adaptive lossless floating-point compression. *Proc. ACM Manag. Data*, 1(4):230:1–230:26, 2023.

[4] Apache IoTDB. https://iotdb.apache.org/, 2025.

[5] Apache Parquet Project. Encodings — parquet format, 2024. GitHub file, commit: 7a1b2c3 (replace with actual SHA).

[6] Apache TsFile. https://tsfile.apache.org/, 2025.

[7] PBI Bench. https://github.com/cwida/public_bi_benchmark, 2018.

[8] Davis W. Blalock, Samuel Madden, and John V. Guttag. Sprintz: Time series compression for the internet of things. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(3):93:1–93:23, 2018.

[9] Giacomo Chiarot and Claudio Silvestri. Time series compression survey. *ACM Comput. Surv.*, 55(10):198:1–198:32, 2023.

[10] Financial dataset used in INFORE project. https://zenodo.org/records/3886895#.Y4DdzHZByM_, 2023.

[11] Frank Eichinger, Pavel Efros, Stamatis Karnouskos, and Klemens Böhm. A time-series compression technique and its application to the smart grid. *VLDB J.*, 24(2):193–218, 2015.

[12] Hazem Elmeleegy, Ahmed K. Elmagarmid, Emmanuel Cecchet, Walid G. Aref, and Willy Zwaenepoel. Online piece-wise linear approximation of numerical streams with precision guarantees. *Proc. VLDB Endow.*, 2(1):145–156, 2009.

[13] EPM. https://doi.org/10.24432/C5NP5K, 2024.

[14] Solomon W. Golomb. Run-length encodings (corresp.). *IEEE Trans. Inf. Theory*, 12(3):399–401, 1966.

[15] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, 2010.

[16] Peter Hall. On the rate of convergence of normal extremes. *Journal of Applied Probability*, 16(2):433–439, 1979.

[17] Elliot Hallmark. Points of interest (poi) database, 2018.

[18] InfluxDB. Influxdb 2.0 sample data. https://github.com/influxdata/influxdb2-sample-data, 2023.

[19] Apache IoTDB. https://github.com/apache/iotdb/tree/research/encoding-subcolumn, 2025.

[20] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A. Chien, Jihong Ma, and Aaron J. Elmore. Good to the last bit: Data-driven encoding with codecdb. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 843–856. ACM, 2021.

[21] Kaggle. https://www.kaggle.com/datasets/mysarahmadbhat/wine-tasting, 2024.

[22] Iosif Lazaridis and Sharad Mehrotra. Capturing sensor-generated time series with quality guarantees. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 429–440. IEEE Computer Society, 2003.

[23] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. Elf: Erasing-based lossless floating-point compression. *Proc. VLDB Endow.*, 16(7):1763–1776, 2023.

[24] Yinan Li, Jianan Lu, and Badrish Chandramouli. Selection pushdown in column stores using bit manipulation instructions. *Proc. ACM Manag. Data*, 1(2):178:1–178:26, 2023.

[25] Yinan Li and Jignesh M. Patel. Bitweaving: fast scans for main memory data processing. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 289–300. ACM, 2013.

[26] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. Chimp: Efficient lossless floating point compression for time series databases. *Proc. VLDB Endow.*, 15(11):3058–3070, 2022.

[27] Chunwei Liu, Hao Jiang, John Paparrizos, and Aaron J. Elmore. Decomposed bounded floats for fast compression and queries. *Proc. VLDB Endow.*, 14(11):2586–2598, 2021.

[28] Jinyang Liu, Sheng Di, Kai Zhao, Xin Liang, Sian Jin, Zizhe Jian, Jiajun Huang, Shixun Wu, Zizhong Chen, and Franck Cappello. High-performance effective scientific error-bounded lossy compression with auto-tuned multi-component interpolation. *Proc. ACM Manag. Data*, 2(1):4:1–4:27, 2024.

[29] Brian W Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Structure*, 405(2):442–451, 1975.

[30] National Ecological Observatory Network (NEON). 2d wind speed and direction (dp1.00001.001). https://data.neonscience.org/data-products/DP1.00001.001/RELEASE-2022, 2022.

[31] National Ecological Observatory Network (NEON). Dust and particulate size distribution (dp1.00017.001). https://data.neonscience.org/data-products/DP1.00017.001/RELEASE-2022, 2022.

[32] National Ecological Observatory Network (NEON). Ir biological temperature (dp1.00005.001). https://data.neonscience.org/data-products/DP1.00005.001/RELEASE-2022, 2022.

[33] National Ecological Observatory Network (NEON). Relative humidity above water on-buoy (dp1.20271.001). https://data.neonscience.org/data-products/DP1. 20271.001/RELEASE-2022, 2022.

[34] Daily Temperature of Major Cities. https://www.kaggle.com/sudalairajkumar/daily-temperature-of-major-cities., 2023.

[35] Marcus Paradies, Christian Lemke, Hasso Plattner, Wolfgang Lehner, Kai-Uwe Sattler, Alexander Zeier, and Jens Krüger. How to juggle columns: an entropy-based approach for table compression. In Bipin C. Desai and Jorge Bernardino, editors, *Fourteenth International Database Engineering and Applications Symposium (IDEAS 2010), August 16-18, 2010, Montreal, Quebec, Canada*, ACM International Conference Proceeding Series, pages 205–215. ACM, 2010.

[36] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, 2015.

[37] SAP SE. Sap hana platform documentation, 2024.

[38] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, pages 23–34. ACM, 1979.

[39] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020.

[40] Sub-columns. https://github.com/thssdb/encoding-subcolumn, 2025.

[41] Apache TsFile. https://github.com/apache/tsfile/tree/research/encoding-subcolumn, 2025.

[42] Chen Wang, Jialin Qiao, Xiangdong Huang, Shaoxu Song, Haonan Hou, Tian Jiang, Lei Rui, Jianmin Wang, and Jiaguang Sun. Apache iotdb: A time series database for iot applications. *Proc. ACM Manag. Data*, 1(2):195:1–195:27, 2023.

[43] Tianrui Xia, Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jian-min Wang. Time series data encoding in apache iotdb: comparative analysis and recommendation. *VLDB J.*, 33(3):727–752, 2024.

[44] Jinzhao Xiao, Wendi He, Shaoxu Song, Xiangdong Huang, Chen Wang, and Jianmin Wang. REGER: reordering time series data for regression encoding. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*, pages 1242–1254. IEEE, 2024.

[45] Jinzhao Xiao, Yuxiang Huang, Changyu Hu, Shaoxu Song, Xiangdong Huang, and Jianmin Wang. Time series data encoding for efficient storage: A comparative analysis in apache iotdb. *Proc. VLDB Endow.*, 15(10):2148–2160, 2022.

[46] Zehai Yang and Shimin Chen. MOST: model-based compression with outlier storage for time series data. *Proc. ACM Manag. Data*, 1(4):250:1–250:29, 2023.

[47] Jiujing Zhang, Zhitao Shen, Shiyu Yang, Lingkai Meng, Chuan Xiao, Wei Jia, Yue Li, Qinhui Sun, Wenjie Zhang, and Xuemin Lin. High-ratio compression for machine-generated data. *Proc. ACM Manag. Data*, 1(4):245:1–245:27, 2023.

[48] Xin Zhao, Jialin Qiao, Xiangdong Huang, Chen Wang, Shaoxu Song, and Jianmin Wang. Apache tsfile: An iot-native time series file format. *Proc. VLDB Endow.*, 17(12):4064–4076, 2024.

[49] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 59. IEEE Computer Society, 2006.

APPENDIX

The Sub-column determination problem is essentially one of bit width β search. Our proposed algorithm is a cost-model-based algorithm to find optimal bit width of sub-columns. The unique challenges specific to this problem is to find optimal storage cost of sub-columns with less compression time. We replace the exhaustive search for the optimal sub-column bit width with a Particle-Swarm-Optimization (PSO) based procedure in Algorithm 4. In Lines 4-9, algorithm initial i-th particle's position p_i, its velocity v_i, the best i-th position $p_{best,i}$, the minimum cost $Cost_{\min,i}$, the global best position p_{best}, and the global minimum cost $C_{\min}$. Then, after $t_{\max}$ iterations, the algorithm updates the minimum cost for each particle and the global minimum cost. Particles update their velocities and positions using the standard PSO rule with inertia w and cognitive/social coefficients c_1, c_2 in Lines 12-13. The global minimum cost $C_{\min}$ is updated whenever any particle finds a lower cost in Lines 14-18. Finally, in the vicinity of the optimal solution β' found by the global search of PSO, another local refinement is performed in Lines 19-23.

Since Algorithm 4 updates s particles with $t_{\max}$ iterations, the total runtime is approximately $s * t_{\max} * n$ and the time complexity is $O(n)$. The parameters s and $t_{\max}$ could be tuned to trade off computational cost against solution quality, which influences compression ratio. This method requires far fewer time cost evaluations than a full exhaustive search for large M in Algorithm 1, when $s * t_{\max}$ is far smaller than M.

Algorithm 4: Sub-column Determination with PSO

Input: Series $X = (x_1, x_2, \ldots, x_n)$, PSO hyperparams: swarmSize s, maxIter $t_{\max}$, inertia w, cognitive coefficients c_1, social coefficients c_2, localRadius R

Output: Bit width β'

1 $M = \lceil \log(x_{\max} - x_{\min} + 1) \rceil$;
2 **for** $i \leftarrow 1$ **to** s **do**
3 $\quad p_i = U(1, M)$;
4 $\quad v_i = U(-(M-1)/2, (M-1)/2)$;
5 $\quad p_{best,i} = p_i$;
6 $\quad Cost_{\min,i} = Cost(X, round(p_i))$;
7 $p_{best} = \arg\min_i Cost_{\min,i}$;
8 $C_{\min} = \min Cost_{\min,i}$;
9 $v_{max} = M$;
10 **for** $t \leftarrow 1$ **to** $t_{\max}$ **do**
11 $\quad$ **for** $i \leftarrow 1$ **to** s **do**
12 $\quad\quad r_1 = U(0,1), r_2 = U(0,1)$, $v_i = w \cdot v_i + c_1 r_1 (p_{best,i} - p_i) + c_2 r_2 (p_{best} - p_i)$, clamp v_i to $[-v_{\max}, v_{\max}]$;
13 $\quad\quad p_i = p_i + v_i$, clamp p_i to $[1, M]$;
14 $\quad\quad \beta = round(p_i)$;
15 $\quad\quad$ **if** $C(X, \beta) < p_{best,i}$ **then**
16 $\quad\quad\quad Cost_{\min,i} = C(X, \beta)$, $p_{best,i} = p_i$;
17 $\quad\quad\quad$ **if** $C(X, \beta) < C_{\min}$ **then**
18 $\quad\quad\quad\quad C_{\min} = C(X, \beta)$, $p_{best} = p_i$;
19 $\beta' = round(p_{best})$;
20 **for** $b \leftarrow \beta' - R$ **to** $\beta' + R$ **do**
21 $\quad$ **if** $1 \le b \le M$ **then**
22 $\quad\quad$ **if** $Cost(X, b) < C_{\min}$ **then**
23 $\quad\quad\quad C_{\min} = Cost(X, b)$, $\beta' = b$;
24 **return** β';