A DEFINITION OF PARTIAL HOMOMORPHIC QUERY

Consider a compression mapping $\varphi : S_u \rightarrow S_c$, where S_u and S_c represent the uncompressed and compressed time series data. For a given query $Q_u = \langle op_1, op_2, \ldots, op_{j+1}, \ldots, op_n \mid op_i \in \Pi \rangle$, there may not exist on operator $op'_{j+1} \in \Theta$ corresponding to $op_{j+1} \in \Pi$ such that $\varphi^{-1}(op'_{j+1}(c)) = op_{j+1}(\varphi^{-1}(c))$, i.e., φ is not surjective. Let op_{j+1} be the first operator that fails to satisfy the homomorphic mapping. When query execution reaches op_{j+1} ($j \in \{0, \ldots, n-1\}$), decompression of the compressed time series becomes necessary. Such queries ensure correctness but fail to satisfy directness. We call it a **Partially Homomorphic Query**.

B IOT-BENCHMARK DATA GENERATION ALGORITHM

Algorithm 19 describes the data generation strategy. For parameter details, please refer to Table 1.

Table 1: Parameters Description of Data Generator

Notation	Data Features
μ_v	Mean of values
μ_d	Mean of deltas
θ_d	Variance of deltas
γ	Repetition Rate
η	Increase rate
l	Series length

Algorithm 2: Numerical Data Generator [87]

```
Input: $\mu_v$, $\mu_d$, $\theta_d$, $\gamma$, $\eta$, length $n$
Output: $TS$
1  $DS \leftarrow$ empty_list();
2  while $|DS| < n$ do
3      $isRepeat \leftarrow$ random_index($\gamma$);
4      if $isRepeat$ then
5          $repeat\_len \leftarrow$ random($8, T$);
6          $DS$.append(0, repeat_len);
7      else
8          $isPositive \leftarrow$ random_index($\eta$);
9          $delta \leftarrow 0$;
10         if $isPositive$ then
11             while $\delta \leq 0$ do
12                 $\delta \leftarrow$ random_gauss($\mu_d, \theta_d$);
13         else
14             while $\delta \geq 0$ do
15                 $\delta \leftarrow$ random_gauss($\mu_d, \theta_d$);
16         $DS$.append($\delta$);
17 $TS \leftarrow$ prefix_sum($DS$);
18 $TS$.zoom($\mu_v$);
19 return $TS$;
```

C PROOF OF PROPOSITION 3.6

The proof of Proposition 3.6 is as follows.

Proof. Let $Cost\left(I_u\left(c\right)\right)$ denote the cost of the traditional query process of a compressed database, i.e. decompressing first, then restore and query on uncompressed data, where

$$\begin{aligned}
Cost\left(I_u\left(c\right)\right) &= Cost\left(\left(Q_u \circ R_u \circ U\right)\left(c\right)\right) \\
&= Cost\left(Q_u\left(R_u\left(U\left(c\right)\right)\right)\right) \\
&= Cost\left(\left\{u : \begin{array}{l} u_1 \leftarrow op_1\left(u_0\right), \\ u_2 \leftarrow op_2\left(u_1\right), \\ \ldots, \\ u \leftarrow op_n\left(u_{n-1}\right) \end{array} \middle| \begin{array}{c} i \in \{1, \ldots, n\}, \\ op_i \in \Pi \end{array} \right\}\right) \\
&\quad + Cost\left(R_u\left(u\right)\right) + Cost\left(U\left(c\right)\right) \\
&= Cost\left(U\left(c\right)\right) + Cost\left(R_u\left(u\right)\right) + Cost\left(\{op_1, op_2, \ldots, op_{n-1}\}\left(u_1\right)\right).
\end{aligned}$$

Let $Cost\left(I_c\left(c\right)\right)$ denote the cost of the partial homomorphic query process of a compressed database, where

$$\begin{aligned}
Cost\left(I_c\left(c\right)\right) &= Cost\left(\left(I'_u \circ Q_c \circ R_c\right)\left(c\right)\right) \\
&= Cost\left(I'_u\left(Q_c\left(R_c\left(c\right)\right)\right)\right) \\
&= Cost\left(I'_u\left(c_j\right)\right) + Cost\left(Q_c\left(c_0\right)\right) + Cost\left(R_c\left(c\right)\right) \\
&= Cost\left(\left\{u : \begin{array}{l} u_j \leftarrow op_j\left(u_{j-1}\right), \\ u_{j+1} \leftarrow op_{j+1}\left(u_j\right), \\ \ldots, \\ u \leftarrow op_n\left(u_{n-1}\right) \end{array} \middle| \begin{array}{c} i \in \{j, \ldots, n\}, \\ op_i \in \Pi \end{array} \right\}\right) \\
&\quad + Cost\left(\left\{c_j : \begin{array}{l} c_1 \leftarrow op'_1\left(c_0\right), \\ c_2 \leftarrow op'_2\left(c_1\right), \\ \ldots, \\ c_j \leftarrow op'_j\left(c_{j-1}\right) \end{array} \middle| \begin{array}{c} i \in \{1, \ldots, j\}, \\ op'_i \in \Theta \end{array} \right\}\right) \\
&\quad + Cost\left(U\left(c_j\right)\right) + Cost\left(R_c\left(c\right)\right) \\
&= Cost\left(R_c\left(c\right)\right) + Cost\left(\{op'_1, op'_2, \ldots, op'_j\}\left(c_0\right)\right) \\
&\quad + Cost\left(U\left(c_j\right)\right) + Cost\left(\{op_{j+1}, op_{j+2}, \ldots, op_{n-1}\}\left(u_{j-1}\right)\right).
\end{aligned}$$

with $\varphi\left(u_i\right) = c_i$, $j \in \{1 \ldots n-1\}$. Then we have

$$\begin{aligned}
&Cost\left(I_u\left(c\right)\right) - Cost\left(I_c\left(c\right)\right) \\
&= Cost\left(U\left(c\right)\right) + Cost\left(R_u\left(u\right)\right) + Cost\left(\{op_1, op_2, \ldots, op_{n-1}\}\left(u_0\right)\right) \\
&\quad - \Bigg[Cost\left(R_c\left(c\right)\right) + Cost\left(\{op'_1, op'_2, \ldots, op'_j\}\left(c_0\right)\right) \\
&\qquad + Cost\left(U\left(c_j\right)\right) + Cost\left(\{op_{j+1}, op_{j+2}, \ldots, op_{n-1}\}\left(u_{j-1}\right)\right)\Bigg] \\
&= \left(Cost\left(U\left(c\right)\right) - Cost\left(U\left(c_j\right)\right)\right) + \left(Cost\left(R_u\left(u\right)\right) - Cost\left(R_c\left(c\right)\right)\right) \\
&\quad + \Bigg(Cost\left(\{op_1, op_2, \ldots, op_{n-1}\}\left(u_0\right)\right) - Cost\left(\{op'_1, op'_2, \ldots, op'_j\}\left(c_0\right)\right) \\
&\qquad - Cost\left(\{op_{j+1}, op_{j+2}, \ldots, op_{n-1}\}\left(u_{j-1}\right)\right)\Bigg) \\
&= \left(Cost\left(U\left(c\right)\right) - Cost\left(U\left(c_j\right)\right)\right) + \left(Cost\left(R_u\left(u\right)\right) - Cost\left(R_c\left(c\right)\right)\right) \\
&\quad + \left(Cost\left(\{op_1, op_2, \ldots, op_j\}\left(u_0\right)\right) - Cost\left(\{op'_1, op'_2, \ldots, op'_j\}\left(c_0\right)\right)\right)
\end{aligned}$$

Note that when $j = n-1$, query Q is a fully homomorphic query.

By Lemma 3.5, $Size\left(c\right) \geq Size\left(c_j\right)$, thus, $Cost\left(U\left(c\right)\right) \geq Cost\left(U\left(c_j\right)\right)$. And by Definition 3.4, we have

$$Cost\left(\{op_1, op_2, \ldots, op_j\}\left(u_0\right)\right) \geq Cost\left(\{op'_1, op'_2, \ldots, op'_j\}\left(c_0\right)\right).$$

And by Definition 3.3, we have

$$Cost\left(R_u\left(u\right)\right) \geq Cost\left(R_c\left(c\right)\right)$$

Thus, we have $Cost\left(I_u\left(c\right)\right) - Cost\left(I_c\left(c\right)\right) \geq 0$, i.e.,

$$Cost\left(I_c\left(c\right)\right) \geq Cost\left(I_c\left(c\right)\right).$$

□

D IMPLEMENTATION

We implement CompressIoTDB in Java in Apache IoTDB, an open-source time series database.

CompColumn. Compressed and uncompressed data have different access patterns, requiring operators to adapt to the compression scheme. This can lead to poor code extensibility. To solve this, we encapsulate CompColumn into an object class with access and evaluation interfaces consistent with uncompressed data, allowing operators to focus on computation. New compression scheme can be added easily by inheriting the object class and implementing all the interfaces. Key data structures, such as *compression offset index* and *HintIndex*, are implemented as member variables in CompColumn. To reduce memory usage, basic operations such as compressed time series data slicing and reversing are implemented as reference to the original data, avoiding extra memory allocation.

Homomorphic operators. Homomorphic operators are integrated into Apache IoTDB's operator classes. Through `if` statements, the operators determine whether the data is compressed and invoke appropriate functions. For operators that do not support direct computation on compressed data, CompColumn's unified interface design allows them to process it as uncompressed data, eliminating the need for format awareness or additional development.

E QUERY DETAILS

On each datasets, we conduct two queries. Each query targets specific metrics within its respective dataset, offering diverse insights into time-series behavior, statistical characteristics, or cumulative values across different IoT domains.

The first two queries, Q1 and Q2, are conducted on the dataset Weather Forecast. Q1 calculates the average temperature when the minimum temperature is above 20 degrees Celsius, and the result is converted to Fahrenheit. Q2 calculates the average wind level and wind direction, grouped by 1-minute intervals within the specified time range. The results represent wind characteristics over short time intervals. Q3 and Q4 are conducted on the AMPds dataset. Q3 providing statistical insights into the variation and dispersion of current values within the dataset, while Q4 calculates the total apparent energy (S) in kilovolt-ampere hours (kVAh) and the total real power (P) in kilowatt-hours (kWh), providing cumulative energy usage metrics. Q5 and Q6 are conducted on the Smart Grid dataset. Q5 indicates daily average plug energy usage. And Q6 calculates cumulative daily household energy usage. Q7 and Q8 are conducted on the Linear Road dataset. Q7 offers insights into the variability of direction data over daily intervals for a specific lane, while Q8 calculates the average speed, grouped by 1-day intervals within the specified time range, representing the average daily speed of vehicles. The last two queries, Q9 and Q10, are conducted on the Computer Monitor dataset. Q9 computes the variance of cpu usage, grouped by 1-day intervals within the specified time range. It highlights daily variations in CPU utilization. And Q10 reveals daily fluctuations in task priority levels.

F DISCUSSION ON SYSTEM COMPABILITY

CompressIoTDB is a versatile framework, adaptable to other mainstream TSDBs [40, 66, 80, 81]. These databases typically use columnar storage and manage in-memory data via specific structures (e.g., *Chunk* in TimescaleDB or *DataPoint* in KairosDB). Integrating CompressIoTDB to these systems involves two key steps: 1) constructing a compressed intermediate representation (i.e., CompColumn) compatible with the native data format; and 2) implementing query operators to enable direct computation on compressed data with CompColumn.

Example. In KairosDB, CompColumn can be implemented by inheriting the *DataPoint* interface and adding compression management interfaces. Instead of processing individual data points iteratively, operators can directly compute on compressed series data in *CompColumn* without decompression. Since KairosDB relies on Cassandra for compression, it supports only general-purpose compression such as LZ4 and Snappy. Preliminary experiments with the Computer Monitor dataset show that integrating RLE as a light-weight homomorphic compression layer yields a 68.5% storage space reduction and a 55.2% memory usage reduction during queries.

However, integrating CompressIoTDB with other systems requires system-specific development. Since this paper focus on designing the homomorphic time series querying framework, we consider integration with other systems as a direction for future work.

G PHASES OF QUERY EXECUTION

In Apache IoTDB, queried data is pre-loaded into a chunk cache to optimize I/O performance. Subsequent queries retrieve data directly from this cache. The *chunk reader construction* phase refers to the process of loading data from the chunk cache. In IoTDB, this includes general-purpose decompression for the entire chunk. In contrast, CompressIoTDB loads only the compressed chunk without decompression at this stage. The *series scan* phase involves scanning the data after it has been loaded from the chunk cache. For IoTDB, this includes light-weight decompression and data filtering. In CompressIoTDB, general-purpose decompression is deferred to this stage, and light-weight decompression is avoided. *Operator execution* time is computed by subtracting the time taken for chunk reader construction and series scan from the total execution time. Short tasks, such as time series metadata loading, are negligible and thus omitted.